Optimization of Seismic Imaging Code, DSFDM/FFWI (Direct Solver Finite Difference Method / Frequency Full Waveform Inversion), on a Shared Next Generation Node, MESCA II

Djibril MBOUP (djibril.mboup@aims-senegal.org)

African Institute for Mathematical Sciences (AIMS)
Senegal

Supervised by:
Benjamin PAJOT, Bull ATOS, France
Abdou Khadir GAYE, ATOS GDC, Senegal

28 January 2018
Submitted in Partial Fulfillment of a Masters II at AIMS

AIMS | African Institute for Mathematical Sciences SENEGAL

Abstract

A short, abstracted description of your essay goes here. It should be about 100 words long. But write it last.

An abstract is not a summary of your essay: it's an abstraction of that. It tells the readers why they should be interested in your essay but summarises all they need to know if they read no further.

The writing style used in an abstract is like the style used in the rest of your essay: concise, clear and direct. In the rest of the essay, however, you will introduce and use technical terms. In the abstract you should avoid them in order to make the result comprehensible to all.

You may like to repeat the abstract in your mother tongue.

Declaration

I, the undersigned, hereby declare that the work contained in this essay is my original work, and that any work done by others or by myself previously has been acknowledged and referenced accordingly.

Firstname Middlename Lastname, 15 May 2014

Contents

1. Introduction

The last decade has witnessed real progress in the geophysical research area. This fact is due to the development of new approach in seismic modelling such as *Full Waveform Inversion* (FWI), and the opportunity that offers the *High Performance Computing* (HPC). The rapid increasing of the HPC technologies allows to deal with large test cases with significant performance. In this insight, a group of researcher working in Geoazur laboratory and *Observatory of Côte d'Azur* (OCA) has developed advanced seismic imaging methods. To share the benefit of their work in the scientific community, they make in open-source distribution many of their packages. At the same time, These researchers are involved the SEISCOPE[1] consortium sponsored by 9 companies Gas and Oil from which their work is promoted.

In this perspective, the SEISCOPE team has developed an efficient code for full waveform inversion, called DSFDM/FFWI (*Direct Solver Finite Difference Method / Frequency FWI*). The DSFDM/F-FWI code was applied successfully in real seismic dataset collected in ocean-bottom cable data from the Valhall oil field (North Sea) in the visco-acoustic *vertical transverse isotropic* (VTI) approximation [OMB+15, ABB+16]. For its running, this code uses different packages especially MUMPS (*MULti-frontal Massively Parallel sparse direct Solver*) to solve the wave propagation equation with direct-solver method which is the most expensive part of the inversion process.

The modelling in a frequency domain is a promising method because it naturally takes into account the attenuation, a physical characteristic that can greatly impact the seismic imaging quality in certain cases, but also because of its speed of calculation for simulations with multiple right-hand side. The backwards of this method, which has prevented its expansion rapidly, is its memory consumption which limits it to medium-sized use cases.

Bull, a company of Atos group, has developed its second generation of server x86 with large shared memory, called MESCA-II (*Multiple Environments on a Scalable Csi-based Architecture*). Technically, a MESCA-II node can have 8 sockets of Intel® Xeon® processor family with 3 Terabytes of memory per sockets. MESCA-II can theoretically offer 24 TB of shared memory . Thus, MESCA-II node give a great opportunity to make performance studies on DSFDM/FFWI including scalability studies and code optimization.

In this report, we will give a brief review of seismic imaging and the context which is used. We study, in chapter 3, how *High Performance Computing* (HPC) is and its use of the large simulation problems. Finally, we study the performance of DSFDM/FFWI code in MESCA-II node.

[1] http://seiscope2.osug.fr

2. Geophysical Seismic Imaging

2.1 Introduction

The Earth is characterised by seismic activities. Human has been always animated by the curiosity to understand these activities. During years, human used different techniques to explore the basement depths. Recently, this exploration is more motivated by multiples reasons that can be economics, social, environment, scientific etc. For instance, the study of the nature the soil is useful for civil engineers, landslide imaging is very useful for estimating the risk of gravity, the detection of oil and mineral resources present a major economic challenge, the monitoring of radioactive waste landfill areas or CO2 injection represents an new environmental challenge.

With the advents of Industrial revolution accompanied by the higher need to find new oilfield by hydrocarbon industry, the *seismic imaging* technique has successfully used from the mid of 19th. By definition, seismic imaging is a tool that bounces sound waves off underground rock structures to reveal the possible tectonic structures [Red12]. In seismic modelling, one need data coming from seismic wave propagation to create good quality images of the subsurface (imaged by acoustic waves). The acoustic waves are generated under the Earth by sources that can be man-made devices or by earthquakes. Receivers or seismometers acquire information that can provide details of the velocity and the geometric structure of the Earth [Red12].

During years scientists invent several techniques of seismic imaging which include *Electrical Resistivity Tomography*, *Ground penetrating radar*, *Induced polarization*, *Seismic Tomography* and *Reflection seismology* [Red12].

2.1.1 Electrical Resistivity Tomography (ERT). The ERT technique is invented by Schlumberger brothers and developed by the work of Andrey Nikolayevich Tikhonov . This method is an electrical testing method where electrical current is induced in the ground between one pair of electrodes and the voltage is measured between another pair. These measurements are used to estimate lateral and vertical variations of resistivity values of the earth. ERT can be used to map geologic variations (soil lithology, presence of ground water, fracture zones, variations in soil saturation, areas of increased salinity or, in some cases, ground water contamination), bedrock depths and geometry, mapping cavities such as caves, karst and/or evaporate dissolution sinkholes. Like other seismic technique, ERT has the capacity to yield either 1D (sounding), 2D (profile) or 3D (volume) imaging.

2.1.2 Ground Penetrating Radar (GPR). GPR systems work by sending a tiny pulse of energy into a material via an antenna. An integrated computer records the strength and time required for the return of any reflected signals. Subsurface variations will create reflections that are picked up by the system and stored on digital media. This technique has many applications: civil engineering, resources explorations.

2.1.3 Induced Polarization (IP). Similar to electrical resistivity tomography, this technique measure the electrical chargeability of subsurface materials. IP provide additional information about the spatial variation in lithology and grain-surface chemistry. It can be made in time-domain and in frequency-domain. IP method is widely used in mineral exploration and mining industry and it has other applications in hydrogeophysical surveys, environmental investigations and geotechnical engineering projects.

2.1.4 Reflection Seismology (RF). This is the most wide used technique when we talk about hydrocarbon exploration, or mineral exploration. RF is a form of echo sounding, detecting echoes from seismic interfaces. It can yield results that are the closest of any geophysical technique to a conven-

tional geological section. The waves propagation under the Earth require to know some features of the wave: transmission, absorption, and attenuation in the earth materials and its reflection, refraction, and diffraction characteristics at discontinuities.

In the context of onshore, geophysicists use geophones as receivers to collect the speeds of particles on the ground. Accelerometer sensors can be used to record these data. These sensors are directional and measure the speed or acceleration in a spatial direction. Sometimes, devices with multi-components are used to measure wavefields in the horizontal directions (parallel to the Earth's surface) and vertical direction, thus identifying the different types of waves whose depend on the directions [BRO09]. The figure 2.1 shows the scenario of data acquisition with geophones.

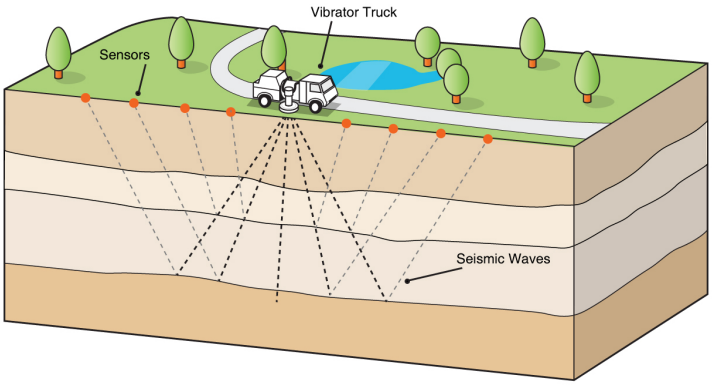


Figure 2.1: Seismic imaging

In the off-shoring scenario, the hydrophones, placed on the surface of the sea, are often used to collect the seismic wavefields information. In the figure 2.2, a ship generates the seismic source by an air gun and drag behind, under the water surface, a raw of hydrophones measuring the pressure field which indicates the waves propagated in the structure of the subsoil. Others sensors named *Ocean Bottom Seismometer* (OBS) are designed to record the earth motion under oceans and lakes from man-made sources. These devices can stay several days under the water. Each OBS receives different acoustic waves for each generation of sound arriving at different times. There exists another device called Ocean Bottom Cable (OBC).The data acquisition is performed by multi-component sensors bound on the cables [BRO09].

2.2 Seismic model

These techniques need numerical methods to solve the seismic imaging problem. In this study, we just focus on Frequency-domain seismic modelling based on the sparse direct solver (DSFDM) and Full Wave Inversion (FWI) method used by Seiscope Consortium to develop the DSFDM/FWI code.

The DSFDM is a finite-difference frequency-domain method, which has aimed to solve linear systems resulting from the discretization of the time-harmonic wave equation with sparse direct solvers. This method with 27 stencils has developed in a visco-accoustic *vertical transverse isotropic* (VTI) [OVA+07, OMB+15, ABB+16, BESJ10, BPC+14]. It is governed by the linear systems resulting by 3D visco-

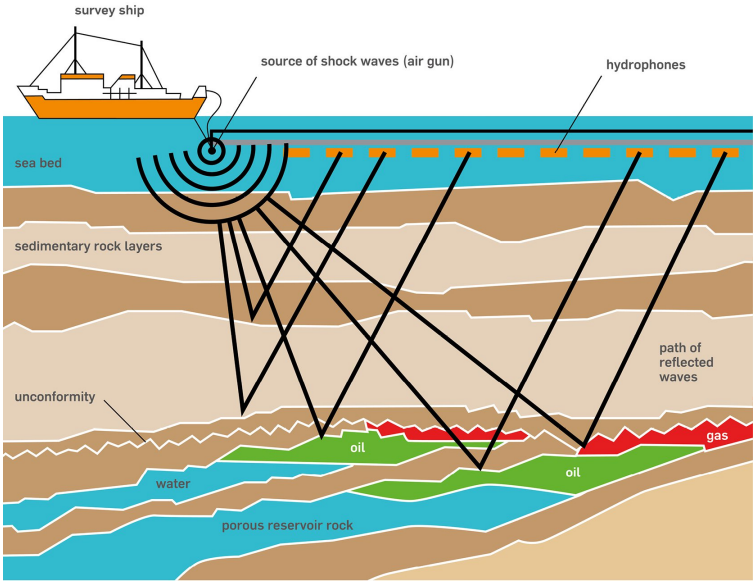


Figure 2.2: Seismic imaging

acoustic VTI wave equation:

$$Ap_h = b \tag{2.2.1}$$

$$p_v = Bp_h + s^{'}, \tag{2.2.2}$$

$$p = (2p_h + p_v)/3 \tag{2.2.3}$$

where the matrices A and B result from the discretization of operators.

$$A = \omega^2 \left[\frac{\omega^2}{\kappa_0} + (1+2\epsilon)(\mathcal{X}+\mathcal{Y}) + \sqrt{1+2\delta}\mathcal{Z}\frac{1}{\sqrt{1+2\delta}} \right] \tag{2.2.4}$$

$$B = \frac{1}{\sqrt{1+2\delta}} + \frac{2(\epsilon-\delta)\kappa_0}{\omega^2\sqrt{1+2\delta}}(\mathcal{X}+\mathcal{Y}) \tag{2.2.5}$$

Similarly, the source terms are given by:

$$b = \frac{\omega^4 s_h}{\kappa_0} s - \omega^2\sqrt{1+2\delta}\mathcal{Z}\left(s_v - \frac{1}{\sqrt{1+2\delta}}s_h\right)s, \tag{2.2.6}$$

$$s^{'} = \left(s_v - \frac{1}{\sqrt{1+2\delta}}s_h\right)s \tag{2.2.7}$$

The wavefields $p_h = \sigma_{xx} = \sigma_{yy}$ and $p_v = \sigma_{zz}$ are the so-called horizontal and vertical pressure wavefields, respectively [PC11]. The angular frequency is denoted by ω, $\kappa_0 = \rho V_0^2$ where ρ is the density and V_0 is the vertical wavespeed, δ and ϵ are the Thomsen's parameters. The seismic source vector is compactly denoted by s. For an explosion and vertical force, the expression of the source vector is given by

$$s_{explosion} = s(\omega)\hat{\delta}(x_s - x) \tag{2.2.8}$$

$$s_{vertical_force} = s(\omega)d_z\hat{\delta}(x_s - x) \tag{2.2.9}$$

where $s(\omega)$ denotes the Fourier coefficient of the temporal source wavelet for the modeled frequency, d_z the derivative with respect to z,$\hat{\delta}$ the delta function, x_s the coordinates of the source position.

The s_h and the $s_v$terms are given by the following formulas.

$$s_h = (2(2+\epsilon) + \sqrt{1+2\delta})/D, \tag{2.2.10}$$

$$s_v = (1 + 2\sqrt{1+2\delta})/D, \tag{2.2.11}$$

$$D = 4\sqrt{1+2\epsilon} + 4\sqrt{1+2\delta} + 1 \tag{2.2.12}$$

The expression of D was corrected relative to the one provided in Operto et al. (2014).
The second-order differential operators $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are given by

$$\mathcal{X} = \partial_{\tilde{x}} b \partial_{\tilde{x}}, \mathcal{Y} = \partial_{\tilde{y}} b \partial_{\tilde{y}}, \mathcal{Z} = \partial_{\tilde{z}} b \partial_{\tilde{z}},$$

where $b = 1/\rho$ is the buoyancy and the complex-valued coordinate system $(\tilde{x}, \tilde{y}, \tilde{z})$ is used to implement perfectly-matched layers absorbing boundary conditions [OVA+07]. If we assume that δ is slowly varying and hence can be considered as locally homogeneous, the operator A can be simplified as

$$A_a = \omega^2 \left[\frac{\omega^2}{\kappa_0} + (1+2\epsilon)(\mathcal{X}+\mathcal{Y}) + \mathcal{Z}\right] 2\mathcal{Z}\kappa_0(\epsilon-\delta)(\mathcal{X}+\mathcal{Y}) \tag{2.2.13}$$

In this case, the wave operator within the bracket describes wave propagation in an elliptic medium, while the second term accounts for anellipticity.

In *elliptic* media $(\delta = \epsilon \neq 0)$, the source become

$$b = \frac{\omega^4 s_h}{\kappa_0} s \tag{2.2.14}$$

$$s^{'} = 0 \tag{2.2.15}$$

The authors mention this point because the second term of the source b, equation 2.2.6, involves a second-order vertical derivative which can be tricky to implement if the source does not match the grid point. Therefore, it is worth assuming that the medium is elliptic at the source positions if this is a reasonable assumption.

In isotropic media $(\delta = \epsilon = 0)$, one can easily check that Equations 3.3.14-2.2.3 reduces to

$$\left[\frac{\omega^2}{\kappa}\mathcal{X} + \mathcal{Y} + \mathcal{Z}\right] p = \frac{\omega^2}{\kappa} s \tag{2.2.16}$$

where the operator $\left[\frac{\omega^2}{\kappa}\mathcal{X} + \mathcal{Y} + \mathcal{Z}\right]$ represents the operator A_{iso}

Similarly, the operator A can be rewritten considering the NMO velocity $(V_{NMO} = V_0\sqrt{1+2\delta})$ in the diagonal term. This introduces the $\eta = \frac{\epsilon-\delta}{1+2\delta}$ parameter in the coefficients of the matrix.

$$A_{NMO} = \omega^2 \left[\frac{\omega^2}{\kappa_{NMO}} + (1+2\eta)(\mathcal{X}+\mathcal{Y}) + \sqrt{1+2\delta}\mathcal{Z}\frac{1}{\sqrt{1+2\delta}}\right] + 2\sqrt{1+2\delta}\mathcal{Z}\frac{\kappa_{NMO}}{\sqrt{1+2\delta}}(\mathcal{X}+\mathcal{Y}) \tag{2.2.17}$$

Algorithm.
The system of three equations 3.3.14-2.2.3 is solved with the following sequence.

- Solve the linear system, equation 3.3.14, for p_h with a sparse direct solver.
- Compute explicitly p_v from p_h, equation 2.2.2.
- Compute explicitly p from p_h and p_v, equation 2.2.3.

The computational cost of the last two steps is negligible relative to that of the first step.

Note that wave-equation operators 2.2.4, 2.2.13, 2.2.16 and 2.2.17 are implemented in DSFDM code. (See Documentation)

2.3 Full Wave Inversion (FWI)

The Full Wave Inversion is a non linear-data fitting method that enable to get detailed estimates of subsurface properties from seismic data, which can be the result of either passive or active seismic experiments. Basically, initialising a guess of the subsurface parameters (a model), the data are predicted from the solution of the wave propagation equation. And then, the model is always updated to minimize the misfit function between the observed and predicted data. The figure 2.3 illustrates the FWI procedure. (3.3.14)

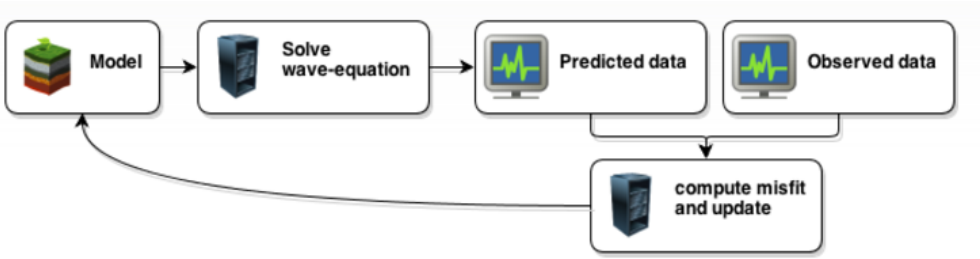


Figure 2.3: Full Wave Inversion (FWI) procedure. from []

Mathematically, the FFWI can be formulated as an optimization problem. In [VO09], Virieux and Operto give the main theoretical aspects of FWI based on a least-squares local optimization approach. In a data-domain frequency-domain, FWI is an iterative reduction of the misfit function C defined as the least-squares norm of the difference between recorded and modelled monochromatic pressure data, d_{obs} and $d(m)$, respectively.

$$\min_m C(m) = \min_m \sum_\omega \|d_{obs} - d(m)\|$$

The subsurface model updated at iteration $k+1$ is given by:

$$m_{k+1} = m_k - \gamma_k \mathbf{H}_k \nabla_m C_k$$

where $\mathbf{H}_k \nabla_m C_k$,the direction of gradient is given by the product of the gradient of C by an approximation of the inverse Hessian, H. The step length γ_k defines quantity of descent. The descent direction can be preconditioned by a diagonal approximation of the Hessian, in our case, the diagonal elements of

the so-called pseudo-Hessian matrix [SYM+01], the aim of which is to balance the gradient amplitudes with respect to depth by removing geometrical spreading effects.

The expression of the gradient preconditioner is given by:

$$H = 1/[P + \epsilon max(P)], \tag{2.3.1}$$

where $P = diag\Re\left\{(\frac{\partial A}{\partial m}p_h)^\dagger, (\frac{\partial A}{\partial m}p_h)\right\}$, $\frac{\partial A}{\partial m}p_h$ are the so-called virtual sources [PSH98] and the damping factor ϵ should be chosen with care to balance properly in depth the gradient without generating instabilities. For more details about full-waveform inversion see [VO09].

The FFWI code is implemented with SEISCOPE optimization toolbox which contains different optimization algorithms (steepest-descent, conjugate gradient, l-BFGS, truncated Newton) including the line search (see documentation).

2.4 Application: Valhall

The Seismic imaging was applied in Vallhal dataset with Full Waveform Inversion method in the context of SEISCOPE project [OBC+14, OMB+15, ABB+16]. The Valhall oil field is an offshore field located in

Figure 2.4: Valhall offshore

North Sea in 70 m of water (see figure 2.4). In this oilfield, the presence of gas cloud is observed in the overburden. The reservoir is approximatively located in 2.5 km depth [BHK+10]. The data acquisition was making possible by wide aperture/azimuth acquisition covering a surface of 145 km^2. A layer of 12 cables contains 2302 hydrophones, which record 49,954 explosive sources located 5 m below the surface of the water (Figure 3.3.a). A distance of 300 m separate two cables except two outer cables at 600 m. The figure 3.3.a shows the gas cloud intersecting the cables (area covered by the 50,000 explosive sources) and the receiver position, whose records are shown in panel. In figure 3.3.b, the solid black arrow indicate precritical reflection from the reservoir, the dashed black arrow points to the critical and postcritical reflection, and the white arrow points to the reflection from the top of the gas [OMB+15, ABB+16].

In the study of [OMB+15], a case study of valhall data set shows approximately the presence of gas at 2.5km in depth []. With initial condition V_0, vertical-velocity, ϵ and δ, Thomsen's parameters, and the density $\rho = -0.0261V_0^2 + 0.373V_0 + 1.458$ are used to apply frequency-domain FWI on the OBC

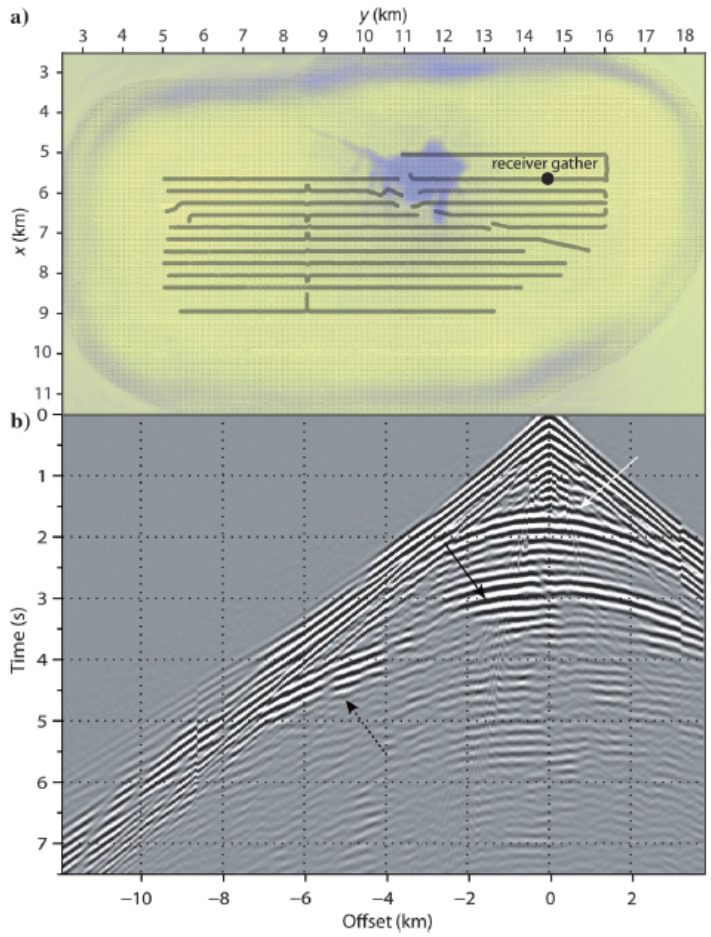
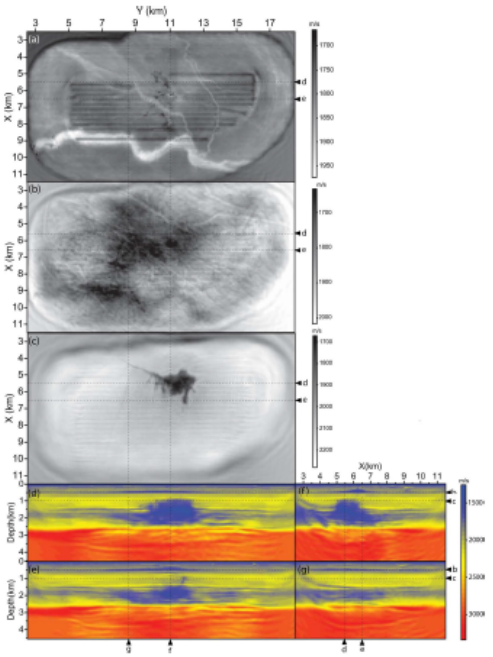


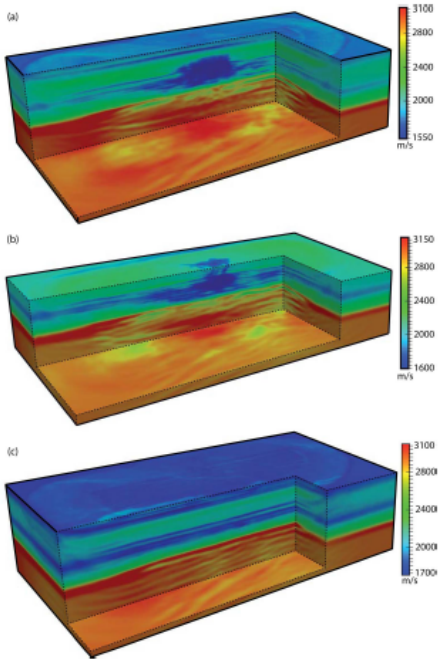
Figure 2.5: North Sea case study. (a) Acquisition layout and Valhall model [ABB+16]

data set [OMB+15, ABB+16]. With discrete frequencies chosen in the 3.5-10 Hz frequency band, the DSFDM/FWI code was performed in computer nodes that are equipped with two 2.5 GHz Intel Xeon IvyBridge E5-2670v2 processors with 10 cores per processor. the shared memory per node is 64 GB. The connecting network is InfiniBand fourth data rate (FDR) at 56 Gb/s. The operations are performed in single-precision complex arithmetic, for which the peak performance of the machine is 10 Gflops/s/core (which corresponds to a double precision peak of 20 Gflops/s/core) [OMB+15]. Figures 2.6a 2.6b show the 2D and 3D visualisations in different levels of the reservoir of the gas cloud. According to the authors, this study reveals memory consumption in the step of LU factorisation. They showed that the complexity can reach $O(n^6)$ [OMB+15].

This application shows how high performance computing (HPC) is important in the geophysical seismic modelling. Because of the complexity of the problem, one can't think to compute seismic imaging in classical machines. The HPC technology has developed in parallel to improve the performance of problems with high complexity.



(a) Slices of the 10Hz FWI model, that can be compared with those of the initial model[OMB+15].



(b) 3-D perspectives of the final FWI model.

Figure 2.6: Seismic imaging of Valhall [OMB+15]

3. High Performance Computing (HPC)

3.1 Introduction

Supercomputing has developed early in the 1970s with CDC 6600, ILLIAC-IV and the generation of CRAY. The high performance computing (HPC) was born in context of scientific computing. In fact, It was a challenge to be fast and more precise when were talking about scientific computing such as simulation purposes like weather forecasting, numerical mechanics for car crash tests, financial market prediction or other various complex phenomena modelings. The HCP's development was also accompanied with new paradigm in the computer design in hardware and software levels. It takes more advantages in the concepts of parallel system and parallel program, shared and distributed memory. Thus, the high performance computers as machines with a good balance among the following major elements [RubXX]

- Multistaged (pipelined) functional units.
- Multiple central processing units (CPUs) (parallel machines).
- Multiple cores.
- Fast central registers.
- Very large, fast memories.
- Very fast communication among functional units.
- Vector, video, or array processors.
- Software that integrates the above effectively.

Today, with Moore's Laws, the performance of computer is continuously increasing. In 2017, the Top500[1], website ranking the list of the top 500 supercomputers, classifies the Sunway TaihuLight (a system developed by *China's National Research Center of Parallel Computer Engineering & Technology* (NRCPC)) to be the most power computer in the world. The Sunway TaihuLight has a note of 93 Petaflops in Linpack Performance, where a Petaflops is 10^{15} floating-point operations per second. Recently, the Atos group has launched a new generation of quantum learning machines that can reach 24 TB of memory and 16 CPU, with power of from 30 to 40 Qubits (see 3.1). These computers allows researchers, engineers and students to develop and experiment with quantum software.

3.2 HPC architecture

Several effort has been doing in single processor performance by the ever-increasing density of transistors-the electronic switches-on integrated. The small dimension of transistors in an integrated circuit has increased the speed of this circuit. Therefore, this fact implies the increase of the heat of these transistors due to the power consumption. However, the air-cooled integrated circuits are reaching the limits of their ability to dissipate heat, in the first decade of the twenty-first century [HW10]. Therefore, it is

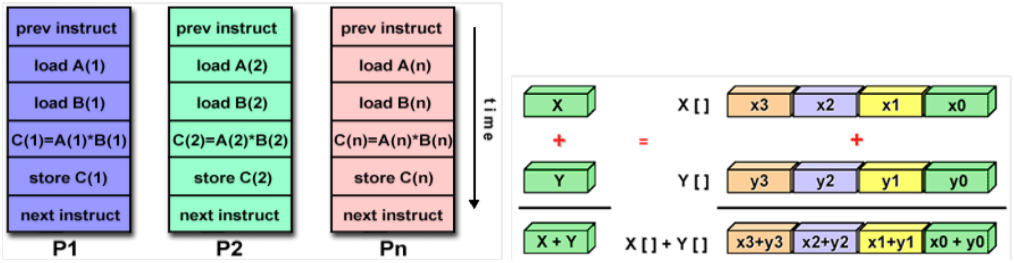
[1] https://www.top500.org

Figure 3.1: ATOS quantum learning machine

not sure to continue to increase the speed of integrated circuits. The chip's industries have another option instead to spend much money for building ever-faster, more complex, monolithic processors. They decide to put multiple, relatively simple, complete processors on a single chip, called multicore. Each core represents a CPU. Now, Programmers have to take into account the parallel architecture to be more efficient and more faster. In the parallel systems, different concepts are used : SIMD (*Single Instruction, Multiple Data*), MIMD (*Multiple Instruction, Multiple Data*), Shared Memory, Distributed Memory, Interconnection networks [HW10].

3.2.1 SIMD. A concept of Flynn's taxonomy, SIMD indicates the fact that multiple processing units (core) execute the same instruction on multiple data streams. Each processing unit can perform on different data elements. The figure 3.2a gives an example of SIMD execution on three CPU.The SIMD is used for specialized problems characterized by a high degree of regularity, such as graphics/image processing (GPUs). In addition, SIMD is used in the architecture of vector processors which enable to vectorize loop in matrix calculation (see figure 3.2b). Example of SIMD Processor Arrays and Vector Pipelines : Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV, IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10.

3.2.2 MIMD. As the name suggests, MIMD or multiple instruction streams on multiple processors (cores) operate on different data items concurrently. MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control and logical unit. The execution process can be synchronous or asynchronous, deterministic or non-deterministic . The MIMD architecture is found in most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs. The figure shows an example of MIMD execution.



(a) SIMD Instruction (b) SIMD Vectorization

Figure 3.2: Single Instruction, Multiple Data (SIMD)

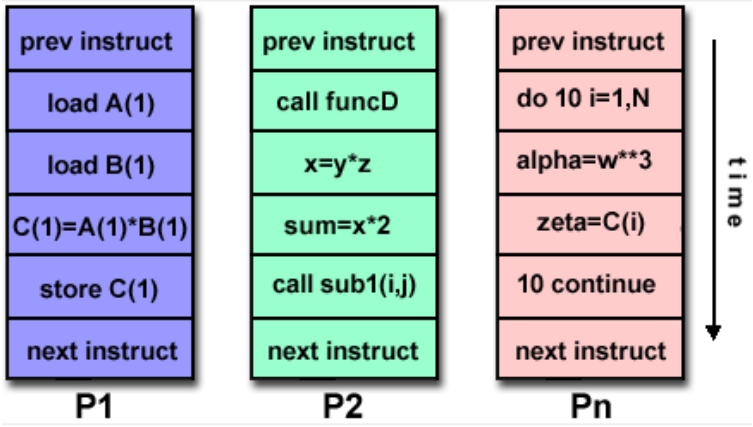
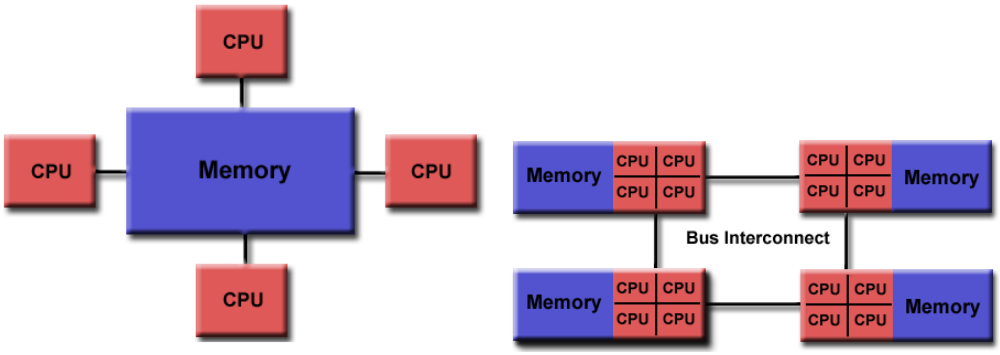


Figure 3.3: Multiple Instruction, Multiple Data (MIMD)

3.2.3 Shared Memory. A shared-memory system is a collection of autonomous processors (cores) connected to memory via an interconnection network, and processors share physical address of memory [HW10]. There are two different memory access in shared-memory system: Uniform Memory Access (UMA) and cache-coherent Non-uniform Memory Access (ccNUMA).

The UMA can be considered a 'flat' memory model, because latency and an bandwith are the same for all cpu and all memory location. Sometimes called Symmetric Multiprocessor (SMP), UMA system enables to have equal access and equal time to memory. The general problem of UMA machines is that bandwidth bottlenecks are bound to occur when the number of sockets is larger than a certain limit. The figure 3.4a show a representation of UMA architecture. In ccNUMA, memory is physically distributed but logically shared (see figure 3.4b). This architecture seems to be distrubuted-memory system, but network logic makes the aggregated memory of the whole system appear as one single address space. This simplify the memory access without resorting to a network of any kind. The locality domain (LD), set of processor cores connected locally to memory,can be considered as a UMA "building block". The ccNUMA principle gives scalable bandwidth for very large processor counts. It can be inexpensive small for two- or four-socket nodes frequently used for HPC clustering [HW10].

The LD can be, sometimes, an obstacle for high performance software on ccNUMA. The second problem is potential contention if two processors from different locality domains access memory in the same locality domain, fighting for memory bandwidth. Both problems can be solved by carefully observing the data access patterns of an application and restricting data access of each processor to its own locality domain [HW10].



(a) Shared-Memory (UMA) (b) Shared-Memory (NUMA)

3.2.4 Distributed Memory. In the distributed-memory system, each processor is connected to exclusive local memory and network communication connect also inter-processor memory (see 3.7). The memory is scalable with the number of processor and we are a rapid access of each processor with its own memory. However, the programmer must explicitly define how to access data when a processor need it in another one. Besides, the programmer is also responsible to handle the synchronization task [HW10].

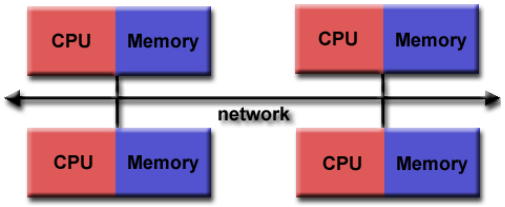


Figure 3.5: Distributed Memory

According to [HW10], there are actually no distributed-memory systems any more that implement such a layout where we HPC clustering. Most of parallel systems couple at same time shared and distributed systems (Hybrid Systems) i.e. there are shared-memory building blocks connected via a fast network. This Hybrid Distributed-Shared Memory make the system to take advantages of the two architectures and the increase of the scalability. The figure 3.6 shows an Hybrid architecture with CPU and GPU (*Graphical Processing Unit*) cores.

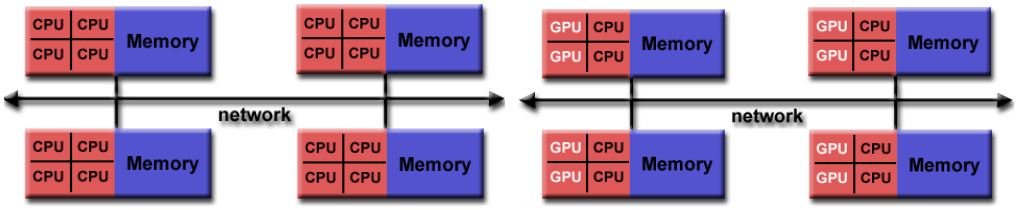


Figure 3.6: Hybrid System with CPU/GPU cores

3.2.5 Interconnection networks. In HPC domain, the interconnect plays a crucial role for latency and bandwidth performance of both distributed- and shared-memory systems.

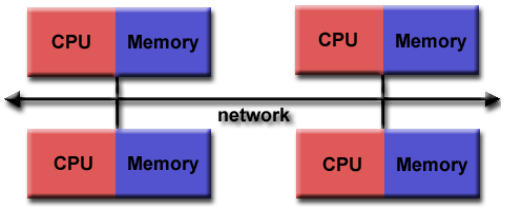


Figure 3.7: Interconnected Networks with different topologies

3.3 Parallel Programming

The parallel programming paradigm has been invented in the purpose of reducing the complexity of computation of certain problem. This implies have to be divided in multiple task that are working simultaneously in different processors unit. Early in 1960's, companies (like Intel, CRAY, IBM and others) and researchers had been working together to define the design of the supercomputer. Amdahl, research at Intel, was the first theoretical approach of speedup and efficiency of the computer system. Amdahl's law has improved the performance of the parallel computers. We witnessed another approach to compute the speedup and efficiency defined by Gustafson. All this approach measure the performance of the system in term of scalability.

3.3.1 Scalability. Whatever parallelization scheme is chosen, this perfect picture will most probably not hold in reality. The performance of parallel programs depends on multiple features such as algorithmic limitations, bottlenecks (see Figure 3.8), load balancing (e Figure 3.9), communication processes (e Figure 3.10) startup overhead etc. All this effects imply the limit of the speedup in other term the scalability. According to [], a parallel program is *scalable* if there is a rate at which the problem size can be increased so that as the number of processes/threads is increased, the efficiency remains constant. In the context parallel system, scalabality can be seen in two ways: *'strong'* scalability and the *'weak'* scalability. We say a program is *strongly* scalable, if the problem size can remain fixed, and it's *weakly* scalable if the problem size needs to be increased at the same rate as the number of processes/threads.

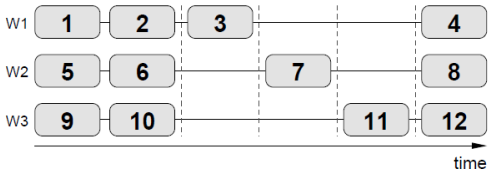


Figure 3.8: Parallelization with a bottleneck. Tasks 3, 7 and 11 cannot overlap with anything else across the dashed "barriers."

3.3.2 Speedup. Finding parallelism is not only a common problem in computing but also in many other areas like manufacturing, traffic flow and even business processes. In a very simplistic view, all execution units (workers, assembly lines, waiting queues, CPUs, ...) execute their assigned work in exactly the same amount of time.
Using parallel hardware, when we run a program with p cores, one thread or process for each core, then the parallel program will run p times faster than the serial program. When the parallel time is noticed by $T_{parallel}$ and the serial time by T_{serial} then the $T_{parallel}$ will now ideally take only T_{serial}/p. We call the program has **linear speedup** (book parallel-program).

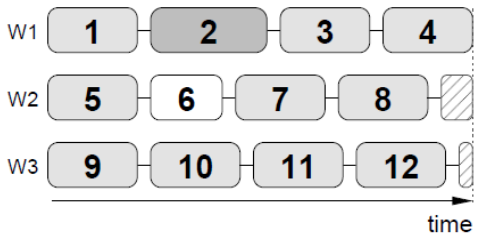


Figure 3.9: Some tasks executed by different workers at different speeds lead to load imbalance. Hatched regions indicate unused resources.

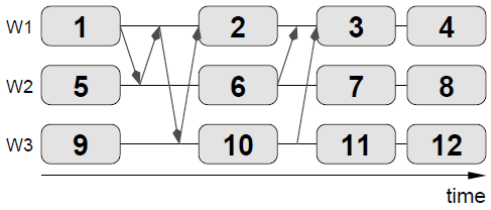


Figure 3.10: Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.

Formally, the *speedup* of a parallel program can be defined to be:

$$S = \frac{T_{serial}}{T_{parallel}} \tag{3.3.1}$$

Moreover, the **efficiency** of parallel program is defined to be the rapport of the speedup and the number of process.

$$\frac{S}{p} = \frac{\frac{T_{serial}}{T_{parallel}}}{p} = \frac{T_{serial}}{T_{parallel} \cdot p} \tag{3.3.2}$$

For a fixed problem solving with a single-worker (serial), the runtime is given by:

$$T_f^s = s + p \tag{3.3.3}$$

where s is the serial(nonparallelizable) part and p is the perfectly parallelizable fraction.
Solving the same problem on N workers (parallel) will require a runtime of

$$T_f^p = s + \frac{p}{N} \tag{3.3.4}$$

This is called *strong scaling* because the amount of work stays constant no matter how many workers are used. Here the goal of parallelization is minimization of time to solution for a given problem.

In the context of *weak scalability*, we assume that the serial runtime for the same problem is.

$$T_f^s = s + pN^{\alpha} \tag{3.3.5}$$

Consequently, the parallel runtime is

$$T_f^p = s + pN^{\alpha - 1} \tag{3.3.6}$$

3.3.3 Amdahl's Law. The application *speedup* can be defined as the quotient of parallel and serial performance for fixed problem size. Here we define "performance" as "work over time," unless otherwise noted. Serial performance for fixed problem size (work) $s + p$ is thus.

$$P_f^s = \frac{s+p}{T_f^s} = 1 \tag{3.3.7}$$

as expected. Parallel performance is in this case

$$P_f^p = \frac{s+p}{T_f^p(N)} = \frac{1}{s + \frac{1-s}{N}} \tag{3.3.8}$$

and application *speedup* ("scalability") is

$$S_f = \frac{P_f^p}{P_f^s} = \frac{1}{s + \frac{1-s}{N}} \tag{3.3.9}$$

This is called *Amdahl's law*, which was first conceived by Gene Amdahl in 1967. It limits application speedup for $N \to N$ to $1/s$. Let's consider only the parallelizable part of the calculation (for which there may be sound reasons at first sight), we get the serial performance :

$$P_f^{sp} = \frac{p}{T_f^s} = p \tag{3.3.10}$$

since $T_f^s = s + p = 1$. We derive the parallel performance to be.

$$P_f^{pp} = \frac{p}{T_f^p} = \frac{1-s}{s + \frac{1-s}{N}} \tag{3.3.11}$$

Therefore, we find the *Amdahl's law*:

$$S_f = \frac{P_f^{pp}}{P_f^{sp}} = \frac{1}{s + \frac{1-s}{N}} \tag{3.3.12}$$

Although scalability does not change with this different notion of "work," performance does, and is a factor of p smaller. If we consider the weak scaling, the serial performance can be defined to be :

$$P_v^s = \frac{s+p}{T_f^s} = 1 \tag{3.3.13}$$

and the parallel performance is .

$$P_f^p = \frac{s + pN^\alpha}{T_f^p(N)} = \frac{s + (1-p)N^\alpha}{s + pN^{\alpha-1}} = S_v \tag{3.3.14}$$

again identical to application speedup. In the special case $\alpha = 0$ (strong scaling) we recover Amdahl's Law. With $0 < a < 1$, we get for large CPU counts.

$$S_v = \longrightarrow^{N \gg 1} \frac{s + (1-p)N^\alpha}{s} = 1 + \frac{p}{s}N^\alpha \tag{3.3.15}$$

As a result, weak scaling allows us to cross the Amdahl Barrier and get unlimited performance, even for small α. In the ideal case $\alpha =$, the equation 3.3.14 is simplified to

$$S_v(\alpha = 1) = s + (1-s)N \tag{3.3.16}$$

This is called *Gustafson's Law*. The speedup is linear. Therefore, Gustafson's law addresses the lack of Amdahl's law, which is based on the assumption of a fixed problem size. Generally, in scientific simulations, researchers tend to increase the size of the problem to know the improvement of the resources. Hence,Gustafson's law allow to exploit the computing power that becomes available as the resources improve.

The parallel efficiency is also defined to be the rapport between performance on N CPUs P_N and the product of the N and the performance on one CPU P_1.

$$\epsilon = \frac{P_N}{P_1} = \frac{Speedup}{N} \tag{3.3.17}$$

3.3.4 Shared Memory Parallel Programming: OpenMP. As we said above, the shared-memory and the multicores systems architectures allow to get more performance with *"multiprocessing"* or *multi-threading* programming paradigm. All cores work concurrently in different parallel regions of the memory to give at the end the result of the instruction. In Shared-Memory programming, **OpenMP** [2] (*Open Multi-Processing*) like Pthreads and other tools have been used to exploit the multicores resources. We give here a brief description of the library OpenMP which is used in DSFDM/FFWI code.

The OpenMP is library created in 1997 and it is used in C, C++, and Fortran languages to support the multiprocessing programming paradigm in parallel systems. OpenMP is a set of compiler directives that a non-OpenMP-capable compiler would just regard as comments and ignore. Hence, a well-written parallel OpenMP program is also a valid serial program. The central entity in an OpenMP program is not a process but a thread. Threads are also called "lightweight processes" because several of them can share a common address space and mutually access data. Each thread can, by means of its local stack pointer, also have "private" variables, but since all data is accessible via the common address space, it is only a matter of taking the address of an item to make it accessible to all other threads as well. However, the OpenMP standard actually forbids making a private object available to other threads via its address.

The figure 3.11 shows in details how OpenMP thread works. In OpenMP program, a single thread, the master thread, runs immediately after startup. Truly parallel execution happens inside parallel regions, of which an arbitrary number can exist in a program. Between two parallel regions, no thread except the master thread executes any code. This is also called the "fork-join model". Inside a parallel region, a team of threads executes instruction streams concurrently. The number of threads in a team may vary among parallel regions.

We give above, as example, a simple OpenMP program in Fortran which enables the numerical integration of the following function:

$$\pi = \int_0^1 dx \frac{4}{1+x^2} \tag{3.3.18}$$

```
1 double precision :: pi,w,sum,x
2 integer :: i,N=1000000
3
4 pi = 0.d0
5 w = 1.d0/N
6 sum = 0.d0
7 !$OMP PARALLEL PRIVATE(x) FIRSTPRIVATE(sum)
8 !$OMP DO
```

[2] http://www.OpenMP.org

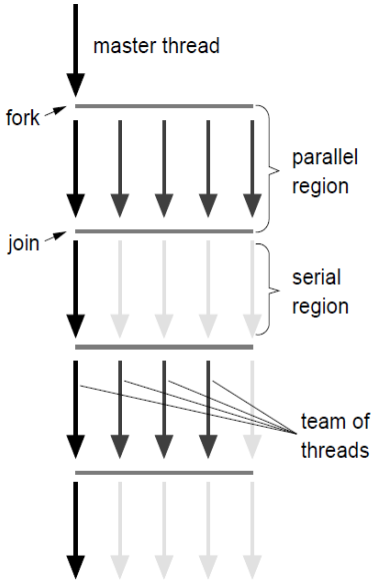


Figure 3.11: Model for OpenMP thread operations

```
do i=1,n
 x = w*(i-0.5d0)
 sum = sum + 4.d0/(1.d0+x*x)
 enddo
 !$OMP END DO
 !$OMP CRITICAL
  pi= pi + w*sum
 !$OMP END CRITICAL
 !$OMP END PARALLEL
```

Listing 3.1: Listing code of numerical integration of 3.3.18

In this simple code, every thread executes all code enclosed in the parallel region and OpenMP library routines are used to obtain thread identifiers and total number of threads. There exists other compiler directives that are available on the official documentation [http://www.OpenMP.org].

3.3.5 Distributed Memory Parallel Programming: MPI. The second programming model is the *Message Passing* which is dedicated for distributed memory system. The *Message Passing Interface* (**MPI**) was designed early 1990s to facilitate the communication between processes. The MPI library was coming after many standardization efforts between industrials and scientific community, to allow parallel program to be easily portable between platforms. It is regarded as a necessary ingredient in any HPC system installation, and numerous types of interconnect are supported [HW10].

We can remember that MPI is just a standard but its implementation is available under different distributions such as *MVAPICH*, *Open MPI*, *Intel MPI*, *IBM BG/Q MPI* and *IBM Spectrum MPI*. The main implementations are written in language like Fortran, C or C++. The DSFDM/FFWI code was compiled with *Intel MPI*.

This simple example of "Hello World" give an overview of a parallel program using MPI [HW10].

```
program helloWorld

use MPI
```

```
4
5  integer :: rank, size, ierror
6
7  call MPI_Init(ierror)
8  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
9  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
10
11 write(*,*) 'Hello World, I am ',rank,' of ',size
12
13 call MPI_Finalize(ierror)
```

Listing 3.2: Listing code of "Hello World" in parallel

In this example, the call of `MPI_Init()` initializes the parallel environment. Upon initialization, MPI sets up the so-called world communicator, which is called `MPI_COMM_WORLD`. A communicator defines a group of MPI processes that can be referred to by a communicator handle. The `MPI_COMM_WORLD` handle describes all processes that have been started as part of the parallel program. If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`. Nearly all MPI calls require a communicator as an argument. The calls to `MPI_Comm_size()` and `MPI_Comm_rank()` in lines 8 and 9 serve to determine the number of processes (size) in the parallel program and the unique identifier (`rank`) of the calling process, respectively. Note that the C bindings require output arguments (like rank and size above) to be specified as pointers. The ranks in a communicator, in this case `MPI_COMM_WORLD`, are consecutive, starting from zero. In line 13, the parallel program is shut down by a call to `MPI_Finalize()`. Note that no MPI process except rank 0 is guaranteed to execute any code beyond `MPI_Finalize()`.

The table 3.1 shows some point to point and collective communication routines for MPI (Fortran subroutines).

3.4 Optimization Techniques in HPC

The principal objective that encouraged the use of HPC comes from a real need to improve the performance of scientific and industrial problems which require a great spatial and temporal complexity. In this optic, HPC applications need to optimize their communications and synchronizations to deliver high performance. Although the advanced of modern CPU, the increasing of memory storage and the improvement of interconnection networks, optimization is a difficult and often tedious process, it is usually more cost-effective than investing in additional system resources. In the context of parallel system, the optimization is composed of set of elements that must be taken into account together to do the program more efficient. These elements includes compilers directives, multi-threaded optimization, SIMD vectorization, placement of MPI processes, optimization libraries, code profiling.

3.4.1 Compiler Optimization. Compilers are the basic tools applied to HPC applications to get good performance from the target hardware. Usually, several compilers are available for each architecture. The most important languages for High Performance Computing are FORTRAN, C, and C++ because they are the ones used by the vast majority of codes. GNU Compiler Collection (GCC) and Intel are the most used compiler in HPC. Here, we gives briefly some important compiler directives to optimize the HPC code.

- **GNU Compiler (GCC)** The GNU compiler tool chain contains front ends for C, C++, Objective-C, FORTRAN, Java, Ada, and Go, as well as libraries for these languages. These compilers are 100% free software and can be used on any system independently of the type of hardware. It is

subroutine	Description
MPI_SEND()	Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.
MPI_RECV()	Receive a message and block until the requested data is available in the application buffer in the receiving task.
MPI_SSEND()	Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.
MPI_SENDRECV()	Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.
MPI_WAIT()	MPI_Wait blocks until a specified non-blocking send or receive operation has completed.
MPI_Probe()	Performs a blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag.
MPI_GET_COUNT()	Returns the source, tag and number of elements of datatype received.
MPI_BARRIER ()	Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.
MPI_BCAST()	Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.
MPI_SCATTER()	Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.
MPI_ALLREDUCE()	Collective computation operation + data movement. Applies a reduction operation and places the result in all tasks in the group.
MPI_REDUCE_SCATTER()	Collective computation operation + data movement. First does an element-wise reduction on a vector across all tasks in the group.
MPI_ALLTOALL()	Data movement operation. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

Table 3.1: MPI communication routines

the default compiler for the Linux operating system and is shipped with it []. The GCC/GFortran compiler has to be configured and compiled on a specific target system, so it may not support some features and compiler technologies depending on the configure arguments. The table 3.2 shows a list of gnu optimizer options. Flags are obtained with gcc/gfortarn -Q –help=optimizers (GNU Compiler documentation[3]).

GCC compiler option	Explanation
-march=	Generate code for given CPU for example "corei7-avx" Sandy Bridge CPU and use autovectorization AVX
-mtune=	Schedule code for given CPU
-O0	Reduce compilation time and keep execution in the same order as the source line listing so that debugging produces the expected results. This is the default.
-O or -O1	The first optimization level. Optimizing compilation takes more time and a lot more memory for a large function. With -O/-O1, the compiler tries to reduce code size and execution time, avoiding any optimizations that take would significantly increase compilation time.
-O2	The second optimization level. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.
-Os	Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.
-O3	The third optimization level. -O3 turns on all optimizations specified by -O2.

Table 3.2: GNU Compiler options

- **Intel Compiler** The Intel compiler suite is traditionally the most efficient compiler for the Intel CPU as it is designed by Intel. The latest features of the Intel processors Intel Xeon are implemented in the Intel IntelÂ® Parallel Studio XE []. It includes a FORTRAN, C, and C++ compiler, an MKL library, VTune and Intel Advisor profiler, and other components (Cilk, IPP, TBB,). The following compiler directives in 3.3 enable the general optimization option in Intel CPU using Intel Compiler (see Intel Compiler documentation[4]).

3.4.2 Vectorization. The HPC program can get more performance by using the vectorization capability of modern CPU. Vectorization allows you to execute a single instruction on multiple data (SIMD) ob-

[3] https://gcc.gnu.org/onlinedocs/
[4] [?]

jects in parallel within a single CPU core (see section 3.2a). This is different from task parallelism using MPI, OpenMP or other parallel libraries where additional cores or nodes are added to take care of data belonging to separate tasks placed on different cores or nodes. The industry of microprocessor has developed SIMD vectorization in the architecture of the modern CPU. AVX (Advanced Vector Extensions) and SSE (Streaming SIMD Extensions) are two vector instructions available in most supercomputers. AVX were introduced by Intel with the *Sandy Bridge* processor shipping in 2011 and later that year by AMD with the Bulldozer processor. AVX instructions simultaneously operate on 8 pairs of single precision (4 bytes) operands or 4 pairs of double precision (8 bytes) operands, etc. This can speed up a vectorizable loop by up to 8 times in case of single precision (4 bytes), or 4 times in case of double precision (8 bytes), etc. There exists an extension AVX 2 which expands most integer commands to 256 bits and introduces fused multiply-accumulate (FMA) operations. The SSE is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III. It use 128-bit registers to operate simultaneously on two 64-bit (8-byte) double precision numbers, four 32-bit (4-byte) floating point numbers, two 64-bit integers, four 32-bit integers, eight 16-bit short integers or sixteen 8-bit bytes.

The vectorization can be done by calling OpenMP directives or it is performed automatically after an using appropriate compiler flags. The following example vectorizes a loop .

```
1  subroutine sub(a,b,n,k)
2  integer n, k
3  real a(n), b(n)
4  integer i
5
6  !$omp simd vectorlength(8)
7  do i=k+1,n
8
9     a(i) = 0.5 * (a(i-k) + b(i-k))
10
11 end do
12
13 end
```

Listing 3.3: Listing code of loop vectorization

We can also use the auto-vectorization with the directive "-O3 SSE3", "-O3 -xAVX" or "-O3 -xCORE-AVX2".

3.4.3 Threading/Multi-threading. Exploiting the multithreading techniques can increase considerably the performance of HPC code. As we said above, the threading model for HPC is handled by OpenMP or Pthread. In the thread level parallelism (TLP), the parallel code will get performance where the number of threads is increased and threads are instantiated and executed independently. However, multi-threading keeps CPU busy by engaging multiple threads. By using OpenMP threads, we study in section the scalability of DSFDM/FFWI code in multi-threading level.

The *Hyper-threading* and *Affinity* are features of performance in many Intel® CPU (Intel® Core™vPro ™processors, the Intel® Core™processor family, the Intel® Core™M processor family, and the Intel ® Xeon® processor family). *Hyper-threading* is Intel's proprietary simultaneous multithreading (SMT) implementation used to improve parallelization of computations performed on x86 microprocessors. Instructions from more than one thread can be executing in any pipeline stage at a time. *Thread Affinity* is the ability to bind threads to physical processing units. Thread affinity restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect

on the execution speed of a program. the `KMP_AFFINITY` environment variable controls the affinity interface in OpenMP. There are three ways to manage the type of affinity: *none*(default), *compact*, *scatter*. The figures [3.13, 3.12] show the difference between *compact* and *scatter*.

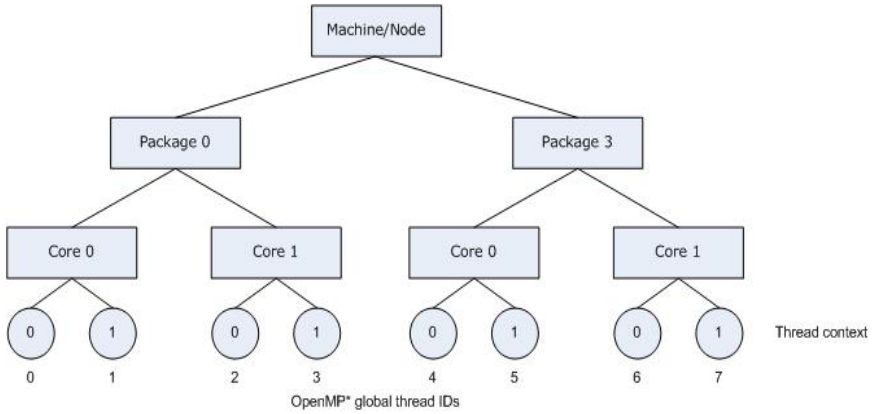


Figure 3.12: Compact affinity: $KMP\_AFFINITY = granularity = fine, compact$.

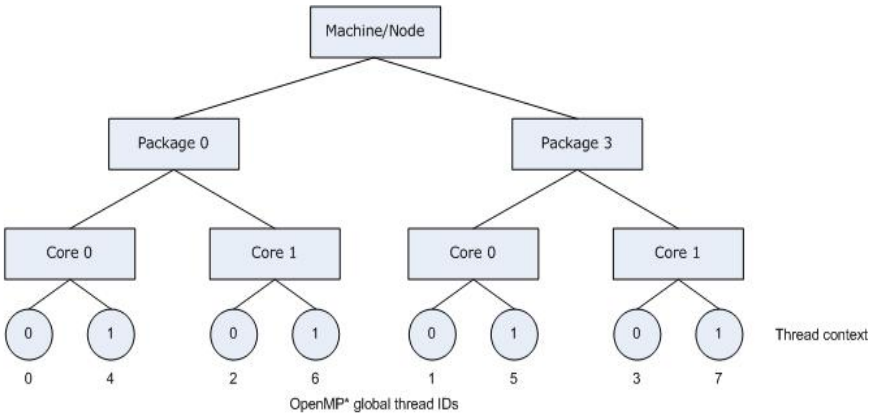


Figure 3.13: Scatter affinity: $KMP\_AFFINITY = granularity = fine, scatter$.

3.4.4 MPI processes. MPI, *Massage Passing Interface*, dedicated for parallel distributed system have great impact for code optimization in HPC environment. For more details see 3.3.5. The study of scalability, in section 4, gives more details of the performance of DSFDM/FFWI.

3.4.5 Code Profiling. Code Profiling helps in identifying algorithmic bottlenecks (hotspots), and also communication overheads in case of distributed code. Most of the execution time can be spent in the function which is repeatedly calling other slower functions or there can be slower pieces in the code. Several tools are available for profiling. *Gprof* is most frequently used in the HPC community. There are advanced profilers used in the context of this *Advisor* and *VTune* Amplifier from Intel®, *Allinea*, *BPM* (old name *MPIPROF*), *Bull IO Instrumentation*.

- **Advisor :** Advisor is a profiler from Intel®. It is used used early in the process of adding vectorization into the code, or while converting parts of a serial program to a parallel (multithreaded)

program. It helps the user to explore and to locate areas in which the optimizations might provide significant benefit. It also used to predict the costs and benefits of adding vectorization or parallelism to those parts of the program, allowing the user to experiment. The figure 3.14 shows the summary of profiling code.

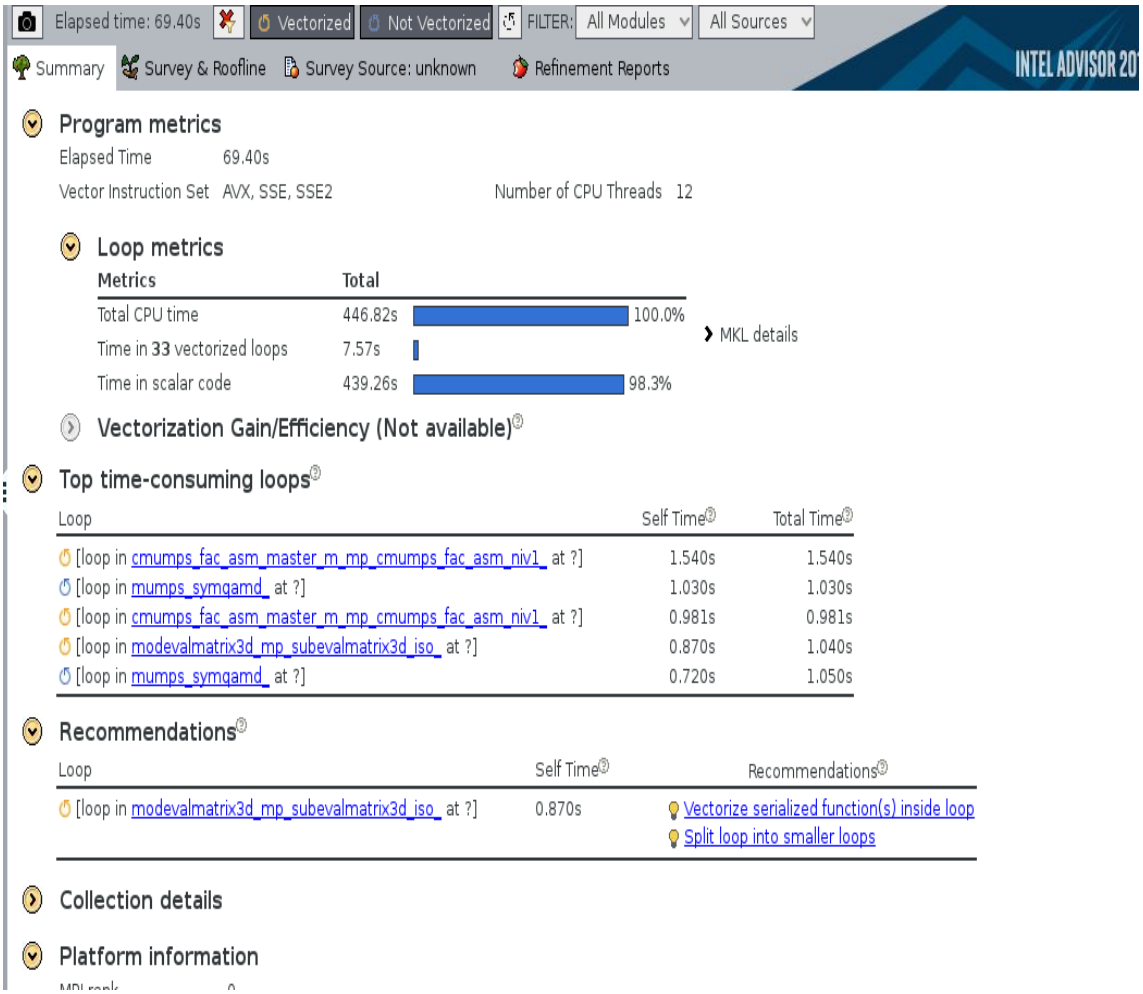


Figure 3.14: Intel® Advisor: Collect survey

- **Intel® VTune Amplifier** VTune Amplifier is a performance profiler from Intel®. it can identify where in the code time is being spent in both serial and threaded applications. For threaded applications, it can also determine the amount of concurrency and identify bottlenecks created by synchronization primitives. Users can use graphic interface (GUI) to get deep analysis of hotspots and bottlenecks. The figure 3.16 shows the user interface of VTune Amplifier.
- **Allinea Map :** Allinea MAP is the profiler for parallel, multithreaded or single threaded C, C++, Fortran and F90 codes. It provides in depth analysis and bottleneck pinpointing to the source line. Unlike most profilers, it's designed to be able to profile pthreads, OpenMP or MPI for parallel and threaded code. It exposes a wide set of performance problems and bottlenecks by measuring:
  - Computation - with self and child and call tree representations over time
  - Thread activity - to identify over-subscribed cores and sleeping threads that waste available CPU time for OpenMP and pthreads

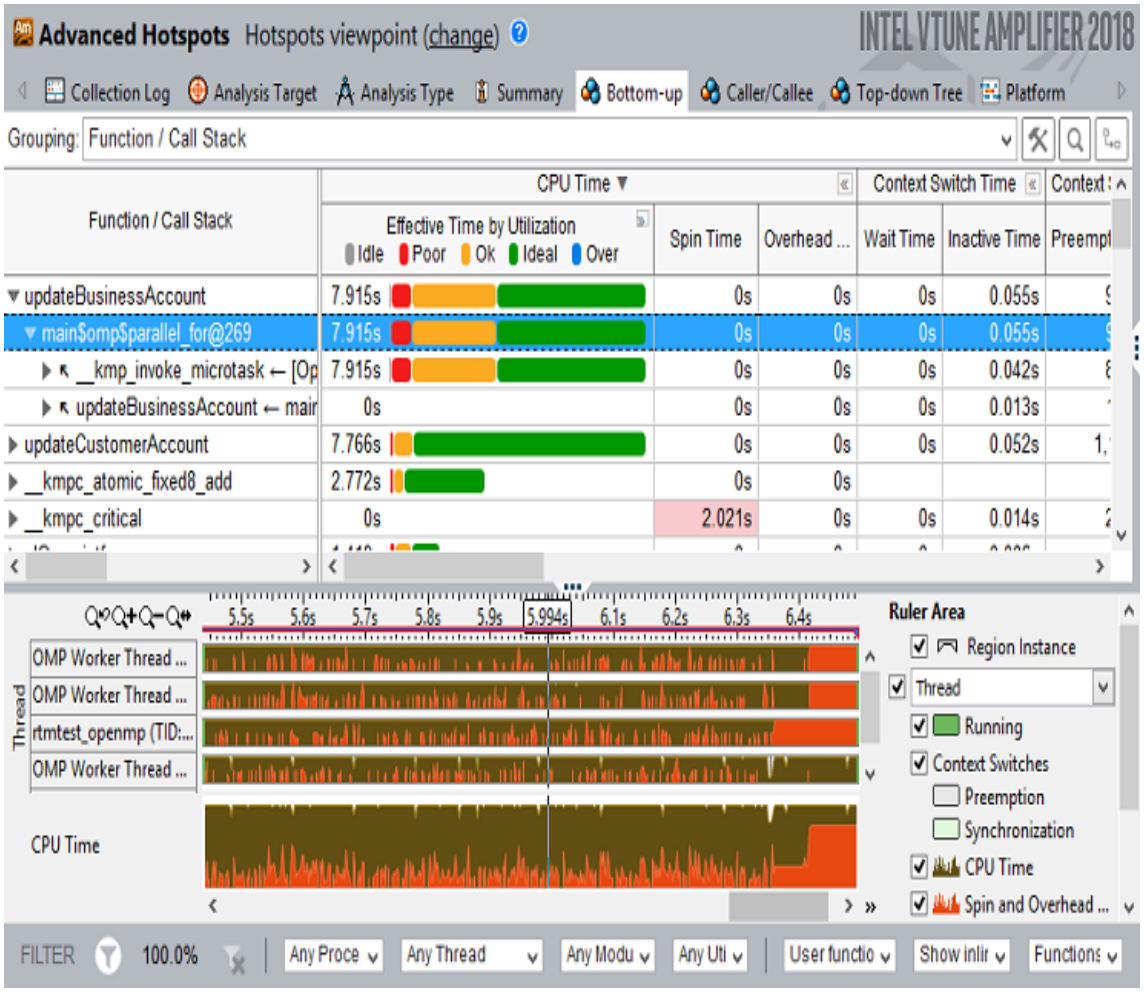


Figure 3.15: Intel® VTune Amplifier: Hotspot analysis

  - Instruction types (for x86_64) - to show use of vector-units or other performance extensions
  - Synchronization, communication and workload imbalance for MPI or multi-process usage
  - I/O performance and time spent in I/O - to identify bottlenecks in shared or local file systems

  The figure **??** shows an example of analysis of selected region.

- **BPM (MPIPROF) :** BPM Profiler developed by Bull Atos group is is a library to extract MPI information like Statistic per process,communication matrix and timeline.
- **Bull IO Instrumentation** (Note available)

3.4.6 Optimization Libraries. There exists many software using by HPC developers to optimize Math calculus, Memory consumption, communication, IA algorithms and so on : *Math Kernel Library* (**MKL**) from Intel, *Basic Linear Algebra Subprograms* (**BLAS**), *Linear Algebra PACKage* (**LAPACK**) and **ScaLAPACK**, *Portable, Extensible Toolkit for Scientific Computation* (**PETSc**), *Hierarchical Data Format 5* (**HDF5**), *Intel® Data Analytics Acceleration Library* (**Intel® DAAL**) etc.

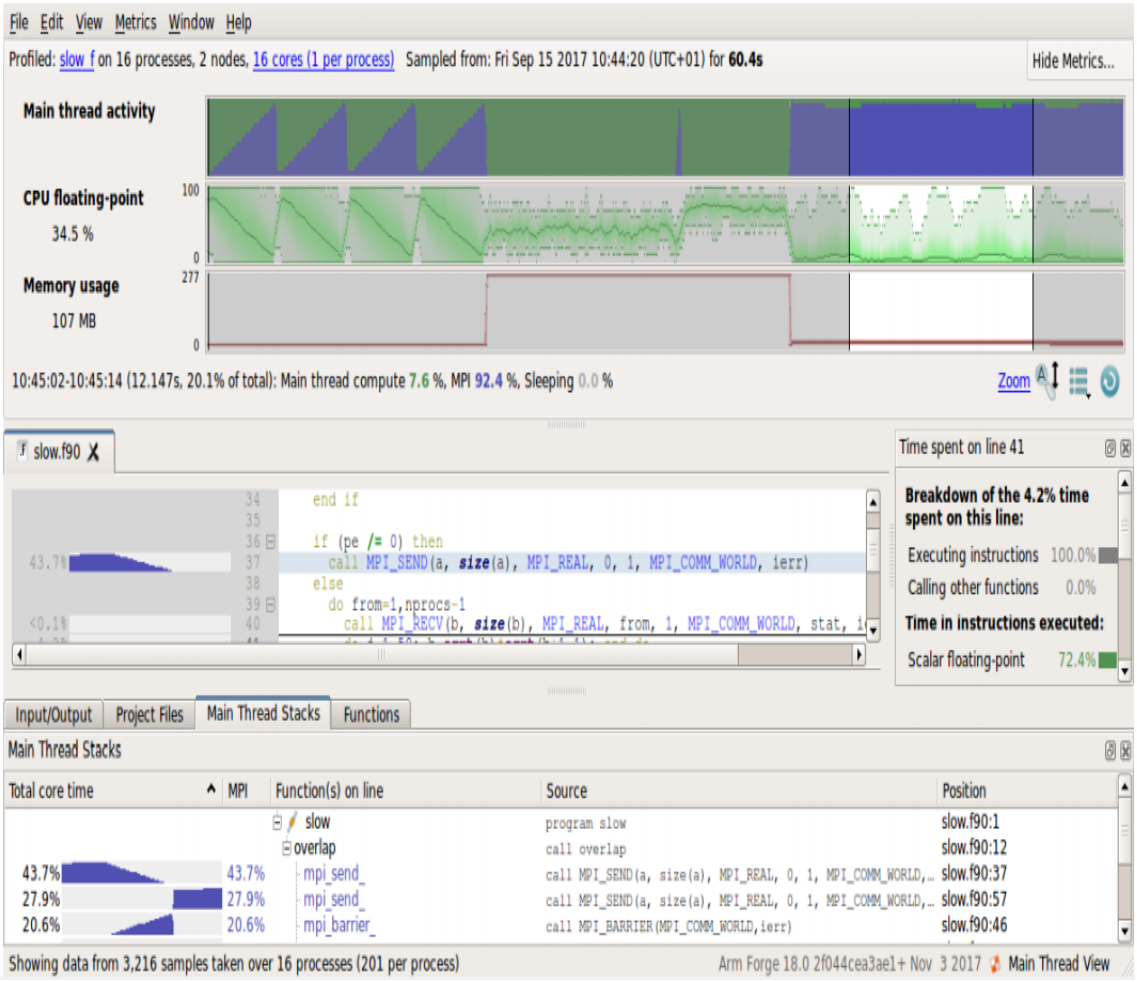


Figure 3.16: Allinea map:Map with a region of time selected

Intel compiler option	**Explanation**
-O0	No optimization. Used during the early stages of application development and debugging. Use a higher setting when the application is working correctly
-Os or -O1	Optimize for size. Omits optimizations that tend to increase object size. Creates the smallest optimized code in most cases. These options may be useful in large server/database applications where memory paging due to larger code size is an issue.
-O2	Maximize speed. Default setting. Enables many optimizations, including vectorization. Creates faster code than -O1 in most cases.
-O3	Enables -O2 optimizations plus more aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking to allow more efficient use of cache and additional data prefetching. The -O3 option is particularly recommended for applications that have loops that do many floating-point calculations or process large data sets. These aggressive optimizations may occasionally slow down other types of applications compared to -O2.
-qOpenMP	Causes multi-threaded code to be generated when OpenMP directives are present. For Fortran only, makes local arrays automatic and may require an increased stack size. See http://www.OpenMP.org for the OpenMP API specification.
-parallel	The auto-parallelizer detects simply structured loops that may be safely executed in parallel, including loops implied by Intel® Cilk™Plus array notation, and automatically generates multi-threaded code for these loops.
-mkl=*name*	Requests that the Intel® Math Kernel Library (Intel® MKL) be linked. Off by default. Possible values of name are: parallel Links the threaded part of Intel MKL (default) sequential Links the non-threaded part of Intel MKL cluster Links the cluster and sequential parts of Intel MKL

Table 3.3: Intel Compiler options

4. Performance study of DSFDM/FFWI code on MESCA II node

4.1 DSFDM/FFWI

4.1.1 Description. The DSFDM/FFWI is a package which contains parallel computer codes to perform seismic modeling and fullwaveform inversion in the frequency domain. These codes have have been designed to execute modeling/inversion in 1D, 2D, 3D. According to the authors [OVA+07, OBC+14, OMB+15, BESJ10, ABB+16], only the 3D version is tested and Documented (see documentation). The DSFDM code is the seismic modelling code, while the FFWI code performs the inversion, using most of the subroutines implemented in DSFDM.Moreover,the DSFDM/FFWI codes assume visco-acoustic submarine supports in which vertical transverse isotropy can be taken into account.

Seismic modeling is performed in the frequency domain with a finite-difference method on a uniform Cartesian grid [OVA+07, OBC+14, BESJ10]. The linear system resulting from the discretization of the time-harmonic wave equation is solved with a sparse direct solver. Sources and receivers, which can be processed in a reciprocal way, can be considered at arbitrary positions in coarse finite-difference grids with the sinc parameterization developed by Hicks [Hic02]. Absorbing boundary conditions are perfectly-matched layers (PMLs) [Ber94, OVA+07]. A free-surface boundary condition can be used along arbitrary tomography by forcing the pressure wave-field to 0 along this boundary. All of the tasks performed during seismic modelling (building impedance matrix, building right-hand side vectors, call of MUMPS subroutines, writing of solutions and extraction at receiver solutions) are implemented in the DSFDM code.

FFWI code is implemented with various local optimization methods. Recorded data are provided in the frequency-domain using an adaptation of the Seismic Unix (SU) format. The data are complex-valued (in single precision) and the frequency interval is uniform. The first frequency, the number of frequency and the frequency interval must be provided at specific locations in the SU headers. The numerical optimization is performed with the SEISCOPE optimization toolbox and allows for steepest-descent, conjugate gradient,l-BFGS and truncated Newton optimizations [MB16]. Seismic modelling and inversion are performed on the same Cartesian grid. Source wavelet estimation can be performed at each iteration of the FWI. Only the preconditioned steepest-descent optimization was tested, although conjugate gradient and l-BFGS optimizations are interfaced with the FFWI code. Interfacing Gauss-Newton and Newton truncated optimizations is the aim of ongoing work. The code has been tested for the update of the vertical wavespeed only. An application to real data is shown in [OMB+15]. Multi-parameter gradients are implemented but still need to be validated.

4.2 MUMPS Solver

MUMPS ("MUltifrontal Massively Parallel Solver") is a package for solving systems of linear equations of the form $Ax = b$, where A is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric, on distributed memory computers. The MUMPS package is designed to solve linear equations by using a direct method based on a multifrontal approach which performs a Gaussian factorization

$$A = LU$$

where L is a lower triangular matrix and U an upper triangular matrix. If the matrix is symmetric then the factorization

$$A = LDL^T$$

where D is block diagonal matrix with blocks of order 1 or 2 on the diagonal is performed.

MUMPS solver results of collaboration between different partners working in the MUMPS project since 1996 (see http://mumps.enseeiht.fr/). Today, the MUMPS team includes **CERFACS**, **CNRS**, **ENS-Lyon**, **INRIA**, **INPT**, **University of Bordeaux**. In the recent years, the popularity of MUMPS package has increased in different countries and different scientific projects. It has known different version during the years with real improvement of the LU factorization methods. The recent research of MUMPS project is focused on the Block-Low rank option that allows decreasing the complexity of sparse direct solvers on problems arising from partial differential equations is provided for experimentation purpose [AAB+15].

A matrix A of size $m \times n$ is said to be low rank if it can be approximated by a low-rank product $\tilde{A} = XY^T$ of rank k_ϵ, such that $k_\epsilon(m+n) \leq mn$ and $\|\tilde{A} - A\| \leq k_\epsilon$. The first condition states that the low-rank form of the matrix requires less storage than the standard form, whereas the second condition simply states that the approximation is of good enough accuracy. Using the low-rank form also allows for a reduction of the number of floating-point operations performed in many kernels (e.g., matrix-matrix multiplication).

Thanks to the low-rank compression, the theoretical complexity of the factorization is reduced from $O(n^6)$ to $O(n^{5.5})$ and can be further reduced to $O(n^5 \log 5)$ with the best variant of the BLR format [ABB+16].

4.3 Tools

As we mentioned above, this DSFDM/FFWI version use MUMPS package to solve the linear system of discretized wave propagation equation. We need to install different packages for MUMPS dependencies as well as DSFDM/FFWI. MUMPS can be installed in sequential version. In the same way, it can be used with multithreaded machine (with OpenMP) or with parallel version (distributed memory MPI based). We need these packages for MUMPS installation:

- **BLAS library**: BLAS (*Basic Linear Algebra Subprograms*) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. BLAS is written in Fortan and it has an optimized sequential or multithreaded version.
- **LAPACK library**: LAPACK or Linear Algebra PACKage routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). LAPACK is designed at the outset to exploit the Level 3 BLAS. Because of the coarse granularity of the Level 3 BLAS operations, their use promotes high efficiency on many high-performance computers, particularly if specially coded implementations are provided by the manufacturer.
- **ScaLAPACK library**: Similar to LAPACK, ScaLAPACK is a library of high-performance linear algebra routines for parallel distributed memory machines. It solves dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems. The use of ScaLAPACK provides efficiency, scalability, reliability, portability, flexibility, and ease of use (by making the interface to LAPACK and ScaLAPACK look as similar as possible).

- **BLACS library**: In the context of distributed parallel machines, the BLACS (Basic Linear Algebra Communication Subprograms) routines is created to make easy linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.
- **OpenMP library**: As we discuss in the section 3.3.4, this package allows parallel programming in shared memory multi-core platforms. The openMP directives improves strongly the performance of MUMPS. OpenMP parallel regions of MUMPS can be set through the OMP_NUM_THREADS environment variable.
- **MPI library**: Intel MPI is installed for parallel distributed version of MUMPS (see section 3.3.5 for more details)
- **Others**: **CMake**, **SCOTCH**, **ParMETIS**.

4.4 Installation of DSFDM/FFWI

This work was doing in a parallel distributed system, called *cluster* . We are installed DSFDM/FFWI code in two cluster nodes serving reference for the study. Each cluster has different nodes working together in parallel. We need login and password to get securely (ssh protocol) the login. Slurm [1], a distributed task manager, allows users to submit, cancel and handle their jobs. The installation of DSFDM/FFWI was done after installing all packages cited below. Script shell and Linux commands are used to do the job. See DSFDM/FFWI documentation for more details about installation. The following packages are used for installation:

- **MUMPS** 5.1.1 is used for direct solver.
- **SEISCOPE Optimization TOOL BOX**[2] is a set of FORTRAN 90 optimization routines dedicated to the resolution of unconstrained and bound constrained nonlinear minimization problems. It contains six optimization methods : (Preconditioned) Steepest Descent, (Preconditioned) Nonlinear Conjugate Gradient, l-BFGS method, Preconditioned l-BFGS method, Truncated Newton method, Preconditioned Truncated Newton method.
- **CWP/SU**[3] is a graphic tool for plotting seismic imaging results.
- **MPI** : Intel MPI is used.
- **Boost**[4] (and **BoostMPI**) tries to become a standard C++ library and provide a set of structure and task using in different domains such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing.

The installation of DSFDM/FFWI in the cluster takes into account the optimisation flags:
`-O3 -xCORE-AVX2 -fma -mkl -g`

[1] https://slurm.schedmd.com
[2] https://seiscope2.osug.fr/SEISCOPE-OPTIMIZATION-TOOLBOX
[3] http://www.cwp.mines.edu/cwpcodes/index.html
[4] http://www.boost.org/

4.5 Methodology

The aim of this study was, firstly, to evaluate the capacity of the DSFDM/FFWI code to process large test cases on MESCA-II node and to improve the performance of the code in this architecture and secondly, to implement another solver like `QR_MUMPS` that will serve us to compare its performance with MUMPS. This implies how to keep the code scalable when the memory consumption and MPI communications are increasing. In [OVA+07], the authors show that the memory consumption and elapsed time are mostly based on LU matrix factorisation doing by MUMPS package. Their results explain in more details this fact. The MUMPS team has developed the BLR option to improve the performance of the storing of LU matrix values [AAB+15].

As we discuss above (see 3.4), the optimization of DSFDM/FFWI can be seen in multiple levels. In this study, we will focus on scalability study (strongly and weakly) and different profiling methods have been used to analyse the performance of the code.

4.6 Results and discussion

The scalability, as we discussed in section 3.3, is the coarse grained of the study of performance. It can be measured in two scales: strong scaling and weak scaling. Here, we study these two scales in different nodes cluster such as Mesca and classical nodes. When we say classical nodes it refers the other nodes different to Mesca.

This study of strong scalability allow us to understand how much faster our DSFDM/FFWI code is with N processors by fixing the size of the problem. The principle the weak scalability is to keep the work per fixed worker and add more workers/processors.Firstly, we perform different test cases in classical nodes to be comparing the performance with Mesca. To study the scalability, we use the homogeneous isotropic model that aims to validate DFSFDM against an analytical solution computed in an homogeneous infinite acoustic isotropic ($ianiso = 0$) medium (see DSFDM/FFWI documentation). According to the documentation, we fix, firstly, the size of the problem to be grid $(n1, n2, n3) = (41, 41, 121)$ ie. $4km \times 4km \times 12km$ (depth,X,Y). The others parameters are the same as the documentation. The wavespeed and the density are respectively equal to $1.5km/s$ and $1000km/m3$. The grid interval is $100m$. The frequency is $3.72Hz$. The shot coordinates in meters are $(x1, x2, x3) = (1000, 2000, 2000)$.

The results obtained in Figures 5.1-5.5 show the comparison of **DSFDM** with the analytical solution (**FDTD**).

In the first test case, we run the code in a classical node such that the architecture is given in Table 4.1

The table 4.2 shows the computational resources used when the code is performing in this node.

Secondly, we execute the same test case with the same parameters in the Mesca node. The architecture of this node is showing in table 4.3.

These results are shown in the table **??**

By using two MPI processes and ten threads, the classical node shows almost the same performance as Mesca($T_{LU}(s) \approx 17.7s$). In addition, the total memory allocated for factorization was 6.7 Gigabyte.The time of LU factorization takes 74 % and 70 % respectively in the classical node and Mesca. This type of node will be used for rest of the simulation. The results are shown in tables **??** and 4.2.

Number of sockets (CPUs)	2 sockets Intel(R) Xeon(R) CPU E5-2697 v2
Number of Core(s)	2 * 12 cores per socket at the nominal frequency 2.70GHz
Hyperthreading	activated
Number of Thread(s)	2 * 24 threads per cpu
Shared Memory	2*32 GB DDR* (* unknown)
L1d/L1i cache	32K
L2 cache	256K
L3 cache	30720 KB

Table 4.1: Architecture of classical nodes

Grid dimensions	npml	#u	#n	#MPI	#th	$Mem_{LU}(Gb)$	$T_{LU}(s)$	$T(s)$
$41 \times 41 \times 121$	8	1	1	2	10	6,7	17.76	23.77

Table 4.2: Homogeneous Isotropic running on classical node: #u(10^6): number of unknowns. #n: number of nodes. #MPI: number of MPI process. #th: number of threads per MPI process. $Mem_{LU}(Gb)$: Memory for LU factorization in Gbytes. $T_{LU}(s)$: Elapsed time for factorization in s. $T(s)$: Running time

Number of sockets (CPUs)	8 sockets Intel(R) Xeon(R) CPU E7-8890 v4
Number of Core(s)	8 * 24 cores cores at the nominal frequency 2.20GHz
Hyperthreading	deactivated
Number of Thread(s)	8 * 24 threads per cpu
Shared Memory	8*340 GB DDR* (* unknown)
L1d/L1i cache	32K
L2 cache	256K
L3 cache	61440K

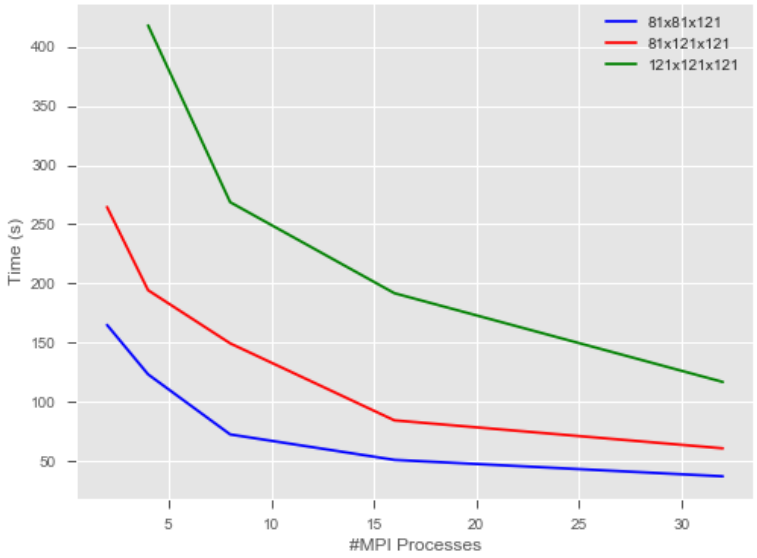
Table 4.3: Architecture of Mesca II node

Grid dimensions	npml	#u	#n	#MPI	#th	$Mem_{LU}(Gb)$	$T_{LU}(s)$	$T(s)$
$41 \times 41 \times 121$	8	1	1	2	10	6,7	17.73	25.048

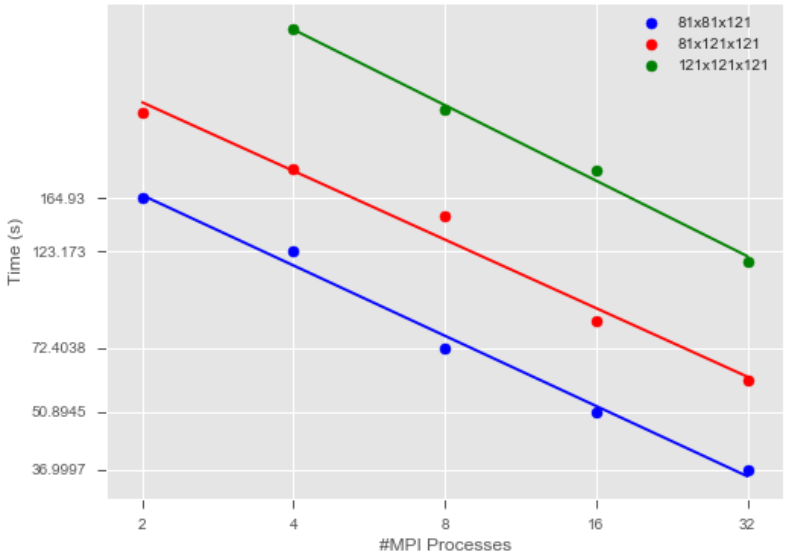
Table 4.4: Homogeneous Isotropic running on Mesca II node

In the following sentences, we define three test cases depending on the size of the grid to study the strong scalability of DSFDM code. This grid is a 3D dimension, $n1, n2, n3$, with PMLs. The chosen sizes of the grid are $(n1, n2, n3) = (81, 81, 121)$, $(n1, n2, n3) = (81, 121, 121)$ and $(n1, n2, n3) = (121, 121, 121)$. The grid interval is 100. We perform these tests in node cluster which contains the classical nodes and Mesca node described above. The resolution of wave propagation equation is handled by MUMPS solver. Here, we are more interested in factorization phase performing by MUMPS solver. This factorization can be doing in two ways: Full Rank and Low-Block Rank. In the following simulation, we will focus on the Full Rank option. In all the tests, we consider the number of MPI processes is always less or equal than the number of sockets (or CPU).

The Figures 4.1a and 4.1b show the elapsed time of three different test cases running on classical nodes. As we can see in Figure 4.1a, the Time of LU factorization depending on numbers of processes decreased



(a) Time Elapsed for LU factorization performing in classical nodes



(b) Elapsed time for LU factorization in log scale performing in classical nodes

by following a power law decay. We remark also, we obtain a good approximation when the numbers of processes is equals to a power of 2 (see Figure 4.1a). A log-log plot, (Figure 4.1b), shows the strong scalability of the tests. In the case of $(n1, n2, n3) = (121, 121, 121)$, we observe a rapid drop of elapsed time between 2 and 8 processes and smooth descent of the curve in 8 and 32. The table 4.5 shows the curve fitting of the the log scale plots (ie. $Y = A * log(N) + B$, with N is number of processes).

Similarly, we get the same remark in Mesca node (see Figures 4.2a and 4.2b). Here, we see a rapid growth between 2 and 4 processes different of the classical nodes. In case of $(n1, n2, n3) = (81, 81, 121)$,

Size	A	B
$81 \times 81 \times 121$	-0.55	2.4
$81 \times 121 \times 121$	-0.54	2.6
$121 \times 121 \times 121$	-0.60	2.9

Table 4.5: Parameters of the fitting curves of Elapsed time in classical nodes

the get weak performance between 4 and 16, we can deduct we have got the limit of the scalability.The elapsed time is fitted in log scale plots such that the slope and the intercept are given by the table 4.6.

Size	A	B
$81 \times 81 \times 121$	-0.62	2.16
$81 \times 121 \times 121$	-0.74	2.5
$121 \times 121 \times 121$	-0.72	2.81

Table 4.6: Parameters of the fitting curves of Elapsed time in Mesca

In [OVA+07], the authors defend that the most execution time is doing by MUMPS solver particularly in LU factorigation stage. Curiously, we have computed the ratio between elapsed time for LU factorisation and the Runtime. Surprisingly, the percentage of LU time decreased rapidly in function of the number of processes (Figures 4.3a and 4.3a). Firstly, we distinguish that the elapsed time of LU factorization take about 90 % of the execution time when the number of process is less than 2. This fact implies the running can be increased by different features such as the time used by communication processes, load imbalance, parallel startup overhead, the hotspots, resolution phase of $Ax = b$ in MUMPS and so one . We can get more details by using some profiling tools (see section 3.4.5). A comparison of these values in the two architectures shows that the Mesca node looses much time in non-LU process (see Figure **??**).

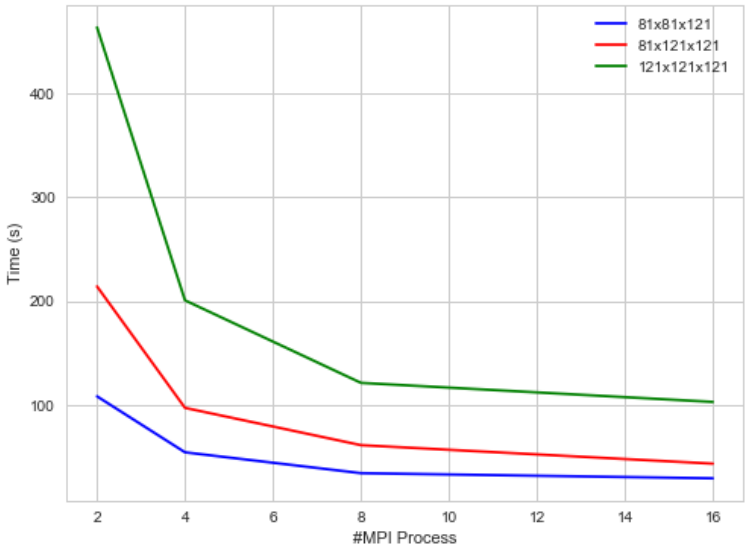
To understand the scalability en term of number of threads in Mesca node, we fix a new test case $(n1, n2, n3) = (202, 202, 202)$. The Math Kernel Library (Intel®MKL) threads are used in this test. The figure 4.4 shows a great performance obtained in Mesca II. We can conclude that the OpenMP threads and the math functions included in Intel MKL improve the scalability of DSFDM/FFWI code. We recall that the Intel MKL have been used in different library (BLAS, ScaLAPACK, BLACS) compiled with MUMPS solver.

The Intel®OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. The interface is controlled using the KMP_AFFINITY environment variable, and with recent versions of Intel compilers, the KMP_PLACE_THREADS environment variable.

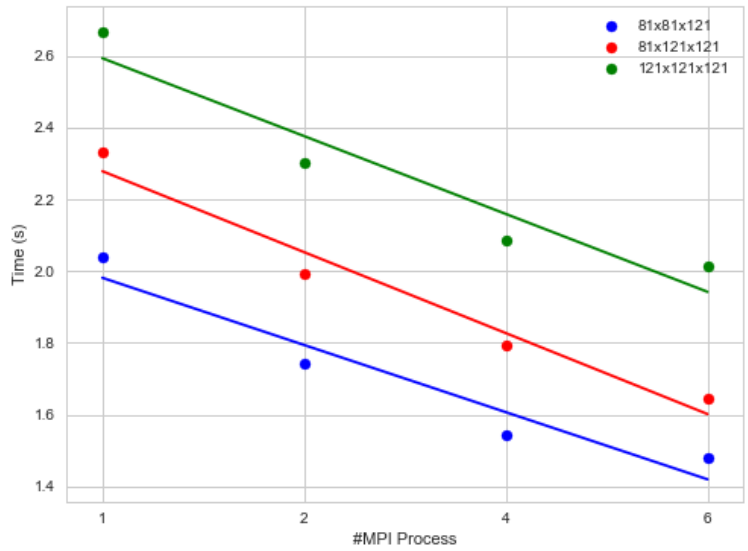
KMP_AFFINITY	Grid dimensions	npml	#MPI	#th	#Cores	$T_{LU}(s)$	$Avg.Mem_{LU}/proc(MB$
scatter	$92 \times 181 \times 321; dz = 50$	8	8	24	192	396.3693	73402
compact	$92 \times 181 \times 321; dz = 50$	8	8	24	192	435.1387	73402

Table 4.7: Homogeneous Isotropic running on Mesca node

We are also investigating the scalability en term of memory consumption of DSFDM code in LU factorization stage. Theoretically, Amestoy and al. have shown the memory complexity of the LU factorization is $O(N^4)$ [ABB+16]. These results show that MUMPS is greedy in memory consumption when the size of the problem is increasing (see Figures 4.5a and 4.6a). Basically, the memory allocation is divided between nodes. The size of processes of every classical node have two shared memories while Mesca



(a) Time Elapsed in LU factorization in performing in Mesca node



(b) Elapsed time for LU factorization in log scale performing in Mesca node

node contain eight modules of shared memories. Here, we use a number of processes to be a multiple of power two. This implies the size of the memory is divided by power two. The figure 4.5b shows the memory utilization in LU factorization in log scale of the three test case running on classical nodes. At the same time, we do the same study in Mesca node, we observe the memory consumption scales strongly when the number of process increased (see Figures 4.2b).

Moreover, the Speedup of Amdhal is also computed to study the strong scalability in Mesca node. The speedup is define to be:

$$\frac{T(1)}{T(p)}$$

where $T(1)$ and $T(p)$ are elapsed time of one and p processes. The Figure 4.7 shows the speedup computed in the three test. We can see the curves which represent the three cases are superimposed between 1 and 8 processes. Consequently, we notice that the obtained results shows the measure of speedup is approximately independent to the choice of the grid size. Thus, the speedup is a good measure of scalability.

For comparing the performance of Mesca and the classical nodes, we can use the execution time or the elapsed time in LU factorization. As the two architectures are different, we may not have a good comparison with these measures. We can can go further by considering the speedup defined by Amdhal. Here, we consider the grid size $81 \times 81 \times 121$. The computation of speedup in these two architecture (Mesca and classical nodes) gives the results shown in figures 4.8. Undoubtedly, we observe a great performance of Mesca node comparing to classical nodes.

In order to study the scalability in term of size of memory required to store the LU factors on Mesca node, we have defined series of tests depending on size of the grid. We performed simulations by multiplying the dimension of models by 40 with PML absorbing boundary conditions along the 6 sides of the model. We keep the number of process equal to the number of CPUs (8 sockets) and OpenMP* threads are binding to the physical core (Each CPUs has 24 cores). The elapsed time of the CPU in the LU factorization and the total memory consumption are given by Figures 4.9 and 4.10. In this study, the largest size of the problem on Mesca is $(n1, n2, n3) = (242, 242, 282)$ ie. 16515048 points stored in 2,7 Terabytes. This size take 3948.772 s (ie. 1h:5mn 48). We can see, the wall time during LU factorization has increased parabolically. Similarly, the memory utilization increase in the same way.

The weak scalability was studied in Mesca in comparison with classical nodes. We increased the size of the problem as function of the resources. That is to say, if one node is used to run a test size $41x41x121$ then we take 2 nodes for $41x82x121$. In Mesca, we did this study in term of module (socket + shared memory). The results shown in figures 4.11 4.12 mean we don't perfect scalability in both.

Finally, we test the performance of Mesca node by reproducing the case study of Valhall model. In [ABB+16], the authors perform the Valhall model on computer nodes that are equipped with two 2.5 GHz Intel Xeon IvyBridge E5-2670v2 processors with 10 cores per processor. The shared memory per node is 64 GB. For $66 \times 130 \times 230$, $92 \times 181 \times 321$, and $131 \times 258 \times 458$ grid sizes, they perform FFW on 240, 320 and 680 cores. The results have shown in Table 4.8, Full Rank option of MUMPS. Lacking of Valhall dataset, we reproduce this parameters of this study on Homogeneous isotropic model. We recall that the KMP_AFFINITY is scatter. In Mesca node, for $66 \times 130 \times 230$, we use the half of the number of core to get 98 seconds instead of 2 times the elapsed time of the first study. We can calculate the ratio of the time for LU factorization in Mesca over the time in Intel Xeon IvyBridge nodes. We have a ratio equal 1.78 . By using the 198 cores of Mesca we get the ratio 1.89. This fact implies the performance of Mesca node related to the nodes used in the work of [ABB+16].

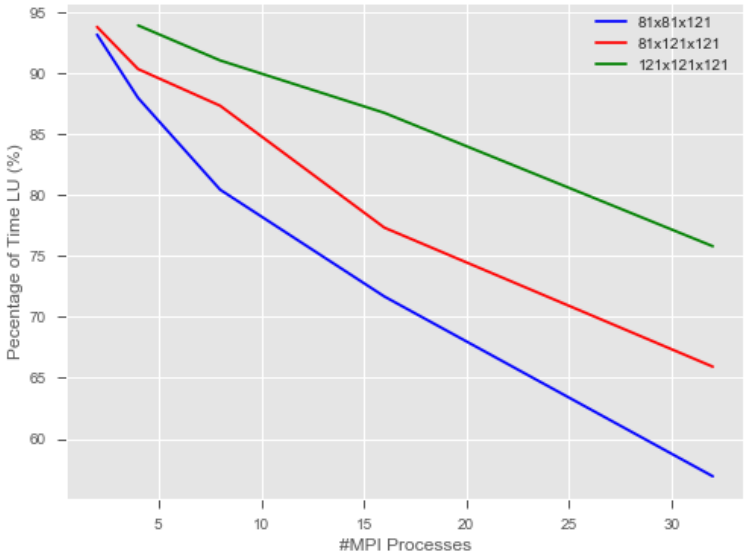
Freq	Grid dimensions	npml		#MPI	#th	#Cores	$T_{LU}(s)$
3.5, 4, 4.5, 5	$66 \times 130 \times 230; dz = 70$	8	24	10	240	78	
7	$92 \times 181 \times 321; dz = 50$	8	32	10	320	322	
10	$131 \times 258 \times 458; dz = 35$	4	68	10	680	1153	

Table 4.8: North Sea case study. Problem size and computational resources in (Amestoy et al. 2016)

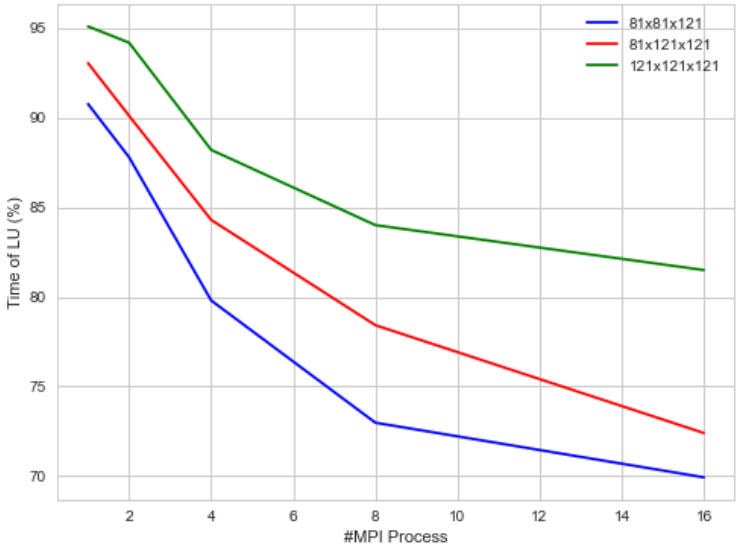
Freq	Grid dimensions	npml	#MPI	#th	#Cores	$T_{LU}(s)$	$T(s)$
3,5	$66 \times 130 \times 230; dz = 70$	8	8	15	120	98,9756	2m4.808s
7	$92 \times 181 \times 321; dz = 50$	8	8	20	160	439.0829	8m21.590s
7	$92 \times 181 \times 321; dz = 50$	8	8	24	192	396.3693	7m39.295s
10	$131 \times 258 \times 458; dz = 35$	4	8	24	192	2182.6717	38m58.811s

Table 4.9: Problem size and computational resources in Mesca node

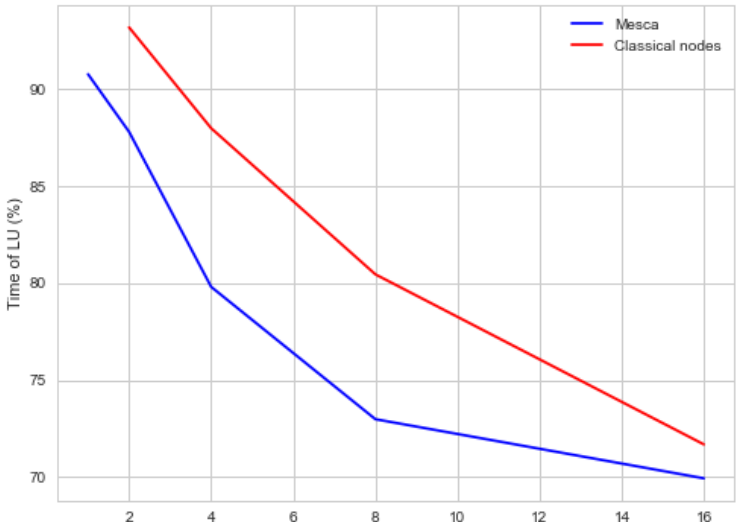
In order to optimize some DSFDM/FFWI routines, code profiler such asIntel®, *Allinea*, *BPM* (old name *MPIPROF*), *Bull IO Instrumentation*, are used to study the code. In this report, as example, we give just give the results obtained where studying MPI communication between Mesca and classical nodes with BPM (see Appendix).



(a) Percentage of Elapsed time for LU factorization over Runtime in classical nodes



(b) Percentage of Elapsed time for LU factorization over Runtime in Mesca



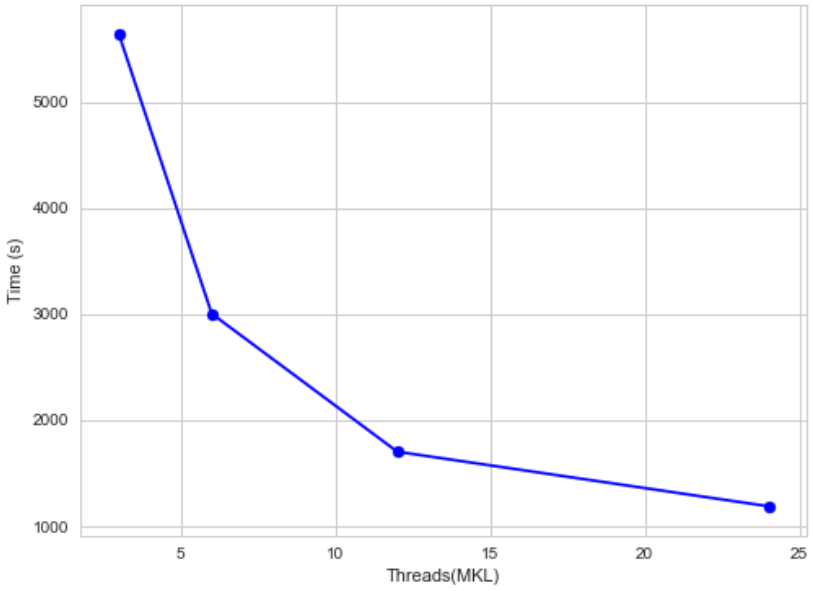
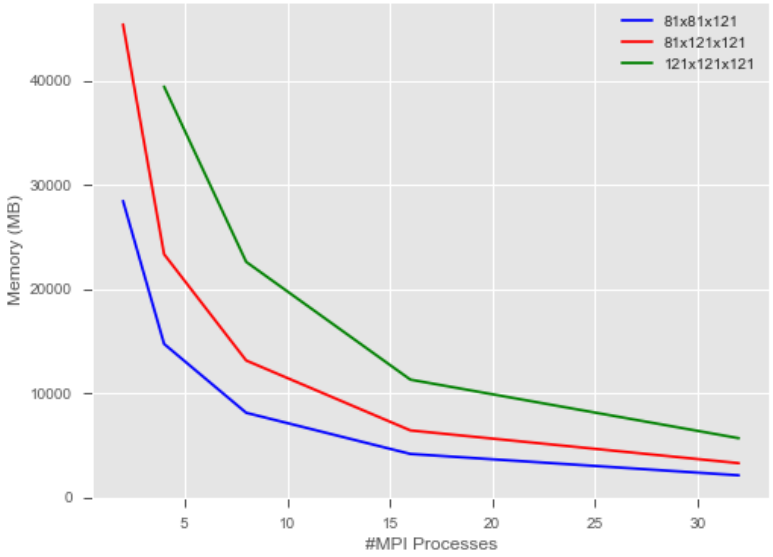
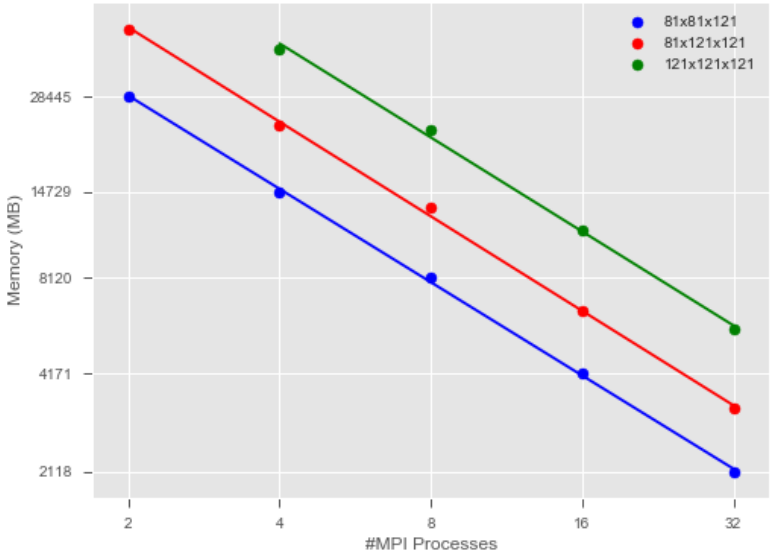


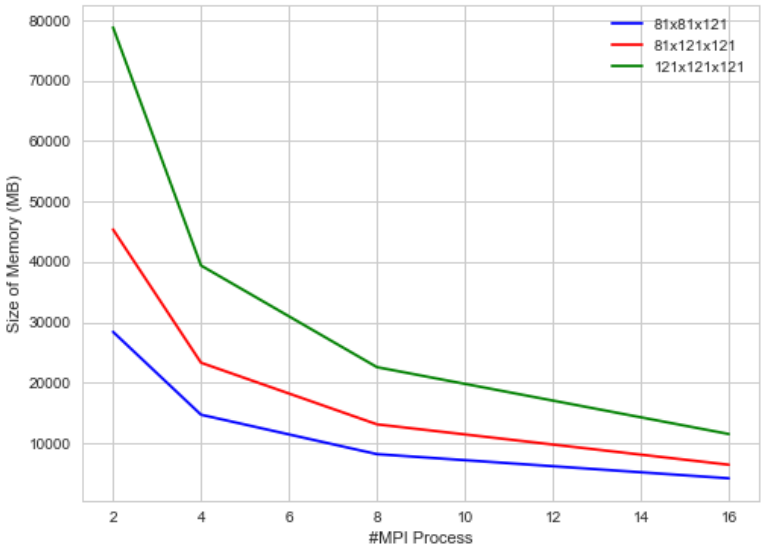
Figure 4.4: Elapsed Time for LU factorization function of number of threads in Mesca node



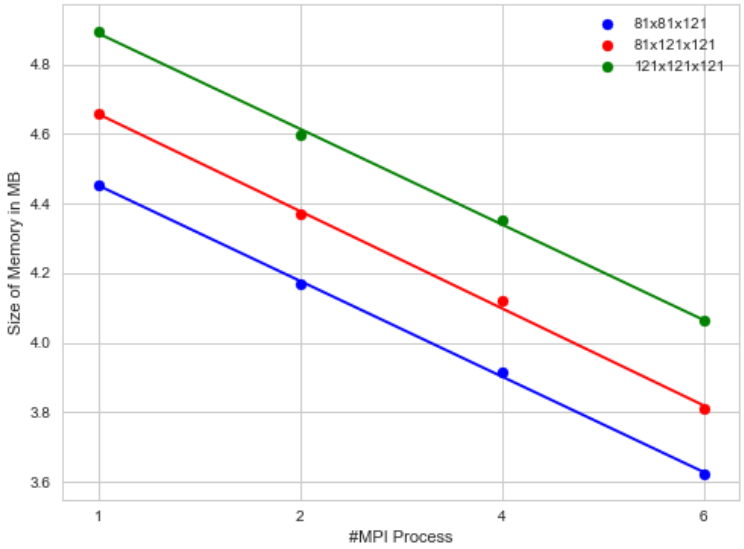
(a) Avg. Space in MBYTES per working proc for LU factorization on classical nodes



(b) Avg. Space in MBYTES per working proc for LU factorization in log scale on classical nodes



(a) Avg. Space in MBYTES per working proc in LU factorization on Mesca node



(b) Avg. Space in MBYTES per working proc in LU factorization in log scale on Mesca

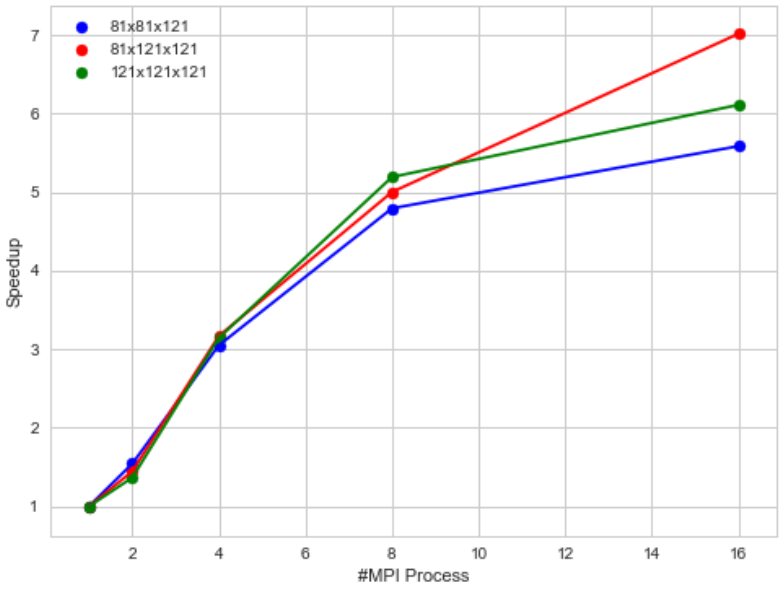


Figure 4.7: Speedup in Mesca node

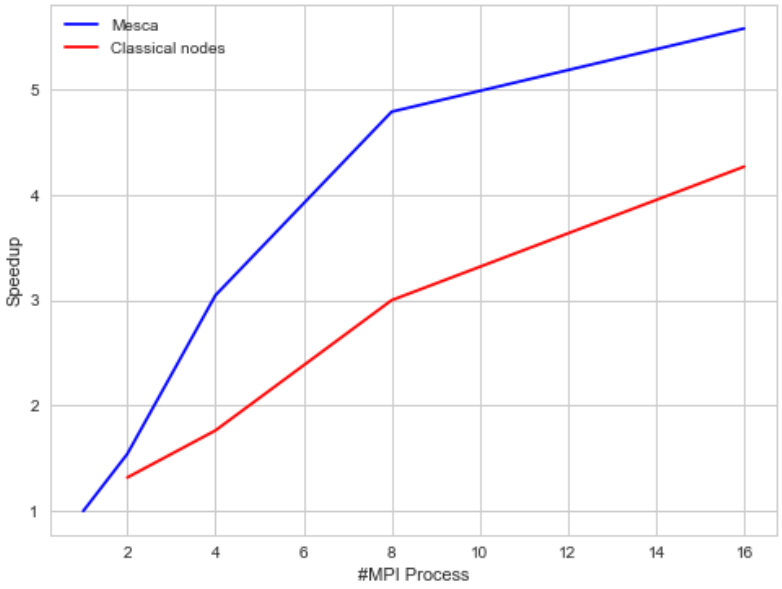


Figure 4.8: Comparison of speedup between Mesca and classical nodes

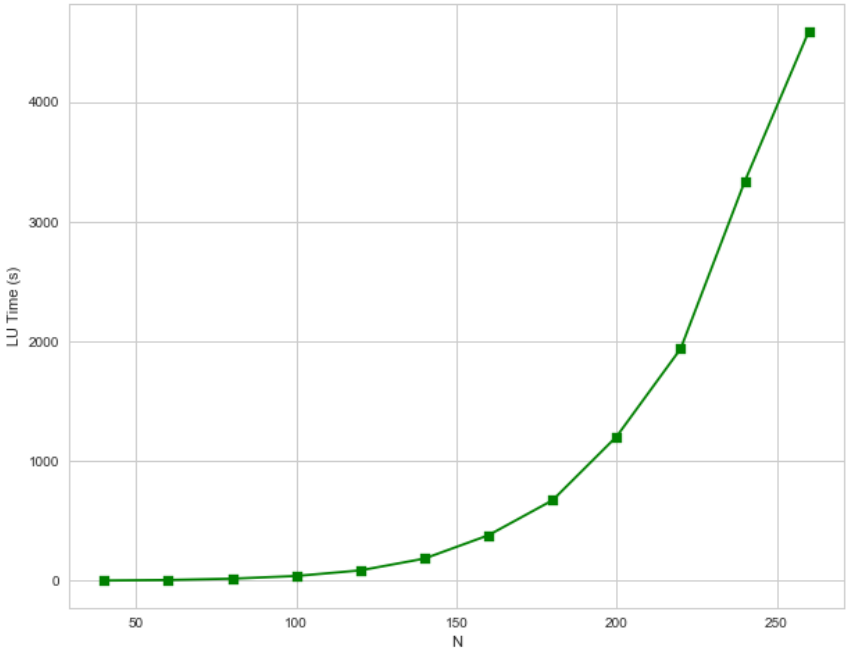


Figure 4.9: Time in LU factorization in Mesca node

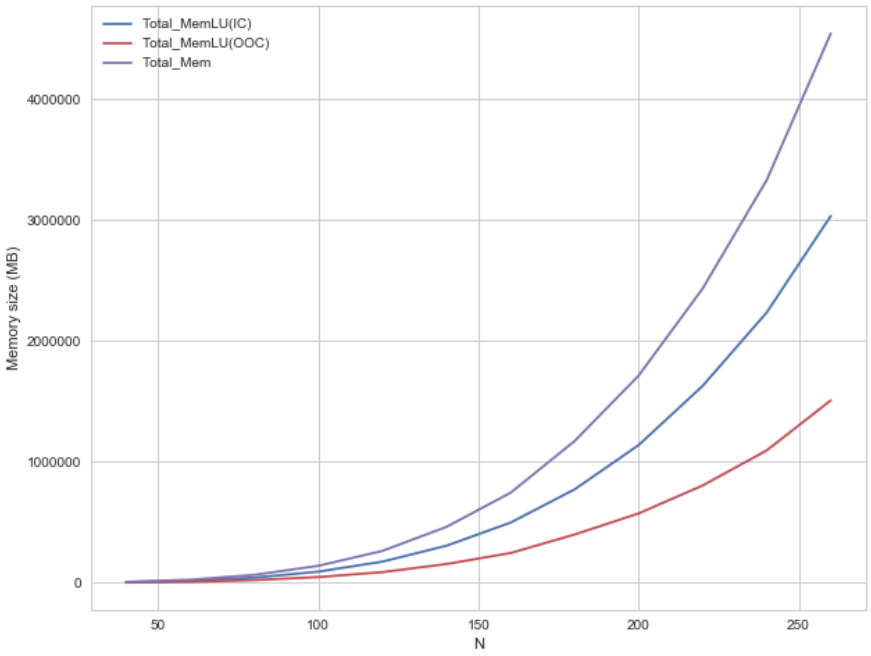


Figure 4.10: Memory consumption in LU factorization in Mesca node

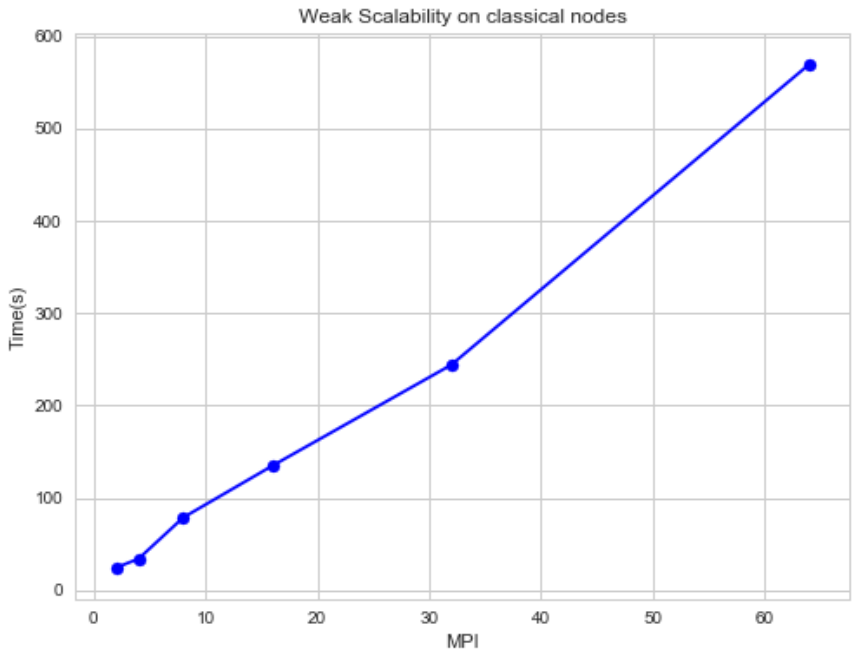


Figure 4.11: Weak scalability in classical nodes

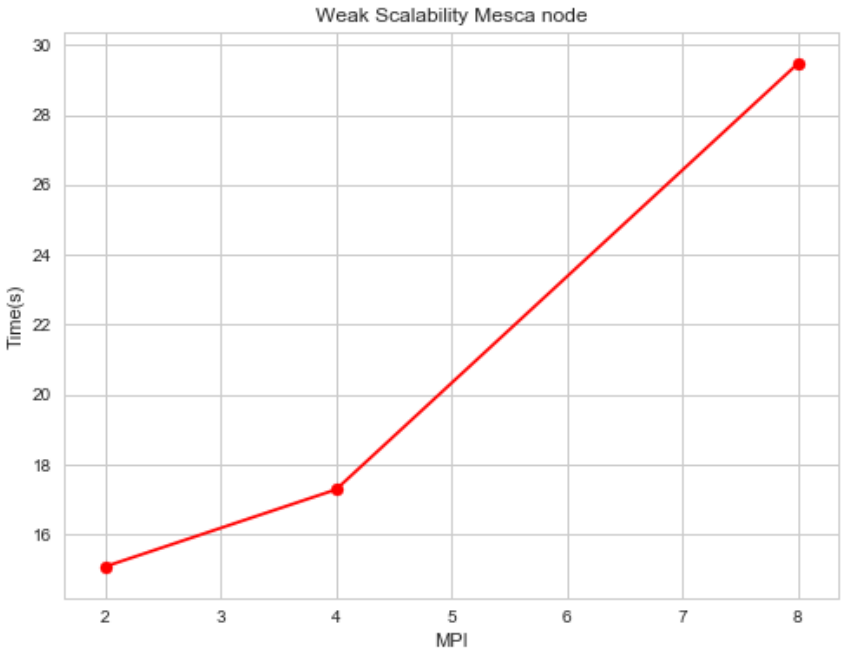


Figure 4.12: Weak scalability in Mesca nodes

5. Conclusion and Perspectives

Seismic modelling is computationally expensive, and use high performance computing to produce timely results. There are many factors that could affect performance such as functional unit utilization, load balance between nodes, contention for resources such as interconnect and memory bandwidth, synchronization delays, memory hierarchy and pipeline utilization. The aim of this work was to evaluate the capacity of DSFDM/FFWI to extend high tests case on a modern share node, MESCA-II. Moreover, we will improve the performance of the code in this architecture. To do this job, we focused on three things: firstly, installation and getting started with DSFDM/FFWI code in a distributed cluster containing the MESCA-II node; studying its performance based on strong and weak scalability, understanding and optimisation of code by using profiling tools. In this study, we have done a deep analysis of performance in two difference architectures. We have found that perfect strong scalability of DSFDM/FFWI on Mesca node. We got the limit of scalability in MESCA-II where the size equals to 260^3 with 3TB with 1h 20mn. However, the weak scalability that we studied in both architectures does not give any relevant results when we want to extend the problem size. With the profiling tools we discovered, most of the resources was consumed by the mumps routines and the time spent in these routines make up almost 90% of the total execution time. In the future work, the case of Block-Low Rank (BLR) of MUMPS will be tested with difference case of study. In perspective, to improve the performance of DSFDM/FFWI and to extend tests case in non-scare matrices, we implemented a new solver interface depending on `QR_MUMPS` (a new solver based on QR factorization). Given some constraints based on the extensibility of the code for the new solver, we have not yet obtained the expected results.

Acknowledgements

This is optional and should be at most half a page. Thanks Ma, Thanks Pa. One paragraph in normal language is the most respectful.

Do not use too much bold, any figures, or sign at the bottom.

Appendix

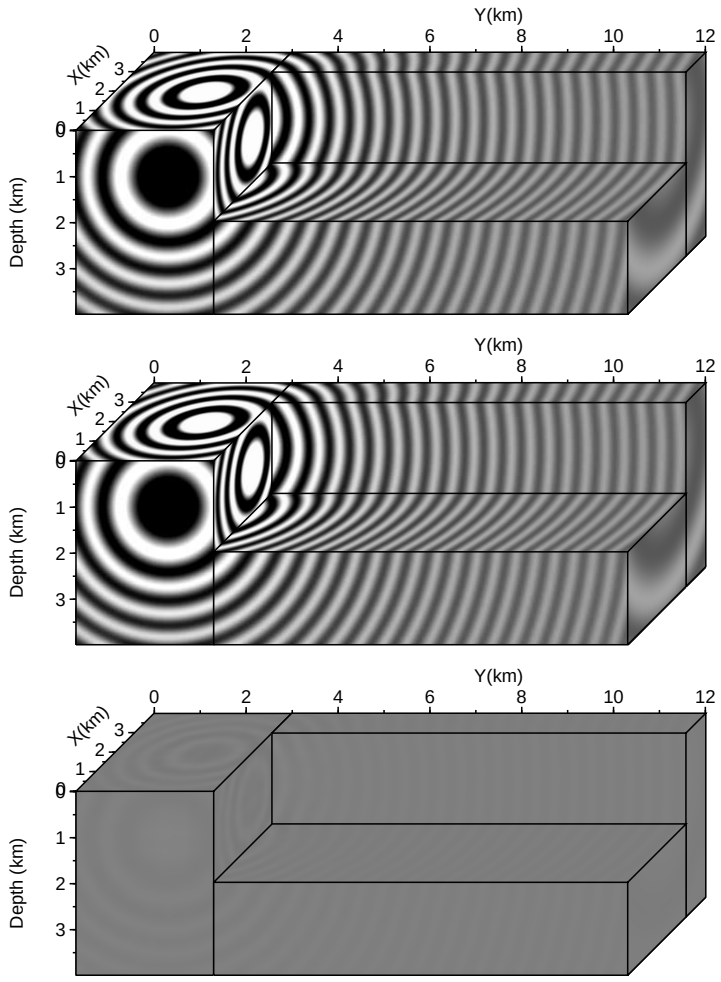


Figure 5.1: Bench Homogeneous Isotropic. Infinite homogeneous isotropic medium. Validation against analytical solution. a) DSFDM solution. b) Analytical solution. c) Difference.

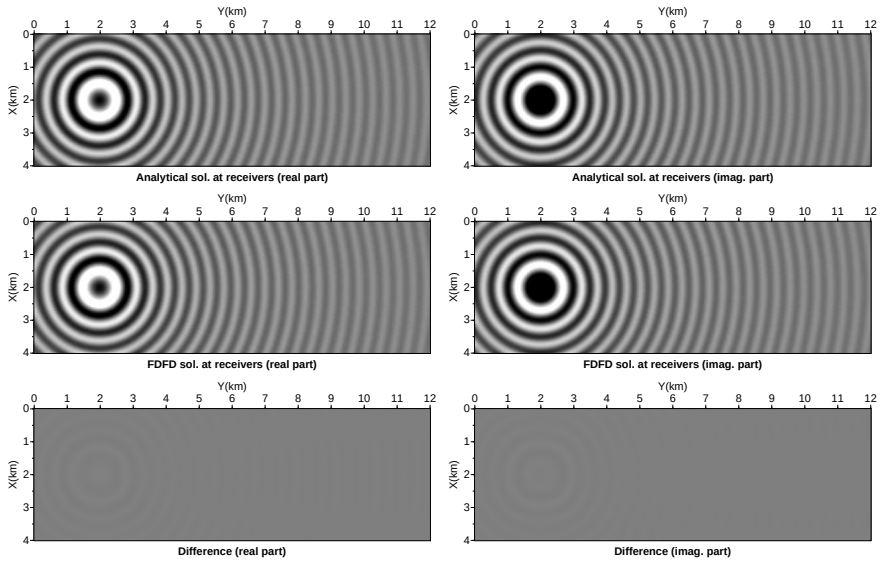


Figure 5.2: Bench Homogeneous Isotropic. Infinite homogeneous isotropic medium. Validation against analytical solution. Solution at receiver positions, 1km below the surface.

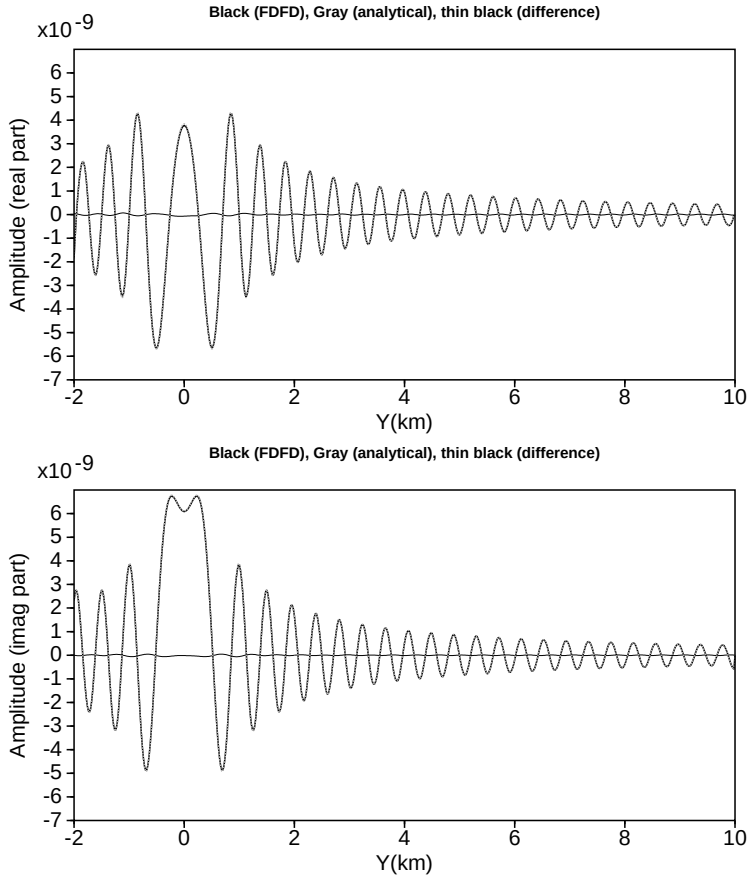


Figure 5.3: Bench Homogeneous Isotropic. Infinite homogeneous isotropic medium. Validation against analytical solution. Solution at receiver positions along the Y profile running across the shot position.

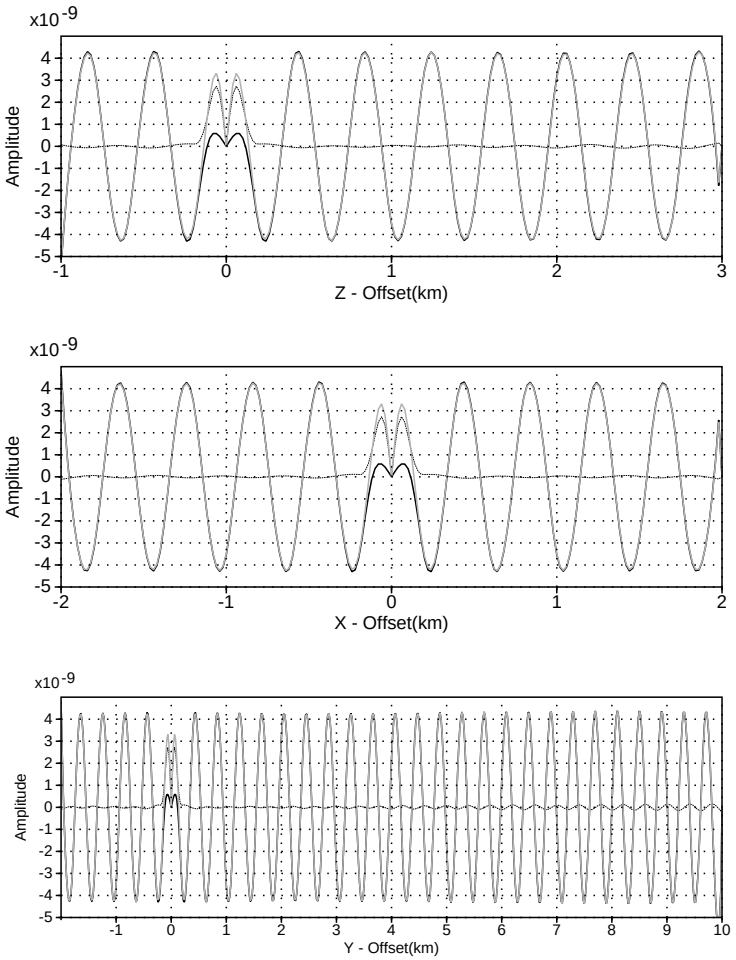


Figure 5.4: Bench Homogeneous Isotropic. Infinite homogeneous isotropic medium. Validation against analytical solution. Logs across shot position with correction for geometrical spreading.

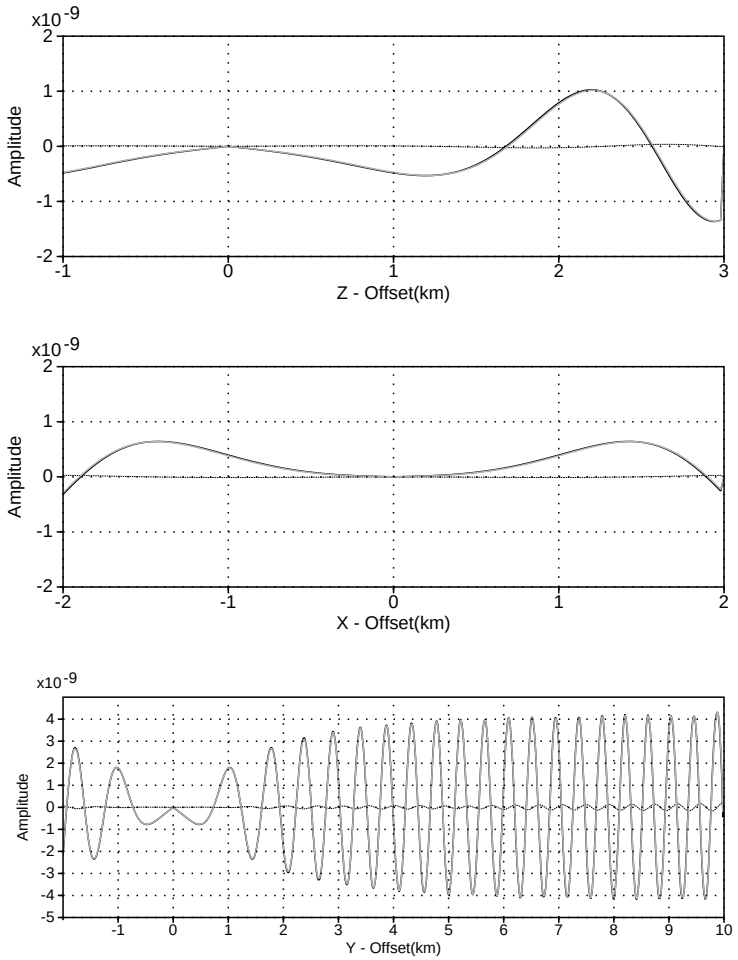
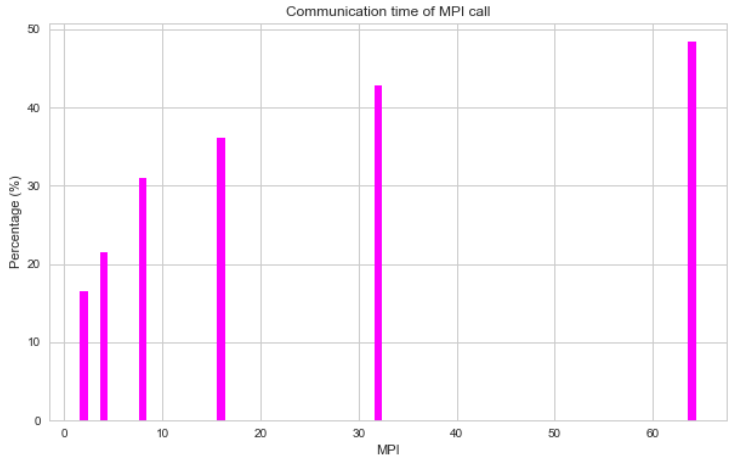
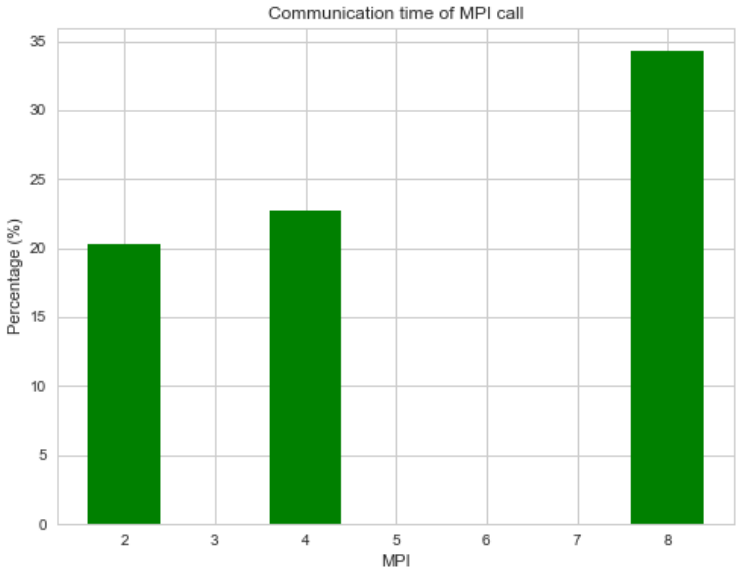


Figure 5.5: Bench Homogeneous Isotropic. Infinite homogeneous isotropic medium. Validation against analytical solution. Logs along the slices of Fig. ??b with correction for geometrical spreading.

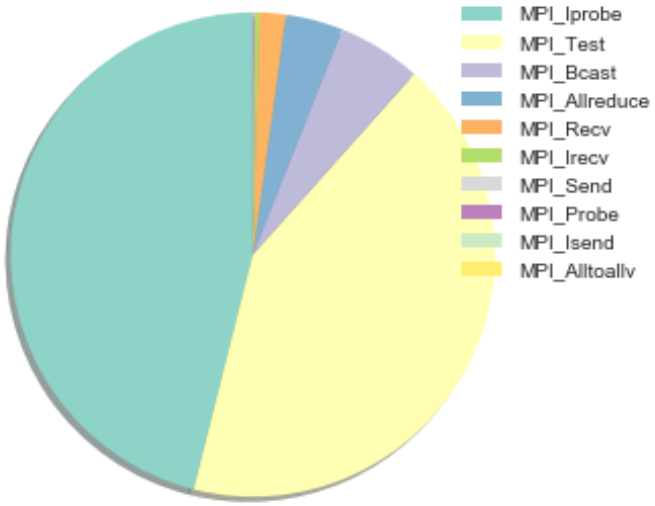


(a) Communication time of MPI call in classical nodes

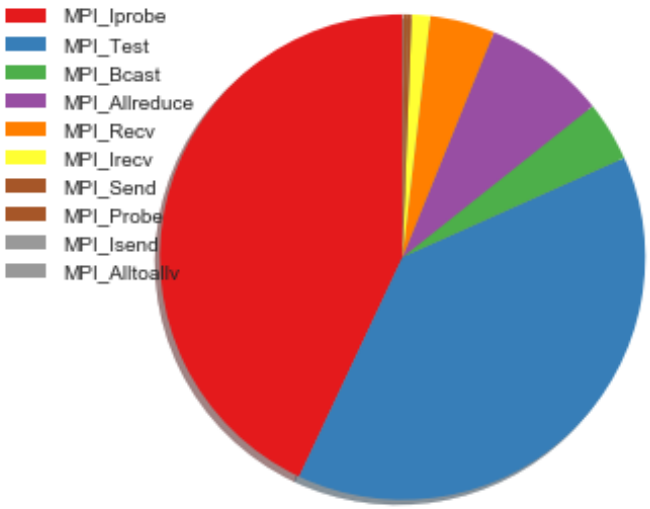


(b) Communication time of MPI call in Mesca node

Figure 5.6: Comparison of Communication time of MPI call between Mesca and classical nodes

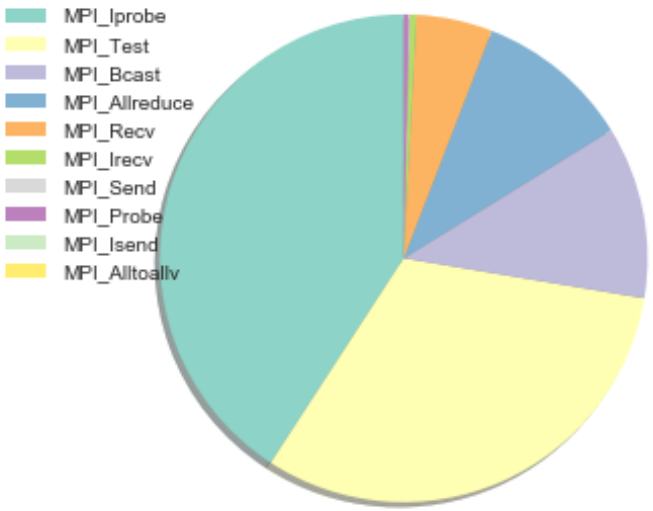


(a) MPI call in classical nodes where number of process 2

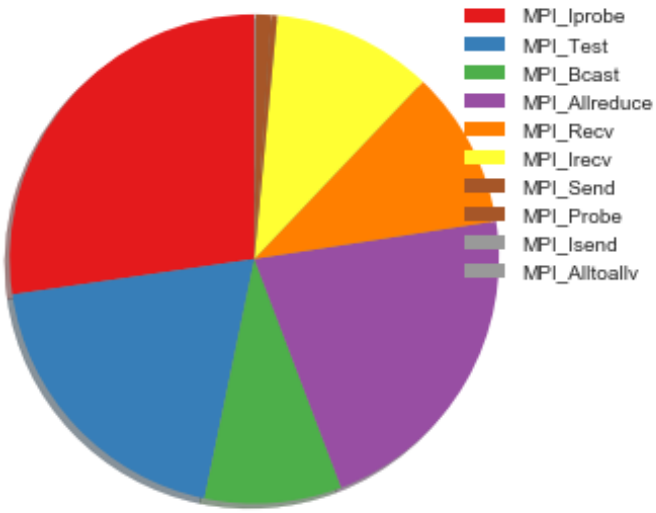


(b) MPI call in classical nodes where number of process 2

Figure 5.7: Comparison MPI call between Mesca and classical nodes where number of process 2

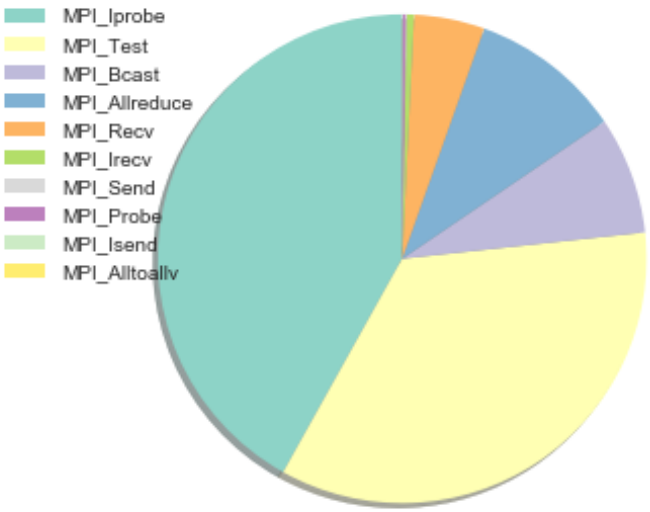


(a) MPI call in classical nodes where number of process 4

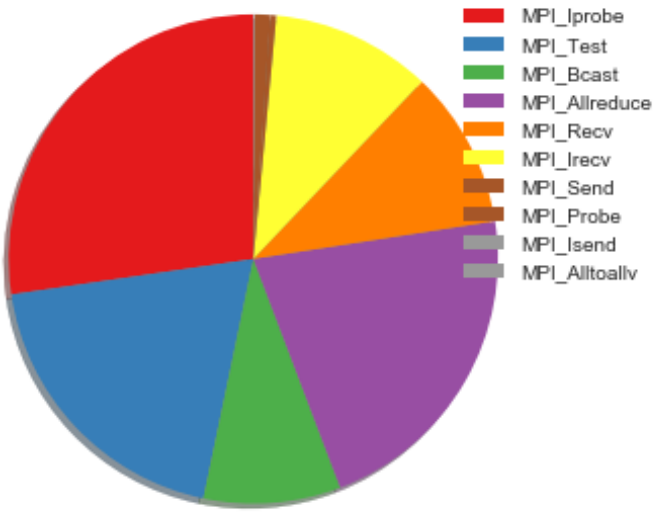


(b) MPI call in classical nodes where number of process 4

Figure 5.8: Comparison MPI call between Mesca and classical nodes where number of process 4



(a) MPI call in classical nodes where number of process 8



(b) MPI call in classical nodes where number of process 4

Figure 5.9: Comparison MPI call between Mesca and classical nodes where number of process 8

References

[AAB+15] Patrick Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker, *Improving multifrontal methods by means of block low-rank representations*, SIAM Journal on Scientific Computing **37** (2015), no. 3, A1451–A1474.

[ABB+16] Patrick Amestoy, Romain Brossier, Alfredo Buttari, Jean-Yves L'Excellent, Theo Mary, Ludovic Métivier, Alain Miniussi, and Stephane Operto, *Fast 3d frequency-domain full-waveform inversion with a parallel block low-rank multifrontal direct solver: Application to obc data from the north sea*, GEOPHYSICS **81** (2016), no. 6, R363–R383.

[Ber94] Jean-Pierre Berenger, *A perfectly matched layer for the absorption of electromagnetic waves*, Journal of Computational Physics **114** (1994), no. 2, 185 – 200.

[BESJ10] R. Brossier, V. Etienne, Operto S., and Virieux J., *Frequency-domain numerical modelling of visco-acoustic waves based on finite-difference and finite-element discontinuous galerkin methods*, ch. 7, 2010.

[BHK+10] Olav Barkved, Pete Heavey, Jan H. Kommedal, Jean-Paul van Gestel, Ruth SynnÃ¸ve, Haga Pettersen, Chris Kent, and Uwe Albertin, *Business impact of full waveform inversion at valhall*, pp. 925–929, 2010.

[BPC+14] R. Brossier, B. Pajot, L. Combe, S. Operto, Metivier L., and J. Virieux, *Time and frequency-domain fwi implementations based on time solver - analysis of computational complexities*, 76th EAGE Conference and exhibition (2014).

[BRO09] Romain BROSSIER, *Imagerie sismique à deux dimensions des milieux visco-élastiques par inversion des formes d'ondes: développements méthodologiques et applications*, Ph.D. thesis, Université de Nice-Sophia Antipolis, 2009.

[Hic02] Graham. J. Hicks, *Arbitrary source and receiver positioning in finite-difference schemes using kaiser windowed sinc functions*, GEOPHYSICS **67** (2002), no. 1, 156–165.

[HW10] Georg Hager and Gerhard Wellein, *Introduction to high performance computing for scientists and engineers*, CRC Press, 2010.

[MB16] Ludovic Métivier and Romain Brossier, *The seiscope optimization toolbox: A large-scale nonlinear optimization library based on reverse communication*, GEOPHYSICS **81** (2016), no. 2, F1–F15.

[OBC+14] S. Operto, R. Brossier, L. Combe, L. Metivier, Alessandra Ribodetti, and J. Virieux, *Computationally efficient three-dimensional acoustic finite-difference frequency-domain seismic modeling in vertical transversely isotropic media with sparse direct solver*, Geophysics **79** (2014), T257–T275.

[OMB+15] S. Operto, A. Miniussi, R. Brossier, L. Combe, L. Métivier, V. Monteiller, A. Ribodetti, and J. Virieux, *Efficient 3-d frequency-domain mono-parameter full-waveform inversion of ocean-bottom cable data: application to valhall in the visco-acoustic vertical transverse isotropic approximation*, Geophysical Journal International **202** (2015), no. 2, 1362–1391.

[OVA+07] StÃ©phane Operto, Jean Virieux, Patrick Amestoy, Jean-Yves L'Excellent, Luc Giraud, and Hafedh Ben Hadj Ali, *3d finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study*, GEOPHYSICS **72** (2007), no. 5, SM195–SM211.

[PC11] R.-É. Plessix and Q. Cao, *A parametrization study for surface seismic full waveform inversion in an acoustic vertical transversely isotropic medium*, Geophysical Journal International **185** (2011), no. 1, 539–556.

[PSH98] R. Gerhard Pratt, Changsoo Shin, and G. J. Hick, *Gauss-newton and full newton methods in frequency-space seismic waveform inversion*, Geophysical Journal International **133** (1998), no. 2, 341–362.

[Red12] P.R. Reddy, *Historical development of seismic imaging technique an overview*, Journal of the Indian Geophysical Union **16** (2012), no. 3, 71–86.

[RubXX] *Beginner's guide to high-performance computing*, Oregon State University, 2XXX.

[SYM+01] Changsoo Shin, Kwangjin Yoon, Kurt J. Marfurt, Keunpil Park, Dongwoo Yang, Harry Y. Lim, Seunghwan Chung, and Sungryul Shin, *Efficient calculation of a partial-derivative wavefield using reciprocity for seismic imaging and inversion*, GEOPHYSICS **66** (2001), no. 6, 1856–1863.

[tag11] *In praise of an introduction to parallel programming*, An Introduction to Parallel Programming (Peter S. Pacheco, ed.), Morgan Kaufmann, Boston, 2011, pp. i –.

[VO09] J. Virieux and S. Operto, *An overview of full-waveform inversion in exploration geophysics*, GEOPHYSICS **74** (2009), no. 6, WCC1–WCC26.