BlackNUFFT: Modular customizable black box hybrid parallelization of type 3 NUFFT in 3D

BlackNUFFT: modular customizable black box hybrid parallelization of type 3 NUFFT in 3D

Nicola Giuliani[a]

[a]*SISSA — International School for Advanced Studies, Via Bonomea 265, 34136 Trieste, Italy*

Abstract

Many applications benefit from an efficient Discrete Fourier Transform (DFT) between arbitrarily spaced points. The Non Uniform Fast Fourier Transform reduces the computational cost of such operation from $O(N^2)$ to $O(N \log N)$ exploiting gridding algorithms and a standard Fast Fourier Transform on an equi-spaced grid. The parallelization of the NUFFT of type 3 (between arbitrary points in space and frequency) still poses some challenges: we present a novel and flexible hybrid parallelization in a MPI-multithreaded environment exploiting existing HPC libraries on modern architectures. To ensure the reliability of the developed library, we exploit continuous integration strategies using Travis CI. We present performance analyses to prove the effectiveness of our implementation, possible extensions to the existing library, and an application of NUFFT type 3 to MRI image processing.

Keywords: NUFFT type 3, TBB, MPI, FFT, modularity, extensibility, C++, MRI Image processing.

PROGRAM SUMMARY

Program Title: BlackNUFFT
Licensing provisions: LGPL
Programming language: C++

Program file DOI: `https://doi.org/10.5281/zenodo.1404191`

External routines/libraries: `deal.II` *, FFTW, PFFT*

Nature of problem: Provide a modular and extensible implementation of a parallel Non Uniform Fast Fourier Transform of type 3.

Email address: `ngiuliani@sissa.it` (Nicola Giuliani)

Solution method: Use of hybrid shared distributed memory paradigm to achieve high level of efficiency. We exploit existing HPC library following best practices in scientific computing (as continuous integration via TravisCI) to reach higher complexities and guarantee the accuracy of the solution proposed.

1. Introduction

The standard Fast Fourier algorithm relies on a distribution of the points on a regular equispaced grid. However, many applications benefit from an accurate and reliable Fourier transform between N arbitrarily spaced points. Many image reconstruction techniques, such as Magnetic Resonance Imaging, are based on a non Cartesian grid in the frequency domain [1, 2, 3]. In [4, 5] the authors propose accelerated convolution techniques based on the discrete Fourier Transform between arbitrary points in both space and frequency domains. The computational cost of the standard discrete transform increases quadratically, quickly becoming unbearable even on modern computational platforms.

A solution to this problem is the Non Uniform Fast Fourier Transform (NUFFT) developed by Dutt and Rokhlin [6]. The authors provide a deep theoretical analysis to approximate the Fourier transform using the classical Fast Fourier Transform algorithm. Another description of such algorithm has been addressed by Leslie Greengard and June-Yub Lee in [7]. In the literature there are 3 different types of NUFFT: type 1 operating between arbitrary points in space and a regular grid in frequency, type 2 correlating a regular grid in space and arbitrary points in frequency, and type 3 dealing with arbitrary points in both spaces.

The key idea of NUFFT is to transfer the non equi-spaced data on a uniform grid to be used with standard FFT algorithms. In [7] the authors use fast convolutions with a Gaussian function to create a uniform grid where standard Fast Fourier Transform algorithms can be used. This is a key step to retrieve the solution in an affordable time, since Gaussian convolutions can be efficiently computed by means of Fast Gaussian Gridding, reducing the overall computational time. Another possible solution is the usage of Kaiser-Bessel window or the min-max interpolator. In recent years the latter has been efficiently implemented in the library PNFFT [8]. In [9] the authors propose a parallelization of type 1 NUFFT based on the P3DFFT [10]. Type 3 NUFFT, developed in [7], has been successfully applied in fast numerical convolutions [4], and has applications in many fields of engineering. Several parallelizations have been performed for such an algorithm: in [11, 12] highly scalable optimisation of the library on modern multicore systems are proposed, in [13] the authors use the hardware to accelerate the NUFFT, while in [14] the multicore architecture of GPUs is used to tune and optimize NUFFT.

While the previous implementations have proved to be effective, they are extremely hardware specific and optimized. Moreover, their modification is far from trivial from the user's perspective. We propose an OpenSOURCE flexible parallelization strategy of the type 3 NUFFT algorithm based on existing High Performance Computing libraries: we call such a library BlackNUFFT [15]. We use Intel Threading Building Blocks (TBB) for shared memory parallelism [16, 17], together with the standard Message Passing Interface (MPI) for distributed parallelism. High modularity is a key aspect of our implementation, allowing the user to plug-in new algorithms for each aspect of NUFFT. We use a black box approach for FFT on the fine grid, making it possible to interchange back-end libraries for the FFT algorithm. We provide a first implementation that uses by default FFTW [18] as back-end FFT. This choice is mainly due to the extensive documentation and high reliability of such a library. We also propose a second implementation using PFFT [19] as back-end FFT library. In this way we improve the parallel efficiency of our library and we show the advantages of using a black-box modular approach that allows the interchanging of its building blocks. We compare the presented BlackNUFFT with the library developed by Lee and Greengard which is freely available under GPL license [20]. We use existing implementations of distributed vectors provided by existing OpenSOURCE libraries. In particular we refer to the implementation of the `deal.II` library [21, 22, 23], which provides both high performance on modern architectures and the possibility of easily switching between 32 and 64 bits indexing. This capability is a keystone to reach higher computational complexity. We present a performance analysis considering multithread and multiprocessor environments for the current implementation of the library. BlackNUFFT is available under LGPL license on github [15].

The work is organized as follows: in Section 2 we collect some considerations on theoretical aspects of the NUFFT [7, 6] and we describe the actual structure of the algorithm. In Section 3 we exploit shared memory parallelism to achieve a first multicore parallel analysis. In Section 4 we combine shared and distributed memory paradigms to reach higher levels of scalability. Section 5 describes the coupling with the parallel pruned library PFFT [19]. In Section 6 we briefly describe the application of BlackNUFFT to the reconstruction of MRI images. Section 7 describes possible procedures to expand BlackNUFFT capabilities.

2. NUFFT type 3

2.1. Mathematical background

We follow [6, 7] to introduce the main aspects of NUFFT of type 3. Given a set of N complex values in the three dimensional space $f(\mathbf{x})$ we define its discrete

Fourier transform (DFT) to a set of N points in the frequency space as

$$F(\mathbf{k}) = \frac{1}{N}\sum_{i=0}^{N-1} f(\mathbf{x})e^{-i\mathbf{x}\cdot\mathbf{k}}, \tag{1}$$

with $\mathbf{k} = (k_1, k_2, k_3)$ and $\mathbf{x} = (x_1, x_2, x_3)$. We also have that $-N/2 \leq k_1, k_2, k_3 < N/2 - 1$. We can define also the inverse DFT as

$$f(\mathbf{x}) = \sum_{i=-N/2}^{N/2-1} F(\mathbf{k})e^{i\mathbf{x}\cdot\mathbf{k}}. \tag{2}$$

DFTs defined in (1) and (2) are characterized by a computational cost $O(N^2)$. NUFFT algorithms are based on an interpolation scheme of the arbitrary points in real and frequency space to a regular grid on which we can apply the standard FFT algorithm. In the present work we use the so-called Gaussian Gridding algorithm to perform this interpolation, see [7]. An alternative gridding techniques is the min-max interpolator, which has proved to be very efficient especially when the dimensionality of the operation increases [8]. In the case of Gaussian Gridding we exploit the following one dimensional estimate to approximate the exponential e^{ikx} function on the regular grid

$$\left| e^{icx} - e^{bx^2} \sum_{l=c-q/2}^{c+q/2} \frac{1}{2\sqrt{b\pi}} e^{-(c-l)^2/4b} e^{ilx} \right| < e^{bx^2} e^{-b\pi^2} (4b + q) = \epsilon, \tag{3}$$

where c represents the nearest point to k on the regular grid, q is usually referred to as spreading and b is a scaling constant. Estimate (3) straighforwardly extends to the three dimensional case. We need to set variables b, q in order to assure that the estimate fulfills the accuracy ϵ for any points in our arbitrary distribution. The fact that we can *a priori* set the accuracy of the computation is a key point of the entire algorithm, and in general of all efficient fast accelerated method [24]. The summation in (3) can be seen as a convolution like $\delta(x - \bar{x}) * g_{\bar{x}}(x)$ where $g_{\bar{x}}(x)$ represents a Gaussian like function centered in $\bar{x}$. Such a convolution can be efficiently performed using the Fast Gaussian Gridding (FGG) depicted in [7]. For the sake of clarity we consider a forward 1D transformation from a space x to s, we let X, S represent the bounding boxes for the two spaces, we denote m_{sp} as the number of points on the regular grid to which we extend the influence of each non uniform point. Given the exponential decay of the Gaussian kernel m_{sp} is easily and safely controlled. We define

$$\Delta_x \leq \frac{\pi}{S}\frac{1}{R}, \tag{4}$$

and

$$\Delta_s \leq \frac{\pi}{X + m_{sp}\Delta_x}\frac{1}{R}, \tag{5}$$

where R represents the oversampling parameter, both R and m_{sp} can be optimized to fulfill the *a priori* accuracy requirement (3). We follow the optimized implementation of NUFFT type 3 developed by Greengard *et al.* [7]. The oversampling ratio R varies from $\sqrt{2.2}$ to 3 depending on the requested accuracy. Then the Gaussian spreading is computed as

$$m_{sp} = -\frac{\log \epsilon}{\pi * (R-1)/(R-0.5)} + 0.5, \tag{6}$$

where ϵ is again the required tolerance. Finally the fine grid size is

$$M_r = \frac{2\pi}{\Delta_s \Delta_x}, \tag{7}$$

this 1D estimate can easily be extended to each 3D coordinate. Once we have obtained a regular grid we apply a FFT to obtain a transformed regular array, and apply again an interpolation to retrieve the result on an arbitrary output grid. We do this with another FGG using again (3) to have an accurate computation.

2.2. Algorithm key-steps

The algorithm consists of 4 key-steps:

1. Computation of the bounding box for the arbitrary grids, and set up of the griddings in order to retrieve the requested accuracy. This operation is not demanding from a computational point of view since it only sets up the spreading constants for the griddings and the dimension of the fine grid array.
2. Transfer of the original data on the uniform grid. Since we have chosen Fast Gaussian Gridding we have:
   - Fast Gaussian Gridding of the original data to the regular fine grid.
   - Scaling to correct the gridding on the fine grid array.
3. FFT on the regular grid: basically a 3d FFT pruning the border values (thus saving computational time) that represent convolution errors, a data shift keeps the low frequencies at the center of the spectrum.
4. Transfer of the transformed data from the uniform grid to the output points. Since we have chosen Fast Gaussian Gridding we have:
   - Fast Gaussian Gridding from the fine grid to the output data.

- Scaling on the output array to correct errors introduced by the second gridding.

The algorithm uses two griddings between the input data and the uniform grid and between the uniform grid and the output data. As preliminary choice we use Fast Gaussian Gridding to perform both. These operations can be highly optimized by means of a Fast Convolution algorithm, making its implementation not trivial even on a single core. We designed the parallelization to work with any parallel implementation of the FFT. We require the chosen library to provide a suitable 3D subdivision of the fine grid array. In particular we use the three dimensional implementation of FFTW. Even the single griddings can be easily interchanged, we just need to guarantee the creation of the fine grid array. In Section 7 we describe the procedure to introduce such new features inside our library. Figure 1 depicts the structure of our library.

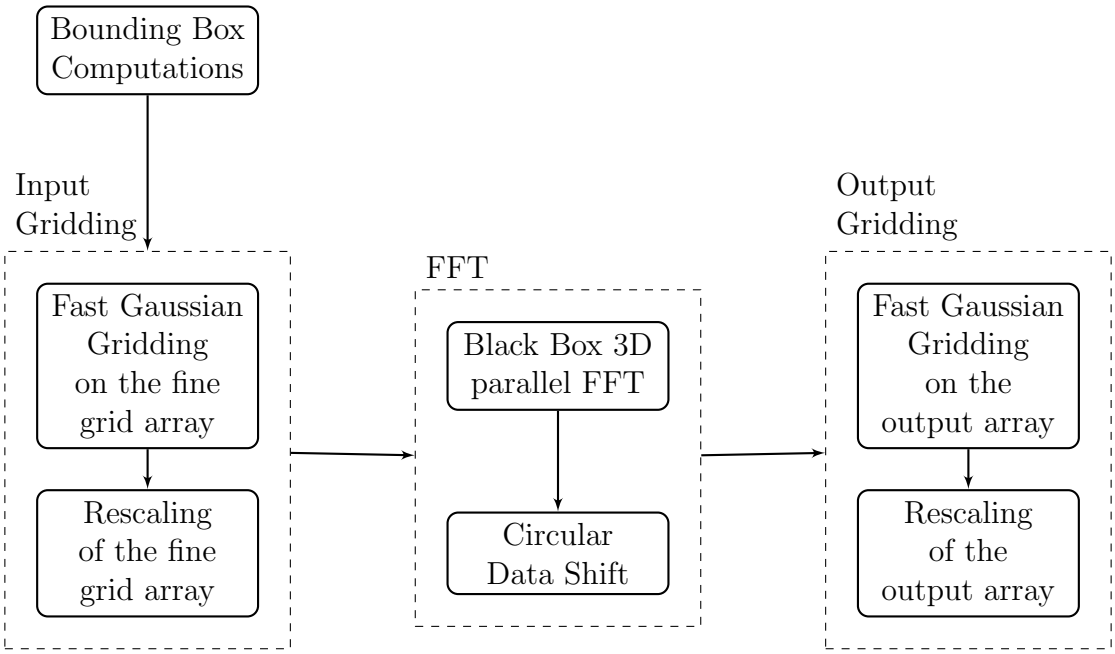

Figure 1: Flow diagram depicting the structure of the library. It can be roughly subdivided in three main parts, and we stress that each of them can be substitute and customized by the user.

Table 1: Profiling of a serial run of BlackNUFFT application.

Function	Time (sec)	%
FGG on Inputs	60.1	45
Scaling on Inputs	4.25	3.2
FFTW 3D	18.7	14
Circular Shifting	3.13	2.4
FGG on Outputs	39.8	30
Scaling on Outputs	2.14	1.6

3. Shared memory parallelism

All modern CPUs support multicore shared memory parallelism. We use Intel Threading Building Block [16] to exploit this possibility and achieve higher efficiency. This tool has been successfully adopted in many high performing library, as ASPECT [25], or the `deal.II` library [26]. Moreover it introduces the use of the TaskScheduler concept that allows for higher level of optimization in our library. We follow the shared memory parallelization strategies introduced in [16, 17] and we apply it to all NUFFT key-steps.

3.1. Implementation

We profiled our serial library on the following test case scenario, a single forward NUFFT from an array of $2^{21} = 2097152$ points to another array of $2^{21} = 2097152$ points. We consider sine cosine distributions both in space and in frequency, as depicted in [20] and we consider an accuracy of $\epsilon = 1 \times 10^{-5}$. It is well known that the spreading constant for the determination of the fine uniform grid are influenced both by the input and the output array, thus for the sake of clarity we consider the following relationship between the maximum frequency K_{max} and the maximum spreading R_{max} in the original space,

$$\frac{R_{max} K_{max}}{\pi} = 100. \tag{8}$$

We get a fine grid array of 373248000 points, requiring approximately 6 GB of memory. The profiling is reported in Table 1. The most demanding functions are the two griddings (64.35 and 41.94 seconds respectively) together with the three dimensional FFT (23.7 seconds).

Gridding on Input. The most demanding part of the algorithm is the Fast Gaussian Gridding. We focused on this particular algorithm for the extensive literature and high accuracy it can guarantee [7]. The study of different efficient griddings, especially the min-max algorithm [8], is undergoing. The first gaussian gridding is in essence a convolution through a Gaussian kernel. We perform it exploiting the Fast Gaussian Gridding (FGG) algorithm developed in [7]. This part of the algorithm presents several race conditions in the writing of the fine grid array, therefore we expect some parallelization issues. A subdivision of the fine grid array reduces the synchronization requirements. The accuracy ϵ we prescribed settles the span of the Gaussian kernel to be used in the FGG: ϵ defines the number of point q that a generic input point can influence through the gridding. We subdivide, in any direction we prefer, the fine array in sections of span $2q$, and we consequently create index sets for the input array that contain the elements that have the nearest fine point in each subdivision. We compute alternate odd-even subdivisions without race conditions between different even-even or odd-odd subsets. We manage the scheduling of the different threads using the standard parallel loop described in [16]. We split the array along its second dimension, leaving the first one to set up the MPI parallelization described in Section 4. A summary of the required steps for the shared memory parallelization follows.

- Subdivision of the fine array along one single dimension. We sketch the subdivision in Figure 2. If we consider a gridding radius q we can split the fine grid array using $2q$ to identify elements that will not have any racing conditions. In Figure 2 we see that all the region with the same color can be written at the same time.
- Subdivision of the input array between different subdivisions.
- Gridding of all the even, colored in yellow in Figure 2, subdivisions using TBB in each subset.
- Gridding of all the odd, shown in blue in Figure 2, subdivisions using TBB in each subset.

We don't expect an optimal scalability for two main issues: the setting of the parallelization introduces a slight overhead, and, since we consider an oversampled FFT, there are empty sets at the boundaries. This situation unbalances the workload between different threads, introducing suboptimality. In Section 5 we show that the usage of a pruned FFT mitigates this kind of work unbalance.

To complete the gridding of the inputs we need a scaling of the fine grid array. With this operation we satisfy equation (3), that is, we need a scaling to correct

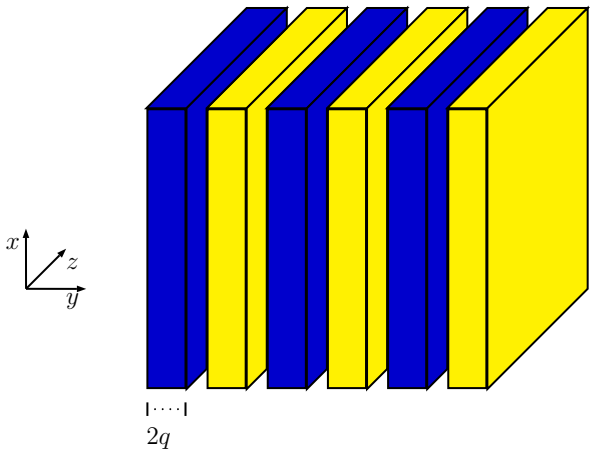

Figure 2: Subdivision for the shared memory parallelization. Even subsets in yellow and odd subsets in blue.

the gridding on the input array. This is a pointwise multiplication by a factor e^{bx^2}, not presenting any parallelization issue.

Compute 3D FFT and Data Shift. We perform a FFT on the fine grid array exploiting the back-end HPC library of our choice. We enable the multithreading in the external FFT library, so we simply need to set the number of threads to be used. It is common to shift the data in order to have the lowest values at the center of the grid. We apply a standard circular shift, which is a local embarrassingly parallel operation. We just need to apply the transformation depicted in (9) to the transformed three dimensional matrix $F(i, j, k)$, namely

$$F(i,j,k)_{shift} = -1^{i+j+k} F(i,j,k) \; i,j,k = 0 \ldots N_1 - 1, N_2 - 1, N_3 - 1, \qquad (9)$$

where N_1, N_2, N_3 represent the dimension of the fine grid. We stress that shifting the input fine matrix would be unbearable especially in distributed memory as in Section 4, where we would need to communicate GigaBytes of data.

Output Gridding. This operation is the adjoint of the first input gridding. We need to recover the data on the output array starting from the fine grid representation, and we use TBB to exploit the multicore parallelism. Since we only have concurrency in reading we expect almost an optimal behavior for this step and we don't need any particular subdivision strategy for the parallelization.

We scale the data to correct the Gaussian gridding we have performed on the output array. This is still a pointwise operation so we use a simple shared memory implementation with no race condition handling.

3.2. Strong scaling analysis up to 16 threads

For the analysis of the strong scalability of our pure TBB parallelization we run the computations on a single node, with a Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz processor with 16 hyperthreaded cores. We consider the same NUFFT setting we aforementioned in Section 3.1, and present the scalability results in the left plot of Figure 3. On the right we plot the relative importance, in terms of computational time, of each function composing the NUFFT algorithm.

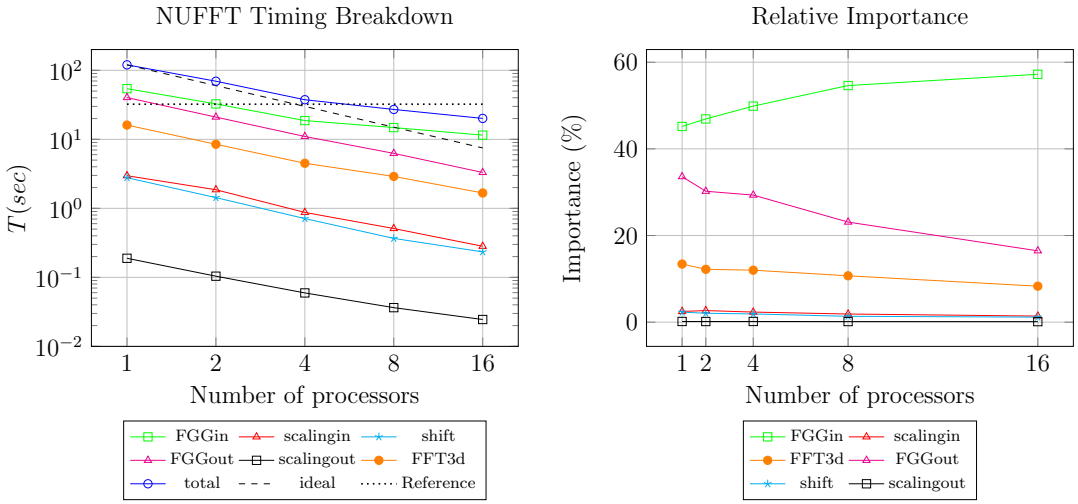

Figure 3: Performance analysis of the pastoralization using shared memory paradigms. Single NUFFT operation on 2097152 arbitrary points, fine grid array of 373248000 points (6 GB of RAM memory). Left figure: timings of all the functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares), total time (blue with circles), linear scalability (dashed), reference algorithm (dotted). Right figure: relative importance plot of the different functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares). We let the number of threads vary from 1 to 16.

We see that the fast gridding on the output array, the two scaling operations, and the shift after the FFT show a nearly optimal behavior. This is expected since these are local operations and almost no write racing condition occurs. However, we see that the parallelization of the fast gridding from the input array is suboptimal. This is due to the setting up time of the parallelization strategy which we have sketched in Figure 2. To reduce the number of racing conditions we need to carefully subdivide the input array in the different subsets, and this

operation introduces a slight overhead. Moreover, since we are considering an oversamplig, there will be some empty sets at the boundaries of the fine grid array. These empty subdivisions, together with the setting up time, unbalance the overall workload between different threads inducing the suboptimality. We see that the three dimensional FFT, which has a multithreaded parallelism enabled, behaves almost optimally. Since multithreaded performance relies on appropriate choices for the number of tasks, the granularity, and probability of task stealing, we are currently testing different choices of the granularity in our parallelized loops. The overall time required by BlackNUFFT is greater than the time required by the reference serial FORTRAN library only if we require less than 8 threads. Fortran has strict aliasing semantics compared to C++ and has been aggressively tuned for numerical performance, for this reason we expect some performance advantage for the reference library if we don't exploit any parallelization.

To better understand the actual importance of each function we draw in the right plot of Figure 3 their relative impact from a computational point of view. We note that the TBB parallelization drastically reduces the computational time of all functions. The gridding on the input array is the real bottleneck of the algorithm since it shows the least optimal behavior. To reach higher computational complexities and reduce even further the computational time, we implement a parallelization strategy based on hybrid shared and distributed memory environment that reduces the overall computational time without loosing the benefits of the shared memory parallelization introduced so far. A hybrid parallelization, combining MPI and TBB, effectively reduces the overall computational time while maintaining a good overall scalability.

4. Distributed memory parallelism

We use the standard Message Passing Interface between different processors to obtain distributed memory parallelism, and to overcome the two main bottlenecks of the multicore parallelization: the input gridding (through FGG) and the three dimensional execution of the FFT on the fine grid. We start by presenting the coding paradigm we have followed and we explain in details each function of NUFFT, and finally we analyze the scaling performances of our new implementation.

4.1. Parallel index set creation

The 3D FFT is the core of the algorithm and we let the back-end FFT library determine the unknown splitting by different MPI processors. We adopt the same unknown splitting over the NUFFT process. We focused our attention on the use of the three dimensional parallel implementation of FFTW and we computed the unknown splitting required by the FFTW MPI routine. This is a simple one dimensional domain decomposition along the coarsest dimension (see Figure

4), and we let the library itself determine the decomposition among the coarsest coordinate of our domain. Subsequently such decomposition is used as a starting point of the overall parallelization. We determine the workbalance among different processors according to the back-end FFT library, and assign the input and output array entries to each processor depending on where their nearest points are placed on the overall fine grid.

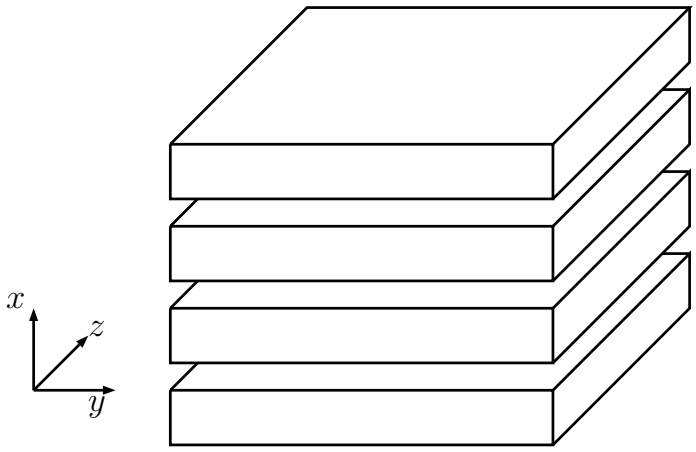

Figure 4: Subdivision for the distributed memory parallelization.

The Gaussian gridding we use has a span q determined by the requested tolerance, and we use this spread to determine the ghost levels of the domain decomposition. In this way we create the distributed array representing the fine grid on which we perform the three dimensional FFT. We represent the fine grid arrays as a distributed ghosted vector subdivided using a mono dimensional decomposition along the coarsest variable. We need to determine two index sets representing the elements owned by a processor and its ghost elements. In this way we handle the ghost cell communications. We need two more sets to complete the necessary MPI pattern. We need to divide the input and output vectors, we do so exploiting the known FFT repartition. More precisely an element of these vectors belongs to the processor if its nearest element on the fine grid belongs to the processor. The ghost cells are already properly set to account for this subdivision. The communications are standard `MPI_Send`, `MPI_Recv`, which are wrapped in the chosen distributed parallel vector [21, 22].

4.2. Implementation

We repeat the breakdown for the MPI-TBB implementation

4.2.1. Input Gridding

We compute the first gaussian gridding from the input array to the fine grid only on those elements that are in the index set we computed following Section 4.1. We don't expect an optimal behavior since we are dealing with an oversampled grid, just as we explained in Section 3.1. If we use a high number of processors some of them will remain idle. At the end we need to communicate the elements eventually added in the ghost cells. This is a suboptimal yet very simple strategy, alternative strategies are currently being developed to prevent processors to become idle. Then we perform the correcting scaling for the first gridding. This is a pointwise operation, and we apply shared memory parallelization directly on all locally owned elements of the fine array. We apply the shift depicted in Section 3.1. We need to check if the elements are stored locally. At the end of the shift we communicate the values updated by the FFT on the fine grid array to the ghost elements of the other processors.

4.2.2. Compute 3D FFT and Data Shift

We simply let the underlying FFT library perform the three dimensional transformation and then we apply standard circular shift to the computed array on the locally owned elements.

Output Gridding. We use TBB as we did for the input, checking that each element of the output vector is on the index set of the current processor. Finally we correct the gridding using a scaling. This is still a pointwise operation so we deal with it with a simple shared memory parallelization after we have checked that the elements are stored locally. At the end we perform a reduction to reconstruct the entire output vector starting from its distributed representation. For the sake of clarity we maintain serial vectors as input-output of BlackNUFFT, since this choice introduces an overhead we are looking for different data structures.

4.3. Strong scaling analysis up to 16 processors

We analyze the characteristic of the MPI parallelization alone, considering a single thread per processor, and compare it with the results obtained with TBB in Section 3.2. We consider the same simulation setting, NUFFT from $2^{21} = 2097152$ to $2^{21} = 2097152$ points. We plot our scalability results in Figure 5. For the time being we use a single computing node with an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz processor. From Figure 5 we see that the most time demanding functions behave similarly. We see how their slope resemble the 3D FFT one. This is due to the communication overhead and to the sub optimality of our Domain Decomposition strategy. However, we see that both the shift function and the scaling on the output vector, which were two functions with optimal TBB

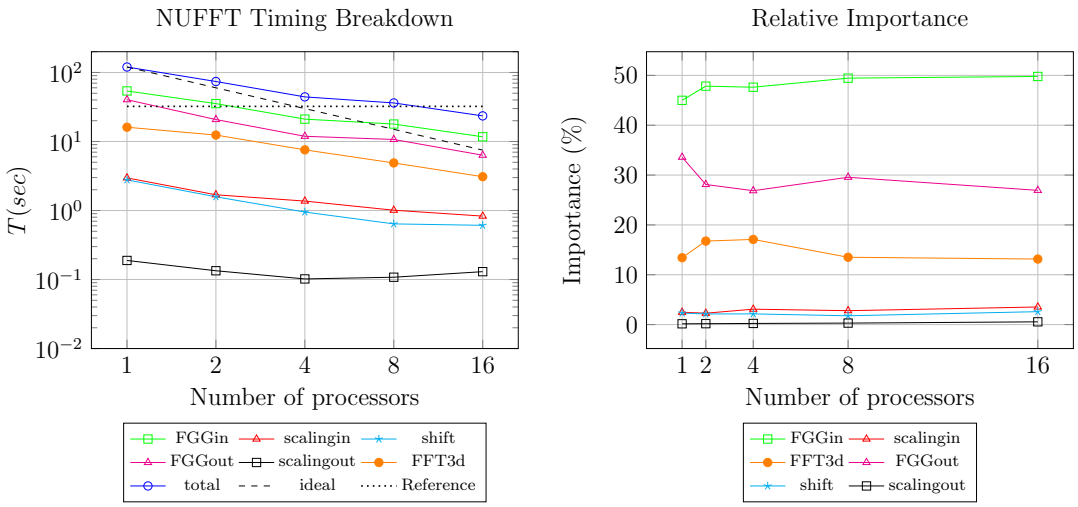

Figure 5: Performance analysis of the parallelization using distributed memory paradigms. Single NUFFT operation on 2097152 arbitrary points, fine grid array of 373248000 points (6 GB of RAM memory). Left figure: timings of all the functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares), total time (blue with circles), linear scalability (dashed), reference algorithm (dotted). Right figure: relative importance plot of the different functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares). We let the number of MPI processors vary from 1 to 16.

scalability (see the left plot of Figure 3) present some issues. In particular both of them do not scale at all with more than 8 processors, however we highlight that we may still use TBB to reduce their timings and that they take less than the 5 % of the overall time. We stress that, for the choices we have made, we can't expect a better overall scalability that the one of the underlying FFT library (FFTW in the present Section), which is the core of our algorithm. From the comparison with the reference FORTRAN library we see that the distributed memory parallelization achieves a performance gain on the original implementation with a number of processors greater than 8.

In the right plot of Figure 3 we report the breakdown of the computational cost of the MPI algorithm. The most demanding functions are almost parallel straight lines, this fact evidences that our parallelization strategy has the same efficiency on all of them. We believe that a coupling of the algorithm with a parallel pruned FFT algorithm, see [8], would significantly reduce the computational cost needed by the FFT on the fine grid array. We are currently addressing this issue.

4.4. MPI TBB comparison

In Sections 3.2 and 4.3 we analyzed separately the two different kinds of parallelism we adopted. In the present Section we perform a comparison between the shared memory and distributed memory paradigms on the same test case scenario introduced in Section 3.2. We fix the overall number of parallel application (processors or threads) and we let the number of MPI processors go from 1 to 16. This provides insights on which parallelization strategy is more convenient, varying from a pure TBB environment (1 processor, with 16 threads) to a pure MPI one (16 processors, each with 1 thread). We use a single socket composed by an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz processor. In the left plot of Figure 6 we draw the timings of our algorithm. We stack all the actual timings to get a clearer representation of the different parallelization efficiencies. Looking to the

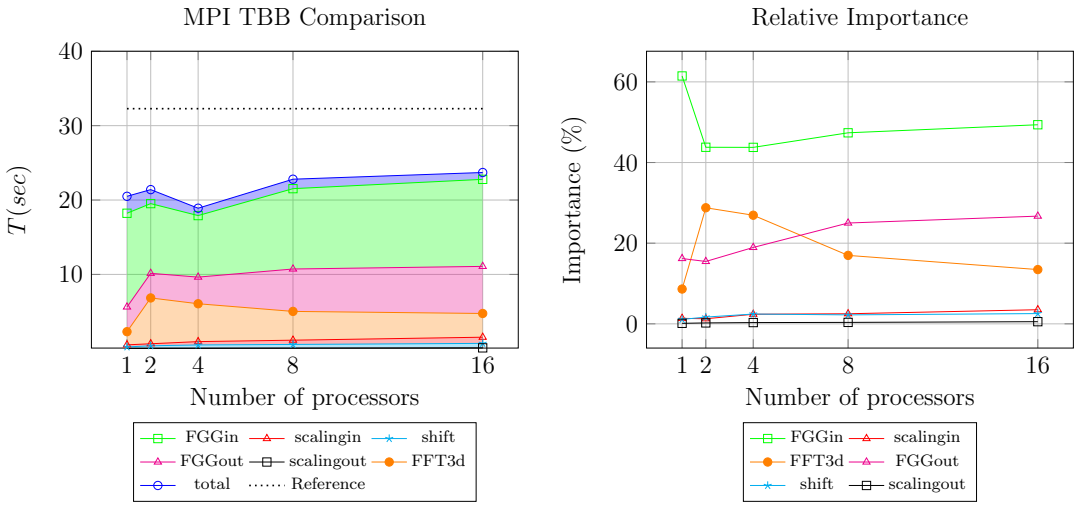

Figure 6: Comparison between the parallelization using different memory paradigms. memory paradigms. Single NUFFT operation on 2097152 arbitrary points, fine grid array of 373248000 points (6 GB of RAM memory). Left figure: timings of all the functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares), total time (blue with circles), reference algorithm (dotted). Right figure: relative importance plot of the different functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares). We let the number of MPI processors vary from 1 to 16 while keeping fixed to 16 the number of overall threads and processors.

overall timing we see that as we begin increasing the number of MPI processors we experience a slight performance gain. We reach a minimum computational time

when we consider 4 MPI processors with 4 threads per processor. This behavior is mainly due to the performances of the hybrid MPI-TBB parallelization of the FGG and FFT algorithms. The other functions behave almost optimally in a pure shared memory environment and this explains the slight increase of the computational time as we required too many MPI processors. In all cases we considered in the present Section our implementation has better performances than the serial reference one.

In the right plot of Figure 6 we draw the relative importances of all the functions varying the number of MPI processors. In this plot a decreasing or flat line means that the MPI implementation behaves better than the TBB one, since the overall time is not varying too much. From the Figure we see that only the input gridding has a performance gain coming from the MPI parallelization. However, we highlight that the possibility of combining shared and distributed memory paradigms is of paramount importance to reach higher complexities that require more than a single computing node as described in Section 4.5.

4.5. 64 bits indices Compatibility

The experimental setting we have considered up to now creates a fine grid array of 373248000 complex values (746496000 actual doubles in the array). However we know that the limit of the 32 bits indexing in C++ is of 4294967296, this means that if we require more data in space or frequency we may overcome such limit. More specifically if we simply consider

$$\frac{R_{max}K_{max}}{\pi} = 200, \tag{10}$$

keeping all the other settings fixed, we get 5971968000 complex elements in the fine grid, and even indexing such an array becomes an issue. We provide BlackNUFFT with the possibility of using 64 bits indexing. We believe that this possibility widens the range of application of the presented library. The only requirement is that the local indexing remains confined to 32 bits indexing. This is required to proper handle local memory, and it does not seem to be a limitation because when we deal with an array of more than 2^{32} elements we unlikely use a single processor. We need to consider that an array of 5971968000 elements can be indexed using 64 bits but, if we consider double precision floating point elements, it needs 48 GigaBytes of RAM memory to be stored. Such a requirement often forces the usage of more than a single computational node and, consequently, more than a single MPI processor.

Analysis up to 32 processors. In Figure 7 we analyse the time needed by each functions of the library to retrieve the results with the computation settings we aforementioned and that guarantee a fine grid array beyond the 32 bits limit, using

$8, 16, 32$ MPI processors. We use 2 nodes composed by an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz socket. The increased dimension of the fine grid worsen

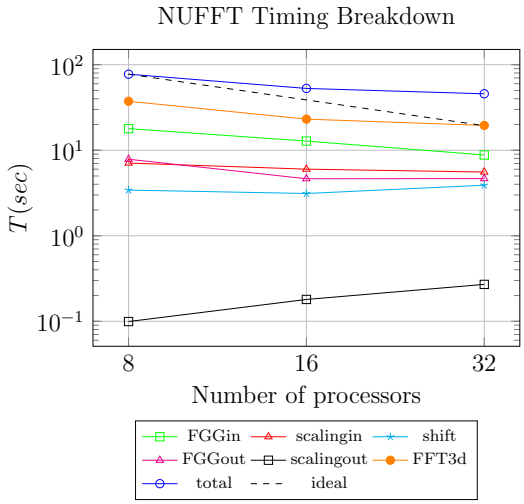

Figure 7: Timing of the functions needed for a single NUFFT operation using a fine grid array exceeding the 32 bits limit: FFT (orange with full circles), FGG on the input array (green with squares), scaling on the input array (red with triangles), shift (cyan with stars), FGG on the output array (magenta with triangles), scaling on the output array (black with squares), total (blue with circles), ideal scalability (dashed).

the performance of the MPI parallelization, this is mostly due to the increased communication overhead. We see that FFTW maintains a good scalability with 64 bit indexing, and we also see that this is the most demanding part of the algorithm. This is induced by the increased number of communications needed by a computation with 64 bits indexing. The scaling on the output vector is not efficient at all and this is due to kind of vector we are requiring as output. We have chosen not to use distributed vector as input-output vectors, therefore we need to recreate the entire array from the distributed memory framework we are using for the NUFFT. This reconstruction requires some communication and this explains the suboptimality. A possible solution to ease the communication effort might be intranode communication if we can exploit a single computational node fulfilling the memory requirements. However we stress that in many cases we cannot require such a node, therefore we need to account for extranode communication if we need to store very big distributed arrays. From Figure 7 we see that the cost of the FFT becomes the major part of the overall computational cost. A possible solution to this issue is the coupling with a parallel pruned FFT algorithm as the one proposed in [8].

Different job scheduling. Using more than a single computational node may lead to another advantage since we are able to use more TBB threads, while keeping the number of MPI processes fixed. We analyze this case scenario comparing the timings of three different executions, exploiting the advantages of both shared and distributed memory parallelizations. We report our results in Table 2

Table 2: Performance comparison with 32 overall parallel processors. We compare three different job scheduling on 32 cores divided in two different nodes. We compute the transformation between 2097152 arbitrary points constructing a fine grid array of 5971968000 complex values. We consider two computing nodes with up to 16 processor per node. We try 2 MPI processors with 16 TBB threads each, 4 processors with 8 threads, 8 processors with 4 threads, 16 processors with 2 threads and 32 MPI processors.

Function	2 MPI 16 TBB	4 MPI 8 TBB	8 MPI 4 TBB	16 MPI 2 TBB	32 MPI 1 TBB
Total	85.1	58.2	40.2	65.7	45.7
Input Gridding	7.56	6.38	8.58	17.1	14.4
FFT	71	45.6	24.8	39	19.5
Data Shift	1.65	1.88	1.96	2.87	3.9
Output Gridding	1.36	1.44	2.32	4.13	4.93

From Table 2 we see that the major benefit of the pure MPI parallelization comes from the FFT routine. However we see that if we require 16 MPI processors with 2 threads per processor the performance of the FFT are worse than if we require 8 processors with 4 threads or 32 MPI processors. This is probably due to the overhead required to spawn the threads with too many MPI processors. Moreover some routines, as the gridding on the input, get advantages also from the hybrid parallelization strategy. All these different effects make the simulation with 8 MPI processors and 4 threads per processor the most convenient from a computational point of view.

5. Coupling with a parallel pruned FFT

We prove the effective modularity of BlackNUFFT interchanging the back-end FFT library. In particular we provide an interface to PFFT [19], which is a generalization of FFTW to pruned FFT. This is straightforward given the modularity of the BlackNUFFT software. Once the back-end library is properly compiled and linked, the modifications required by BlackNUFFT are minimal and mainly regard the peculiarity of using a pruned FFT library instead of a classical FFT. We believe that the usage of such a library improves the performances of BlackNUFFT especially for what concerns the two griddings. The major benefit lies in the fact that the MPI subdivision, which is the one required by the FFT library, is focused

on the meaningful part of the fine grid disregarding completely the oversampling regions. This prevents the stalling of some processors when the number of MPI processors increases, as we pointed out in Section 4.3. With the usage of this library the ghost cell managing and communication strategy of BlackNUFFT is comparable with other library dealing with non uniform FFT as PNFFT [8] which is actually based on PFFT [19] as back-end FFT library.

We test the MPI performances of our implementation on the same test case of Section 4.3, and we report the result in Figure 8. On the left we report the analysis concerning BLackNUFFT with PFFT while on the right we report the timings for the standard BlackNUFFT implementation using FFTW. We see that the two griddings (green squares and magenta triangles) present a more regular scaling using PFFT than using FFTW. This is due to the more efficient implementation of the corresponding index sets that are computed disregarding the oversampling regions. We also note that the overall performance of the NUFFT is comparable and this is due to the great optimization of FFTW that allows for a faster computation of the 3D FFT on the considered grid. However we would like to stress that the better handling of the griddings is a major advantage because it prevents any processor to be idle, as it happens when using FFTW and more than 4 processors.

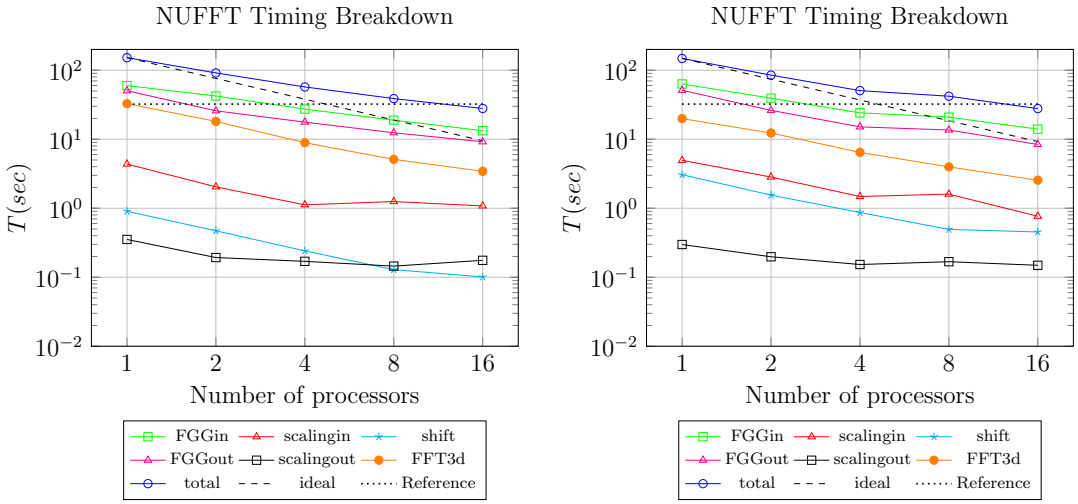

Figure 8: Performance analysis comparison of the parallelization using distributed memory paradigms and different back-end FFT libraries. On the left we analyze the performances using PFFT, on the right we show the performances using FFTW. We report the timings of all the major functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares), total time (blue with circles), linear scalability (dashed), reference algorithm (dotted). We let the number of MPI processors vary from 1 to 16.

We highlight that the overall performance of the griddings is deeply influence by work unbalance between the processors, to better remark the advantages of using PFFT we present another test case where we consider input and output data displaced on a line, so that to erase any kind of work unbalance. We report these results in Figure 9 where we compare the timings of BlackNUFFT and PFFT on the left with the timings of BlackNUFFT coupled with FFTW on the right. We see that the performances of the two grddings greatly increases as a consequence of the pruned FFT library, moreover PFFT behaves more linearly than FFTW on this particular test case. We maintain FFTW as default FFT back-end library to ease the use of BlackNUFFT, FFTW is extremely easy to install (and even already install on many clusters) on any kind of architecture and its usage is supported by a very extensive and complete documentation. The PFFT wrappers can be eventually turned on by the user.

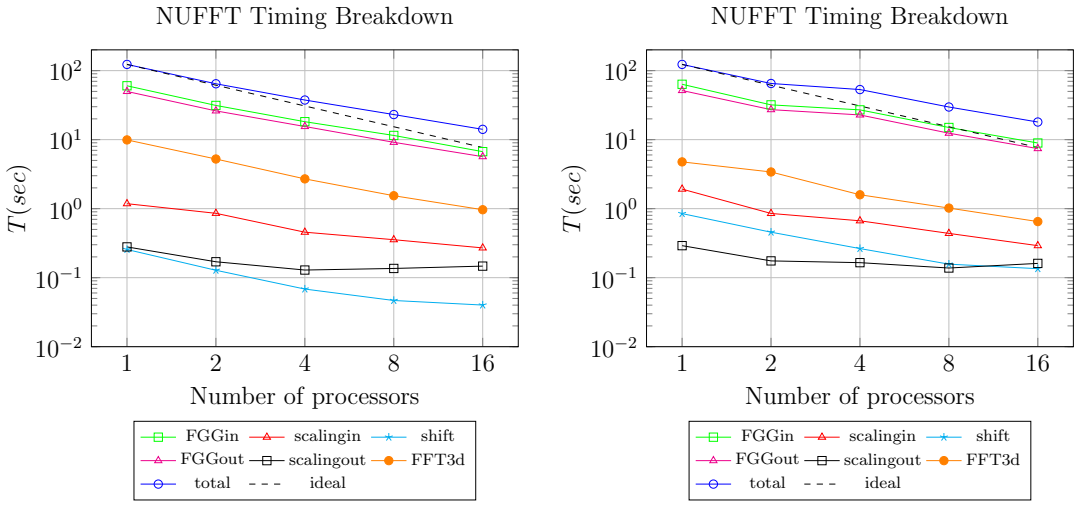

Figure 9: Performance analysis comparison of the parallelization using distributed memory paradigms and different back-end FFT libraries on data distributed along a line. On the left we analyze the performances using PFFT, on the right we show the performances using FFTW. We report the timings of all the major functions, FGG on input (green with square), scaling on input (red with triangles), 3D FFT (orange with full circles), shift (cyan with stars), FGG on output (magenta with triangles), scaling on output (black with squares), total time (blue with circles), linear scalability (dashed). We let the number of MPI processors vary from 1 to 16.

6. Image reconstruction from MRI non uniform data

In this Section we describe a possible application of the presented library. The raw data of Magnetic Resonance Imaging (MRI) is measured in the domain of

spatial frequency (k-space). Such data needs to be transformed into an image in the space domain (r-domain) for diagnosis purposes. While the space domain can be assumed to be a Cartesian grid, the k-space data is usually sampled following non uniform schemes like spiral or radial scans. Such sampling trajectories benefits from non uniform FFT as the one proposed in this work.

We call the sampling frequencies k_j with $j = 0, \ldots, M-1$, we denote the corresponding k-space values as $s(k_j)$, therefore we can introduce an approximation of the image in the space domain, see [1, 3], as

$$p(r) \sim \sum_{j=0}^{M-1} s(k_j) e^{-2\pi i r k_j} w_j, \tag{11}$$

where w_j are weights that compensate the local variations of the sampling density. Following [1] we introduce the operator A^H and the diagonal matrix W representing the weights,

$$A^H_{tj} = e^{-2\pi i r_t k_j}, \tag{12}$$

$$W_{jj} = w_j, \tag{13}$$

that allows us to write (11) as

$$p \sim A^H W s. \tag{14}$$

Following the notation introduced in equation (12) we can introduce the operator representing the forward Fourier transform as

$$A_{tj} = e^{2\pi i r_j k_t}. \tag{15}$$

Equation (14) provides an approximation of the image corresponding to the available sampling in the frequency domain. The bi-dimensional Shepp-Logan phantom, see Figure 10, is a well known benchmark for MRI algorithms [2]. We reconstruct such phantom on a 256×256 Cartesian grid. We consider different samplings of the phantom in the k-space domain and we analyze the results of the reconstruction using a single NUFFT operation. One of the most used sampling strategies is given by the Archimedian spiral, namely

$$k = \pi \frac{\sqrt{j}}{2\sqrt{M_1}} (\cos(\omega_j), \sin(\omega_j)), \tag{16}$$

where $\omega_j = 8\pi/5\sqrt{j}$ with $j = 0, \ldots, M-1$, we consider $M = 65536$ sampling points. Another possibility is given by a standard radial sampling trajectory, namely

$$k = \pi(-1)^r \left(\frac{r}{R} - 0.5\right) \left(\cos\frac{\pi p}{P}, \sin\frac{\pi p}{P}\right), \tag{17}$$

Figure 10: The bi-dimensional Shepp-Logan phantom

where $p = 0, \ldots, P-1$, and $r = 0, \ldots, R-1$, we consider $P = R = 256$. We also consider a sampling density placed on a bi-dimensional Gauss-Legendre quadrature rule composed by 65536 points in $[-\pi, \pi]^2$. Figure 11 reports the three different k-space densities considering 49 samplings for the sake of clarity.

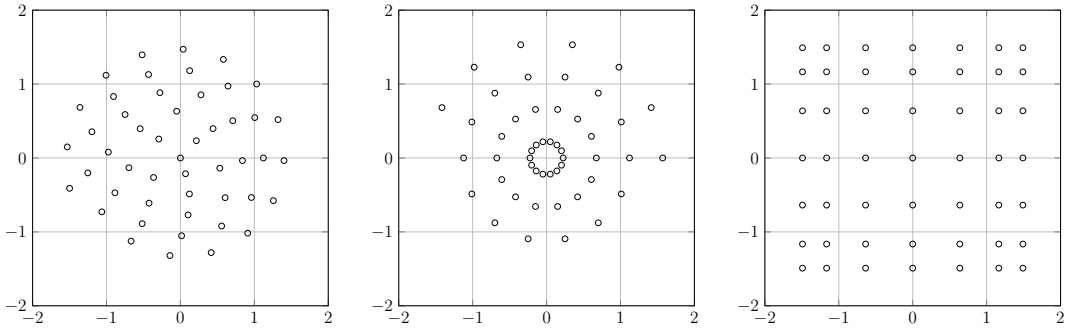

Figure 11: Sampling locations in the k-space domain. On the left we consider the Archimedian spiral, in the middle a radial sampling, and on the left a two dimensional Gauss quadrature.

Following [1] we compute the weights for the spiral and radial case approximating the area of the Voronoi cell around each sampling density. For the sampling following the Gauss-Legendre quadrature rule the weights are the standard bi-dimensional gaussian weights. We use BlackNUFFT to compute (11) considering the three aforementioned samplings. Figure 12 presents the results. We see that the reconstruction is particularly good if we consider the Gauss-Legendre sampling in the k-space domain. The L_2 norm of the error is 23% for the radial sampling,

24% for the spiral sampling and only 9% for the Gauss-Legendre rule.

Figure 12: Images obtained using a single backward NUFFT. On the left the image obtained using the samplings using the Archimedean spiral, in the middle the reconstruction using the radial distribution and on the right the image obtained using the Gaussian sampling.

The non uniform sampling, see [1], in the k-space causes

$$A^H W A \neq I. \tag{18}$$

To improve the image recovering of Figure 12 we can use a larger sampling in the k-space domain and then perform an iterative method to improve the results. In [1] the authors propose a conjugate gradient method to solve

$$A^H W A p = A^H W s. \tag{19}$$

We adopt this strategy to improve the spiral sampling reconstruction, we double the sampling in the frequency domain and we use two conjugate gradient steps. The error of the final image is 9.8%. Figure 13 reports the reconstruction using the enriched spiral sampling and the conjugate gradient method.

The conjugate gradient method we applied is particularly effective if we choose W to be the identity matrix. In Table 3 we report the convergence of the CG method using the aforementioned enriched spiral sampling for both the weighting technique considered, namely using the Voronoi approximation or the identity. We see that the iterative method reduces the error below 10% for both weighting techniques.

Figure 13: Image obtained using 2 steps of the conjugate gradient method proposed in [1] starting from an enriched spiral sampling.

Table 3: Convergence of the conjugate gradient method starting from an enriched spiral sampling. We compare the convergence using the Voronoi weights are a unitary matrix.

Weights / Iteration	0	1	2
None	38%	9.9%	9.8%
Voronoi	11%	9.8%	9.8%

Over the last decade other iterative method have been proposed to improved the reconstruction of MRI images. In [2] the authors propose the use of NUFFT transformation to approximate either the sinc or the sinc^2 kernel to obtain optimal compensation weights. The study of both such techniques is undergone at the moment.

To further test the capabilities of the NUFFT algorithm of type 3 we reconstruct the Shepp Logan phantom on a non regular grid. We consider a ball centered in $(5, 5, 0)$ and having radius 50 and consisting of 7825 points. For such a non standard grid NUFFT of type 2 is not sufficient and we necessarily need a NUFFT of type 3. We consider the same frequency sampling of the previous test case, the resulting image is plotted in Figure 14. We see that the details are well recovered proving that the NUFFT of type 3 we present is not only capable to recover the standard results of [1, 3] but it can also be applied to extend the possibilities of image reconstruction to non standard output grids. We also remarks that the NUFFT type 3 algorithm could be used to reconstruct the frequency sampling given partial image reconstructions as reported in [20].

Figure 14: Deatail of the Shepp Logan phantom obtained using a type 3 NUFFT from arbitrary point in frequency to arbitrary point in space. Image reconstructed on a ball centered in $(5, 5, 0)$ and having radius 50.

7. Introducing new features in the BlackNUFFT framework

This Section demonstrates how new features can be incorporated into the BlackNUFFT framework. We focus our attention on the 2 most time demanding algorithms of the library, the griddings and the three dimensional FFT. Both options are selected through the initializing function `init_nufft` through two different string options provided by the main function.

Introducing a new FFT back-end library. As we showed in Section 5, once the new FFT library has been compiled and properly linked, or included, to the BlackNUFFT library we can easily modify the FFT depending functions using the modularity of our implementation.

- `create_index_sets`: since we have chosen to adopt the MPI redistribution of the back-end FFT library we must provide BlackNUFFT with the proper data distribution coming from the MPI requirements of the external library.
- `compute_fft_3d`: the second change required is the actual call for the back-end FFT library when the fine grid array has been properly set up.

Introducing a new Gridding. We briefly describes how to modify the gridding required by the NUFFT algorithm, thanks to the modularity we have provided this change can easily be achieved by providing new options to the corresponding functions.

- `compute_tolerance_infos`: since each gridding routine determines how the prescribed tolerance is used this function must be set to the specific gridding

requirement. It is sufficient to implement the new case providing a new identifying string.

- `input_gridding`: it is only required to provide the new gridding function. If required it is also possible to split the routine in more functions as we did for the default Fast Gaussian Gridding scheme.

- `output_gridding`: lastly the gridding on the output vector must be modified accordingly to the new chosen gridding.

8. Conclusions

In this work we presented BlackNUFFT, a parallelization of three-dimensional type 3 NUFFT, from arbitrary points in space to different arbitrary points in frequency and viceversa. We have shown that such an algorithm can be easily applied to MRI imaging reconstruction techniques as the ones proposed in [1, 2]. We have implemented the library in C++ and then parallelized it using a hybrid paradigm combining MPI and Intel Threading Building Blocks. We have followed [16, 17] for the shared memory part. The combination of MPI and TBB has already been successfully exploited in [27, 26, 22]. The library is constantly tested using Continuos Integration via TravisCI. We maintained all the characteristics of the algorithm in terms of accuracy of the library by Greengard and Lee [7]. We followed [6] to make sure that the user-required tolerance is respected. We propose two different back-end FFT libraries: FFTW [18] that we consider state-of-the-art, and PFFT [19] that extends FFTW to pruned FFTs. We have implemented Fast Gaussian Gridding on both input and output vectors as a preliminary test of our parallel implementation. However we point out that our approach, thanks to the modularity it applies, allows for a straightforward interchange of the FFT library and of the gridding functions. We saw that a shared memory paradigm, as Intel Threading Building Block, offers some advantages but a distributed memory paradigm is necessary to reach higher efficiency and complexity. We have moved to a distributed memory paradigm, MPI, using the domain decomposition of the fine grid provided by the back-end parallel 3D FFT library. By so doing we gain a significant performance edge with respect to the TBB implementation. We have shown that there is an optimum balance between MPI and TBB depending on all the settings of the NUFFT. We provided the opportunity of using 64 bits indexing. We believe this to be an essential feature if we want to tackle input and output arrays that are widely spread in space or frequency. The spreading of these arrays determines the number of elements in the fine grid array (that we use for the standard 3D FFT), the indexing of such a vector can easily exceed the 32 bit limit. We have found that the communication overhead due to extranode communication may justify the reduction of MPI processors with respect to the number of TBB

threads. However this highly depends on the infiniband connections between different nodes, and on the memory requirements of the computation. We are currently studying the implementation of different gridding and FFT function [8, 9, 10], as well as the application of the developed library to fast convolution methods, and in particular to boundary element method acceleration [4, 5, 28, 29, 30, 31, 32].

Acknowledgements

The author would like to acknowledge Professor Franois Alouges and Doctor Matthieu Aussal from *CMAP - Centre de Mathmatiques Appliques - Ecole polytechnique* for the support and the continuous insights. The author would also like to thank Doctor Michael Pippig for the suggestions regarding the usage of PFFT.

[1] T. Knopp, S. Kunis, D. Potts, International Journal of Biomedical Imaging 2007 (2007) 1–9.

[2] L. Greengard, J.-Y. Lee, S. Inati, Communications in Applied Mathematics and Computational Science 1 (2006) 121–131.

[3] B. Desplanques, J. Cornelis, E. Achten, R. V. D. Walle, I. Lemahieu, IEEE Transaction on Nuclear Science 49 (2002) 2268–2273.

[4] F. Alouges, M. Aussal, Numerical Algorithms 70 (2015) 427–448.

[5] F. Alouges, M. Aussal, A. Lefebvre-lepot, F. Pigeonneau, A. Sellier, 3, 9th International Conference on Multiphase Flow, Florence, Italy, 2016.

[6] A. Dutt, V. Rokhlin, SIAM Journal on Scientific Computing 14 (1993) 1368–1393.

[7] L. Greengard, J.-Y. Lee, SIAM Review 46 (2004) 443–454.

[8] M. Pippig, D. Potts, SIAM Journal on Scientific Computing 35 (2013) C411–C437.

[9] Y. B. Bao, A Parallel Implementation of the 3D NUFFT on Distributed-Memory Systems, 2015.

[10] D. Pekurovsky, SIAM Journal on Scientific Computing 34 (2012) C192–C209.

[11] D. D. Kalamkar, J. D. Trzaskoz, S. Sridharan, M. Smelyanskiy, D. Kim, A. Manduca, Y. Shu, M. a. Bernstein, B. Kaul, P. Dubey, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, 2012, pp. 449–460.

[12] Y. Shu, M. A. Bernstein, J. Huston, D. Rettmann, Journal of Magnetic Resonance Imaging 30 (2009) 1101–1109.

[13] T. Schiwietz, T.-c. Chang, P. Speier, R. Westermann, SPIE Medical Imaging (2006) 1279–1290.

[14] S. Nam, T. Basha, Proc. Intl. Soc. Mag. Reson. Med. 19 (2011) 730339.

[15] N. Giuliani, BlackNUFFT, 2017.

[16] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism, O'Reilly \& Associates, Inc., first edition, 2007.

[17] B. Turcksin, M. Kronbichler, W. Bangerth, ACM Transactions on Mathematical Software 43 (2016) 1–29.

[18] M. Frigo, S. G. Johnson, Proceedings of the IEEE 93 (2005) 216–231.

[19] M. Pippig, SIAM Journal of Scientific Computing 35 (2013) C213–C236.

[20] J. Y. Lee, L. Greengard, Journal of Computational Physics 206 (2005) 1–5.

[21] W. Bangerth, C. Burstedde, T. Heister, M. Kronbichler, ACM Trans. Math. Softw. 38 (2011) 14/1–28.

[22] W. Bangerth, J. Austermann, B. Markus, J. Dannberg, W. Durkin, T. Geenen, A. Glerum, R. Grove, E. Heien, M. Kronbichler, E. Mulyukova, J. Perryhouts, I. Rose, C. Thieulot, I. V. Zelst, S. Zhang, ASPECT: Advanced Solver for Problems in Earth's ConvecTion, 2015.

[23] W. Bangerth, R. Hartmann, G. Kanschat, ACM Trans. Math. Softw. 33 (2007) 24/1–24/27.

[24] L. Greengard, V. Rokhlin, Journal of Computational Physics 73 (1987) 325–348.

[25] M. Kronbichler, T. Heister, W. Bangerth, Geophysical Journal International 191 (2012) 12–29.

[26] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, Journal of Numerical Mathematics 24 (2017).

[27] W. Bangerth, G. Kanschat, Concepts for Object-Oriented Finite Element Software - the deal.II library, Preprint 1999-43, SFB˜359, Heidelberg, 1999.

[28] N. Giuliani, A. Mola, L. Heltai, L. Formaggia, Engineering Analysis with Boundary Elements 59 (2015) 8–22.

[29] N. Giuliani, A. Mola, L. Heltai, Advances in Engineering Software 121 (2018) 39–58.

[30] A. Mola, L. Heltai, A. DeSimone, Engineering Analysis with Boundary Elements 37 (2013) 128–143.

[31] A. Mola, L. Heltai, A. Desimone, in: 18th International Conference on Ships and Shipping Research.

[32] A. Mola, L. Heltai, A. DeSimone, Journal of Ship Research 61 (2017) 1–14.