# Introduction to Parallel Computing. Homework 1: Implicit parallelism techniques and performance models.

October 9, 2023

**Abstract**

This assignment is intended to evaluate your effort, knowledge, and originality. Although you may discuss some issues related to the proposed problems with your classmates, each student should work independently on their assignment and write a short report. The methodology used, the quality of the deliverable, and its originality (based on the state of the art and other solutions) will be assessed.

**Instructions**. Write a short report that summarises the solution to each problem. Use the UNITN cluster to run experiments. Try always to use the same type of machine (CPU type and memory). Send the document and the link to the git repository to us by **October 25, 2023**, at 23.59 at the following emails: `flavio.vella@unitn.it`, `laura.delrio@unin.it` and `anas.jnini@unitn.it`. The repository for each exercise must contain in the README the instructions to reproduce the results (instructions for the compilation, the code, and the script to run the experiments.).

## 1 Array addition and vectorization

Consider two arrays, $A$ and $B$, of random numbers of size $n$.

Task 1. Write a C/C++ sequential program and the related routine, namely *routine1*, that computes the element-wise addition of ***floats*** and stores the result in an array $C$. Use a wall-clock time, at the program level, that measures the time of the *for loop* only. The size $n$ should be taken by the program as an input parameter.

Task 2. Write a function, *routine2*, that utilizes *implicit parallelism techniques* like vectorization or software prefetching to improve performance[1].

Task 3. Benchmark and compare the performance of the three routines, assuming that $n$ can vary from $2^4$ to $2^{22}$. Summarize the results in a plot showing on the x-axis the size of the arrays and on the y-axis the time of each routine[2].

Task 4. Write a short document describing the solution implemented, the performance, and comment on the results you have obtained.

## 2 Matrix copy via block reverse ordering

Consider a matrix $M$ of size ($n$-by-$n$). To simplify, $n$ should be a power of two.

Task 1. Write a C/C++ sequential program and the related routine, namely *routine1*, that takes the matrix $M$ (use again random numbers to initialize it), its size $n$ and a parameter block dimension $b$ (integer) such that the matrix can be split into $b$-by-$b$ blocks. Copy each block and its related

[1]Notice that you can also play with compilation flags to enable vectorization. In this case, you should measure the performance of *routine1* on two different compilations and executions (with and without flags).

[2]Check on Lecture 5 - Performance Models, the best practice to benchmark programs.

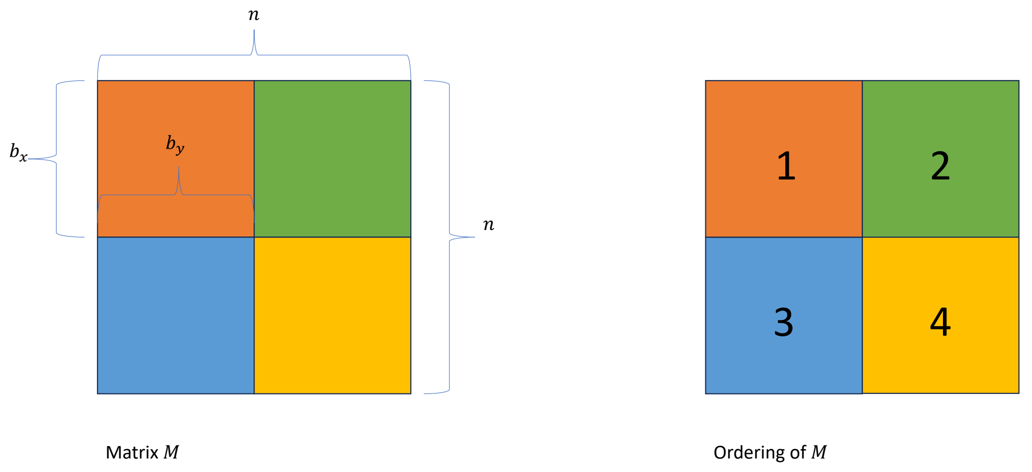

Figure 1: $M$ of size $n$-by-$n$. The block is given by the $b_x$ and $b_y$ dimensions. We assume row-major ordering

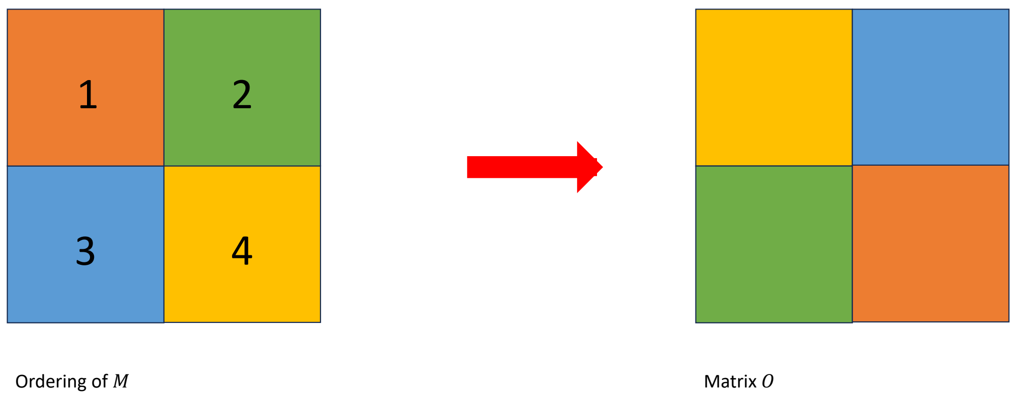

Figure 2: The output matrix $O$ after the application of *routine1*.

elements, in the reverse order (assume row-major ordering) in the output matrix $O$ of the same size. Figure 1 and 2 show an example. Use a wall-clock time, at the program level, that measures the time of the *for loop* only.

Task 2. Write a function, *routine2*, that utilizes *implicit parallelism techniques* like vectorization or software prefetching to improve performance[3].

Task 3. Benchmark and compare the performance of the two routines assuming that $n$ is equal to $2^{12}$, while $b$ can vary from $2^2$ to $2^8$. Summarize the results in a plot showing on the x-axis the size of the block size and on the y-axis the effective bandwidth of each routine[4].

Task 4. Write a short document, describing the solution implemented, the performance, and comment on the results you have obtained.

Task 5. (*extra-bonus task*) Answer the following question. How far is your routine from the peak? Visualize in the performance plot the peak of the system.

[3]Notice that you can also play with compilation flags to enable vectorization. In this case, you should measure the performance of *routine1* on two different compilations and executions (with and without flags).

[4]Check on Lecture 5 - Performance Models, the best practice to benchmark programs.