Lab 4: Getting started with parallel performance metrics

authors: Arias Rodrigo, Burca Horia
date: 12/11/2018

Task 4.1

Populate the Table 4.1 and analyze the results for the program that estimates the number π.

We slightly modified the table to include the mean and std of 200 runs, measuring the running time of the program.

Table 4.1 OpenMP execution time of the program that estimates the Pi number

nr_threads	runs	mean_time(ms)	std_time(ms)
2	200	136.260248	0.540395
4	200	68.743345	1.807676
8	200	34.402916	2.805065
16	200	17.196037	0.919341
32	200	8.641697	0.534061

We can observe that with OpenMP, by increasing the number of threads used, the mean execution time decreases significantly.

Task 4.2

Populate the Table 4.2 and analyze the results for the program that estimates the number π. Compare the results from Table 4.1 and Table 4.2.

Table 4.2 MPI execution time of the program that estimates the Pi number (one node)

nr_threads	runs	mean_time(ms)	std_time(ms)
2	200	136.010271	1.422983
4	200	67.964889	0.698149
8	200	34.164717	1.013572
16	200	16.987933	0.360736
32	200	8.511111	0.267816

We see that with MPI as the number of processes p doubles, the average running time almost halves.

The results are similar to what we observed in the OpenMP version (Table 4.1).

Task 4.3

Populate the Table 4.3 with 16 nodes and analyze the results.

Table 4.3 MPI execution time of the program that estimates the Pi number (16 nodes, one process

per node)

task/node	p	runs	mean_time(ms)	std_time(ms)
1	16	200	261.763766	50.791005
16	16	200	16.989603	0.478375

For the execution, srun was used with the number of nodes and tasks per node:

```
% srun -N 1 --ntasks-per-node 16 mpirun./pi
% srun -N 16 --ntasks-per-node 1 mpirun ./pi
```

Task 4.4

Compare and analyze the results from Table 4.3 and Table 4.2.

We see that using multiple nodes increased the time one order of magnitude. It makes sense, as we need to send the data over the network to the other nodes, and send the results back. Compared with only shared memory, we see a big difference.

With respect to the openmp results, we see similar times for the same number of processes, when local memory is used.

Task 4.5

Populate the Speedup table (Table 4.6) for this example and analyze the results. Remark the situations for n = 65536 and p increases. Justify the behaviour of the results.

Notice! For tasks 4.5 and 4.6 we have used a different sequential time value for a problem size of *n=16777216*, namely:

$T_{seq,\ n\ =\ 16777216}$ = 56156.

This ensured consistent results for speedup and efficiency.

Table 4.6 Speedups for the Parallel MPI version of the trapezoidal rule for Pi number

p \ n	65536	1048576	16777216	268435456
2	1.946	2.014	1.987	2.055
4	1.685	3.431	3.886	4.031
6	1.387	5.414	7.366	7.925
8	0.839	5.675	13.342	14.924

We can observe that in general the speedup increases with the number of processors. However, in the case of *n = 65536* the speedup actually decreases when we increase *p*. This indicates that for small problem sizes the overhead of spawning more processes is too important compared to the parallel execution time thus decreasing the speedup.

Task 4.6

Populate the efficiencies table (Table 4.7) for this example and analyze the results.

Table 4.7 Efficiencies for the Parallel MPI version of the trapezoidal rule for Pi number

p \ n	65536	1048576	16777216	268435456
2	0.973	1.007	0.994	1.028
4	0.421	0.858	0.971	1.008
6	0.231	0.902	1.228	1.321
8	0.105	0.709	1.668	1.865

A good parallelization is characterised by an efficiency close to *1*. In this example the best results are the ones with a bigger problem size. A super-speedup situation happens with $p=2$ and $n=1048576$, $p=\{6,8\}$ and $n=16777216$, and $p=\{2,4,6,8\}$ and $n=268435456$.

Task 4.7

According Tables 4.9 and 4.10 for this example, justify why for this parallel version on OpenMP we can say that this parallel algorithms is strongly scalable only for a big size problems, and for small size the algorithm does not scale.

We know that a program is strongly scalable if the efficiency stays fix as we increase the number of processes. We can see that in this example for $n=268435456$ we do indeed have a constant efficiency (close to *1*) as we increase the number of processes. However for smaller problem sizes if we increase the number of processes, we see the efficiency dropping.