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 vectorisation, parallelisation, and scalability. Use either `compute-641-15` or `compute-641-16` nodes, each with dual Intel Xeon E5-2650 @ 2.60GHz CPU, to measure code execution times.

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 scripts to adequately measure the code execution times and to plot/visualize the results.

To load the compiler in the environment use the one of the following commands:

GNU Compiler: `module load gnu/4.9.0`.

Intel Compiler: `module load gnu/4.7.0 && module load intel/2013.1.117`.

If you want to switch compilers execute `module purge` before loading the compiler to use.

1.1 Vectorisation

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

HW 1.1 Compile the provided code with the supplied matrix-matrix multiplication function. Use either Intel or GNU compilers (Intel is strongly recommended), with the respective vectorisation flags. Do not forget to add a flag to request a full report on the vetcorisation 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 -vec-report3`.

1.2 Parallelisation with OpenMP

Goals: to develop skills in advance usage of OpenMP parallelisation `pragma`.

Lab 1.2 Parallelise with OpenMP the code developed in the previous exercise. Use the provided scripts to measure its performance. Evaluate the impact on performance due to the individual (or combined) `pragma` clauses below. Justify how/why they improve/worsen the algorithm execution time.

nowait: disables the default synchronisation of threads at the end of each parallel loop iteration.

schedule: describes the work sharing technique among threads; It can be defined as static, dynamic, guided, runtime, and auto, and each heuristic is best suited for specific characteristics of algorithms, such as their irregularity or task granularity.

Extend the functionality of the algorithm to add (accumulate) all computed values at each element of the resulting matrix (a simple addition outside the k loop). Implement this feature using either (i) a shared variable among all threads or (ii) a thread-private variable with a final reduction.

Beware of the thread concurrency when accessing shared resources (see OpenMP `critical` and `atomic` constructs)!

Ext 1.2 Repeat **Lab 1.2** with the original matrix multiplication function. Comment the results (to be solved at home, after the lab session).

1.3 Performance Scalability

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

Lab 1.3 Consider the parallel matrix-matrix multiplication code from the previous exercise. Measure the algorithm execution time for the different number of threads supplied in the script.

Does the application performance always improve with the increased number of threads?

Which is the configuration that provides the highest efficiency (best ratio between performance and number of threads)?

Justify your answers and provide possible solutions.

Ext 1.3 Modify the Perl script to plot the performance values so that it generates an efficiency plot instead of the speedup one (to be solved at home, after the lab session).