OpenMP course

Hands-on exercises

1 – Introduction

The hands-on exercises are done on the Jean Zay machine (HPE SGI 8600, 71560 cores, 40 cores per SMP node) in the $WORK/OpenMP_tp directory. There are eleven independent exercises. Each exercise is found in a directory named tp0 to tp10 which systematically contains a Makefile for the compilation, a batch.sh file for submitting in processing by lot and one or more source files to complete. The source files are available in Fortran and in C.

- ☞ The sans_indications_openmp directory contains the sequential code sources.
- ☞ In the avec_indications_openmp directory, we help you by explicitly indicating the places where the `OpenMP`directives must be inserted.
- ☞ The solution directory contains a solution for each exercise but, of course, it should not be consulted until you have exhausted your own resources to find a solution.

Hands-on exercises : Introduction

General comments

☞ Use the make mono command to compile a sequential version.

☞ Use the make para command to interpret the `OpenMP`directives and generate a parallel version.

☞ Use the make clean command to erase the object and core files or the make cleanall command to erase the object, core and executable files.

☞ If submitting in batch, use the sbatch batch.sh command. This includes a sequential and parallel execution on 2, 4, 6 and 8 threads. Be careful, the monoprocessor and parallel executables must be generated before. You may use the squeue -u $USER command to follow the evolution of the submitted job. When the job finishes normally, the result of the execution will be in a file whose name is suffixed with .res.

☞ Use the make visu command to generate and display the acceleration curve corresponding to the result of the execution stored in the .res file.

General instructions for the hands-on exercises

For each exercise, you must :

1. Analyse the status of the variables and parallelize the code by using the OpenMP directives.
2. Analyse the code performance on 2, 4, 6 and 8 threads compared to a sequential execution (submit in batch using the batch.sh file).
3. Plot the acceleration curves obtained.

Hands-on exercises : Introduction

Good luck !

2 – Ex.0 : Hello World

In this very simple exercise, you need to :

1. Write an OpenMP program displaying the number of threads used for the execution and the rank of each of the threads.
2. Compile the code manually to create a monoprocessor executable and a parallel executable.
3. Test the programs obtained with different numbers of threads for the parallel program, without submitting in batch.

Output example for the parallel program with 4 threads :

```
Hello from the rank 2 thread
Hello from the rank 1 thread
Hello from the rank 3 thread
Hello from the rank 0 thread
Parallel execution of hello_world with 4 threads
```

3 – Ex.1 : matrix product

The code contained in the prod_mat.f90 file calculates the matrix product :

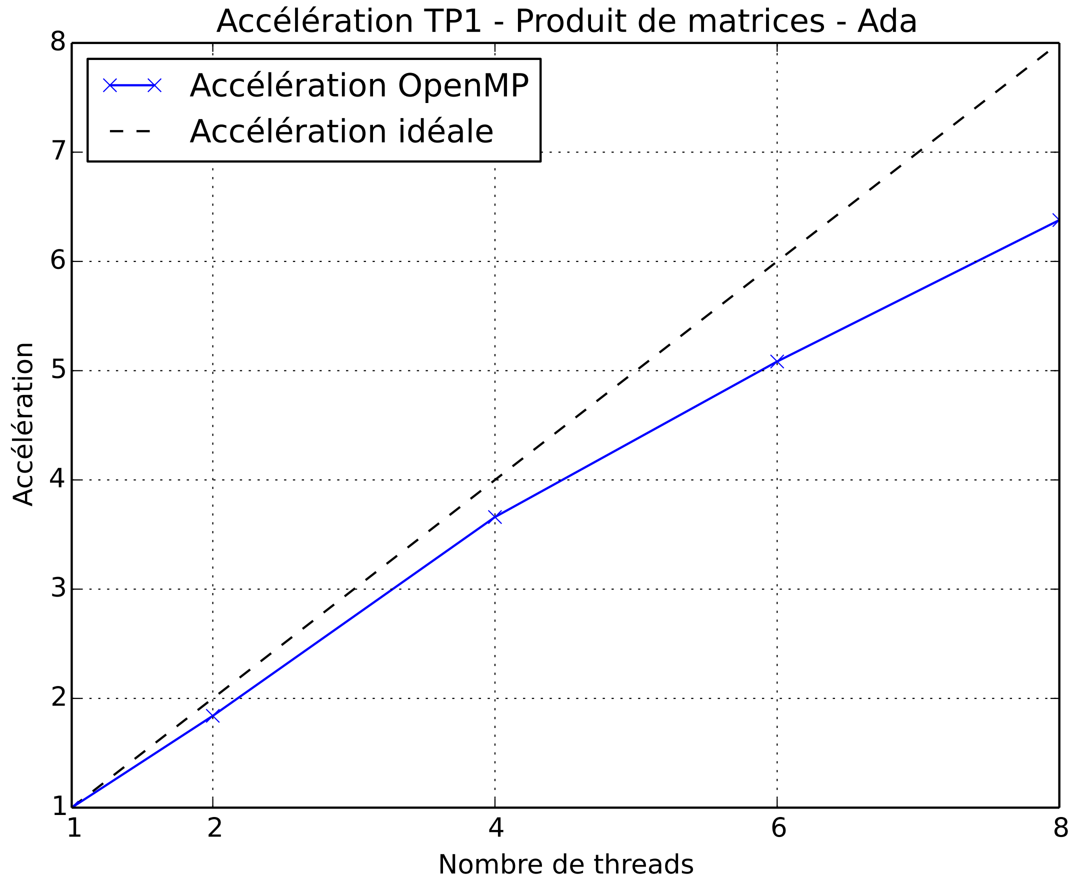
$$C = A \times B$$

In this exercise, you must :

1. Insert the appropriate OpenMP directives and analyse the code performance.
2. Test the loop iteration repartition modes (STATIC, DYNAMIC, GUIDED) and vary the chunk sizes.

Instructions for hands-on exercises : Ex.1

Nb. of threads	Elapsed time	Speedup
mono		
1		
2		
4		
6		
8		



4 – Ex.2 : Jacobi method

The program, contained in the jacobi.f90 file, solves a general linear system

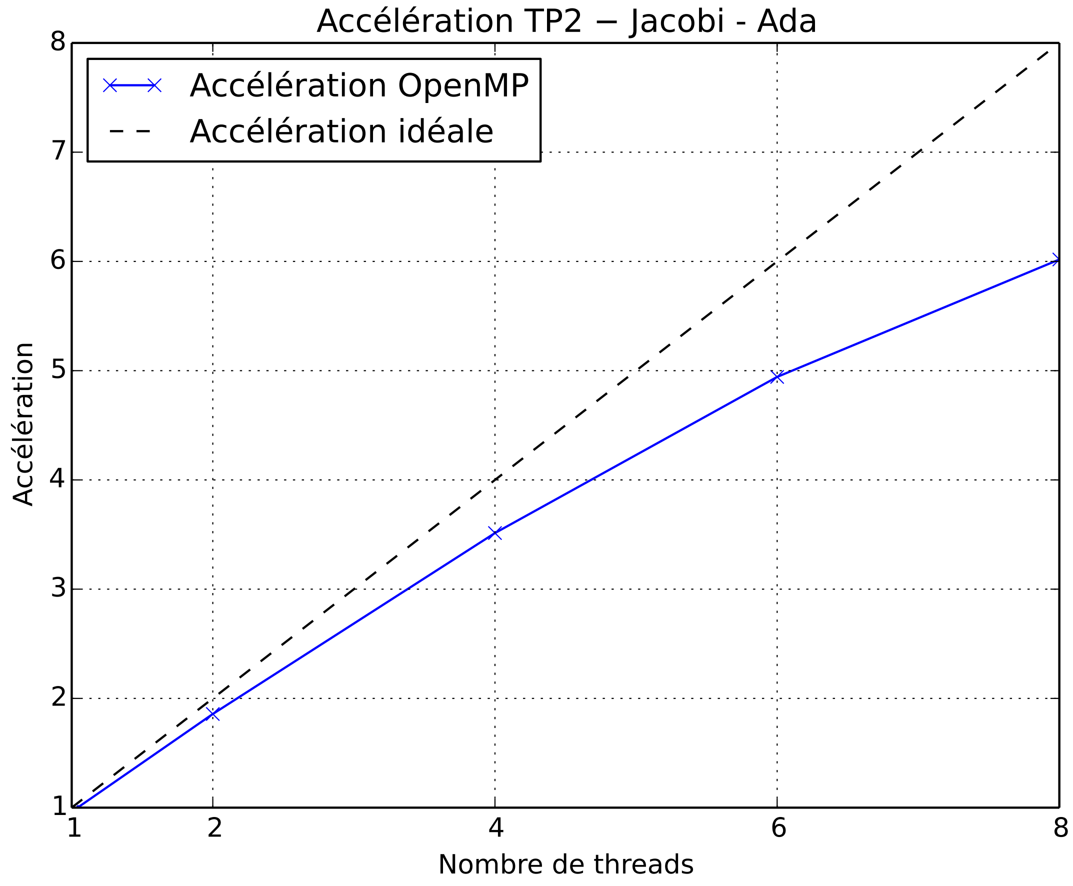
$$A \times x = b$$

using the JACOBI iterative method.

In this exercice, you must solve the system in parallel.

Instructions for hands-on exercises : Ex.2

Nb. of threads	Elapsed time	Speedup
mono		
1		
2		
4		
6		
8		



5 – Ex.3 : Calculation of π

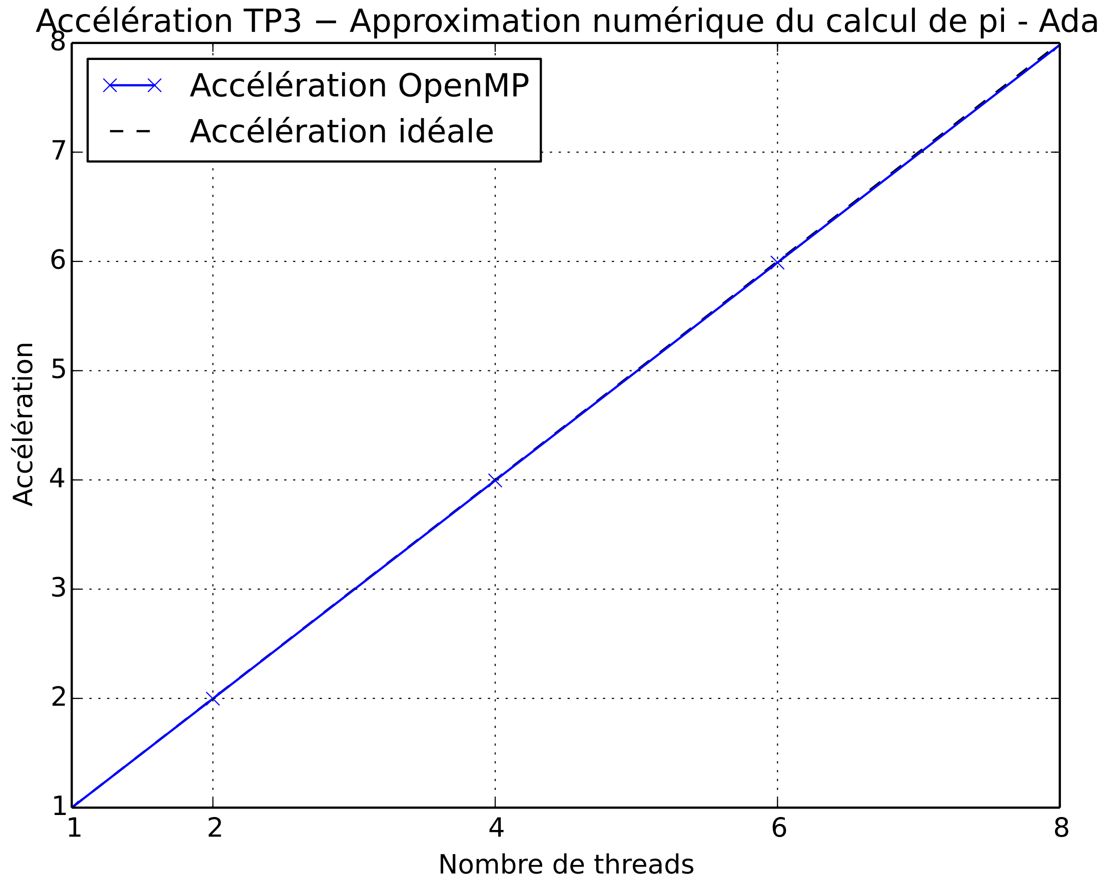
The aim of this exercise is to calculate π by numerical integration knowing that :
$\int_0^1 \frac{4}{1+x^2} dx = \pi$

The pi.f90 file contains the program for calculating the value of π by the rectangle method (mid-point). Let $f(x) = \frac{4}{1+x^2}$ be the function to integrate, N and $h = \frac{1}{N}$ (respectively) the number of points, and the discretization width on the integration interval $[0, 1]$.

This exercice can be parallelized in three different ways (i.e. using different OpenMP directives for each version). Analyse the performance of the three codes, then optimise the least efficient versions (without changing the type of OpenMP directives used), in order to obtain the same performance for the three parallelized versions.

Instructions for hands-on exercises : Ex.3

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



6 – Ex.4 : The conjugate gradient method

The program contained in the gradient_conjugue.f90 file solves a symmetric linear system

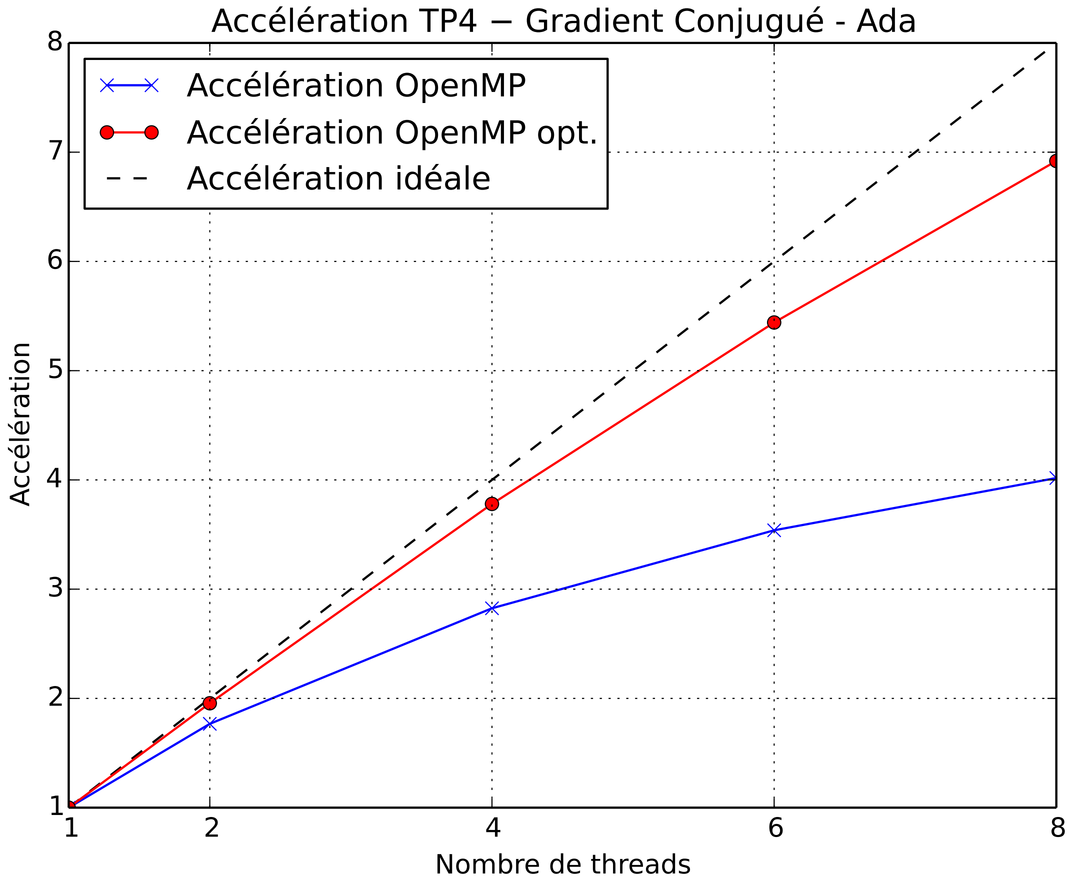
$$A \times x = b$$

using the preconditioned conjugate gradient method. In Fortran, this program can be parallelized primarily by using the WORKSHARE constructions.

1. After introducing the appropriate OpenMP directives, analyse the code performance.
2. What are your conclusions about the effectiveness of the WORKSHARE directive ?
3. Optimise the parallel version of the code by slightly modifying the source code in order to avoid using the WORKSHARE directive in places where it is problematic.

Instructions for hands-on exercises : Ex.4

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



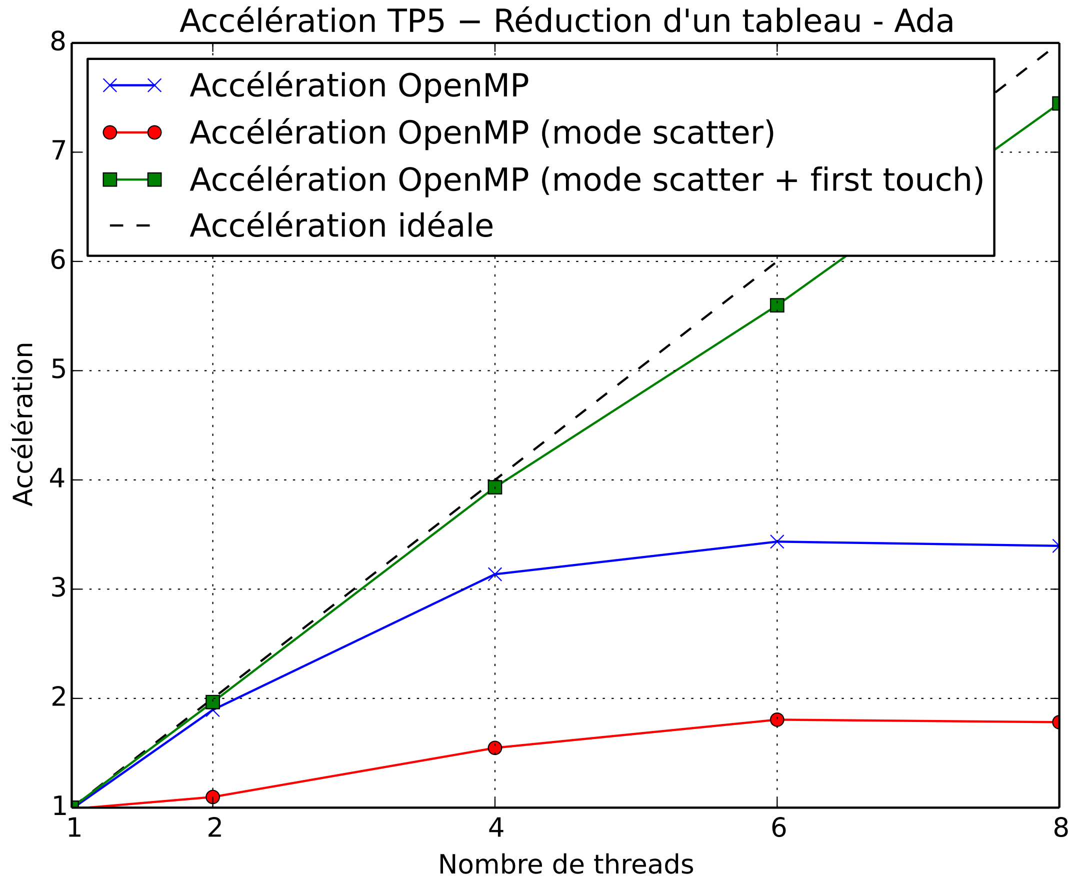
7 – Ex.5 : Reduction of an array

The program contained in the reduction_tab.f90 file is extracted from a chemistry code. It reduces a three-dimensional array into a vector. The aim of this exercise is to parallelize this calculation kernel without changing the loop order in the provided code (i.e. k,j,i).

1. Analyse the data-sharing attributes of the variables and adapt the source code so that the K outermost loop is parallelized.
2. Compare the performance obtained by using the thread/core binding default execution on Ada and by using scatter binding. Suggest an explanation for the poor performance of the latter.
3. Optimise the source code for the scatter mode with taking into account the memory affinity. Why does this third series of executions give the best performance ?

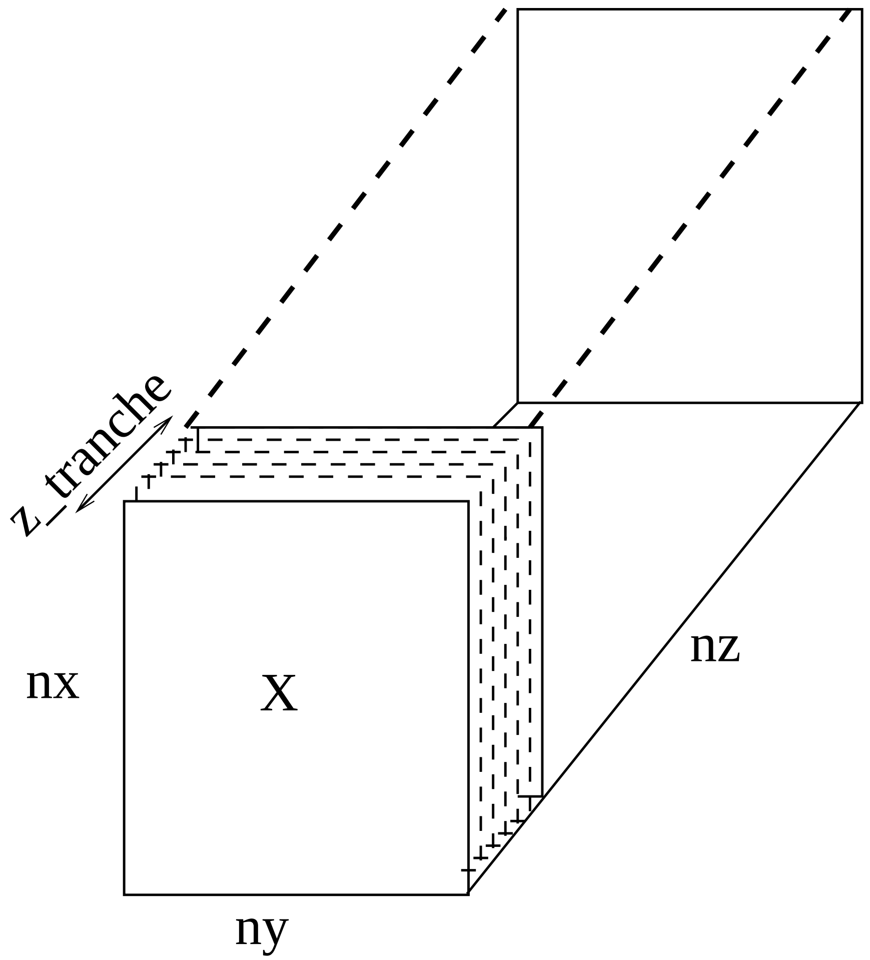
Instructions for hand-on exercises : Ex.5

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



8 – Ex.6 : Multiple Fast Fourier Transform

The program contained in the fft.f90 file computes the real-to-complex forward and inverse FFT of an x 3D matrix. The parallelization is carried out by explicit job distribution by slicing the x array in the 3rd dimension with as many slices as there are threads. Each thread then applies the FFT on its assigned slice, independently of the others.



Instructions for hands-on exercises : Ex.6

1. Insert the appropriate OpenMP directives into the fft.f90 file (use conditional compilation to allow for an eventual sequential execution).
2. Analyse the code performance and plot the speedup curves obtained.
3. Do the same for the scatter thread/core binding mode. Why do we observe a better performance without even having to modify the source code ?

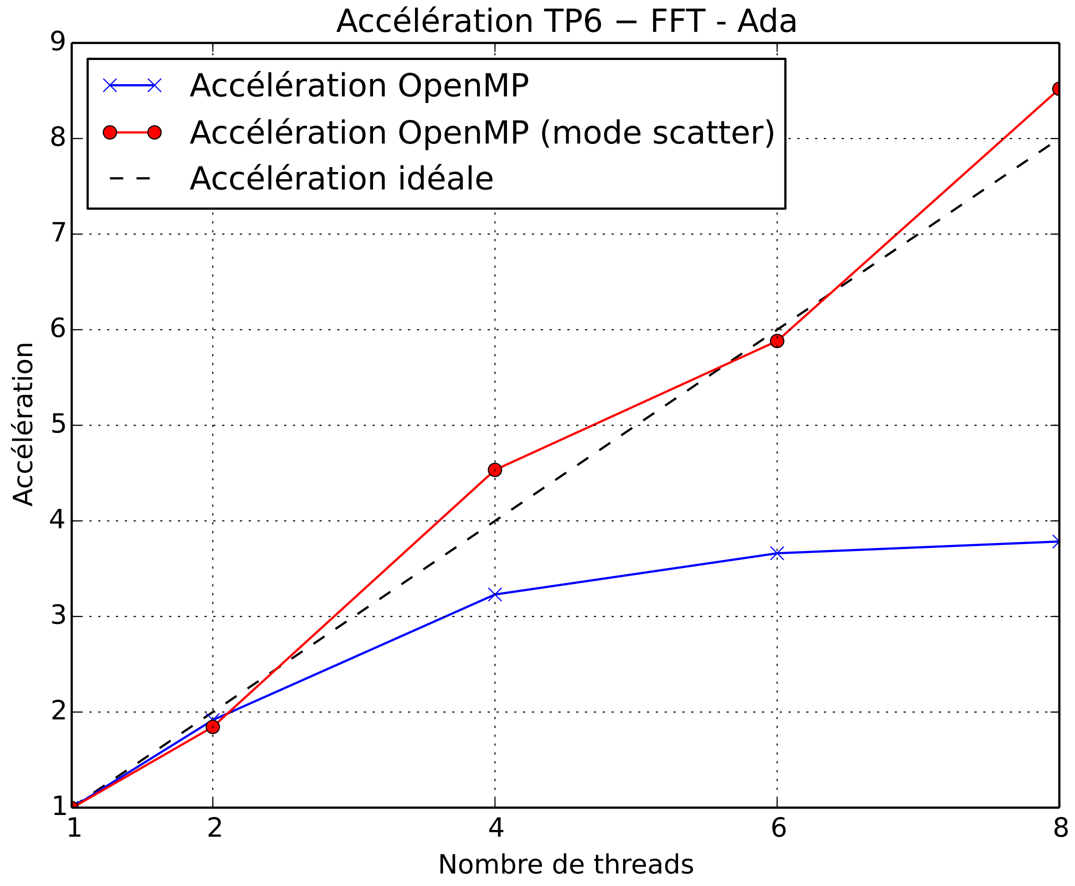
Comment : The FFT libjmfft.a library must not be modified.

It contains references to the scfftm and csfftm

subroutines used by the principal program.

Instructions for hands-on exercises : Ex.6

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



9 – Ex.7 : The BiConjugate Gradient Stabilized method

The principal program, contained in the principal.f90 file, calls the bi-cgstab subroutine defined in the bi-cgstab.f90 file, to solve a linear system with multiple right-hand sides

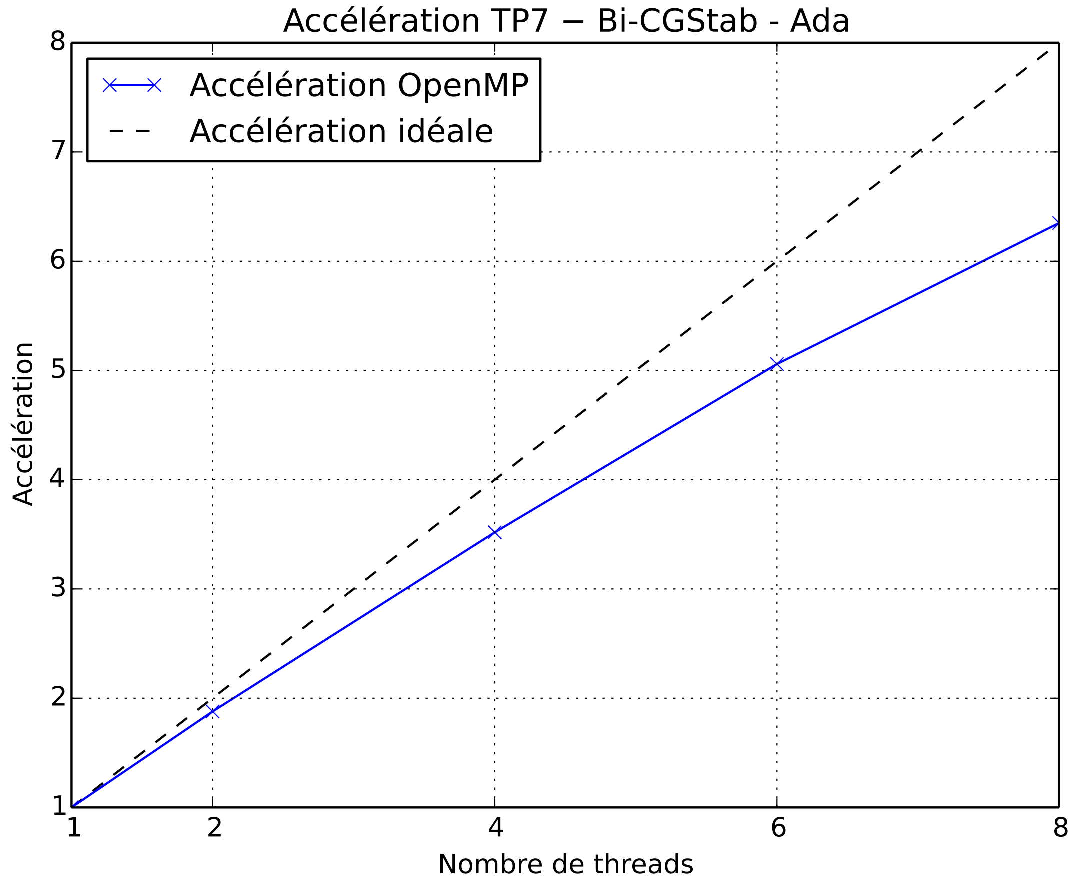
$$A \times x = b$$

using the BiConjugate Gradient Stabilized method (Bi-CGSTAB).

1. Insert the appropriate OpenMP directives into the principal.f90 and bi-cgstab.f90 files by considering the bi-cgstab subroutine as orphan.
2. Analyse the code performance and plot the speedup curves obtained.

Instructions for hands-on exercises : Ex.7

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



10 – Ex.8 : Poisson

The poisson.f90 and gradient_conjugue.f90 files (here extended to the solution of multiple independent linear systems) allows resolving the POISSON (1) equation for which the analytical solution $u_a(x, y)$ is given as :

$$u_a(x, y) = \cos \pi x \times \sin \pi y \quad ; \quad (x, y) \in [0, 1] \times [0, 1]$$

$$\begin{cases} -\dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial^2 u}{\partial y^2} = b(x, y) \\ u(0, y) = u_a(0, y) \\ u(1, y) = u_a(1, y) \\ u(x, 0) = u_a(x, 0) \\ u(x, 1) = u_a(x, 1) \end{cases} \tag{1}$$

Instructions for hands-on exercises : Ex.8

The numerical method adopted is mixed. We will apply a method of finite differences centered in the x direction followed by a sine FFT in the y direction. For this, let $\tilde{u}$ and $\tilde{b}$ represent, respectively, the sine FFT of u and b with respect to y and apply this FFT to POISSON's (1) equation which becomes :

$$-\frac{\partial^2 \tilde{u}}{\partial x^2} - \widetilde{\frac{\partial^2 u}{\partial y^2}} = \tilde{b}(x, y)$$

The sine transform $\widetilde{\frac{\partial^2 u}{\partial y^2}}$ of the $\frac{\partial^2 u}{\partial y^2}$ operator is a diagonal operator of which the elements represent the eigenvalues of the associated matrix obtained by finite differences of the operator in question. These eigenvalues are analytically known (which is the beauty of this method). If $j = 1, \ldots, N_j$ represents the index of the discretization point and h_y designates the discretization width in the y direction, these eigenvalues are expressed according to the following formula :

$$\mathrm{vp}_j = \frac{4}{h_y^2} \sin^2 \frac{\pi(j-1)}{2(N_j - 1)} \quad ; \quad j = 2, \ldots, N_j - 1$$

Instructions for hands-on exercises : Ex.8

Therefore, in the eigenvector basis, solving POISSON's equation is equivalent to solving $N_j - 2$ independant symmetric tridiagonal systems (use the conjugate gradient algorithm), each of size $(N_i - 2) \times (N_i - 2)$ where N_i represents the number of discretization points in the x direction :

$$\begin{pmatrix} d_j & -c_x & 0 & \dots & \dots & 0 \\ -c_x & d_j & -c_x & 0 & \dots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & -c_x & d_j & -c_x \\ 0 & \dots & \dots & 0 & -c_x & d_j \end{pmatrix} \begin{pmatrix} \tilde{u}_{2,j} \\ \tilde{u}_{3,j} \\ \vdots \\ \vdots \\ \tilde{u}_{N_i-2,j} \\ \tilde{u}_{N_i-1,j} \end{pmatrix} = \begin{pmatrix} \tilde{b}_{2,j} + \mathrm{CL} \\ \tilde{b}_{3,j} \\ \vdots \\ \vdots \\ \tilde{b}_{N_i-2,j} \\ \tilde{b}_{N_i-1,j} + \mathrm{CL} \end{pmatrix}$$

where $c_x = \frac{1}{h_x^2}$, $c_y = \frac{1}{h_y^2}$, $d_j = 2c_x + \mathrm{vp}_j$ and h_x is the width discretization in the x direction. The term CL contains the contribution of the boundary conditions.

Finally, $(N_i - 2)$ independent inverse FFT of $\tilde{u}$ with respect to y allows computing the final solution u in the canonical basis.

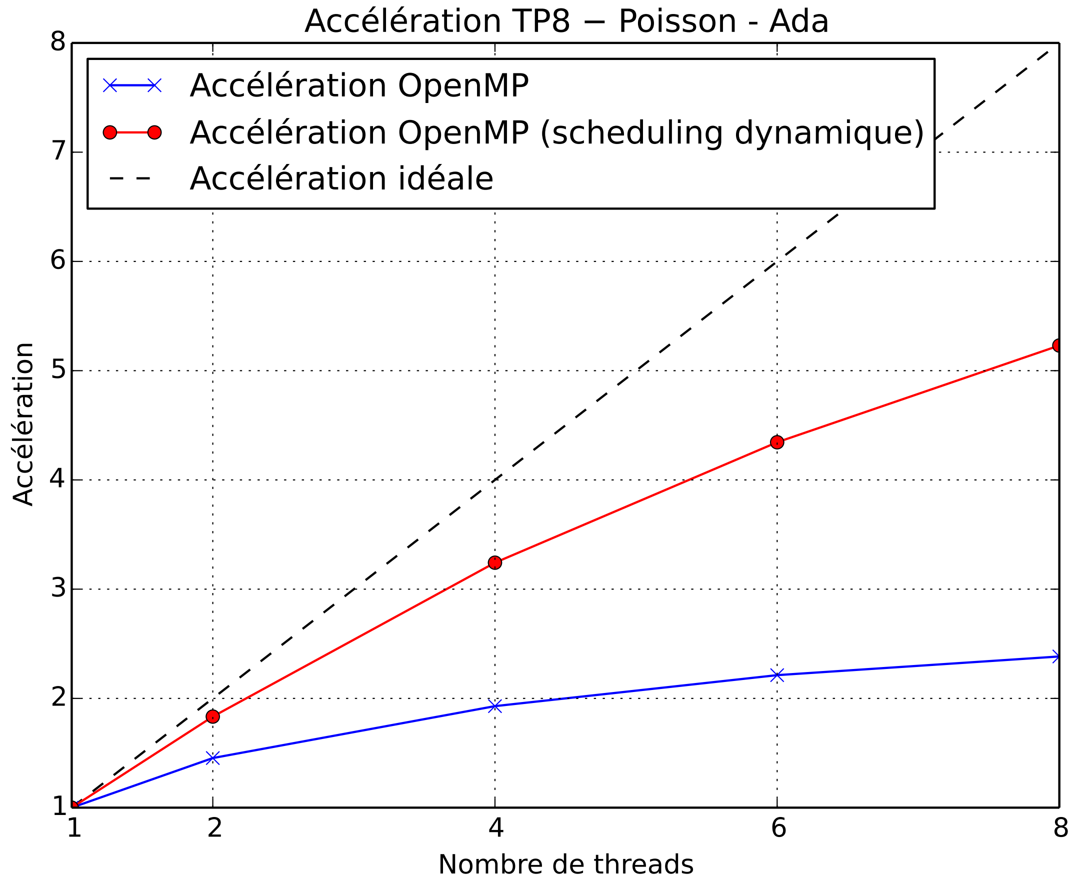
Instructions for hands-on exercises : Ex.8

1. Insert the appropriate OpenMP directives in the poisson.f90 and gradient_conjugue.f90 files by considering the gradient_conjugue subroutine as orphan and using only one parallel region.
2. Analyse the code performance and plot the speedup curves obtained.
3. Repeat this but use the DYNAMIC distribution mode of the iterations. Propose an explanation for the difference in performance observed.

Note : The c06haf.o file must not be modified. It contains the reference to the c06haf subroutine (it carries out the FFT in sinus and its inverse) called in the poisson.f90 file.

Instructions for hands-on exercises : Ex.8

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



11 – Ex.9 : Loop nest with dependencies

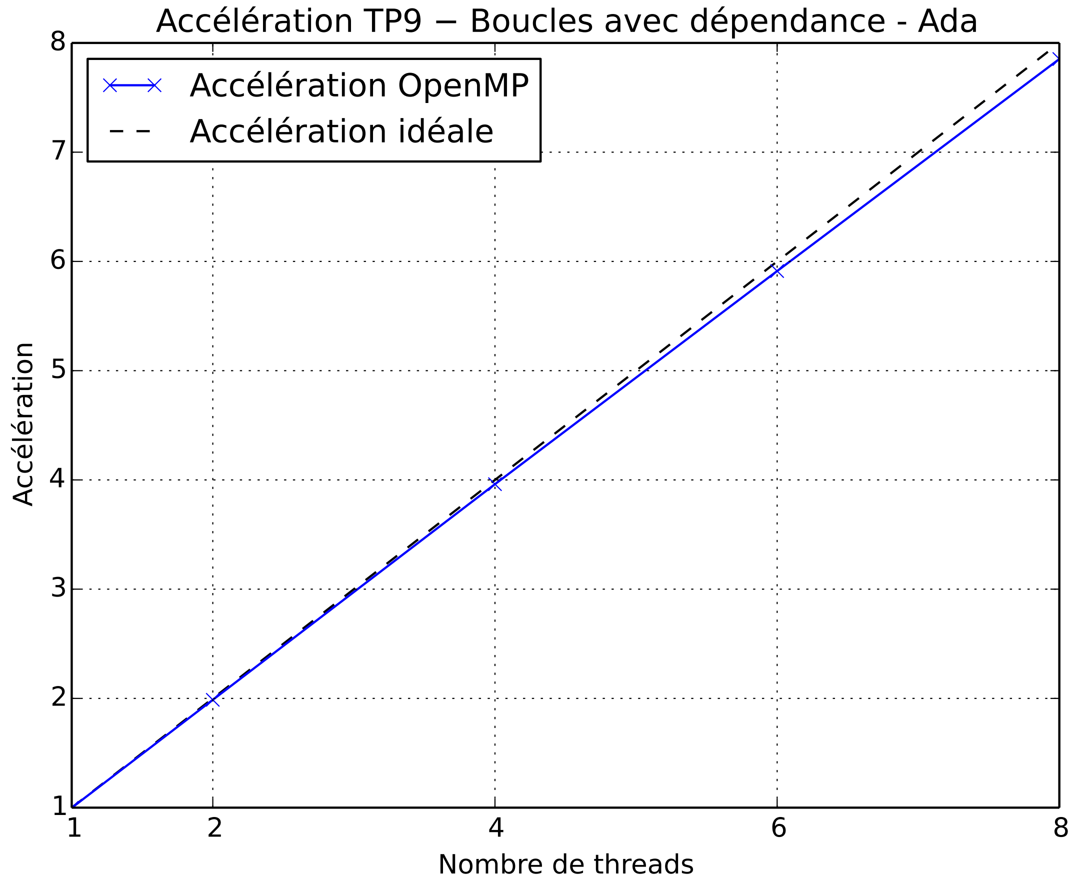
The code, stored in the dependance.f90 file, contains two nested loops.

In this exercice, you must :

1. Determine if the loops are parallel loops (i.e. no dependencies between iterations). If you force parallelization of the loops, what happens ?
2. Parallelize the code by inserting the appropriate OpenMP directives into the dependance.f90 file. Two approaches are possible, either with the flush or by using the OpenMP tasks. The difficulty of this exercise lies in correctly synchronising the various threads in a way which respects the dependencies between the iterations.
3. Analyse the code performance and plot the speedup curves obtained. Attention, the parallel version of the code is only valid if the value displayed on the screen for the variable norm equals 0.

Instructions for hands-on exercises : Ex.9

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		



12 – Ex.10 : Matrix product by the Strassen algorithm

The code, contained in the strassen.F90 file, calculates the matrix product :

$$C = A \times B$$

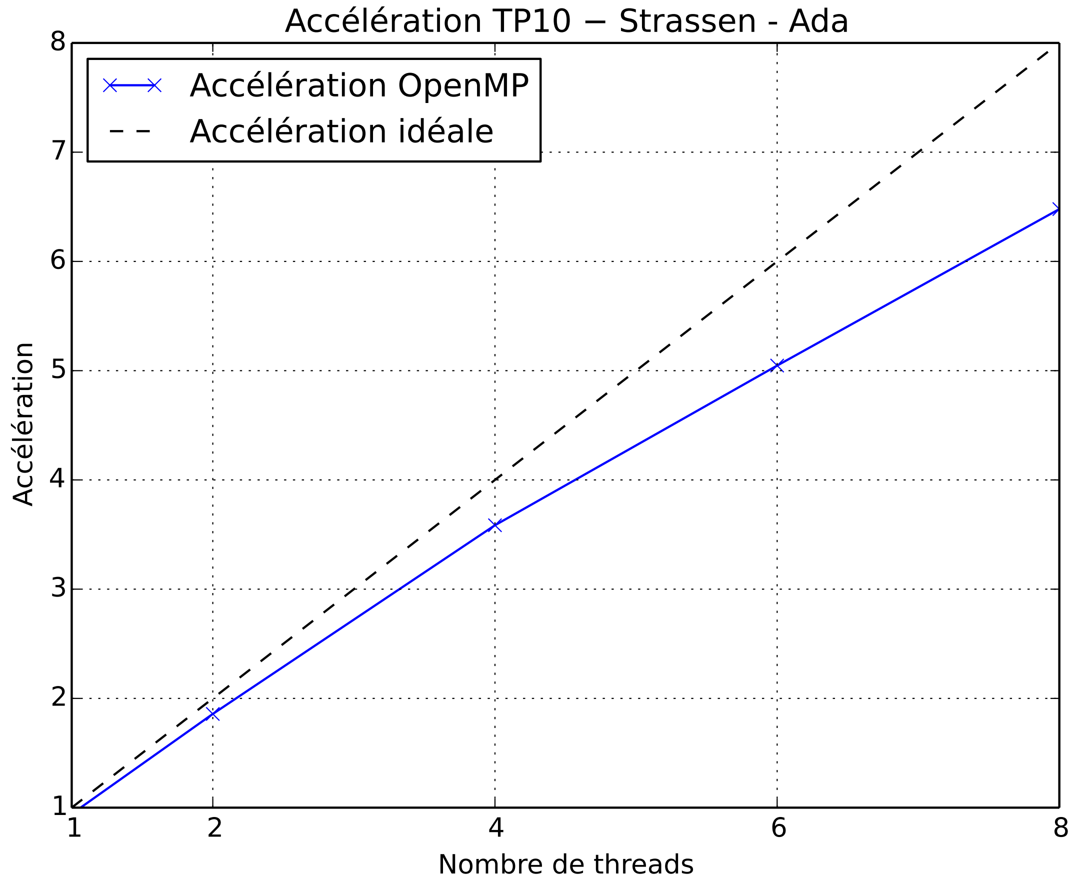
by using Strassen's recursive algorithm.

In this exercise, you must :

1. Analyse and parallelize the code by using OpenMP tasks.
2. Measure the code performance and plot the speedup curves obtained.

Instructions for hands-on exercises : Ex.10

Nb. of threads	Elapsed time	Speedup
seq.		
1		
2		
4		
6		
8		


Accélération TP10 − Strassen - Ada
Accélération OpenMP
Accélération idéale
Accélération
Nombre de threads