A1: Weak/Strong Scalability

Francesca Cairoli

All the executions have been performed on the SISSA Ulysses cluster.

Introduction

Strong scaling

In parallel computing the speedup is defined as: $speedup(n) = T(1)/T(n)$, where $T(1)$ is the computational time for running the program using one processor and $T(n)$ is the computational time for running the same program with n processors. Ideally, we would like the program to have a linear speedup, that is, equal to the number of processors, i.e. $speedup(n) = n$. However, it is very hard to reach this goal for real applications. In particular, Amdahl pointed out that the speedup is limited by the fraction of the serial part of the software that is not parallelizable. The Amdahl's law can be formulated as: $speedup(n) = \frac{1}{(s+p/n)}$, where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and n is the number of processors. In practice, Amdahl's law states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code. This is called strong scaling. The parallelization efficiency is defined as $\epsilon(n) = speedup(n)/n$. Therefore, we notice how the parallelization efficiency decreases as the amount of resources increases. For this reason, parallel computing with many processors is useful only for highly parallelized programs.

Weak scaling

The Amdahl's law gives the upper limit of speedup for a problem of fixed size, whereas Gustafson pointed out that in practice the sizes of problems scale with the amount of available resources. If a problem only requires a small amount of resources, it is not beneficial to use a large amount of resources to carry out the computation. A more reasonable choice is to use small amounts of resources for small problems and larger quantities of resources for big problems. This is known as Gustafson's law and it is based on the approximations that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem. The formula for the scaled speedup is $scaledspeedup(n) = s + p \times n$, where s, p and n have the same meaning as before. With Gustafson's law the scaled speedup increases

linearly with respect to the number of processors (with a slope smaller than one), and there is no upper limit for the scaled speedup. This is called weak scaling.

Results

The first assignment was about studying the scalability on a toy HPC application, in particular a Monte-Carlo integration of a quarter of a unit circle to compute the number π. The basic implementation of the algorithm was given (file `pi.c`) as well as the parallel MPI implementation (file `mpi_pi.c`). The exercise consists in analyzing how well this application scales up to the total number of cores of one node.

First of all, we were asked to determine the CPU time required to calculate π with the serial code using 10 millions iterations. It takes around 0.196 seconds. We also run the same code for larger numbers of iterations. The execution times grows almost linearly: it takes around 1.97 seconds for $Niter = 10^8$ and around 19.77 seconds for $Niter = 10^9$. After that we run the MPI code for the same number (10 millions) of iterations.

We keep the problem size constant, i.e., $Niter = 10$ millions, and run the MPI code with only one processor. We compare it with the serial code in order to quantify the overhead associated with MPI. It results that the parallel execution can improve the performaces only for very large problem sizes, since the overhead is approximately one second.

After that we run the MPI code for an increasing number of MPI process, from 2 up to 20. In order to asses the relation between strong scalability and the number of iterations considered, we executed the same process for different problem sizes. In particular: $Niter = 1 \times 10^5, 5 \times 10^5, 10 \times 10^5$.

As suggested, we compiled the `mpi_pi.c` script with the `mpicc` command and we run it with the `mpirun` command. The time was measured using the walltime already computed inside the `mpi_pi.c` script. This choice is motivated by the fact that we assume the serial workload to remain constant throughout the experiments.

Under the parallel execution, we notice how increasing the number of iterations leads to a better speedup, i.e., graph is a line closer to the linear scaling one.

The weak scalability needs a base measure for the problem size which is then multiplied by n, i.e., the number of processors. The test has been repeated for three different base sizes: $Niter = 1 \times 10^4, 5 \times 10^4, 10 \times 10^4$.

In order to achieve weak scalability we expect the execution time to remain constant when we gradually increase the problem size and the number of processors. Figure 2 shows how the execution time stays more constant if the baseline measure for the problem size is small.

From the plots, shown in Figure 1 and Figure 2, we can conclude that the program scales well both in weak and strong settings. The weak scaling is better reached for small problem sizes, while the strong scaling is better for large problem sizes.

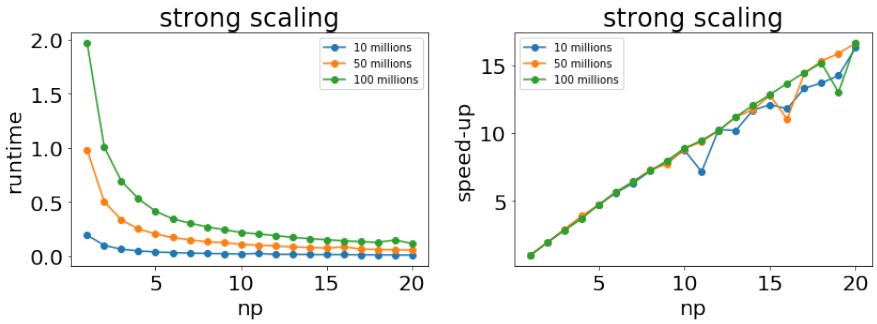

Figure 1: Runtime and speed-up under strong scaling for three different base problem sizes.

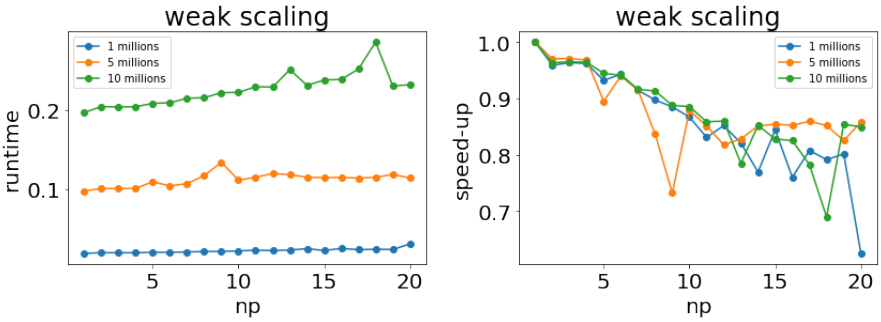

Figure 2: Runtime and speed-up under weak scaling for three different base problem sizes. The fluctuations seems to reduce as the base size increases.

Scalability bash scripts

```
module load openmpi

echo "Strong Scalability: base size is $1"

for ((procs=1;procs<=20;procs++)); do
   echo "Niter_per_CPU =$(($1/${procs})), np=${procs}"
   time mpirun -np ${procs} ./mpi_pi $(($1/${procs}))

done
```

```
echo "Strong scalability test:"

for n in 10000000 50000000 100000000; do
  bash ./strong_one_iter.sh ${n}
done
```

```
echo "Weak scalability: base size is $1"

```

```
for ((procs=1;procs<=20;procs++)); do
   echo "Niter_for_CPU=$1, np=${procs}"
   /usr/bin/time -p mpirun -np ${procs} ./mpi_pi $1

done
```

```
echo "Let's test weak scalability:"

for n in 1000000 5000000 10000000; do
    bash ./weak_one_iter.sh ${n}
done
```