Benchmarking of the linear solver PASTIX for integration in LESCAPE

Lucas Lestandi

Supervisor: Héloïse Beaugendre

March 9, 2015

Contents

1 The test case **2**

2 Fast Fourier Transform using le Intel MKL library : benchmarking **5**
2.1 Context 5
2.2 Benchmark 5

3 Pastix Benchmarking **6**
3.1 Presentation of Pastix 6
3.2 Accuracy comparison between the methods 6
3.3 Benchmarking 7
3.3.1 Time efficiency of Pastix 7
3.3.2 Multi-solve 8
3.4 Parallelism setting influence 8
3.4.1 Thread number scaling on one node 8
3.4.2 MPI instances influence 10
3.4.3 Multiprocesses and threads combination 10

4 Iterative solvers **11**
4.1 "Smart" direct solvers 11
4.2 Pastix implemented direct solvers 12
4.3 Pastix mixed direct-iterative approach 12

5 Results and beyond **13**
5.1 Comparison of FFT, direct and iterative solvers 13
5.2 3-D problem 13
5.2.1 Positionning of the problem 13
5.2.2 Results 14

Introduction

In order to upgrade the LESCAPE code to the third dimension, it is necessary to be able to implement mixed boundary conditions. This is not available in the current solver (Fast Fourier Transform -FFT- based on the mkl package). Consequently a new solver shall be implemented. Pastix has been elected.

Pastix is a direct linear solver developed at Inria Bordeaux. It is based on LU factorization with substantial preprocessing on the matrix. It is also fully compatible with both MPI and openMP allowing high efficiency on modern supercomputers.

Even though fft is unbeatable for few iterations its advantage might melt compared to simple Up-Down algorithm.

In this report we focus on benchmarking the two solvers and exploring Pastix options to compute efficiently on several nodes.

1 The test case

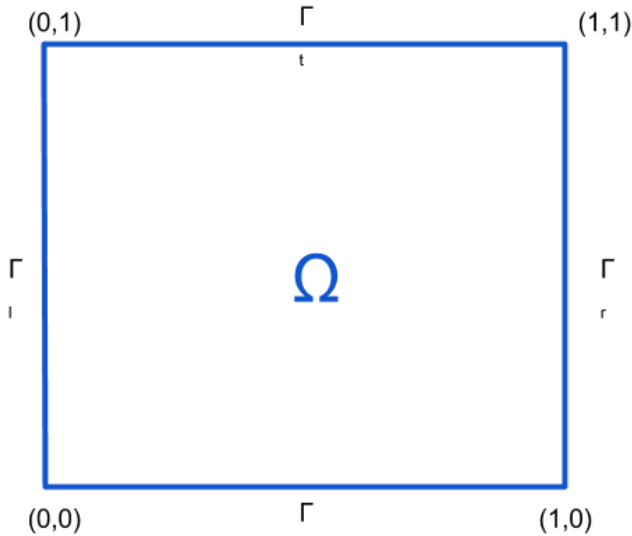

Figure 1: Description of the computing domain

In order to compare both solvers, we have to implement a simple problem for both of them. In order to compute the relative accuracy of each one a stationary problem P which has an analytic solution.

$$(P) \begin{cases} \Delta u = -f(x,y) & \text{on} \quad \Omega \\ u(x,y) = q & \text{on} \quad \Gamma_l \bigcup \Gamma_r \\ \partial_y u = 2\pi \sin(2\pi x) & \text{on} \quad \Gamma_t \bigcup \Gamma_b \end{cases} \tag{P}$$

Where $\Omega = [0,1] \times [0,1]$ and $f(x,y) = 8\pi^2 \sin(2\pi x) \sin(2\pi y)$. The domain is described in figure 1.

One can easily ensure that this equation possess a solution which is :

$$u(x,y) = \sin(2\pi x) \sin(2\pi y) \tag{1}$$

Then it will be easy to compute the error committed during the computation.

This problem is then discretized on a Cartesian grid containing $Nx \times Ny$ points. The Laplacian operation is discretized by the usual five points stencil.

$$-\Delta u_{ij} = \left(\frac{2}{\delta x^2} + \frac{2}{\delta y^2}\right) u_{ij} - \frac{u_{i+1j} + u_{i-1j}}{\delta x^2} - \frac{u_{ij+1} + u_{ij-1}}{\delta y^2} \tag{2}$$

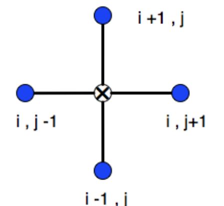

Figure 2: Five point stencil

In order to discretize the boundary condition of the domain, three methods have been applied depending on the meshes described in Figure 3,4 and 5. As expected these matrix are clearly sparse $nnz \leqslant 5n$ where $A \in \mathcal{M}_n$. Then a sparse storage will be used.

- In method one, the boundary conditions are implemented implicitly since no value is computed on the boundary. The following discretisization is used.

$$\begin{cases} U_{ij} = q \quad \text{for_} i = 0\text{_for_} i = Ny + 1 \\ \frac{u_{i0} - u_{i1}}{\delta y} = -2\pi \sin(2\pi x) \\ \frac{u_{iNy+1} - u_{iNy}}{\delta y} = 2\pi \sin(2\pi x) \end{cases} \tag{3}$$

As one can see in figure 3 the matrix is symmetric, this property is very interesting since the storage is cheaper and solver for symmetric are cheaper than the others. For direct solvers one can use a Crout decomposition or a Conjugate Gradient if an iterative solver is needed.

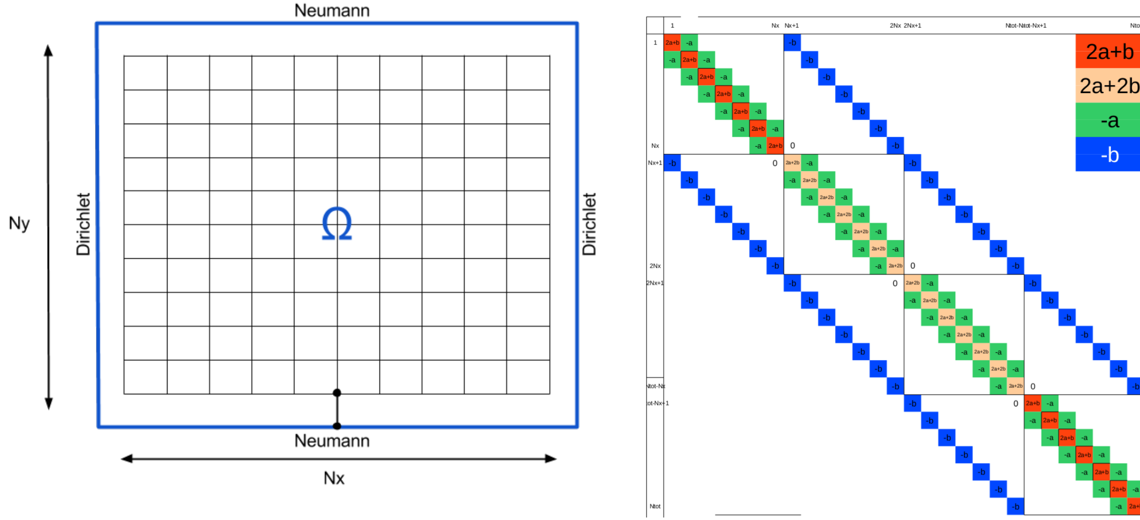

Figure 3: Method 1
Mesh without points on the boundary & Shape of the associated matrix

- In method 2, the boundary conditions are implemented explicitly there are points on the boundary. The following second order discretization is used for the Neumann boundary condition.

$$\begin{cases} U_{ij} = q \quad \text{for_} i = 1\text{_for_} i = Ny \\ \frac{u_{i0} - u_{i2}}{\delta y} = -2\pi \sin(2\pi x) \\ \frac{u_{iNy+1} - u_{iNy-1}}{\delta y} = 2\pi \sin(2\pi x) \end{cases} \tag{4}$$

The matrix thus obtained is the most complicated of the three methods since its graph is not symmetric for rigorous sparse storage. For the graph to be symetric (which is required by Pastix as explained later), one need to add some zeros (in red on figure 4) to the sparse storage. Nonetheless, the matrix is not symetric and adapted methods shall be used. Pastix enables the use of LU decomposition for direct solve and GMRES or BiCGstab for iterative solve. These methods are more expensive than the one presented before. One might question himself about the relative advantage of both discretisation.

- In method 3, the boundary conditions are implemented explicitly for Neumann and implicitly for Dirichlet. We use the following expression to discretize.

$$\begin{cases} U_{ij} = q \quad \text{for_} i = 0\text{_for_} i = Ny + 1 \\ \frac{u_{i0} - u_{i2}}{\delta y} = -2\pi \sin(2\pi x) \\ \frac{u_{iNy+1} - u_{iNy-1}}{\delta y} = 2\pi \sin(2\pi x) \end{cases} \tag{5}$$

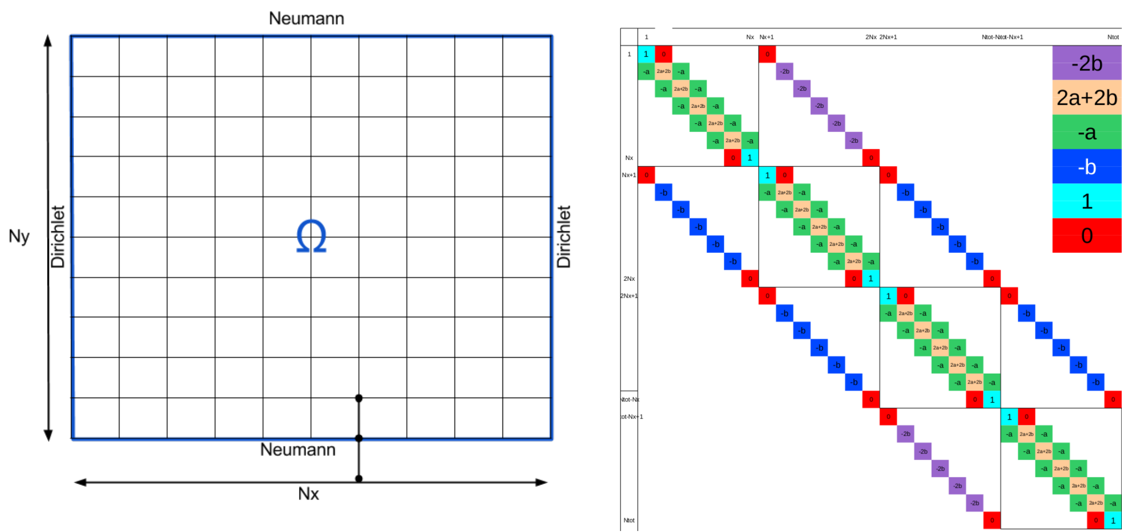

Figure 4: Method 2
Mesh with points on the boundary & Shape of the corresponding matrix

This discretization tries to get the advantages of method 1 and 2. The Dirichlet implicit implementation removes the irregularity (and may improve the conditioning of the matrix) induced by solving on Dirichlet B.C. in method 2. Also it preserves the second order approximation of the Neumann BC. The graph of the matrix is symetric which means that no zero has to be stored but the matrix is not symmetric.

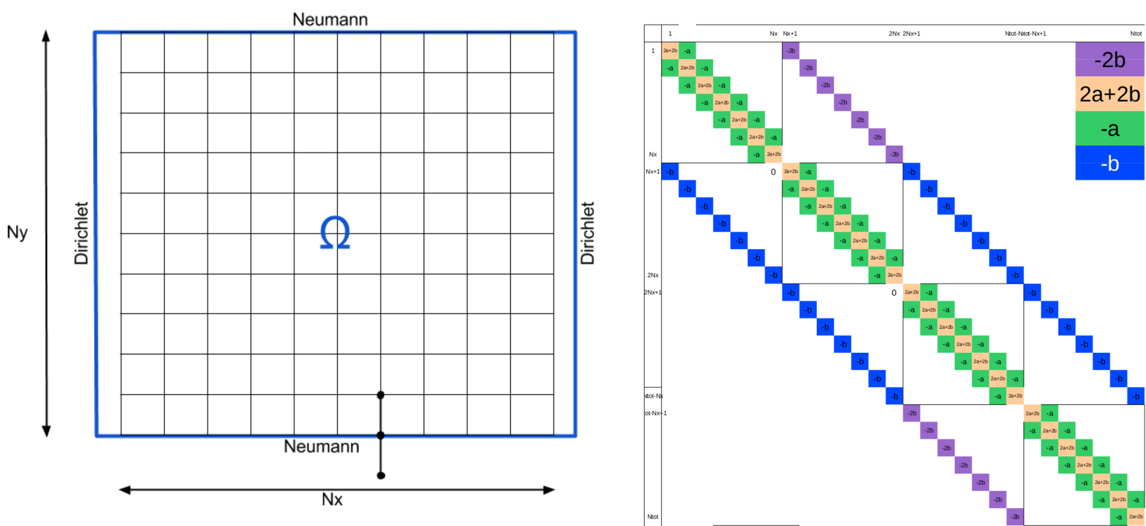

Figure 5: Method 3
Mesh with mixed boundary interpretation & Shape of the corresponding matrix

2 Fast Fourier Transform using le Intel MKL library : benchmarking

2.1 Context

The Fast Fourier Transform (FFT) is one of the most efficient way to solve a partial differential equation and it can be applied to discrete problems. It relies on properties of the Fourier Transform, indeed it can be solved very quickly if the space size N is a power of two (slower for other prime decomposition). It is also very efficient in term of computing time (derivative are solve with simple complex multiplication) with a complexity of $O(N\log(N))$.

The current implementation is based on the Intel Math Kernel Library.

2.2 Benchmark

Since the method is very fast, one iteration takes less than one second on one note of Plafrim, tests need to last longer to prevent parasite to matter. Then it is chosen to run the tests on a minimum of 100 iterations.

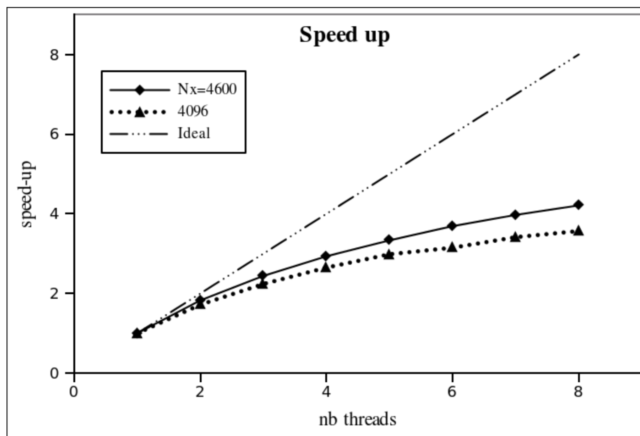

Figure 6: FFT of the mkl library speed-up, 1 Fourmi node

	8 thread		1 threads	
Nx	100 it	1 it	100 it	1 it
4096	18.47	0.18	65.9	0.66
4100	21.6	0.22	88	0.88
4129	73.2	0.73	433	4.33
4600	26.6	0.27	112	1.12

Table 1: FFT execution figures for 100 iteration and the average duration of 1 iteration depending on Nx

The speed-up graph presented in figure 6 show a relatively good scalebility, however one can see that increasing further the number of core would not be very efficient as shows "4600-mirabelle" in figure 6.

In table 1, one can see that there is roughly a factor 4 between execution time of 1 and 8 threads. Each iteration is almost instantaneous, then solving the Poisson problem might not be the slowest part of LESCAPE. As exepected for a fft we observe large differences in efficiency depending on wether the prime factorization of Nx releases small primes as in $4096 = 2^{12}$ or large primes as 4129 is prime. This feature of the fft is not generally a big issue, nonetheless it must be taken into account.

The best performance for **Nx=4600** is reached on one node of mirabelle, with **12 threads** and the same procedure, one iteration takes **0.206s**.

As a conclusion, the FFT is a very efficient method to solve a Poisson problem, its only limitations are the integration of mixed boundary condition in 3D and in the case of the mkl implementation that in does not feature MPI library and must be run onto a single node. This might limit the scalability to bigger problems for memory reasons.

3 Pastix Benchmarking

3.1 Presentation of Pastix

Pastix is a direct linear solver based on the symbolic factorization. Further information is available in the Pastix Manual. Pastix enables natively both openMP and MPI and handle all the required distribution of data and communications. The user only has to provide the number of threads he requires in the program and the number of MPI processes when calling the programm.

Mainly, in order to solve a linear equation Pastix goes through the following steps. Each one can be done separatly.

1. **Ordering** using Scotch. The best quality is obtain using a single process. Speed can be increased (linearly) if several MPI processes are opened. This is generally the longest step. The ordering can be done once and for all as a preprocessing step and stored as long as the matrix remains the same. The efficiency of the whole process (fill-in, dependencies between the calculations) is decided in this step.

2. **Symbolic factorization**, computes the structure of the factorized matrix. This step is cheap.

3. **Distributing and scheduling**.

4. **Numerical Factorization** using LU,LL^T or LDL^T algorithms depending on the shape of the matrix. Is the most time consuming step after the ordering step.

5. **Solve** Finally solves the matrix with up-down algorithm, then is really fast.

6. **Refinement** Due to bad conditioning, bad precision can occur, then a few iteration of an iterative solveur (BiCGStab,GMRES,CG) can improve the overall accuracy with a small cost. Moreover these solvers can also be used alone as block iterative solvers.

3.2 Accuracy comparison between the methods

In order to have a good view of the situation, first a method comparison benchmark is performed. The three methods are compared on the following criteria: time for ordering, time to perform numerical factorization and time to solve explicitly the problem. The relative errors are also compared for different space discretization number Nx.

First, it is observe that the error behaves the same way for each method converging to a very close value. The order of the method is 2 as shown in figure 7. This is the order of discretization of the Laplacian in space, then the result is satisfying concerning numerical accuracy.

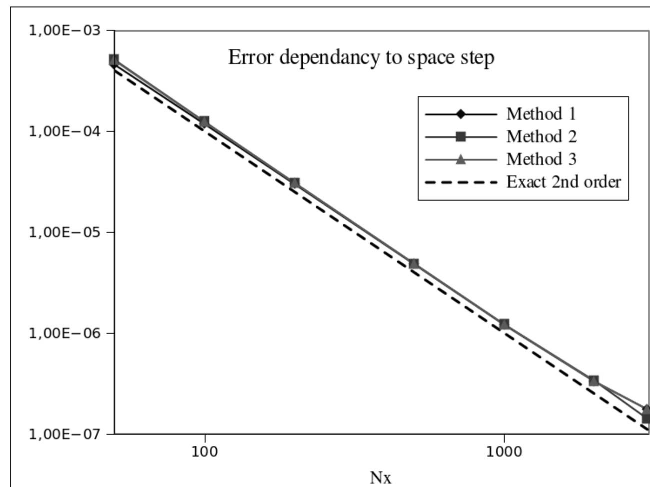

Figure 7: Compared error of the 3 methods

For the implementation to be the most efficient, only time efficiency shall be considered to compare these methods since no clear advantage emerges from the error analysis.

3.3 Benchmarking

3.3.1 Time efficiency of Pastix

In this subsection, the results where obtained on one node of plafrim with **1 MPI process** and **8 threads**. Other parallelism combination will be investigated later.

The time consumption of each part is investigated. For a simple solve of the discrete system $AX = b$ the numbering step (done by SCOTCH) represents around **90%** of the computing time. Luckily it is possible to do this task in a preconditioning step and save its result. The ordering remains the same as long as the matrix does not change. The ordering duration is a linear function of the size of the matrix. A typical time for ordering (and symbolic factorization) is **8 minutes** for $Nx = Ny = 4600$. It is then compulsory only to make one iteration of this step even if it can be done faster several a parallel version of Scotch. Nonetheless, reading the saved file is not free and takes a couple second for large meshes.

Then the scheduling takes place, its execution time also depend on the size of the matrix but is negligeable (1 or 2% of the execution time) compared to ordering and numerical factorization.

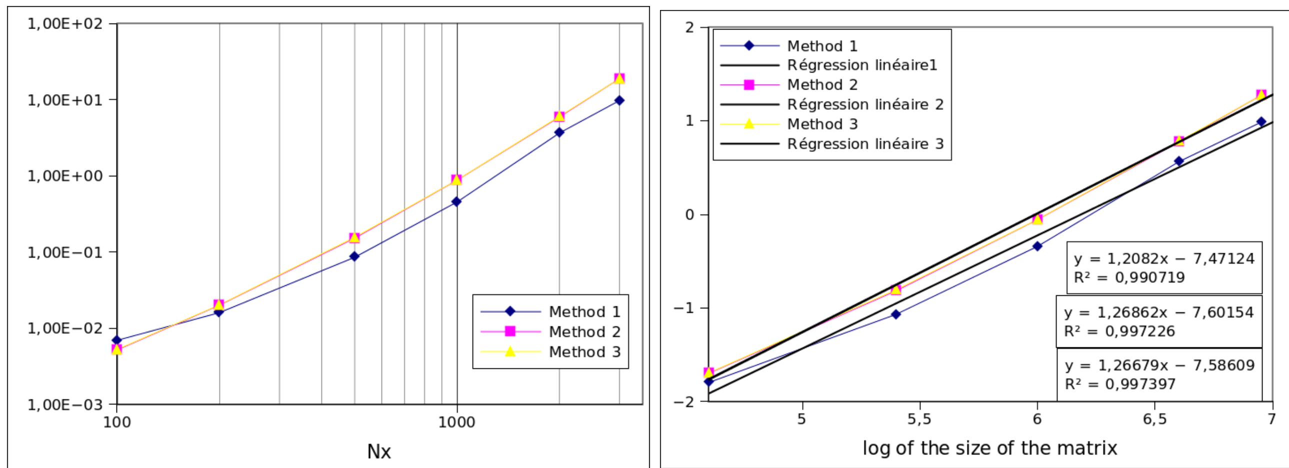

Figure 8: Time consumption of numerical factorisation on the left and correlation to matrix size on the right

The second most time consuming step is the numerical factorization, representing an **average of 6%** of the computational time. Once again, luckily, in this problem as in the Lescape code the matrix that has to be solved remain the same if there are several iterations. This must be done at least one time in the run since this data is not stored as default. The time of factorization is in $O(n^1.2)$ with a good correlation coefficient as shown in figure 8.

One interesting fact, is that method 1 uses a symmetric matrix and consequently factorization is done with a Cholesky algorithm which is supposed to be more efficient than a LU algorithm. This is in fact what is observed in this experiment. **The average factorization time is 44% shorter for LL^T decomposition.** Thus one should consider use symmetric matrix as in method 1.

The final and most important step is the actual solve step consisting in a up-down algorithm. It is really fast, less than 0.4% of a single execution. This is the part that must be compared to the FFT. In fact runtimes are too small to be able to be relied upon for analysis, thereby the effect mention in the previous paragraph is less important for the solve step. This lead to the next subsection where iterated efficiency of Pastix is analyzed.

3.3.2 Multi-solve

According to Pierre Ramet, the good way to do solve several $AX = b_i$ for A a constant matrix is simply to put a loop around the solve step of pastix and update the rhs at each passing in the loop (without changing the nrhs value).

According to the results showed in the previous section, it is pointless to investigate the ordering time and we consider that the ordering and symbolic factorization are know and thus negligible during the experiment. The numerical factorization only happens in the first step, then the first step will only be considered when looking at numerical factorization effect in this section.

Nx	t_{100}	t_{facto}	t_{solve}
1000	35.4	2.5	$7.7 \cdot 10^{-2}$
	32.9	0.6	$6.2 \cdot 10^{-2}$
2000	87	2.1	$2.6 \cdot 10^{-1}$
	98	4.3	$2.6 \cdot 10^{-1}$
3000	189	6.68	$5.4 \cdot 10^{-1}$
	215	22.4	$6.0 \cdot 10^{-1}$
4000	321	15.5	1.06
	365	30.6	1.10
4600	424	24.7	1.45
	510	83.8	1.44
5000	494	28.9	1.6
	555	58.0	1.58

Table 2: Observed time for 100 solve of (P), ordering known,1 node, 12 threads.
Grey lines for method 1 and white lines for method 2, red boxes are for absurd values.

Except from absurd values, is is observed the expected pattern for t_{solve}, the ratio between the two methods is 2. However for numerous solves, the influence of this step decreases. Concerning the solve step duration, no clear difference appears between the two method as expected since the same "up-down" algorithm is used. Even for large meshes (25.10^6 points) the solve time remains small, roughly one second. Though bigger than the FFT, it is in the same range knowing that it can be used on several nodes to improve absolute speed.

Also it is not presented in table 2, around 5% of the solve iterations are significantly longer (up to 3 folds the usual duration) for no apparent reasons.

3.4 Parallelism setting influence

It is already known that Pastix is not as efficient as the FFT on one node but one might wonder what would happen if there where more or less threads, or what combination between threads and mpi processes performs the best. This is the topic of this section.

3.4.1 Thread number scaling on one node

First, the influence of the number of thread is investigated using several iterations with Nx=2000 (for speed reasons). It is observed that minor variations (0.1s) in the solve step can notably alter the quality of this evaluation ,up to 50% as shown in figure 9 for instance for 5 threads. That is the reason why several iterations are used in this experiment, only to keep the most relevant information. However it was not always possible to obtain reliable results for each

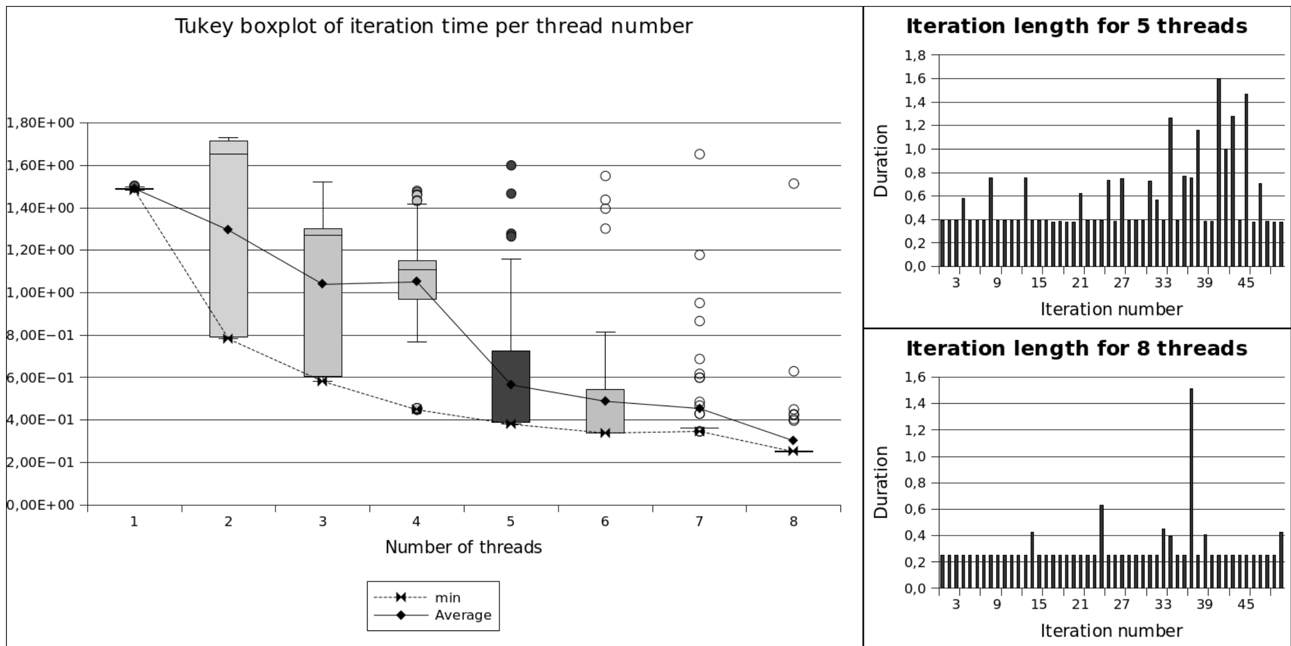

Figure 9: Variability on 1 iteration length, Nx=2000, 1 process

number of threads. Figure 9 displays a Tukey boxplot of the length of iteration in function of the number of thread. It clearly shows that for 2 to 5 threads, variability is high and the minimum length is substantially different from the average. This means that often the solve duration is far from its best value. On the other hand, for 1 thread the duration is always the same, meaning that variation occurs when several threads are involved. Finally, when the number of thread is close to the number of cores on the node, the variability seems to diminish. This is a motivation to use the as much threads as there are cores on a node to ensure the best efficiency.

The speed-up and efficiency graphs presented in figure 10 show that scaling is mostly satisfying, thus validating the use of 1 thread per core. Results slightly differ from the minimum length to the average length but the conclusion remains the same.

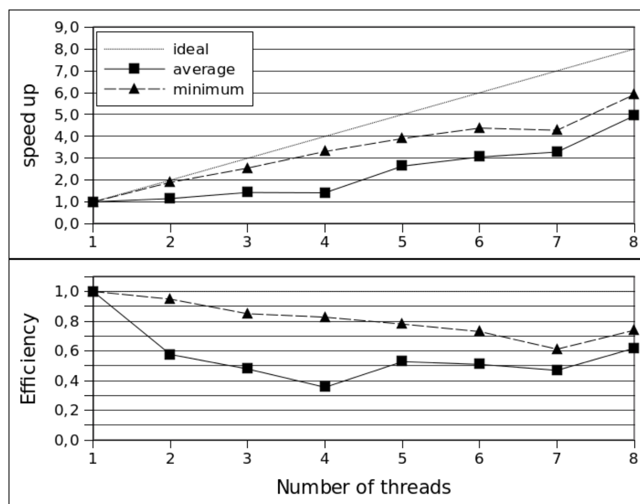

Figure 10: Speed up on 1 iteration length, Nx=2000, 1 process

The drop observed in the average speed up is clearly produced by the variability that was described earlier.

The numerical factorization was not discussed so far, this is due to the poor quality of

the results. Nonetheless we observe that the fastest factorization occurred for 8 threads and is roughly 7 time faster than for one threads. This is a positive results since the numerical factorization is a major source of calculation.

3.4.2 MPI instances influence

The use of several MPI processes is studied in this subsection, the same operating conditions as in the previous subsection are used.

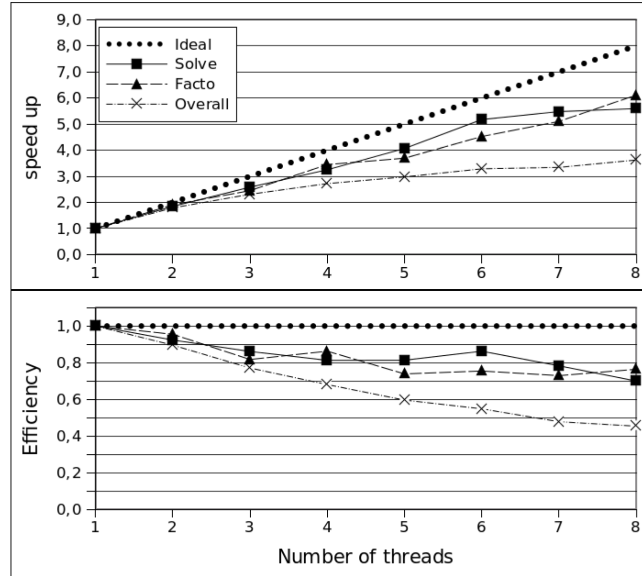

Figure 11: Speed up on 1 iteration length, Nx=2000, 1 thread per process

This experiment shows different properties than thread number variation, if fact standard deviation for solve length is near 0 meaning that all the durations are nearly identical for a given number of MPI processes. In figure 11 one can see that solve and factorization scalability is good with an efficiency of approximately 0.7 for 8 processes. However, the overall speed-up is significantly poorer, the most convincing explanation is that the program enveloping pastix call is virtually sequential which shrinks the overall scalability.

3.4.3 Multiprocesses and threads combination

Even though MPI speed up graph is smoother, the solve scalability is really close between threads and processes, espacially in the case of 1 per core. However, the fastest solve occurs for 8 threads at 0.25s which is better than 0.27s for 8 MPI processes. This section is an attemps to find the best combination between MPI and threads.

It was shown that there is no point to use a number of thread between 2 and 6 in section 3.4.1 then the only possibility is to use 8 threads since no efficient intermediate position is available. Using the `-pernode` i.e. one MPI process per node, is crucial as figure 12 displays. However,in this small experiment the number of processes used does not improve significantly the solve duration. Further experiment shall be realized with a whole program made for parrallel execution.

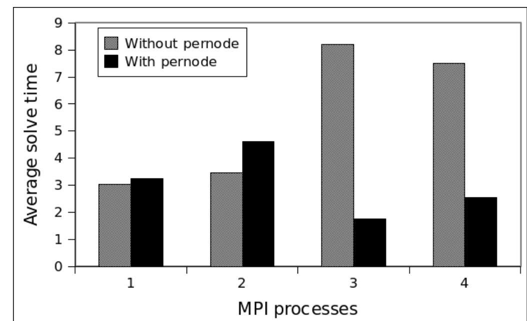

Figure 12: Influence of -pernode on 1 iteration length for several MPI processes, Nx=2000

4 Iterative solvers

The main problem of iterative function is that the number of iteration can be as high as the size of the matrix represent at each step a maximum of $O(n^2)$ operations. Even if the number of iteration is a lot smaller than the size of the matrix it becomes mechanically big as n increases causing the method to loose efficiency when ther is a time step. On the contrary direct solvers are really fast once the first iteration is done.

This section is a basic attempt to show that this assertion is true for a Poisson problem solved on a Cartesian grid.

4.1 "Smart" direct solvers

Since there is no clear advantage in term of precision between the tree methods when Nx is large enough, method one is chosen for its matrix is symmetric and can be solved easily with a Conjugate Gradient (CG). The test code only involves mpi implementation so far but gives sufficient information concerning the reliability of these kind of solvers.

The implementation proposed here is based on the Parallel Computing project from Enseirb-Matmeca with only a few modification to solve the same test problem as in the previous sections. There is no storage of the matrix, only its coefficient. This is not an optimised solver. It can handle any kind of BC as long as the matrix is kept symmetric. The vectors are solved by part on each processor depending on the load function.

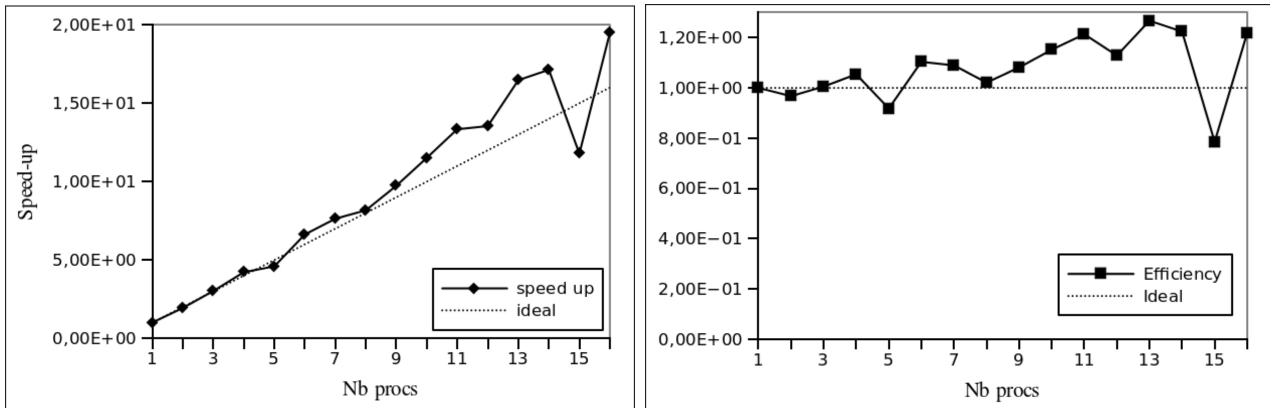

Figure 13: Speed-up and efficiency curve of an adapted CG solver for (P), on 2 nodes, $Nx = Ny = 1000$

The speed-up of this program and the efficiency curves are very satisfying and one can assume that the program has a good scalability. Figure 13 shows these curves for 2 nodes on Plafrim. These test were done on a Cartesian grid of 1000 points in each direction. The matrix is building is included in the execution time. Nonetheless, the performance of this method appears to be inferior to Pastix. It can also be seen that the accuracy decreases with the number of processes used to compute. For the same resources, even a factorization pastix (0.8s) is faster than this CG-based implementation (12.9s) as one can see in table 3.

As a conclusion, in regards of integration in time depending LESCAPE, CG-based solvers appears to be irrelevant.

Method	Nx	Nb Processes	exec time	error (L^2)
1	1000	8	12.9	2.7E-5
1	1000	16	5.4	5.4E-5

Table 3: Extract of the profiling tests on CG solver

4.2 Pastix implemented direct solvers

The refinement step of pastix can be used as an iterative solver using the bloc matrices obtained during the previous step. As in the previous section, the results presented in table 4 are not satisfying for integration in LESCAPE since again, ordering is not taken into account in the results displayed here. The iterative solver timing is roughly the whole program timing then it is the only one discussed.

Method	Nx	Nb threads	algorithm	internal iterations	execution time
1	1000	8	CG	1455	137 s
2	1000	8	BiCGStab	2423	450 s

Table 4: Results for Pastix integrated iterative solvers for 1 solve, 1 Fourmi node,refinement critirion $\varepsilon = 10^{-8}$

This solver is clearly ineffective in this case, the time required is 100 folds bigger than mkl or Pastix, mainly because there are too many internal iterations for a single solve due to slow converge. In fact, the residual (r) reduces extremely slowly until 1427 iterations for method 1 and even goes up sometime. At iteration 1427, $r \simeq 0.1$ an then only 20 iteration are required to converge. As consequence one might think that, if this "threshold" is reached faster then an iterative method might be worth considering.

4.3 Pastix mixed direct-iterative approach

Pastix also allows to use the direct solver as a preconditioner for iterative methods. Unless it enables very fast convergences, this method will display the same flaws as basic iterative solvers. In this section, some experiments are presented to evaluate the potential of this approach.

There are mainly 2 parameters that influence the efficiency of this preconditionning method: the amalgamation level (α) and the level of fill (k) authorized during the incomplete factorization. In order to quickly obtain a good view of the best tuning, an extensive experiment was conducted for small meshes ($Nx \leq 500$). For $k \in [0, 7]$ and $\alpha \in [5, 40]$ the quickest solve occurs at $k =$ and $\alpha =$ with 9.9 seconds to perform refinement. The number of iteration is contained to low levels but requires long computation. Unfortunately, even for $Nx = 500$ the iterative solver (refinement step alone) last around 10s which is a 100 time greater than usual pastix solve time.

As a conclusion, incomplete factorization techniques as preconditioning in Pastix is dismissed.

5 Results and beyond

5.1 Comparison of FFT, direct and iterative solvers

Table 5 presents a general view of the work presented in this report. It confirms that the FFT originally implemented in Lescape is very efficient and was the best choice in the series of method that were studied. However since this solver cannot handle the required boundary condition in 3D, it dismissed for the 3D version of LESCAPE. The three iterative solvers at the end of table 5 are clearly inefficient. Then the best option is to use Pastix in its normal implementation, with a preprocessed ordering and a single numerical factorization for as many time step (solve in the test case) as required. In this condition, a poisson like problem requires 1.5 s to be solved for a given time step for Nx=Ny=4600. This is roughly 5 times longer than the FFT but this loss of performance is acceptable.

3D	Nx	Nb procs	Nb threads	solver	time num fact	time solve
0	4600	1	8	FFT mkl		0.27 s
1	4600	1	8	pastix	24.7s	1.45 s
1	4600	8	1	pastix	147s	4.7 s
1	1000	8	1	parallel CG		12.9s
1	1000	8	1	pastix CG		137s
1	1000	8	1	ILU & CG		73.7s

Table 5: Overview of the efficiency of each solver

5.2 3-D problem

5.2.1 Positionning of the problem

The Three Dimensions problem is a simple extention of the 2D problem used above. Now $\Omega = [0,1]^3$ and there are 6 faces that can have different BC that are presented in figure 14 of Appendix.

Then we introduce the following test case. The problem can be defined for both Dirichlet or Neumann boundary conditions on $\Gamma_f \bigcup \Gamma_b$.

$$\begin{cases} -\Delta u = f(x,y,z) & \text{on} \quad \Omega \\ u(x,y,z) = q & \text{on} \quad \Gamma_l \bigcup \Gamma_r \\ \partial_z u = 2\pi \sin(2\pi x)\sin(2\pi y) & \text{on} \quad \Gamma_u \bigcup \Gamma_d \\ \partial_y u = 2\pi \sin(2\pi x)\sin(2\pi z) \text{ or } u(x,y,z) = q & \text{on} \quad \Gamma_f \bigcup \Gamma_b \end{cases} \tag{P3D}$$

Where $f(x,y,z) = 12\pi^2 \sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$ and $u = \sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$ is the exact solution of P3D.

Then we have the following discretization of the gradient in 3D $\forall (i,k,k) \in [2, Nx-1] \times [2, Ny-1] \times [2, Nz-1]$:

$$-\Delta u_{ijk} = \left(\frac{2}{\delta x^2} + \frac{2}{\delta y^2} + \frac{2}{\delta z^2}\right) u_{ijk} - \frac{u_{i+1jk} + u_{i-1jk}}{\delta x^2} - \frac{u_{ij+1k} + u_{ij-1k}}{\delta y^2} - \frac{u_{ijk+1} + u_{ijk-1}}{\delta z^2} \tag{7}$$

The boundary conditions are handled exactly the same way than in 2D. For method 2, the shape of the matrix is described in figure (15).

$$
\begin{cases}
\text{Let}(x, y, z) \in \Omega \\
\Gamma_l = [x, 0, z] \\
\Gamma_r = [x, 1, z] \\
\Gamma_u = [x, y, 1] \\
\Gamma_d = [x, y, 0] \\
\Gamma_b = [x, y, 0] \\
\Gamma_l = [x, y, 1]
\end{cases} \tag{6}
$$

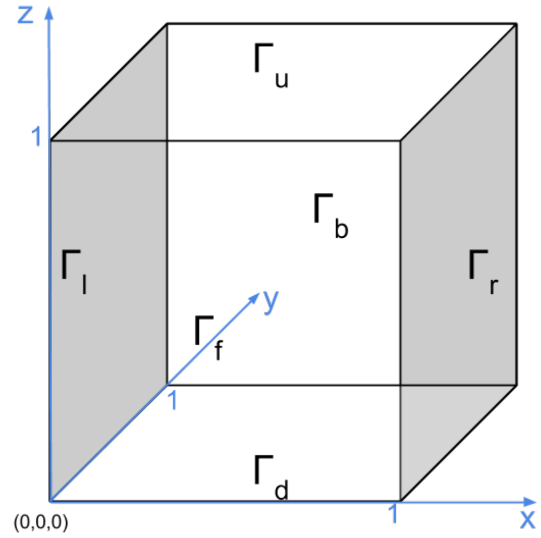

Figure 14: Ω for 3D problem and definition of faces

5.2.2 Results

Conclusion

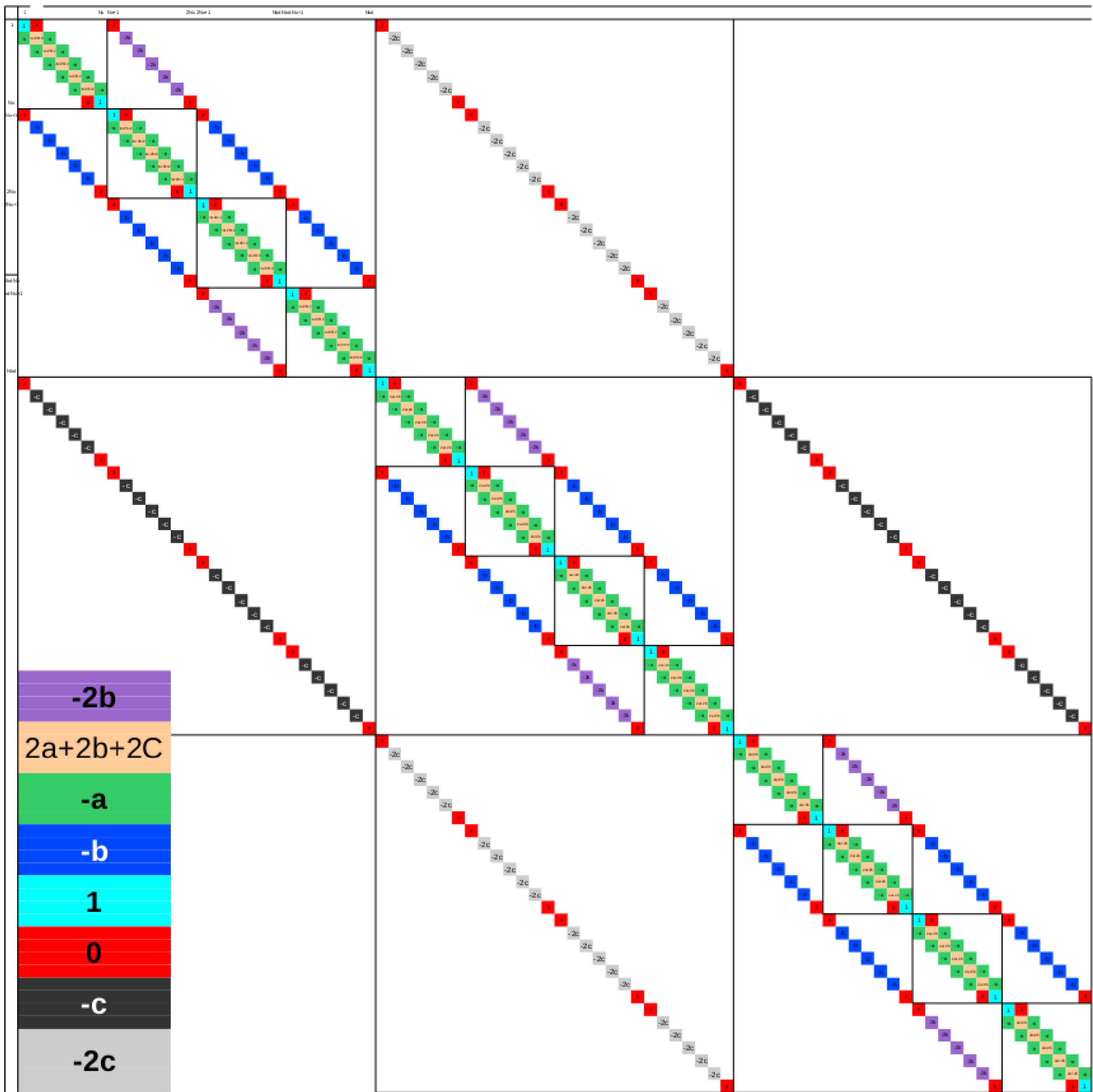

Figure 15: Shape of the matrix for P3D - 3D problem