Bulk Synchronous Parallel implementation of the 2-Dimensional Fast Fourier Transform

Davide Taviani
(d.taviani@students.uu.nl)

19 January 2012

Abstract

This report focus on the parallelization of the 2-Dimensional Fast Fourier Transform using the Bulk Synchronous Parallel programming model. After a brief introduction on the Discrete Fourier Transform and Fast Fourier Transform, two different approaches to its implementation are discussed and their BSP cost analyzed. In addition, some numerical experiment help us evaluate our theoretical analysis and understand when a parallel approach is convenient and to what extent.

Contents

1 Introduction

In the field of Digital Signal Processing, when dealing with periodic functions that represent signals or images, a very important role is played by Fourier Analysis, a branch of Mathematical Analysis that studies the decomposition of functions into their frequency components.

With such a decomposition, we are able to effectively remove desired frequencies (which, for instance, correspond to noise in a image) o enhance some others; its application really are countless and affect, on a daily basis, almost everyone.

To give a formal definition of our framework, following the scheme of [1], let $f : \mathbb{R} \to \mathbb{C}$ be a T-periodic function, i.e. such that $f(t+T) = f(t)\ \forall t \in \mathbb{R}$.

The **Fourier Series** associated with f is

$$\tilde{f}(t) = \sum_{k=-\infty}^{+\infty} c_k e^{\frac{2\pi i k t}{T}} \tag{1}$$

where the Fourier coefficient c_k are given by

$$c_k = \frac{1}{T} \int_0^T f(t) e^{\frac{2\pi i k t}{T}} \mathrm{d}t \tag{2}$$

and i represents the complex number such that $i^2 = -1$.

We can see that in order to compute c_k, we must deal with an integral; in every real world application, however, dealing with the continuous is not possible, and we must necessarily discretize. For this reason, one of the most important concepts in Fourier Analysis is the so-called **Discrete Fourier Transform** (DFT), which is the discrete analogous of the Fourier Transform.

As previously mentioned, our goal is to manipulate signals in some way: to do so, however, we must first be able to extract the required information from the signal, and this is where the DFT comes at play. For a more comprehensive overview of the meaning and the application of the DFT see [1, p. 151], and [2].

In practice, such operator transforms a discrete, complex vector $x \in \mathbb{C}^n$ (which for example represents a sampling of someone's voice) in the vector $y \in \mathbb{C}^n$ in the following way [1, p.101]:

$$y_k = \sum_{j=0}^{n-1} x_j e^{\frac{2\pi i j k}{n}} \qquad 0 \leq k < n \tag{3}$$

Furthermore, often it is convenient to express the DFT algorithm using a matrix notation, introducing the concept of the Fourier matrix[3, 1].

If we consider the n-th root of unity

$$\omega_n = e^{\frac{2\pi i}{n}}$$

the formula (3) can be restated in the following, more common, way:

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} \tag{4}$$

It is fairly easy to see that, even under the assumption that ω_n^m has been precomputed for all required m and stored on a table, the cost of the straightforward computation of the DFT is still $8n^2 - 2n$ flops [1, p. 102], therefore less than desirable for most cases.

For example, using (4) to perform the decoding of 0.74 seconds of music from a CD Audio, thus with a vector length of $n = 32768$, 43 minutes are needed on a 3.3 Mflop/s computer. Such performance is far from impressive, since ideally everything should be done in real-time to listen continuously to music.

To solve this inconvenience, an algorithm for the fast computation of the DFT has been devised (it was discovered in its most basic by Gauss in 1805, too early to benefit from it). The rediscovery by Cooley and Tukey [4] is the most widespread (it is from the 1965, already in the era of digital computing) and goes by the name of **Fast Fourier Transform** (FFT), which has the much reduced cost of $5n \log_2 n$ flops; with this algorithm, the previously mentioned decoding from an audio CD can be done in real time.

In a vast amount of real world applications, the core of the computations are essentially FFTs; for this reason in many cases further improvements in its computation speed are very welcome; this is the main reason for a parallel approach toward the computation of such FFT: doing so, we are in fact effectively able to solve bigger instances in less time.

In particular, our interest is directed toward the parallelization of the FFT using the Bulk Synchronous Parallel programming model proposed by Valiant [5]; for this reason, we closely follow the scheme outlined by Rob H. Bisseling

in [1], where an implementation of the FFT, together with a wide analysis of the costs, is provided.

In the next section we focus on the parallelization of the 2-Dimensional FFT, which can find its application in the vast field of Digital Image Restoration; in particular, we proceed to study two possible extension of the function `bspfft` found in `BSPedupack` [6].

2 Parallel 2-Dimensional Fast Fourier Transform

The 2-Dimensional Discrete Fourier Transform (2D DFT) of an $n_0 \times n1$ matrix X is defined as the $n_0 \times n_1$ matrix Y given by:

$$Y(k_0, k_1) = \sum_{j_0=0}^{n_0-1} \sum_{j_1=0}^{n_1-1} X(j_0, j_1) \omega_{n_0}^{j_0 k_0} \omega_{n_1}^{j_1 k_1} \tag{5}$$

where, similarly as before, ω_n denotes the n-th root of unity.

We can see easily that (5) can be rewritten as

$$Y(k_0, k_1) = \sum_{j_0=0}^{n_0-1} \left(\sum_{j_1=0}^{n_1-1} X(j_0, j_1) \omega_{n_1}^{j_1 k_1} \right) \omega_{n_0}^{j_0 k_0} \tag{6}$$

Since the content inside the brackets is none other than a 1-Dimensional DFT, the 2D DFT is equivalent to a set of DFTs in the row direction of the matrix, followed by DFTs on the column direction.

For this reason, any efficient algorithm for a 1-Dimensional FFT can be adapted to perform the 2D DFT with little effort. As already mentioned, we will use the `bspfft` function provided in [6] to this purpose.

The only different aspect from a 1-Dimensional framework is that now we have to distribute not just a vector, but a whole matrix to p processors: we assume, similarly as the one dimensional case, that both n_0, n_1 are powers of 2. To distribute the matrix X we will examine two different ways:

- $M \times N$ cyclic distribution
- Row cyclic distribution

Also in this framework, in order to be able to work with complex numbers, for each entry we use one row and two columns: instead of storing $x_i \in \mathbb{C}$, we store $Re(x_i)$ and $Im(x_i)$ next to each other, representing them as `doubles` in the C programming language.

2.1 $M \times N$ cyclic distribution

In the $M \times N$ distribution, M denotes the number of processor rows and N the number of processor columns [1, Chapter 2] such that $p = MN$ and both M and N are powers of 2. In this way, when we distribute the data cyclically, the matrix fits perfectly, thus preventing any imbalance. In this way, each processor is denoted with the 2D numbering $P(s,t)$, where $0 \leq s < M$ and $0 \leq t < N$.

Our implementation of the 2D FFT using this distribution is based on the fact that, after we fix s, if we consider $P(s,*)$, we are simply doing a 1D FFT on N processors.

Moreover, for our local matrix, since the distribution is cyclic, we have that any row of the starting matrix is split among the processors in such a way that the local row index is the same. For this reason, from the processor point of view, when we are in the k-th row of our local matrix, we know that we need to communicate with the k-th row of the matrices that belong to the processors in the same row.

In addition, we can further optimize by performing all the computations simultaneously on every row and by "packing" all the data that needs to be sent away (after all, the destination processors are the same, since we deal only with our processor row) and by performing only one actual communication, thus decreasing the total overhead.

Treating each row separately (but doing everything simultaneously), we effectively perform the 1D FFT on the rows; for the columns, the reasoning is quite similar, although there are a couple of necessary differences: first of all, since the local matrix is stored row by row, its transposition is necessary to perform computations column-wise. Secondly, to perform the actual communication between the processors that belong to the the same processor column, we can recycle the code for the previous part by performing a "processor transposition", which consists in the swap of the values of s and t. Doing so, if we "trick" the program to perform a communication between the same processor row, in reality we are working with the same processor column.

After the required computation for the 1D FFTs on the columns has finished, we have to perform again a local transposition in order to obtain a matrix with the same distribution as the input one.

The computational cost of the `bspfft` function for a vector of length n spread among p processors in general is [1]

$$T_{\text{FFT}} = \frac{5n \log_2 n}{p} + 2 \left\lceil \frac{\log_2 p}{\log_2(n/p)} \right\rceil \frac{n}{p} g + \left(2 \left\lceil \frac{\log_2 p}{\log_2(n/p)} \right\rceil + 1 \right) l$$

Our 1D FFT on the rows is pretty similar: in every processor we are performing $\frac{n_0}{M}$ times that computation and that communication, but due to our optimizations, the number of supersteps remains the same. Then we have

$$T_{\text{rows}} = \frac{n_0}{M}\left(\frac{5n_1 \log_2 n_1}{N} + 2\left\lceil \frac{\log_2 N}{\log_2(n_1/N)} \right\rceil \frac{n_1}{N} g\right) + \left(2\left\lceil \frac{\log_2 N}{\log_2(n_1/N)} \right\rceil + 1\right) l \tag{7}$$

Technically, since in the transposition step we are not doing any floating point operation, its cost is to be neglected.

Performing the FFT on the columns yields then the following cost

$$T_{\text{cols}} = \frac{n_1}{N}\left(\frac{5n_0 \log_2 n_0}{M} + 2\left\lceil \frac{\log_2 M}{\log_2(n_0/M)} \right\rceil \frac{n_0}{M} g\right) + \left(2\left\lceil \frac{\log_2 M}{\log_2(n_0/M)} \right\rceil + 1\right) l \tag{8}$$

If we add up (7) and (8) we obtain the general cost of a 2D FFT.

However, since in the 1D FFT, if $1 < p \leq \sqrt{n}$, then only one communication superstep is needed along with two computation supersteps, we have a total cost of

$$\tilde{T}_{\text{FFT1D}} = \frac{5n \log_2 n}{p} + 2\frac{n}{p} g + 3l$$

Similarly, in our 1D FFT on the rows, if $1 < N \leq \sqrt{n_1}$

$$\tilde{T}_{\text{rows}} = \frac{n_0}{M}\frac{5n_1 \log_2 n_1}{N} + 2\frac{n_0}{M}\frac{n_1}{N} g + 3l \tag{9}$$

and on the columns, if $1 < M \leq \sqrt{n_0}$,

$$\tilde{T}_{\text{cols}} = \frac{n_1}{N}\frac{5n_0 \log_2 n_0}{M} + 2\frac{n_1}{N}\frac{n_0}{M} g + 3l \tag{10}$$

which brings the total cost for the 2D FFT of

$$\tilde{T}_{\text{FFT2D}} = 5\frac{n_0 n_1}{MN}\left(\log_2 n_1 + \log_2 n_0\right) + 4\frac{n_0 n_1}{MN} g + 6l \tag{11}$$

2.2 Row cyclic distribution

Technically, the row cyclic distribution is a particular case of the $M \times N$ distribution: we just spread the n_0 rows between the processors, thus it is a $p \times 1$ cyclic distribution.

The implementation of the 2D FFT using this data distribution is much simpler than the previous one: the FFT on the rows is done sequentially and is performed using the `ufft` function (after the necessary bit reversal [1]) which constitutes a part of the function `bspfft`. However, while previously there was apparently no difference between performing the FFT on the rows and on the columns (we just instructed each processor to communicate with a different set of processors) now clearly there is, since the columns are not spread but the rows are.

In order to perform the FFT on the columns using the same code of the rows, there is again the need of a transposition; this time, however, such transposition is not just local, but involves actual communication between the p processors and it is done as follows:

Let us consider processor s and let a be the local $\frac{n_0}{p} \times n_1$ matrix.

For any element a_{ij} of such local matrix, j denotes the global and local column index (after all, columns are not spread) and i denotes the local row index. The relationship between the local row index i and the global row index k is given by

$$k = s \cdot i + p \qquad\qquad i = k \mod p$$

Then, during the transposition, a_{ij} must be sent to the processor that owns the j-th global row, $j \mod p$, and it must be put at the row $\frac{j}{p}$, at the index k, where k is again the global index of row i.

For processor p, the total number of numbers sent (and relation number, in the sense of BSP communication cost) is exactly $\frac{n_0 n_1 (p-1)}{p^2}$. Thus the total communication cost of the transposition (in flops) is

$$T_{\text{transp}} = \frac{n_0 n_1 (p-1)}{p^2} g$$

With that in mind, our algorithm can be summarized in the following steps:

- $\frac{n_0}{p}$ sequential 1D FFTs of length n_1: $\frac{n_0}{p} 5 n_1 \log_2 n_1$ flops
- Transposition: $\frac{n_0 n_1 (p-1)}{p^2} g$ flops

- $\frac{n_1}{p}$ sequential 1D FFTs of length n_0: $\frac{n_1}{p} 5 n_0 \log_2 n_0$ flops
- Transposition: $\frac{n_0 n_1 (p-1)}{p^2} g$ flops

Since we have 2 computations supersteps and 2 communication supersteps, the synchronization cost is $4l$ flops.

The total cost of our 2D FFT is then

$$T_{\text{transpose}} = 5\frac{n_0 n_1}{p}(\log_2 n_1 + \log_2 n_0) + 2\frac{n_0 n_1 (p-1)}{p^2} g + 4l \qquad (12)$$

2.3 Comparison of the efficiency

As we have seen from the previous Sections, the two different approaches toward data distribution lead to a different BSP cost function.

Now, it is natural to wonder whether one data distribution is better than the other or if the best one depends on the structure of the problem, the number of processors, and so on.

Firstly, we can see that, provided that p is still a power of two and $n_0 \geq n_1$, if $n_0 < p$ with the row cyclic distribution we will use only n_0 processors to perform FFT on the rows; with the $M \times N$ distribution, instead, and the right choice of M and N, we can effectively employ all $p = MN$ processors. If $n_1 \geq n_0$ and $n_1 < p$ we can do a similar reasoning, but the inefficient step with the row cyclic distribution (with respect to the number of processors used) now lies in the FFT on the columns.

The main difference in the BSP cost of (11) and (12), provided that $p \leq \sqrt{n_0}$ and $n_0 \geq n_1$, is on the communication part: since this often constitutes the bottle-neck of any parallel system, this difference is very important: keeping in mind that $MN = p$, we have that the communication cost of the 2D FFT with the row cyclic distribution is exactly $\frac{p-1}{2}$ times the one of the $M \times N$ distribution, and since often $p \gg 2$, this is not negligible element.

This reasoning holds only for relatively big instances with respect to the number of processors, since if $p > \sqrt{n_0}$ we cannot use the reduced cost formula (10) for the FFT on the columns with the $M \times N$, but we must use (8). Then, it might be the case that using the row cyclic distribution is more efficient.

3 Numerical Experiments

In order to compare the two chosen data distributions, a number of numerical tests has been run. The machine used was "Huygens", the dutch national supercomputer, a clustered SMP (Symmetric Multiprocessing) IBM pSeries 575 system, with a total of 3456 IBM Power6 cores with clock speed of 4.7 GHz, a total performance of 65 Tflop/s, an internal transfer speed of 160 Gbit/s, 15.75 TB of memory and 700 TB storage capacity.

The performance of the machines measured with the program `bspbench` are summarized in the Table 1, where all times are in flop units and $r \approx 195$ Mflop/s :

p	g	l
1	51.1	554.6
2	55.0	1976.0
4	57.0	4106.9
8	61.9	7476.1
16	59.9	15964.7
32	61.3	32712.1
64	64.3	100689.8
128	135.7	209302.8

Table 1: Parameters of g and l with different number of processors p. With $p = 256$ the benchmark program `bspbench` did not return any parameters for g and l.

Both algorithms have been implemented in C and compiled with the `-O3` flag and the library BSPonMPI [1].

All the code, along with the necessary output, can be found at

`https://github.com/Heliosmaster/bspfft2d`

Since a natural application of the 2D DFT is image processing, the values of n_0 and n_1 have been set toward realistic parameters. In particular, we are performing DFT on matrices 128×128, 256×256 and 512×512. Furthermore, in order to try and look the performance with a high number of processors (up to 256), also the sizes 1024×1024 and 2048×2048 have been tested (which correspond to roughly 1 Megapixel and 4 Megapixel, the normal performance of digital cameras from approximately 5-10 years ago), along with the case of $n_0 \neq n_1$, with 1024×2048 and 2048×1024.

[1] `https://github.com/BSPWorldwide/BSPonMPI`

With fixed p, regarding the $M \times N$ distribution, we run the algorithms for different values of M, N, but, since the cost is the same if M and N are swapped, only the case $M \geq N$ is shown.

The tests consist in a 2D FFT followed by an inverse 2D FFT, in order to obtain the starting matrix. The times shown below refer to these two operations, they are expressed in seconds and were obtained with the function `bsp_time`.

p	T_{transp}	M	N	$T_{M\times N}$
1	0.011475	1	1	0.001936
2	0.011205	2	1	0.002351
4	0.006117	2	2	0.002003
8	0.003562	4	2	0.001692
16	0.002793	4	4	0.002254
		8	2	0.002527
32	0.003673	8	4	0.005417
		16	2	0.007677
64	0.020745	8	8	0.030606
		16	4	0.037794
128	0.058380	16	8	0.107684
		32	4	0.125310
256		16	16	0.307677
		32	8	0.302438
		64	4	0.417350

Table 2: 128×128

p	T_{transp}	M	N	$T_{M\times N}$
1	0.058001	1	1	0.009144
2	0.012944	2	1	0.008162
4	0.044485	2	2	0.005957
8	0.023550	4	2	0.004379
16	0.007572	4	4	0.004288
		8	2	0.005036
32	0.006981	8	4	0.007834
		16	2	0.011015
64	0.023713	8	8	0.033500
		16	4	0.033725
128	0.056572	16	8	0.091102
		32	4	0.111660
256		16	16	0.205015
		32	8	0.257110
		64	4	0.303008

Table 3: 256×256

p	T_{transp}	M	N	$T_{M\times N}$
1	0.243473	1	1	0.060243
2	0.178164	2	1	0.034558
4	0.094468	2	2	0.022320
8	0.050251	4	2	0.013944
16	0.027930	4	4	0.010625
		8	2	0.011825
32	0.018270	8	4	0.015527
		16	2	0.024390
64	0.026790	8	8	0.037611
		16	4	0.041072
128	0.059766	16	8	0.092810
		32	4	0.127867
256		16	16	0.214214
		32	8	0.280381
		64	4	0.252669

Table 4: 512×512

p	T_{transp}	M	N	$T_{M\times N}$
1	1.066994	1	1	0.311058
2	0.745790	2	1	0.183062
4	0.380951	2	2	0.103594
8	0.201670	4	2	0.054790
16	0.108293	4	4	0.034840
		8	2	0.037436
32	0.063436	8	4	0.037611
		16	2	0.047321
64	0.064602	8	8	0.062703
		16	4	0.063691
128	0.073268	16	8	0.111057
		32	4	0.134782
256		16	16	0.237892
		32	8	0.228929
		64	4	0.333084

Table 5: 1024×1024

p	T_{transp}	M	N	$T_{M\times N}$
1	4.632781	1	1	1.369651
2	3.214293	2	1	1.000243
4	1.595267	2	2	0.489667
8	0.844813	4	2	0.288404
16	0.442577	4	4	0.165189
		8	2	0.153864
32	0.246716	8	4	0.103001
		16	2	0.117332
64	0.261612	8	8	0.118652
		16	4	0.110963
128	0.172449	16	8	0.179095
		32	4	0.189339
256		16	16	0.260139
		32	8	0.311035
		64	4	0.508711

Table 6: 2048×2048

p	T_{transp}	M	N	$T_{M\times N}$
1	2.221304	1	1	0.648421
2	1.515910	2	1	0.392416
4	0.786731	2	2	0.234820
8	0.418917	4	2	0.133531
16	0.216868	4	4	0.069914
		8	2	0.072064
32	0.126588	8	4	0.062563
		16	2	0.073711
64	0.109697	8	8	0.094562
		16	4	0.084194
128	0.119880	16	8	0.144182
		32	4	0.156932
256		16	16	0.239072
		32	8	0.248411
		64	4	0.410399

Table 7: 1024×2048

p	T_{transp}	M	N	$T_{M\times N}$
1	2.223130	1	1	0.658685
2	1.506204	2	1	0.386310
4	0.785664	2	2	0.234376
8	0.416552	4	2	0.138940
16	0.217014	4	4	0.068770
		8	2	0.069898
32	0.124131	8	4	0.059659
		16	2	0.067814
64	0.110285	8	8	0.087100
		16	4	0.083930
128	0.115097	16	8	0.138621
		32	4	0.147189
256	0.160212	16	16	0.242359
		32	8	0.245621
		64	4	0.413940

Table 8: 2048×1024

3.1 Remarks

Our implementation of the algorithm using the row cyclic distribution aborts when $p > n_0$, as noticeable in Table 2. Moreover, with $p = 256$ the program fails to return an output with any matrix but the 2048×1024 one. This may be due to implementation issues and probably needs further investigation.

The most important result is that the numerical experiments seem to agree with our theoretical analysis of Section 2.3: when p is sufficiently small compared to n_0 (we skip talking about n_1 because the small matrix considered were square) the $M \times N$ data distributions provides better performances, while when the number of processor is much more similar to the size of the matrix considered, the other data distribution seems to be more efficient.

Regarding the $M \times N$ distribution, from the tables we see that is much preferable to have an almost "square" processor grid (i.e. $M = N$ or as close as possible), in order to have more work balance in both phases (rows and columns), consistently with our expectations.

Another important result from the numerical experiments is that we are able to understand the underlying structure of the parallel machine: when the number of processor used is small enough we are working with very "close" (by meaning of connection) CPUs. If this number is somewhat high ($p \geq 32$) a sharp fall in the performance is observed, as expected.

The following Figures show the speedup plot with the various matrix sizes:

for the $M \times N$ data distribution, the best result for every p has been selected for comparison. Only up to $p = 32$ has been considered, due to the noticeable fall in performance for a higher number of processors.

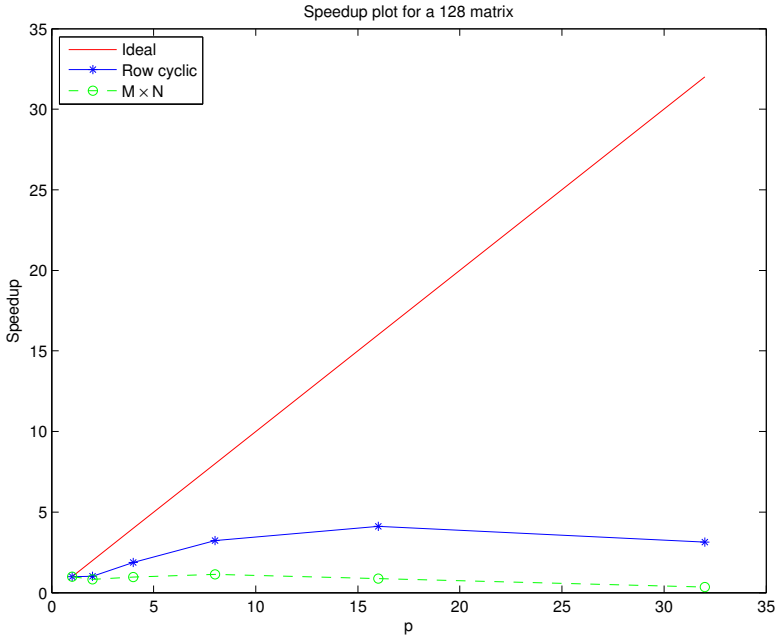

Figure 1: Speedup plot for a matrix 128×128.

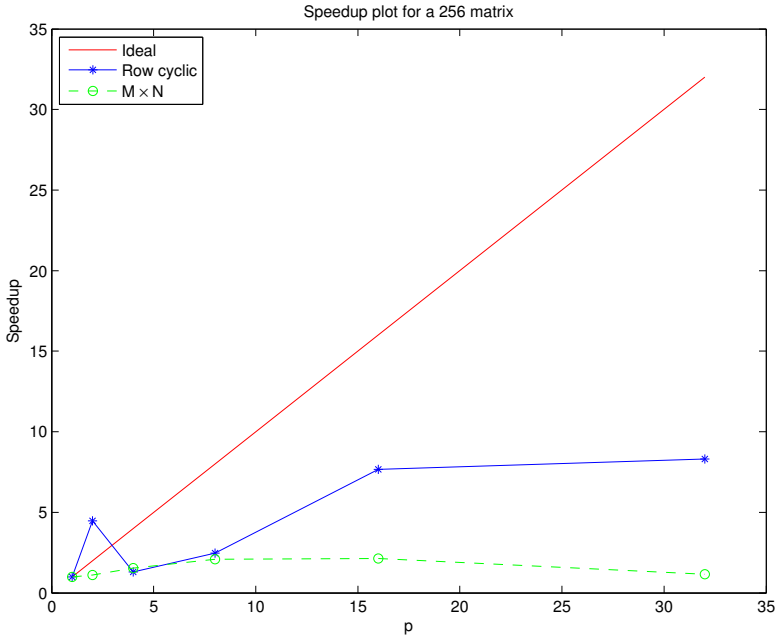

Figure 2: Speedup plot for a matrix 256×256.

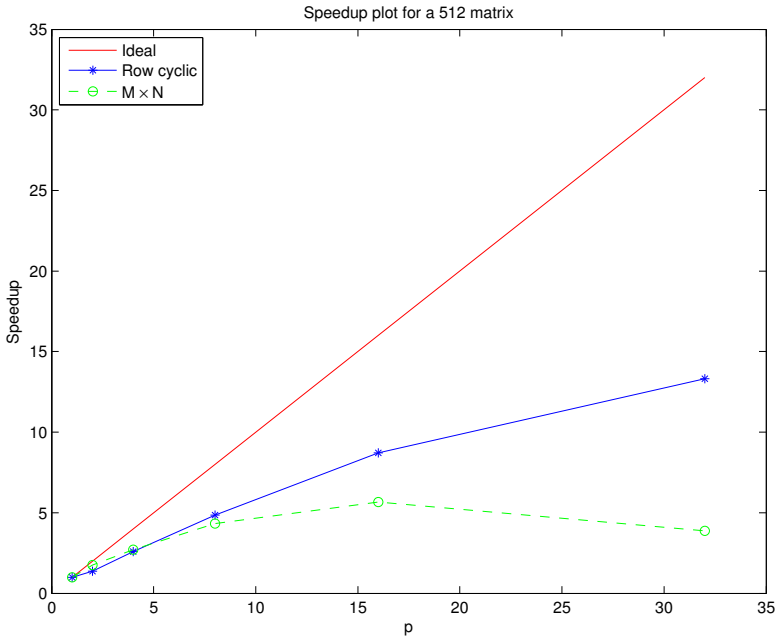

Figure 3: Speedup plot for a matrix 512×512.

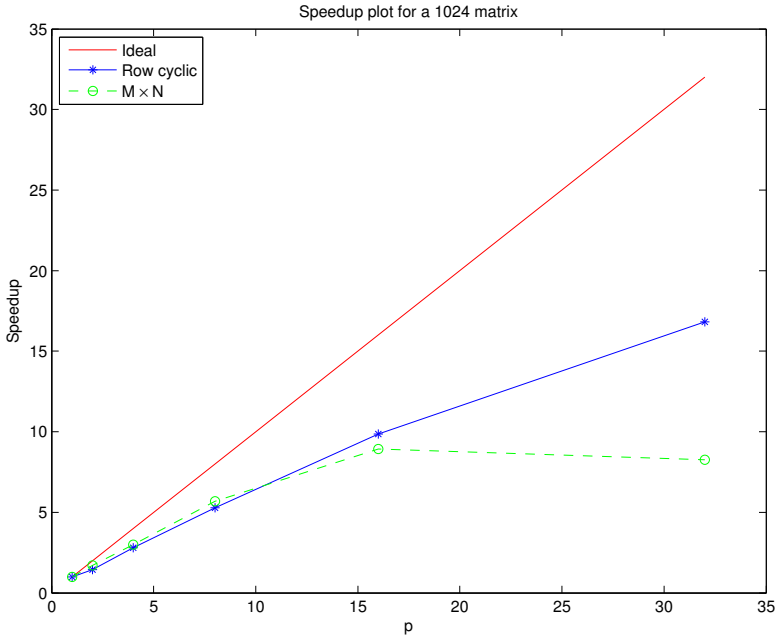

Figure 4: Speedup plot for a matrix 1024×1024.

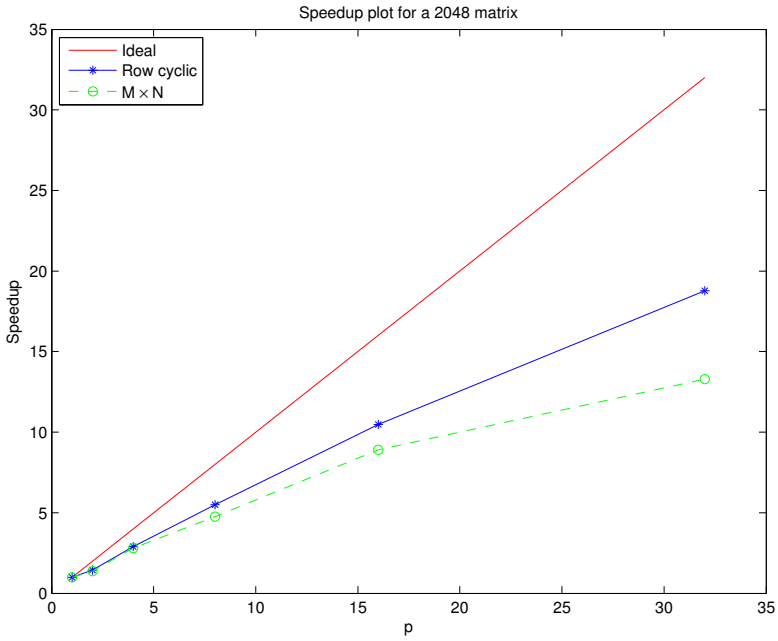

Figure 5: Speedup plot for a matrix 2048×2048.

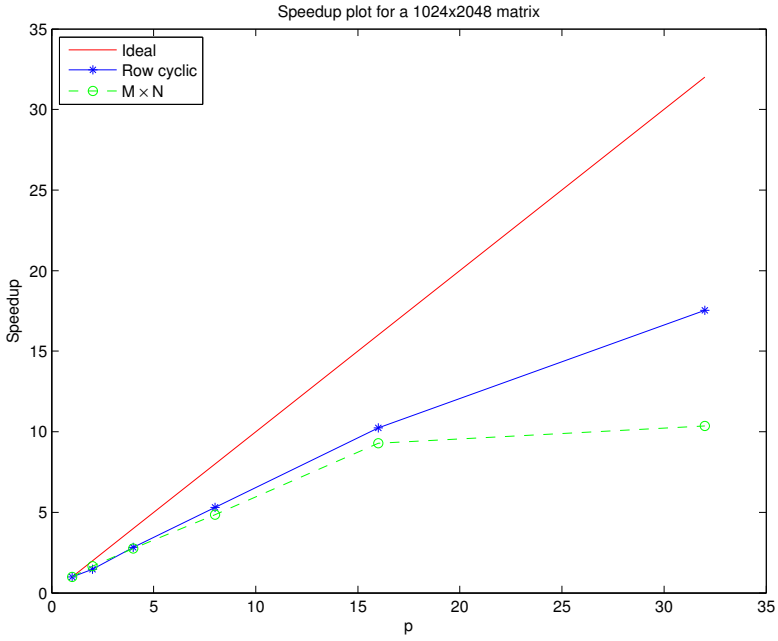

Figure 6: Speedup plot for a matrix 1024×2048.

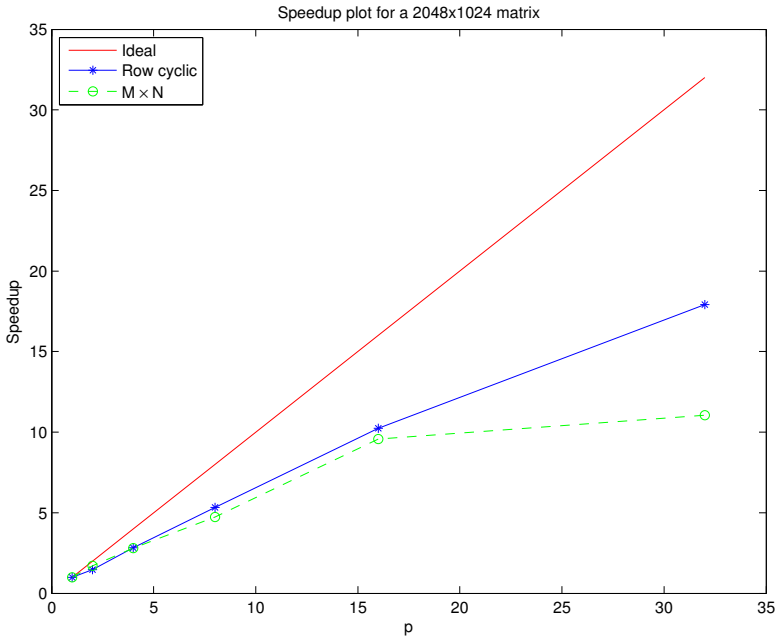

Figure 7: Speedup plot for a matrix 2048×1024.

4 Conclusions and further developments

From the numerical tests and the speedup plots it appears evident that a parallel approach toward the computation of the 2-Dimensional FFT is convenient: we have been able to speed up its computation considerably (for example, with the 2048×2048 matrix, in one case we improved from 2.2 to 0.1 seconds and in the other from 0.6 to 0.05 seconds).

We also have seen that, for relatively small sized problems (128×128 and 256×256, for instance), probably it's better to avoid using too many processor, leaving them free for other programs to run, where their impact would be much greater.

In literature there exist many other implementation of the FFT that would probably lead to better performance: one of them is the famous "Fast Fourier Transform in the West" [2] which provides both an implementation for 1D and 2D FFT. In that library there is already a parallel version of these implementations, aimed at SMP machines (like the one we used) for Cilk (a superset of C invented by one of the FFTW's creators aimed at easy parallelization and automatic load balancing between processors on shared memory machines) and MPI. For this reason, a comparison in both the implementation and the performance of our version of the 2D FFT and the one in FFTW could lead to even more insights about the exploitability of the parallelization for this particular operation.

References

[1] Rob H. Bisseling. *Parallel Scientific Computation: A structured approach using BSP and MPI*. Oxford: Oxford University Press, 2004.

[2] H.C. Andrews and B. R. Hunt. *Digital image restoration*. Prentice Hall, 1977.

[3] Davide Taviani. *Total Variation image denoising from Poisson data: Split Bregman and Alternating Extragradient methods*. 2011.

[4] J.W. Cooley and J.W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19 (1965), pp. 297–301.

[5] L.G. Valiant. "Bulk-synchronous parallel computers". In: *Parallel Processing and Artificial Intelligence* (1989).

[6] Rob H. Bisseling. *BSPedupack*. URL: `http://www.staff.science.uu.nl/~bisse101/Software/software.html`.

[2] `http://www.fftw.org/`