You Robin *23/04/2017*
Houlier Nicolas *Norwegian University of Science and Technology*

TMA 4280: Supercomputing

Project 2

**** The Poisson Problem ****

Introduction :

This project deals with the parallelization of the **Fast Diagonalization method.**

The problem here is to approximate the solution u to the two-dimensional Poisson problem with homogeneous Dirichlet boundary conditions.

To do this we will approximate u by getting values on a grid, sized with h = 1/n where n will be the size of the problem.

Mathematics give us a way to solve the problem. The programs that have been implemented only follow this recipe.

After implementing this solution, the most important thing will be to analyse the speed and the efficiency of our program. We will implement it using MPI technology alone and also an hybrid version based on the coalition between MPI and openMP.

This work will be launched on a supercalculator, named Vilje :

https://www.hpc.ntnu.no/display/hpc/About+Vilje

here are the main caracteristics of Vilje :

- Manufacturer: SGI
- 19.5 Racks (1404 nodes)
- Number of cores are 22464
- 467 Teraflop/s theoretical peak performance
- 396.70 Teraflop/s LINPACK rating
- Interconnect: Mellanox FDR infiniband, Enhanced Hypercube Topology

Linux System , Scheduler PBS , compiler GNU C .

1404 nodes

2 eight_core processors per node

Node Type : Intel Xeon E5-2670

Processor Speed : 2,6GHz / 16 cores per nodes / L3 Cache → 20 Mb , 8 cores

Memory : 2 Gb per core , 32 Gib per Node , type : DDR3 1600 ↔ infiniband interconnect / FDR

Question 1.

Write a program to solve the Poisson problem with P processes using the algorithm described. You can choose f = 1 for the initial development.

As mentionned before, three programs have been implemented :

the poisson.c file looks like what have been sent in the initial provided documents such as CmakeList.txt , poisson-f90 or fst.f files.

To know how to run it, some ReadMe files have been added on the Git depository.

These programs are located in TMA4280 LABS / Project2 / Programs

Question 2.

Run your parrallel program on Lille. Obtain detailed timing results for different combinations of n = 2**k and P,t. In particular, demonstrate that your program functions correctly for selected values of P in the range 1 < P < 36.

We connect ourselvers on Vilje using :

ssh robinay@training.hpc.ntnu.no

and my password : *********

We also connect ourselves to vilje graphically to submit our work

we send to vilje our job with : *qsub job.sh*

we can check the state of our work : *qstat*

we look at the results : *cat ***.o****

We applied it with f = 5pi²sin(pix)sin(2piy) as mentionned in the Appendix B, we should expect a value of umax equal to 1, then we are able to deduct the error for the maximum value of u on [1,1], emax.

The walltime corresponds to the time required to execute the program on Vilje.

To prove that these tests have been runned, all the .o files with these results will be contained in a special folder on Github.

Here is the signification of the titles of the files :

nXpYtZ.oN → n = X , P = Y , t = Z , N is the work numerous on training.hpc.ntnu.no

We run our program for N in [1-16384] , we take for example P = 8 and t = 4 :

Test Serie n°1

N= ?	walltime	umax	error
1	0,02	0	0
2	0,02	0	0
4	0,02	0	0
8	0,02	0	0
16	0,08	1,01	0,01
32	0,08	1	3,00E-003
64	0,08	1	6,00E-004
128	0,11	1	2,00E-004
256	0,12	1	4,00E-005
512	0,33	1	1,00E-005
1024	0,34	1	3,00E-006
2048	0,76	1	6,70E-007
4096	2,35	1	2,00E-007
8192	8,38	1	4,00E-008
16384	33,62	1	1,00E-008

In the file : *n16386p8t4.o62686* we have shown that if n is not a power of 2 (here 16386) then the program raise an exception : ***« Problem size is not a power of 2 . Aborting »***

This is a verification test .

If we won't mention the number of threads in the mpiexec command we also have a problem

This is a parameters verification test.

Now we fix a problem size of 1024. We simulate it for P in range (1,35) , n=1024 & t = 4

Test Serie n°2

P = ?	walltime	error
1	0,47	3,00E-006
2	0,38	3,00E-006
4	0,24	3,00E-006
8	0,36	3,00E-006
16	0,58	3,00E-006
32	1,57	3,00E-006
64	5,68	3,00E-006

We find out that all the results are the same for various number of processes, for a given f function, which is the same as in the subject.

The error is equal to 2,67 e – 6 .

What is important here is the walltime.

These two test series have been launched on two cores not to false the real value of time execution. At least two cores are needeed to simulate the interraction between processes.

We can see that after 4 processes the execution time becomes more and more long because of communication between processes, exchange of information, which is more important than the calculations to compute solution u.

Note 1 : the file p3.o try to simulate it with P = 3 but a condition in a program refuses this possibility. P has to be a power of two.

This part of the code is tested.

Note 2 : I deleted it when I run the simulations but usually the files .o contains the matrix of values u_{ij} . So we can verify that our solution is correct and that the error decreases in $O(h^2)$ where h = 1/n is the grid scale.

This print is a verification test, We used it with f=1 at the beginning to check that our matrix was correctly represented.

Now we are going to prove, mathematically than our programs stop :

→ First of all when we launched it on Vilje, on the training queue, a timeout is set equal to 15s to make the program crash if it cannot stop. So in any case we will not monopolize the supercalculator.

Now let's analyse our hybrid code which as complexity of $O(n^2\log(n))$ as shown in the course.

The program is launched on a certain number of processes, threads, for a certain size n.

All the processes execute the main program.

This one includes no while conditions, and the functions he is using neither. Only boolean tests and for conditions are written.

The MPI executions raise exceptions when a problem comes.

The algorithm implemented follows the mathematical model, so the proof of the convergence and the accuracy is done in the course and resume at the beginning of the subject.

The convergence has been proved using an analytical solution.

Another test is to print the number of rows per process, m, to check that are repartition is correct and that all the rows are taken into account.

Finally, even if it's not so important many local test , with printf and simple examples have been done during the implementation of the source code but they have been deleted to make the code more readable.

Question 3 .

Run your program with n = 2**14 and pt=36. Does the hybrid model works better, worse or equivalent to the pure distributed memory model ? Explain .

With P = 36 and t = 1 : Pt = 36 and it correspond to the pure distributed sharing model. We obtain :

Time elapsed = 40,967s and the error was : 7 ,0 e-7

Now we modify the number of process and threads : pt = 36

Test Serie n°3

P	T	Walltime	Error
2	18	37,66	1,00E-008
3	12	45,5	< 1e-8
4	9	33,6	1,00E-008
6	6	33,56	<1e-8

These .o files are contained in the folder : Test_Serie_3

In all the cases the hybrid model works better, the error is smaller and time elapsed is smaller. (The case 3-12 is a little bit strange). The communication between processes is too important for the pure distributed model compare to calculations.

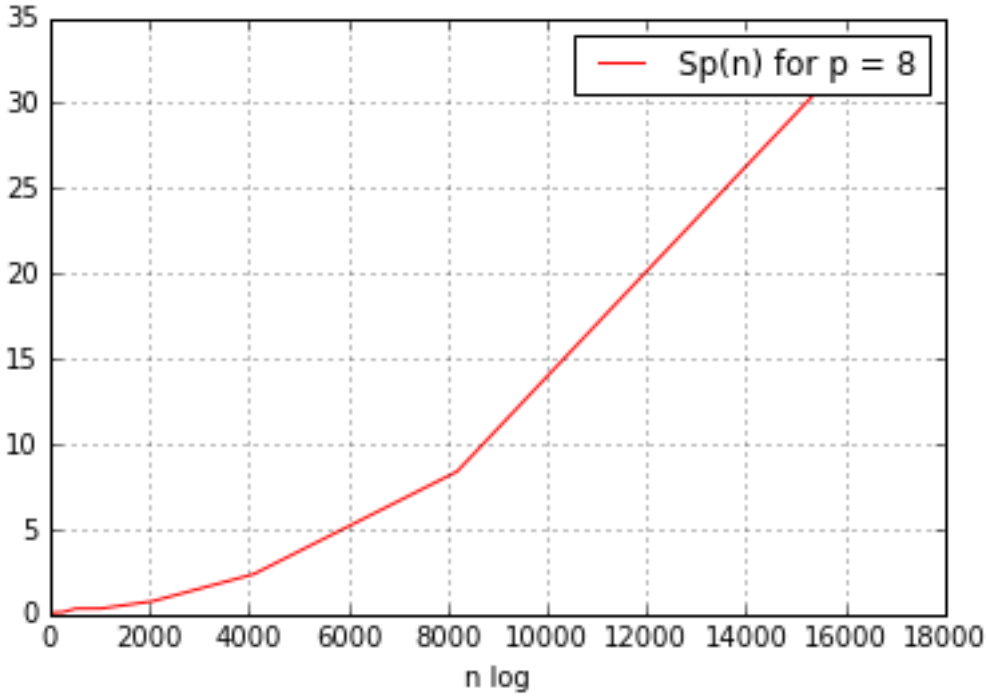
Question 4.

Report the speedup Sp and the efficiency $\mu = Sp/p$ for different values of n and p.

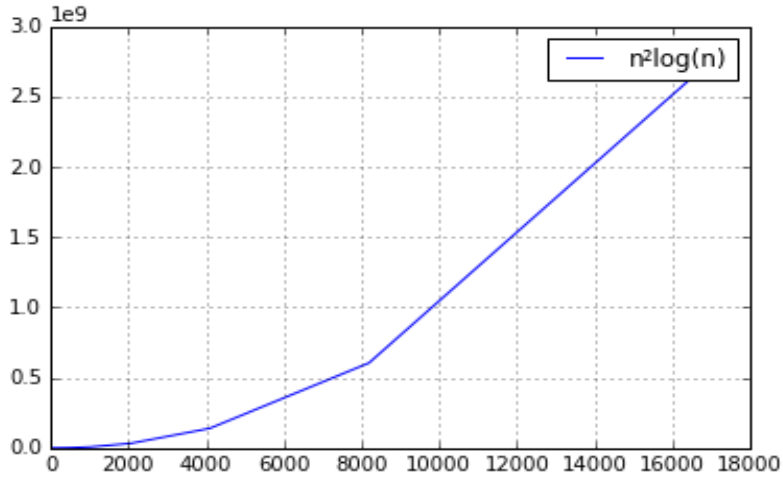
How do your timing result scale with the problem size n^2 for a fixed number of processors ? Is it as expected ? Do you see an improved speedup if you increase the problem size ?

To plot these curbs I will implement a Python program using matplotlib. This is more precise than Excel for example. You'll find the code in Annexe A.

I will use the first three Test Series . First plot : P = 8 , evolution of Sp as a function of n :



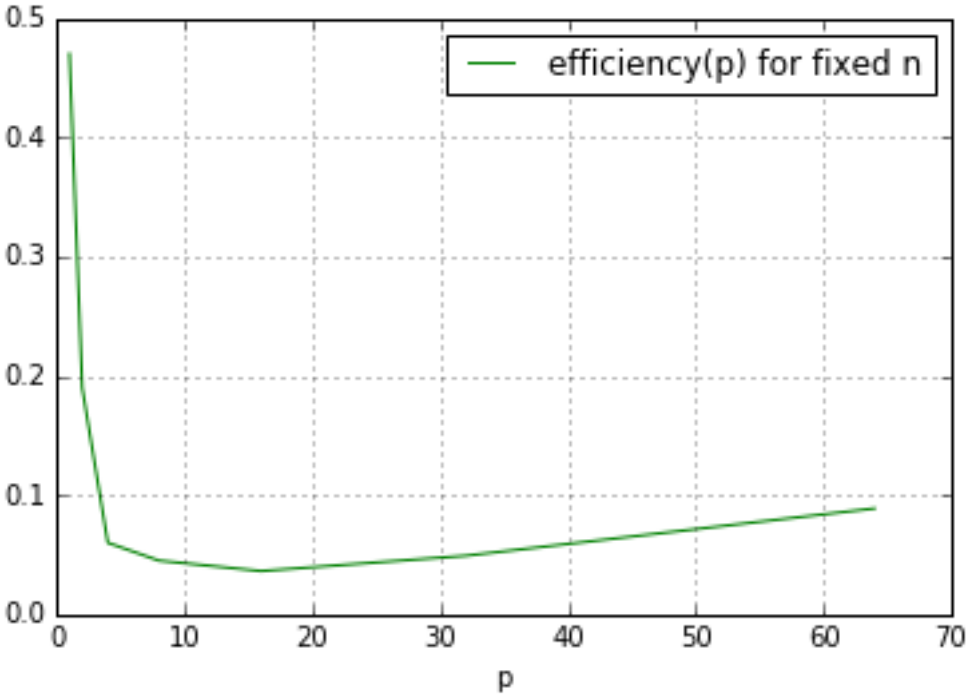
We want to compare it to the predicted complexity which is $O(n^2\log(n))$, to do that we plot the look of $n^2\log(n)$ with the same x-axis than in the first plot using logarithmic scale :



We obtain exactly the same appearance which shows that the complexity of the program is $n^2\log(n)$.

Now we will use the results of the of the second Test Serie to plot the efficiency $\mu = Sp/p$ for a given n.

A second python program gives us this curb :



This curb looks a little bit strange because we should expect something of this type :

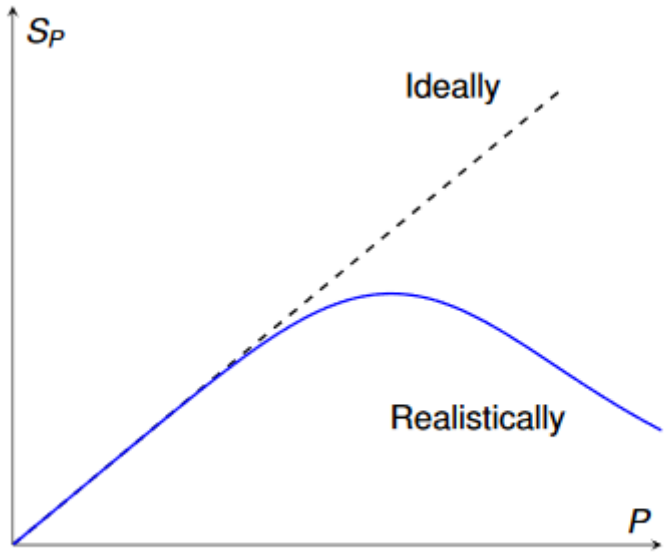
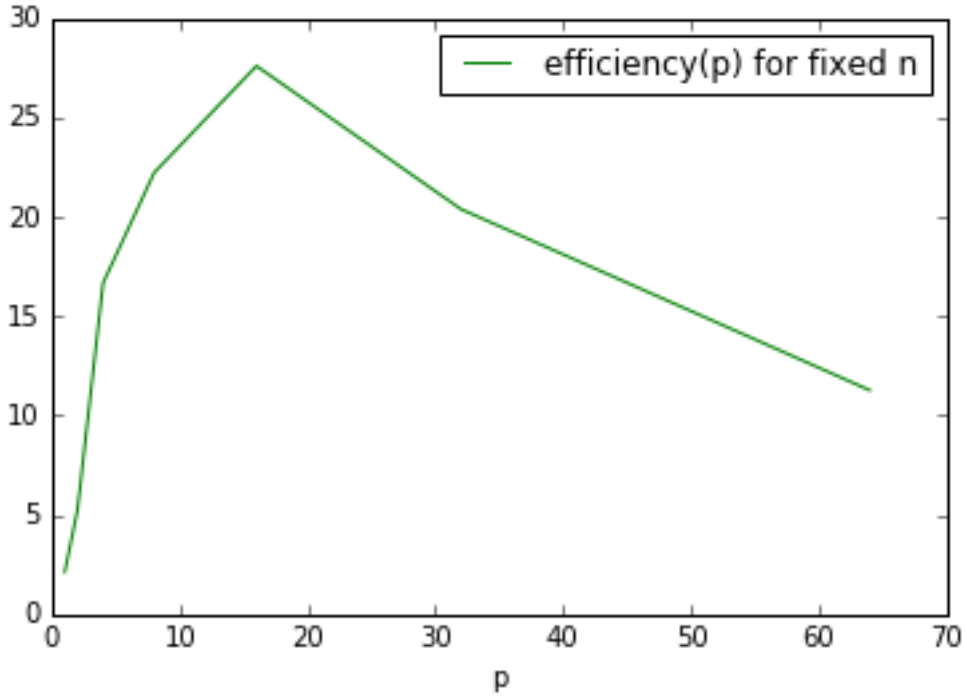
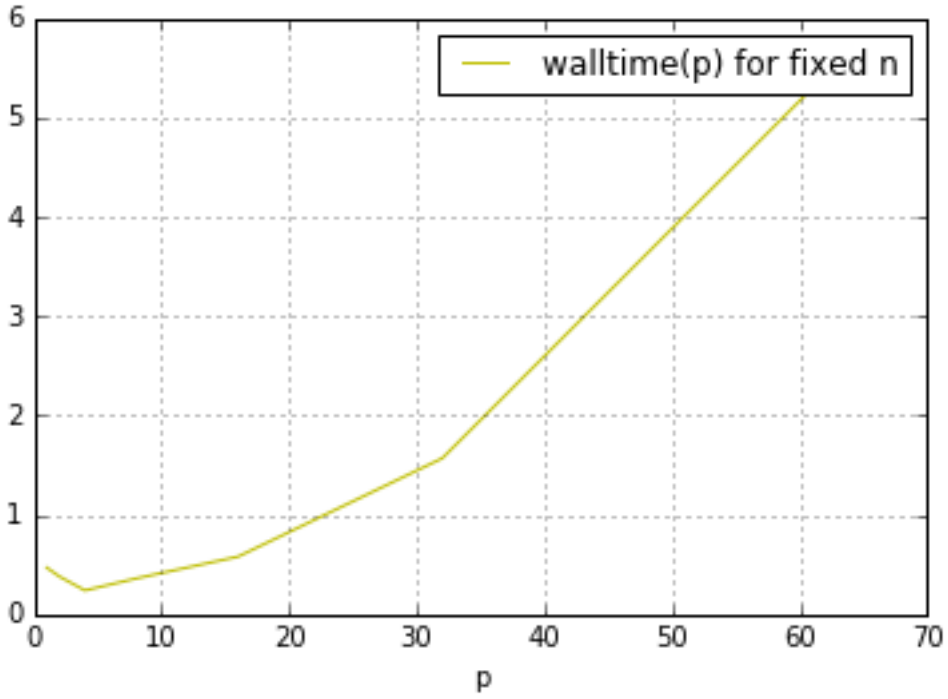


Figure: Ideal speedup ($S_P = P$) and realistic speedup.

It looks like we have plot the inverse. If we plot p/Sp we obtain a similar curb :



Finally we can plot the walltime as a function of P , for a given n size fixed :



If the number of processes is too high, the speed of the program decreases, maximum at about 5 processors.

Here could be solutions to increase the speed of the algorithm :

• Use nested loops, where none of the loops go higher
• Use an unsigned datatype. This gives us twice the range since the sign bit isn't needed.

Question 5.

Modify the function f to be a function of your own choice. Do you have to modify anything related to the parallel implementation when you change f ? ie changing Poisson problem ?

We changed the function to be exp(x)sin(2*pi*x)*sin(2*pi*y)

We don't calculate the analytical solution. We won't pay attention to the analytical error in the .o file because we don't know the analytical solution.

We expect to obtain a maximum of 2,7e-2 (verified with an internet website)

We simulate a first test for n = 128 , P = 8 and t = 4, also on two cores and a second test for n = 1024.

For these two test we obtain this value.

At this moment we can conjecture that our program can work on all the functions that are defined on [0-1]*[0-1] , (ie : unit square) , and which satisfies Dirichlet Boundary conditions, which is the case for this f, or for f = 0 except in two points where she is equal to 1 & -1 respectively, not on the bound.

Our program doesn't take into account the caracteristics of the function. Except the hypothesis on the subect (definition domain and Dirichlet conditions) , our parallel implementation just work mechanically and can solve various problem.

We don't have to change anything in our implementation.

Question 6.

Discuss how you would modify the numerical algorithm to deal with the case where u!= 0 on σ , i.e. for non-homogeneous Dirichlet boundary conditions. You do not have to implement this.

If the function f given as a right member is different from the given boundary condition, I think that I will extend the definition domain for the problem, using 0 values for the extended domain outside the unit square, and just run the algorithm.

Therefore the problem will have a different size so I will need to change some parameters, after that I will return the solution only on the unit square.

I will also need a more precised algorithm to enter the boundary conditions at the corresponding grid points.

Question 7.

In the exercise and lectures we have assumed that the domain is the unit square, i.e. (0; 1)(0; 1). Discuss how you would modify the Poisson solver based on diagonalization techniques if the domain instead is a rectangle with sides Lx and Ly. You can still assume a regular finite difference grid with n + 1 points in each spatial direction. Does this extension of the original method change anything in terms of your parallel implementation? You do not have to implement this.

I think I will try to reduce the problem to a [0-1]*[0-1] using a linear translation for the values of f.

I believe this solution will work because the Poisson problem is a linear equation. I will calculate the corresponding solution u to this problem and modify this values to get back on the problem on Lx*Ly.

In terms of parallel implementation, nothing will be changed, my program won't need any modifications.

ANNEXE A :

Python Program for Question 4

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 14 13:23:57 2017
@author: Robin - Nicolas
"""
# example of a source code to plot the curbs asked in question 4 .

from __future__ import division
import numpy as np
import matplotlib.pyplot as pl
from pylab import xlim
from math import log

X = [1,2,4,8,16,32,64,128,256,512,1024,2056,4096,8192,16384]
Y = [0.02,0.02,0.02,0.02,0.08,0.08,0.08,0.11,0.12,0.33,0.34,0.76,2.35,8.38,33.62]

pl.grid(True,which="both")
pl.plot(X,Y,"r",label="Sp(n) for p = 8")
pl.xlabel("n log")
pl.legend()
pl.show()

Z = [X[i]**2*log(X[i])for i in range(len(X))]

pl.plot(X,Z,"b",label="n²log(n)")
pl.grid(True,which="both")
pl.legend()
pl.show()

A = [1,2,4,8,16,32,64]
B = [0]*len(A)
C = [0.47,0.38,0.24,0.36,0.58,1.57,5.68]

for i in range(len(A)):
    B[i] = C[i] / A[i]

pl.plot(A,B,"g",label="efficiency(p) for fixed n")
pl.xlabel("p")
pl.grid(True)
pl.legend()
pl.show()

pl.plot(A,C,"y",label="walltime(p) for fixed n")
pl.xlabel("p")
pl.grid(True, which="both")
pl.legend()
pl.show()
```

ANNEXE B :

Script job.sh
#!/bin/bash #PBS -N jobyouhoulier #PBS -q training #PBS -A imf_lille-tma4280 #PBS -W group_list=imf_lille-tma4280 #PBS -l walltime=00:15:00 #PBS -l select=2:ncpus=20:mpiprocs=8:ompthreads=4 cd $PBS_O_WORKDIR echo $PBS_O_WORKDIR module load gcc/6.3.0 module load openmpi/2.0.1 mpirun ./poisson 1024 4

****** END ******