The tensor product of bulk synchronous parallel algorithms

Master thesis

45 ECTS

Universiteit Utrecht

Thomas Koopman

Student number: 5885701

Supervisor: Prof. Dr. Rob H. Bisseling

Second reader: Dr. Palina Salanevich

Advisor: Dr. Jaap van Oosten

January 19, 2022

Abstract

A Bulk Synchronous Parallel (BSP) algorithm is a type of parallel algorithm where communication and computation is separated. We present a way to generalise BSP algorithms for linear functions to a BSP algorithm for the tensor product of linear functions. This is applied to the discrete Fourier transform in higher dimensions, yielding a novel parallel algorithm.

Contents

Chapter 1

Introduction

In this thesis we present a parallel algorithm for calculating a linear function called the discrete Fourier transform in higher dimensions. In doing so, we develop a technique to transform a set of parallel algorithms (satisfying some requirements) for linear functions $f_1, \cdots, f_d$ into a parallel algorithm for their tensor product $f_1 \otimes \cdots \otimes f_d$.

1.1 Discrete Fourier transform

The one-dimensional discrete Fourier transform F_n of length n is a function that takes an array x of n complex numbers, and returns an array y of n complex numbers. We write ω_n for the nth root of unity $e^{-2\pi i/n}$. The discrete Fourier transform is given by

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}.$$

We also have a higher-dimensional variant, which takes $n_1 \times \cdots \times n_d$ multidimensional arrays of complex numbers as both input and output. This one is given by

$$Y[k_1, \cdots, k_d] = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, \cdots, j_d] \omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d}.$$

This thesis will focus on developing a parallel algorithm for the discrete Fourier transform (DFT) in arbitrary dimension. Sometimes the abbreviation FFT for Fast Fourier transform is also used instead of DFT, although this technically refers to $O(n \log n)$ algorithms for the DFT, and not the function itself.

1.2 BSP model

The Bulk Synchronous Parallel (BSP) model comprises a parallel computer architecture, a class of parallel algorithms, and a function for charging costs to algorithms [10]. In this thesis we use the variant by [2].

Definition 1 (The BSP model [2])**.**

- A *BSP computer* consists of a collection of processors, each with private memory, and a communication network that allows processors to access memories of other processors. Each processor can read from or write to every memory cell in the entire machine. The access time for all non-local memories is assumed to be the same.

- *A BSP algorithm* consists of a sequence of supersteps. A superstep contains either a number of computation steps or a number of communication steps, or in certain cases both, followed by a global barrier synchronisation. In a computation superstep, each processor performs a sequence of operations on local data. In a communication superstep, each processor sends and receives a number of messages. In a mixed superstep, both computation and communication take place. At the end of a superstep, all processors synchronise, as follows. Each processor checks whether it has finished all its tasks of that superstep. Processors wait until all others have finished. When this has happened, they all proceed to the next superstep. This form of synchronisation is called bulk synchronisation, hence the name.
- Finally, we have a *cost function* associated to a BSP algorithm. An h-relation is a communication superstep where each processor sends at most h data words to other processors and receives at most h data words, and where at least one processor sends or receives h words. A data word is a basic data type such as a real, integer, or complex number. We denote the maximum number of words sent by any processor by h_s and the maximum number received by h_r. Therefore,

$$h = \max\{h_s, h_r\}.$$

  This equation reflects the assumption that a processor can send and receive data simultaneously. The cost of the superstep depends solely on h. Note that two different communication patterns may have the same h, so that the cost function of the BSP model does not distinguish between them. The cost of an h-relation is

$$T_{comm}(h) = hg + l,$$

  where g and l are machine-dependent parameters and the time unit is the time of one floating point operation. The cost of a computation superstep is

$$T_{comp}(w) = w + l,$$

  where the amount of work w is defined as the maximum number of flops performed in the superstep by any processor. The cost function is now the sum of the cost of the communication supersteps and computation supersteps.

Remark 2. The BSP model was developed in the 1980s. The common Intel 8086 processor of that time had a clock speed of 5 to 10 MHz. CPUs of today have clock speeds of around 3 GHz, so almost a thousand times faster. However, data transfer rate has not kept up. This was mitigated in part by introducing small and fast on-chip memory called cache, but when data is not present there, we still have to wait a long time compared to CPU operations. This discrepancy is made worse by S(ingle) I(nstruction) M(ultiple) D(ata) parallelism, which allows an instruction to be performed on multiple data elements at the same time. Also modern CPUs are pipelined, allowing the execution units on a CPU to simultaneously do their job. For example it could perform a loop counter update, floating-point operation in the loop, and load operation at the same time.

The upshot of this, is that flops are not necessarily the limiting factor in a sequential program, especially if the number of flops per data element is low (and this is the case in the DFT, the complexity is $O(n\log(n))$). Instead, it will be limited by the bandwidth, so the speed at which we can transfer data from RAM to the CPU chip. This is called the Von Neumann bottleneck [1]. In multicore processors this bottleneck is even more of a problem because the cores share the same bus to RAM. That is why we will give w in big O notation in this thesis.

1.3 Thesis structure

The goal of this thesis is to prove and implement the d-dimensional DFT, Algorithm 4 from Chapter 3, Section 2. There are two routes one may take, depending on the background and interest from the reader. If the reader has no background and interest in abstract algebra, but is interested in the computational aspect and DFT specifically, they may skip Chapter 2 and proceed to Chapter 3 of this thesis, where a direct proof is given, as well as chapters 4 and 5. Chapter 4 contains details about the implementation, and Chapter 5 reports on the results.

If the reader does know category theory, or is willing to learn, they may read chapters 2 and 6, and can skip the proof of Chapter 3, Section 2. The prerequisites are categories, functors, natural transformations, (co)limits and adjoint functors. This is covered in chapters 1, 2 and 5 of [6], and we also need Theorem 6.3.1. We only use adjointness to show a certain functor has a certain property (preservation of colimits), so if you are willing to take my word for that, it suffices to read chapters 2 and 3.

Chapter 2 contains some algebra that is not very difficult, but may not be covered in a standard maths curriculum, and Chapter 5 contains the abstract technique of proving Algorithm 4. This can also be applied to generalise other BSP-style parallel algorithms for the DFT and related transforms to higher dimensions.

1.4 Literature

To the best of my knowledge, all the parallel DFT libraries use that the sum describing the DFT factorises as follows

$$\sum_{j_1=0}^{n_1-1} \cdots \sum_{j_2=0}^{n_2-1} X[j_1, \cdots, j_d]\omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d} = \sum_{j_1=0}^{n_1-1} \left(\sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, \cdots, j_d]\omega_{n_2}^{j_2 k_2} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d} \right) \cdot \omega_{n_1}^{j_1 k_1}.$$

We see that the d-dimensional Fourier transform can be calculated in two steps: we apply a $(d-1)$-dimensional DFT to $X[j_1, -, \cdots, -]$ for all j_1, calling the result Y (usually this is done in place, so using the same data structure for X and Y). Then we transform $Y[-, k_2, \cdots, k_d]$ by a one-dimensional DFT for all tuples $(k_2, \cdots, k_d)$. This can be applied inductively to calculate a d-dimensional DFT by doing one-dimensional transforms along each dimension.

This is then parallelised as we can see in the following figures from [3]. Implementations such as FFTW [4] take a block distribution in one dimension, see Figure 1.1, such that the DFTs in every direction but the distributed one are local. It is then redistributed so the last direction becomes local. This means we get one communication step, or two if you want to end up in the same distribution again. Such as communication step is called a transpose because we transpose the array along an axis. If the number of processors divides the size of the array in the distributed dimensions, the communication and computation is balanced. A limitation of this method for high-dimensional arrays, say d-dimensional and let us assume for simplicity that the shape is $n \times \cdots \times n$, is that the number of processors is bounded by n for a $O(n^d \log n^d)$ algorithm. We can use more processors by distributing among more dimensions, see Figure 1.2. This may require more communication steps though, two in that figure. If the problem is d-dimensional and the processor distribution is l-dimensional, we can do the DFTs locally in $d - l$ dimensions per such distribution. So it takes $\lceil \frac{d}{d-l} \rceil - 1$ communication steps, or $\lceil \frac{d}{d-l} \rceil$ if we want to end up in the same distribution. This is implemented in PFFT [9] and mpi4pyfft [3].

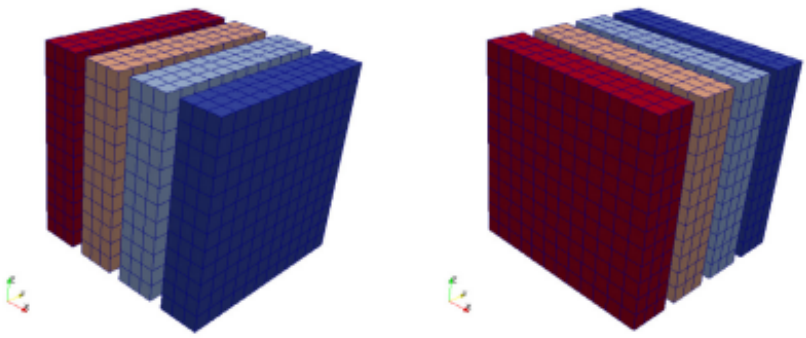


Figure 1.1: Distributing one dimension over four processors, indicated by colour. The vertical one-dimensional DFTs and the horizontal DFTs from the rib of the cube facing us, towards the right, are local. The one-dimensional DFTs going from the left side of the cube towards the rib facing us, are not local as the elements have different colours (and so belong to different processors). This requires a redistribution to the right-hand figure. This is called a transpose in the literature, because it looks like the cube was transposed on its axis. This makes the remaining DFTs from the left to the side facing us local. Note that we end in a different distribution than we started in. Source [3]

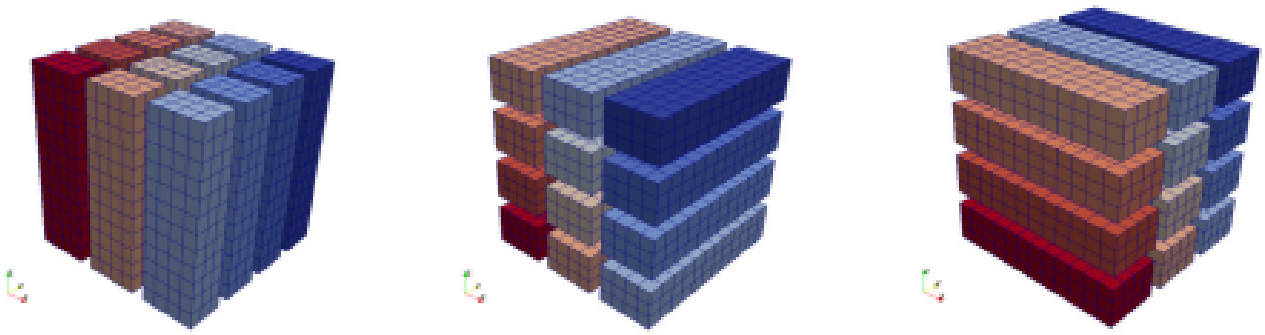


Figure 1.2: Distributing two dimensions over 12 processors, indicated by colours. Now the one-dimensional DFTs are local in only one direction. In the first figure, the vertical ones are local, in the second one, the DFTs from the rib facing us, towards the right, and in the last figure the DFTs from the left side towards the rib facing us. Source [3]

Chapter 2

Tensor product

In this chapter we explore the tensor product. The lecture notes [5] contain proofs for all statements concerning tensor products made here. The lecture notes work with modules, but taking $R = \mathbb{C}$ turns them into complex vector spaces. After that we see how the free functor maps the Cartesian product to the tensor product.

2.1 Prerequisites and notation

We denote the product of a collection of objects $\{X_s \mid s \in S\}$ in Set with $\left(\prod_{s\in S} X_s, \pi\right)$ and the coproduct with $\left(\coprod_{s\in S} X_s, \iota\right)$. For a cone $f_s : Z \to X_s$ we denote the unique morphism of cones into the product with $\langle f_s \mid s \in S\rangle$ and for cocones we write $[f_s \mid s \in S]$. In Set (and any category $\mathcal{C}$ with products), we can extend the product to a functor $\prod_{s\in S} \mathcal{C} \to \mathcal{C}$ by mapping $(f_s : X_s \to Y_s)$ to $\langle f_s \circ \pi_s \mid s \in S\rangle$, i.e. the unique morphism $\prod_{s\in S} f_s : \prod_{s\in S} X_s \to \prod_{s\in S} Y_s$ such that the following diagrams commute for all s.

$$\begin{array}{ccc} X_s & \xleftarrow{\pi_{X_s}} & \prod_{s\in S} X_s \\ {\scriptstyle f_s}\downarrow & & \vdots{\scriptstyle \prod_{s\in S} f_s} \\ Y_s & \xleftarrow[\pi_{Y_s}]{} & \prod_{s\in S} Y_s \end{array}$$

We may also write $f_1 \times \cdots \times f_d$ instead of $\prod_{s\in\{1,\cdots,d\}} f_l$.

In Vect, the category of vector spaces, the coproduct is often called the direct sum. Notation: $\bigoplus_{s\in S} V_s$. The finite coproduct is the Cartesian product of the underlying sets of $(V_s)_s$ with the canonical linear structure. The projections are inherited from Set, and the inclusions are given by $\iota_s : b \mapsto (0, \cdots, 0, b, 0, \cdots, 0)$ with b on the sth place. Both the finite product functor and its dual act the same on morphisms (namely $(v_s)_s \mapsto (f_s(v_s))_s$), so we denote it by $\bigoplus_{s\in S} f_s$.

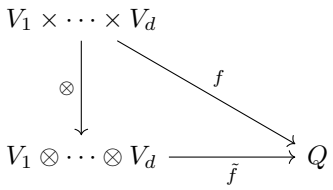
2.2 Tensor products in Vect

2.2.1 Basics

The tensor product is connected to the idea of a multilinear map, that is: a map $V_1 \times \cdots \times V_d \to Q$ linear in each of its arguments.

Definition 3 (Tensor product)**.** The tensor product of vector spaces $V_1, \cdots, V_d$ is a vector space $V_1 \otimes \cdots \otimes V_d$ together with a multilinear map $\otimes : V_1 \times \cdots \times V_d \to V_1 \otimes \cdots \otimes V_d$ that satisfies the following property: any multilinear map $f : V_1 \times \cdots \times V_d \to Q$ factors uniquely through $\otimes$.

More formally, there exists a unique linear map $\tilde{f} : V_1 \otimes \cdots \otimes V_d \to Q$ such that $f = \tilde{f} \circ \otimes$.



(This diagram commutes on the level of sets.)

For vectors $v_l \in V_l$ we write $v_1 \otimes \cdots \otimes v_d$ instead of $\otimes(v_1, \cdots, v_d)$. An element of a tensor product is called a tensor, and an element in the image of $\otimes$ is called a pure tensor.

It turns out that such an object exists [5, Proposition 2.0.2.] and is unique up to isomorphism [5, Proposition 2.0.1.].

We can build a basis of the tensor product straightforwardly out of bases of its components. For that reason we refer to it as the standard basis of the tensor product.

Definition 4 (Standard basis)**.** For some chosen bases $\mathcal{B}_l$ for V_l, we call $\{b_1 \otimes \cdots \otimes b_d \mid b_l \in \mathcal{B}_l,\ 1 \leq l \leq d\}$ the standard basis of $V_1 \otimes \cdots \otimes V_d$.

Proof. All vector spaces are free modules, and if inclusion $i : X \to M$ generates $\mathbb{C}$-module M, then X is a basis of M. So for $d = 2$ linear indepence follows from (the proof of) [5, Proposition 27.11]. If $d > 2$, we apply [5, Proposition 27.11] repeatedly with associativity [5, Proposition 27.17]. □

Proposition 1 ([5, Section 4])**.** *Given linear maps $(f_l : V_l \to W_l)_{l=1}^d$, there exists a unique linear map $f_1 \otimes \cdots \otimes f_d : V_1 \otimes \cdots \otimes V_d \to W_1 \otimes \cdots \otimes W_d$ which takes $v_1 \otimes \cdots \otimes v_d$ to $f_1(v_1) \otimes \cdots \otimes f_d(v_d)$.*

Let us now check that this turns the tensor product into a functor.

Proposition 2. *The tensor product is a functor.*

Proof. By the previous characterisation $\mathrm{id}_{V_1} \otimes \cdots \otimes \mathrm{id}_{V_d}$ sends $e^1_{j_1} \otimes \cdots \otimes e^d_{j_d}$ to $\mathrm{id}_{V_1}(e^1_{j_1}) \otimes \cdots \otimes \mathrm{id}_{V_d}(e^d_{j_d}) = e^1_{j_1} \otimes \cdots \otimes e^d_{j_d}$, so restricted to the standard basis of $V_1 \otimes \cdots \otimes V_d$, the map $\mathrm{id}_{V_1} \otimes \cdots \otimes \mathrm{id}_{V_d}$ is the identity. That means the non-restricted version is also the identity.

It remains to be shown that the tensor product respects composition. Given linear maps $(f_l : U_l \to V_l)_{l=1}^d$ and $(g_l : V_l \to W_l)_{l=1}^d$, we have that $(g_1 \otimes \cdots \otimes g_d) \circ (f_1 \otimes \cdots \otimes f_d)$ sends basis element $e^1_{j_1} \otimes \cdots \otimes e^d_{j_d}$ to $(g_1 \otimes \cdots \otimes g_d)(f(e^1_{j_1}) \otimes \cdots \otimes f(e^d_{j_d})) = (g_1 f_1)(e^1_{j_1}) \otimes \cdots \otimes (g_d f_d)(e^d_{j_d})$ by Proposition 1. But this is equal to $(g_1 f_1 \otimes \cdots \otimes g_d f_d)(e^1_{j_1} \otimes \cdots \otimes e^d_{j_d})$, also by Proposition 1. As the two maps coincide on basis elements, they must coincide everywhere.

We conclude that the tensor product is functorial. □

2.2.2 Free functor and Cartesian product

As you may already have noticed, the standard basis of $V_1 \otimes \cdots \otimes V_d$ looks suspiciously much like the product of the bases of $V_1, \cdots, V_d$. Let $\mathcal{B}_l$ be a basis of V_l and $F : \mathrm{Set} \to \mathrm{Vect}$ the free functor. Then $V_l = F\mathcal{B}_l$ and

$$\mathcal{B}_1 \times \cdots \times \mathcal{B}_d \to F\mathcal{B}_1 \otimes \cdots \otimes F\mathcal{B}_d$$

$$(b_1, \cdots, b_d) \mapsto b_1 \otimes \cdots \otimes b_d$$

linearly extends to an isomorphism $F(\mathcal{B}_1 \times \cdots \times \mathcal{B}_d) \cong F\mathcal{B}_1 \otimes \cdots \otimes F\mathcal{B}_d$.

Proposition 3. *These isomorphisms assemble into a natural isomorphism between the functors*

$$\begin{array}{ccc} (\mathcal{B}_1, \cdots, \mathcal{B}_d) & \longmapsto & F(\mathcal{B}_1 \times \cdots \times \mathcal{B}_d) \\ {\scriptstyle (f_1, \cdots, f_d)}\downarrow & \longmapsto & \downarrow{\scriptstyle F(f_1 \times \cdots \times f_d)} \\ (\mathcal{D}_1, \cdots, \mathcal{D}_d) & \longmapsto & F(\mathcal{D}_1 \times \cdots \times \mathcal{D}_d) \end{array}$$

and

$$\begin{array}{ccc} (\mathcal{B}_1, \cdots, \mathcal{B}_d) & \longmapsto & F\mathcal{B}_1 \otimes \cdots \otimes F\mathcal{B}_d \\ {\scriptstyle (f_1, \cdots, f_d)}\downarrow & \longmapsto & \downarrow{\scriptstyle Ff_1 \otimes \cdots \otimes Ff_d} \\ (\mathcal{D}_1, \cdots, \mathcal{D}_d) & \longmapsto & F\mathcal{D}_1 \otimes \cdots \otimes F\mathcal{D}_d \end{array}$$

from Set $\times \cdots \times$ Set to Vect.

Proof. The naturality square is

$$\begin{array}{ccc} F(\mathcal{B}_1 \times \cdots \times \mathcal{B}_d) & \xrightarrow{\cong} & F\mathcal{B}_1 \otimes \cdots \otimes F\mathcal{B}_d \\ \downarrow{\scriptstyle F(f_1 \times \cdots \times f_d)} & & \downarrow{\scriptstyle Ff_1 \otimes \cdots \otimes Ff_d} \\ F(\mathcal{D}_1 \times \cdots \times \mathcal{D}_d) & \xrightarrow{\cong} & F\mathcal{D}_1 \otimes \cdots \otimes F\mathcal{D}_d \end{array}$$

It suffices to check that this commutes on basis elements because all maps are linear. Going down and then right we get

$$(b_1, \cdots, b_d) \mapsto (f_1(b_1), \cdots, f_d(b_d)) \mapsto f_1(b_1) \otimes \cdots \otimes f_d(b_d).$$

Going right and down we get the same

$$(b_1, \cdots, b_d) \mapsto b_1 \otimes \cdots \otimes b_d \mapsto f_1(b_1) \otimes \cdots \otimes f_d(b_d).$$

Hence the naturality square commutes. The components of the natural transformation are obviously isomorphisms, so it is a natural isomorphism. □

Remark 5. The existence of this natural isomorphism is not a coincidence. There is a categorical notion of the tensor product in arbitrary categories, which is a functor acting like multiplication in a monoid. In Set the product functor already discussed can be chosen as a tensor product. The free functor is a strong monoidal functor, which means it preserves all of this structure. The categories Vect and Set together with the (tensor) product and coproduct form a distributive category, which means that the coproduct distributes over the (tensor) product like summation does over multiplication in a ring as we will see later. Developing this theory is not necessary for this thesis, but if you are interested you can look at [7], [8] for more information.

2.3 Multidimensional DFT as tensor product

We can view the DFT F_n from the introduction as a linear function $\mathbb{C}^n \to \mathbb{C}^n$. Let $(e_0^l, \cdots, e_{n_l-1}^l)$ be a basis of $\mathbb{C}^{n_l}$. Interpreting $X[j_1, \cdots, j_d]$ as the coefficient in front of $e_{j_1}^1 \otimes \cdots \otimes e_{j_d}^1$, we can view the d-dimensional DFT as a linear map from $\mathbb{C}^{n_1} \otimes \cdots \otimes \mathbb{C}^{n_d}$ to itself. From now on, we suppress superscripts on the basis elements, trusting the position in the tensor to tell us which copy of $\mathbb{C}^\bullet$ it belongs to.

The following statement without proof can be found in [11, p.148]. As a quick side note: van Loan uses $\otimes$ to denote the Kronecker product instead of the tensor product. It boils down to the same thing though, because this is the matrix representation of the tensor product of maps, with respect to some ordering on the standard basis of the tensor product.

Proposition 4. *The d-dimensional DFT is the tensor product of one-dimensional DFTs.*

Proof. As $\otimes$ is multilinear, addition distributes over it like in a monoid: $e \otimes (a + b) = e \otimes a + e \otimes b$. This (together with induction) gives the second equality of the following calculation.

$$(F_{n_1} \otimes \cdots \otimes F_{n_d})(e_{j_1} \otimes \cdots \otimes e_{j_d}) =$$

$$\left(\sum_{k_1=0}^{n_1-1} \omega_{n_1}^{j_1 k_1} e_{k_1}\right) \otimes \cdots \otimes \left(\sum_{k_d=0}^{n_d-1} \omega_{n_d}^{j_d k_d} e_{k_d}\right) =$$

$$\sum_{k_1=0}^{n_1-1} \cdots \sum_{k_d=0}^{n_d-1} \omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d} (e_{k_1} \otimes \cdots \otimes e_{k_d}).$$

By linearity we have that $X[j_1, \cdots, j_d](e_{j_1} \otimes \cdots \otimes e_{j_d})$ gets mapped to

$$\sum_{k_1=0}^{n_1-1} \cdots \sum_{k_d=0}^{n_d-1} X[j_1, \cdots, j_d] \omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d} (e_{k_1} \otimes \cdots \otimes e_{k_d}).$$

So writing a general element

$$x = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, \cdots, j_d](e_{j_1} \otimes \cdots \otimes e_{j_d})$$

as a sum of basis elements, we have that each summand

$$X[j_1, \cdots, j_d](e_{j_1} \otimes \cdots \otimes e_{j_d})$$

contributes

$$X[j_1, \cdots, j_d]\omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d}$$

to the coefficient in front of $(e_{k_1} \otimes \cdots \otimes e_{k_d})$ in the output. Writing Y for the representation of $(F_{n_1} \otimes \cdots \otimes F_{n_d})(x)$, we conclude by linearity that

$$Y[k_1, \cdots, k_d] = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} X[j_1, \cdots, j_d]\omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d}.$$

□

2.4 Distributivity of the coproduct

Here we explore the promised distributivity of the coproduct over the tensor product. We see that the names direct **sum** and tensor **product** are aptly chosen!

Proposition 5. *Suppose we have collections $\{A_s^l \mid s \in S_l\}$ of sets for $l = 1, \cdots, d$. Then there exists an isomorphism*

$$\prod_{l=1}^{d} \coprod_{s \in S_l} A_{s_l}^l \cong \coprod_{s \in S_1 \times \cdots \times S_d} A_{s_1}^1 \times \cdots \times A_{s_d}^d$$

such that

$$\begin{array}{ccc} \prod_{l=1}^d \coprod_{s\in S_l} A_s^l & \xrightarrow{\cong} & \coprod_{s\in S_1\times\cdots\times S_d} A_{s_1}^1 \times \cdots \times A_{s_d}^d \\ \Big\downarrow {\scriptstyle \prod_{l=1}^d [f_s^l \mid s\in S_l]} & \swarrow {\scriptstyle [f_{s_1}^1 \times\cdots\times f_{s_d}^d \mid s\in S_1\times\cdots\times S_d]} & \\ \prod_{l=1}^d C^l & & \end{array}$$

commutes for any collection of maps $\{f_s^l : A_s^l \to C^l \mid l = 1, \cdots, d, s \in S_l\}$.

Proof. We construct the product the usual way and the coproduct $\coprod_{s\in S} X_s$ by $\{(s, x) \mid s \in S, x \in X_s\}$. Now we can give the isomorphism explicitly by

$$((s_1, a_1), \cdots, (s_d, a_d)) \mapsto ((s_1, \cdots, s_d), (a_1, \cdots, a_d)).$$

To see that the diagram commutes, simply note that going right and then down yields

$$((s_1, a_1), \cdots, (s_d, a_d)) \mapsto ((s_1, \cdots, s_d), (a_1, \cdots, a_d)) \mapsto (f_{s_1}^1(a_1), \cdots, f_{s_d}^d(a_d))$$

which is what we get when we go down immediately. □

Remark 6. This holds for any construction of products and coproducts, which can be proven by using that $(-) \times X$ preserves coproducts (it is left adjoint to $(-)^X$) inductively. However, the proof becomes more complicated and there is nothing wrong with this construction.

Remark 7. Denote the isomorphism from Proposition 5 as σ. As the free functor $F : \mathrm{Set} \to \mathrm{Vect}$ is left adjoint to the forgetful functor $U : \mathrm{Vect} \to \mathrm{Set}$, the free functor preserves colimits from Set to Vect. For that reason we may construct the coproduct/direct sum $\bigoplus$ in Vect as the free functor of the coproduct in Set, which we will do from now on.

Proposition 6. *For any collection of linear maps $f^l_{s_l} : FA^l_{s_l} \to FB^l_{s_l}$ the following diagram commutes. (Using the previous remark, so $\bigoplus_{s_l \in S_l} FA^l_{s_l} = F\left(\coprod_{s_l \in S_l} A^l_{s_l}\right)$.)*

$$\begin{array}{ccc}
\bigotimes_{l=1}^{d} \bigoplus_{s_l \in S_l} FA^l_{s_l} & \xrightarrow{\bigotimes_{l=1}^{d} \bigoplus_{s \in S_l} f^l_s} & \bigotimes_{l=1}^{d} \bigoplus_{s_l \in S_l} FB^l_{s_l} \\
\downarrow \cong & & \downarrow \cong \\
F\left(\prod_{l=1}^{d} \coprod_{s_l \in S_l} A^l_{s_l}\right) & & F\left(\prod_{l=1}^{d} \coprod_{s_l \in S_l} B^l_{s_l}\right) \\
\downarrow F\sigma & & \downarrow F\sigma \\
F\left(\coprod_{s \in S_1 \times \cdots \times S_d} A^1_{s_1} \times \cdots \times A^d_{s_d}\right) & & F\left(\coprod_{s \in S_1 \times \cdots \times S_d} B^1_{s_1} \times \cdots \times B^d_{s_d}\right) \\
\downarrow \cong & & \downarrow \cong \\
\bigoplus_{s \in S_1 \times \cdots \times S_d} (FA^1_{s_1} \otimes \cdots \otimes FA^d_{s_d}) & \xrightarrow{\bigoplus_{s \in S_1 \times \cdots \times S_d} (f^1_{s_1} \otimes \cdots \otimes f^d_{s_d})} & \bigoplus_{s \in S_1 \times \cdots \times S_d} (FB^1_{s_1} \otimes \cdots \otimes FB^d_{s_d})
\end{array}$$

where the first and last downwards isomorphisms come from proposition 3.

Proof. Take a basis element $(s_1, a_1) \otimes \cdots \otimes (s_d, a_d)$. Going down and then to the right corresponds to

$$(s_1, a_1) \otimes \cdots \otimes (s_d, a_d) \mapsto ((s_1, a_1), \cdots, (s_d, a_d)) \mapsto ((s_1, \cdots, s_d), (a_1, \cdots, a_d))$$

$$\mapsto ((s_1, \cdots, s_d), a_1 \otimes \cdots \otimes a_d) \mapsto ((s_1, \cdots, s_d), f^1_{s_1}(a_1) \otimes \cdots \otimes f^d_{s_d}(a_d)).$$

Going right and then down corresponds to

$$(s_1, a_1) \otimes \cdots \otimes (s_d, a_d) \mapsto (s_1, f^1_{s_1}(a_1)) \otimes \cdots \otimes (s_d, f^d_{s_d}(a_d)) \mapsto (s_1, f^1_{s_1}(a_1)), \cdots, (s_d, f^d_{s_d}(a_d)) \mapsto$$

$$((s_1, \cdots, s_d), (f^1_{s_1}(a_1), \cdots, f^d_{s_d}(a_d))) \mapsto ((s_1, \cdots, s_d), f^1_{s_1}(a_1) \otimes \cdots \otimes f^d_{s_d}(a_d)).$$

These are equal, so the diagram commutes on basis elements. By linearly extending we see that it commutes on any element. □

Remark 8. For any assignment of bases to vector spaces, the downward isomorphisms of the previous proposition assemble into a natural isomorphism between the tensor product of the direct sum, and the direct sum of the tensor product:

$$\left(\bigoplus_{s_1 \in S_1} V^1_{s_1}\right) \otimes \cdots \otimes \left(\bigoplus_{s_d \in S_d} V^d_{s_d}\right) \cong \bigoplus_{s \in S_1 \times \cdots \times S_d} V^1_{s_1} \otimes \cdots \otimes V^d_{s_d}.$$

The naturality square is the diagram from the previous proposition. Of course we can see the existence of such a natural isomorphism immediately by repeatedly using that the tensor product preserves colimits (being left adjoint to the hom functor), but we need the explicit natural isomorphism in chapter 6. Note the similarity between the equality

$$\left(\sum_{s_1 \in S_1} x_{s_1}\right) \cdots \left(\sum_{s_d \in S_d} x_{s_d}\right) = \sum_{s \in S_1 \times \cdots \times S_d} x_{s_1} \cdots x_{s_d}$$

in a ring.

Chapter 3

Four-step framework

3.1 One-dimensional case

Suppose x is an array of length n, that $p|n$ and $p|\frac{n}{p}$ (or equivalently that $p^2|n$), and that we want to calculate $y := F_n(x)$. We write $v(a : b : c)$ to mean the strided subarray of v which starts at index a and has stride b. The last variable c is the length of v. If we write $v(a : b)$ we mean the subarray that starts at a and ends at b (inclusive).

3.1.1 Sequential algorithm

As any $0 \leq j < n$ can be written uniquely in the form $kp + s$ with $0 \leq k < n/p$, $0 \leq s < p$, we have

$$y_a = \sum_{j=0}^{n-1} x_j \omega_n^{ja} = \sum_{s=0}^{p-1} \sum_{k=0}^{n/p} x_{kp+s} \omega_n^{(kp+s)a} = \sum_{s=0}^{p-1} \omega_n^{sa} \sum_{k=0}^{n/p-1} x_{kp+s} (\omega_n^p)^{ka} = \sum_{s=0}^{p-1} \omega_n^{sa} \sum_{k=0}^{n/p-1} x_{kp+s} \omega_{n/p}^{ka}. \tag{3.1}$$

Write $x^{(s)}$ for $x(s : p : n)$ and $z^{(s)}$ for $F_{n/p}(x(s : p : n))$. We have correspondence $x_k^{(s)} = x_{kp+s}$ and $x_j = x_{j \operatorname{div} p}^{(j \bmod p)}$ and likewise for z, $z^{(s)}$. By periodicity we have $\omega_{n/p}^{ka} = \omega_{n/p}^{k(a \bmod (n/p))}$. Armed with this knowledge we can state

$$\sum_{k=0}^{n/p-1} x_{kp+s} \omega_{n/p}^{ka} = \sum_{k=0}^{n/p-1} x_k^{(s)} \omega_{n/p}^{k(a \bmod (n/p))}.$$

But this is the definition of the $a \bmod (n/p)$th entry of the DFT of $x^{(s)}$, so we can write equation (3.1) as

$$y_a = \sum_{s=0}^{p-1} \omega_n^{sa} z_{a \bmod (n/p)}^{(s)}.$$

This looks suspiciously much like a DFT of length p. In order to massage this into that form, we write a as $t(n/p) + k$ for $0 \leq k < n/p$, $0 \leq t < p$.

$$y_{t(n/p)+k} = \sum_{s=0}^{p-1} \omega_n^{stn/p+sk} z_k^{(s)} = \sum_{s=0}^{p-1} \omega_p^{st} \left(z_k^{(s)} \omega_n^{sk} \right).$$

So $y(k : n/p : n)$ is $F_p(w^{(k)})$ where $w_s^{(k)} = z_k^{(s)} \cdot \omega_n^{sk}$.

We can summarise this as Algorithm 1

Algorithm 1 Sequential four-step framework

Input: x : array of length n, number p such that $p^2 \mid n$.
Output: y : array of length n, such that $y = F_n(x)$.

```
1: for s := 0 to p − 1 do                                   ▷ Step 1
2:     x^(s) := x(s : p : n);
3:     z^(s) := F_{n/p}(x^(s));
4: for s := 0 to p − 1 do                                   ▷ Step 2
5:     for k := 0 to n/p − 1 do
6:         z_k^(s) = ω_n^{ks} z_k^(s);
7: for s := 0 to p − 1 do                                   ▷ Step 3
8:     for k := 0 to n/p do
9:         w_s^(k) = z_k^(s);
10: for k := 0 to n/p − 1 do                                ▷ Step 4
11:     y(k : n/p : n) = F_p(w^(k));
```

This is the four-step framework and can be found with a different proof and formulation in [11]. Step 2 is sometimes called twiddling in [11] and elsewhere. We can easily avoid using extra arrays z and w by reusing x, y which gives Algorithm 2. The only step which is not immediately obvious, is how we used y to store w. We do this because $\{y(k : n/p : n) \mid 0 \leq k < n/p\}$ splits up y into n/p strided subarrays of length p each, so we can use those to store the n/p arrays $w^{(k)}$ of length p.

Algorithm 2 Sequential four-step framework, less memory

Input: x : array of length n, number p such that $p^2 \mid n$.
Output: y : array of length n, such that $y = F_n(x)$.

```
1: for s := 0 to p − 1 do                                   ▷ Step 1
2:     x^(s) := x(s : p : n);
3:     x^(s) = F_{n/p}(x^(s));
4: for s := 0 to p − 1 do                                   ▷ Step 2
5:     for k := 0 to n/p − 1 do
6:         x_k^(s) = ω_n^{ks} x_k^(s);
7: for s := 0 to p − 1 do                                   ▷ Step 3
8:     for k := 0 to n/p do
9:         y(k : n/p : n)_s = x_k^(s);
10: for k := 0 to n/p − 1 do                                ▷ Step 4
11:     y(k : n/p : n) = F_p(y(k : n/p : n));
```

3.1.2 Parallel algorithm

We use the parallelisation strategy of [2] to turn the sequential algorithm in a parallel one.

Step 1 and 2 are trivial to parallelize, and they determine the distribution: namely cyclic over p processors. We would like to end in this distribution as well. Let $y^{(s)} := y(s : p : n)$ be the part of y stored on $P(s)$.

As $p|n/p$, we have that $y(k : n/p : n)_c = y_{c(n/p)+k}$ is stored on $P((c(n/p) + k) \bmod p) = P(k \bmod p)$ in local position $(c(n/p) + k) \operatorname{div} p = c(n/p^2) + k \operatorname{div} p$. So $y(k : n/p : n)$ is stored in its entirety on $P(k \bmod p)$.

Increasing c by one increases the local position by n/p^2 and $y(k : n/p : n)_0$ is stored in local position $k \operatorname{div} p$, so $y(k : n/p : n) = y^{(k \bmod p)}(k \operatorname{div} p : n/p^2 : n/p)$. As k ranges from 0 to $n/p - 1$, we have that $k \operatorname{div} p$ ranges from 0 to $n/p^2 - 1$. This parallelizes step 4, namely we locally transform $y^{(s)}(t : n/p^2 : n/p)$ by F_p for $0 \leq t < n/p^2$.

We implement step 3 as a communication step. As $y(k : n/p : n)_s = y^{(k \bmod p)}(s(n/p^2) + k \operatorname{div} p)$, we need to send $x_k^{(s)}$ to $P(k \bmod p)$ in local position $s(n/p^2) + k \operatorname{div} p$. So $x^{(s)}(k : p : n/p)$ ends up in $P(k)$ for $0 \leq k < p$. The first element corresponds to $y^{(k)}(sn/p^2)$, and every time we increase k by p, we increase $k \operatorname{div} p$ by one, so $x^{(s)}(k : p : n/p)$ corresponds to $y^{(k)}(sn/p^2 : (s+1)n/p^2 - 1)$.

We get parallel algorithm 3.

Algorithm 3 Parallel four-step framework for $P(s)$, in place

Input: x : array of length n, distr(x) = cyclic over p processors such that $p^2 | n$.
Output: y : array of length n, distr(y) = cyclic, such that $y = F_n(x)$.

1: $x^{(s)} = F_{n/p}(x^{(s)})$; ▷ Superstep (0)
2: **for** $k := 0$ **to** $n/p - 1$ **do**
3: $\quad x_k^{(s)} = \omega_n^{ks} x_k^{(s)}$;
4: **for** $k := 0$ **to** $p - 1$ **do** ▷ Superstep (1)
5: $\quad$ Put $x^{(s)}(k : p : n/p)$ in $P(k)$ as $y^{(k)}(sn/p^2 : (s+1)n/p^2 - 1)$;
6: **for** $t := 0$ **to** $n/p^2 - 1$ **do** ▷ Superstep (2)
7: $\quad y^{(s)}(t : n/p^2 : n/p) = F_p(y^{(s)}(t : n/p^2 : n/p))$;

We can also do this in-place by taking $y = x$.

3.2 Higher-dimensional parallel algorithm

We will prove Algorithm 4 directly. We use vector notation to abbreviate dimension-wise strides, subarrays, etc. So $X(t_1 : n_1/p_1^2 : n_1/p_1, \cdots, t_d : n_d/p_d^2 : n_d/p_d) =: X(t : n/p^2 : n/p)$. The distribution over a d-dimensional grid of $p_1 \times \cdots \times p_d$ processors is cyclic in each dimension. We call this dimension-wise cyclic. We use the notation $[n] := \{0, 1, 2, \cdots, n-1\}$.

Algorithm 4 Parallel four-step framework for processor $P(s)$ in d dimensions

Input: X : multidimensional array of size $n_1 \times \cdots \times n_d$, $X = X_0$. distr(x) = d-dimensional cyclic over $p_1 \cdots p_d$ processors such that $p_l^2 | n_l$.
Output: X : vector of length n, distr(x) = d-dimensional cyclic, such that $X = (F_{n_1} \otimes \cdots \otimes F_{n_d})(X_0)$.

1: $X^{(s)} = (F_{n_1/p_1} \otimes \cdots \otimes F_{n_d/p_d})(X^{(s)})$; ▷ Superstep (0)
2: **for** $k \in [n_1/p_1] \times \cdots \times [n_d/p_d]$ **do**
3: $\quad X^{(s)}[k_1, \cdots, k_d] = \prod_{l=1}^{d} \omega_{n_l}^{k_l s_l} X^{(s)}[k_1, \cdots, k_d]$;
4: **for** $k \in [p_1] \times \cdots \times [p_d]$ **do** ▷ Superstep (1)
5: $\quad$ Put $X^{(s)}(k : p : n/p)$ in $P(k)$ as subarray $X^{(k)}[sn/p^2 : (s+1)n/p^2 - 1]$;
6: **for** $t \in [n_1/p_1^2] \times \cdots \times [n_d/p_d^2]$ **do** ▷ Superstep (2)
7: $\quad X^{(s)}(t : n/p^2 : n/p) = (F_{p_1} \otimes \cdots \otimes F_{p_d}) \left(X^{(s)}(t : n/p^2 : n/p)\right)$;

We restate the algorithm and distinguish the array X at different stages as Algorithm 5 to make the proof easier. Warning: this is gross, I would not read it if you can understand the categorical proof of Chapter 6!

Proof. We have

Algorithm 5 Parallel four-step framework for processor $P(s)$ in d dimensions

Input: X : multidimensional array of size $n_1 \times \cdots \times n_d$, distr$(X)$ = d-dimensional cyclic over $p_1 \cdots p_d$ processors such that $p_l^2 | n_l$.
Output: V : multidimensional array of size $n_1 \times \cdots \times n_d$, distr$(V)$ = d-dimensional cyclic, such that $V = (F_{n_1} \otimes \cdots \otimes F_{n_d})(X)$.

1: $Y^{(s)} = (F_{n_1/p_1} \otimes \cdots \otimes F_{n_d/p_d})(X^{(s)})$; ▷ Superstep (0)
2: **for** $k \in [n_1/p_1] \times \cdots \times [n_d/p_d]$ **do**
3: $\quad Z^{(s)}[k_1, \cdots, k_d] = \prod_{l=1}^{d} \omega_{n_l}^{k_l s_l} Y^{(s)}[k_1, \cdots, k_d]$;
4: **for** $k \in [p_1] \times \cdots \times [p_d]$ **do** ▷ Superstep (1)
5: $\quad$ Put $Z^{(s)}(k : p : n/p)$ in $P(k)$ as subarray $W^{(k)}[sn/p^2 : (s+1)n/p^2 - 1]$;
6: **for** $t \in [n_1/p_1^2] \times \cdots \times [n_d/p_d^2]$ **do** ▷ Superstep (2)
7: $\quad V^{(s)}(t : n/p^2 : n/p) = (F_{p_1} \otimes \cdots \otimes F_{p_d}) \left(W^{(s)}(t : n/p^2 : n/p)\right)$;

$$Y^{(s)}[k_1, \cdots, k_d] = \sum_{j_1=0}^{n_1/p_1-1} \cdots \sum_{j_d=0}^{n_d/p_d-1} X^{(s)}[j_1, \cdots, j_d] \omega_{n_1/p_1}^{j_1 k_1} \cdots \omega_{n_d/p_d}^{j_d k_d}. \tag{3.2}$$

As $\omega_{n_l/p_l}^{j_l k_l} = \omega_{n_l}^{j_l k_l p_l}$, twiddling $Y^{(s)}$ gives

$$Z^{(s)}[k_1, \cdots, k_d] = \sum_{j_1=0}^{n_1/p_1-1} \cdots \sum_{j_d=0}^{n_d/p_d-1} X^{(s)}[j_1, \cdots, j_d] \omega_{n_1}^{k_1(j_1 p_1 + s_1)} \cdots \omega_{n_d}^{k_d(j_d p_d + s_d)}. \tag{3.3}$$

Now $W^{(s)}(t : n/p^2 : n/p)[s']$ corresponds to $W^{(s)}[t + s'n/p^2]$, which corresponds to $W^{(s)}(s'n/p^2 : (s'+1)n/p^2 - 1)[t]$, so it is equal to $Z^{(s')}(s : p : n/p)[t]$, which corresponds to $Z^{(s')}[s + tp]$.

Hence

$$V^{(s)}[t + kn/p^2] = V^{(s)}(t : n/p^2 : n/p)[k] = \sum_{s_1'=0}^{p_1-1} \cdots \sum_{s_d'=0}^{p_d-1} Z^{(s')}[s + tp] \omega_{p_1}^{s_1' k_1} \cdots \omega_{p_d}^{s_d' k_d}. \tag{3.4}$$

Using $\omega_{p_l}^{s_l' k_l} = \omega_{n_l}^{s_l' k_l n_l / p_l}$ gives

$$V^{(s)}[t + kn/p^2] = \sum_{s_1'=0}^{p_1-1} \cdots \sum_{s_d'=0}^{p_d-1} Z^{(s')}[s + tp] \omega_{n_1}^{s_1' k_1 n_1/p_1} \cdots \omega_{n_d}^{s_d' k_d n_d/p_d}. \tag{3.5}$$

We substitute

$$Z^{(s')}[s + tp] = \sum_{j_1=0}^{n_1/p_1-1} \cdots \sum_{j_d=0}^{n_d/p_d-1} X^{(s')}[j_1, \cdots, j_d] \omega_{n_1}^{(s_1 + t_1 p_1)(j_1 p_1 + s_1')} \cdots \omega_{n_d}^{(s_d + t_d p_d)(j_d p_d + s_d')}$$

in equation (3.5) to get

$$V[s_1 + t_1 p_1 + k_1 n_1/p_1, \cdots, s_d + t_d p_d + k_d n_d/p_d] = V^{(s)}[t + kn/p^2] =$$

$$\sum_{s'_1=0}^{p_1-1} \cdots \sum_{s'_d=0}^{p_d-1} \left(\sum_{j_1=0}^{n_1/p_1-1} \cdots \sum_{j_d=0}^{n_d/p_d-1} X^{(s')}[j_1, \cdots, j_d] \omega_{n_1}^{(s_1+t_1p_1)(j_1p_1+s'_1)} \cdots \omega_{n_d}^{(s_d+t_dp_d)(j_dp_d+s'_d)} \right)$$

$$\omega_{n_1}^{s'_1k_1n_1/p_1} \cdots \omega_{n_d}^{s'_dk_dn_d/p_d}$$

Now for any j_l, we have $\omega_{n_l}^{s'_lk_ln_l/p_l} = \omega_{n_l}^{(s'_l+j_lp_l)k_ln_l/p_l}$ because the extra factor $\omega_{n_l}^{j_lp_lk_ln_l/p_l} = \left(\omega_{n_l}^{n_l}\right)^{j_lk_l} = 1$. Leveraging this, we get

$$\sum_{s'_1=0}^{p_1-1} \cdots \sum_{s'_d=0}^{p_d-1} \left(\sum_{j_1=0}^{n_1/p_1-1} \cdots \sum_{j_d=0}^{n_d/p_d-1} X^{(s')}[j_1, \cdots, j_d] \omega_{n_1}^{(s_1+t_1p_1+k_1n_1/p_1)(j_1p_1+s'_1)} \cdots \omega_{n_d}^{(s_d+t_dp_d+k_dn_d/p_d)(j_dp_d+s'_d)} \right).$$

As we can write any number j'_l between 0 and n_l uniquely as $s'_l + j_lp_l$ for $0 \leq s_l < p_l$, $0 \leq j_l < n_l/p_l$ and $X^{(s)}[j] = X[s + jp]$, we can simplify to

$$\sum_{j'_1=0}^{n_1-1} \cdots \sum_{j'_d=0}^{n_d-1} X[j'_1, \cdots, j'_d] \omega_{n_1}^{(s_1+t_1p_1+k_1n_1/p_1)j'_1} \cdots \omega_{n_d}^{(s_d+t_dp_d+k_dn_d/p_d)j'_d}.$$

And this is precisely the image of X under the d-dimensional Fourier transform. □

The computational complexity of a Fourier transform on n elements in total (no matter the dimension) is $O(n \log n)$. So the first part of computation superstep (0) is $O(n/p \log(n/p))$ and computation superstep (2) is $O(n/p^2 \cdot p \log(p))$. As $p^2 \leq n$, we have $p \leq n/p$, so together this is $O(n/p \log(n/p))$. The twiddling is $O(n/p)$ with proper implementation, we can see in Algorithm 6 of the next chapter that there are two complex multiplications in the inner loop no matter the dimension. Each element is communicated once and we have three supersteps, so the BSP-cost of this algorithm is

$$O\left(\frac{n}{p} \log\left(\frac{n}{p}\right)\right) + \frac{n}{p}g + 3l.$$

Chapter 4

Implementation

We discuss the implementation of the higher-dimensional DFT algorithm of this thesis. The implementation can be found at `https://gitlab.com/Thomas637/FFT` and contains both an MPI implementation and a BSPlib implementation. Only the MPI implementation is discussed because the BSPlib version is very similar. The only difference is that in the BSPlib version we unpack the data packets instead of letting MPI do this. The unpacking is straightforward and should be clear from the code. Both programs work in arbitrary dimension, including $d = 1$. For the BSP-version we provide a specific one-dimensional program as well, which is slightly faster than using $d = 1$ for the arbitrary-dimensional variant. This is because $X^{(s)}[sn/p^2 : (s+1)n/p^2 - 1]$ is contiguous for $d = 1$, meaning it does not have to be unpacked.

In this chapter we start our indices at 0 instead of 1 to closer match the implementation in the programming language C, which indexes arrays starting at 0.

Remark 9. We are lucky in that we can use the optimised library FFTW (Fastest Fourier Transform in the West) [4] for step 1 and 4 of the four-step framework. This library uses a so-called planner function, which does experiments to determine a good algorithm for your hardware.

4.1 Communication and twiddle

For the parallelism, we use the Message-Passing Interface (MPI). This is a standard for distributed parallel computing available on many different machines. We use the MPI_Alltoallv function for communication. For that we need to move the data we will send into a buffer, and then send it to the correct place. The data $P(s)$ sends to $P(t)$ is not stored contiguously. Using a more minimalistic library like BSPlib, we would need to send the data to a receive buffer and then unpack to its final location. With MPI we can circumvent this extra data movement by directly specifying to the library where the received data needs to be stored in a strided fashion. The mechanism for this is called a type map, which contains the data types and displacements.

4.1.1 Pack and twiddle

We want to divide $X^{(s)}$ up into packets $\{\text{packet}_t \mid t \in S\}$, so that we send packet_t to $P(t)$. As it is expensive to get data from main memory into the CPU registers, we combine this with the twiddling. We save the packets in a contiguous buffer, and place packet_t before $\text{packet}_{t'}$ precisely when the row-major index of t is smaller than the row-major index of t'.

We need to traverse $X^{(s)}$ in row-major order in order to use the full cache line. That way we amortise the number of cache misses and use the full bandwidth between main memory and cache. This culminates in

Algorithm 6. We write $T_{n,p,s}$ for the twiddling done on an array of size n, on p processors, by $P(s)$. So multiplying $X[k]$ by $\prod_{l=1}^{d} \omega_{n_l}^{k_l s_l}$.

Algorithm 6 Packing and twiddling

Input: a d-dimensional array $X^{(s)}$ of size $n_0/p_0 \times \cdots \times n_{d-1}/p_{d-1}$ in row-major format.
Output: d-dimensional arrays packet_k containing $T_{n,p,s}(X^{(s)}(k : p : n/p))$ without the extra space.

```
1: for t_0 := 0 to n_0/p_0 − 1 do
2:     factor_0 := ω_{n_0}^{t_0 s_0};
3:     for t_1 := 0 to n_1/p_1 − 1 do
4:         factor_1 := factor_0 · ω_{n_1}^{t_1 s_1}
5:         ⋮
6:         for t_{d−1} := 0 to n_{d−1}/p_{d−1} − 1 do
7:             factor_{d−1} := factor_{d−2} · ω_{n_{d−1}}^{t_{d−1} s_{d−1}};
8:             packet_{t mod p}[t div p] = X[t] · factor_{d−1};
```

As d is not known at compile time, we need to implement this recursively, depth l of the recursion being responsible for the lth nested loop. Trigonometric functions are expensive, so we use tables for the twiddle factors. The only problems left are that going from row-major indices to multidimensional indices is expensive, and that mod and div operations in the inner loops are expensive. The first problem is easily solved by passing the row-major index of t along in the recursion and noting that increasing t_l by one, increases the row-major index by $(n_{d-1}/p_{d-1}) \cdots (n_{l+1}/p_{l+1})$.

For the second problem, we unroll the latter loop in pieces of length p_{d-1}. We will need some more information from the loops above it, but this gives Algorithm 7. This algorithm is correct because k cycles between 0 and $p_{d-1} - 1$, packetIndex is increased once every p_{d-1} iterations, and increasing either $t_{d-1} \bmod p_{d-1}$ or $t_{d-1} \operatorname{div} p_{d-1}$ by one, increases the row-major index of $(t_0 \bmod p_0, \cdots, t_{d-2} \bmod p_{d-2}, t_{d-1} \bmod p_{d-1})$, respectively $(t_0 \operatorname{div} p_0, \cdots, t_{d-2} \operatorname{div} p_{d-2}, t_{d-1} \operatorname{div} p_{d-1})$ by one.

Algorithm 7 Unrolled inner loop

Input: localDataIndexStart: the row-major index of $(t_0, \cdots, t_{d-2}, 0)$ with respect to $n_0/p_0 \times \cdots \times n_{d-1}/p_{d-1}$,
factor $= \omega_{n_0}^{s_0 t_0} \cdots \omega_{n_{d-2}}^{s_{d-2} t_{d-2}}$,
packetIndexStart: the row-major index of $(t_0 \operatorname{div} p_0, \cdots, t_{d-2} \operatorname{div} p_{d-2}, 0)$ with respect to $n_0/p_0^2 \times \cdots \times n_{d-1}/p_{d-1}^2$
m : the row-major index of $(t_0 \bmod p_0, \cdots, t_{d-2} \bmod p_{d-2}, 0)$ with respect to $p_0 \times \cdots \times p_{d-1}$.
Output: the same as the last loop of Algorithm 6.

```
1: localDataIndex := localDataIndexStart, packetIndex := packetIndexStart, t_{d−1} := 0;
2: while t_{d−1} < n_{d−1}/p_{d−1} do
3:     for k := 0 to p_{d−1} − 1 do
4:         packet_{m+k}[packetIndex] = X[localDataIndex] · factor · ω_{n_{d−1}}^{s_{d−1} t_{d−1}};
5:         t_{d−1}++;
6:         localDataIndex++;
7:     packetIndex++;
```

We pass the inputs as arguments of the recursion to make sure that the last loop has them. Updating these values is not trivial, so let us explore that now. Increasing t_l by one, increases localDataIndexStart by $(n_{d-1}/p_{d-1}) \cdots (n_{l+1}/p_{l+1})$. packetIndexStart increases by $(n_{d-1}/p_{d-1}^2) \cdots (n_{l+1}/p_{l+1}^2)$ every p_l iterations and m cycles between 0, $p_{d-1} \cdots p_{l+1}$, $2p_{d-1} \cdots p_{l+1}$, $\cdots$, $(p_{l-1} - 1)p_{d-1} \cdots p_{l+1}$ in those p_l iterations. So we need to unroll the other loops as well. This gives recursive Algorithm 8 where we call Algorithm 7 when depth hits $d - 1$.

Algorithm 8 Pack and twiddle

Input: a d-dimensional array $X^{(s)}$ of size $n_0/p_0 \times \cdots \times n_{d-1}/p_{d-1}$ in row-major format.
Output: d-dimensional arrays packet$_k$ containing $T_{n,p,s}(X^{(s)}(k:p:n/p))$ without the extra space.

```
1: function PACKANDTWIDDLE(localDataIndex, factor, packetIndex, m, depth)
2:     localDataStride := (n_{d-1}/p_{d-1}) ··· (n_{depth+1}/p_{depth+1});
3:     packetStride := (n_{d-1}/p_{d-1}^2) ··· (n_{depth+1}/p_{depth+1}^2);
4:     mStride := p_{d-1} ··· p_{depth+1};
5:     t_l := 0;
6:     while t_l < n_l/p_l do
7:         for k := 0 to p_l − 1 do
8:             PACKANDTWIDDLE(localDataIndex, factor ·ω_{n_depth}^{s_depth·t_l}), packetIndex, m + k · mStride, depth
       +1);
9:             localDataIndex += localDataStride;
10:            t_l++;
11:        packetIndex += packetStride;
```

4.1.2 Type map

There is a standard function for creating the type map of a subarray, so the only thing we need to do is specify the starting points. The starting point of the data from $P(s)$ is the row-major index of $(s_0 n_0/p_0^2, \cdots, s_{d-1} n_{d-1}/p_{d-1}^2)$ with respect to an array of size $n_0/p_0 \times \cdots \times n_{d-1}/p_{d-1}$, so $s_{d-1} n_{d-1}/p_{d-1}^2 + \cdots + s_0 n_0/p_0^2 \cdot n_{d-1}/p_{d-1} \cdots n_1/p_1$. This needs to be stored in an array of length the number of processors. It is useful to interpret this as a d-dimensional array as well. This gives Algorithm 9, where the updating of sum only carries downwards (as we must implement this recursively). So on the first iteration, if $s_0 = k$, then sum $= (k n_0/p_0^2)(n_{d-1}/p_{d-1}) \cdots (n_1/p_1)$.

Algorithm 9 Displacements for the type map

Input: empty multidimensional array startingPoints of size $p_0 \times \cdots \times p_{d-1}$.
Output: startingPoints filled with startingPoints[s] $:= s_{d-1} n_{d-1}/p_{d-1}^2 + \cdots + s_0 n_0/p_0^2 \cdot n_{d-1}/p_{d-1} \cdots n_1/p_1$.

```
1: sum := 0;
2: for s_0 := 0 to p_0 − 1 do
3:     sum += (s_0 n_0/p_0^2)(n_{d-1}/p_{d-1}) ··· (n_1/p_1);
4:     ⋮
5:     for s_{d-1} := 0 to p_{d-1} − 1 do
6:         startingPoints[s] := sum;
7:         sum += s_{d-1} n_{d-1} p_{d-1}^2;
```

We again pass the row-major index and implement this recursively, similarly to the pack and twiddle. We have $n_0/p_0 + \cdots + n_{d-1}/p_{d-1}$ weights, but we multiply by these weights a total of $(n_0/p_0) \cdot \cdots \cdot (n_{d-1}/p_{d-1})$ times. So each weight is used a lot, and they are expensive to compute. For this reason, we calculate the weights once and store them in a table.

Chapter 5

Results

Here we present experimental results and analyse them. We compare our implementations with FFTW [4] for numbers of processors FFTW can handle, and with PFFT [9] for numbers of processors FFTW cannot.

5.1 Experiment description

We have timings for four different programs. All are compiled with Intel 2021.2.0 and flags -O3 -march=native unless otherwise specified. The first program MPI is the BSP-algorithm of this thesis implemented with MPI, the second program BSP is the same algorithm implemented with BSPlib. We used BSPonMPI [12] version 1.1.1. The third program is the MPI version supplied by FFTW. The FFTW version is 3.3.9. Finally we use PFFT commit e4cfcf9 on `https://github.com/mpip/pfft` (no version number available). We have tried Intel MPI version 2021.2.0 and OpenMPI version 4.1.1. OpenMPI was very inconsistent (but in some cases faster than Intel MPI), so the timings are from Intel MPI unless otherwise specified.

We have run experiments on arrays of 2^{30} elements total, in shapes $1024\times1024\times1024$ and $64\times64\times64\times64\times64$.

The timings were obtained on the supercomputer Snellius, on the thin node partition. A node consists of two 64 core AMD Rome 7H12 processors in a dual socket configuration, running at 2.6GHz. So 128 cores per node. There is 2GiB of RAM available per core. The interconnect is HDR100 (100Gbps), fat tree. The job scheduler used is SLURM, and we do not share nodes. The operating system is CentOS7. We have used a single switch in our experiments and bound the MPI processes to cores with $--$cpu_bind=cores.

Ideally, we would write down the time at the start of the DFT, at the end, and subtract the two values. The problem with this is that the synchronising function MPI_Barrier of the MPI library only guarantees that no processor leaves the barrier before all processors have entered. So there is no guarantee that every processor starts the DFT function at the same time. That is why we apply the DFT 100 times, so that the small difference in leaving the barrier becomes negligible.

5.2 Theoretical comparison with FFTW

We first give a theoretical comparison of our implementation with FFTW and PFFT, see Section 1.4 for how these implementations work. FFTW and PFFT use about $5n \log n$ flops in the total number of elements n, ours uses about $6n$ more because of the twiddle step. This twiddle step is combined with packing the data into a buffer, so the twiddling can be done (in part) while we wait for the next element to arrive in L1 cache from RAM. Commenting out the twiddling in the implementation of Algorithm 6 improves the speed

by approximately 10%. As the amount of computational work is roughly the same, differences will have to be explained by communication.

FFTW can use up to

$$\min(n_i, \prod_{j \neq i} n_j)$$

processors to transform a $n_1 \times \cdots \times n_d$ array with one communication step when we distribute among the ith dimension. If $n = n_1 \cdots n_d$ is the total number of elements, that means it can use at most $\sqrt{n}$ processors, but often it can use only less. This is also an upper bound on the number of processors our algorithm can use. Namely $p_1 \cdots p_d$ for $p_i^2 \mid n_i$. If n_i does not have nice divisors, we can use less processors. In practice n_i are chosen highly composite, often powers of two, so this is not a problem. If the array is a cube in odd dimension, for example $n \times n \times n$, the FFTW approach can only use n processors, whereas we can use $n\sqrt{n}$ processors if n is a square. PFFT does not have this limitation, but as mentioned in the introduction, it may need more communication supersteps. So it can use up to $\prod_{j \in J} n_j$ processors for $J \subset \{1, \cdots, d\}$, and it needs $\lceil \frac{d}{d-|J|} \rceil - 1$ communication steps or $\lceil \frac{d}{d-|J|} \rceil$ depending on whether we want to end up in this same distribution or not.

Furthermore, our algorithm can use more processors when transforming very rectangular multidimensional arrays. For example, if we have a $2^{20} \times 2^6$ array, our algorithm can use $2^{10} \cdot 2^3 = 8192$ processors, and the FFTW/PFFT approach can only use $2^6 = 64$.

Finally, let us reiterate that FFTW and PFFT end in a different distribution, and need to communicate (almost) all elements one more time to end up in the same distribution they started in. This communication step is called a transpose, and is symmetric to the communication superstep(s) used while calculating the output. So we expect our algorithm to be as fast as FFTW and PFFT in all cases, faster when FFTW or PFFT have to end up in the same distribution, and faster than PFFT when it has to use sufficiently many processors that an extra communication step is required. We expect the MPI implementation of the BSP algorithm to outperform the BSPlib implementation, because we can use derived data types to circumvent extra data movement using MPI. BSPonMPI could theoretically do this as well, but does not because it is difficult to implement. However, this should not affect the scaling.

5.3 Observations

We have done experiments on Snellius. In Table 5.1 we see the results for transforming an array of size $1024 \times 1024 \times 1024$ for different numbers of processors p. In Table 5.2 we see the results for transforming an array of size $64 \times 64 \times 64 \times 64 \times 64$. In the FFTW/PFFT columns, the left number is the time it takes when we ask to end in the same distribution we started in (flags - / PFFT_TRANSPOSED_NONE), and the right number is the time when we end in a different distribution (flags FFTW_TRANSPOSED_OUT / PFFT_TRANSPOSED_OUT). PFFT uses a two-dimensional processor distribution, putting as many processors along the first dimension as possible. We use the FFTW_MEASURE or PFFT_MEASURE flags. This carries out timings on your hardware, and chooses an algorithm accordingly. This makes the program faster, but it might perform differently on different hardware. On Snellius the results presented here are reproducible. For $p = 1, 2$ the MPI version with Intel MPI timed out on ten hours and only one iteration. For this reason, we used OpenMPI in Table 5.2 for $p = 1, 2$. The BSPonMPI version timed out for $p = 8, 16$ on also a very generous time limit. I expect this is also due to a problem with Intel MPI. On 4096 processors and the full number of iterations, the BSPlib version timed out as well. It takes at least 6 seconds, which is in line with the increasing costs as p becomes larger than 1024.

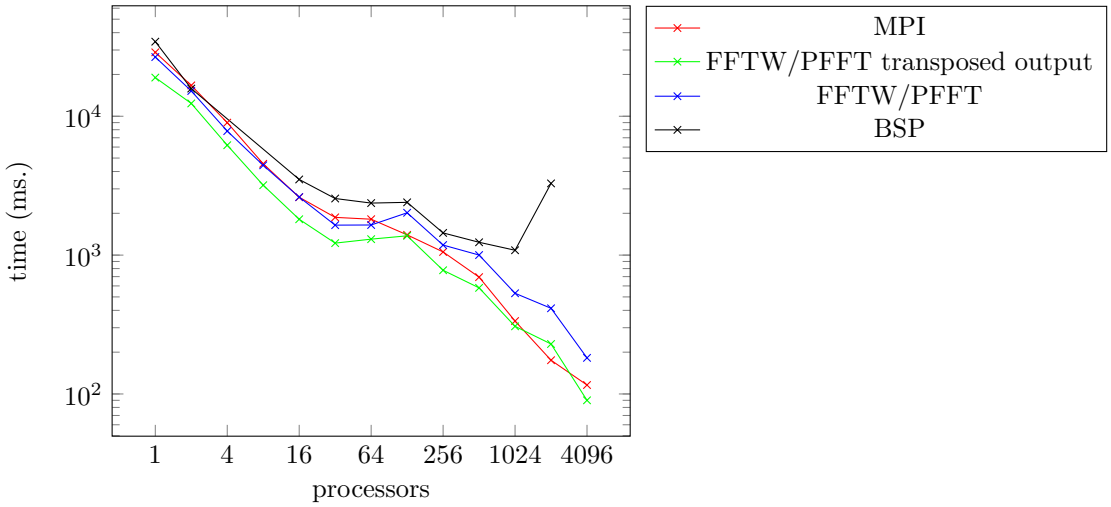


Figure 5.1: Graph corresponding to Table 5.1.

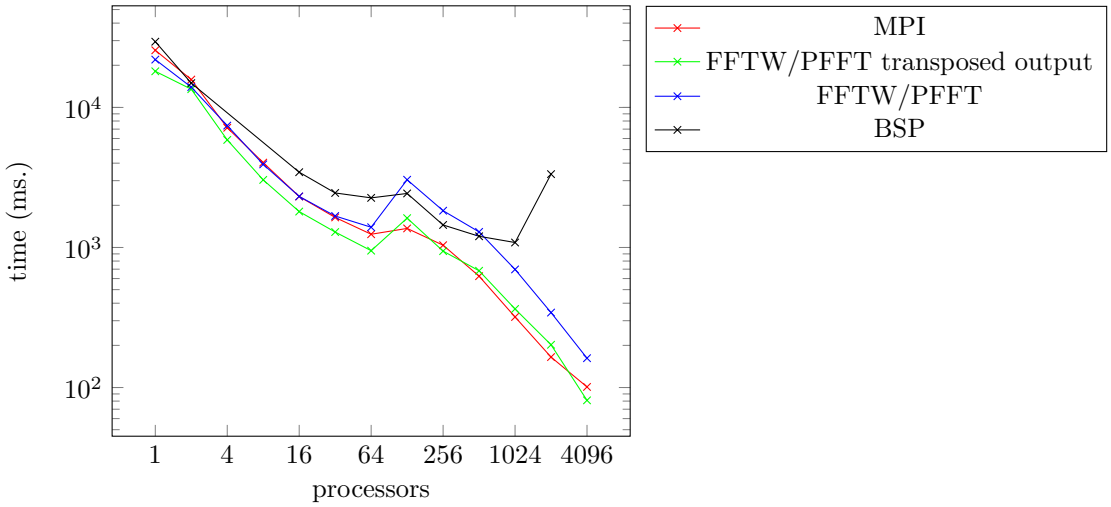


Figure 5.2: Graph corresponding to Table 5.2.

p	MPI	FFTW	PFFT	BSP
1	28,861	26,685 / 18,989		34,455
2	16,678	15,215 / 12,369		15,827
4	8,996	7,821 / 6,186		
8	4,545	4,433 / 3,187		
16	2,611	2,618 / 1,812		3,506
32	1,870	1,644 / 1,220		2,556
64	1,814	1,648 / 1,303		2,370
128	1,399	2,018 / 1,377		2,399
256	1,054	1,181 / 778		1,445
512	694	1,001 / 581		1,237
1024	335	531 / 307		1,084
2048	175	-	414 / 229	3,284
4096	116	-	182 / 90	$> 6{,}000$

Table 5.1: Time to transform a $1024 \times 1024 \times 1024$ array in ms.

p	MPI	FFTW	PFFT	BSP
1	25,590	21,955 / 18,115		29,477
2	15,813	13,964 / 13,496		14,911
4	7,162	7,418 / 5,862		
8	4,036	3,921 / 3,038		
16	2,311	2,316 / 1,805		3,448
32	1,639	1,678 / 1,289		2,451
64	1,243	1,395 / 949		2,261
128	1,368	-	3,045 / 1,620	2,431
256	1,038	-	1,831 / 942	1,448
512	624	-	1,292 / 682	1,202
1024	319	-	697 / 364	1,085
2048	165	-	343 / 202	3,340
4096	101	-	162 / 81	$> 6{,}000$

Table 5.2: Time to transform a $64 \times 64 \times 64 \times 64 \times 64$ array in ms.

Sequentially transforming an array of size $64 \times 64 \times 64 \times 64 \times 64$ takes 17,381 ms. Sequentially transforming an array of size $1024 \times 1024 \times 1024$ takes 17,541 ms.

5.4 Discussion

We will now try to analyse the results. As FFTW decides at runtime which algorithm to use, and the exact implementation of both FFTW and MPIs collective routines/derived data types are very complicated, it is beyond my ability to completely explain everything with certainty, but I will give some reasons I think are likely to explain the observations.

5.4.1 Analysis of BSP algorithm with MPI, FFTW, and PFFT

In Table 5.2, we see a speedup of about 170 (compared to the sequential FFTW), attained on 4096 processors. Counting $5n\log(n)$ flops to transform an array with n elements, we reach $1.6 \cdot 10^{12}$ flops per second or 1.6 Tflops/s.

In both Table 5.1 and Table 5.2 we see that our MPI implementation is faster than FFTW/PFFT when we require that the input and output are in the same distribution and we go out of core (so $p > 128$). This is as expected. When we do not go out of core, FFTW is significantly faster. This is likely due to FFTW taking advantage of shared memory in these cases. Let us now investigate the case where FFTW/PFFT does not require that the input and output are in the same distribution.

In Table 5.2 we see that our MPI program performs about as well as PFFT when we do not require the output to be in the same distribution as the input. This makes sense as a two-dimensional processor distribution suffices, meaning we need $\lceil \frac{5}{5-2} \rceil - 1 = 1$ communication superstep. For $p = 128, 512, 1024$ BSP is a good bit faster, for $p = 256, 4096$ PFFT is a good bit faster. These differences can be explained by the implementation of the communication step. We simply describe to MPI what we want by use of MPI_Alltoallv and derived data types, and let the MPI library handle the exact algorithm for communication. FFTW and PFFT experiment with several algorithms during the planning phase, and then choose the fastest option. Apparently, sometimes MPI chooses the superior algorithm, and sometimes FFTW/PFFT does.

In Table 5.1 we see that FFTW output in a different distribution is faster than our MPI program. This can be explained by FFTW doing a better job than Intel MPI at communicating. Using OpenMPI version 4.1.1. it took only 515 ms. on 512 processors, instead of the 694 ms. by Intel MPI, beating even FFTW. So there is room for improvement in the MPI implementation and it is possible to attain the same performance as FFTW. It is strange that our program performs about as well as PFFT for $p = 2048, 4096$. As $p > 1024$, we need a two-dimensional distribution, leading to $\lceil \frac{3}{3-2} \rceil - 1 = 2$ communication supersteps. When looking at $p = 4096$, we see that the final transpose putting the output for PFFT in the same distribution as the input, takes as long as the entire calculation. (Namely $182 - 90 = 92$ ms.) This is very strange because calculating the output takes two similar transposes, so we would expect the cost of the final transpose to be less than half the cost of calculating the output. I cannot explain this.

In both cases, we see only marginal improvements as we get around 64 processors, that is, one of the two CPUs in a node. This makes sense as the DFT is bottlenecked by bandwidth, and all cores of a CPU share the same data bus to RAM, so the maximum available bandwidth per processor goes down when using more processors within the same CPU. In the three-dimensional case this happens from 32 to 64 processors, in the five-dimensional case when going from 64 to 128 processors. Perhaps SLURM divides 64 total processes over two CPUs in one case, and gives it to only one CPU in the other case.

We observe that every program is slightly faster in the five-dimensional case compared to the three-dimensional case, even though the cost is theoretically the same. If calculating a DFT takes x seconds, then FFTW can typically transform k identical DFTs in less than kx seconds. This is in part because it can use SIMD parallellism more effectively. The five-dimensional case consists of 64^4 one-dimensional arrays in one dimension, and the three-dimensional case of $1024^2 < 64^4$. Therefore, we have more identical DFTs in the five-dimensional case, causing the computation steps to run slightly faster.

5.4.2 Analysis of BSPlib

We see a speedup of 17 when using 1024 processors. This means we reach a top speed of 170 Gflops/s. Like the other programs, we scale decently except for when we get close to using a full node. Unlike the other programs, this scaling stops when we reach 1024 processors. This could be because we send more messages, making it harder for BSPonMPI to choose a good algorithm. It should use a collective here instead of individual remote memory writes. With so many messages, it becomes complex to use remote memory writes, and the MPI implementation will have optimised the communication pattern of a collective function. BSPonMPI is able to use MPI collectives, trying to switch at an optimal point. Perhaps it did not succeed in this case.

Overall, the BSPlib version is slower than its MPI equivalent as well, which can be explained by the extra data movement caused by the lack of derived data types and internal buffers.

Chapter 6

Tensor product of parallel algorithms

In this chapter we show how we may transform BSP algorithms for linear functions $f_1, \cdots, f_d$ and turn them into a BSP algorithm for $f_1 \otimes \cdots \otimes f_d$. This is applied to the d-dimensional DFT and the four-step framework to give an alternative proof of Algorithm 4. However, this can also be applied to other BSP algorithms. For example, if the input is real, there are faster algorithms. There are also discrete Fourier-like transforms such as the Hartley transform, sine-cosine transform, which have higher-dimensional variants formed by taking the tensor product.

6.1 Interpretations

Technically, the Fourier transform of length n is a linear map $\mathbb{C}^n \to \mathbb{C}^n$, but most people regard it as a function between $\{[x_1, \cdots, x_n] \mid x_j \in \mathbb{C}\}$ and itself. Of course this set is isomorphic to $U\mathbb{C}^n$ and they really mean the function

$$\{[x_1, \cdots, x_n] \mid x_j \in \mathbb{C}\} \cong U\mathbb{C}^n \xrightarrow{UF_n} U\mathbb{C}^n \cong \{[x_1, \cdots, x_n] \mid x_j \in \mathbb{C}\}.$$

We will continue in this vein, and identify $\{[x_1, \cdots, x_n] \mid x_j \in \mathbb{C}\}$ with $UF([n])$ in order to regard F_n as either a function between $UF([n])$ and itself, or a linear function between $F([n])$ and itself. This is not strictly true, but inserting these isomorphisms everywhere clutters the page, and does not offer any insight (in my opinion). Similarly we will identify sets of multidimensional arrays with $UF([n_1] \times \cdots \times [n_d])$. When we do not need to distinguish between the cases, we write UFB. We construct the finite (co)product in Vect as the free functor applied to the coproduct in Set. We are allowed to do this as $F \dashv U$ means F preserves coproducts (and in Vect the finite product functor and finite coproduct functor act the same on objects and morphisms). So we have equality $\bigoplus_{s \in S} FB_s = F\left(\coprod_{s \in S} B_s\right)$. At first glance it might look odd to have a function of the form $F \cdots$ coming out of $\bigoplus_{s \in S} FB_s$, but it is allowed for that reason. Also, we may write $f_1 \otimes \cdots \otimes f_d : F(B_1 \times \cdots \times B_d) \to F(B_1 \times \cdots \times B_d)$, in which case we mean to compose with the isomorphism of Proposition 3 on both sides.

6.2 Decomposition

In this section, we will see that BSP algorithms (with linear computation steps) of a linear function correspond to decompositions of that function.

6.2.1 Computation superstep

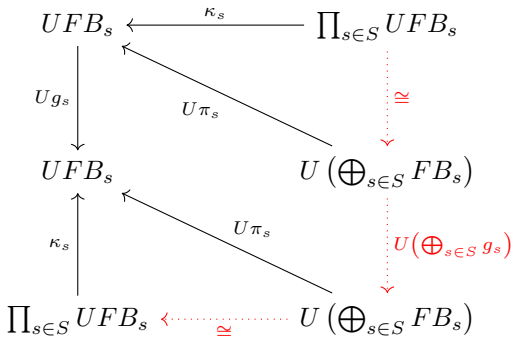
For local data structures of the form UFB_s, with B_s a set like $[n_s]$ or $[n_{s_1}] \times \cdots \times [n_{s_d}]$, a computation superstep consists of a collection of functions $\{UF(B_s) \to UF(B_s) \mid s \in S\}$. In other words, a function $\prod_{s\in S} UF(B_s) \to \prod_{s\in S} UF(B_s)$. We restrict ourselves to functions of the form Ug_s for linear maps $g_s : F(B_s) \to F(B_s)$. When working with linear algebra, this is not a strong assumption. The four-step framework, the parallel algorithm of [2], and parallel algorithms of matrix-vector multiplication have computation steps of this form. As $F \dashv U$, the free functor preserves finite products. This isomorphism

$$U\left(\bigoplus_{s\in S} F(B_s)\right) \cong \prod_{s\in S} UF(B_s)$$

respects products of maps as well in the sense that the following square commutes.

$$\begin{array}{ccc} U\left(\bigoplus_{s\in S} F(B_s)\right) & \xleftarrow{\cong} & \prod_{s\in S} UF(B_s) \\ {\scriptstyle U(\bigoplus_{s\in S} g_s)}\downarrow & & \downarrow{\scriptstyle \prod_{s\in S} Ug_s} \\ U\left(\bigoplus_{s\in S} F(B_s)\right) & \xrightarrow[\cong]{} & \prod_{s\in S} UF(B_s) \end{array}$$

To see this, consider the following diagram with the arrows drawn in that induce the red route. We see that the dotted red route fits into the diagram defining $\prod_{s\in S} Ug_s$, so they must be equal. We write κ for the projections in Set and π for the ones in Vect.



6.2.2 Distribution

A distribution is defined to be a bijection between the global data structure and the local data structures. As we just saw, the global data structure can be identified by UFB and the local data structures by $\prod_{s\in S} UFB_s$. These bijections correspond to collections of functions $(\phi_s : B_s \to B)_{s\in S}$ such that $[\phi_s \mid s \in S]$ is invertible. We interpret $\phi_s(k) = j$ as $X^{(s)}[k] = X[j]$. So the bijection between data structures is given by

$$\prod_{s\in S} UFB_s \cong U\left(\bigoplus_{s\in S} FB_s\right) \xrightarrow{UF([\phi_s \mid s\in S])} UFB.$$

6.2.3 Redistribution

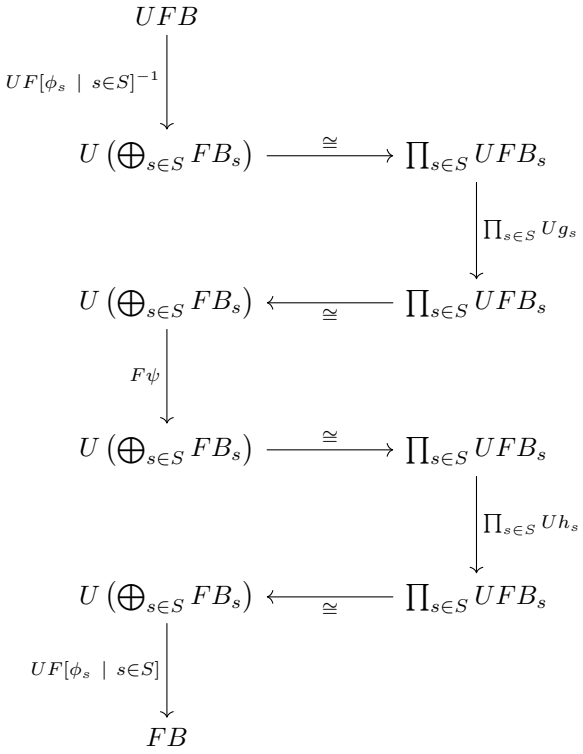
A redistribution is defined to be an automorphism on the local data structures. They correspond with automorphisms ψ on $\bigsqcup_{s\in S} B_s$, where we associate "put $X^{(s)}[j]$ in $P(t)$ as $X^{(t)}[k]$" with "$\psi(s, j) = (t, k)$".

To get the automorphism on the local data structures, we compose $F\psi$ by the isomorphism $U\left(\bigoplus_{s\in S} FB_s\right) \cong \prod_{s\in S} UFB_s$ on both sides.

6.2.4 Factorisation

Suppose we can calculate $Uf : UB \to UB$ in parallel over a set $\{P(s) \mid s \in S\}$ of processors, in two computation supersteps. We have a distribution ϕ, a redistribution ψ, local computations $(Ug_s : UFB_s \to UFB_s)_{s\in S}, (Uh_s : UFB_s \to UFB_s)_{s\in S}$.

By the discussion above, this is equivalent to Uf decomposing as



Using the commutative square of the computation superstep we get that Uf is equal to the forgetful functor applied to the following composition. As U is faithful (it only forgets structure), this composition is equal to f.

$$FB \xrightarrow{F[\phi_s \mid s\in S]^{-1}} \bigoplus_{s\in S} FB_s \xrightarrow{\oplus_{s\in S} g_s} \bigoplus_{s\in S} FB_s \xrightarrow{F\psi} \bigoplus_{s\in S} FB_s \xrightarrow{\oplus_{s\in S} h_s} \bigoplus_{s\in S} FB_s \xrightarrow{F[\phi_s \mid s\in S]} FB$$

As the translation from algorithms to (linear) functions goes both ways, a decomposition in this form leads to a parallel algorithm. This is a fact we will exploit in the next section.

6.3 Tensor product of the decompositions

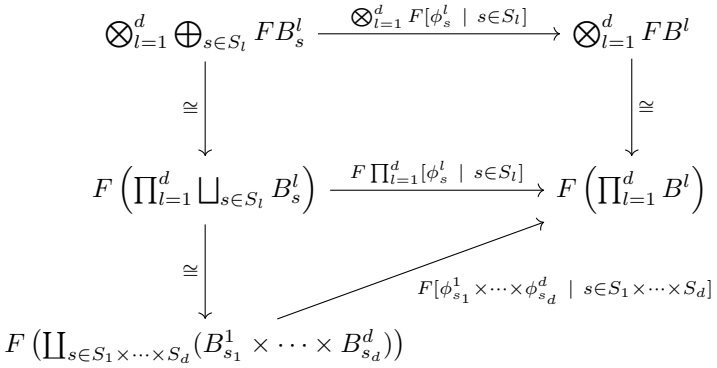
Suppose we have a parallel algorithms to calculate some functions $f_1, \cdots, f_d$, as in the previous chapter, so decompositions

$$FB^l \xrightarrow{F[\phi^l_s \mid s\in S_l]^{-1}} \bigoplus_{s\in S_l} FB^l_s \xrightarrow{\bigoplus_{s\in S_l} g^l_s} \bigoplus_{s\in S_l} FB^l_s \xrightarrow{F\psi^l} \bigoplus_{s\in S_l} FB^l_s \xrightarrow{\bigoplus_{s\in S_l} h^l_s} \bigoplus_{s\in S_l} FB^l_s \xrightarrow{F[\phi^l_s \mid s\in S_l]} FB^l.$$

As the tensor product is functorial, we can tensor each factor in the decomposition separately. It will turn out that we end up with a decomposition of $f_1 \otimes \cdots \otimes f_d$ of that form, so a parallel algorithm for $f_1 \otimes \cdots \otimes f_d$.

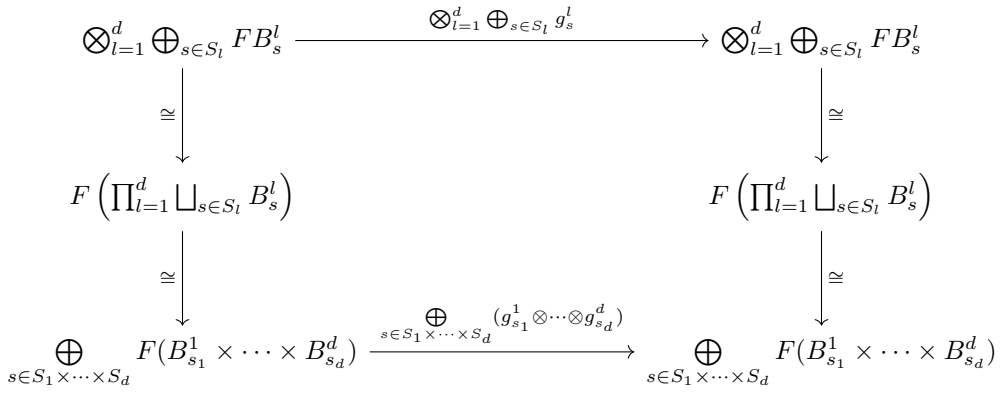
6.3.1 Distribution

Using Proposition 5, we get the following commutative diagram.



6.3.2 Computation step

By Proposition 6, we have a commutative diagram

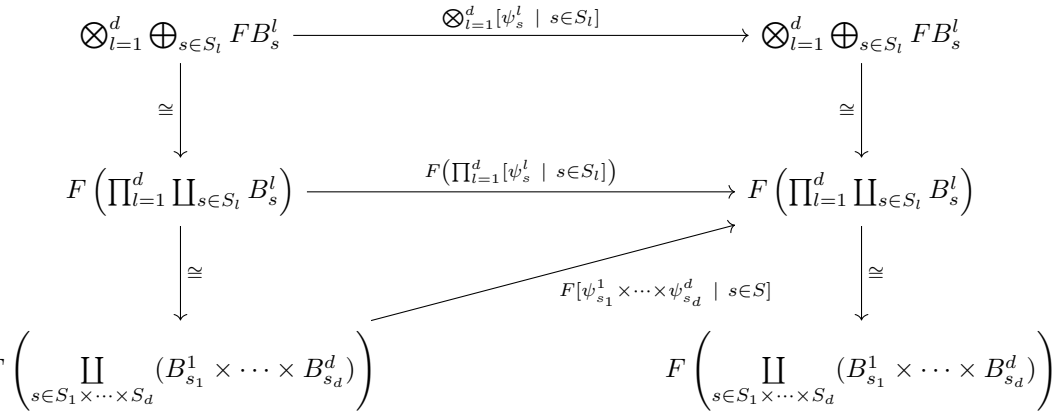


Recall that the vertical sides are the same as in the distribution, because of how we constructed $\bigoplus$. So we can paste them together.

6.3.3 Redistribution

The redistribution $\psi : \coprod_{s\in S} B_s \to \coprod_{s\in S} B_s$ is equal to $[\psi \circ \iota_s \mid s \in S]$. We name such a component ψ_s and note that $\psi_s(j) = (t, k)$ is interpreted as "we put $X^{(s)}[j]$ in $P(t)$ as $X^{(t)}[k]$".

We have a commutative diagram



In the diagonal arrow, I mean $s \in S_1 \times \cdots \times S_d$, but that did not fit. Composing the diagonal arrow with the downward isomorphism yields F applied to the composition

$$\coprod_{s\in S_1\times\cdots\times S_d} (B_{s_1}^1 \times\cdots\times B_{s_d}^d) \xrightarrow{[\psi_{s_1}^1 \times\cdots\times \psi_{s_d}^d \mid s\in S_1\times\cdots\times S_d]} \prod_{l=1}^{d} \coprod_{s\in S_l} B_s^l \xrightarrow{\cong} \coprod_{s\in S_1\times\cdots\times S_d} (B_{s_1}^1 \times\cdots\times B_{s_d}^d)$$

We denote the first component (the processor index) of $\psi_s^l(k)$ by just $\psi_s^l(k)$ and the second component (the local position) by $(\psi')_s^l(k)$. This composition is given by

$$((s_1,\cdots,s_d),(k_1,\cdots,k_d)) \mapsto \left((\psi_{s_1}^1(k_1),\cdots,\psi_{s_d}^d(k_d)),((\psi')_{s_1}^1(k_1),\cdots,(\psi')_{s_d}^d(k_d))\right).$$

So interpreting this as a redistribution over a set of processors indexed by $S_1 \times \cdots \times S_d$, we just combine the components of $\psi^1,\cdots,\psi^d$ dimension-wise. Element $X^{(s)}[k]$ is put in $P((\psi_{s_1}^1(k_1),\cdots,\psi_{s_d}^d(k_d)))$ as $X^{(\psi_{s_1}^1(k_1),\cdots,\psi_{s_d}^d(k_d))}[(\psi')_{s_1}^1(k_1),\cdots,(\psi')_{s_d}^d(k_d)]$. That is why we call this composition $\psi^1 \times \cdots \times \psi^d$ by some abuse of notation.

Again note that we can paste this on the square from the computation step as

$$F\left(\coprod_{s\in S_1\times\cdots\times S_d} (B_{s_1}^1 \times\cdots B_{s_d}^d)\right) = \bigoplus_{s\in S_1\times\cdots\times S_d} F(B_{s_1}^1 \times\cdots\times B_{s_d}).$$

6.3.4 Putting it all together

We can glue all of these squares together, to get the following decomposition of $f_1 \otimes \cdots \otimes f_d$:

$$\bigotimes_{l=1}^{d} FB^l \cong F\left(\prod_{l=1}^{d} B^l\right) \xrightarrow{F[\phi_{s_1}^1 \times\cdots\times \phi_{s_d}^d \mid s\in S_1\times\cdots\times S_d]^{-1}} \bigoplus_{s\in S_1\times\cdots\times S_d} F\left(B_{s_1}^1 \times\cdots\times B_{s_d}^d\right)$$

$$\xrightarrow{\bigoplus_{s\in S_1\times\cdots\times S_d} g_{s_1}^1\otimes\cdots\otimes g_{s_d}^d} \bigoplus_{s\in S_1\times\cdots\times S_d} F\left(B_{s_1}^1 \times\cdots\times B_{s_d}^d\right) \xrightarrow{F\psi^1\times\cdots\times\psi^d}$$

$$\bigoplus_{s\in S_1\times\cdots\times S_d} F\left(B^1_{s_1}\times\cdots\times B^d_{s_d}\right) \xrightarrow{\bigoplus_{s\in S_1\times\cdots\times S_d} h^1_{s_1}\otimes\cdots\otimes h^d_{s_d}} \bigoplus_{s\in S_1\times\cdots\times S_d} F\left(B^1_{s_1}\times\cdots\times B^d_{s_d}\right)$$

$$\xrightarrow{F[\phi^1_{s_1}\times\cdots\times\phi^d_{s_d}\ \mid\ s\in S_1\times\cdots\times S_d]} F\left(\prod_{l=1}^d B^l\right)\cong\bigotimes_{l=1}^d FB^l$$

(The large diagram does not fit on a page, so you will have to do the diagram chase in your mind.)

We identify tensors as multidimensional arrays, so let us ignore the $\bigotimes_{l=1}^d FB^l$ on the outsides. We see that we have a decomposition corresponding to a parallel algorithm again. The set of processors is indexed by $S_1\times\cdots\times S_d$ and as we saw in the section on the redistribution, the redistribution is done dimension-wise. The distribution is done dimension-wise as well because it is given by $[\phi^1_{s_1}\times\cdots\times\phi^d_{s_d}\mid s\in S_1\times\cdots\times S_d]$. For the computation steps we also tensor the local computations. This is all very abstract, so in the next section we will show how this leads to an alternative proof of 3. (Contrary to the order of chapters in this thesis, I actually used this decomposition to derive the higher-dimensional parallel algorithm, and only found the direct proof later.)

6.4 Application to the DFT

We now apply this theory to Algorithm 3. We do this in-place, so $y=x$. If we use p_l processors in the lth dimension, the tensor product of the distributions has $X^{(s)}[k]$ corresponding with $X[k+sp]$ (again using abbreviations $X[k+sp]$ for $X[k_1+s_1p_1,\cdots,k_d+s_dp_d]$ etc.).

Let us call the linear map calculated in steps two and three $T_{n,p,s}$. By functoriality, the first computation step becomes

$$\bigotimes_{l=1}^d\left(T_{n_l,p_l,s_l}F_{n_l/p_l}\right)=\bigotimes_{l=1}^d(T_{n_l,p_l,s_l})\bigotimes_{l=1}^d(F_{n_l/p_l}).$$

We see directly by multilinearity that the second factor sends $\iota_{s_1}(b_{k_1})\otimes\cdots\otimes\iota_{s_d}(b_{k_d})$ to $\prod_{l=1}^d\omega_{n_l}^{s_lk_l}(\iota_{s_1}(b_{k_1})\otimes\cdots\otimes\iota_{s_d}(b_{k_d}))$. So $\bigotimes_{l=1}^d(T_{n_l,p_l,s_l})$ multiplies $X^{(s)}[k_1,\cdots,k_d]$ by $\prod_{l=1}^d\omega_{n_l}^{s_lk_l}$. We use FFTW to calculate the d-dimensional DFT, so this is all we need to know in order to program the first computation step.

The second computation step of the one-dimensional algorithm consists of n/p^2 linear maps, each evaluated on its own data structure. That means we could parallelize this, the only reason we do not is that we do not have sufficient processors. For this reason, the analysis of this map is analogous to the theory we developed on linear BSP-algorithms. The data structures $\{x^{(s)}(t:n/p^2:n/p)\mid t\in[n/p^2]\}$ are obtained from $x^{(s)}$ by distributing it cyclically in n/p^2 parts. So we can calculate the tensor of the second local computation step by distributing dimension-wise and then tensoring the F_{p_l}s. Explicitly, we transform $X^{(s)}[t_l:n_l/p_l^2:n_l/p_l]$ by $F_{p_1}\otimes\cdots\otimes F_{p_d}$ for $0\le t_l<n_l/p_l^2$. This gives the d-dimensional Algorithm 4.

Conclusion

In conclusion, writing a parallel algorithm BSP style enables us to easily generalise algorithms for linear functions to algorithms for tensor products of linear functions. The algorithm obtained this way has the same number of supersteps as the original algorithms, and in a nutshell the new algorithm just performs the old algorithms dimension-wise. In the case of the discrete Fourier transform, this algorithm theoretically performs roughly as well as current state of the art algorithms in all cases where the same number of processors can be used. In some cases, such as very rectangular arrays, more processors can be used, or more processors can be used without incurring extra communication costs (odd dimensions). It also starts and ends in the same distribution by default. This theoretical performance is not always reflected in the experiments, but in the case where we need to start and end in the same distribution, our algorithm performs significantly better whenever more than one node is used. We implemented two versions: in BSPlib and MPI. The BSPlib implementation is slower than its equivalent MPI implementation.

6.5 Further work

On the theoretical part of this thesis, we can apply the theory of Chapter 6 to different tensor products of linear functions, such as the Hartley transform in higher dimensions, or to different BSP algorithms for the discrete Fourier transform. For example, faster algorithms exist when the input is real.

Concerning the implementation of Algorithm 4, someone could copy FFTWs strategy of trying several communication functions, and choosing the fastest with a planner function. Hopefully we can match FFTWs performance in all cases this way. Finally, there are two unanswered questions. Firstly, why PFFT is not slower in three dimensions and 2048 or 4096 processors, than in five dimensions, even though it needs more communication steps. Secondly, why the final transpose of PFFT putting the output in the proper distribution is slower than the two analogous transposes used when calculating the DFT.

Finally, we have the problem of the BSPlib implementation being slower than its MPI equivalent. It is tempting to think that this is a problem inherent to BSPlib and its minimalistic nature, but I can think of no solid argument to prove that. After all, C is a lot more minimalistic than assembly languages, but an optimising compiler can convert a C program to faster assembly than most programmers could write themselves. Perhaps a program similar to a compiler can be written to turn a C file using BSPlib, into a C file using MPI with derived data types and proper collectives (I think MPI_Alltoallv, in some cases on a subgroup of processors, is very close to a BSP communication step). Some simplifying assumptions are probably needed, such as the ability to express the communication pattern at compile time, using only variables that are known at the start of the program (such as the number of processors, and size of the data structure).

Bibliography

[1] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, aug 1978.

[2] Rob H Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Press, USA, second edition, 2020.

[3] Lisandro Dalcin, Mikael Mortensen, and David E. Keyes. Fast parallel multidimensional FFT using advanced MPI. *Journal of Parallel and Distributed Computing*, 128:137–150, 2019.

[4] Matteo Frigo and Steven G. Johnson. The fastest fourier transform in the west. In *the Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing, ICASSP '98*, 1997.

[5] P. Garrett. Tensor products. Available at `https://www-users.math.umn.edu/~garrett/m/algebra/notes/27.pdf`.

[6] Tom Leinster. *Basic category theory*, volume 143. Cambridge University Press, 2014. Available at `https://arxiv.org/pdf/1612.09375.pdf`.

[7] nLab authors. distributive monoidal category. `http://ncatlab.org/nlab/show/distributive%20monoidal%20category`, July 2021. Revision 10.

[8] nLab authors. monoidal category. `http://ncatlab.org/nlab/show/monoidal%20category`, July 2021. Revision 132.

[9] Michael Pippig. PFFT - An extension of FFTW to massively parallel architectures. *SIAM J. Sci. Comput.*, 35:C213 – C236, 2013.

[10] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[11] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA, 1992.

[12] Wijnand Suijlen. BSPonMPI. `https://github.com/wijnand-suijlen/bsponmpi`.