Bachelor Thesis

Multipoint Incremental Fourier Transformation

Author:
Dimitri Degkwitz
ID number:
13-928-734

Supervisors:
Muhammad Saad
Prof. Michael H. Böhlen

Department of Economics
University of Zurich

May 16, 2021

Acknowledgements

I would like to thank my supervisor Muhammad Saad who took time to guide me threw the process of writing this Thesis.

I would like to thank Prof. Michael H. Böhlen and the Institut für Informatik for making this thesis possible.

I would like to thank Jana Cslovjecsek for helping me plan and structure this thesis.

I would like to thank Michael Dohmen, Geethan Karunaratne, Michael, Lucia and Claudia Degkwitz for proofreading this thesis.

I would like to thank Selina Reich for freeing up time on my behalf, so I could focus on this thesis.

Abstract

In radio astronomy the *discrete Fourier Transformation* is crucial for evaluating telescope data. An increased interest in a streaming approach to the Fourier Transformation has led to the development of the *Single Point Incremental Fourier Transformation* (SPIFT), by Saad et al. [3]. SPIFT only handles one datapoint at a time. We developed two new algorithms which evaluate data points in batches. The first algorithm groups datapoints with similar characteristics into a dictionary before SPIFT is applied. The main focus of this report is on the second algorithm, that uses dictionaries and an algorithm we named *Doublestep*. Doublestep uses symmetries in SPIFT to reduce the necessary number of complex additions. In theory MPIFT with dictionaries and Doublestep reduces the asymptotic complexity of SPIFT by $\mathcal{O}(\frac{N}{\log N})$ for batches of smaller sizes. Our implementations confirmed that they reduced the complexity significantly.

Zusasmmenfassung

In der Radioastronomie spielt die *diskrete Fourier Transformation* eine zentrale Rolle bei der Auswertung von Teleskopdaten. Ein erhöhtes Interesse an kontinuierlichen Auswertungen der Daten führte Saad et al. [3] dazu, den *Single Point Incremental Fourier Transformation* (SPIFT) Algorithmus zu entwickeln. SPIFT wertet Datenpunkte einzeln aus. Wir entwickelten zwei neue Algorithmen zur Auswertung von Datenpünkten in Bündeln. Der erste gruppiert Datenpunkte zuerst basierend auf gemeinsamen Charakteristika in Zuordnungstabellen, bevor SPIFT durchgeführt wird. Der Fokus dieser Arbeit ist auf dem zweiten Algorithmus, der neben Zuordnungstabellen auch einen von uns entiwckelten Algorithmus den wir *Doublestep* genannt haben, benutzt. Doublestep nutzt Symetrien in SPIFT um die notwendige Anzahl komplexen Additionen zu reduzieren. Theoretisch reduziert Doublestep die asymptotische Komplexität um $\mathcal{O}(\frac{N}{\log N})$ für kleinere Bündel. Unsere Implementationen haben bestätigt, dass sie die Laufzeit siginifikant reduzieren.

Contents

1 Introduction

In radio astronomy, there has been a shift away from large parabolic dishes and towards fields of antennas [1]. This increases the precision, but also leads to an increase in the amount of data that needs to be evaluated. Planned telescopes often will produce over 100 terabyte of Data per day [4].

The data from such telescopes needs to be processed before researchers can derive meaning from it. A common operation that is needed is the *Discrete Fourier Transformation*. Traditional telescopes wait until the end of the measurements and only then apply the Fourier Transformation on the collected data, usually using the *Fast Fourier Transformation* (FFT) [2]. In view of the volume of storage required for storing the data of the new telescopes, researchers have investigated methods to perform the Fourier Transformation directly on the produced data in a streaming fashion [1]. This method requires less storage capacity and has the bonus, that researchers can start evaluating the data already during the measurements.

In 2020, Saad et al. [3] developed the *Single Point Incremental Fourier Transformation* (SPIFT) algorithms. It allows to update the *image matrix* produced by the Fourier Transformation, as new data points come in. It is intended to be used in a streaming pipeline with radio astronomy telescopes. If a telescope collects data on a raster of size $N \times N$, then SPIFT takes N^2 complex additions and N complex multiplications to update the image matrix. To reduce the time required, SPIFT uses *multiprocessing*, in order to distribute the complex additions over multiple processes.

In this report we analyze the possibility of handling batches of data points to increase throughput while staying true to the streaming idea. Our algorithm, *Multipoint Incremental Fourier Transformation* (MPIFT) *using dictionaries and Doublestep*, decreases computational cost by using shared memory to compute the Fourier Transformation of a batch more efficiently. It combines data points with similar properties and it uses symmetries in the computation of SPIFT to reduce the necessary additions. By doing this, we were able to reduce the computational time by

$$\mathcal{O}(\frac{N}{\log N})$$

for an image-matrix of size $N \times N$.

On tests we performed with MPIFT with dictionaries and Doublestep and a Naive MPIFT implementation, the former outperformed the latter with a ratio of $\frac{1}{243}$, for N=8192 and a batch-size of 42642. When MPIFT just combined datapoints with similar properties, an implementation using Doublestep still outperformed it with a ratio of $\frac{1}{85}$.

In Section 2.1 we define the problem we are trying to solve. Then in Section 2.2 we explain how the existing SPIFT algorithm functions.

In Section 2.3 we explain what batch-handling entails and introduce the three implementations we will compare in this report. In Subsection 2.3.1 we introduce the naive way of applying SPIFT to batches. In Section 2.3.2 we explain how similar data points can be bundled and we show an example in Section 2.3.3.

Section 2.4 introduces Doublestep. In Subsection 2.4.1 we explain the basic idea behind the concept and in Subsection 2.4.2 how Doublestep works algorithmically. Subsection 2.4.3 calculates the complexity of Doublestep and Subsection 2.4.4 gives a working example of Doublestep.

In Section 2.5 we expand Doublestep to work with multiple processes. Subsection 2.5.1 explains how to use Doublestep with multiple processes and row-shiftable visibilities that were introduced in Section 2.2. Subsection 2.5.2 does the same for column-shiftable visibilities. 2.5.3 discusses the complexity of using Doublestep with multiple processes.

In Section 2.6 we compare the three implementations of MPIFT we introduced.

In Section 2.7 we describe the experiments we ran on our implementations. Subsection 2.7.1 describes the setup we used and Subsection 2.7.2 shows and explains the results obtained.

In Section 3 we discuss the results of our findings and give suggestions for possible future research.

2 Main Body

2.1 Problem definition

SPIFT (Single Point Incremental Fourier Transform) is an algorithm developed by Saad et all [3]. It takes a stream of incoming Datapoint u,v,vis, where (u, v) are discrete 2D coordinates, and vis is the complex value measured at (u, v). As the name implies, SPIFT performs a incremental Fourier Transformation on this data stream. This means that preliminary results of the Fourier-Transformation are always available, leading to an image matrix I_t, that represents the result of the Fourier-Transformation after visibility (u_t, v_t, vis_t) has been taken into account.

SPIFT achieves this more efficiently than the traditional *Discrete Fourier Transformation* (DFT) by doing two things. For one, it uses symmetries in the computation of DFT to reduce the number of complex operations. But it also uses multiprocessing, to distribute the work and increase efficiency. SPIFT only works for matrices of a size that is a prime numbers or a powers of two. In this report, ω will denote how many machines are available to an algorithm, and λ how many processes can run on each machine in parallel. Further, $\phi = \omega \cdot \lambda$ denotes the total number of processes available. Since ϕ should divide N, we will restrict our self to N and ϕ that are powers of two. Consequently ω and λ are powers of two as well.

SPIFT can still be too slow for practical use. The idea of this report is to try to improve the efficiency of SPIFT further, by considering batch-processing. In this report we introduce the MPIFT, that calculates the next image-matrix for a batch of visibilities at once. Since it works on multiple points, we call it Multipoint Incremental Fourier Transformation (MPIFT). The amount of points per batch is denoted by β in this report. Table 1 shows the notations we will use.

Notation	Description
N	Amount of rows and columns in Image-matrix
log	the logarithm of base two
n	$\log(N)$
β	batch size
λ	number of local processes
ω	number of machines
$\phi = \lambda \cdot \omega$	total degree of parallelism

Table 1: Notation used in this report

2.2 SPIFT

The Discrete Fourier Transformation updates the value of the Image-Matrix when a new visibility comes in, in the following way:

$$I_t = I_{t-1} + \text{vis}_t \cdot L^{(u,v)}$$

where:

$$L^{(u,v)} = \begin{bmatrix} l_{0,0} & l_{0,1} & ... & l_{0,N-1} \\ l_{1,0} & l_{1,1} & ... & l_{1,N-1} \\ ... & ... & ... & ... \\ l_{N-1,0} & l_{N-1,1} & ... & l_{N-1,N-1} \end{bmatrix} \text{ with } l_{j,k} = (e^{\frac{i \cdot 2 \cdot \pi}{N}})^{u \cdot j + v \cdot k}.$$

and

$$I_0 = \begin{bmatrix} 0 & 0 & ... & 0 \\ 0 & 0 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 0 \end{bmatrix} \text{ a } N \times N \text{ matrix.}$$

The variables $l_{j,k}$ are called *twiddle factors*. Saad et al used the fact that for a fixed N, the twiddle factors can only have N distinct values. They found that for values of N that are prime or power of two, they could compute the image matrix much more efficiently. Under those conditions, every visibility can be column-shifted or row-shifted. Column-shifted means, that all columns are the same as the first one, except that they are shifted in regards to the previous one by a fixed amount. Figure 1 shows $L^{(1,2)}$ for $N = 4$, which is column-shift-able and has shift-index 2. As we can see the values of the first column repeat in the same order in every column. The only difference is, that they are moved down by two with respect to the previous column. Row-shift works analogous. Algorithm 1 show, how to check if a visibility is column-shift-able and how to calculate the shift-index using the *greatest common divisor* (gcd).

$$\begin{pmatrix} 1 & -1 & 1 & -1 \\ i & -i & i & -i \\ -1 & 1 & -1 & 1 \\ -i & i & -i & i \end{pmatrix}$$

Figure 1: $L^{(1,2)}$ for N = 4

After knowing shift-type and shift-index of a visibility, SPIFT calculates it's shift-vector. This is simply the first column or first row, depending on type, of $L^{u,v}$, multiplied with the visibility. By multiplying the visibility beforehand only N complex multiplications are required, instead of the N^2 ones that would happen, if it was multiplied with the computed $L^{(u,v)}$ afterwards.

When using SPIFT, the image-matrix gets divided into ϕ slices. This can be done by row or by column, either creating

$$\frac{N}{\omega \cdot \lambda} \times N \text{ or } N \times \frac{N}{\phi}$$

image-matrix slices. Each process gets one slice assigned. When performing SPIFT, each process updates its slice by shifting the shift-vector over it. Algorithm 3 shows how SPIFT is executed on one process using Algorithm 2 that updates it's image matrix when processing a visibility. In this example the image-matrix is sliced into sub-matrices of size $\frac{N}{\phi} \times N$ and the variable *key* $\in \{0, 1, \ldots, \phi - 1\}$ indicates the process.

2.3 SPIFT with batch

In this section, we will present three different version of MPIFT, that extend SPIFT to handle batch visibilities of batch-size β. The first Subsec-

Algorithm 1 CalculatingShiftIndex(u,v)

```
1: $isCS \leftarrow (v = 0)$ **or** $(u\%2 = 1$ **and** $v\%2 = 0)$ **or**
       $(v\%2 = 0$ **and** $\gcd(u, N) < \gcd(v, N))$
2: **if** $u = 0$ or $v = 0$ **then**
3:     $shiftIndex \leftarrow 0$
4: **else**
5:     **if** $isCS$ **then**
6:         **for** $j \leftarrow 0$ to $N$ **do**
7:             **if** $v = j \cdot u\%N$ **then**
8:                 $shiftIndex \leftarrow j$
9:     **else**
10:        **for** $j \leftarrow 0$ to $N$ **do**
11:            **if** $u = j \cdot v\%N$ **then**
12:                $shiftIndex \leftarrow j$
13: **return** $(isCS, shiftIndex)$
```

Algorithm 2 CalculatingSPIFTnode($I_{t-1,key}, isCS, shiftIndex, shiftVector, key$)

```
1: $rows \leftarrow N/(\phi)$
2: **if** isCS **then**
3:     **for** $k \leftarrow 0$ to $N$ **do**
4:         $startIdx \leftarrow ((key \cdot rows) + (shiftIndex \cdot k))\%N$
5:         **for** $j \leftarrow 0$ to $rows$ **do**
6:             $idx \leftarrow (startIdx + j)\%N$
7:             $I_{t,key}[j, k] \leftarrow I_{t-1,key}[j, k] + shiftVector[idx]$
8: **else**
9:     **for** $j \leftarrow 0$ to $rows$ **do**
10:        $startIdx \leftarrow (shiftIndex \cdot (j + key \cdot rows))\%N$
11:        **for** $k \leftarrow 0$ to $N$ **do**
12:            $idx \leftarrow (startIdx + k)\%N$
13:            $I_{t,key}[j, k] \leftarrow I_{t-1,key}[j, k] + shiftVector[idx]$
```

Algorithm 3 SPIFT(*visibility*, *key*)

```
1: $isCS, shiftIndex =$
       $CalculatingShiftIndex(visibilities.u, visibilities.v)$
2: $vis, u, v \leftarrow visibility.vis, visibility.u, visibility.v$
3: $shiftVector \leftarrow$ emptyvector($N$)
4: for $k \leftarrow 0$ to $N$ do
5:     if ($isCS$) then
6:         $shiftVector[k] \leftarrow vis \cdot W^{(k \cdot u)\%N}$
7:     else
8:         $shiftVector[k] \leftarrow vis \cdot W^{(k \cdot v)\%N}$
9: CalculatingSPIFTnode($I_{t-1,key}, isCS, shiftIndex, shiftVector, key$)
```

tion (2.3.1) presents a naive approach, that runs every visibility in a batch through plain SPIFT. We call this naive MPIFT. The second Subsection (2.3.2) combines visibilities with the same shift-type and shift-index before performing SPIFT. We call this MPIFT with dictionaries. In the last Section (2.5) we use symmetries in addition of shift-vectors to reduce the amount of complex additions that are needed. We call this MPIFT with dictionaries and Doublestep.

2.3.1 Naive MPIFT

A naive approach to handle batch-processing using SPIFT would be to simply feed every visibility in a batch to SPIFT and have it calculate the new image-matrices. Algorithm 4 implements this idea. The *key* $\in \{0, 1, \ldots, \omega - 1\}$ refers to the process. This is intuitively a bad approach, but we will use it as a benchmark to compare the other approaches to. Figure 2 shows the flow of naive MPIFT.

Calculating one visibility with SPIFT takes N complex multiplications to calculate the shift-vector, and $\frac{N^2}{\phi}$ complex additions per process to add it to the image matrix. So handling the full batch would lead to $N \cdot \beta$ complex multiplications and $\frac{N^2 \cdot \beta}{\phi}$ complex multiplications.

Algorithm 4 naiveMPIFT($visibilities_t$, *key*)

```
1: for $visibility$ in $visibilities_t$ do
2:     SPIFT($visibility, key$)
```

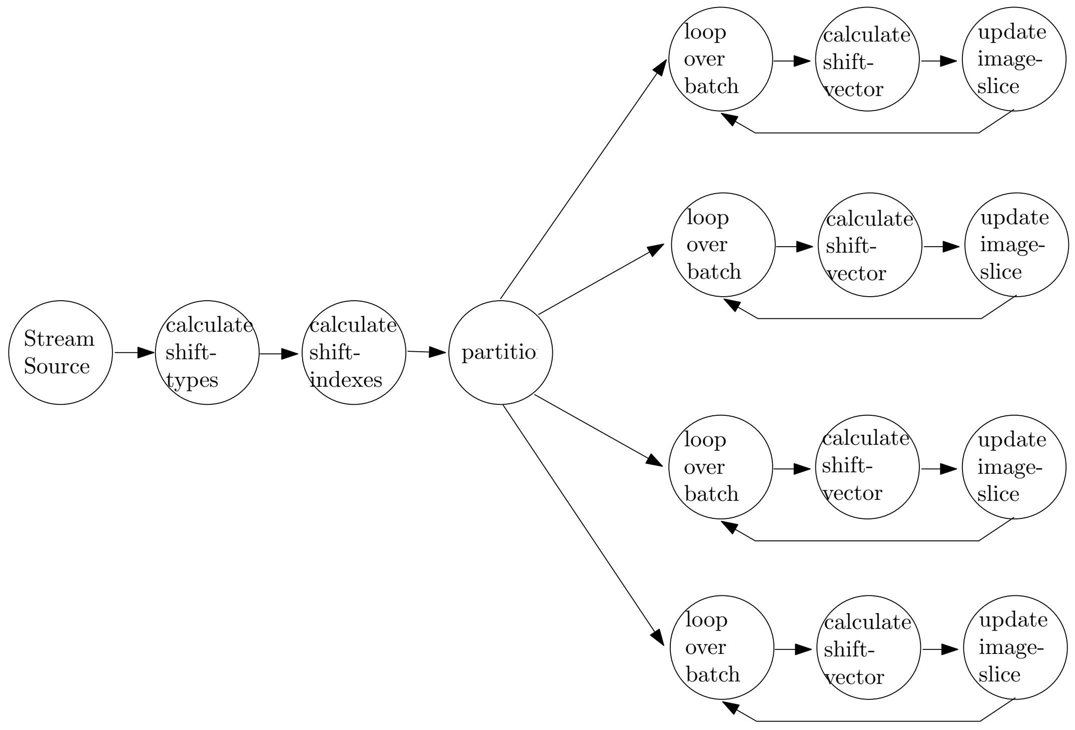

Figure 2: Flow chart of naive MPIFT using one master node that does initial calculation and passes it to $\phi = 4$ processes

2.3.2 MPIFT with Shift-Dictionary

Consider two visibilities in the same batch, $visibility_1$ and $visibility_2$. If they both have the same shift-type and shift-index, then we can add their shift-vectors together and handle it as one thing. This saves $\frac{N^2}{\phi}$ complex additions and costs Ncomplex multiplications. If we consider that:

$$\phi \text{ divides } N \implies \phi \leq N \implies N \leq \frac{N^2}{\phi}.$$

So combining the two visibilities is at leas as efficient as handling them separately.

The idea of shift-dictionaries is to create a row-shift-dictionary and a column-shift-dictionary. These dictionaries contain the N combined shift-vectors for the N shift-indices. Algorithm 5 shows how to compute the shift-dictionary.

Since communications-costs between different machines is high, it is inefficient to compute the shift-vectors on the master-node and then send them to each process. But on a physical machine, processes can use shared memory, basically eliminating the communications-costs.

The intuitive way to do this, is to divide the visbilities among the local processes, to ensure everyone does the same amount of work. But this could lead to a situation where two processes are handling two visibilities with the same shift-type and shift-index at the same time. They would both access the same part of the shift-dictionary at the same time, leading to simultaneous access and possible loss of data. This could be solved by locks, but this would increase the amount of computational overhead, and would defeat the purpose of every process taking the same amount of time, since processes would need to wait for memory access.

Instead, we can order the visibilities by shift-type and shift-index in the master-node before sending them. This doesn't increase the communication-costs. Then we divide the possible shift-types and shift-indices among the processes, and have them each compute an equal part of the shift-dictionary. If the visibilities in the batch are well distributed, this would lead to an acceptable distribution of work. Figure 3 shows the flow of creating the dictionaries and calculating MPIFT.

The first entry-addition of a shift-vector into a shift-dictionary doesn't need to be added, all following ones do. So in the worst case, all visibilities have the same shift-type and shift-index. This would mean that one process has to calculate $N \cdot (\beta - 1)$ complex additions and all $\beta \cdot N$ complex multiplications. In the best case the visibilities are evenly distributed among the shift-types and -indices. This leads to $\frac{\beta \cdot N}{\lambda}$ complex multiplications and $N \cdot \frac{\beta - N}{\lambda}$ complex additions. Since β usually is large, the average case is much closer to the best case than to the worst case.

Algorithm 5 CalculatingShiftDictionary($visibilities_t, N$)

```
1:  $dRow = new$ dic
2:  $dCol = new$ dic
3:  for $(u_t, v_t, \mathrm{vis}_t)$ in $visibilities_t$ do
4:      $isCS, si =$CalculatingShiftIndex$(u_t, v_t)$
5:      $sv \leftarrow$ emptyvector$(N)$
6:      for $k \leftarrow 0$ to $N$ do
7:          if $(isCS)$ then
8:              $sv[k] \leftarrow vis \cdot W^{(k \cdot u)\%N}$
9:          else
10:             $sv[k] \leftarrow vis \cdot W^{(k \cdot v)\%N}$
11:     if isCS then
12:         if $dCol$ contains $si$ then
13:             $dCol[si] \leftarrow dCol[si] + sv$
14:         else
15:             $dCol[si] \leftarrow sv$
16:     else
17:         if $rCol$ contains $si$ then
18:             $rCol[si] \leftarrow rCol[si] + sv$
19:         else
20:             $rCol[si] \leftarrow sv$
    return $(dRow_t, dCol_t)$
```

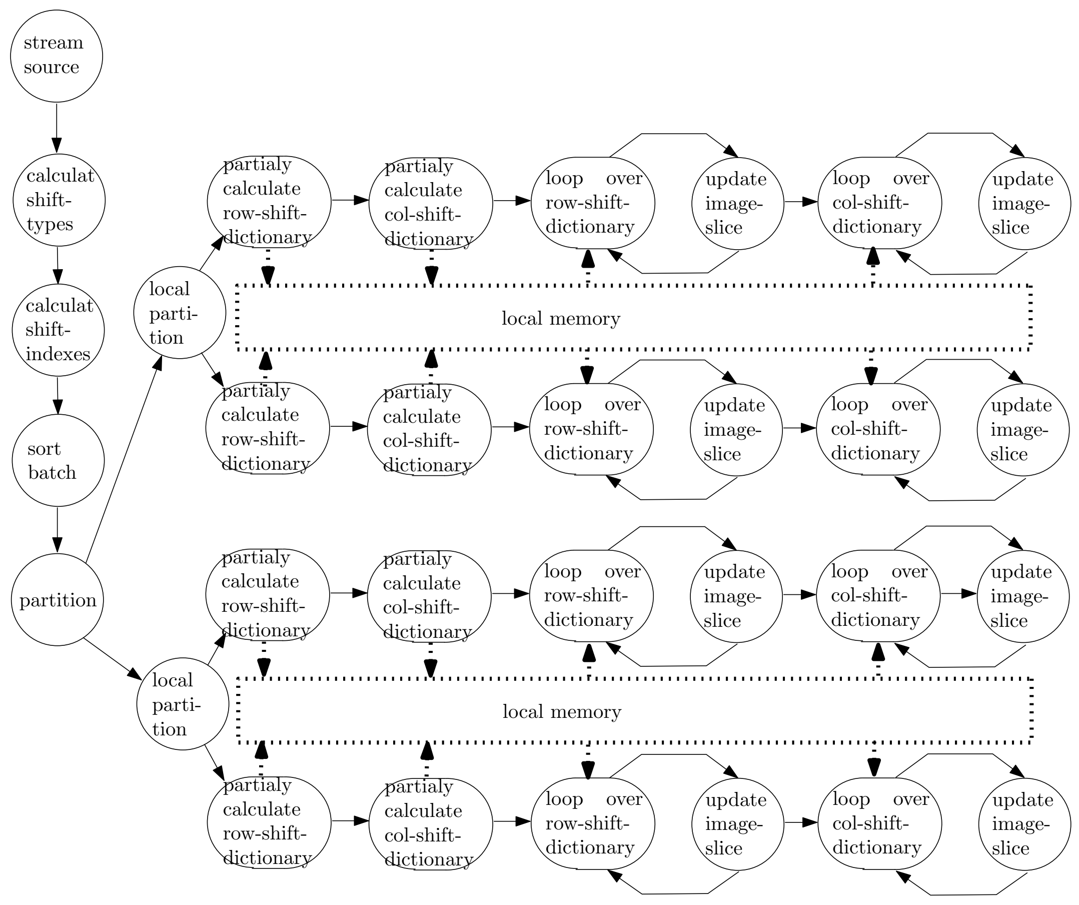

Figure 3: Flow chart of MPIFT with shift-dictionaries. One master node does initial calculation and sorts the batch and passes it to $\omega = 2$ machines running $\lambda = 2$ processes each. Each machine has local memory that is shared between processes of that machine.

2.3.3 Example Shift-Dictionary

In this section we will give a small example how to calculate a shift-dictionary. We will consider measurements on $N = 4$ with an incoming batch of size $\beta = 12$, shown in Table 2. To keep it simple, we ignore that the visibilities would be sorted and handled by multiple processes.

We first create an empty row-shift dictionary and an empty column-shift dictionary. The first three values can simply be added to them, since they

Number	u	v	vis	$isCS$	shift-index	shift-vector
1	1	2	$1 + 0i$	true	2	(1+0i,0+1i,-1+0i,0-1i)
2	0	2	$2 + 0i$	false	0	(2+0i,-2+0i,2+0i,-2+0i)
3	2	1	$3 + 0i$	false	2	(3+0i,0+3i,-3+0i,0-3i)
4	0	3	$0 + 4i$	false	0	(0+4i,4+0i,0-4j,-4+0j)
5	3	1	$5 + 0i$	false	3	(5+0i,0+5i,-5+0i,0-5i)
6	1	0	$0 + 6i$	true	0	(0+6i,-6+0i,0-6i,6+0i)
7	3	3	$7 + 7i$	false	1	(7+7i,7-7i,-7-7i,-7+7i)
8	1	1	$8 + 0i$	false	1	(8+0i,0+8i,-8+0i,0-8i)
9	0	3	$0 + 9i$	false	0	(0+9i,9+0,0-9i,-9+0i)
10	2	1	$10 + 10i$	false	2	(10+10i,-10+10i,-10-10i,10-10i)
11	3	3	$11 + 0i$	false	1	(11+0i,0-11i,-11+0i,0+11i)
12	2	0	$0 + 12i$	true	0	(0+12i,0-12i,0+12i,0-12i)

Table 2: Example Batch for a 4×4 image matrix

differ in shift-index and shift-type. Table 3 shows what the shift-dictionaries would look like after this.

The fourth visibility is row-shiftable and has a shift-index of 0. The dictionary already has an entry for this, so the shift-vector gets added to the existing one. This is shown in Table 4. The other visibilities then get added in the same fashion. Table 5 shows what the shift-dictionary looks like after all 12 values have been added.

column-shift-dictionary	
0	2
–	$1 + 0i$
–	$0 + 1i$
–	$-1 + 0i$
–	$0 - 1i$ ”

row-shift-dictionary			
0	1	2	3
$2 + 0i$	–	$3 + 0i$	–
$-2 + 0i$	–	$0 + 3i$	–
$2 + 0i$	–	$-3 + 0i$	–
$-2 + 0i$	–	$0 - 3i$	–

Table 3: Shift Dictionaries after first 3 visibilities from Table 2 have been added

column-shift-dictionary	
0	2
–	$1 + 0i$
–	$0 + 1i$
–	$-1 + 0i$
–	$0 - 1i$ ”

row-shift-dictionary			
0	1	2	3
$2 + 4i$	–	$3 + 0i$	–
$2 + 0i$	–	$0 + 3i$	–
$2 - 4i$	–	$-3 + 0i$	–
$-6 + 0i$	–	$0 - 3i$	–

Table 4: Shift Dictionaries after first four value from batch in Table 2 was added to Table 3

column-shift-dictionary	
0	2
$0 + 18i$	$1 + 0i$
$-6 - 12i$	$0 + 1i$
$0 + 6i$	$-1 + 0i$
$6 - 12i$	$0 - 1i$ ”

row-shift-dictionary			
0	1	2	3
$2 + 13i$	$26 + 7i$	$13 + 10i$	$5 + 0i$
$11 + 0i$	$7 - 10i$	$-10 + 13i$	$0 + 5i$
$2 - 13i$	$-26 - 7i$	$-13 - 10i$	$-5 + 0i$
$-15 + 0i$	$-7 + 10i$	$10 - 13i$	$0 - 5i$

Table 5: Shift Dictionaries after adding all visibilities from Table 2

2.4 Doublestep

Doublestep is an algorithm that we have developed to reduce the number of complex additions when updating the image matrix. In this section we will discuss how Doublestep works, not considering multiple processes. In Section 2.5 we will extend the algorithm to work with multiple processes and shared memory.

2.4.1 Idea

Consider two shift-vectors a_1 and a_2 with shift-index s_1 and s_2 where $s_2 = s_1 + \frac{N}{2}$ which are both row-shiftable. In the first row, $\vec{a}_1$ and $\vec{a}_2$ are added

$$(a_1^0 + a_2^0).$$

The superscript denotes by how much the vector is shifted. On the next row, we get

$$(a_1^{s_1} + a_2^{s_2}).$$

On the third row, it is

$$a_1^{2\cdot s_1} + a_2^{2\cdot s_2} = a_1^{2\cdot s_1} + a_2^{2\cdot s_1 + 2\cdot \frac{N}{2}} = a_1^{2\cdot s_1} + a_2^{2\cdot s_1} = (a_1^0 + a_2^0)^{2\cdot s_1},$$

and on the forth

$$a_1^{3\cdot s_1} + a_2^{3\cdot s_2} = a_1^{3\cdot s_1} + a_2^{3\cdot s_1 + \frac{N}{2}} = (a_1^{s_1} + a_2^{s_1+\frac{N}{2}})^{2\cdot s_1}$$

If we continue this line of thought, we get $(a_1^0 + a_2^0)^{4\cdot s_1}$ for the fifth and $(a_1^{s_1} + a_2^{s_1+\frac{N}{2}})^{4\cdot s_1}$ for the sixth row.

As we can see, we only need to perform two vector-additions, $a_1^0 + a_2^0$ and $a_1^{s_1} + a_2^{s_2}$. These two rows form a $2 \times N$-matrix. If we shift this matrix by a multiple of $2 \cdot s_1$, it corresponds to the shifted addition of a_1 and a_2. We will call this matrix $m^2_{2\cdot s_1}$, where the subscript denotes the shift-index of the shift-matrix, and the superscript denotes the number of rows. We can do this with all shift-vector-pairs to get $\frac{N}{2}$ shift-matrices, that all have an even shift-index.

Consider now two shift-matrices m_s^2 and $m^2_{s+\frac{N}{2}}$. In the first two rows of the image matrix the two matrices are added together:

$$(m_s^{2,0} + m_{s+\frac{N}{2}}^{2,0}).$$

The second superfix denotes the shift. In the third and fourth row, they both get shifted by s and $s + \frac{N}{2}$ respectively

$$(m_s^{2,s} + m_{s+\frac{N}{2}}^{2,s+\frac{N}{2}}).$$

The fifth and sixth row are again just the result from the first two rows, shifted by $2 \cdot s$,

$$(m_s^{2,2\cdot s} + m_{s+\frac{N}{2}}^{2,2\cdot s+2\cdot\frac{N}{2}}) = (m_s^{2,0} + m_{s+\frac{N}{2}}^{2,0})^{2\cdot s}.$$

Similarly, the seventh and eight row are simply the third and fourth, shifted by $2 \cdot s$,

$$\left(m_s^{2,3\cdot s} + m_{s+\frac{N}{2}}^{2,3\cdot s+3\cdot\frac{N}{2}}\right) = \left(m_s^{2,s} + m_{s+\frac{N}{2}}^{2,s+\frac{N}{2}}\right)^{2\cdot s}.$$

As we see, we can use the same method as before to calculate $4 \times N$ shift-matrices. We then use the same method to calculate the shift-matrices of size $8 \times N$. We call the calculation of a new set of shift-matrices a step of Doublestep.

In the *i-th* step of Doublestep, we calculate 2^{n-i} matrices of size $k \times N$, $k = 2^i$. Since the shift-matrices of step i must have shift-indices that are twice the shift-index of a matrix of step *i-1*, all shift-matrices of step i are multiples of k. The shift-matrices of the step i can be calculated with using equation (1), where $k = 2^i$ and s is a multiple of k that is smaller than N. $\times$ denotes the concatenation of two matrices. Figure 4 shows how shift-matrices combine to make shift-matrices of a higher degree.

$$[h]m_s^k = (m_{\frac{s}{2}}^{\frac{k}{2},0} + m_{\frac{s+N}{2}}^{\frac{k}{2},0}) \times (m_{\frac{s}{2}}^{\frac{k}{2},\frac{s}{2}} + m_{\frac{s+N}{2}}^{\frac{k}{2},\frac{s+N}{2}}) \tag{1}$$

At the n^{th} step of Doublestep, we get just one $N \times N$ matrix with shift-index 0, m_0^N. Adding this matrix to the old image-matrix is equivalent to updating it with all visibilities from the batch.

2.4.2 MPIFT with dictionaries and Doublestep algorithm

Algorithm 6 performs MPIFT with dictionaries and Doublestep, which consists of three phases. In the first phase, the shift-dictionary is calculated. This happens in the same way as in the MPIFT with dictionary algorithm (Algorithm 5). In the second phase, the steps of Doublestep are performed for row- and column-shifts, to calculate the two shift-matrices $m_{0,row}^N$ and $m_{0,col}^N$. This is shown in Algorithm 7. In the third and final step, the two shift-matrices are added to the existing image matrix.

Algorithm 6 MPIFTwithDictionariesAndDoublestep($visibilities_t, N, I_t$)

```
1: $dRow_t, dCol = CalculatingShiftDictionary(visibilities_t, N)$
2: $(m_{0,row}^N, m_{0,col}^N) = Doublestep(dRow_t, dCol_t, N)$
3: $I_{t+1} \leftarrow I_t + m_{0,row}^N + m_{0,col}^N$
4: return $I_{t+1}$
```

2.4.3 Complexity

Calculating a single shift-vector implies N complex multiplications. This means that $N \cdot \beta$ complex multiplications are required for the full shift-dictionary. The first shift-vector for each shift-index does not need to be added to anything. But each subsequent shift-vector with the same shift-index must be added to the existing one. Each vector-addition needs N

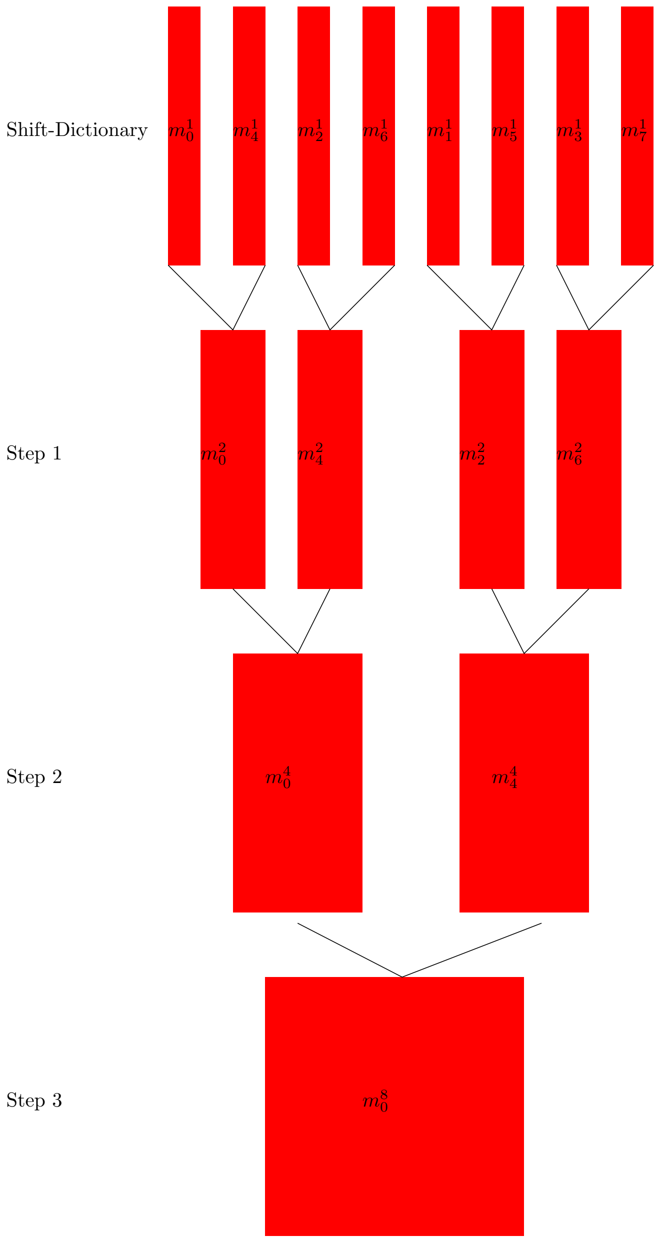

Figure 4: Illustration on how Doublestep combines smaller shift-matrices to build bigger ones for column-shift and $N = 8$

Algorithm 7 Doublestep($dRow_t, dCol_t, N$)

```
1: for i ← 0 to $N$ do
2:     $m^1_{i,row} = dRow[i]$
3:     $m^1_{i,col} = dCol[i]$
4: for $i \leftarrow 1$ to $\log(N)$ do
5:     $k = 2^i$
6:     for $s \leftarrow 0, k, 2k$ to $N - k$ do
7:         $m^k_{s,row} \leftarrow (m^{\frac{k}{2},0}_{\frac{s}{2},row} + m^{\frac{k}{2},0}_{\frac{s+N}{2},row}) \times (m^{\frac{k}{2},\frac{s}{2}}_{\frac{s}{2},row} + m^{\frac{k}{2},\frac{s+N}{2}}_{\frac{s+N}{2},row})$
8:         $m^k_{s,col} \leftarrow (m^{\frac{k}{2},0}_{\frac{s}{2},col} + m^{\frac{k}{2},0}_{\frac{s+N}{2},col}) \times (m^{\frac{k}{2},\frac{s}{2}}_{\frac{s}{2},col} + m^{\frac{k}{2},\frac{s+N}{2}}_{\frac{s+N}{2},col})$
   return $(m^N_{0,row}, m^N_{0,col})$
```

complex additions. So, in the best case, there is a visibility for every shift-index, and $N \cdot (\beta - \frac{3N}{2})$ complex additions are required. In the worst case, all visibilities have the same shift-index, and $N \cdot (\beta - 1)$ complex additions are required. With large buffer-size, it does not matter and we can approximate and say it requires $N \cdot \beta$ complex additions.

In each step of Doublestep, $\frac{N}{2^i} = \frac{N}{k}$ shift-matrices of size $k \times N$ or $N \times k$ need to be calculated. Each requires $k \cdot N$ complex additions, so one step requires

$$k \cdot N \cdot \frac{N}{k} = N^2$$

complex additions.

Doublestep has n steps for row- and n steps for column-shift. In total, we therefore get

$$2 \cdot n \cdot N^2 = 2 \cdot \log(N) \cdot N^2$$

complex additions

The two final shift-matrices m^N_0 for row- and column-shift are added to the existing image-matrix. Both have size $N \times N$. Accordingly, $2 \cdot N^2$ complex additions are required for this step.

Complex multiplications only appear in the calculation of the shift-dictionary. Consequently

$$N \cdot \beta$$

complex additions are required. Complex additions are required in for the shift-dictionary and Doublestep.

$$N \cdot \beta + 2 \cdot \log(N) \cdot N^2 + 2 \cdot N^2$$

complex additions are therefore approximately needed to perform MPIFT with dictionaries and Doublestep, which means it has an asymptotic complexity of

$$\mathcal{O}(N\beta)$$

complex multiplications and

$$\mathcal{O}(N\beta + N^2 \log(N))$$

complex additions.

2.4.4 Example

This section shows an example of how Doublestep works. We will continue using the example from Sections 2.3.3, and compute Doublestep on the row-shift-dictionary from Figure 3. From that section we know that $N = 4$. Since the row-shift-dictionary is already computed, we already have all shift-matrices of size 1×4. Each matrix corresponds to the entry in the shift-dictionary with the same shift-index, as shown in Figure 5.

$$\begin{aligned}
m_0^1 &= \big((2+13i) \quad (11+0i) \quad (2-13i) \quad (-15+0i)\big) \\
m_1^1 &= \big((26+7i) \quad (7-10i) \quad (-26-7i) \quad (-7+10i)\big) \\
m_2^1 &= \big((13+10i) \quad (-10-13i) \quad (-13-10i) \quad (10-13i)\big) \\
m_1^1 &= \big((5+0i) \quad (0+5i) \quad (-5+0i) \quad (0-5i)\big)
\end{aligned}$$

Figure 5: Doublestep example step 0

Since $N = 4$, we know that $n = 2$, so we need to compute 2 steps. First, we will calculate m_0^2 and m_2^2. To calculate m_0^2, we need

$$m_{\frac{0}{2}}^{\frac{2}{2}} = m_0^1 \text{ and } m_{\frac{0+4}{2}\%4}^{\frac{2}{2}} = m_2^1.$$

The first row of m_0^2 is m_0^1 and m_0^2 added together without any shift. For the next row, both m_0^1 and m_2^1 are shifted by their shift-index. In the case of m_0^1 this means that it is shifted by 0, i.e. not actually shifted.

For m_2^2 we need

$$m_{\frac{2}{2}}^{\frac{2}{2}} = m_1^1 \text{ and } m_{\frac{2+4}{2}\%4}^{\frac{2}{2}} = m_3^1.$$

Again, the first row is just the two rows without any shift happening. In the second row both vectors get shifted by their shift-index, so 1 and 3. The result can be seen in Figure 6.

Note that the rule is not, that the first row is not shifted, and the second is. For a shift-matrix m_i^d, the first $\frac{d}{2}$ rows are not shifted. In step one this is just one row.

$$m_0^2 = \begin{pmatrix} (15+23i) & (1-13i) & (-11-23i) & (-5-13i) \\ (-11-3i) & (22-13i) & (15-3i) & (-25-13i) \end{pmatrix}$$
$$m_2^2 = \begin{pmatrix} (31+7i) & (7-5i) & (-31-7i) & (-7-5i) \\ (-7+15i) & (21+7i) & (7-15i) & (-21-7i) \end{pmatrix}$$

Figure 6: Doublestep example step 1

In step 2 we calculate one shift-matrix m_0^4. It is composed of

$$m_{\frac{0}{2}}^{\frac{4}{2}} = m_0^2 \text{ and } m_{\frac{0+4}{2}\%4}^{\frac{4}{2}} = m_2^2.$$

For the first two rows, we just add the values of m_0^2 and m_2^2. For the third and fourth row, we shift m_0^2 by 0 (which is again the same as not shifting), and m_2^2 by 2, before adding them together. This is shown in Figure 7. Thus we have calculated the final shift-matrix. Adding this to the existing image-matrix will be equivalent to performing SPIFT on all twelve visibilities of Table 2.

$$m_0^4 = \begin{pmatrix} (46+30i) & (8-18i) & (-42-30i) & (-12-18i) \\ (-18+12i) & (43-6i) & (22-18i) & (-46-20i) \\ (16+16i) & (-6-18i) & (20-16i) & (2-18i) \\ (-4-18i) & (1-20i) & (8+12i) & (-4-6i) \end{pmatrix}$$

Figure 7: Doublestep example step 2

2.5 Multiprocessing MPIFT with dictionaries and Doublestep

In Section 2.4 we only used one process. MPIFT can be improved by using multiple processes to calculate the shift-dictionaries and Doublestep. For the processes to cooperate in performing Doublestep, they need to share a lot of data. In practice, this is only feasible if the processes run on the same physical machine, otherwise the communication costs become too large. When performing MPIFT with dictionaries and Doublestep in parallel, we divide the image matrix into sub-grids. Unlike naive MPIFT, not every process gets its own sub-matrix. Rather, all processes on the same machine work together to calculate one sub-matrix. So, we end up with ω sub-matrices. Figure 8 shows the flow of MPIFT with dictionaries and Doublestep.

When slicing the image-matrix into sub-matrices, we can either slice with the rows or with the columns. If we slice with the rows we get sub-matrices of size $D \times N$, if we slice with columns they are of size $N \times D$, where $D = \frac{N}{\omega}$. The choice is arbitrary. In this report, we assume that we sliced with the rows.

After having decided if we slice with rows or columns, some vector-shifts will go with the slice, and others against it. Here the row-shift goes with and column-shift against the slice. This has ramifications on how Doublestep is performed. We need to compute row-Doublestep differently then column-Doublestep.

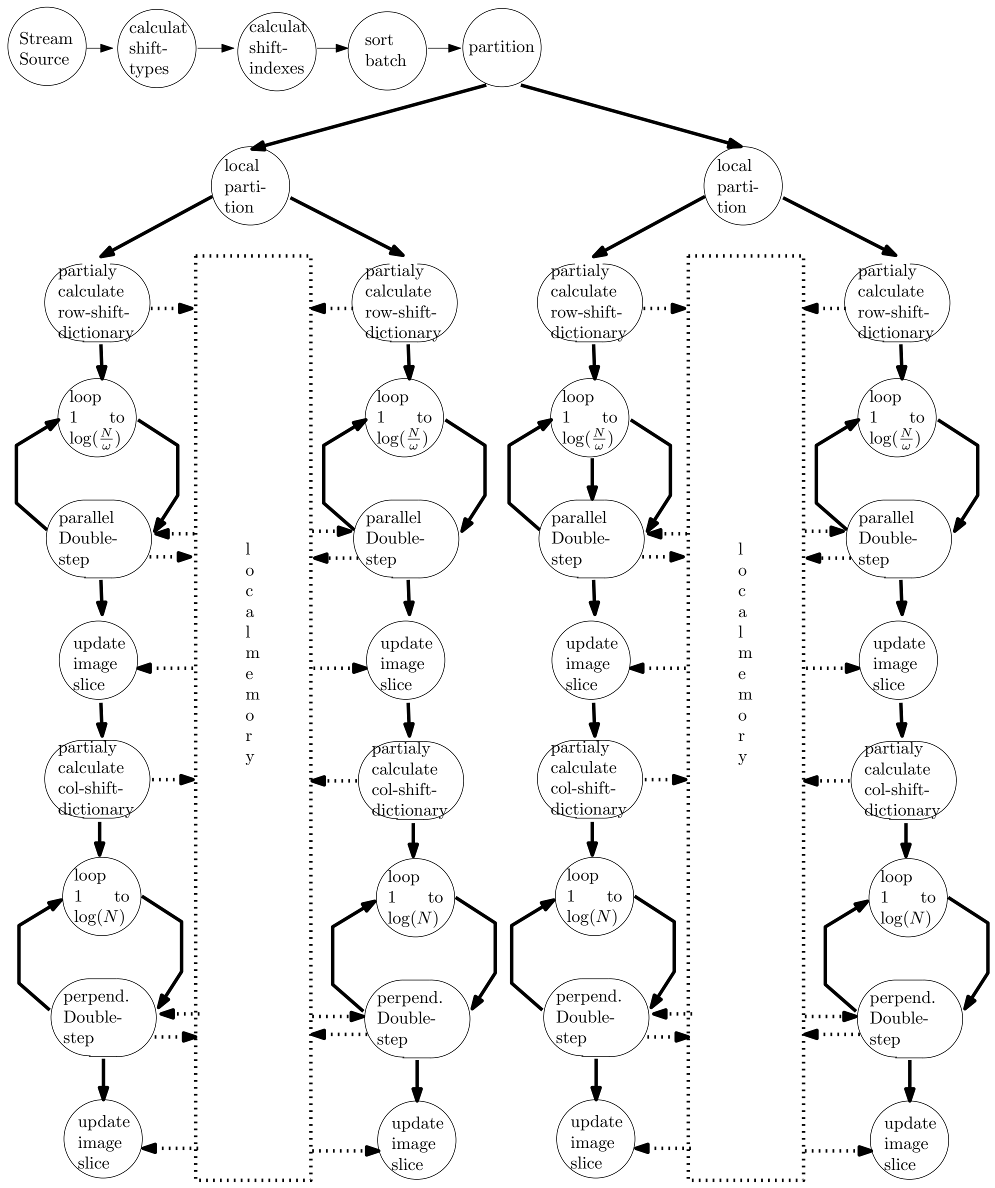

Figure 8: Flow chart of MPIFT with dictionaries and Doublestep. One master node does initial calculations, sorts the batch and passes it to $\omega = 2$ machines running $\lambda = 2$ processes each. Each machine has local memory that is shared between processes of that machine.

2.5.1 MPIFT with dictionaries and row-Doublestep

In the row-shift case, a machine first receives a dictionary containing all visibilities with row-shift, organized by shift-index. Since all row-shift-vectors are needed in every slice, each machine needs to calculate the full shift-dictionary first. All processes can work together on this, using shared memory. This is done the same way as in Section 2.3.2.

N shift-vectors need to be calculated and each machine has λ local processes to do so. Each process has to calculate r row-shift-vectors and store them in the row-shift-dictionary, where $r = \frac{N}{\lambda}$. Since $N > \lambda$ and both N and λ are powers of two, this is always possible. Each process can look up the visibilities it needs to calculate its shift-vectors. Since it is possible that a shift-index has more visibilities associated with it, some processes will take longer than others. The processes cannot work together to share the work more evenly, because this would mean that multiple process must write to the same part of memory in the shift-dictionary, leading to multithreading problems.

After having calculated the row-shift-dictionary, row-Doublestep needs to be performed. Since in the end, only a $D \times N$, with $D = \frac{N}{\omega}$, slice of the image-matrix needs to be calculated, it is more efficient to only use Doublestep until a shift-matrix size of $D \times N$ is reached. Since N and β are powers of two, there will always be a set of shift-matrices with exactly this size. They are the shift-matrices $m^{D}_{i \cdot D}$, with $i \in \{0, 1, ..., \frac{N}{D} - 1\}$.

The processes can calculate each step of Doublestep together. Since every row of every shift-matrix is only dependent of the shift-matrices of the previous step, we can just distribute all the rows of all the shift-matrices that need to be calculated in each step evenly among all processes. This means that early on a process will calculate one or multiple shift-matrices alone. In later steps, the rows of a single shift-matrix will be calculated by different processes. Figure 9 shows how the shift-matrices calculation of row-Doublestep would be divided among four processes. As we see in the last step multiple processes calculate one shift-matrix together.

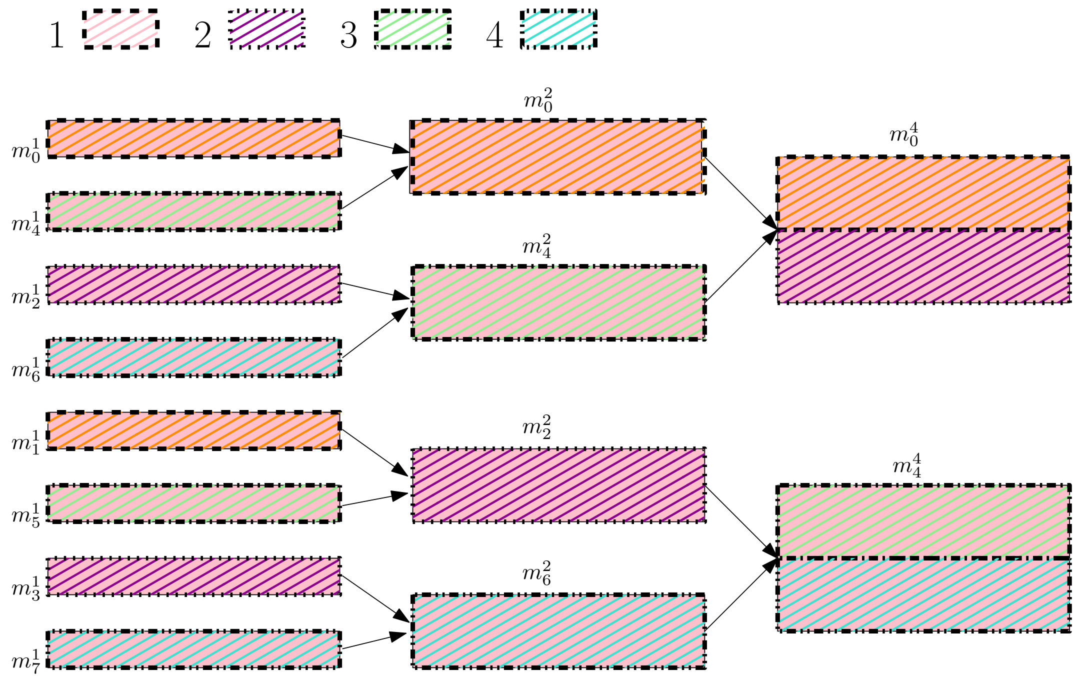

Figure 9: Calculation of row-dictionary and row-Doublestep performed by 4 processes. The color in each step shows, which section gets performed by which process. $N = 8$, $\omega = 2$, $\lambda = 4$.

In the end, the shift-matrices $m^D_{i \cdot D}$ need to be added to the image-matrix-slice. The rows of the slice can be distributed evenly to the processes, so each updates the same number of rows. When adding the values of $m^D_{i \cdot D}$ to the slice, each shift-matrix needs to be shifted by $rank \cdot (i \cdot D)$, where $rank$ is the rank of the machine that is hosting the sub-matrix-slice.

Figure 9 shows how $\lambda = 4$ processes calculate row-Doublestep together on one of two machines $\omega = 2$.

Before they can start with Doublestep, they need to construct the row-shift-dictionary together. Every process has a list of all visibilities. Process one calculates all shift-vector for visibilities with shift-index zero and build their combined shift-vector. It then stores it into shared machine-memory, that is accessible by all processes. It then does the same for shift-index one. The second process calculates the shift-dictionary entries for shift-indices two and three. Process three does shift-indices four and five, and process four calculates for shift-index six and seven. These corresponds to the colored $1 \times N$-sections in Figure 9.

Once all processes have calculated their part of the dictionary and stored it into shared machine-memory, they can calculate the first step of row-Doublestep. Since four shift-matrices need to be calculated and the machine is using four processes, every process calculates exactly one shift-matrix. Process one calculates m_0^2. To do so, it needs to read two entries from the shift-dictionary, m_0^1 and m_4^1. Since both have been written to shared machine-memory, process one can access them both. Process one adds m_0^1 and m_4^1 together to form the first row of m_0^2. It then shifts m_0^1 and m_4^1 by their shift-index before adding them together again to get the second row of m_0^2. It then writes m_0^2 to shared machine-memory. Process two does the same for m_4^2. Process two was not involved in calculating the shift-dictionary entries it needs, m_2^1 and m_6^1. This does not matter, since the whole shift-dictionary is stored in shared machine-memory, so process two can read them. Process three calculates and stores m_2^2 and process four m_6^2.

In the second step, there are more processes then shift-matrices. So multiple processes have to calculate one matrix. Process one and two calculate shift-matrix m_0^4 together. Process one will calculate the first two rows and process two the second two. Both will need access to m_0^2 and m_4^2. Since both these shift-matrices are stored in shared machine-memory, both processes can access them. And since they only need to read the values, they can access them simultaneously. Process one adds together m_0^4 and m_4^4 to geth the first two rows of m_0^4. Process two shifts m_0^4 and m_4^4 by their shift-index before adding them together to get the third and fourth row of m_0^4. They both write there results to the same shift-matrix m_0^4 in shared machine-memory. Since they write to different sections, there is no danger in them accessing the shift-matrix simultaneously, as long as an architecture has been chosen, that allows the change of one value without affecting the other ones. Processes three and four calculate the shift-matrix m_4^4 in the same fashion, by accessing m_2^2 and m_6^2 from shared machine-memory.

The amount of machines used in this example is $\omega = 2$. Row-Doublestep only needs to be performed, until we get shift-matrices of size $\frac{N}{\omega} \times N = 4 \times 8$. Since this is reached after step two of row-Doublestep, there is no third step. Both shift-matrices m_0^4 and m_4^4 can now be used to update the image-matrix.

2.5.2 MPIFT with dictionaries and column-Doublestep

When we consider column-shift, we see that not every row of each shift-matrix will appear in the final image-slice. Take for example the column-shift-matrix m_0^4. Since it has shift-index 0, only the rows that correspond to the image-slice being calculated appear in the final image-slice. Figure 10 shows which lines would be needed to calculate the sub-matrix of a 8×8 image-matrix by machine 2 of $\omega = 4$.

Before we calculate Perpendicular Doublestep, we need to find out, which rows of which shift-matrices are even needed. An intuitive way of doing this, is to look what lines are needed from m_0^N, and calculate recursively, what lines are needed on shift-matrices of earlier steps to compute this matrix. This can be seen in Algorithm 8. Note that in reality, an iterative approach tends to be faster, than a recursive one.

Algorithm 8 NeededLinesColumnDoublestepRec($N, D, lines$)

```
1: if D = 1 then
2:     return {(D,lines)}
3: neededLns = {}
4: for $(l, si)$ in $lines$ do
5:     neededLns ← $neededLns \cup \{(l, \frac{si}{2}), ((l + \frac{si}{2})\%N, \frac{si}{2})\}$
6:     neededLns ← $neededLns \cup \{(l, \frac{si+N}{2}\%N), ((l + \frac{si+N}{2})\%N, \frac{si+N}{2}\%N)\}$
7: return $\{(D, lines)\} \cup NeededLinesPerpendicularRec(N, \frac{D}{2}, neededLns)$
```

The recursive functions calculated the lines needed from every shift-matrix. Since the shift-matrices of size $N \times 1$ are the entries of the shift-matrix, we can use this to only calculate the parts of each shift-vector that are actually needed, saving some complex operations.

Then the processes can go on and calculate the steps of Doublestep. Since different matrices will have different amounts of required lines, we cannot just distribute the shift-matrices in each step to different processes. Instead, we evenly distribute the set of all required lines among the processes. Each line of each shift-matrix is only dependent on the shift-matrices in the previous step, so there are no problems with simultaneous write access. Algorithm 9 implements column-Doublestep computation. Figure 11 shows an example of how the lines to calculate are divided among processes.

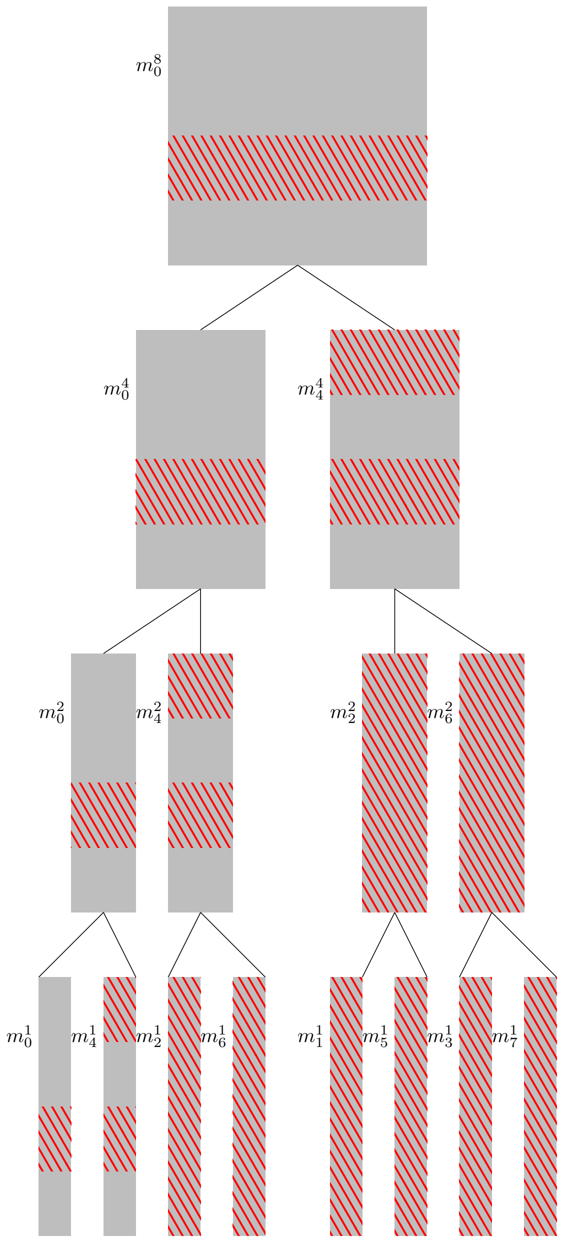

Figure 10: Required Lines for column-Doublestep. $N = 8$, $\omega = 4$, $rank = 2$. Red marks the lines that need to be calculated for each shift-matrix.

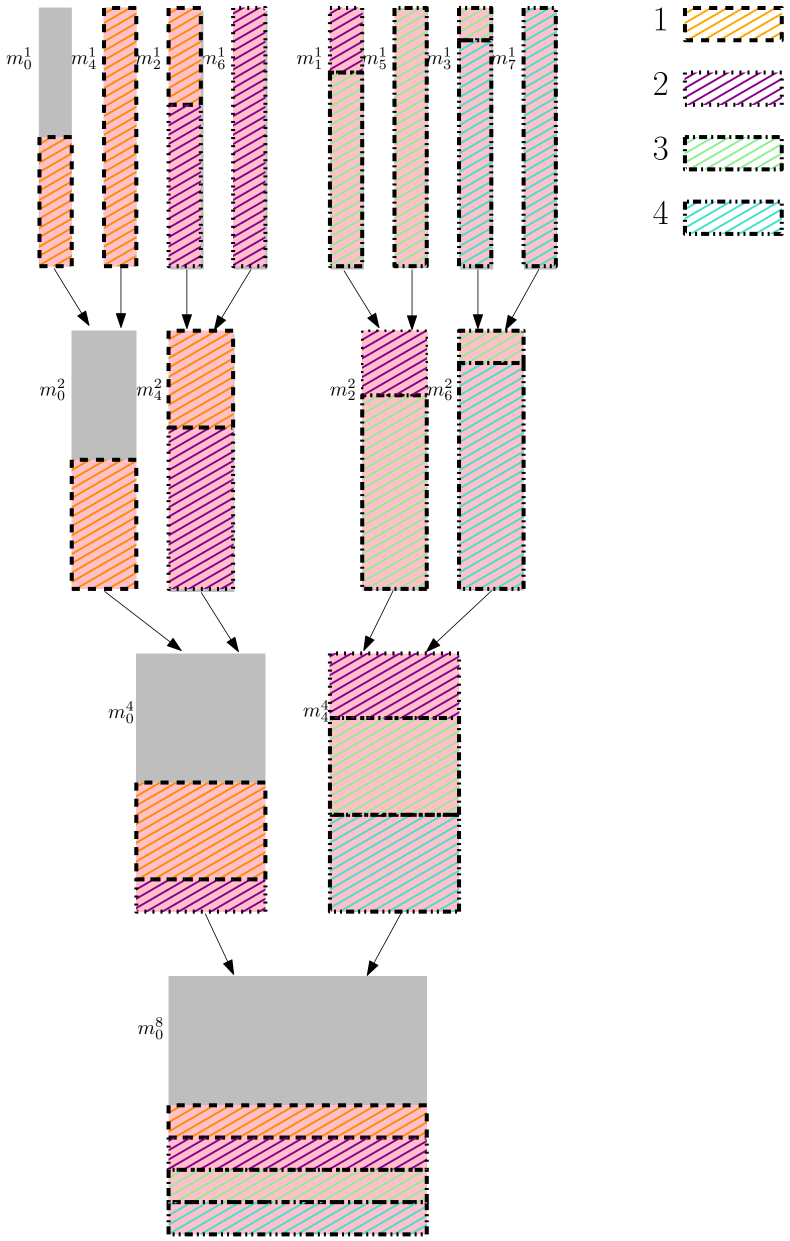

Figure 11: Column Doublestep performed by 4 processes. The color in each step shows, which section gets performed by which process. $N = 8$, $\omega = 2$, $rank = 2$, $\lambda = 4$.

Algorithm 9 StepOfColumnDoublestep($N, step, lines, \lambda, key$)

```
1: $\delta = \frac{lines.size}{\lambda}$
2: $D = 2^{step}$
3: for $i \leftarrow 0$ to $\delta$ do
4:     $(l, si) = lines[\delta \cdot key + i]$
5:     if $l < \frac{D}{2}$ then
6:         $m^D_{si}[l] \leftarrow m^{\frac{D}{2}}_{\frac{si}{2}}[l] + m^{\frac{D}{2}}_{\frac{si+N}{2}\%N}[l]$
7:     else
8:         $m^D_{si}[l] \leftarrow m^{\frac{D}{2}}_{\frac{si}{2}}[(l + \frac{si}{2})\%N] + m^{\frac{D}{2}}_{\frac{si+N}{2}\%N}[(l + \frac{si+N}{2})\%]$
```

To update the image-matrix-slice, the processes need to add the final shift-matrix m^N_0. We can again distribute the rows to the different processes evenly, as we did with parallel Doublestep. The only difference is that there is just one shift-matrix to add. How to update the image matrix for Perpendicular Doublestep is shown in Algorithm 10. The $key \in \{0, 1, \dots, \omega - 1\}$ refers to the machine.

Algorithm 10 UpdateImagePerpendicular(N, m^N_0, λ, key)

```
1: $\Delta = \frac{N}{\omega}$
2: $\delta = \frac{\Delta}{\lambda}$
3: for $l \leftarrow \delta \cdot key$ to $\delta \cdot (key + 1)$ do
4:     $I_t \leftarrow I_t[\Delta + l] + m^N_0[\Delta + l]$
```

2.5.3 Complexity

Since Multiprocessing Doublestep consists of a row- and a column part, we will discuss the complexity of these two parts separately.

The relevant parts of the complexity of MPIFT with dictionaries and row-Doublestep have been discussed before. Calculating a full row-shift dictionary takes

$$\frac{N \cdot \beta}{\lambda}$$

complex multiplication per kernel and

$$\frac{(\beta - N) \cdot N}{\lambda}$$

complex multiplication per kernel. Doublestep takes N^2 complex additions per step, as seen in Section 2.4.3. For row-Doublestep, we only perform $\log \frac{N}{\omega}$ steps, so it requires

$$\log \frac{N}{\omega} \cdot N^2$$

complex additions. Calculating the image-matrix-slice means adding ω $\frac{N}{\omega} \times N$ matrices to the image matrix. This takes

$$\omega \cdot \frac{N}{\omega} \cdot N = N^2$$

complex additions.

For MPIFT with dictionaries and column-Doublestep the calculations are more complicated. Equation (2) shows how many values are needed in the column-shift-dictionary. Unless $\omega = 1$ this is smaller than N^2, but for small ω it's asymptotic complexity still is N^2. So we need

$$\mathcal{O}(\beta \cdot N^2)$$

complex additions and multiplications.

$$N^2 - N \cdot \omega - \frac{N}{\omega} + \frac{2(\omega^2 - 1)}{3 \cdot \omega} \tag{2}$$

Equation (3) shows how many complex additions are needed for column-Doublestpe. Like row-Doublestep, this has an asymptotic complexity of

$$\mathcal{O}(N^2 \cdot \log N)$$

complex addtions.

$$\begin{split} 2 \cdot \frac{N}{\omega} \cdot (N-1) - \frac{4 \cdot (\frac{N}{\omega} - 1)((\frac{N}{\omega})^2 - N^2)}{3 \cdot \frac{N}{\omega}} + \\ \frac{2}{3} \cdot (2 \cdot (\frac{N}{\omega})^2 - 3 \cdot \frac{N^2}{\omega} + N^2) - 2 \cdot N^2 \cdot \frac{N - \omega}{N} + N^2 \cdot (n - \log(\omega)) \end{split} \tag{3}$$

MPIFT with dictionaries and Doublestep gets performed by λ processes working together. After distributing the work, every process performs

$$\mathcal{O}(\frac{\log{(N)} \cdot N^2}{\lambda})$$

complex additions.

2.6 Comparing

In this section, we will compare, how efficient the three algorithms naive MPIFT, MPIFT with dictionaries and MPIFT with dictionaries and Doublestep are. Table 6 shows how many complex multiplications and complex addition these different approaches take. Since the differences are big enough, we will use the asymptotic complexity to compare. The amount of machines ω is not expected to be big, so we will assume $\omega = 1$ for these comparisons.

Both approaches that use shift-dictionaries do a lot better when it comes to complex multiplications. This has nothing to do with the shift-dictionaries themselves, and only with the fact that they use shared memory, and can therefore distribute all the shift-vector calculations over multiple machines.

The only case where naive MPIFT has less complex additions is when $\beta \leq N$. In all other cases, MPIFT with dictionaries and Doublestep has the fastest asymptotic complexity, when it comes to complex additions. If $\beta < N^2$, the asymptotic complexity for MPIFT with dictionary is the same as naive MPIFT, regardless of the distribution of processes.

MPIFT Algorithm	complex multiplications	complex additions
naive	$\mathcal{O}(\beta \cdot N)$	$\mathcal{O}(\frac{\beta \cdot N^2}{\phi})$
with dictionary	$\mathcal{O}(\frac{\beta \cdot N}{\lambda})$	$\mathcal{O}(\frac{\beta \cdot N}{\lambda} + \frac{N^3}{\phi})$
with dictionary textbfand Doublestep	$\mathcal{O}(\frac{\beta \cdot N}{\lambda})$	$\mathcal{O}(\frac{\beta \cdot N}{\lambda} + \frac{N^2 \cdot \log N}{\lambda})$

Table 6: Comparing theoretical asymptotic complexity of different MPIFT algorithms. All values are per process

Table 7 shows the ratios for the three approaches regarding the same measurements. When using MPIFT with dictionary we can reduce the amount of complex additions needed by $\frac{\beta}{N}$. The improvement gets better when we additionally use Doublestep, which will reduce it further by $\frac{N}{\log N}$ to a total of $\frac{\beta}{\log N}$.

By using shift-dictionaries The amount of complex multiplications only gets reduced by λ. Additional using Doublestep doesn't impact it significantly. This means that the efficiency increase we get is dependent on the physical infrastructure available and does not scale with N. In cases where λ is limited, we foresee a new computational bottleneck.

MPIFT Algorithm 1	MPIFT Algorithm 2	complex multiplication ratio	complex additions ratio
naive	with Dictionary	$\frac{\lambda}{1}$	$\frac{\beta}{N}$
naive	with dictionary and Doublestep	$\frac{\lambda}{1}$	$\frac{\beta}{\log N}$
with Dictionary	with dictionary textbfand Doublestep	1	$\frac{N}{\log N}$

Table 7: Comparing ratio of significant measures per process between MPIFT-implementations. We assume $\beta << N^2$.

2.7 Experiment

2.7.1 Setup

To test the three Algorithms, we implemented them in *C++*, using MPI (*Message Passing Interface*) to communicate between nodes and to enable shared memory. The paper from Saad et al. used Apache Flink. We did not use this, since Apache Flink doesn't offer possibilities so share local memory. We used the Open MPI library, that implements MPI for *C++*.

We used two servers, provided by the institute for computer science of the University of Zurich. Each of them has 48 cores, and 188 GB of memory using Intel Xeon Silver 4214 Processors. In order to record the measured time, each implementation wrote to the file-system at fixed point in its program.

Naive MPIFT only recorded the total time elapsed to handle one batch. We did this, because the constant writing that would result from writing after each single visibility would slow down the system and the overall elapsed time would be less precise. Also since naive MPIFT is the only one that handles visibilities one by one, we would have nothing to compare the time for a single visibility to.

For MPIFT with dictionary we measured the overall elapsed time, the time it took to calculate each dictionary and the time to run SPIFT on all values of all dictionaries. This way, we can compare the total time elapsed for the naive approach, and the time to calculate the dictionaries with MPIFT with dictionaries and Doublestep.

For MPIFT with dictionaries and Doublestep we measured the time to calculate each dictionary and the time it took to calculate row- and column-Doublestep. This way, we can see if the bottleneck is more with column-Doublestep, as we would expect. We can also compare the row- and shift-dictionary creation with the previous approach. We are calculating the row-dictionary the same way, so the times should be more or less the same. The column-dictionary is calculated differently, and it will be interesting to see if the times are significantly different.

We used a constant $N = 2^{13} = 8192$ for all experiments. We started by using two machines $\omega = 2$ with $\lambda = 16$ processes each. After that we varied the amount of processors per machine $\lambda = 8, \lambda = 32$. Initially we used a batch-size of $\beta = 42642$. Later we also used $\beta = 170568$ and $\beta = 340992$, to confirm that only the calculation of the shift-dictionary and the whole of the naive approach would suffer under this.

2.7.2 Result

Figure 12 shows how fast the three implementations were per batch. Naive MPIFT with a speed of 2'500 seconds was clearly the slowest. Including shift-dictionaries made it almost three times as fast with an average of 890 seconds per batch. MPIFT with dictionaries and Doublestep took an average of 10.5 seconds and was by far the fastest, 240 times faster than the naive approach and 80 times faster than only with shift-dictionaries.

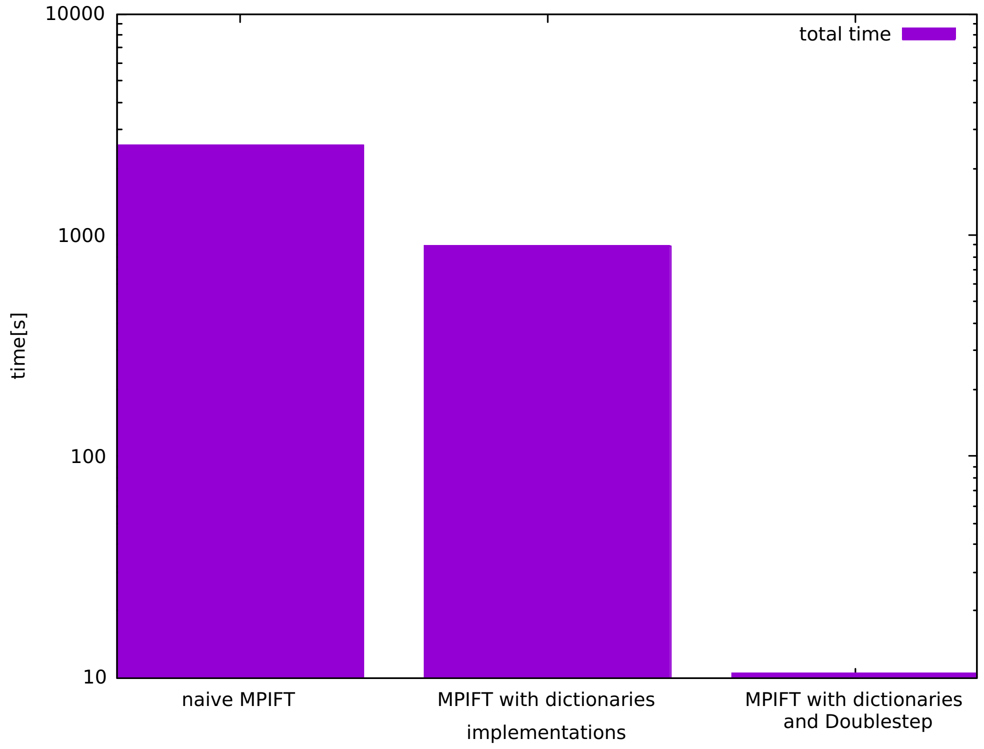

Figure 12: Total time required by naive MPIFT, MPIFT with dictionaries and MPIFT with dictionaries and Doublestep with $N = 8192$, $\phi = 64$, $\omega = 2$, $\lambda = 32$, $\beta = 42642$

Figure 13 compares the times that MPIFT with dictionaries and MPIFT with dictionaries and Doublestep need to update the image-matrix after having calculated the shift-dictionaries. With 1.7 seconds, Doublestep is 90 times faster than SPIFT to add the row-shift-dictionary, and with 4.9 seconds 55 times faster with the column-shift dictionary. Both row- and column-Doublestep have the same asymptotic complexity, but column-Doublestep requires more complex additions. This explains why the ratio for column-shift is smaller than for row-shift.

It is also of note that for MPIFT with dictionaries and Doublestep, row-shift is significantly faster than column-shift. This is on one hand because column-Doublestep requires more complex additions, and on the other hand because column-Doublestep calculates more steps of Doublestep and therefore has a larger computational overhead. For MPIFT with dictionary, column-shift is also slower. This is because in our implementation, we iterated over the rows and then over the columns, leading to a loss of time because of poor memory access.

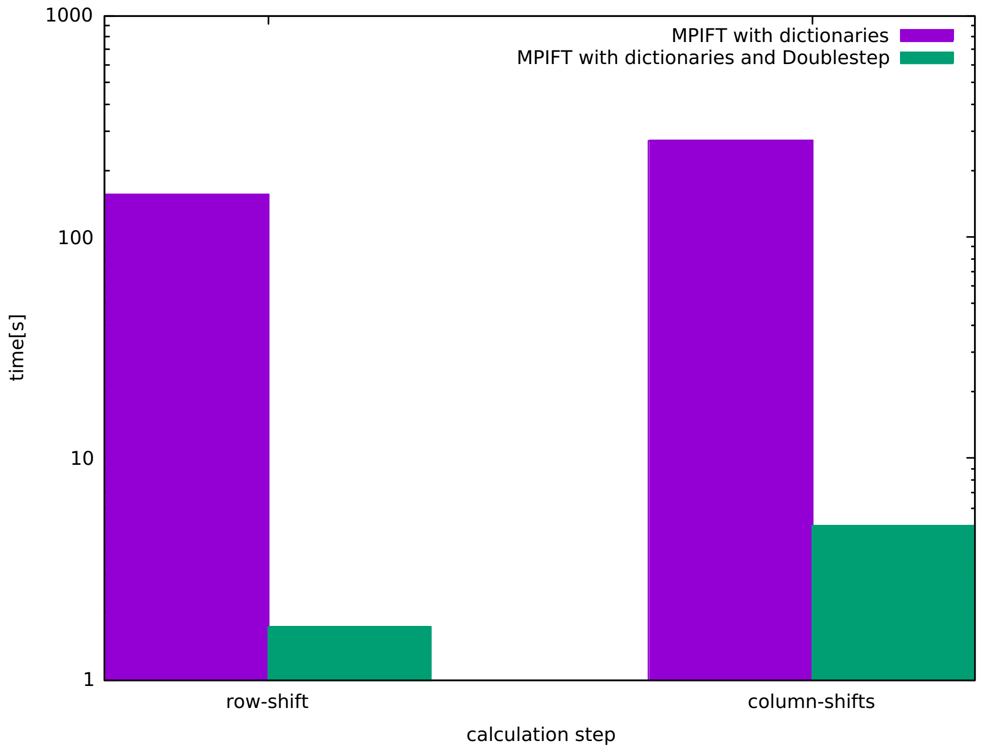

Figure 13: Time required to update the image matrix after having calculated the shift-dictionaries by MPIFT with dictionaries and Doublestep, $N = 8192$, $\phi = 64$, $\omega = 2$, $\lambda = 32$, $\beta = 42642$

Figure 14 shows a breakdown of the times for MPIFT with dictionaries and Doublestep for different degrees of parallelism $\lambda \in \{8, 16, 32\}$. When the amount of kernel increases, the computational time goes down, but it doesn't go down as much as hoped. The amount of complex additions is halved when the amount of processes is doubled. We expected that the time for row- and column-Doublestep would be close to halved, but we don't see that. We assume this is because we implemented Doublestep without any optimizations for multi-threading. So threads are created and destroyed more often than needed, leading to an increased overhead that gets very noticeable when times get as low as they are here.

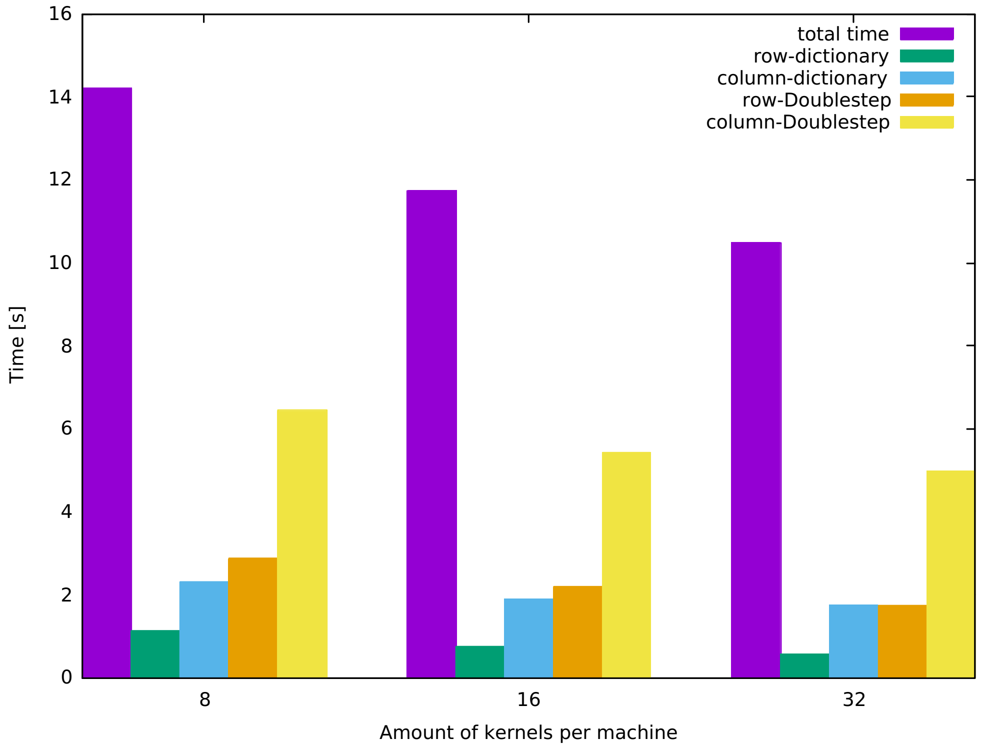

Figure 14: Breakdown of MPIFT with dictionaries and Doublestep with varying amounts of local processes λ, $N = 8192, \beta = 42642$

Figure 15 shows how Doublestep deals with different batch-sizes $\beta \in \{43643, 170568, 340992\}$. As expected, the time for row- and column-Doublestep do not change. This is because they work only on shift-dictionaries and not directly on the batch.

We also see for the parallel dictionary that the jump between $\beta = 42642$ to $\beta = 170568$ is bigger than from $\beta = 170568$ to $\beta = 340992$. This is because for smaller batch-sizes, there are still shift-indices that only have one visibility, therefore saving on complex additions. When the batch sizes increases, this effect becomes increasingly unlikely.

We also see that calculating the column-shift-dictionary takes significantly longer than the row-dictionary. When calculating the column-shift-dictionary, we only calculated the rows for each shift-vector, that were needed to perform column-Doublestep. This decreases the amount of complex operations needed to compute the shift-dictionary, but again at the cost of computational overhead. This is why initially the difference between row- and column-dictionary computation is big, but decreases when the batch-size increases, since the saved operations become more important than the computational overhead. But the column-shift dictionary only has half as many visibilities and is even for bigger batch-sizes only as efficient as calculating the row-shift dictionary. This is an indication that the optimization here is not worth the effort and it would be faster to simply compute the full shift-dictionary as normal.

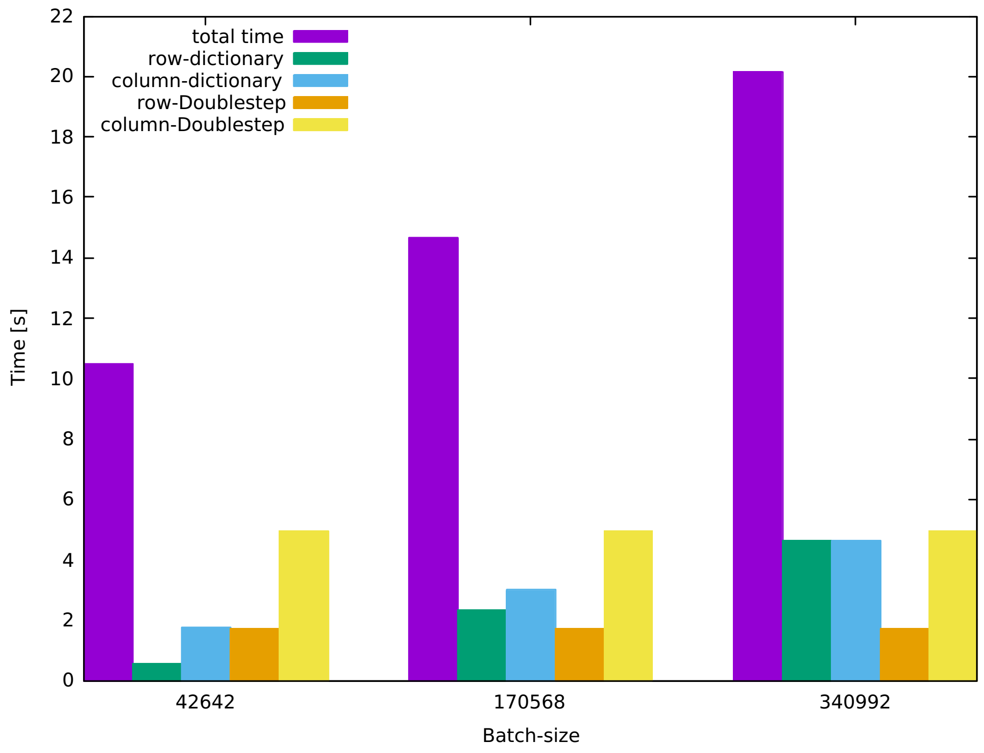

Figure 15: Breakdown of MPIFT with dictionaries and Doublestep with varying batch size β, $N = 8192, \phi = 64$.

3 Conclusion

MPIFT with dictionaries and Doublestep is significantly faster than the naive approach or only using dictionaries, both in theory and in an experimental setting. We couldn't reach the full theoretical improvement of $\mathcal{O}(\frac{N}{\log N})$ in an experimental setting. Presumably this is because of the computational overhead our implementation of Doublestep introduced. MPIFT with dictionaries and Doublestep scales better than both other MPIFT variants.

With Doublestep, updating the image-matrix becomes more efficient and the amount of complex additions is reduced. Introducing a shift-dictionary can further reduce the amount of complex additions. But neither of these methods can reduce the amount of complex multiplications significantly. The creation of the shift-dictionary and the complex multiplications needed for it become the new computational bottleneck.

Since this report build on the work of Saad et al. [3], its main focus are applications in radio-astronomy. But as with SPIFT, Doublestep could proof useful in other fields that use Discrete Fourier Transformation as well.

Further research could go into increasing the efficiency of Doublestep by reducing the computational overhead and optimizing threading. Since Doublestep mostly requires complex additions and shared memory it could be a worthwhile endeavor to analyze if it runs efficiently on a graphic card.

References

[1] A. Biem, B. Elmegreen, O. Verscheure, D. Turaga, H. Andrade, and T. Cornwell. A streaming approach to radio astronomy imaging. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1654–1657. IEEE, 2010.

[2] A. Richard Thompson, J. M. Moran, and G. W. Swenson Jr. *Interferometry and synthesis in radio astronomy*. Springer Nature, 2017.

[3] M. Saad, M. H. Böhlen, A. Bernstein, and D. Dell' Aglio. Single point incremental fourier transform on 2d data streams. In *37th IEEE International Conference on Data Engineering*, Chania, Greece, 2021. IEEE.

[4] R. V. van Nieuwpoort and J. W. Romein. Correlating radio astronomy signals with many-core hardware. *International journal of parallel programming*, 39(1):88–114, 2011.

4 Appendix

The complete source code used for testing is accessible on *Github* at https://github.com/degenwitz/SPIFT_Thesis.