Lab 1 - Profiling Parallel Code (1^{st} part)

Advanced Computer Architectures

University of Minho

The Lab 1 focus on the development of efficient CPU code by revisiting the programming principles that have a relevant impact on performance, such as 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., the nodes in the *cpar* partition).

This lab tutorial includes three exercises to be solved during the lab class (Lab 1.x) and suggested additional exercises (Ext 1.x).

Consider that a solution for the shared queue master-worker code from the previous lab session is available in the SeARCH cluster. You can copy it to your home directory by:

```
cp -r /share/cpar/ACA/lab1 .
```

To load the compiler in the environment use one of the following commands (GCC is advised):

Intel Compiler: `module load oneapi/intel_ippcp_intel64/latest`

GNU Compiler: `module load gcc/11.2.0`

Remember that this must also be done inside a script if you are not using the compute node interactively. All performance measurements for the entire session must be documented.

1 Profiling with VTune

Goals: to develop skills in the fundamentals of thread programming.

Lab 1.1 Consider the Social Media Analyser code developed in the previous lab. You should have two versions of the code following the master-worker pattern: (i) a shared work queue, where the master adds work to that queue and workers compete to process that workload; (ii) private work queues, where the master adds work to the queue of each individual worker.

If you do not have both versions working, finalise their development before moving to the next exercise. The folder provided in the introduction contains an implementation of (i), as well as `job.sh` script. Ensure that the code compiles and executes on the SeARCH cluster.

Submit the binary to be executed on the cluster using the `job.sh` script (understand it well, you will need to adapt it later!):

```
sbatch job.sh
```

The outputs should appear on two files, once the job is executed: slurm-*job-id*.out and slurm-*job-id*.err. You can check the status of your job through the command `squeue -u username`.

Lab 1.2 Install Intel VTune on your system. It will only be used to visualise the data gathered at SeARCH, and any other system you may use in the future. Download it at:

```
https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtuneprofilerdownload.html
```

Perform some tests to assess a dataset size (i.e., the number of users in the network) that allows the code to run with a single master and worker for at least 5 seconds. The runtime has to be substantial enough to maximise the possible gains from increasing the number of workers in the following exercises.

Lab 1.3 Create two executables of version (ii) from the previous exercise, resultant from different compilation options:

```
gcc -Wall -O2 -march=ivybridge -g -o sma_O2 ...
```

```
gcc -Wall -O3 -march=ivybridge -g -o sma_O3 ...
```

Use VTune to profile both executables for a dataset size that does not fit in cache. Do not forget to load the necessary modules in both the frontend and compute nodes:

```
module load gcc/11.2.0
```

```
module load vtune/2020
```

Profile both executables using VTune for a single thread (sequential) and 10 threads. Choose VTune's analysis type that allows the identification of hotspots on the code.

```
vtune -collect ???  -result-dir sma_???_O2  ./sma_O2 exec_options
```

Copy the results folder to your own laptop or use the frontend to visualise the outputs:

On your system: `scp -r user@s7edu.di.uminho.pt:~/???/sma_???_O2 .`

Using the frontend: `vtune-gui /???/sma_???_O2` (really not advisable)

Does the code use the available CPU cores adequately? Can you identify any bottleneck from your analysis? If the code appears limited by access to data (i.e. probably not limited by CPU power), rerun an adequate VTune analysis and characterise how the application interacts with memory

Lab/Ext 1.4 Perform a scalability analysis of the performance of the code with 2, 4, 8, 10, 15, and 20 threads (think about the reason for choosing these #threads). To achieve this, you should plot the speedups (given #threads *vs* a single master and worker) on a graph for the version (ii) of the code.

Does the code scale as expected? What was your expectation for the performance improvement and why?

2 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 options passed to the executable.

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.

Ext 1.3 Identify and characterise the bottleneck limiting the performance of the irregular workload using the dynamic scheduler for 8 threads. Perform this analysis using the VTune profiler, considering the most adequate analyses to get this information.

Remember to copy the results folder, `vtune_results` to your laptop and visualise the results with your own installation of VTune.