ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High Performance Computing for Science and Engineering I

Autumn semester 2013

Prof. Dr. P. Koumoutsakos
ETH Zentrum, CLT F12
CH-8092 Zürich

Project 2

Issued: October 29, 2013
Hand in: November 19, 2013

Diffusion

Diffusion is the process that describes the spreading of a quantity of interest driven by its concentration gradient towards regions with lower density. In this exercise we will consider heat flow in a medium that can be described by the diffusion equation of the form:

$$\frac{\partial \rho(\mathbf{r},\, t)}{\partial t} \;=\; D\nabla^2 \rho(\mathbf{r},\, t) \tag{1}$$

where $\rho(\mathbf{r}, t)$ is a measure for the amount of heat at position $\mathbf{r}$ and time t and the diffusion coefficient D is constant. Lets define the domain Ω in two dimensions as $x, y, \in [0,\, 1]$. We will use Dirichlet boundary conditions

$$\rho(0, y, t) = \rho(x, 0, t) = \rho(1, y, t) = \rho(x, 1, t) = 0 \quad \forall t > 0 \tag{2}$$

and an initial density distribution

$$\rho(x, y, 0) = sin(x \cdot 2\pi) \cdot sin(y \cdot 2\pi), \tag{3}$$

for which the analytical solution is given by

$$\rho(x, y, t) = sin(x \cdot 2\pi) \cdot sin(y \cdot 2\pi) \cdot e^{-8D\pi^2 t}. \tag{4}$$

Numerical integration of Equation 1 depends on the time and space discretization. We will denote by n the index of time $t_n = n \cdot \delta t$ and by i and j the indices of the discretized spatial points $x_i = i \cdot \delta x$, $y_j = j \cdot \delta y$. Then the amount of heat at position (x_i, y_i) at time t^n is denoted as $\rho^n_{i,j}$.

ADI

Alternating Direction Implicit (ADI) is a finite difference scheme based on the idea of operator splitting. The ADI method applied to the diffusion equation (Equation 1) is summarized in the following two steps:

Step 1 Implicit Euler in x direction; Explicit Euler in y direction:

$$\rho_{i,j}^{n+\frac{1}{2}} = \rho_{i,j}^{n} + \frac{D\delta t}{2}\left[\frac{\partial^2 \rho_{i,j}^{n+\frac{1}{2}}}{\partial x^2} + \frac{\partial^2 \rho_{i,j}^{n}}{\partial y^2}\right] \tag{5}$$

Step 2 Explicit Euler in x direction; Implicit Euler in y direction:

$$\rho_{i,j}^{n+1} = \rho_{i,j}^{n+\frac{1}{2}} + \frac{D\delta t}{2}\left[\frac{\partial^2 \rho_{i,j}^{n+\frac{1}{2}}}{\partial x^2} + \frac{\partial^2 \rho_{i,j}^{n+1}}{\partial y^2}\right] \tag{6}$$

The algorithm takes advantage of both implicit and explicit schemes and results in an integration scheme which is unconditionally stable and second order in space and time.

Moreover, the implicit integration step, which usually requires iterative methods in order to invert a matrix, consists in a tridiagonal system of equations which can be efficiently solved by the Thomas algorithm.

In order to use the latter method for the implicit step in x direction, we need to rewrite for each row j the implicit step

$$\rho_{i,j}^{*} = \rho_{i,j}^{n} + \frac{\delta t}{2}\frac{D}{\delta x^2}\left(\rho_{i-1,j}^{*} - 2\rho_{i,j}^{*} + \rho_{i+1,j}^{*}\right) \tag{7}$$

in the form $\mathbf{A}\boldsymbol{\rho}^{*} = \boldsymbol{\rho}^{n}$, where $\mathbf{A}$ is a tridiagonal matrix of coefficients a_i (lower diagonal, $i = 1..n$), b_i (main diagonal, $i = 0..n$) and c_i (upper diagonal, $i = 0..n-1$), $\boldsymbol{\rho}^{*}$ is the unknown vector of $\rho_{i,j}^{*}$ with $i = 0..n$ and $\boldsymbol{\rho}^{n}$ is the corresponding solution vector of $\rho_{i,j}^{n}$ with $i = 0..n$ at time n for row j.

An implicit step of ADI in x direction therefore consists in solving m tridiagonal systems of equations, where m is the size of the grid in y direction.

For the implicit Euler step in y direction we proceed analogously by writing a m systems of equations, where m is the size of the grid in x direction.

Thomas Algorithm

The Thomas algorithm[1], named after Llewellyn Thomas, is an $\mathcal{O}(n)$ algorithm used to solve a tridiagonal system of equations and can therefore be used to solve the implicit component of ADI.

In the following, a C code is given for the Thomas algorithm as a reference (from Wikipedia):

```
1 void solveMatrix(int n, double *a, double *b, double *c,
    double *v, double *x)
2 {
3     /**
4      * n - number of equations
```

[1] from Wikipedia

```
5        * a - sub-diagonal (means it is the diagonal below the
           main diagonal) -- indexed from 1..n-1
6        * b - the main diagonal
7        * c - sup-diagonal (means it is the diagonal above the
           main diagonal) -- indexed from 0..n-2
8        * v - right part
9        * x - the answer
10       */
11      for (int i = 1; i < n; i++)
12      {
13          double m = a[i]/b[i-1];
14          b[i] = b[i] - m*c[i-1];
15          v[i] = v[i] - m*v[i-1];
16      }
17
18      x[n-1] = v[n-1]/b[n-1];
19
20      for (int i = n - 2; i >= 0; i--)
21          x[i] = (v[i] - c[i] * x[i+1]) / b[i];
22  }
```

Note that the algorithm is not parallel due to dependencies between iterations.

Performance Measurements

Understanding the characteristics of the platform on which performance tests are executed is of fundamental importance since it gives a context in which to read performance results.

We consider here the roofline model[2], a simple visual tool that can be used to understand performance on a multicore architecture. In this model, the hardware architecture is abstracted to its memory bandwidth and its peak floating point performance, while a program is characterized by its operational intensity, a quantity representing the ratio of floating point operations to amount of memory accesses to the main memory. Bandwidth and peak floating point performance are used to draw the so-called roofline on a log-log plot of GFLOP/s performance against operational intensity, in which the bandwidth is represented as an oblique line passing through the point of operational intensity 1 and floating point performance equal to its value (see Figure 1).

The point in which the bandwidth and peak performance line cross each other is called ridge point and defines a region where the computation is memory bound (on its left) and one which is compute bound (on its right).

Question 1: A-Priori Performance Estimation

a) Draw the roofline for a Brutus node with 48 cores that can perform 844.8 GFLOP/s and has a bandwidth of 96 GB/s.

See Figure 2.

[2]Williams et al, 2009: the original paper can be found on the course website.

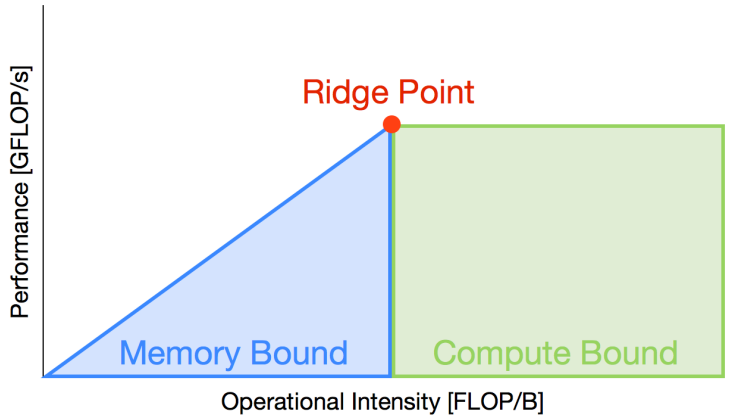

Figure 1: Roofline model depicting memory bound and compute bound regions.

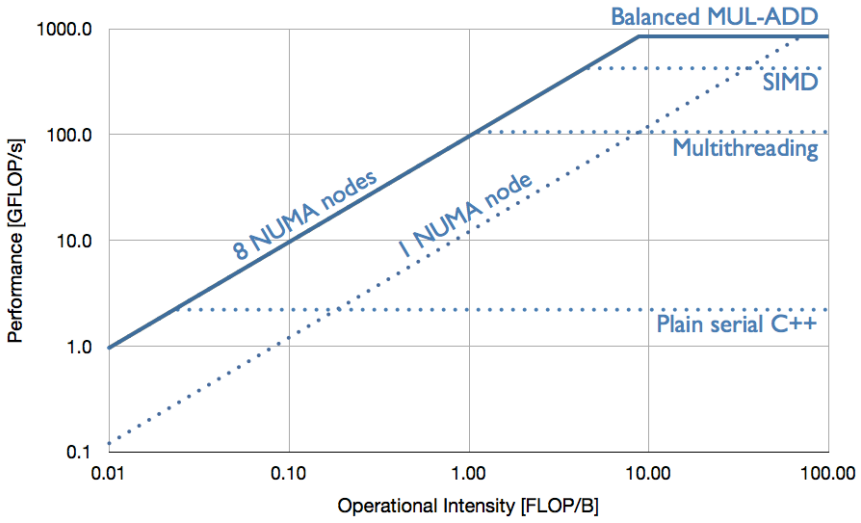

Figure 2: Roofline for one Brutus node (four 12-core AMD Opteron 6174 CPUs and 64 GB of RAM). Different ceilings are depicted.

b) How many floating point operations are required for the computation of a single time step with ADI? How many memory accesses are required if we assume that there is no caching? Compute the operational intensity from the two values you computed.

State any assumption you make while counting operations and memory accesses.

c) How many floating point operations are required for the computation of a single time step with ADI? How many memory accesses are required if we assume perfect caching (with an infinite sized cache)? Compute the operational intensity from the two values you computed.

State any assumption you make while counting operations and memory accesses.

Let us calculate the operational intensity of the ADI. We will denote the number of floating point operations (flops) as C, number of memory accesses as M. The two ADI sub-steps given in Equations 5 and 6 are equivalent if the number of point n per dimension is the same, so it is sufficient to examine only one of them, e.g. Equation 5. We rewrite the step in the following

form:

$$\underbrace{-\frac{\delta t}{2}\frac{D}{\delta x^2}}_{a_{ij}}\rho_{i-1,j}^{n+\frac{1}{2}} + \underbrace{\left(\delta t\frac{D}{\delta x^2}+1\right)}_{b_{ij}}\rho_{i,j}^{n+\frac{1}{2}} - \underbrace{\frac{\delta t}{2}\frac{D}{\delta x^2}}_{c_{ij}}\rho_{i+1,j}^{n+\frac{1}{2}} =$$

$$= \underbrace{\overbrace{\left(1-\delta t\frac{D}{\delta y^2}\right)}^{\text{const}}\rho_{i,j}^{n} + \frac{\delta t}{2}\frac{D}{\delta y^2}\left(\rho_{i,j-1}^{n}+\rho_{i,j+1}^{n}\right)}_{v_{ij}}. \quad (8)$$

The evaluation of the step consists of the following operations:

1. Compute the coefficients of the left-hand side for the n tridiagonal systems of equations in direction x.

   The coefficients a_{ij}, b_{ij}, c_{ij} of the left-hand side can be precomputed and don't require flops or memory accesses. So the number of flops is $C_{LHS} = 0$ and the number of memory accesses is $M_{LHS} = 0$.

2. Compute the corresponding right-hand sides of the systems.

   Computing v_{ij} requires 4 floating point operations as the coefficients can be precomputed. So:

   $$C_{RHS} = 4n^2.$$

   For calculating memory accesses we will consider two cases.

   - No caching. Then we always read and write and

     $$M_{RHS}^{\text{No cache}} = (\underbrace{1}_{\text{writes}} + \underbrace{3}_{\text{reads}})n^2 = 4n^2.$$

   - Perfect caching, but the problem size is big enough so that the data doesn't fully fit in cache. If we now iterate over j, then for v_{ij} we will only need to read $\rho_{i,j+1}^n$ as the values $\rho_{i,j}^n$ and $\rho_{i,j-1}^n$ are in cache due to the previous iterations. A write is still necessary though. So

     $$M_{RHS}^{\text{Small cache}} = (1+1)n^2 = 2n^2.$$

3. Solve the n systems with Thomas algorithm.

   Looking at the Thomas algorithm for one tridiagonal system, we observe following amount of flops:

   $$C_{\text{Thomas}} =$$
   $$= \underbrace{(n-1)}_{\text{loop on line 11}} \cdot (\underbrace{1}_{\text{line 13}} + \underbrace{2}_{\text{line 14}} + \underbrace{2}_{\text{line 15}}) + \underbrace{1}_{\text{line 18}} + \underbrace{(n-1)}_{\text{loop on line 20}} \cdot \underbrace{3}_{\text{line 21}} =$$
   $$= 8n - 7$$

   For memory accesses we follow the assumptions for caching specified above and assume that the temporary variable `m` will be stored in register. Additionally we note that $a_{ij} = c_{ij} = \text{const}$ for all $1 \leqslant i, j \leqslant n$. Therefore we don't need arrays to store a_{ij} or c_{ij}, instead we will use just a single variable that will effectively be stored in a register. Note also that

we still need an array to store b_{ij} as these values are modified by the algorithm. Then for no caching:

$$M^{\text{No cache}}_{\text{Thomas}} = \\ = \underbrace{(n-1)}_{\text{loop on line 11}} \cdot (\underbrace{0+1}_{\text{line 13}} + \underbrace{1+1}_{\text{line 14}} + \underbrace{1+2}_{\text{line 15}}) + \underbrace{1+2}_{\text{line 18}} + \underbrace{(n-1)}_{\text{loop on line 20}} \cdot (\underbrace{1+3}_{\text{line 21}}) = \\ = 10n - 7$$

For small cache we can make an additional reasonable assumption that for $n < 100$k the data of a single Thomas algorithm will fit in cache and arrays `b` and `v` will still stay in it for the loop on line 20 of the listing.

$$M^{\text{Small cache}}_{\text{Thomas}} = \\ = \underbrace{(n-1)}_{\text{loop on line 11}} \cdot (\underbrace{0+0}_{\text{line 13}} + \underbrace{1+0}_{\text{line 14}} + \underbrace{1+1}_{\text{line 15}}) + \underbrace{1+0}_{\text{line 18}} + \underbrace{(n-1)}_{\text{loop on line 20}} \cdot (\underbrace{1+0}_{\text{line 21}}) = \\ = 4n - 3$$

And as we have n systems:

$$\begin{aligned} nC_{\text{Thomas}} &= 8n(n-1) + n, \\ nM^{\text{No cache}}_{\text{Thomas}} &= 10n(n-1) + 3n, \\ nM^{\text{Small cache}}_{\text{Thomas}} &= 4n(n-1) + n. \end{aligned}$$

Note that for infinitely large cache you will end up with *NO* reads and writes except for the first and last steps of simulation. In this case to compare against roofline you should use the assumptions of no cache and instead of RAM memory bandwidth use the cache bandwidth in the roofline.

For the operational intensity (OI) of ADI we assume double precision computations (8 bytes per floating point number):

$$OI^{\text{No cache}} = \frac{1}{8}\frac{C_{LHS} + C_{RHS} + nC_{\text{Thomas}}}{M_{LHS} + M_{RHS} + nM^{\text{No cache}}_{\text{Thomas}}} = \frac{1}{8}\frac{0 + 4n^2 + 8n(n-1) + n}{0 + 4n^2 + 10n(n-1) + 3n} \approx \frac{12}{112} \approx 0.11$$

$$OI^{\text{Small cache}} = \frac{1}{8}\frac{C_{LHS} + C_{RHS} + nC_{\text{Thomas}}}{M_{LHS} + M_{RHS} + nM^{\text{Small cache}}_{\text{Thomas}}} = \frac{1}{8}\frac{0 + 4n^2 + 8n(n-1) + n}{0 + 2n^2 + 4n(n-1) + n} \approx \frac{12}{80} = 0.25$$

If the code is NUMA-aware, the expected performance P for the corresponding operational intensity will be

$$P^{\text{No cache}} = \text{Bandwidth} \cdot OI^{\text{No cache}} \approx 96 \cdot 0.11 \approx 10.6 \text{ Gflops/s}, \tag{9}$$

$$P^{\text{Small cache}} = \text{Bandwidth} \cdot OI^{\text{Small cache}} \approx 96 \cdot 0.25 = 24 \text{ Gflops/s}, \tag{10}$$

whereas, if the solver does not consider NUMA, the operational intensity becomes

$$P^{\text{No cache}} = \text{Bandwidth} \cdot OI^{\text{No cache}} \approx 12 \cdot 0.11 \approx 1.3 \text{ Gflops/s}, \tag{11}$$

$$P^{\text{Small cache}} = \text{Bandwidth} \cdot OI^{\text{Small cache}} \approx 12 \cdot 0.25 = 3 \text{ Gflops/s}. \tag{12}$$

We note that even in the case where NUMA is not considered, the total bandwidth at our disposal is still 96 GB/s but higher latencies and serialization of memory accesses might lead in the worst case to an effective bandwidth of 12 GB/s (all data is found within a single NUMA node).

For ADI it is very likely that one of the sub-steps has threads that access most of the data within their own NUMA domain (only local data is used by each processor), while the other sub-step requires a lot of cross communication between NUMA nodes. So we can expect that the code will be executed in the regime of small cache with a total performance of $P = 2\left(\frac{1}{24} + \frac{1}{3}\right)^{-1} \approx 5.3$ GFLOP/s.

Question 2: Diffusion with ADI, serial implementation

a) Implement ADI for the 2D diffusion problem with boundary conditions and initial conditions given in Equations 2 and 3. Choose your time step δt size such as:

$$0.5 = \frac{D\delta t}{\delta x^2}, \tag{13}$$

where δx is the grid point spacing and is equal in both directions.

b) Perform a spatial and a temporal convergence study and plot your results for the L_∞, L_1 and L_2 error norms to verify the correctness of your program.

c) Make a 2D contour plot of the density $\rho(x, y, t)$ at $t = 0.1$ with diffusion constant $D = 1$.

See corresponding figures 3, 4, 5. All the following runs of the solution were performed on a single Brutus node with four 12-core AMD Opteron 6174 CPUs and 64 GB of RAM. Note that after a certain point the error in temporal convergence study doesn't decrease anymore with the decrease of the time-step. It happens when the spatial error dominates the temporal one: note the difference between left and right of Figure 3.

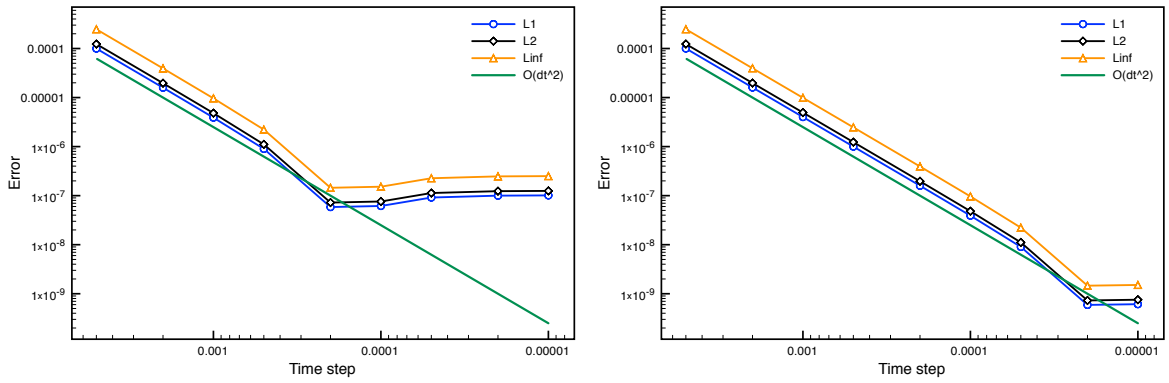

Figure 3: Temporal convergence of the ADI solution. The left plot is made with 1000 points per dimension, the right one with $10,000$ points.

Question 3: Diffusion with ADI, multithreaded implementation

a) Parallelize your implementation of ADI using OpenMP.

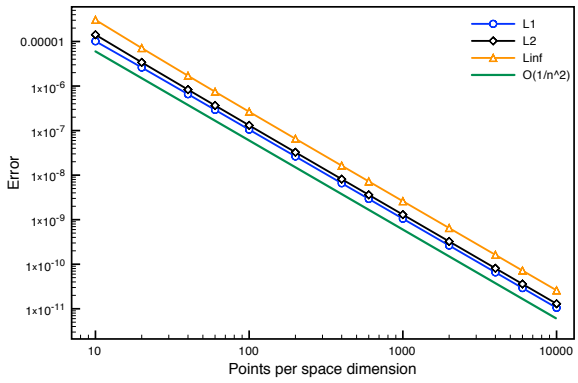

Figure 4: Spatial convergence of the ADI solution. $\delta t = 10^{-8}$, 1000 steps.

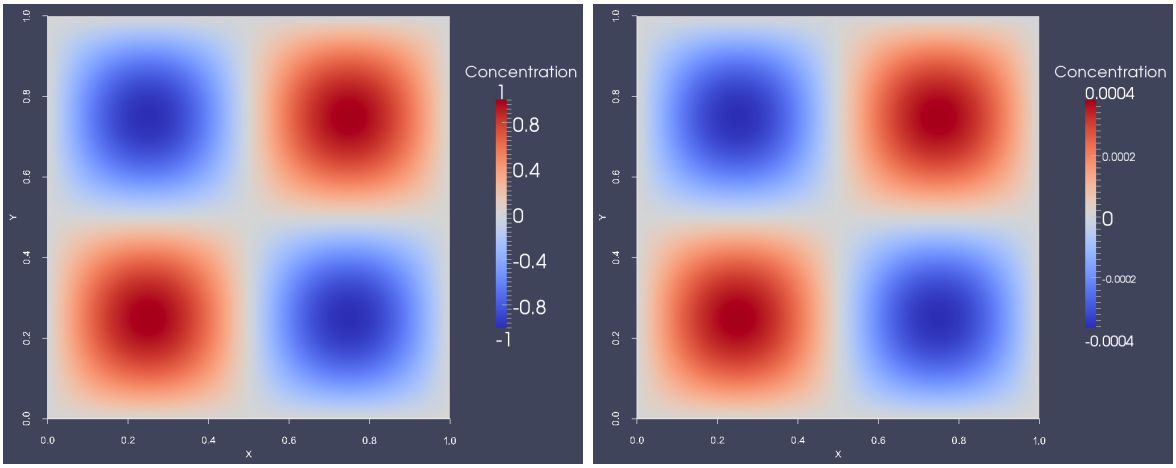

Figure 5: Initial conditions (left) and the concentration at $t = 0.1$ (right). 1000×1000 points, $\delta t = 10^{-4}$. Note the different range of values of the colormap.

b) Perform a spatial and a temporal convergence study and plot your results for the L_∞, L_1 and L_2 error norms to verify the correctness of your program. The error norms are given by:

$$L_\infty = \max |\tilde{u}_i - u_i|, \tag{14}$$

$$L_1 = \frac{\sum_i |\tilde{u}_i - u_i|}{N}, \tag{15}$$

$$L_2 = \sqrt{\frac{\sum_i (\tilde{u}_i - u_i)^2}{N}}, \tag{16}$$

where N is the total number of grid points, $\tilde{u}_i$ is the numerical solution at point x_i and u_i the analytical solution. Note that these formulations for the error norms assume that the grid points are equally spaced.

c) Plot strong and weak scaling using 1, 2, 3, 6, 12, 24 and 48 cores. Define the problem size as the number of grid points in the discretization of Ω. How does your code scale? How does the problem size affect the scaling? Comment on your results.

The convergence plots didn't change with respect to ones obtained with serial code. The solutions are just identical up to machine precision. For the scaling see Figure 6.

A few words should be spent about the performance of the code. With 48 threads, 1000×1000 points and $10,000$ time-steps the execution time took 46 seconds (averaged over 10 runs). This corresponds to $12 \cdot 1000^2 \cdot 10,000 \cdot 2 = 2.4 \cdot 10^{11}$ floating point operation and $\frac{2.4 \cdot 10^{11}}{46} = 5.2$ GFLOP/s. According to the a-priory analysis we expected the performance to be about 5.3 GFLOP/s Note that having made really rough but reasonable estimations we obtained the performance which is in a good agreement with the measurement.

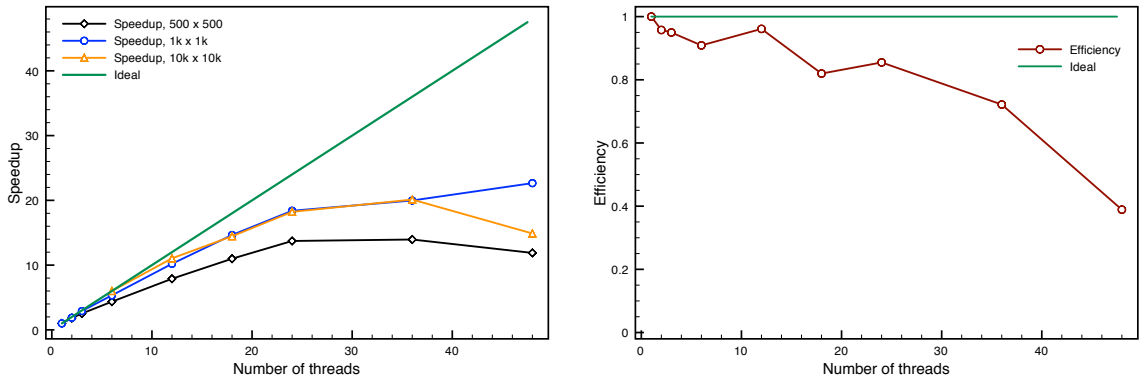

Figure 6: Strong scaling (left) and weak scaling (right). Used 1, 2, 3, 6, 12, 18, 24, 36 and 48 cores. $\delta t = 10^{-8}$, 1000 steps.

d) How can you improve the performance of your code? Explain your reasoning.

As the bottleneck of the code is memory bandwidth, we have to improve the memory accesses. For instance, one should improve locality as most of the memory accesses are done in large strides. NUMA related issues should also be studied and identified as each thread, in the present version of the code, accesses data in all NUMA nodes.

Summary

Summarize your answers, results and plots into a PDF document. Furthermore, elucidate the main structure of the code and report possible code details that are relevant in terms of accuracy or performance. Send the PDF document, source code and related movies to your assigned teaching assistant.