Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Auto-tuning of the FFTW Library for Massively Parallel Supercomputers

Massimiliano Guarrasi[a], Giovanni Erbacci[a] and Andrew Emerson[a]

[a]*CINECA, Italy*

Abstract

In this paper we present the work carried out by CINECA in the framework of the PRACE-2IP project which had the aim of improving the performance of the FFTW library by refining the auto-tuning mechanism that is already implemented in this library. This optimization was realized with the following activities:

Identification of the major bottlenecks present in the current FFTW implementation;

Investigation of the auto-tuning mechanism provided in FFTW in order to understand how performance is affected by domain decomposition;

Introduction of a new parallel domain decomposition;

Construction of a library to improve the performance of the auto-tuning mechanism.

In particular, we have compared the performance of the standard Slab Decomposition algorithm already present with that obtained using the 2D Domain Decomposition and we found that on massively parallel supercomputers the performance of this new algorithm is significantly higher.

1.Introduction

Currently many challenging scientific problems require the use of Discrete Fourier Transform algorithms (DFT, e.g.:(1)) with one of the most popular libraries used by the scientific community being the FFTW library ((2),(3)).

This library, which is free software, is a C subroutine library for computing DFTs in one or more dimensions with arbitrary input size consisting of both real and complex data. FFTW can also compute discrete Hartley transforms (DHT) of real data and can have arbitrary length. FFTW employs O(n·log(n)) algorithms for all lengths, supports arbitrary multi-dimensional data and includes parallel (multi-threaded) transforms for shared-memory systems and distributed-memory parallel transforms using MPI libraries. FFTW does not use a fixed algorithm for computing the transform, but instead it adapts the DFT algorithm to the underlying hardware in order to maximize performance. Hence, the computation of the transform is split into two phases. First, FFTW's *planner* "learns" the fastest way to compute the transform on the selected machine. The planner produces a data structure called a *plan* that contains this information. Subsequently, the plan is *executed* to transform the array of input data as dictated by the plan. The plan can be reused as many times as needed. In typical high-performance applications, many transforms of the same size are computed and, consequently, a relatively expensive initialization of this sort is acceptable. On the other hand, if you need a single transform of a given size, the one-time cost of the planner becomes significant. For this case, FFTW provides fast planners based on heuristics or on previously computed plans. During plan creation, users can choose the method that he/she prefers using the FFT_MEASURE (obtaining a more accurate plan) flags or FFTW_ESTIMATE flags (obtaining the plan faster).

Currently, particularly using small size data arrays, the FFTW libraries have been shown to not scale well beyond a few hundred cores. Considering that current PRACE Tier-0 systems consist of several hundreds of thousands of cores, and that in order to obtain an access on these systems a good scalability of a few thousand of cores at least is required, there is a clear need to improve FFTW implementations on massively parallel supercomputers.

For this purpose, a large amount of time must be first dedicated to performing extensive benchmarks in order to find the major

bottlenecks of the auto-tuning mechanism. Thus, a significant part of the next two sections will be devoted to the description of these benchmarks that were performed using two parallel supercomputers present in the CINECA infrastructure:

- The PLX cluster (4) chosen in order to test the performance on a typical Tier-1 system. The PLX system is an IBM iDataPlex DX60M3 Linux Infiniband cluster. It consists of 274 IBM X360M2 12-way compute nodes Each one contains 2 Intel(R) Xeon(R) Westmere six-core E5645 processors (5), with a clock of 2.40GHz. All these compute nodes have 47GB of memory.
- The FERMI cluster (6), chosen in order to test the performance on a typical Tier-0 system. FERMI is a Blue Gene/Q system (7)and has a massively parallel architecture. Each Compute Card (which we call a "compute node") features an IBM PowerA2 chip with 16 cores working at a frequency of 1.6 GHz, with 16 GB of RAM and network connections. A total of 32 compute nodes are plugged into a so-called Node Card. Then 16 Node Cards are assembled in one midplane which is combined with another midplane and two I/O drawers to give a rack with a total of 32x32x16 = 16K cores.. In our BG/Q configuration there are 10 racks for a total of 160 K cores.

Our optimization activities were mainly aimed at improving three aspects of the FFTW library that seemed to us to be the most promising: the optimization of the number of cores used during a DFT execution (this is particularly important for low-dimensional arrays, e.g. 2D arrays), the implementation of a new domain decomposition algorithm and a new auto-tuning algorithm used to switch from the standard parallel domain decomposition algorithm to the new one (we focus our attention on 3D DFT in particular).

Before starting to describe in detail our development activities, it will be useful to describe the FFTW mechanisms used to make the parallel domain decomposition. With a serial or multithreaded FFT, all of the inputs and outputs are stored as a single contiguous chunk of memory. With a distributed-memory FFT, the inputs and outputs are broken into disjoint blocks, one per process.

In particular, FFTW uses a *1D block distribution* of the data, distributed along the *first dimension*. For example, if you want to perform a 100 × 200 complex DFT, distributed over 4 processes, each process will get a 25 × 200 slice of the data. This method is also known as the *Slab Decomposition* algorithm. Clearly, it cannot be used on 1D arrays so for these another method is used, but the intrinsic structure of FFT algorithm in any case does not lead to a good parallelization of the 1D transforms. Thus we excluded the 1D DFT in our optimization work.

Furthermore, it is quite easy to see that the size of the index involved in the parallel domain decomposition may provide us with a limit of the maximum number of usable cores. In addition, if the size of this index isn't a multiple of the number of used cores, it could cause a large load unbalance, which would also affect the performance.

Section 2 will be mainly devoted to investigate these features, especially in the 2D cases, where we might expect that these issues are more important. For many-dimensional arrays (3D, 4D, and so on), the issues related to the size of the index involved in the parallel domain decomposition are even more evident, but in these cases other parallel domain decomposition algorithms could be used. For simplicity, our work focused on the implementation of one of the algorithms, namely the case of 3D DFTs (that are also the most currently used).

Using this Slab Decomposition, the parallel computation of the 3D DFT can be divided into four steps:

- 1D Parallel Domain Decomposition
- 2D FFT along the two local dimensions;
- Global transposition;
- 1D FFT along the third dimension.

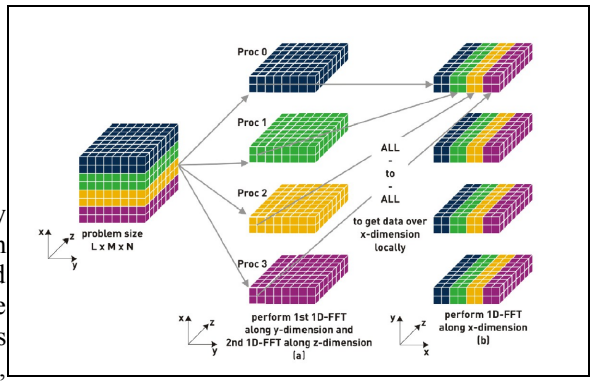

Figure 1: Slab decomposition of a 3D DFT. From (8).

This decomposition is faster on a limited number of cores because it only needs one global transpose, minimizing communication. The main disadvantage of this approach is that the maximum parallelization is limited by the length of the largest axis of the 3D data array used. The performance can be further increased using a hybrid method, combining this decomposition with a thread-based parallelization of the individual FFTs, but the performance increases only slightly. In any case, for a cubic array with N^3 data points, the maximum number of usable cores scales only as N.

In order to improve the performance on massively parallel supercomputers significantly, an innovative approach is necessary. Thus, in order to reduce the scaling limitation we use a 2D Domain Decomposition ((8), henceforth referred as $2D^3$). In this case, the computation will be done in six steps:

- 2D Parallel Domain Decomposition
- 1D FFT along the first local dimension;
- Global transposition;
- 1D FFT along the second dimension;
- Global transposition;
- 1D FFT along the third dimension.

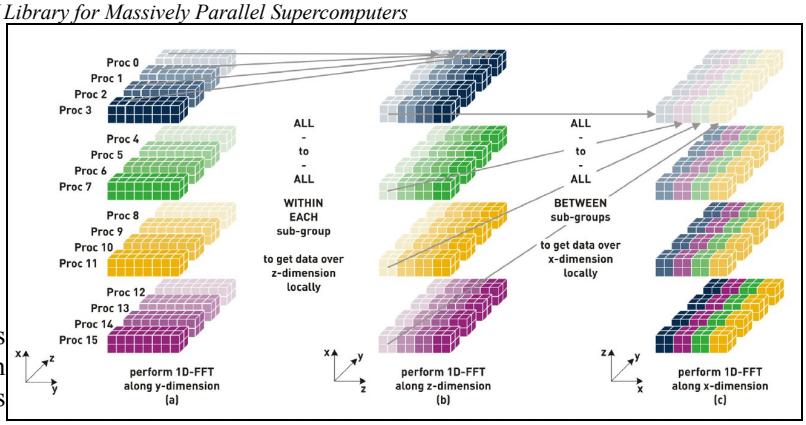

Figure 2: 2D Decomposition of a 3D DFT. The array was divided between 16 MPI process. From (8).

Using this method, another global communication is required. Nevertheless, these global transposition steps require communication only between subgroups of all nodes. In other words, in the DFT computation, this algorithm pays a higher cost in terms of MPI communications, but it ensures higher scalability. Indeed, for a cubic data array with N^3 data points, this means that the maximum number of cores scale as N^2, significantly improving the number of usable cores. Therefore, we can expect that the performance of the $2D^3$ algorithm (if compared with the performance of standard FFTW) will be higher using a larger number of cores, and lower using fewer cores. We also expect that this gap will be even higher on a cluster whose communication network is particularly powerful, (e.g. Blue Gene/Q network). In order to implement this new $2D^3$ algorithm on FFTW we use the one provided in the 2Decomp&FFT library (9).
In Section 3 we will show the performances of this new algorithm compared to the one used by the standard FFTW and describe the improved auto-tuning mechanism used to switch from the standard to the new algorithm.

2. Performance of the 2D FFTW auto- tuning algorithm

This section will be devoted to improving the FFTW auto-tuning mechanism by investigating the major bottlenecks in the 2D DFT execution using FFTW on a moderate number of cores. For this reason the tests reported in this section were performed on the PLX cluster. In the first part of the section we will show the bottleneck of 2D DFT execution, reporting a series of targeted tests, whereas in the last part we will show how we improved the efficiency of the standard FFTW algorithm.
For our benchmark activity we selected five 2D arrays, with sizes : 256x256, 1024x1024, 8192x9192, 1024x16384 and 16384x1024. These arrays are sufficiently large to show us the limits of the parallel implementation of the FFTW library, and they are reasonably representative of a large class of 2D problems solved using FFTW libraries on modern medium-size parallel supercomputers.

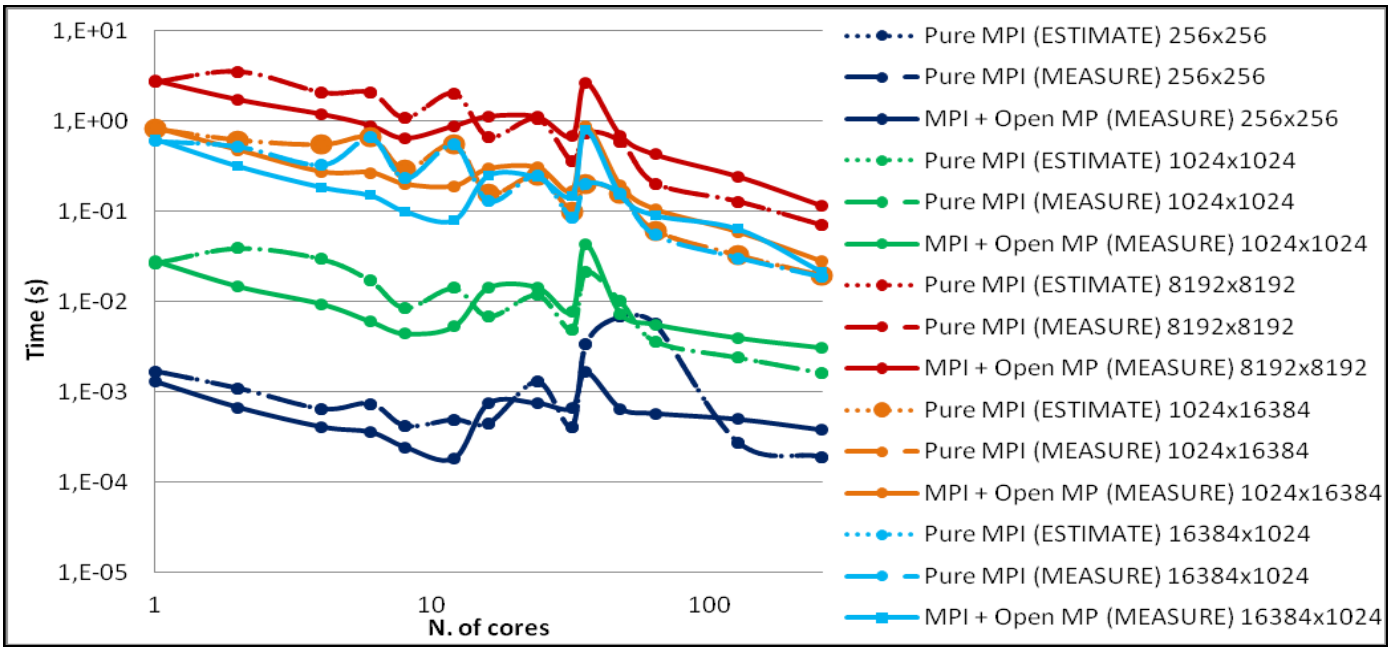

Figure 3: Plot of the execution times reported in Table 1, using MPI communications (both FFTW_MEASURE and FFTW_ESTIMATE cases), and hybrid MPI+OpenMP communications (only FFTW_MEASURE case).

MPI	Number of cores	ESTIMATE					MEASURE				
		256 x 256	1024 x 1024	8192 x 8192	1024 x 16384	16384 x 1024	256 x 256	1024 x 1024	8192 x 8192	1024 x 16384	16384 x 1024
	1	1.7	26.7	2685.9	812.0	602.9	1.7	26.7	2685.9	812.0	602.9
	2	1.1	39.1	3497.4	608.4	522.8	1.1	39.1	3497.4	608.4	522.8
	4	0.6	29.6	2054.3	542.5	326.9	0.6	29.6	2054.3	542.5	326.9
	6	0.7	17.0	2040.0	652.2	651.6	0.7	17.0	2040.0	652.2	651.6
	8	0.4	8.5	1074.4	286.1	230.8	0.4	8.5	1074.4	286.1	230.8
	12	0.5	14.1	2007.0	546.5	542.8	0.5	14.1	2007.0	546.5	542.8
	16	0.4	6.9	664.4	156.5	129.7	0.4	6.9	664.4	156.5	129.7
	24	1.3	11.8	1046.3	246.6	247.5	1.3	11.8	1046.3	246.6	247.5
	32	0.4	5.0	362.1	99.2	85.2	0.4	5.0	362.1	99.2	85.2
	36	3.4	21.0	730.9	198.9	201.5	3.4	21.0	730.9	198.9	201.5
	48	6.8	10.1	583.6	155.8	156.2	6.8	10.1	583.6	155.8	156.2
	64	5.6	3.6	202.3	60.7	54.4	5.6	3.6	202.3	60.7	54.4
	128	0.3	2.4	127.3	32.4	30.2	0.3	2.4	127.3	32.4	30.2
	256	0.2	1.6	69.7	19.3	18.6	0.2	1.6	69.7	19.3	18.6
MPI + Open MP	**Number of cores**	**ESTIMATE**					**MEASURE**				
		256 x 256	**1024 x 1024**	**8192 x 8192**	**1024 x 16384**	**16384 x 1024**	**256 x 256**	**1024 x 1024**	**8192 x 8192**	**1024 x 16384**	**16384 x 1024**
	1	1.3	27.6	2765.9	832.9	624.7	1.3	28.3	2756.6	832.8	625.7
	2	0.7	14.6	1685.5	443.6	311.4	0.7	14.7	1697.1	472.7	314.1
	4	0.4	8.8	970.7	276.5	172.8	0.4	9.4	1175.8	272.7	184.3
	6	0.4	5.8	889.0	266.4	151.3	0.4	6.0	881.3	267.1	149.8
	8	0.2	4.3	639.5	198.8	102.1	0.2	4.4	640.0	199.1	99.5
	12	0.2	5.2	874.6	189.8	79.3	0.2	5.3	876.1	189.8	79.9
	16	0.8	14.5	1107.5	294.6	249.3	0.7	14.3	1112.6	294.1	248.9
	24	0.8	14.5	1097.2	303.0	237.1	0.8	14.4	1102.6	303.5	237.6
	32	0.7	7.7	635.0	167.4	148.2	0.7	7.7	676.4	168.3	149.1
	36	1.7	43.1	2838.0	789.2	786.9	1.7	43.3	2601.5	867.0	790.7
	48	0.7	7.4	684.0	191.1	151.9	0.7	7.4	685.7	191.1	152.4
	64	0.6	5.6	395.2	102.8	89.4	0.6	5.6	423.7	105.5	90.6
	128	0.5	3.9	218.4	58.7	63.0	0.5	3.9	242.0	59.2	63.6
	256	0.4	3.1	95.8	28.1	21.5	0.4	3.1	115.9	28.2	21.6

Table 1: Execution times (ms) of five 2D complex-to-complex DFT, performed using FFTW 3.2 on PLX using from 1 to 256 cores. We used both Pure MPI, and Hybrid MPI + Open MP parallel communication methods, generating plans using both the FFTW_ESTIMATE and FFTW_MEASURE flags.

Using these data arrays we performed a scalability test from1 to 256 cores of the PLX cluster, comparing the execution times of the complex-to-complex DFT obtained using both pure MPI then the hybrid MPI+OpenMP implementation of FFTW v3.2. We repeated the same tests also generating the corresponding plans using both the FFTW_ESTIMATE and FFTW_MEASURE flags. These execution times were averaged over 512 runs and reported in Table 1. They are also plotted in Figure 3, where we compare only the execution times obtained using:

- FFTW routines for MPI communications and the FFTW_ESTIMATE flag (dotted lines);
- FFTW routines for MPI communications and the FFTW_MEASURE flag (dashed lines);
- FFTW routines for hybrid MPI + Open MP communications and the FFTW_MEASURE flag (continuous line).

It can be seen from the figure that the execution times obtained using FFTW_MEASURE are essentially identical to the results obtained using FFTW_ESTIMATE (in fact they differ by only about 1-3%). This evidence allows us to consider only the results from one of the two methods (we chose the FFTW_MEASURE flag) as representative of the performance of the FFTW library. This is also the reason why we omitted the results of scalability tests obtained using the hybrid implementation and the FFTW_ESTIMATE flag.

In Figure 4 we compare the efficiency obtained in the scalability tests shown in Figure 3. We plot only the scalability test obtained using only the FFTW_MEASURE flag and pure MPI communications (dashed lines) or using MPI + Open MP communications (continuous lines). This allows us to confirm what has already been obtained from Figure 3, i.e.:

- Using a large number of cores, the use of pure MPI communication give us a better efficiency if compared with the results obtained from the hybrid implementation. Since we are interested in the performance of the FFTW library on a large number of cores (from hundreds to thousands), we will focus our attention only on the pure MPI implementation of the library.
- When the number of cores is small (≤12 cores on PLX, i.e. 1 node card) the performance of the hybrid implementation (MPI + Open MP) is greater than the pure MPI one. This difference is probably due to the higher efficiency of the Open MP communications on a single shared memory node card.
- For moderately large number of cores (e.g. ~100) the efficiency of the execution of the DFT computation becomes small. This is particularly evident when the size of the last index (the one involved in parallel domain decomposition in our case) is comparable whit the number of cores used. On the other hand, if we use the hybrid approach the performance increases beyond this limitation, depending on the size of the second index of the data array. Moreover, generally speaking, when the number of cores is comparable with the size of the last index, the efficiency of FFTW becomes very low.
- If more than one node card was used, the efficiency of the codes increased with the size of the array.

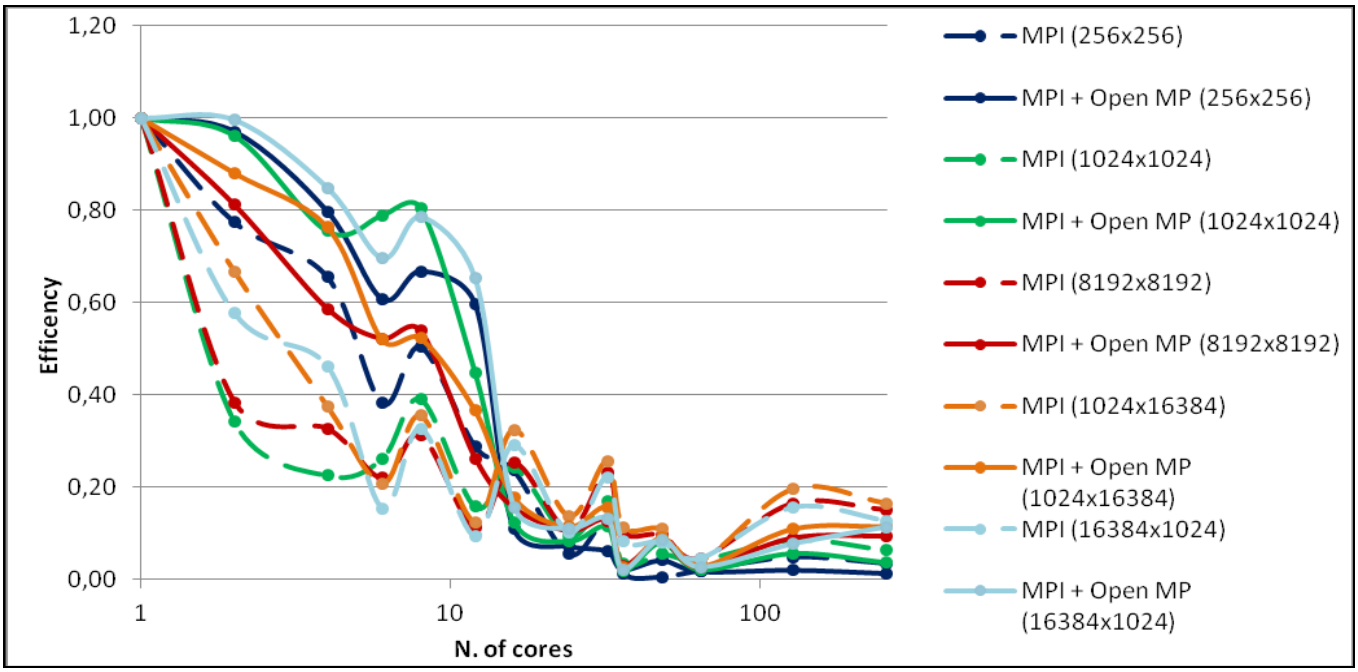

Figure 4: Efficiency obtained during the scalability test of Figure 3

Furthermore, using this figure we can also clearly see another direct consequence of the slab decomposition algorithm: considering that, in our case, for the FFTW library, the index involved in the parallel domain decomposition is the last[1], we see that the greater the ratio between the size of the last index of our array and the number of used cores, the greater will be the efficiency of FFTW. This is more evident comparing the efficiency on 256 cores for sky-blue, orange, and red lines (corresponding to arrays of 16384x1024, 1024x16384, and 8192x8192 points respectively). As expected, we obtain the best performance for the orange line (last index = 16384), followed by the red line (last index = 8192), and then the sky-blue line (last index = 1024). Thus, in order to improve the performance, it would be better rearrange the array to have the maximum size on the last index.

Moreover, from the same figure we also notice that, although generally the execution time decreases with the number of cores, when the size of the first index isn't a multiple of the numbers of cores, the execution time increases again (due to load unbalancing, as expected).

Finally, in Table 2, we have also analyzed the time needed to create the plans used for the execution of the DFT reported in Table 1. It can be seen from Table 1 and Table 2 that since the time needed to create a plan with FFTW_MEASURE may be hundreds of times that needed for the execution of the plan itself, it makes sense to use this flag in preference to FFTW_ESTIMATE only when the execution of the plan must be repeated very frequently, for example, many thousands of times.

[1] Please note that it is true only for FORTRAN users. The index involved in the parallel domain decomposition for C or C++ users is the first.

MPI	Number of cores	ESTIMATE					MEASURE				
		256 x 256	**1024 x 1024**	**8192 x 8192**	**1024 x16384**	**16384 x 1024**	**256 x 256**	**1024 x 1024**	**8192 x 8192**	**1024 x16384**	**16384 x 1024**
	1	1.3	1.1	1.3	1.1	1.3	137.3	1076.5	23813.3	8738.2	11841.8
	2	2.9	2.4	2.8	2.6	5.1	145.9	1846.3	44204.6	15210.0	22637.4
	4	3.2	2.6	3.1	3.2	4.1	92.3	1014.3	28683.4	15199.2	15892.3
	6	1.8	2.3	15.1	4.3	5.0	23.3	199.8	8627.8	2870.5	2928.3
	8	5.2	2.9	2.9	2.7	4.1	92.8	608.7	21744.5	12617.5	13340.7
	12	2.2	2.9	68.1	7.9	8.7	26.4	235.4	4837.7	2721.2	2644.6
	16	9.3	4.0	4.8	6.6	6.5	103.1	585.2	18686.2	9674.2	7932.9
	24	2.9	3.6	19.4	6.4	4.7	20.8	759.7	2432.0	2124.6	2111.2
	32	15.2	7.8	12.4	7.6	4.4	163.0	1128.8	11889.0	8123.7	3495.5
	36	3.3	4.1	6.2	17.9	6.4	20.6	202.5	1755.1	1592.0	1602.4
	48	4154.0	4.6	24.1	14.1	14.7	20.2	358.4	1419.6	1655.2	1642.8
	64	512.0	18.4	8.9	6.4	5.7	11200.0	1868.8	9247.2	5566.4	2599.2
	128	34.0	3406.1	8.2	15.1	17.5	706.7	4297.1	4316.8	5284.6	3816.8
	256	23.1	57.3	18.1	41.5	41.1	616.7	13406.3	6032.2	6377.4	5282.7
MPI + Open MP	**Number of cores**	**ESTIMATE**					**MEASURE**				
		256 x 256	**1024 x 1024**	**8192 x 8192**	**1024 x 16384**	**16384 x 1024**	**256 x 256**	**1024 x 1024**	**8192 x 8192**	**1024 x 16384**	**16384 x 1024**
	1	0.2	0.6	3.0	1.7	1.7	92.0	1136.4	23475.1	8920.0	11790.0
	2	0.2	0.6	3.0	1.6	1.8	140.0	1451.8	44363.1	11254.3	16421.6
	4	0.4	0.9	1.1	2.1	2.0	121.3	1291.2	41573.8	8988.7	14626.0
	6	0.5	0.8	1.3	2.2	2.2	129.6	1431.8	50393.2	11497.5	18232.2
	8	0.6	1.0	3.1	5.9	2.5	109.3	1219.3	42246.1	8067.0	11327.3
	12	0.9	2.7	4.0	2.7	4.1	132.3	1332.0	49665.1	9813.3	14687.7
	16	1.4	2.1	2.2	3.3	3.2	188.4	1155.6	41630.9	15371.8	21510.2
	24	1.7	3.3	4.9	4.2	4.9	218.7	1490.1	46190.4	17656.9	23818.4
	32	1.7	2.6	4.9	3.7	4.2	118.4	691.9	21420.6	10963.8	12754.6
	36	3.7	3.7	104.4	44.8	8.0	76.3	580.6	21054.1	5873.7	5770.3
	48	2.7	2.9	23.7	6.3	7.0	146.6	794.7	24380.8	11959.5	13919.0
	64	2.1	2908.0	22.3	7.2	7.6	139.2	603.7	16676.5	8915.7	8342.5
	128	3.6	3.1	5.4	5.4	11.8	157.2	537.3	12982.7	6673.0	5885.1
	256	5.6	7.6	13.8	6.4	6.2	136.7	9704.8	7622.9	5661.6	2429.8

Table 2: Time needed for plan creation relating to the same 2D complex to complex DFT performed on *Table 1*.

By summarizing the results of our initial benchmarks, we can identify some strategies for improving the performance of the FFTW library:

- Using small size arrays, the parallel performance of FFTW degrades significantly when we use more than one node (e.g. when we pass from a shared memory to a distributed memory system);
- Using more than one node the best performance will be obtained using pure MPI communications;
- If we increase the number of points in the arrays (especially on the last index, but see footnote 1), the performance should significantly improve;
- Generally, it may not be advantageous to use more cores than the number of points of the index involved in the domain decomposition;
- When the size of the first index is not a multiple of the numbers of cores, the performance decreases.

Thus, considering the limitations above, we have developed a routine that improves the auto-tuning mechanism of the FFTW

library by changing the number of cores used. A simple scheme of the structure of the algorithm that we use in this routine is shown in Algorithm 1. This subroutine, having as input the size of the index involved in the parallel domain decomposition of the array that we want to use (Nx), and the number of available cores (Np), gives us the optimal number of cores in order to maximize the performance.

```
IF (Np > Nx) then
    Np = Nx
ELSE
    IF (Nx is NOT a whole multiple of Np) then
         FOR (NN < NP and NN ~ NP)
             IF (Nx is a whole multiple of NN) then
                Np = NN
             END IF
        END FOR
        ENDIF
    END IF
END IF
```

Algorithm 1:Algorithm used to adapt the number of cores used for computation to the size of the array.

3, Performance of the 3D FFTW auto-tuning algorithm using a 2D Domain Decomposition algorithm

This section will be devoted to removing some bottlenecks of the FFTW library related to the DFT execution of multidimensional arrays on massively parallel supercomputers. For this reason the tests reported in this section were performed on the FERMI cluster. For simplicity we will analyze only the case of 3D DFT execution, but our results can be easily extended to other multidimensional arrays. The first part of this section will be devoted to the comparison between the performance of the standard parallel decomposition used by the FFTW library and the performance of the 2D Domain decomposition that we have implemented. The last part of this section will be dedicated to the description of our auto-tuning mechanism, used to switch from the standard domain decomposition algorithm to the new one.

For our comparison we selected seven 3D arrays, whose sizes are: 128x128x128, 256x256x256, 512x512x512, 1024x1024x256, 1024x1024x512, 1024x1024x1024, 1024x1024x2048. We can see that the sizes of the index involved in the standard parallel domain decomposition are comparable with the ones used in Section 2 (NB: the last index in the case of FORTRAN, see footnote 1).

We have carried out a scalability test, using from 256 to 2048 cores, comparing the performance of the two algorithms using these seven data arrays. As before, the execution times for these tests are first averaged over 256 runs, and then reported in Table 3.

Sistem & Solver		**Number of Cores**				
Domain Decomposition	**N. of Points**	**256**	**512**	**1024**	**2048**	**4096**
Slab Decomposition	128x128x128	8	10	14	21	31
	256x256x256	22	28	35	42	57
	512x512x512	175	119	161	154	142
	1024x1024x256	376	420	530	524	514
	1024x1024x512	788	499	640	615	575
	1024x1024x1024	1626	1047	741	693	702
	1024x1024x2048	3275	2160	1598	814	798
2D Decomposition	128x128x128	23	2	1	1	0
	256x256x256	188	13	8	4	2
	512x512x512	1718	111	73	33	17
	1024x1024x256	1723	963	625	310	150
	1024x1024x512	1718	980	632	310	151
	1024x1024x1024	1722	984	632	313	151
	1024x1024x2048	1722	984	631	311	152

Table 3: Execution times (ms) of seven 3D complex-to-complex DFT, performed using FFTW 3.3 on FERMI cluster using from 256 to 2048 cores, comparing the performances of both the Slab and the 2D decomposition algorithms.

RATIO Slab/2DDD						
Configuration	**N. of points**	**Number of Cores**				
		256	***512***	***1024***	***2048***	***4096***
128x128x128	2,10E+6	0.4	5.2	14.4	21.6	63.9
256x256x256	1,68E+7	0.1	2.2	4.5	10.7	29.1
512x512x512	1,34E+8	0.1	1.1	2.2	4.6	8.5
1024x1024x256	2,68E+8	0.2	0.4	0.8	1.7	3.4
1024x1024x512	5,37E+8	0.5	0.5	1.0	2.0	3,8
1024x1024x1024	1,07E+9	0.9	1.1	1.2	2.2	4,6
1024x1024x2048	2,15E+9	1.9	2.2	2.5	2.6	5.2

Table 4: Ratio between execution times using the 2D Domain Decomposition algorithm and the same time using standard Slab Decomposition. Color code: red cells: time of standard FFTW < time of 2D decomposition, yellow cells: time of standard FFTW ~ time of 2D decomposition, green cells: time of standard FFTW > time of 2D decomposition and blue cells: time of standard FFTW >> time of 2D decomposition.

In Table 4 we show the ratio between the execution times of the DFT using standard Slab Decomposition and 2D Decomposition respectively. In this table the red cells indicate the cases where it is more opportune to use slab decomposition (ratio $\leq$ 1), yellow cells where the results of the two algorithms are comparable (1 < ratio $\leq$ 2), green cells where it is advantageous to use the $2D^3$ algorithm (2 < ratio $\leq$ 10), and finally blue cells where it is very advantageous to use the 2D Decomposition (ratio > 10). We can see from the table that if we are using few cores (i.e. 256, as reported on Table 4), then the best performances will be obtained using Slab Domain decomposition. On the other hand, if we use thousands of cores, especially on small size arrays, the 2D Domain Decomposition algorithm will be much more efficient than the standard Slab Decomposition method. Our tests show in particular that the $2D^3$ algorithm appears to be up to 63 times faster than the Slab Decomposition algorithm.

The significance of these benchmarks becomes clearer if we plot the execution times reported in Table 3 (see Figure 5). It is evident from the figure that when using small size arrays the standard FFTW does not scale beyond 256 cores. Indeed, as expected, when the number of cores exceeds the size of the first index of the array, the time of execution of the DFT begins to increase. Conversely, the time of execution of the 2D Decomposition algorithm continues to scale ideally even using thousand of cores, although when we use only 256 cores its execution time is longer than the time obtained using the Slab Decomposition algorithm.

This is more evident from Figure 6, where we show the ratio between the execution time of DFT of Table 3 and the corresponding time obtained using standard FFTW on 256 cores. In this figure, all lines starts from the same position. Dotted lines are related to standard FFTW efficiency profile, whereas dashed lines are related to 2D Domain Decomposition ones. In other words, this figures plots the times of execution of the algorithms normalized to the standard FFTW execution time on 256 cores. As a comparison, in the same figure we plot also the line of ideal scalability (red continuous line). It is easy to see that all 2D Decomposition lines scale ideally, whereas when the number of cores exceed the size of the index involved in the parallel domain decomposition (the last in our case), the standard FFTW shows a very bad scalability (ratio > 1).

Finally, in Table 5 we show the times needed to create the corresponding plans for DFT in Table 3. In order to quantify better the advantages obtained by using this algorithm, we report also the time cost to create the same plans using standard FFTW and the FFT_ESTIMATE flag. As we can easily see from this table, the computational cost for the creation of a plan using 2D Decomposition is always comparable with the one obtained using the FFTW_ESTIMATE flag (i.e. the same order of magnitude of the time needed to execute the DFT) and significantly smaller than the one obtained using the FFTW_MEASURE together with the Slab Decomposition. This observation is not surprising, since it is due to the different method used in the 2D algorithm to create a plan: using 2D Domain Decomposition, we compute 1D DFTs for each core, whereas using Slab Decomposition we perform a series of 2D DFTs for each core and moreover creating a 1D plan is clearly faster than creating a 2D plan.

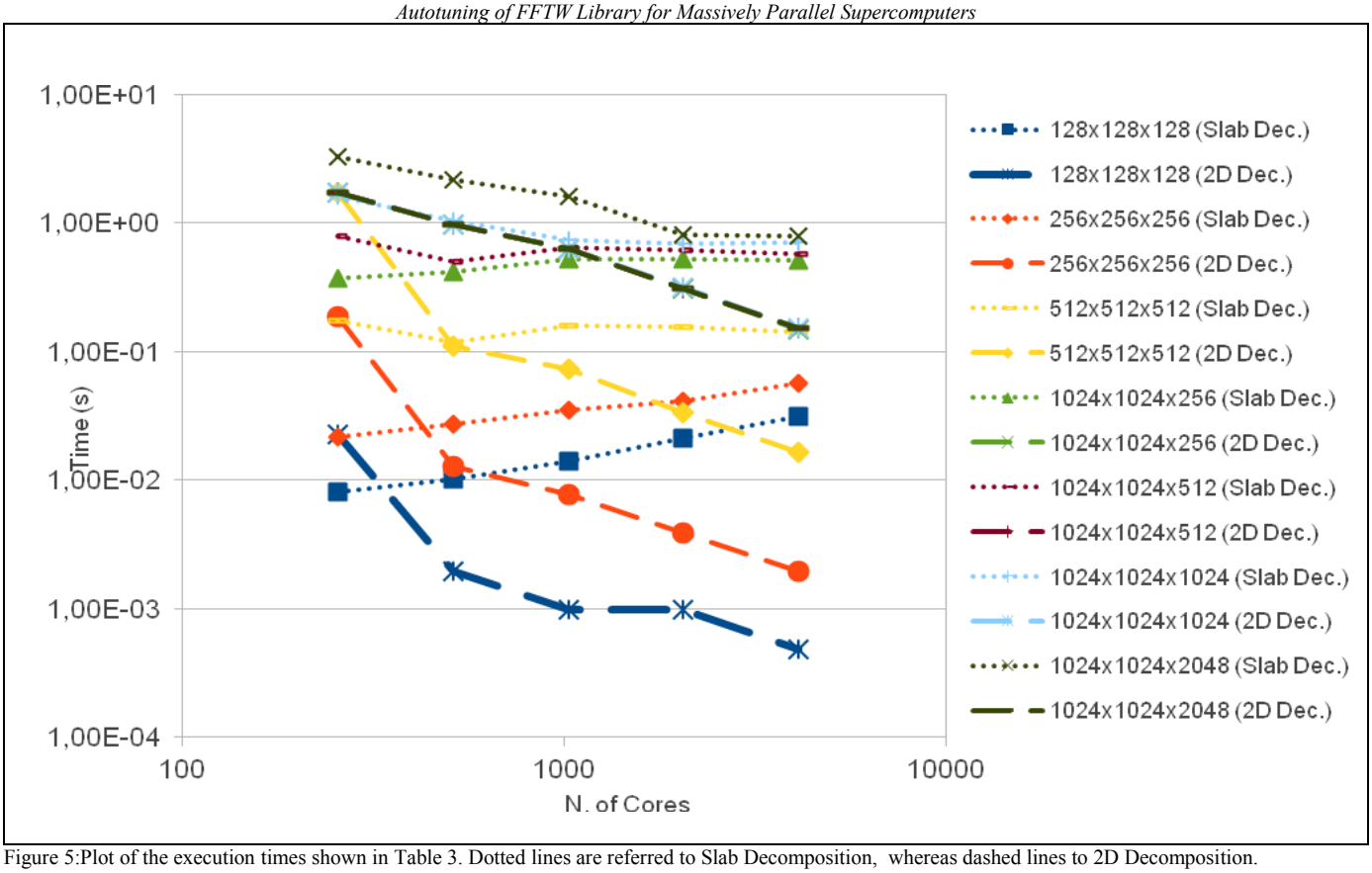

Figure 5:Plot of the execution times shown in Table 3. Dotted lines are referred to Slab Decomposition, whereas dashed lines to 2D Decomposition.

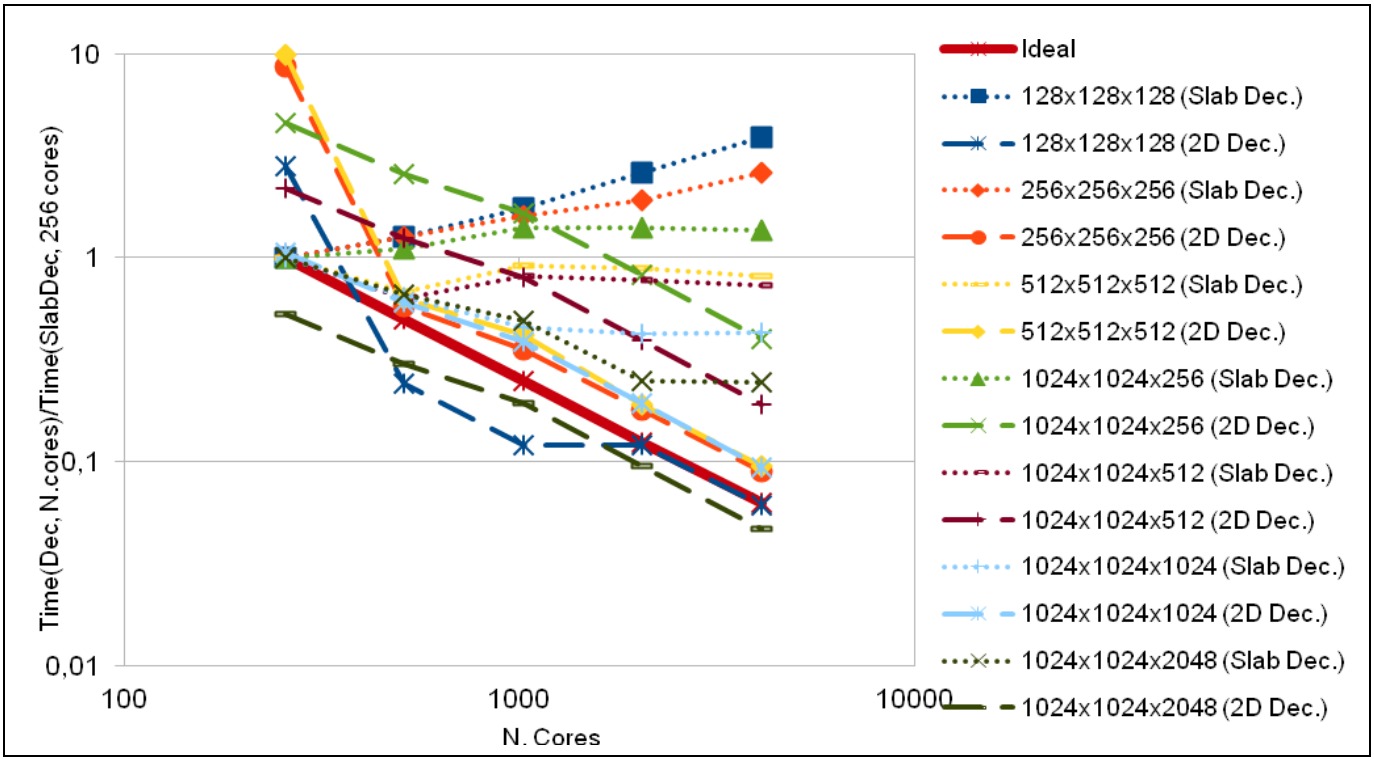

Figure 6: Ratio between the times of execution (see Tab.3) and the corresponding time obtained using standard FFTW on 256 cores. The red and continuous line is the ideal scalability line. Dotted lines are related to standard FFTW efficiency, whereas dashed lines are related to $2D^3$ efficiency.

Sistem & Solver			Number of Cores									
			256		512		1024		2048		4096	
Domain Decomposition	Method	N. of Points	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
		128x128x128	437	1.0	545	1.0	644	1.0	644	1.0	725	1.0
		256x256x256	270	1.0	320	1.0	445	1.0	517	1.0	597	1.0
		512x512x512	45	1.0	172	1.0	200	1.0	271	1.0	382	1.0
		1024x1024x256	65	1.0	169	1.0	255	1.0	356	1.0	355	1.0
		1024x1024x512	60	1.0	74	1.0	250	1.0	311	1.0	389	1.0
		1024x1024x1024	65	1.0	73	1.0	174	1.0	272	1.0	385	1.0
	Estimate	1024x1024x2048	74	1.0	110	1.0	88	1.0	246	1.0	364	1.0
		128x128x128	2252	5.1	2731	5.0	3569	5.5	4098	6.4	4872	6.7
		256x256x256	3137	11.6	4242	13.2	5826	13.2	6884	13.3	8086	13.5
		512x512x512	3346	74.4	7836	45.7	11342	45.7	11639	43.0	14213	37.2
		1024x1024x256	16150	275.0	27737	164.0	37568	147.1	43094	121.1	42621	119.9
		1024x1024x512	21288	352.6	19060	257.4	39868	159.7	37256	120.0	40797	104.7
		1024x1024x1024	16404	251.2	14472	198.8	34047	196.2	39178	144.2	44176	114.7
Slab Decomposition	Measure	1024x1024x2048	22125	297.8	21209	192.2	12809	146.2	50450	205.2	45103	124.0
		128x128x128	1958	4.5	608	1.1	573	0.9	443	0.7	343	0.5
		256x256x256	7903	29.3	1381	4.3	1171	2.6	1063	2.1	998	1.7
		512x512x512	42684	949.1	4741	27.6	2984	14.9	2419	8.9	2092	5.5
		1024x1024x256	239	3.7	32205	190.4	17605	69.0	8077	22.7	5851	16.5
		1024x1024x512	238	3.9	145	2.0	154	0.6	65	0.2	46	0.1
		1024x1024x1024	234	3.6	150	2.1	138	0.8	68	0.3	48	0.1
2D Decomposition	Measure	1024x1024x2048	234	3.1	146	1.3	141	1.6	67	0.3	49	0.1

Table 5: Time needed to create a Plan (ms). We analyze the same cases of Tables.3 and 4, including also the time needed to create a plan using the FFTW_ESTIMATE flag as comparison. In the ratio columns we show the ratio between the time of creation, and the ones obtained using standard Slab Decomposition algorithm and the FFTW_ESTIMATE flag. We have: a red box when Ratio ≥ 100; a yellow box when 10 ≤ Ratio < 100; a green box when 1 ≤ Ratio < 10; and a blue box when Ratio < 1. It is evident that the time cost of a plan creation using 2D3 is comparable with the time needed to create a plan using standard FFTW library and the FFTW_ESTIMATE flag.

Summarizing the results of our comparison activity, we found that the use of the 2D domain Decomposition algorithm is particularly useful when the number of usable cores exceeds the size of the index involved in the parallel domain decomposition. When the size of this index is comparable with the number of usable cores, it may be advantageous using 2D decomposition, since it takes much less time to create the plan.

Starting from these results it is relatively simple to improve the performance of FFTW library enhancing its auto-tuning mechanism. We check if the number of cores exceed the size of the index involved in parallel domain decomposition, and thus change the parallel domain decomposition algorithm used. One simple algorithm that we can implement is reported in Algorithm 2.

```
IF (Np > Nx) then
    IF (Np =< (Nx * Ny) ) then
        USE 2D DECOMPOSITION
    ELSE
        Np = Nx * Ny
        USE 2D DECOMPOSITION
    END IF
ELSE
    USE SLAB DECOMPOSITION
END IF
```

Algorithm 2: The first simple algorithm used to switch from standard slab to 2D domain decomposition

```
IF (Np > Nx) then
    IF (Np = (Nx * Ny) ) then
        USE 2D DECOMPOSITION
    ELSE IF (Np < (Nx * Ny) ) then
        IF ((Nx * Ny) is NOT a whole multiple of Np) then
            FOR (NN < NP and NN ~ NP)
               IF ((Nx*Ny) is a whole multiple of NN) then
                   Np = NN
               END IF
            END FOR
        END IF
        USE 2D DECOMPOSITION
    ELSE
        Np = Nx * Ny
        USE 2D DECOMPOSITION
    END IF
ELSE
    IF (Nx is NOT a whole multiple of Np) then
        FOR (NN < NP and NN ~ NP)
            IF (Nx is a whole multiple of NN) then
               Np = NN
            END IF
        END FOR
        ENDIF
    END IF
    USE SLAB DECOMPOSITION
END IF
```

Algorithm 3: An improved version of *Algorithm 2*, that includes also the main features of *Algorithm 1*.

In order to further improve the performance of this auto-tuning algorithm it is possible to include a modified version of Algorithm 1. The result of this merging operation is shown in Algorithm 3. These algorithms were used to create a library that can significantly improve the performance of standard the FFTW library on massively parallel supercomputers. This library has already been tested on the FERMI cluster, obtaining the best performances possible for the two algorithms.

Conclusions

We have improved the performance of the auto-tuning mechanism already present in FFTW using two different approaches:

- For multidimensional arrays, we changed the number of cores involved in the DFT execution. Indeed, it is already known that the performance of FFTW is severely limited by the size of the index in which the domain decomposition is done. Our approach was to create a FORTRAN tool that changes the number of used cores to the size of the index involved in the domain decomposition, if the number of usable cores exceed the size of this index. This approach has proven to be particularly useful for solving 2D DFT problems.
- For 3D arrays, we change the domain decomposition algorithm from Slab to 2D. This new algorithm gave us performances up to 63 times better than standard algorithm (see Table 4). Since this algorithm scales almost up to N^2 cores on cubic arrays with size of N^3 points (whereas standard FFTW scales as N), if used on massively parallel supercomputers, this new algorithm can give us a nearly ideal scalability of at least up to several thousands of cores (also for relatively small data arrays). We have already implemented this algorithm in a library that switches from standard to 2D domain decomposition allowing us to always achieve the best performance possible for the two algorithms.

As expected, the use of these algorithms (especially Algorithm 3) can greatly improve the performances of the FFTW library on modern massively parallel supercomputers. The 2D Domain Decomposition algorithm could be extended to multi-dimensional systems (higher than 3D systems), and in the next months it will be the subject of further research by CINECA.

Acknowledgements

This work was financially supported by the PRACE-2IP project (10) funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.

The work is achieved using the PRACE Research Infrastructure resources FERMI and PLX at CINECA in Italy.

We thank S. Soner and D. Erwin for providing constructive comments and help in improving the contents of this paper.

References

(1) Cooley, J. W. & Tukey, J., 1965. An algorithm for the machine calculation of complex Fourier Series. Mathematics of Computation, Issue 19, pp. 297-301

(2) M. Frigo & S. G. Johnson, 2013, Available at: http://www.fftw.org/

(3) M. Frigo & S. G. Johnson, "The Design and Implementation of FFTW3", Proceedings of the IEEE 93 (2005) 216

(4) CINECA, s.d. PLX cluster. Available at: http://www.hpc.cineca.it/content/ibm-plx-gpu-user-guide

(5) Intel, s.d. Xeon Westmere. Available at: http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-(12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI)

(6) CINECA, s.d. FERMI cluster. Available at: http://www.hpc.cineca.it/content/fermi-reference-guide

(7) IBM BLUE GENE/Q. Available at http://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/

(8) R. Schultz, 2008, 3D FFT with 2D decomposition, CS project report, Center for molecular Biophysics, Available at: https://cmb.ornl.gov/members/z8g/csproject-report.pdf

(9) N. Li, & S. Laizet, 2010. "2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface.", Cray User Group 2010 Conferences, Edinburgh.

(10) PRACE: Partnership for advanced Computing in Europe. Available at http://www.prace-ri.eu