VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PARALLELIZATION OF ULTRASOUND SIMULATIONS USING 2D DECOMPOSITION

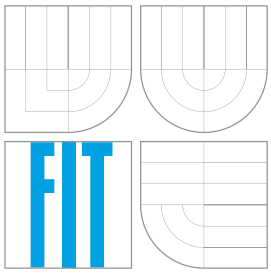
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH NIKL

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PARALELIZACE ULTRAZVUKOVÝCH SIMULACÍ POMOCÍ 2D DEKOMPOZICE

PARALLELIZATION OF ULTRASOUND SIMULATIONS USING 2D DECOMPOSITION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. VOJTĚCH NIKL
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce je součástí projektu k-Wave, což je simulační nástroj akustické tomografie sloužící k simulaci a rekonstrukci akustických vlnových polí a jeho hlavním přínosem je plánování ultrazvukových operací lidské tkáně, např. nádoru na mozku. Dopředné simulace jsou založeny na řešení k-prostorové pseudospektrální časové domény.

Simulace jsou časově a výpočetně velice náročné, není výjimkou, že jedna simulace trvá několik desítek hodin na moderním výpočetním clusteru se stovkami jader. Simulace probíhají na 3D maticích, které reprezentují určité vlastnosti reálné tkáně, např. hustotu absorbce nebo rychlost šíření zvuku. K výpočtu gradientu se používá Rychlá Fourierova transformace (dále jen FFT), jejíž výpočet zabere zhruba 60% simulačního času. 3D FFT byla do této doby počítána pomocí softwarové knihovny FFTW, která interně využívá 1D dekompozici, tj. dekompozici podél jedné osy. Hlavní nevýhoda 1D dekompozice je relativně malý maximální počet výpočetních jader, přes které lze paralelizovat výpočet. Matice mají velikost řádově $1024 \times 2048 \times 3072$, tím pádem lze efektivně paralelizovat přes maximálně 1024 jader. Dnešní superpočítače umožnují využít až stovky tisíc jader a tomu bychom se rádi přiblížili. Řešením je využití 2D dekompozice, která by teoretický maximální počet jader posunula až do řádu milionů. Její efektivní implementací se zabývá právě tato práce.

2D dekompozice je obecně paralelizována pouze pomocí MPI procesů, např. v knihovnách PFFT nebo P3DFFT, v této práci ale využíváme pokročilejší kombinace MPI procesů a OpenMP vláken, kterou jsme nazvali hybridní 2D dekompozice (HybridFFT). Má tři hlavní části: výpočet 1D FFTs, lokální transpozice dat a globální transpozice dat. Pro výpočet sérií 1D FFT je využita knihovna FFTW.Lokální transpozice jsou implementovány pomocí blokové transpozice 2D matice, která je vektorizována pomocí SSE/AVX instrukcí. Jak 1D FFT, tak lokální lokální transpozice, jsou akcelerovány pomocí OpenMP vláken. Globální transpozice je opět implementována prostřednictvím knihovny FFTW, která při použití pokročilého plánování dokáže výrazně snížit dobu potřebnou pro její realizaci. Hlavním cílem této práce je tedy dosáhnout maximálního možného zrychlení a škálovatelnosti oproti předchozímu řešení, zároveň ale i zachovat kompatibilitu a přenositelnost. Hybridní transformace pracuje nejlépe, pokud na jednom socketu spustíme jeden MPI proces a v rámci tohoto socketu využijeme tolik vláken, kolik máme k dispozici jader. Díky tomu nemusí jádra v rámci jednoho socketu komunikovat přes MPI zprávy, ale využívají rychlejší sdílenou paměť, a zároveň je MPI komunikace efektivnější, protože máme pouze jeden MPI proces na socket a tím pádem jsou MPI zprávy vetší a je jich méně, což vede k menšímu zahlcení propojovací sítě a lepší efektivitě komunikace.

Řešení bylo testováno na superpočítačích Anselm (Ostrava), Zapat (Brno) a Supernova (Wroclaw). Pro nižší počty jader, v řádu několika set, je výkon přibližně stejný nebo o pár procent lepší, než původně použitá 1D dekompozice FFTW knihovny nebo knihony PFFT a P3DFFT. Jeden z velmi dobrých výsledků je např. 512^3 FFT na 512 jádrech, kde hybridní dekompozice dosáhla času 31 ms, zatímco FFTW 39 ms a PFFT 44ms. Na stroji Anselm jsme spustili výpočet až na 2048 jádrech a škálovatelnost byla stále lineární. Nejlepší nárust výkonu oproti ostatním knihovnám by se měl projevit při počtu zhruba 8–16 tisíc jader pro kostky velikosti 1024^3, protože v této konfiguraci bude mít jeden MPI proces na starosti jednu desku matice a zároveň budou MPI zprávy dostatečně velké a v takovém počtu, že by se měla projevit lepší efektivita komunikace oproti ostatním knihovnám. Bohužel zatím nemáme přístup na superpočítač, který by měl k dispozici takovéto prostředky.

Testování korektnosti výpočtu bylo testováno pomocí porovnání výsledků s výsledky

v Matlabu a vypočtení maximální odchylky, která se v absolutních hodnotách pohybovala okolo 10^{-5} a méně.

Abstract

This thesis is a part of the k-Wave project, which is a toolbox for the simulation and reconstruction of acoustic wave felds and one of its main contributions is the planning of focused ultrasound surgeries (HIFU). One simulation can take tens of hours and about 60% of the simulation time is taken by the calculation of the 3D Fast Fourier transforms. Up until now the 3D FFT has been calculated purely by the FFTW library and its 1D decomposition, whose major limitation is the maximum number of employable cores. Therefore we introduce a new approach, called the 2D hybrid decomposition of the 3D FFT (HybridFFT), where we combine both MPI processes and OpenMP threads to reach as best performance as possible.

On a low number of cores, on the order of a few hundreds, we are about as fast or slightly faster than FFTW and pure MPI 2D decomposition libraries (PFFT and P3DFFT). One of the best results was achieved on a 512^3 FFT using 512 cores, where our hybrid version run 31ms, FFTW run 39ms and PFFT run 44ms. The most significant performance advantage should be seen when employing around 8–16 thousand cores, however we haven't had an access to a machine with such resources. Almost a linear scalability has been proven for up to 2048 employed cores.

Klíčová slova

k-Wave, FFT, Rychlá Fourierova transformace, transpozice, dekompozice, optimalizace, cluster, superpočítač, MPI, OpenMPI, OpenMP, vlákna, FFTW, SSE, AVX

Keywords

k-Wave, FFT, fast Fourier transform, transposition, decomposition, optimization, cluster, supercomputer, MPI, OpenMPI, OpenMP, threads, FFTW, SSE, AVX

Citace

Vojtěch Nikl: Parallelization of Ultrasound Simulations Using 2D Decomposition, diplomová práce, Brno, FIT VUT v Brně, 2014

Parallelization of Ultrasound Simulations Using 2D Decomposition

Declaration

I declare that this thesis has been written and developed independently under the supervision of dr. Jiří Jaroš. I have ensured that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

.....
Vojtěch Nikl
June 4, 2014

Acknowledgment

I would like to express the deepest appreciation to my supervisor, dr. Jiří Jaroš, who has the attitude and the substance of a genious: he continually and convincingly conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regard to teaching. He introduced me to the world of high-performance computing in a way that I will never forget. He has been available and ready to answer any of my questions and to help me whenever I needed to. He has also been very patient, friendly, focused and kind. It has always astonished me how much of his work and even free time is dedicated to his students. Without his guidance and persistent help this thesis would not have been possible.

I acknowledge and thank very much the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033), for providing the resources and the core hours of its Anselm cluster located in Ostrava, Czech Republic.

I also acknowledge PRACE for awarding us access to resource Supernova based in Poland at Wroclaw.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme „Projects of Large Infrastructure for Research, Development, and Innovations“ (LM2010005), is greatly appreciated.

Access to the CERIT-SC computing and storage facilities provided under the programme Center CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, reg. no. CZ. 1.05/3.2.00/08.0144, is also greatly appreciated.

I dedicate this thesis to my parents and grandparents who have always been so close to me and have supported me throughout my whole life. It is their unconditional love that

motivates me to set higher targets. I also dedicate this thesis to my only sibling Patrick who has provided me with a strong love shield that always surrounds me and never lets any sadness enter inside.

Contents

Chapter 1

Introduction

According to the Czech Society of Oncology, more than 73,000 tumour diseases are newly diagnosed in the Czech Republic every year and this number is continuing to grow. This contributes to cancer being the cause of nearly 1 in 3 deaths every year. Unfortunately, current cancer treatment procedures including external beam radiation therapy (EBRT), chemotherapy and surgical interventions have severe limitations and side effects (radiation and drug dosage limits, operability, repeatability, long-lasting consequences) that reduce the chances of successful cure [21].

A very promising alternative to the standard treatment procedures is high intensity focused ultrasound (HIFU), also known as focused ultrasound surgery [23]. The technique works by sending a focused beam of ultrasound into the tissue, typically using a large transducer. At the focus, the acoustic energy is sufficient to cause cell death in a localised region while the surrounding tissue is left unharmed. One of the major challenges is the computational scale. This arises because the treatment area is often very large, usually on the order of $20 \times 20 \times 20$ cm. This usually requires 10^9–10^{12} computational grid points, making many simulations intractable. Therefore, new approaches are needed to allow accurate large-scale ultrasound simulations using more economical computational resources. One of very recent ones is the k-Wave project.

The k-Wave project-toolbox (see Chapter 2) is a toolbox for the simulation and reconstruction of acoustic wave fields, whose simulations require large amounts of performance and memory. Simulations run on 3D matrices of real or complex values, which represent specific properties of a certain part of a real tissue. These properties are for example density absorption or the speed of sound. The bigger the matrix is, the more detailed and accurate the simulation results are, however the performance and memory demands rise rapidly. One of the advantages of the k-Wave simulation lies in the fact that its computational grids are homogeneous in terms of calculation. This means that equally big sections are calculated equally long and therefore it is much easier to evenly distribute the work among compute cores.

The core of the simulation algorithm is the fast Fourier transform (see Chapter 3). This task requires to decompose the matrix, because the simulation is run on a distributed cluster where ~2TB of working dataset is the standard size (see Chapter 8). Up until now k-Wave has used the 1D decomposition (see Chapter 8), whose most significant limitation is the maximum number of cores running in parallel. For a $N \times N \times N$ matrix we can employ only up to N cores, which is a major limitation in the case where the value of N is in the thousands and today's large supercomputers, such as german SuperMUC [9], allow parallelizing over hundreds of thousands of cores. One simulation may take up to several

days, so the speedup potential definitelly needs to be explored. In k-Wave the domain sizes of a matrix can be up to $3072 \times 2048 \times 1024$, which means we can effectively parallelize only over 1024 cores. Even a small simulation of size $512 \times 512 \times 512$ takes about 2 hours to calculate using 512 cores. If we could employ 8192 cores or more, we could run almost an interactive simulation. Another reason, why we need to employ more cores is a decreasing amount of RAM per core of newer clusters. Today's clusters have ~1–2GB of RAM per core, which makes it more and more difficult to fit the whole simulation into RAM.

The solution to this problem is to decompose the matrix across two dimensions instead of just one. This approach is expectedly called the 2D decomposition (see Chapter 8). Its main advantage is that it can employ up to $N \times N$ cores simultaneously, so we could theoretically reach the *exascale* scaling - the parallelization on the order of over a million computing units. The CRESTA project [2] is currently working on delivering an exaflop by the end of this decade. The down side is the need for two nonlocal memory operations instead of one, requiring expensive interprocess communication.

Chapter 2

k-Wave toolbox

k-Wave [35, 34] is an open source acoustics toolbox for MATLAB and C++ developed by Bradley Treeby and Ben Cox (University College London) and Jiří Jaroš (Brno University of Technology). The software is designed for time domain acoustic and ultrasound simulations in complex and tissue-realistic media. The simulation functions are based on the k-space pseudospectral time domain solution [37] to coupled first-order acoustic equations for homogeneous or heterogeneous media in one, two, and three dimensions. This method is used for the reconstruction of the photoacoustic image, HIFU treatment planning etc.

Photoacoustic tomography (PAT) is a noninvasive biomedical imaging modality that allows the *in vivo* visualization of embedded light absorbing structures [39]. The technique works by externally illuminating a tissue sample with short pulses of visible or near-infrared (NIR) laser light. The localized absorption of this light particularly by the hemoglobin chromophores present in blood produces broadband ultrasonic waves via thermoelastic expansion. By measuring the ultrasonic waves that propagate back to the tissue surface, images of the initial photoacoustic pressure which is related to the absorbed optical energy distribution can then be reconstructed. These images may be used to quantify tissue properties [25, 15], or to identify pathological structures [29]. The technique has been demonstrated via high-resolution *in vivo* imaging of vasculature in both small animals [40, 24] and humans [41]. Similar images may also be formed using microwave frequencies (an analogous technique often called thermoacoustic tomography), where water is the primary absorber [38].

For a representative simulation, around 60% of the total computation time is spent performing the forward and inverse FFT [37]. FFT is used to calculate spatial gradients, that implies that the wave field is periodic in Fourier pseudospectral and k-space numerical models [36]. Depending on the complexity of the simulation, up to fourteen FFTs are calculated for each time step. There are generally 20–50 thousands time steps in one simulation [37].

But what is actually calculated by k-Wave from the mathematical point of view? When an acoustic wave passes through a compressible medium, there are dynamic fluctuations in the pressure, density, temperature, particle velocity, etc. These changes can be described by a series of coupled first-order partial diferential equations based on the conservation of mass, momentum, and energy within the medium. Often in acoustics, these equations are combined together into a single „wave equation“ which is a second-order partial diferential equation in a single acoustic variable (most often the acoustic pressure) [36].

In many situations in biomedical ultrasonics, the magnitude of the acoustic waves is high enough that the wave propagation is no longer linear. In this case, additional nonlinear

terms also need to be included in the governing equations. k-Wave doesn't model all the possible nonlinear effects that might occur in a fluid; it is not a computational fluid dynamics (CFD) solver. Instead, it currently includes two additional nonlinear terms that account for cumulative nonlinear effects to second-order in the acoustic variables. This is an accurate model for many situations in biomedical ultrasound [36]. The system of coupled first-order equations solved by k-Wave becomes

$$\frac{\partial u}{\partial t} = -\frac{1}{\rho_0}\nabla p \tag{2.1}$$

Momentum conservation $(1 \times FFT, 3 \times iFFT)$

$$\frac{\partial \rho}{\partial t} = -(2\rho + \rho_0)\nabla \cdot u \tag{2.2}$$

Mass conservation $(3 \times FFT, 3 \times iFFT)$

$$p = c_0^2(\rho + \frac{B}{2A}\frac{\rho^2}{\rho_0} - L_\rho) \tag{2.3}$$

Pressure-density relation

Here $\boldsymbol{u}$ is the acoustic particle velocity, $\boldsymbol{p}$ is the acoustic pressure, ρ is the acoustic density, ρ_0 is ambient (or equilibrium) density, and c_0 is the isentropic sound speed and $\boldsymbol{d}$ is the acoustic particle displacement. These equations assume the background medium is quiescent (meaning there is no net flow and the other ambient parameters don't change with time) and isotropic (meaning the material parameters do not depend on the direction the wave is travelling) [36].

The operator used in k-Wave has two terms both dependent on a fractional Laplacian and is given by

$$L = \tau\frac{\partial}{\partial t}(-\nabla^2)^{\frac{y}{2}-1} + \eta(-\nabla^2)^{\frac{y+1}{2}-1}. \tag{2.4}$$

Absorption term $(2 \times FFT, 2 \times iFFT)$

An example of one HIFU treatment planning in kidney is shown in Figure 2.1. The image represents the maximal acoustic pressure of the tissue in the domain and the red parts are focal points.

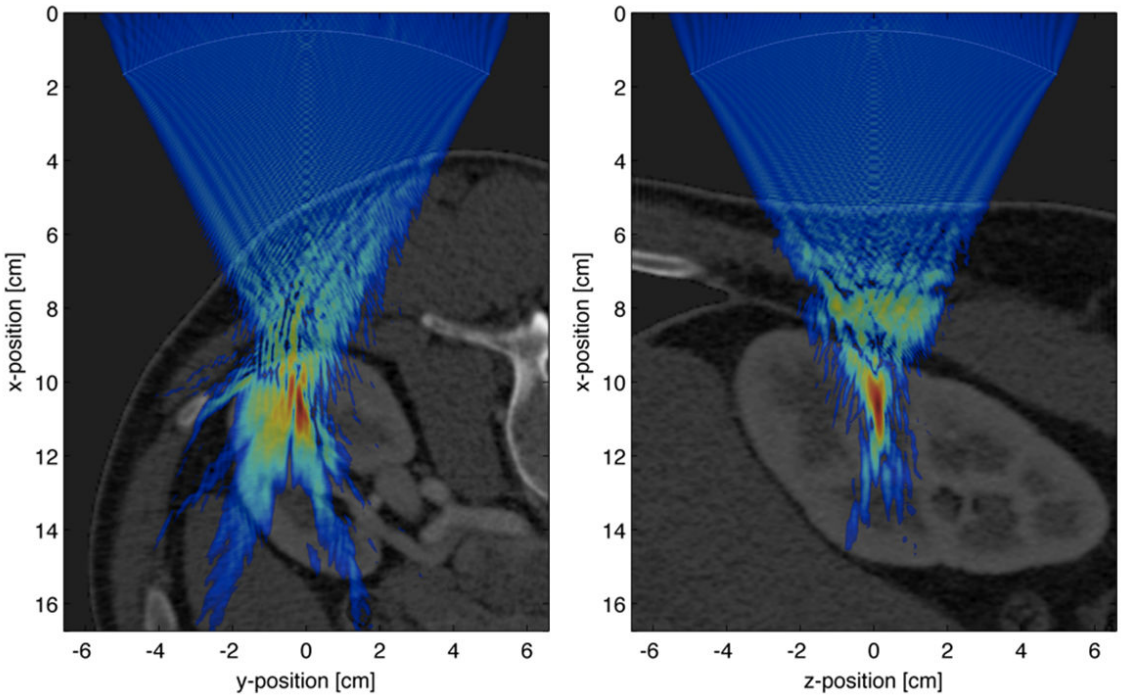


Figure 2.1: HIFU treatment planning in kidney.

Chapter 3

Fast Fourier Transform

The fast Fourier transform (FFT) [14] is an effective algorithm to compute the discrete Fourier transform (DFT) and its inverse. The Fourier analysis converts time (or space) to frequency and vice versa. The FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result, fast Fourier transforms are widely used for many applications in engineering, science, and mathematics, where it is used for digital signal processing, partial differential equations or algorithms for fast multiplication of large integers. The basic ideas were popularized in 1965, but some FFTs had been previously known as early as 1805. Fast Fourier transforms have been described as „the most important numerical algorithm of our lifetime“.

The DFT is obtained by decomposing a sequence of values into components of different frequencies, however computing it directly from the definition is often too slow to be practical, because the time complexity of computing a DFT of N values is $O(N^2)$. Therefore, a FFT is often used to compute a DFT instead, because its time complexity is $O(NlogN)$ while giving the same result. The difference in speed can be very significant for data where N is a big number on the order of thousands or more.

Consider a sequence of N complex numbers $x_0, \ldots, x_{N-1}$. This sequence is transformed into another sequence of N complex numbers $X_0, \ldots, X_{N-1}$, according to the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \qquad k = 0, \ldots, N-1. \tag{3.1}$$

An inverse DFT is calculated according to a very similar formula where the only difference is the sign in the exponent and the normalizations factor.

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} nk} \qquad n = 0, \ldots, N-1 \tag{3.2}$$

The sequence X_k represents the amplitude and phase of each sinusoidal component of the input signal x_n. The DFT calculates the X_k sequence from the sequence x_n, while the inverse DFT calculates x_n as a sum of the sinusoidal components having frequencies k/N cycles per sample. In other words, X_k is a sequence of time coefficients and x_n is a sequence of frequency images X_k.

3.1 Cooley-Tukey algorithm

The Cooley-Tukey algorithm [14] is the most common FFT algorithm, especially its *Radix-2* variant. It is named after J. W. Cooley and John Tukey. Radix-2 means that the number of processed elements must be equal to a non-negative integer of the power of two. If not, we can fill in zeros at the end of the sequence and discard these elements from the result. The algorithm has two variants, *decimation in time* (DIT) and *decimation in the frequency domain* (DIF).

DIT splits the DFT into two interleaved DFTs of size $N/2$ with each recursive step. That means we firstly calculate the DFT of elements on the even indices $x_{2m}(x_0, x_2, \ldots, x_{N-2})$ and on the odd indices $x_{2m+1}(x_1, x_3, \ldots, x_{N-1})$ and these subresults form together the final result. Mathematically

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}. \tag{3.3}$$

We can recursively apply this splitting step on every new even and odd sequence which results in the $O(NlogN)$ time complexity.

Other algorithms that are used to calculate the FFT are, among others, *Split Radix FFT*, *Prime-factor FFT* or *Rader's FFT* [14]. All variants of FFT algorithms have one common feature: the number of elements of the input can't be just any number. Most often, this must be the power of two or a multiple of two prime numbers.

3.2 N-dimensional transform

If we want to calculate the FFT of a N-dimensional matrix, one way, which we implement in this project, is to do a series of 1D FFTs along each axis.

- Series of 1D FFTs along the first axis.
- Series of 1D FFTs along the second axis.
- ...
- Series of 1D FFTs along the n-th axis.

3.3 Real-to-Complex and Complex-to-Real transforms

Since all the acoustic quantities in k-Wave are in the real domain, the spectrum is symmetrical. This saves nearly 50% of memory requested by FFT and also reduces the size of dependent matrices in the frequency domain.

Real-to-Complex (R2C) FFT takes a 3D matrix of real values (single precision floats in the case of k-Wave) as an input and produces a 3D matrix of complex values. Therefore the FFTs of the series along the first axis have to calculate Real-to-Complex transforms and FFTs along other axis calculate Complex-to-Complex transforms.

- Series of 1D R2C FFTs along the first axis.
- Series of 1D C2C FFTs along the second axis.
- ...

- Series of 1D C2C FFTs along the n-th axis.

Very similar case is the backward Complex-to-Real inverse FFT, where the last 1D FFT along the n-th axis is Complex-to-Real and all other 1D FFTs are Complex-to-Complex.

- Series of 1D C2C iFFTs along the first axis.
- Series of 1D C2C iFFTs along the second axis.
- ...
- Series of 1D C2R iFFTs along the n-th axis.

3.4 Software libraries implementing FFT

The FFTW software library

One of the implementations of the FFT, used in k-Wave, is the FFTW software library [17]. FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data. Benchmarks [18], performed on on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software, and is even competitive with vendor-tuned codes. In contrast to vendor-tuned codes, however, FFTW's performance is portable: the same program will perform well on most architectures without modification. FFT performance comparison on a 3.0 GHz Intel Core Duo, Intel compiler and 64-bit mode is shown in Figure 3.1 and 3.2.

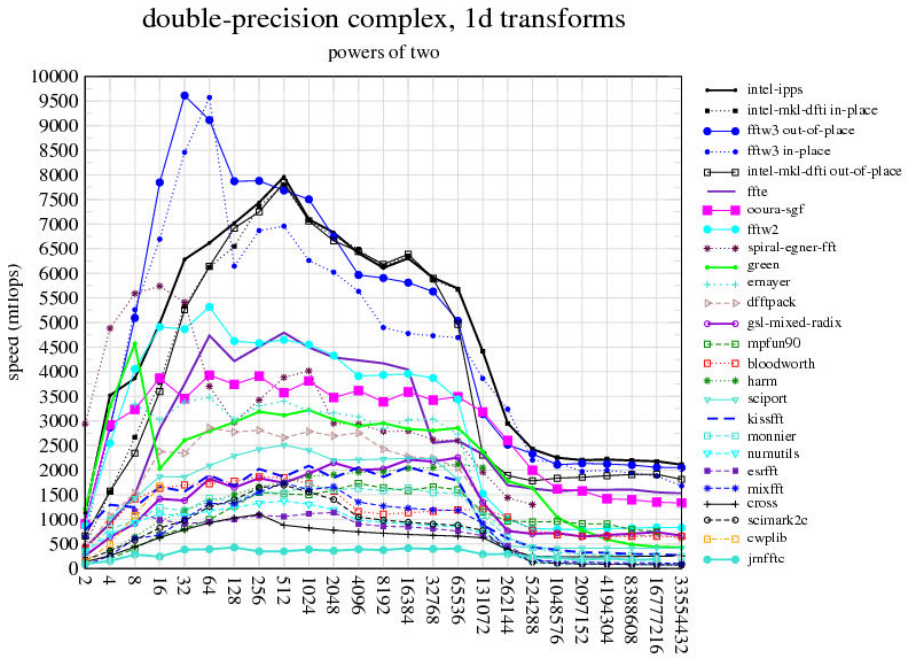


Figure 3.1: FFT benchmark (powers of two) [18].

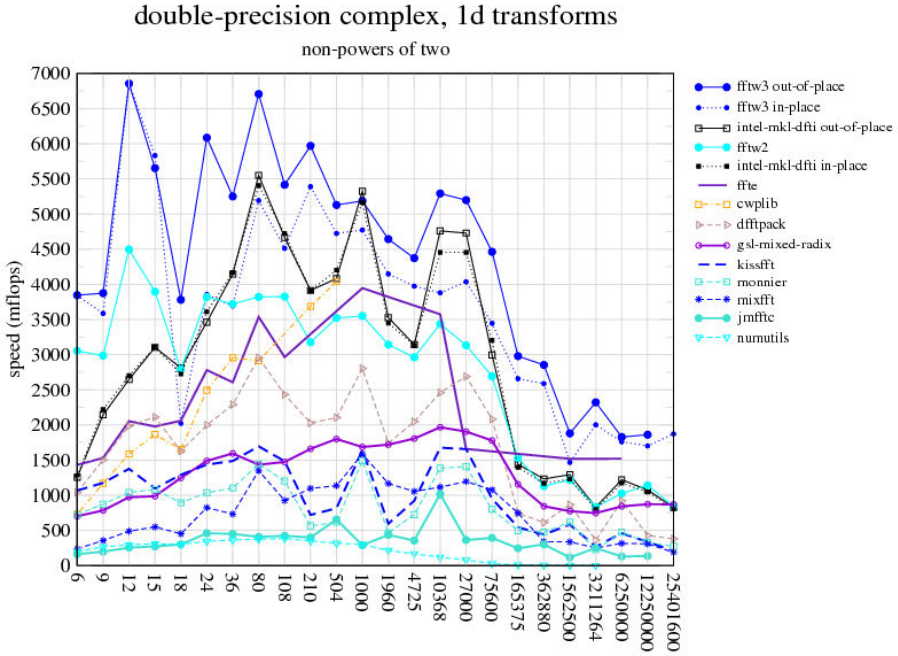


Figure 3.2: FFT benchmark (non-powers of two) [18].

FFTW supports the so called *plan and execute* approach. It means prior to the calculation the most effective algorithm and its parameters for a given architecture are selected to achieve as best performance as possible. This is especially useful in the case where a large number of FFTs needs to be calculated as the overhead of planning prior to the calculation is well worth it. There are four main planner flags and one wisdom flag:

`FFTW_ESTIMATE` specifies that, instead of actual measurements of different algorithms, a simple heuristic is used to pick a (probably sub-optimal) plan quickly. With this flag, the input/output arrays are not overwritten during planning.

`FFTW_MEASURE` tells FFTW to find an optimized plan by actually computing several FFTs and measuring their execution time. Depending on the machine, this can take some time (often a few seconds). It is the default planning option.

`FFTW_PATIENT` is like `FFTW_MEASURE`, but considers a wider range of algorithms and often produces a „more optimal“ plan (especially for large transforms), but at the expense of several times longer planning time (especially for large transforms).

`FFTW_EXHAUSTIVE` is like `FFTW_PATIENT`, but considers an even wider range of algorithms, including many that we think are unlikely to be fast, to produce the most optimal plan but with a substantially increased planning time.

`FFTW_WISDOM_ONLY` is a special planning mode in which the plan is only created if wisdom is available for the given problem, and otherwise a NULL plan is returned. This can be combined with other flags, e.g. `'FFTW_WISDOM_ONLY | FFTW_PATIENT'` creates a plan only if wisdom is available that was created in `FFTW_PATIENT` or `FFTW_EXHAUSTIVE` mode. The `FFTW_WISDOM_ONLY` flag is intended for users who need to detect whether wisdom is available; for example, if wisdom is not available one may wish to allocate new arrays for planning so that user data is not overwritten.

FFTW also supports SIMD instructions. SIMD, which stands for „Single Instruction Multiple Data“, is a set of special operations supported by some processors to perform a single operation on several numbers (usually 2 or 4) simultaneously. SIMD floating-point instructions are available on several popular CPUs: SSE/SSE2/AVX on recent x86/x86-64 processors, AltiVec (single precision) on some PowerPCs (Apple G4 and higher), NEON on some ARM models, and MIPS Paired Single (currently only in FFTW 3.2.x). FFTW can be compiled to support the SIMD instructions on any of these systems.

SSE [20] (Streaming SIMD Extensions) is an SIMD (Single Instruction Multiple Data) instruction set extension to the x86 architecture, designed by Intel and introduced in 1999. SIMD instructions can greatly increase performance when exactly the same operation is to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing. SSE was subsequently expanded by Intel to SSE2, SSE3, SSSE3, and SSE4. Today's (May 2014) latest version is SSE4.2.

SSE originally added eight new 128-bit registers known as XMM0 through XMM7. In this project we work with single precision floating point numbers, so one register can hold up to 4 real values or 2 complex values.

Next type of popular SIMD extensions is **AVX** [20] (Advanced Vector Extensions). AVX are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008. AVX provides new features, new instructions and a new coding scheme. AVX2 expands most integer commands to 256 bits and introduces FMA (Fused multiply-add). AVX-512 expands AVX to 512-bit support utilizing a new EVEX prefix encoding proposed by Intel in July 2013 and first supported by Intel with the Knights Landing processor scheduled to ship in 2015. AVX uses the same registers as SSE, but their width is inscreased from 128 bits to 256 bits and they are renamed from XMM to YMM.

A program linking to an FFTW library compiled with SIMD support can obtain a nonnegligible speedup for most complex and r2c/c2r transforms. In order to obtain this speedup, however, the arrays of complex (or real) data passed to FFTW must be specially aligned inmemory (16-byte aligned for SSE, 32-byte for AVX-256), and often this alignment is more stringent than that provided by the usual malloc allocation routines.

In order to guarantee proper alignment for SIMD, its recommended to allocate the space for data with `fftw_malloc` and de-allocating it with `fftw_free`. Using `memalign` or its equivalent directly is also possible.

FFTW allows users to use its transpose routines separatedly. That is very useful for this project since we can use its MPI transpose routine to do a global transposition for us. These routines also support the planner flags. Advanced planning using more time consuming flags can save a lot of time due to the ability to merge groups of MPI message together and send one group as one bigger message. Another very important feature is the

ability to work with transposed outputs and inputs. Generally when calculating a FFT on a distributed cluster, the output data have to be in the same shape as the input data and vice versa, meaning that two global transpositions are necessary. FFTW allows us to work with transposed input/output data, meaning only one global transposition is necessary, which can save up to 50% of the time in some cases. These flags are `FFTW_MPI_TRANSPOSED_OUT` and `FFTW_MPI_TRANSPOSED_IN`.

FFTW can calculate all transforms in-place or out-of-place.

FFTW's 1D decomposition (see Chapter 8) has unfortunatelly become a limiting factor, as we discuss later, and the 2D decomposition seems to be the best solution.

Libraries using 2D decomposition

To my best knowledge there are presently three non platform specific parallel FFT libraries using 2D decomposition. These are PFFT by Michael Pippig [31], FFTE by Daisuke Takahashi [33] and P3DFFT by Dmitry Pekurovsky [30]. All support transforms in single and double precision and are all pure MPI libraries.

PFFT's all performance-relevant building blocks are implemented with the help of the FFTW software library. In fact, it can be understood as an extension of FFTW to multidimensional process grids. The API (Application User Interface) is also very similar to the one of FFTW, so transition from one to the other is very simple. Similarly to FFTW, PFFT is able to compute FFTs of complex data, real data, and even- or odd-symmetric real data, in both single and double precision. All the transforms can be performed completely in place. PFFT uses a virtual n-dimensional mesh of $P_0 \times P_1 \times P_{n-1}$ MPI processes for a $(n+1)$-dimensional FFT.

P3DFFT is currently built on top of Message Passing Interface (MPI) and focuses only on 1D, 2D and 3D FFT using 1D or 2D processor grids. It is written in Fortran, but both Fortran a C interfaces are supported. It supports both in-place and out-of-place transforms in single and double precision, similarly to FFTW and PFFT. It does not support Complex-to-Complex transforms, only Real-to-Complex or Complex-to-Real. The global transposition is done via `MPI_Alltoall` or `MPI_Alltoallv`.

FFTE does not support in-place transforms, therefore we will omit it.

FFTW, PFFT and P3DFFT were tested and compared [31] on BlueGene/P [3], BlueGene/Q [4] and JuRoPa [5] machines.

BlueGene/P in Research Center Jülich (JuGene)	One node of a BlueGene/P consists of 4 IBM PowerPC 450 cores that run at 850 MHz. These 4 cores share 2 GB of main memory. Therefore, we have 0.5 GB RAM per core whenever all the cores per node are used. The nodes are connected by a three-dimensional torus network with 425 MB/s bandwidth per link. In total JuGene consists of 73728 nodes, i.e., 294912 cores.
BlueGene/Q in Research Center Jülich (JuQueen)	One node of a BlueGene/Q consists of 16 IBM PowerPC A2 cores that run at 1.6 GHz. These 16 cores share 16 GB SDRAM-DDR3. Therefore, we have 1 GB RAM per core whenever all the cores per node are used. The nodes are connected by a five-dimensional torus network. In total JuQueen consists of 24576 nodes, i.e., 393216 cores.
Jülich Research on Petaflop Architectures (JuRoPA)	One node of Juropa consists of 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors that run at 2.93 GHz. These 8 cores share 24

GB DDR3 main memory. Therefore, we have 3 GB RAM per core whenever all the cores per node are used. The nodes are connected by an Infiniband QDR with nonblocking fat tree topology. In total JuRoPA consists of 2208 nodes, i.e., 17664 cores.

The authors tested complex-to-complex out-of-place FFTs of size 512^3 and 1024^3. Since P3DFFT supports only real to complex FFTs, they applied P3DFFT to the real and imaginary parts of the complex input matrix to get times comparable to those of the complex-to-complex FFTs of the PFFT package. The test runs consisted of 10 alternate calculations of forward and backward FFTs. Some of the most interresting results are shown in figures below.

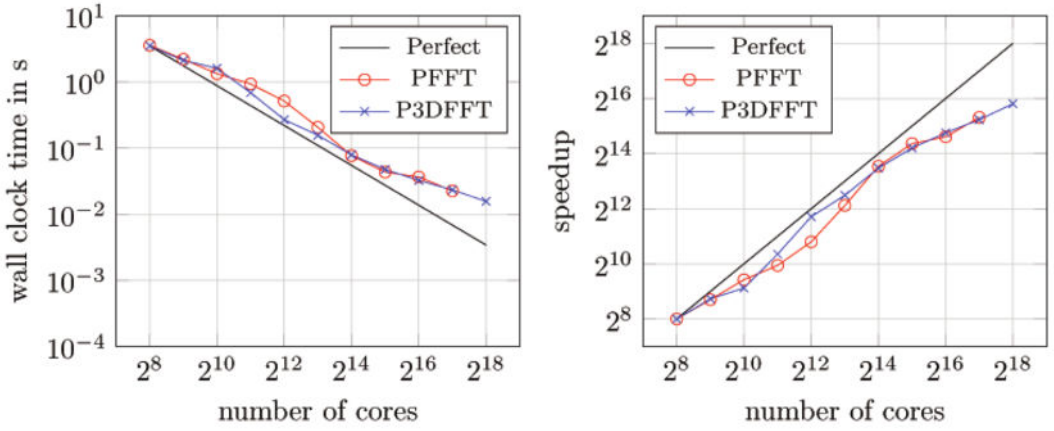


Figure 3.3: Wall clock time (left) and speedup (right) for FFT of size 1024^3 up to 262144 cores on BlueGene/P [31].

Three most important points based on these results in regards to k-Wave are:

- It is not the performance per core that matters the most, but rather the fast interconnecting network. Figure 3.5 shows that even with relatively slow CPUs, the communication time still dominates when calculating big FFTs.
- Figures 3.4 and 3.3 show that FFTW, PFFT and P3DFFT perform very similarly, but 2D decomposition further increases the scalability.
- The scalability is not very good when a massive number of CPUs is used as shown in Figure 3.3. Therefore it is more important to have a fast network, then having a big number of CPUs and the least important aspect is having very powerful CPUs. The idea of using low-power processors to compute FFTs efficiently has become very interresting and we will discuss that later in this thesis.

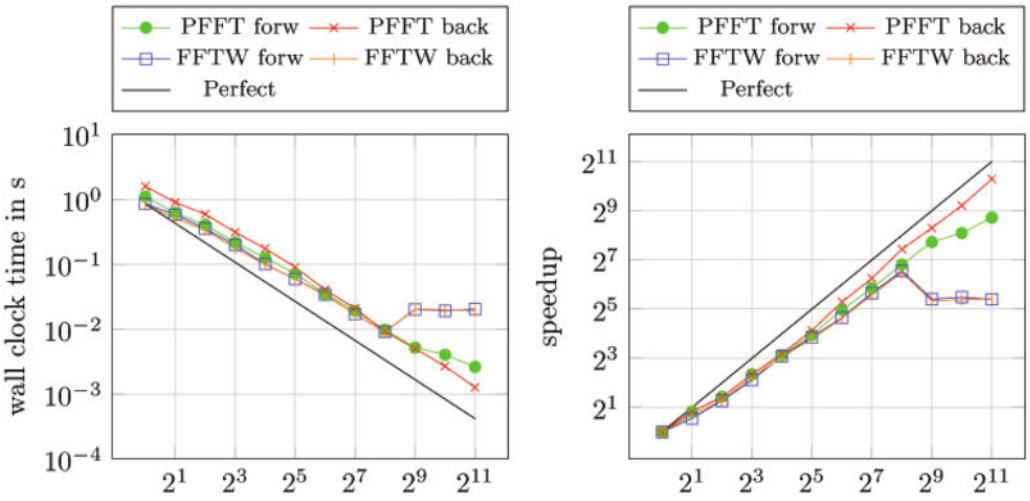


Figure 3.4: Wall clock time (left) and speedup (right) for FFT of size 256^3 up up to 2048 cores on JuRoPA [31].

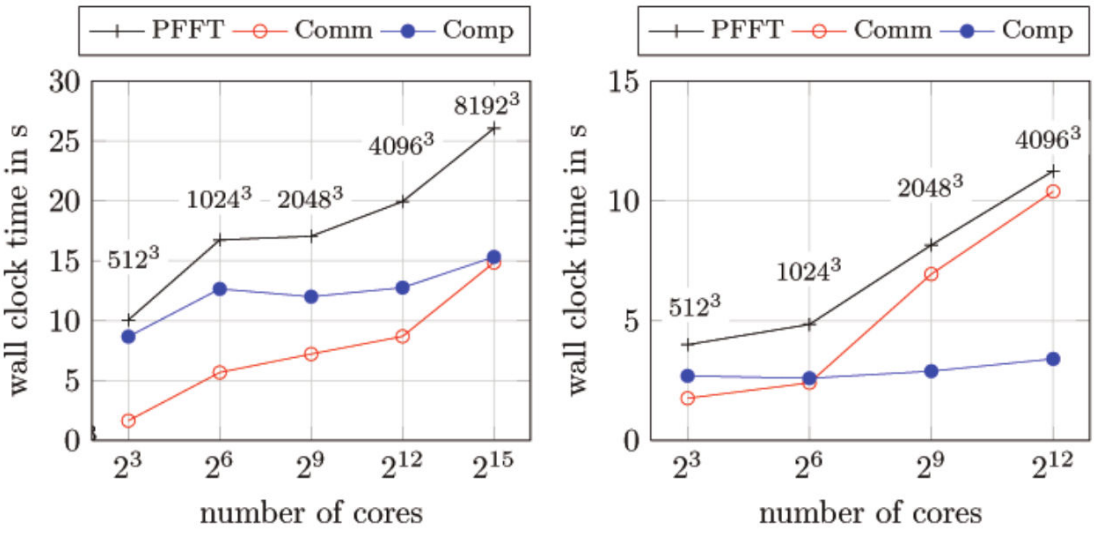


Figure 3.5: Wall clock time for FFT of constant local array size 256^3 per core up to P = 32768 cores on BlueGene/Q (left) and up to P = 2048 cores on JuRoPA (right). The figure includes the whole runtime of one forward and one backward FFT (PFFT) and the time spent for communication (Comm) and computation (Comp). The numbers next to data points indicate the the total FFT size [31].

Chapter 4

Message passing interface

Message Passing Interface (MPI) is a standardized and portable message-passing system designed tu run on a wide variety of parallel computers. The standard defines the syntax and semantics of library routines useful to a wide range of users writing portable message-passing programs in Fortran or the C/C++ programming language. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and encouraged the development of portable and scalable large-scale parallel applications. In k-Wave, we use the *Open MPI* implementation, which was introduced in 2004 [19] and its full documentation is available here [28].

MPI is also a language-independent communications protocol used to program parallel computers with both point-to-point and collective communication are supported. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in HPC (high-performance computing) today.

Although MPI belongs to the layers 5 and higher of the OSI Reference Model, implementations may cover most layers, with sockets and Transmission Control Protocol (TCP) used in the transport layer. It has also support for RMDA, OFED (Infiniband) and many other protocols reducing the latency of message dispatch.

MPI is built on the concept of distributed memory environment. Each process has its own local memory, which only it can access directly. Sharing data between processes is available only via messages. Main advantages of this approach are no need to synchronize the processes (with some exceptions, which we will discuss later in this chapter) and the programmer's full control of the processes' data access. The disadvantages are the communication overhead due to the need of communication over the network to share data, duplications of data on multiple processes in the case they all require these data for their calculation and also a much more complicated development of MPI applications.

MPI is often compared with Parallel Virtual Machine (PVM), which is a popular distributed environment and message passing system developed in 1989. Originally, only pure MPI was supported on distributed systems. Threaded shared memory programming models (such as Pthreads and OpenMP) and message passing programming (MPI/PVM) can be considered as complementary programming approaches. Today's supercomputers are designed to have „fat“ nodes, which means many cores on a single node. Therefore communication-heavy MPI applications loose a lot of time by sending and receiving MPI messages among cores on a single node, where they could share data much faster via shared memory instead. This is called the hybrid approach - the efficient combination of threads and MPI processes to reach higher performance. This is also the case of this project, as we

combine both MPI and OpenMP together.

4.1 MPI operations

The basic work unit in MPI is the *process*. To achieve maximum performance, one process is assigned to exactly one core of a processor, if possible. The placement is carried out by an agent who runs the MPI program and is usually called *mpirun* or *mpiexec*. Processes communicate via messages, their sending and receiving is defined by the programmer via MPI routines.

MPI routines can be divided into four basic groups, *Environment Management*, *Group and Communicator Management*, *point-to-point* and *Collective*. More details can be found in the Open MPI manual [28]. Here we only show a basic overview of the MPI standard.

Environment Management Routines

This group of routines is used for interrogating and setting the MPI execution environment, and covers many purposes, such as initializing and terminating the MPI environment, querying a rank's identity, querying the MPI library's version, etc. Most of the commonly used routines are `MPI_Init` a `MPI_Finalize`, which have to be called by every process in every MPI application.

Group and Communicator Management Routines

The second group of MPI operations works with *groups* a *communicators*. A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to $N - 1$, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a „handle“. A group is always associated with a communicator object.

A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra „tag“ that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by „handles“. For example, the handle for the communicator that comprises all tasks is `MPI_COMM_WORLD`.

From the programmer's perspective, a group and a communicator are the same. The group routines are primarily used to specify which processes should be used to construct a communicator.

A communicator allows the programmer to organize processes, based upon function, into groups. It also enables the Collective Communications operations across a subset of related tasks, provides basis for implementing user defined virtual topologies and provides for safe communications. One process can be a member of multiple communicators at the same time.

The most common routines includes `MPI_Comm_size`, which returns the total number of MPI processes in the specified communicator, `MPI_Comm_rank` to find out the rank of the calling process and *MPI_Comm_split* to create new communicators.

Point-to-Point Communications

MPI point-to-point routines typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation. MPI guarantees that

- messages will not overtake each other,
- if a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2 and
- if a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.

Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode with the following properties.

- A blocking send routine will only „return“ after it is safe to modify the application buffer (the send data) for reuse. Safe means that modifications will not affect the data intended for the receive process. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A blocking receive only „returns“ after the data has arrived and is ready for use by the program.
- Non-blocking send and receive routines returns almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- Non-blocking operations simply „request“ the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (the variable space) until we know for a fact the requested non-blocking operation was actually performed by the library. There are „wait“ routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

The mose commmonly used point-to-point operations are `MPI_Send`, `MPI_Isend`, `MPI_Recv` and `MPI_Irecv`, where the `I` prefix indicates a non-blocking operation.

Collective Communication Routines

Collective communication routines are the last group of MPI operations. One routine is performed between all processes in the communicator. If one or more processes do not cooperate, it may result in undefined program behavior. Cooperation of all processes must be ensured by the programmer.

There are three basic groups of routines, *Synchronization*, *Data movement* and *Collective computation*.

A synchronization routine establishes a synchronization point where the incoming processes are blocked until all processes have reached this point. They are then free to proceed. The only MPI synchronization operation is `MPI_Barrier`. This operation can for example be used to test core dumps or it can ensure the writing sequence of processes into the file.

The most used data movement routines are `MPI_Bcast`, `MPI_Scatter` and `MPI_Gather`. But for this project the most important routine is **`MPI_Alltoall`**. Its principle is shown in Figure 4.1.

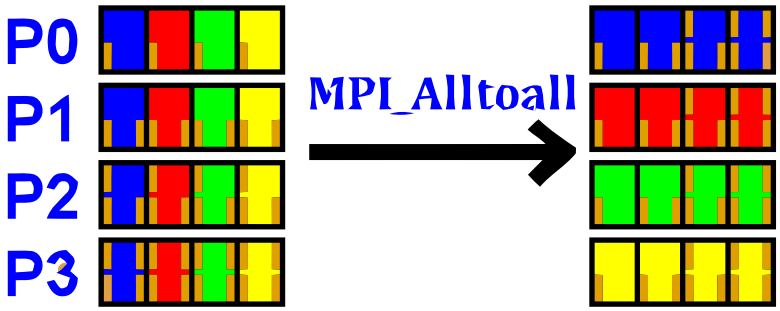


Figure 4.1: Demostration of the MPI_Alltoall routine.

This routine does a total exchange of data between all processes in the communicator and is very extensively used in the parallel matrix decomposition and its global transposition (see Chapter 8). Very similarly works `MPI_Alltoallv` routine, but it allows each process to send data of different sizes.

In collective computation routines, one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on these data. As an example we can mention `MPI_Reduce` or `MPI_Allreduce`.

Chapter 5

OpenMP

OpenMP (Open Multi-Processing) [10] is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. It may be used to explicitly direct multi-threaded, shared memory parallelism. The API supports C/C++ and Fortran on a wide variety of architectures and operating systems. In this work we will cover only some of the major features of OpenMP which we use in this project.

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA, as shown in Figure 5.1.

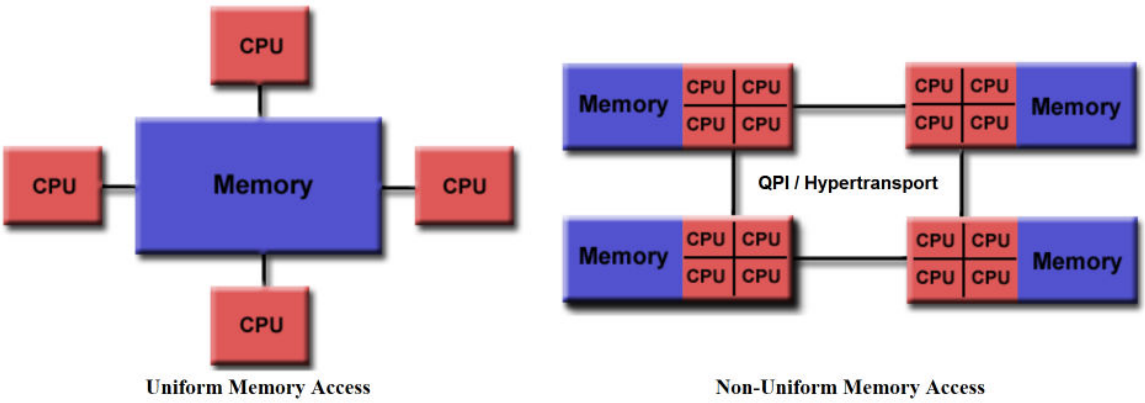


Figure 5.1: NUMA and UMA architectures [10].

UMA (Uniform memory access) [22] is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. Uniform memory access computer architectures are often contrasted with NUMA architectures. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

NUMA (Non-uniform memory access) [27] is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the

processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

Limiting the number of memory accesses provided the key to extracting high performance from a modern computer. For commodity processors, this meant installing an ever-increasing amount of high-speed cache memory and using increasingly sophisticated algorithms to avoid cache misses. But the dramatic increase in size of the operating systems and of the applications run on them has generally overwhelmed these cache-processing improvements. Multi-processor systems without NUMA make the problem considerably worse. Now a system can starve several processors at the same time, notably because only one processor can access the computer's memory at a time [13].

NUMA attempts to address this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory. For problems involving spread data (common for servers and similar applications), NUMA can improve the performance over a single shared memory by a factor of roughly the number of processors (or separate memory banks) [26].

Basics of OpenMP thread based parallelism are:

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.
- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives.

5.1 The fork-join model

OpenMP uses the fork-join model of parallel execution shown in Figure 5.2. All OpenMP programs begin as a single process - the master thread. The master thread executes sequentially until the first parallel region construct is encountered. The master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread. The number of parallel regions and the threads that comprise them are arbitrary.

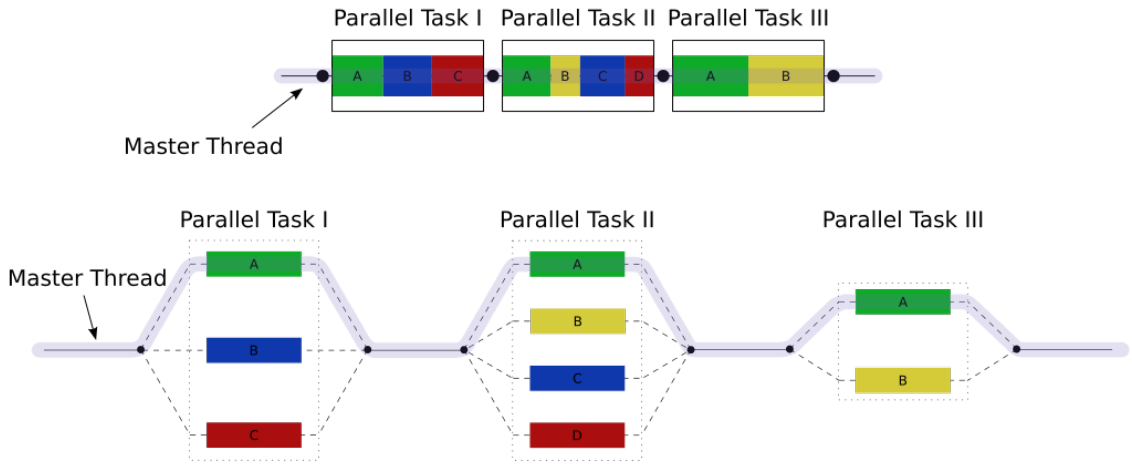


Figure 5.2: Fork-join model of OpenMP [10].

5.2 API components

OpenMP has three primary API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

Compiler directives appear as comments in the source code and are ignored by compilers unless we tell them otherwise - usually by specifying the appropriate compiler flag. The common format is

```
#pragma omp directive-name [clause, ...]  newline.
```

An important note is that each directive applies to at most one succeeding statement, which must be a structured block. These directives are used for spawning a parallel region, usually a loop, where the work is divided among threads, synchronization among threads etc.

A very important directive is the *parallel* directive. When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team. Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code. There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

One of the clauses used for an effective parallelizing of nested loops is the *collapse* clause. It specifies how many loops in a nested loop should be collapsed into one large iteration space, allowing for deeper parallelism and a more efficient use of threads in the case where the number of iterations of one loop is smaller than the actual number of spawned threads.

Another important clause is the *schedule*. It describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. In this project we tested all schedule variants and the results show that the performance benefit can vary very significantly on different machines, even by tens of percent. This

will have to be further tested in near future and possibly tuned for each specific machine. Currently the STATIC variant is used, as it gives the most stable results because of the data locality of NUMA. The main types are:

`STATIC` Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

`DYNAMIC` Loop iterations are divided into pieces of size *chunk* and then dynamically scheduled among the threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

`GUIDED` Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to

```
number_of_iterations / number_of_threads.
```

Subsequent blocks are proportional to

```
number_of_iterations_remaining / number_of_threads.
```

The chunk parameter defines the minimum block size. The default chunk size is 1.

`RUNTIME` The scheduling decision is deferred until runtime by the environment variable `OMP_SCHEDULE`. It is illegal to specify a chunk size for this clause.

`AUTO` The scheduling decision is delegated to the compiler and/or runtime system.

The opposite to the parallel is the *single* directive, which specifies that the enclosed code is to be executed by only one thread of the thread team. It is mainly usefull in the case of a code inside a parallel region that is not thread safe, such as I/O. Threads in the team that do not execute the single directive, wait at the end of the enclosed code block, unless a *nowait* clause is specified. The global MPI transposition is enclosed in a single region and therefore is done only by the master thread, otherwise a serious perfomance degradation occures.

Runtime library routines are used for setting and querying the number of threads, their unique identifiers, the thread team size, nested parallelism, setting, initializing and terminating locks etc. For C/C++, all of these routines are actual subroutines. As an example, `void omp_set_num_threads(int)` or `int omp_get_num_threads(void)`.

Environment variables can be used to control the number of threads, specifying how loop interactions are divided, binding threads to processors and their cores, setting the maximum level of nested parallelism etc. Setting these variables is done the same way any other environment variables are set, and depends upon which shell is used. In this project, the only variable we set is `export OMP_NUM_THREADS=8`.

Chapter 6

Cluster Computing Today

The possibility for parallel execution of computations strongly depends on the architecture of the execution platform. Today's clusters are built on the NUMA approach, as described earlier, and have usually two sockets per one node, but each node has only one network card attached to one of the two sockets. Therefore if we want to run one MPI process per node, it is more efficient to run it on the socket „closer“ to the network card.

A physical connection between the different components of a parallel system is provided by an interconnection network and it can also be used for a classification of parallel systems. Internally, the network consists of links and switches which are arranged and connected in some regular way. In multicomputer systems, the interconnection network is used to connect the processors or nodes with each other. Interactions between the processors for coordination, synchronization, or exchange of data are obtained by communication through message-passing over the links of the interconnection network.

The bandwidth of local RAMs is around 50GB/s in the case of Anselm cluster [1] (but communication over NUMA is only 10GB/s). For 64 nodes, each consisting of 16 cores, the total bandwidth is 3,2TB/s. However the network cards' bandwidth is only $64 \times 3.5 = 220$GB/s. This means that the global communication is atleast $14\times$ slower than the local communication, if we neglect the latency! It implies that we would much prefer communicating locally than communicating globally.

Networks can be divided into two main cathegories [16], *direct* (static, distributed switches) and *indirect* (dynamic, centralized switches). Direct networks consist of a number of point-to-point links. Every node is both a terminal and a switch. Indirect networks consist of switching elements that the various processors attach to. A node is either a terminal or a switch.

Examples of direct networks are shown in Figure 6.1 (red circles are switches, green squares are nodes). *Full* connection has a direct link between every pair of nodes. *Torus* (k-ary n-cube) consists of $N = k^n$ nodes arranged in a n-dimensinal cube with k nodes along each dimension. *Meshes* do not have wrap-around links. *Ring* topology is a 1D torus. *Hypercubes* are binary cubes and meshes at the same time, hierarchically recursive (n-cube contains cubes of dimensions less than n as its subgraphs). *Fat hypercubes* are more economical in terms of a switch count at a cost of lower performance.

Examples od indirect networks are shown in Figure 6.2. *Clos* network is a 3-stage network, each stage consists of a number of small crossbars (a switch connecting multiple inputs to multiple outputs in a matrix manner). Clos network is symetric when (m, n, r) = (3, 3, 4). From each input there are m paths to each output. Clos network in strongly non-blocking if $m >= 2n - 1$ and rearrangeably nonblocking if $m >= n$. A bidirectional

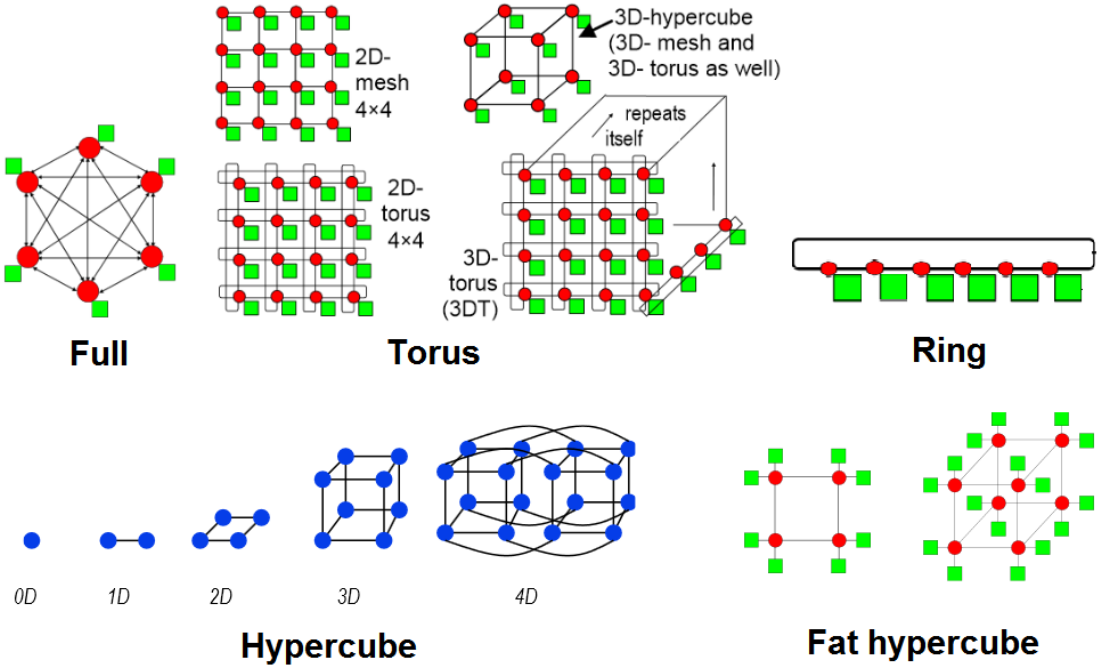


Figure 6.1: Examples of direct networks [16].

butterfly network is known as a *fat tree*. Distances between node pairs are non-uniform!

6.1 Portable Batch System

One cluster usually has to handle requests from many of its users. To divide the cluster resources fairly and evenly among all users, there needs to be a scheduler that performs job scheduling. One of the most used today is PBS.

PBS (Portable Batch System) [7] is the name of computer software that performs job scheduling. Its primary task is to allocate computational tasks, i.e., batch jobs, among the available computing resources. It is often used in conjunction with UNIX cluster environments.

Requests for allocating resources are done via bash PBS scripts. Each script has to specify needed resources, such as walltime, number of nodes etc. and has to be put into the queue by `qsub` command. PBS then puts the request into the waiting queue and when the resources are free and the requirements are met, PBS runs the script on the target machine.

A simple example of such script is shown in Figure 6.3. We request the *short* queue, *walltime* is set to 2h (if the program does not end before the 2h mark, PBS will kill it), we also request 16 nodes, 16 processes per node, 16GB of memory per node, on ZAPAT cluster. The job is named *runtime-test*. 'cd $PBS_O_WORKDIR' ensures that the target cluster starts the program from the right directory. We load the openmpi module by the *module load* command and finally we run the *hybrid-test* application using 32 MPI processes, 1 MPI process per socket (each node has 2 sockets) and 8 OpenMP threads per socket.

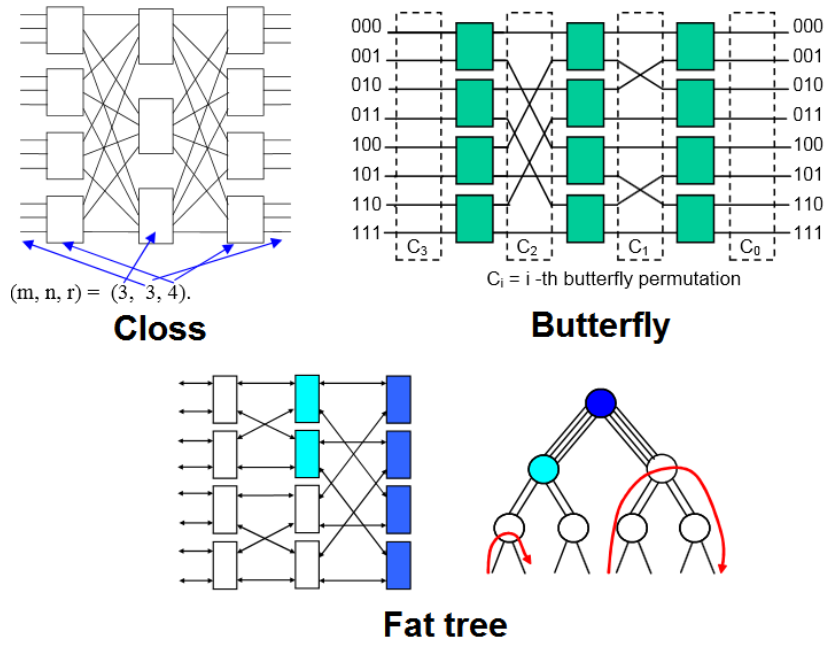


Figure 6.2: Examples of indirect networks [16].

```
#!/bin/bash
#PBS -q short
#PBS -l walltime=02:00:00
#PBS -l nodes=16:ppn=16:zapat,mem=16gb

#PBS -N runtime-test

cd $PBS_O_WORKDIR

module load openmpi-1.6.5-gcc

OMP_NUM_THREADS=8 mpirun -np 32 -bysocket hybrid-test
```

Figure 6.3: Example of a PBS script.

Chapter 7

Matrix Transposition

As we said earlier in Chapter 3, we need to calculate a series of 1D FFTs along each axis of the matrix to compute the whole N-dimensional FFT. But how do we index these 1D arrays? Since the 3D matrix is stored in the row-major order, only elements along the first axis are stored contiguously. If we wanted to access elements along any other axis, we would have to do big jumps in memory to access following elements. Unfortunately that goes completely against the principle of data locality on today's architectures and it would cause an inacceptable number of cache misses and therefore ruin the overall performance. It becomes even more of a problem in a distributed memory environment, where parts of the matrix are stored on different machines. In order to preserve the data locality, we have to transpose the matrix so the elements along the current axis are stored in contiguously.

The matrix transpose is a simple swap of two coordinates of all its elements. Nothing changes by transposing a 1D matrix, because its elements have only one coordinate. By transposing a 2D matrix, we swap the x and y coordinates of each element of the matrix, so it turns the matrix over its main diagonal. For n-dimensional matrices where $n > 2$, we must specify which two coordinates we want to swap. For example there are 3 options to transpose a 3D matrix, $x \leftrightarrow y$, $x \leftrightarrow z$ and $y \leftrightarrow z$.

At first, we will discuss the 2D matrix transposition, because it transposes $x \leftrightarrow y$ dimensions locally, and then we move to the global MPI transposition, which is used to transpose $x \leftrightarrow z$ among MPI processes.

7.1 2D Matrix Transposition

There are many ways how to transpose a 2-dimensional matrix. The most obvious one is to swap pairs of inverse elements over the diagonale one by one. The pseudocode of this naive algorithm is shown in Figure 7.1.

```
PROCEDURE naive_transpose(matrix, matrix_size) BEGIN
  FOR i=0 TO matrix_size STEP 1
    FOR j=0 TO i-1 STEP 1
      swap(matrix[i,j], matrix[j,i])
END
```

Figure 7.1: Pseudocode of the naive 2D square matrix transposition

This approach can be improved by dividing the matrix into separate square blocks. The

main advantage is that we don't move across the whole matrix, but only within each block of the matrix, improving the data locality. The size of a block is usually between 8×8 and 64×64, depending on the architecture and its cache sizes. After transposing two inverse blocks over the diagonale, these blocks are then swapped. The pseudocode is shown in Figure 7.2. Note that the blocks on the main diagonale are not treated. We will focus on that later in this chapter.

```
PROCEDURE block_transpose(matrix, matrix_size, block_size) BEGIN
  FOR i=0 TO matrix_size STEP block_size
    FOR j=0 TO i STEP block_size
      FOR k=i TO block_size STEP 1
        FOR l=j TO block_size STEP 1
          swap(matrix[k,l], matrix[l,k])
END
```

Figure 7.2: Pseudocode of the block 2D square matrix transposition

In this project, run times with different block sizes were measured, and it turned out that the best size in terms of performance is 8×8 in the scalar pair swapping within each block, 16×16 when using the SSE extension and 32×32 when using the AVX extension.

The transposition within each block using SSE is very similar to previously shown common block transposition algorithm, but instead of just swaping two elements, we swap 2 blocks of 4 (16) elements at the same time using 4 (8) SSE registers, depending of the float precision. The pseudocode is shown in Figure 7.3 (consider that one element is a 64-bit real number or a 2×32-bit complex number).

```
PROCEDURE SSE_block_transpose(matrix, matrix_size, block_size) BEGIN
  FOR i=0 TO matrix_size STEP block_size
    FOR j=0 TO i STEP block_size
      FOR k=i TO block_size STEP 4
        FOR l=j TO block_size STEP 4
          swap(matrix[k..k+3,l], matrix[l,k..k+3])
END
```

Figure 7.3: Pseudocode of the SSE block 2D square matrix transposition

We can also use AVX registers to do the transposition. The only difference is the lenght of the registers (AVX has 256-bit registers). We will focus more on the AVX version in Chapter 9.

7.2 Distributed 3D Matrix Transposition

The parallel global transposition among all MPI processes is done by the FFTW software library. Its planning routines try and measure multiple algorithms and choose to best one for the current architecture it runs on. The authors say that the chosen algorithm is never slower than the `MPI_Alltoall` routine (see Chapter 4), which is also considered as one of the options by the planner.

To calculate the FFT of a 3D matrix, we need to do one global transposition to swap the x and z coordinates of each element of the matrix in case of the 1D decomposition (see Chapter 8).

Chapter 8

Matrix Decompositions

In order to compute the FFT (see Chapter 3) of a N-dimensional matrix, we need to decompose it along one or more axis [32]. In our project it is basically a question of how to distribute the matrix among all compute units, which in our case are MPI processes and OpenMP threads.

k-Wave has up until now used the 1D decomposition of FFTW and the pure MPI approach. Its principle is shown in Figure 8.1. This decomposition uses only MPI processes as compute units. Each process has generally one or more slabs in its local memory (specifically two in Figure 8.1) at the beginning and perfoms these 6 steps:

- Series of 1D FFTs along the x axis.
- Local transposition of slabs on each process.
- Series of 1D FFTs along the y axis.
- Global transposition over all processes.
- Series of 1D FFTs along the z axis.
- Backward transpositions to get the data into the original shape (can be omitted under some circumstances).

The FFTW and its 1D decomposition is very flexible and simple to use and is perfectly usable in case we do not need to parallelize over a high number of cores and the amount of memory per core is sufficient for our needs. Unfortunatelly this is not the case of k-Wave anymore. This decomposition is optimal on limited number of processors because it only needs one global transposition. The disadvantage is that the maximum parallelization is limited to the length of the largest axis of the 3D data. The maximum number of CPUs scales as $O(N^{1/3})$ and the work scales as $O(NlogN)$ thus resulting in poor weak scaling.

This scaling limitation can be overcome by using a 2D decomposition [32] as shown in Figure 8.2. The computation is done in these six steps similar to the 1D decomposition, but the local transposition is replaced by the second global transposition, which transposes withing subgroups of processes.

- Series of 1D FFTs along the x axis.
- Global transposition within subgroups of processes.

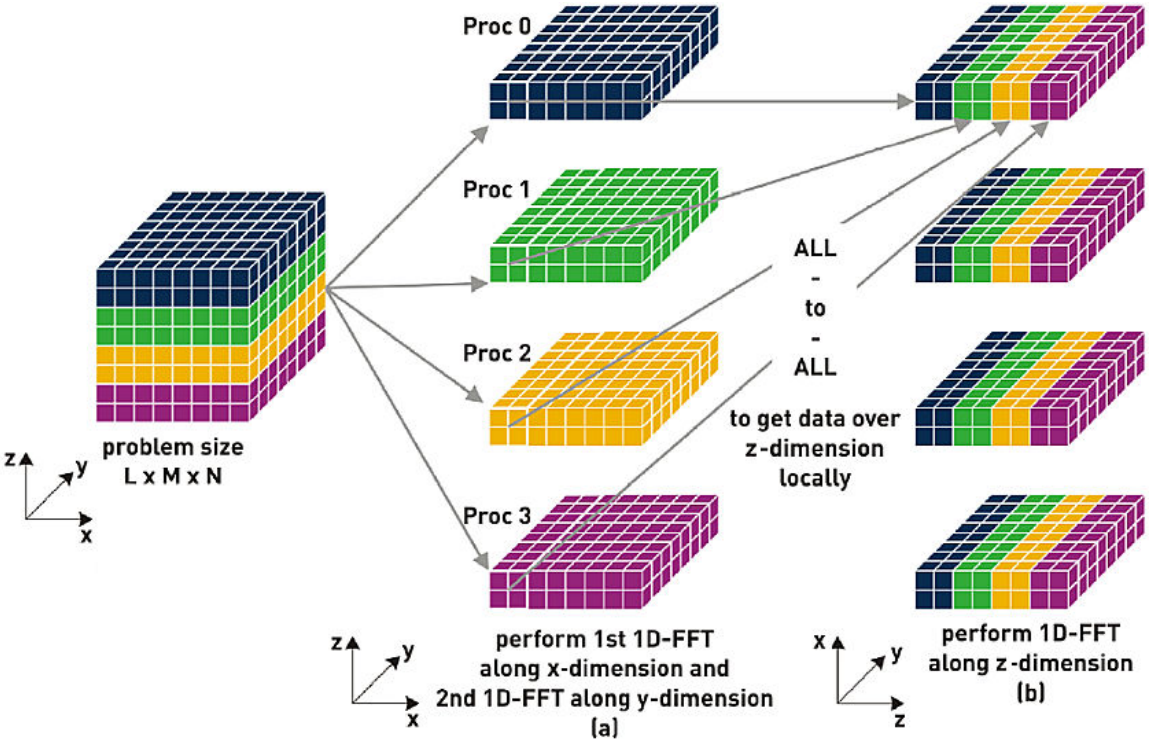


Figure 8.1: 1D decomposition of the 3D FFT [32].

- Series of 1D FFTs along the y axis.
- Global transposition among subgroups of processes.
- Series of 1D FFTs along the z axis.
- Backward transpositions.

The 2D decomposition has still an inherent scaling limitation because the maximum number of CPUs only scales as $O(N^{2/3})$ but this limitation is of no practical relevance because for any practical size the network communication time limits the number of nodes more significantly. This is also related to the disadvantage of the 2D decomposition. As it requires two global transpositions instead of one it might be slower than the 1D decomposition with the same number of processes. However our tests showed that the difference is negligible (see Chapter 10).

Even though we overcame the limitation of maximum processes, scaling itself becomes worse and worse with the increasing number of processes due to the bandwidth and latency of the network. Therefore we came with the idea to use threads instead of processes to parallelize work along the second axis. The principle is very similar to the 1D decomposition shown in Figure 8.1, but 1D FFTs and local transpositions are accelerated by OpenMP threads.

This so called *hybrid* approach has multiple advantages over the pure MPI approach of the 2D decomposition shown above.

- Only one global transposition is needed, the second transposition is done by threads in shared memory.

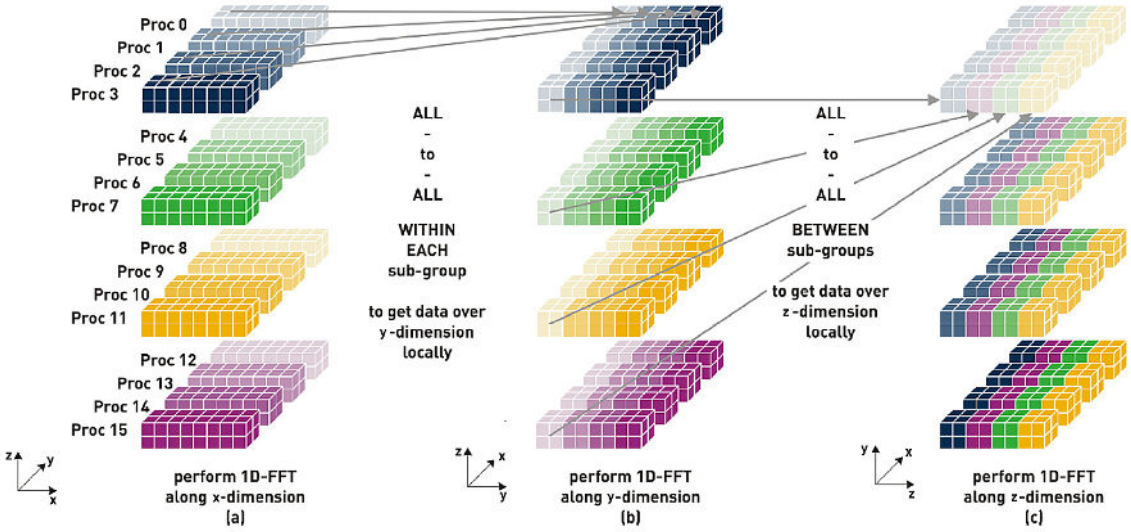


Figure 8.2: 2D decomposition of the 3D FFT using only MPI processes [32].

- Lower number of MPI processes, therefore bigger MPI messages and less flooded network.
- Flexible ratio of processes and threads.
- Utilizing the shared memory, therefore each core has more work oportunities without the need to communicate with other cores on the same socket.

Chapter 9

Implementation

This project is implemented in C/C++. The external software libraries used are FFTW v. 3.3.4 [17] and Open MPI v. 1.6.5 [28]. As for compilers we tested both GCC v. 4.8.1 [12] and Intel C++ Compiler v. 13.5 [11], though we noticed no difference in performance between these two.

9.1 High-level Overview

Firstly, the input 3D matrix is distributed among all MPI processes by the parallel HDF5 software library [6]. Each process has one or more slabs stored locally in its memory. On each of these slabs, OpenMP threads perform series of 1D FFTs and the local AVX block transposition in parallel. The global MPI transposition is done only by processes, without the presence of threads. A complete diagram of this process is shown in Figure 9.1 and details are explained in the following sections.

Running one MPI process per one socket proved to be most efficient since today's clusters use non-uniform memory access. The first touch strategy (each core accesses its part of the allocated array right after the allocation, for example by writing zeros) was also applied so that each core has its part of the matrix in its own local memory.

As we can see in Figure 9.1, two global transpositions are necessary to do a FFT and return data into the right shape. Some applications, including k-Wave, can work with transposed data. What it means is that we can omit the second global transposition. Since global transpositions consume the most time, this optimization can save up to 50% of the time.

It is necessary for our HybridFFT to work with the exact same inputs and and return the same outputs as FFTW due to preserving the compatibility (otherwise we would have to do major changes in the k-Wave core). Therefore Complex-to-Complex, Real-to-Complex and Complex-to-Real transforms were implemented. The correctness of our FFT was tested via several utilities. We implemented a random 3D matrix generator, which created a 3D matrix with random values and stored it into an HDF5 file. A FFT of this random matrix was then calculated by our HybridFFT, FFTW and Matlab and the results were stored again into separate HDF5 files. As the last step, we implemented a comparing utility that compared these results and printed the maximum and the average deviation. The absolute value of the maximum deviation was usually around 10^{-4} and less for random input values ranging from 0.0 to 1000.0 which is a sufficient value for our needs.

A simple example explains how we use OpenMP to parallelize a series of 1D FFTs, the

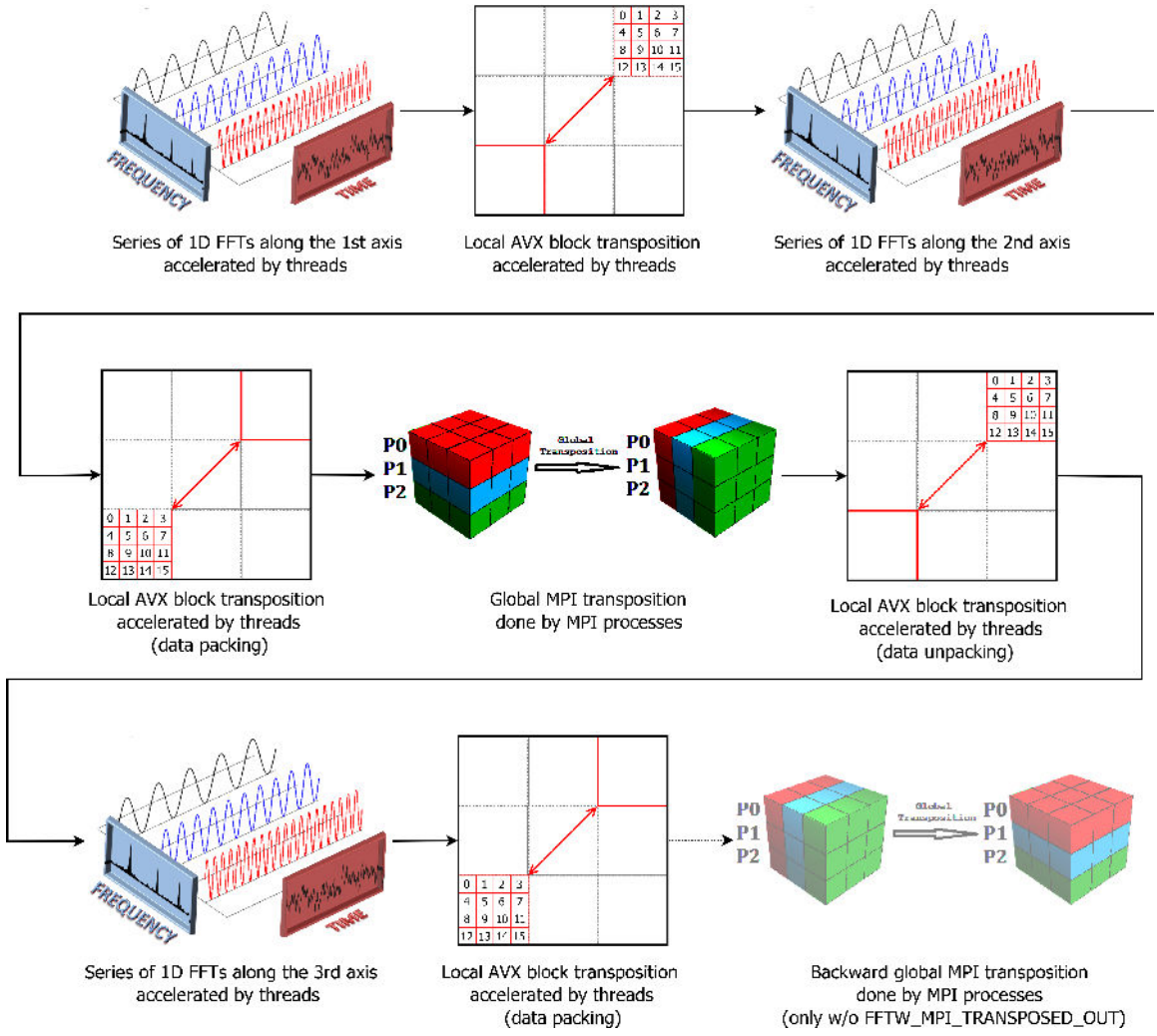


Figure 9.1: One forward FFT of the HybridFFT

AVX block transposition and using the *master* clause (because the MPI buffers are located on the master thread) to run the global MPI transposition by only the master thread, is shown in Figure 9.2.

The number of blocks along one axis can be smaller than the number of spawned threads. This is very often the case of our block transposition, therefore we parallelize blocks along both axis at the same time using a *collapse(2)* clause.

9.2 Series of 1D FFTs

1D FFTs are calculated by the FFTW and its `fftwf_plan_dft_1d`, `fftwf_plan_r2c_1d` and `fftwf_plan_c2r_1d` planning routines and `fftwf_execute_dft`, `fftwf_execute_r2c` and `fftwf_execute_c2r` executing routines. The adressing of the 1D arrays is done by a movable pointer. Transforms are done in parallel by OpenMP threads (1 transform = 1 thread).

9.3 2D Matrix Transposition

As we described in Chapter 7, we use the block transposition with the help of SSE or AVX, if supported on a target machine. The local slab is divided into blocks, typically of the size of 32×32 in the case of AVX, and each of these blocks is internally divided into 4×4 miniblocks, which equals the size of four AVX registers. These four registers transpose their values and then two transposed miniblocks inverse over the diagonale are swapped.

Since transposing a matrix using AVX is faster than both SSE and scalar, we will focus more closely on the AVX version (scalar and SSE are done very similarly). In Figure 9.3, the principle is shown more closely. As we previously said, the matrix is divided into blocks. To keep things as simple as possible, the size of each block is equal to four 256-bit AVX registers, which is 4×4 in terms of 64-bit double precision (optimal size is 32×32). The transposition of this single 4×4 block is done via Intel Intrinsics [20] functions. These functions are directly mapped to their SIMD instructions, but the compiler can still optimize some properties such as register allocation or the instruction order.

Each block is transposed using 8 AVX instructions (not including the load and store instructions). After transposing two inverse blocks over the diagonale, we swap these blocks simply by storing the register values into the correct memory addresses.

To further increase the performance, we use the OpenMP threading capabilities to transpose blocks in parallel on multiple cores of a single socket.

9.4 Distributed Matrix Transposition

The global MPI transposition is done by FFTW and its `fftwf_mpi_plan_many_transpose` planning routine and `fftwf_execute` executing routine. Its functionality is equal to the `MPI_Alltoall` routine, but it supports multiple algorithm and it chooses the best one for the current architecture. Ilustration of this process is shown in Figure 9.4 on a $3 \times 3 \times 3$ matrix using 3 MPI processes, each starting with one slab in the z axis.

```
#pragma omp parallel
{

  ...

  #pragma omp for schedule(static)
  for (int i = 0; i < slabSize; i+=dimensionSize)
    fftwf_execute_dft(transformPlan1D, &data[i], &data[i]);

  ...

  #pragma omp for schedule(static) collapse(2)
  for(int i=0; i<dimSizeX; i+=blockSize)
  { // over blocks along the first axis
    for(int j=i; j<dimSizeY; j+=blockSize)
    {  // over blocks along the second axis
      for(int i2=i; i2<max_i2; i2+=blockStep) {
        int j2;
        if (i==j) //treat blocks on the diagonale
          j2=i2;
        else
          j2=j;
        for( ; j2<max_j2; j2+=blockStep) {
             // transpose and swap two inverse blocks over
             // the diagonale
          (*transposeBlock)(&matrix[i2*dimSize+j2],
                            &matrix[j2*dimSize+i2]);
        }
      }
    }
  }

  ...

  #pragma omp master
  {
    // execute the MPI global transposition
    fftwf_execute(globalTransposePlan);
  }

  ...

}
```

Figure 9.2: OpenMP compiler directives example

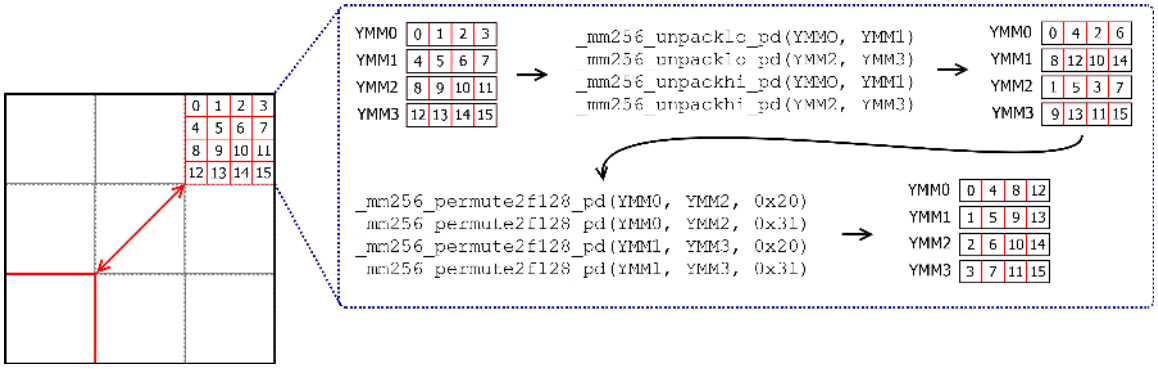


Figure 9.3: 2D matrix transposition using AVX

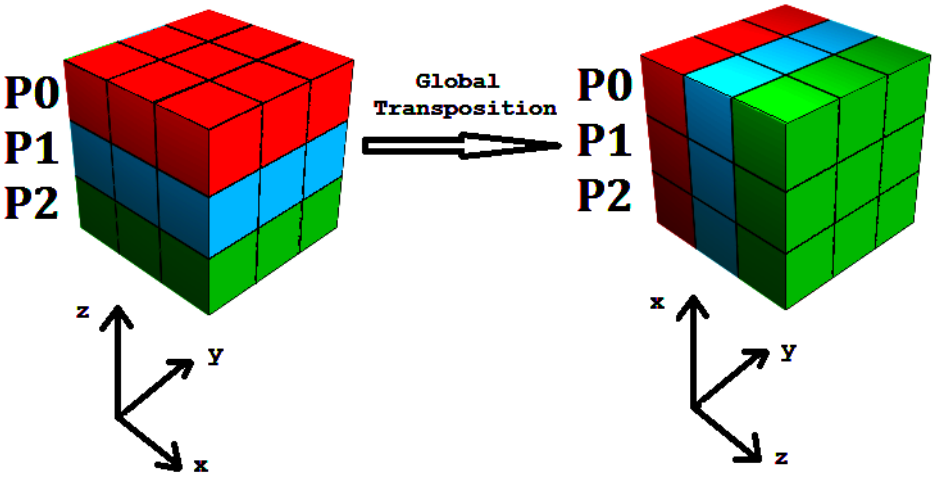


Figure 9.4: Global MPI transposition example.

Chapter 10

Experimental results

We tested on Zapat cluster [8] located in Jihlava, Czech Republic and Anselm cluster [1] located in Ostrava, Czech Republic. The hardware configurations are:

ZAPAT 112 nodes (1792 CPUs), one node has 2× 8-core Intel E5-2670 2.6GHz, 128GB RAM (14.336TB total), 2 × 600 GB 15k hard drives, Infiniband 40 Gbit/s.

ANSELM 209 nodes (3344 CPUs), one node has 2× 8-core Intel E5-2665 2.4GHz, 64–512GB RAM (15.136TB total), Infiniband 40 Gbit/s QDR, fully non-blocking fat-tree.

To test our HybridFFT against PFFT and P3DFFT, small sample programs were implemented. We tested forward complex-to-complex single precision FFTs of the size ranging from 128^3 to 2048^3. Since P3DFFT does not support complex-to-complex transforms, we simulated it by doing real-to-complex transforms of both real and imaginary parts of the input. Execution times were measured by the `MPI_Wtime` routine. Each forward FFT was run 100× in a loop to make sure everything settles down properly (jump predictors etc.). We tested using the `FFTW_PATIENT` planner flag. `FFTW_EXHAUSTIVE` had sometimes worse runtimes than the other flags for some strange reason, so we used `FFTW_PATIENT` instead.

Firstly we tested FFTs of the size of 128^3–1024^3 using 128–512 cores on Zapat. Our HybridFFT run one MPI process per socket, eight OpenMP threads per socket (one per core), so we had 8× lesser MPI processes than in the case of FFTW, PFFT and P3DFFT, which run one MPI process per core.

The scaling behaviour shown in Figures 10.1, 10.2, 10.3 and 10.4 indicates that these four libraries perform very similarly on a relatively low number of cores, but we can still notice some specific behaviour of our HybridFFT. HybridFFT shows superiority over PFFT and P3DFFT in the case of a 256^3 matrix. This is most likely due to the size of MPI messages sent during the global transposition, where HybridFFT's messages have 2MB (which is most likely the „the sweet spot“ of ZAPAT) and the other libraries send 256KB messages, in the case of employing 512 cores. In other tests all libraries perform very similarly. It would be very interresting to run these tests on a higher number of cores, around 8–32 thousand cores, because the MPI messages of HybridFFT would still be 8× bigger than in the case of the other libraries, but the number of messages sent over the network would definitelly reach the latency threshold of the network, therefore our HybridFFT is expected to show superior results over the others.

The strong scaling shown in Figure 10.5 presents a below linear scaling of smaller matrices, again due to a high number of relatively small MPI messages send over the network. But super-linear scaling can be seen in the case of the 1024^3 FFT. This is again due to the properties of the network, where „adjusting“ the size of MPI messages towards „the sweet spot“ of the network is very beneficial. Big messages are limited by the bandwidth and small message are limited by the latency, therefore it is most beneficial to be as close as possible to having messages of an optimal size and having „not too many“ of them.

As we noticed earlier, the global transposition takes the most time to compute of all the steps needed to calculate the whole FFT. In Figure 10.6, we can see that in the case of relatively big matrices, 512^3 and more, the global transposition takes up to 90–95% of the total time. Again we must stress that it is not the performance per core that helps the most with reaching better performance, but rather the fast interconnecting network. Therefore the best theoretical system for computing a distributed FFT would be the combination of relatively slow, but very simple and low-power CPUs and a fast network with a high bandwidth and low latencies.

The next thing we tested is the scalability of our HybridFFT on a higher number of cores, up to 2048, this time on Anselm. In the case of smaller matrices, 128^3 and 256^3, the scalability is not linear, but that was expected since a high number of very small MPI messages sent over the network is limited by the network latency. But in the case of 512^3, 1024^3 and especially 2048^3, the scalability is linear. It would be very interresting to test with a higher number of cores to further see the progress of the scalability.

As the last test, run on ZAPAT, the performance of our block transposition was measured (Figure 10.8). We tested the scalar, SSE and AVX version on 1–16 OpenMP threads and a rather big 16384^2 complex single precision matrix. The size was chosen so there are enough blocks in both axis so the SIMD extensions do not get an advantage due to the better data locality. Nevertheless the results of using smaller matrices were almost identical. Also we wanted the total time to be around 0,1–1s to avoid any minor inaccuracies during the time measurement. Expectedly the AVX version performs better than the SSE, which performs better than the scalar. We can see that the SSE and AVX versions perform very similarly when a higher number of threads is used, this is due to reaching the maximum memory bandwidth of the system.

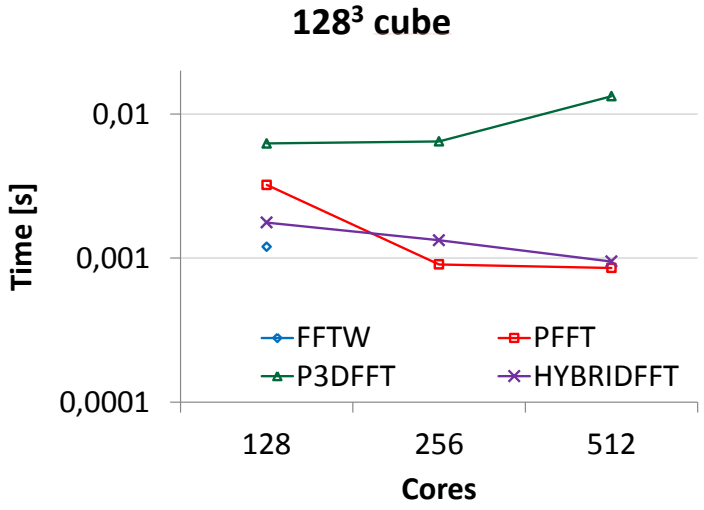


Figure 10.1: Runtime comparison of libraries - 128^3 Complex-to-Complex FFT.

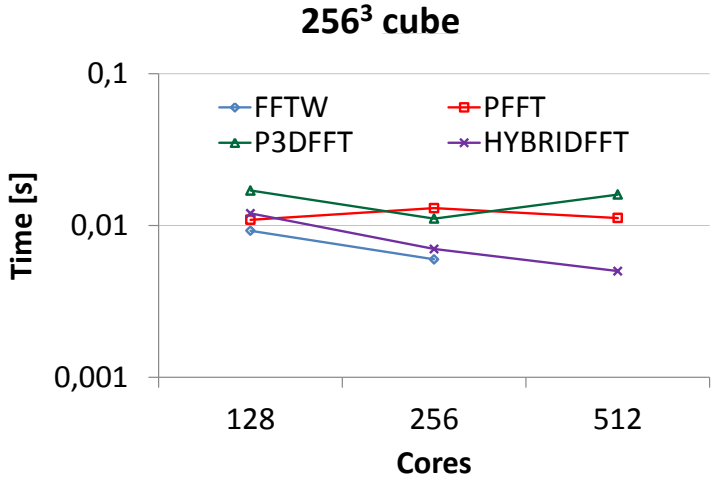


Figure 10.2: Runtime comparison of libraries - 256^3 Complex-to-Complex FFT.

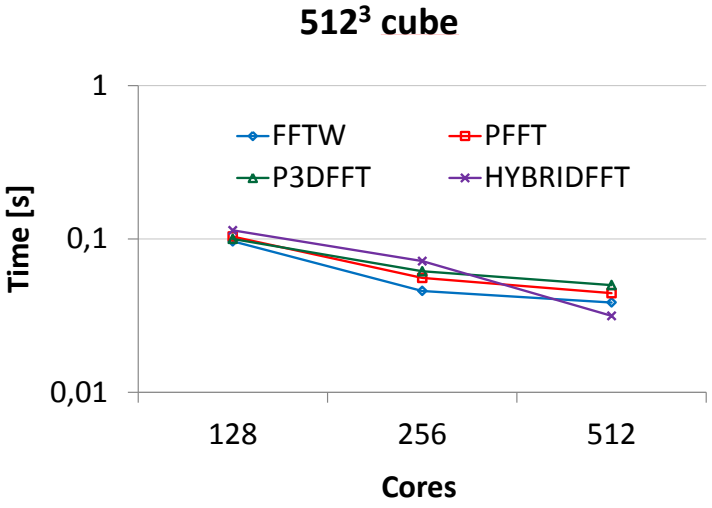


Figure 10.3: Runtime comparison of libraries - 512^3 Complex-to-Complex FFT.

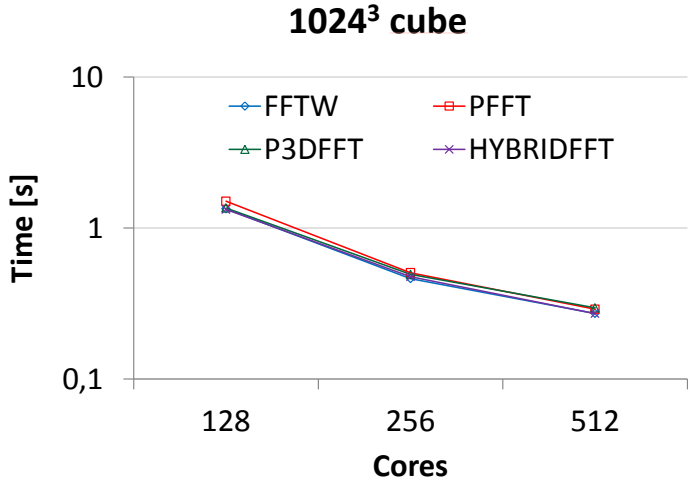


Figure 10.4: Runtime comparison of libraries - 1024^3 Complex-to-Complex FFT.

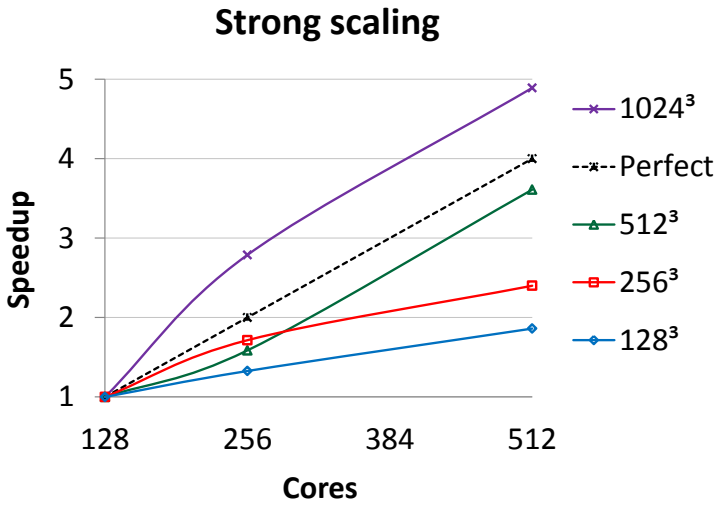


Figure 10.5: Strong scaling of the HybridFFT.

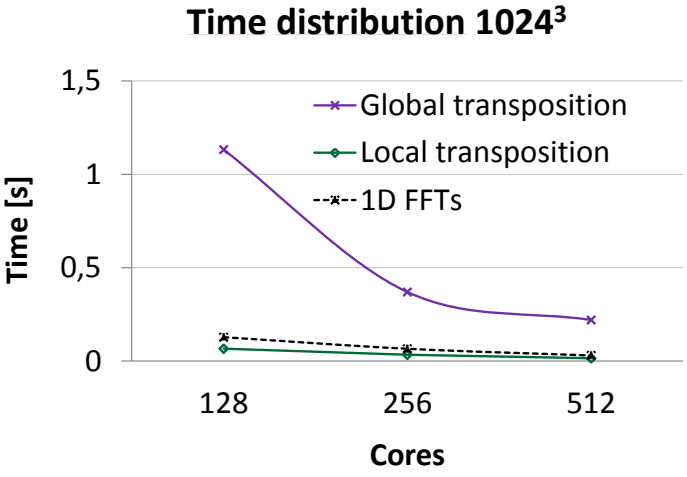


Figure 10.6: Time distribution of HybridFFT.

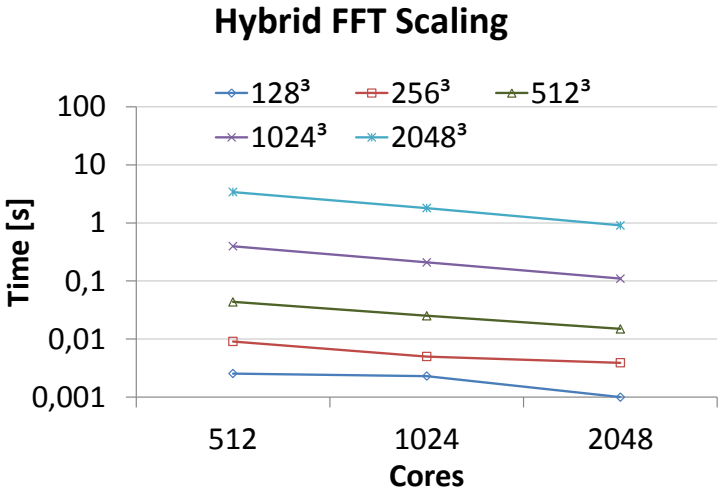


Figure 10.7: Scaling of HybridFFT.

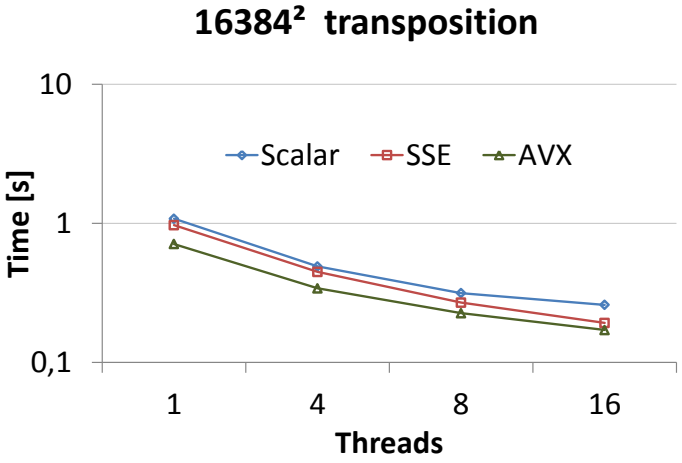


Figure 10.8: Comparison of scalar vs. SSE vs. AVX transposition on a 16384^2 matrix

Chapter 11

Conclusion

k-Wave has up until now used FFTW and its 1D decomposition to compute FFTs. Unfortunatelly 1D decomposition has become a limiting factor, mainly due to the maximum number of employable cores. Therefore we introduced a new approach, called HybridFFT, built mainly on FFTW, MPI, OpenMP and the 2D decomposition principle. HybridFFT combines advantages of both distributed and shared memory. Unlike other 2D decomposition libraries, such as PFFT or P3DFFT, it needs only one global MPI transposition and 1D FFTs and local transpositions are done in shared memory. Cores on one socket communicate via shared memory (not via unefficient MPI messages) and due to the lower number of MPI processes, MPI messages are bigger, there are fewer of them sent and therefore the network does not get so flooded as with the pure MPI approaches.

HybridFFT has three main parts: series of 1D FFTs, local transpositions and global transpositions. 1D FFTs are calculated by FFTW and accelerated by OpenMP threads. For local transpositions, we implemented a block matrix transposition algorithm, vectorized by the SSE and AVX instruction extensions via Intel Intrinsic functions and accelerated by OpenMP threads. The global transposition is done by FFTW, which can shorten the execution time by planning the transposition beforehand, where it tries to merge smaller MPI messages to one bigger message and optimize the overall communication.

We measured the execution times on two clusters, Zapat and Anselm, both built on the modern Sandy Bridge architecture with 16 cores per node and 40 Gbit/s Infinibands. The results showed that HybridFFT performs very similarly or slightly faster than the other libraries. The linear scalability was proved for up to 2048 cores in the case of bigger matrices. The time distribution test showed that most of the total execution (about 90–95% in the case of big matrices) time needed to compute a FFT is taken by the global MPI transposition. Therefore we made the conclusion that it is not the performance per core that matters the most, but rather the fast interconnecting network.

It would be very interresting to use low power processors, such as ARM or Intel Atom, instead of the modern powerfull CPUs. Because the heat production is much lower with these processors, we could put more of them into one case and have around 1024 cores in one rack. Because of the global communication taking so much time, the computation could easily overlap with the communication and the performance/power consumption ratio could improve drastically. This is hopefully going to be the main subject of my Ph.D. studies.

For future improvements, we could use non-blocking MPI communications and overlap the communication steps with the computation steps. Since the global transposition is so time consuming, we could also further optimize the algorithms used by FFTW or develop new ones.

I attended PRACE Spring school 2014 held in Hagenberg, Austria, and actively presented a poster with the results of my work. The poster is attached at the end of this thesis, Appendix A.

Bibliography

[1] Anselm cluster, Ostrava, Czech Republic. Available at https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview.

[2] Cresta, the exascale project. Available at http://cresta-project.eu/.

[3] JuGene: Julich BlueGene/P. Available at http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUGENE/JUGENE node.html.

[4] JuQueen: Julich BlueGene/Q. Available at http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN node.html.

[5] JuRoPa: Julich Research on Petaflop Architectures. Available at http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/JUROPA node.html.

[6] Parallel hdf5 software library. Available at http://www.hdfgroup.org/HDF5/.

[7] PBS home page, accessed 24 march 2014. Available at http://pbs.mrj.com/main.html.

[8] Zapat cluster, Jihlava, Czech Republic. Available at http://www.cerit-sc.cz/cs/Hardware/#clust4.

[9] SuperMUC Petascale System, Leibniz Supercomputing Center, Germany, accessed 23 May 2014; last updated 22 May 2014. Homepage available at http://www.lrz.de/services/compute/supermuc/.

[10] OpenMP Application Program Interface, Version 4.0, accessed 4 May 2014; last updated July 2013. Available at http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[11] Intel Compilers, accesses 2 May 2014. Available at https://software.intel.com/en-us/intel-compilers.

[12] GCC, the GNU Compiler Collection, accesses 2 May 2014; last modified 22 March 2014. Available at https://gcc.gnu.org/.

[13] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorov. A Case for NUMA-aware Contention Management on Multicore Systems. Simon Fraser University. Retrieved 2014-01-27. Available at https://www.usenix.org/legacy/event/atc11/tech/final_files/

Blagodurov.pdf.

[14] Jan Cernocky. Study materials for Signals and Systems course. Brno University of Technology, Faculty of Information Technology, Czech Republic, updated May 2014.

[15] B. T. Cox, S. R. Arridge, and P. C. Beard. Estimating chromophore distributions from multiwavelength photoacoustic images. *J. Opt. Soc. Am. A 26(2)*, pages 443–455, 2009.

[16] Vaclav Dvorak. Study materials for Parallel System Architecture and Programming. Brno University of Technology, Faculty of Information Technology, Czech Republic, updated May 2014.

[17] Matteo Frigo and Steven G. Johnson. *The FFTW documentation v.3.3.4.*, accessed 4 May 2014; last updated April 2014. Available at http://www.fftw.org/fftw3_doc/.

[18] Matteo Frigo and Steven G. Johnson. The benchfft benchmark, accessed 6 May 2014; last updated April 2014. Available at http://www.fftw.org/benchfft/.

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. Available at http://www.open-mpi.org/papers/euro-pvmmpi-2004-overview/euro-pvmmpi-2004-overview.pdf.

[20] Milind Girkar. Intel Instruction Set Architecture Extensions. Intel Developer Zone. Available at https://software.intel.com/en-us/intel-isa-extensions.

[21] W.F. Hartsell, C.B. Scott, D.W. Bruner, et al. Randomized trial of short-versus long-course radiotherapy. *JNCI:Natl. Cancer Inst.*, 2005;97(11):798–804.

[22] Kai Hwang. Advanced Computer Architecture. ISBN 0-07-113342-9.

[23] J.E. Kennedy, G.R. ter Haar, and D. Cranston. High intensity focused ultrasound: surgery of the future? *Brit J. Radiol.*, 2003;76(909):590–599.

[24] J. Laufer, E. Zhang, G. Raivich, and P. Beard. Three-dimensional noninvasive imaging of the vasculature in the mouse brain using a high-resolution photoacoustic scanner,. *Appl. Opt. 48(10)*, pages D299–D306, 2009.

[25] J. G. Laufer, D. Delpy, C. Elwell, and P. C. Beard. Quantitative spatially resolved measurement of tissue chromophore concentrations using photoacoustic spectroscopy: application to the measurement of blood oxygenation and haemoglobin concentration. *Phys. Med. Biol. 52(1)*, pages 141–168, 2007.

[26] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. ACM. Retrieved 2014-01-27. Available at http://people.inf.ethz.ch/zmajo/publications/11-systor.pdf.

[27] Manchanda N. and Anand K. Non-Uniform Memory Access (NUMA). New York University. Retrieved 2014-01-27. Available at http://cs.nyu.edu/Ĩerner/spring10/projects/NUMA.pdf.

[28] The Open MPI Development Team. *Open MPI documentation*, accessed 1 May 2014; last updated 23 April 2014. Available at http://www.open-mpi.org/doc/current/.

[29] A. A. Oraevsky, L. V. Wang, and Ed. Optoacoustic tomography of the breast in Photoacoustic Imaging and Spectroscopy. *CRC Press, London*, pages 411–429, 2009.

[30] Dmitry Pekurovsky. P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, 2012. Available at http://www.sdsc.edu/us/resources/p3dfft.php.

[31] Michael Pippig. PFFT: An extension of FFTW to Massively Parallel Architectures. *SIAM J. SCI. COMPUT.*, 35(3):C213–C236, 2013.

[32] Roland Schulz. 3D FFT with 2D decomposition. Technical report, 27 April 2008.

[33] Daisuke Takahashi. FFTE software library. Available at http://www.ffte.jp/.

[34] B. E. Treeby and B. T. Cox. k-Wave: Matlab toolbox for the simulation and reconstruction of photoacoustic wave-fields. *J. Biomed. Opt.*, 15(2):021314, 2010.

[35] B. E. Treeby and B. T. Cox. Homepage of the k-Wave project, accessed 18 May 2014; last updated 13 November 2012. Available at http://www.k-wave.org/.

[36] B. E. Treeby, B. T. Cox, and J. Jaros. k-Wave User Manual version 1.0.1 (november 15, 2012). Available at http://www.k-wave.org/manual/k-wave_user_manual_1.0.1.pdf.

[37] B. E. Treeby, J. Jaros, A. P. Rendell, and B. T. Cox. Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a k-space pseudospectral method. *J. Acoust. Soc. Am.*, 131(6):4324–4336, 2012.

[38] L. V. Wang. Microwave-induced acoustic (thermoacoustic) tomography in Photoacoustic Imaging and Spectroscopy. *CRC Press, London*, pages 339–347, 2009.

[39] L. V. Wang and Ed. Photoacoustic Imaging and Spectroscopy. *CRC London*, 2009.

[40] X. Wang, Y. Pang, G. Ku, X. Xie, G. Stocia, and L. V. Wang. Noninvasive laser-induced photoacoustic tomography for structural and functional in vivo imaging of the brain,. *Nat. Biotechnol. 21(7)*, pages 803–806, 2003.

[41] E. Z. Zhang, J. G. Laufer, R. B. Pedley, and P. C. Beard. In vivo high-resolution 3d photoacoustic imaging of superficial vascular anatomy. *Phys. Med. Biol. 54(4)*, pages 1035–1046, 2009.

Appendix A

Poster