EPFL

Parallel and High Performance Computing

Dr. Pablo Antolín

Series 2

March 6 2025

Measuring CPU performances

Exercise 1: ***Theoretical analysis: Amdhal's and Gustafson's laws***

Recall the 2D Poisson solver from the Exercise 6 of Series 0. We now assume a very simple parallelization strategy for the solver:

Each processor i gets N/p lines of the grid, where p is the total number of processors, and N is the total number of lines in the grid. We consider sd_i the subdomain of size $N \times \frac{N}{p}$ belonging to processor i. Processor 0 is in charge of the initialization stage. Afterwards, at each iteration k the algorithm is:

a) Compute a step of the Jacobi solution on sd_i.

b) Send the last line of sd_i to processor $i+1$ and the first line of sd_i to processor $i-1$.

c) Receive the last line of sd_{i-1} from processor $i-1$ and the first line of sd_{i+1} from processor $i+1$.

d) Compute the local L^2-norm.

e) Send the local L^2-norm to processor 0.

f) If $i = 0$, compute the global L^2-norm, by suming all the local L^2-norms, and send it to all processors.

g) Receive the global L^2-norm from process 0.

h) Compare with espilon. Exit if reached.

Note in the second and third steps, adjustments should be required for the first and last subdomains.

Answer the following questions:

a) What is the definition of the Amdahl's law (with the serial part $1-\alpha$ and the serial execution time T_1)? What does this law measure?

b) What is the definition of the Gustafson's law (with the non-parallelizable part $1-\alpha(N)$ and problem size N)? What does this law measure?

c) Considering the algorithm above and information gathered in Table 1, provide an estimation of $1 - \alpha$, the serial part of the code that can not be parallelized.

d) What is the upper bound of the speedup according to Amdahl's law?

e) What would be the maximum efficiency with 128 processors?

Exercise 2: ***Theoretical roofline***

Run the following command (in a single node) in `helvetios`:

```
$> srun -n 1 --qos=math-454 --account=math-454 cat /proc/cpuinfo
```

Determine:

a) The theoretical peak performance of a single core.

b) The theoretical performance of the memory.

c) The roofline model, in particular the ridge point.

Note: You will need to check some information on https://en.wikichip.org

N	t_{total}	t_{init}	n_{steps}	t_{step}
128	0.003	0.000	109	0.00003
256	0.496	0.004	8194	0.00006
384	1.083	0.011	7445	0.00015
512	2.242	0.021	8238	0.00027
640	4.215	0.043	8234	0.00051
768	5.500	0.051	6088	0.00090
896	7.736	0.078	5655	0.00137
1024	15.246	0.153	7395	0.00206
1152	23.317	0.234	7450	0.00313
1280	34.051	0.341	7724	0.00441
1408	40.636	0.407	8371	0.00485
1536	75.720	0.758	13049	0.00580
1664	87.591	0.874	13222	0.00662
1792	85.202	0.851	13255	0.00643
1920	107.046	1.071	13652	0.00784
2048	126.070	1.261	13964	0.00903
2176	132.905	1.329	13521	0.00983
2304	236.203	2.361	18680	0.01264
2432	246.091	2.461	18814	0.01308
2560	258.232	2.582	17515	0.01474

Table 1: Time measurements of a 2D Poisson solver. N is the size of the problem (grid size = $N \times N$); t_{total} is the total time (in seconds); t_{init} is the initialization time (in seconds); n_{steps} is the number of iterations steps to reach an espilon of 0.005; t_{step} is the time (in seconds) per iteration.

Exercise 3: ***Measured roofline*** Let's now measure the CPU and memory performances using some tools.

a) Compile and run the code in `Stream` to compute the sustained memory performance (consider the minimum value which you see).

b) Compile and run the code in `Dgemm` to compute the sustained peak performance.

c) Compute the roofline model, in particular the ridge point.

How to compile:

```
$> module load intel
$> module load intel-oneapi-mkl
$> cd <FOLDER>
$> make
```

How to run:

```
$> module load intel
$> module load intel-oneapi-mkl  # only when running ./dgemm
$> export KMP_AFFINITY=compact
$> export granularity=fine
$> export OMP_NUM_THREADS=1
$> srun -n 1 -N 1 --qos=math-454 --account=math-454
↪  --cpus-per-task=$OMP_NUM_THREADS ./stream
$> srun -n 1 -N 1 --qos=math-454 --account=math-454
↪  --cpus-per-task=$OMP_NUM_THREADS ./dgemm
```

Compare the results using `OMP_NUM_THREADS=8`.

Exercise 4: ***Jacobi stencil*** We want to solve the Jacobi stencil:

```
u(i,j) = 1/4*( u(i-1, j) + u(i+1, j) + u(i, j-1) + u(i, j+1) )
```

a) What is the arithmetic intensity (AI) of this equation?

b) According to the roofline model, what is the maximum performance we can get?

Go to the directory `Jacobi`:

- `jacobi.c` is the main driver.
- `jacobi-naive.c` is the "classic" implementation.

Compile and run this code using the `Makefile`. Launch it as:

```
$> module load intel
$> export KMP_AFFINITY=compact
$> export granularity=fine
$> export OMP_NUM_THREADS=1
```

```
$> srun -n 1 -N 1 --qos=math-454 --account=math-454
↪  --cpus-per-task=$OMP_NUM_THREADS ./jacobi-naive 1000
```

where `1000` is the number of points per direction of the stencil. Repeat the same operation, but using 8 threads instead of 1. Repeat it again, but using `10000` points per direction. Consider the median value over the iterations, hence discarding possible outliers in the performance values which you obtain.

c) Report the computed performance for all the cases.

d) What are the differences between the expected maximum performance and the obtained ones? How can these differences been explained?

Exercise 5: (if you finish early...) The files `jacobi-sse.c` and `jacobi-avx*.c` containt different implementations using DLP with intrinsics. Compile and run them in the same setting as in the previous exercise. Consider as well the executable `jacobi-naive-auto-vec`.

a) Which one is the fastest and why?

b) Can you beat those implementations?