

Lab 1 - Parallel and Vectorisable Code

Advanced Architectures

University of Minho

The Lab 1 focus on the development of efficient CPU code by covering the programming principles that have a relevant impact on performance, such as cache usage, vectorisation and scalability of multithreaded algorithms. Submit jobs to the mei queue in SeARCH to measure code execution times, but be careful to always use the same node architecture (i.e., compute-6xx or compute-7xx).

This lab tutorial includes one homework assignment (HW 1.1), three exercises to be solved during the lab class (Lab 1.x) and suggested additional exercises (Ext 1.x).

A separate compacted folder (lab1.zip) contains the template for an example code (a squared integer matrix-matrix multiplication, and a derived sample irregular workload) and a simple script to measure the code execution times. To load the compiler in the environment use one of the following commands:

GNU Compiler: `module load gcc/5.3.0`

Intel Compiler: `source /share/apps/intel/parallel studio xe 2019/ compilers and libraries 2019/linux/bin/compilervars.sh intel64`

PAPI: `module load papi/5.4.1`

Remember that this must be done inside a script if you are not using the compute node interactively. All performance measurements for the entire session must be documented and plotted in the provided spreadsheet, otherwise the exercises will be considered incomplete.

1.1 Efficient Cache Usage

Goals: to develop skills in code profiling, common optimisation techniques, and efficient cache usage.

HW 1.1 Study the efficiency of the sequential matrix multiplication code `regularMatrixMult` by assessing the impact of the optimisations studied in previous classes (see the functions from line 353 on and optimise them according to the comments), which are available on the code file. Measure and document (on a spreadsheet) the execution of each optimisation for a matrix size that fits in the L1, L2 and L3 caches, and in RAM (by modifying the `#define SIZE` clause in the code). Include the necessary PAPI code (see the comments in the `main` function in the code - choose only the best result from all repetitions) and measure the L2 and L3 miss rates for these matrix sizes.

Lab 1.1 Measure the L1 miss rate and the amount of load and store instructions for the two larger matrix sizes. What behaviour differences can you observe between these two matrix sizes, according to all recorded metrics?

1.2 Vectorisation

Goals: to develop skills in vector report analysis and optimisation.

Lab 1.2 Compile the provided code with the original supplied matrix-matrix multiplication function. Use either Intel or GNU compilers (Intel is strongly recommended as its report is easier to understand), with the respective vectorisation flags. Do not forget to add a flag to request a full report on the vectorisation results.

Complete the provided code with a new version of the matrix multiplication function, containing the necessary modifications to the code and adding `pragma` clauses to aid the compiler to generate vector code. Analyse the performance to assess the impact of the optimisations.

GNU Compiler: `-O2 -ftree-vectorize -fopt-info-vec-all`

Intel Compiler: `-O2 -qopt-report=2 -qopt-report-phase=vec`

1.3 Performance Scalability

Goals: to comprehend the concepts restricting performance scalability of multithreaded algorithms.

Lab 1.3 Consider two similar synthetic parallel algorithms, one with regular and the other with irregular workloads. It is not necessary to analyse the algorithms or the code. Assess the scalability of these algorithms when using static and dynamic workload distributions (functions `(ir)regularWorkload(Static)Dynamic`) for several number of threads and a matrix size that does not fit in the cache. You only need to call the `regular` or `irregular` functions, which will use a static or dynamic scheduler and execute the respective function from `(ir)regularWorkload(Static)Dynamic`. This is set through the environment variable `DYNAMIC` that by having the string "yes" (`export DYNAMIC=yes`) and recompiling the code, both `regular` and `irregular` functions will use a dynamic scheduling strategy. If this is not set these functions will use a static scheduler.

Which scheduler is best fit for each type of workload? Plot the results using a column chart for 1, 2, 4, 8, max #cores, 1.5x max #cores, 2x max #cores, 3x max #cores and 4x max #cores.