Weak and Strong Scalability of an HPC Application

Gabriele Sarti

November 26, 2018

Abstract

In this exercise we compute the weak and strong scalability for a Monte-Carlo Pi computation based on a quarter of unit circle. We are provided with a serial and an MPI parallel implementation for the program, and we should see how the application scales to the total number of cores in a node.

1 Introduction

Scalability is defined as the capability of a system, network or process to handle a growing amount of work. In the field of high performance computing, we use two types of scalability to define the performance of a system on an application:

- Strong scalability $\implies$ How the performance of the system increases by increasing the number of processors and keeping the problem size fixed.
- Weak scalability $\implies$ How the performance of the system remains constant by increasing both the number of processors and the problem size.

Strong scalability is quantifiable through the Speedup S, which is computed as follows:

$$S = \frac{T_1}{T_N}$$

Where T_1 is the problem execution time on a single processor and T_N is the problem execution time on N parallel processors. A high value of S with respect to N is an indicator of strong scaling, although $S = N$ could not be achieved due to Amdahl's Law [1].

Weak scalability is present if the execution time remains approximately constant, implying that the execution time of the serial part of the code also remain constant, as the problem size grows. There could be minor variations due to an increased communication time, but this generally holds true thanks to Gustafson's Law [2].

2 Execution

2.1 Serial Execution

We start by determining the CPU time required to compute Pi with the serial code contained in the file pi.c using a parameter $N = 10'000'000$ (ten millions), which represents the number of iterations of the Monte-Carlo process.

We performed 10 test runs of the serial program with the given N parameter running the command

```
/usr/bin/time ./serial_pi 100000000
```

on execution node cn06-23 through qsub interactive mode. Thanks to my results, we found that the average time of execution is quite stable at 0.19s, and that the generated value of Pi in the pi.dat file differs by the true value by an approximated average of 0.0002. Since the execution time is quite short, we also tried to run the same code for other values of N, namely 10^8 and 10^9. Our results show that for those values execution time scales almost perfectly (1.97 for 10^8, 19.77 for 10^9).

2.2 Parallel Execution

We set up the execution of the parallel version of pi.c by executing the following commands on node cn06-23:

```
# Load mpi module
module load openmpi
# Compile the code in parallel
$ mpicc mpi_pi.c -o parallel_pi
# Run the code with mpi
mpirun parallel_pi 500000
```

Listing 1: Setting up parallel execution

Since we didn't specify the number of processes taken by mpirun, it will take all the available ones on the node. In order to compare the performance in the three cases considered before, we use $5 \times 10^6, 5 \times 10^7$ and 5×10^8 as parameters of parallel_pi since the final number of stone throws is scaled with respect to the number of processors used in this case (Ex. $5 \times 10^6 \times 20 = 10^7$). We obtained the following results:

- For $N = 5 \times 10^6$, running on 20 processors (10^7 trials), we have an average execution time of 1.88s, which is almost ten times slower than the serial execution for this problem size.

- For $N = 5 \times 10^7$, running on 20 processors (10^8 trials), we have an average execution time of 1.91s, which is only slightly faster than the serial execution for this problem size.

- For $N = 5 \times 10^8$, running on 20 processors (10^9 trials), we have an average execution time of 2.94s, which is almost seven times faster than the serial execution for this problem size.

It is self-evident how a parallel execution could be very beneficial to improve execution time, but only for very large problem sizes since the MPI library adds an overhead of approximately one second to normal execution.

3 Scaling

3.1 Strong scaling

To perform a strong scalability test, we kept the problem size constant at 10^8 iterations, while increasing the number of processors gradually. We also executed this process with different problem sizes in order to assess the relation between strong scalability and number of iterations for this program. For the number of MPI processes, we used the ones indicated in the assignment. We performed five runs for each combination of iterations and MPI processes and calculated the average speedup to reduce the variability error of our results. The execution script is available in Appendix A

Execution results are presented in Table 1. Also, Figures 1 and 2 provide a visualization of our results with respect to speedup and runtime.

Table 1: Strong scaling. $T(N)$ represents the average execution time for the problem size N over five runs, without MPI overhead. S is the speedup. NProc is the number of MPI processes used for the execution.

NProc	T(10^7)	S(10^7)	T(10^8)	S(10^8)	T(10^9)	S(10^9)
1	0.196 s	1.00	1.962 s	1.00	19.656 s	1.00
2	0.101 s	1.94	1.031 s	1.90	10.104 s	1.95
4	0.051 s	3.84	0.506 s	3.87	5.058 s	3.88
8	0.028 s	7.00	0.268 s	7.32	2.682 s	7.32
16	0.016 s	12.25	0.145 s	13.53	1.427 s	13.77
20	0.013 s	15.08	0.116 s	16.91	1.148 s	17.12

From the results and the graphs we obtained it can be noticed that increasing the number of iterations leads to a better speedup using parallel execution, resulting in a line in the plot which is closer to the one representing a linear scaling. Ideally the speedup results should be presented with error bounds, but we decided not to include them given our time limitations and the fact that they were not mandatory.

Figure 1: Strong scaling speedup for Monte-Carlo Pi approximation with parallel execution.

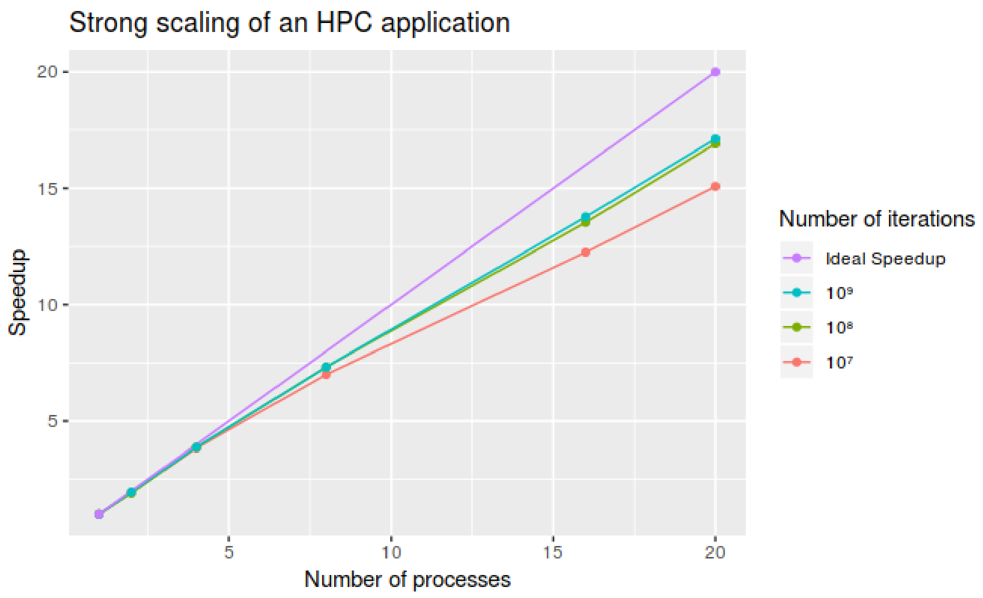
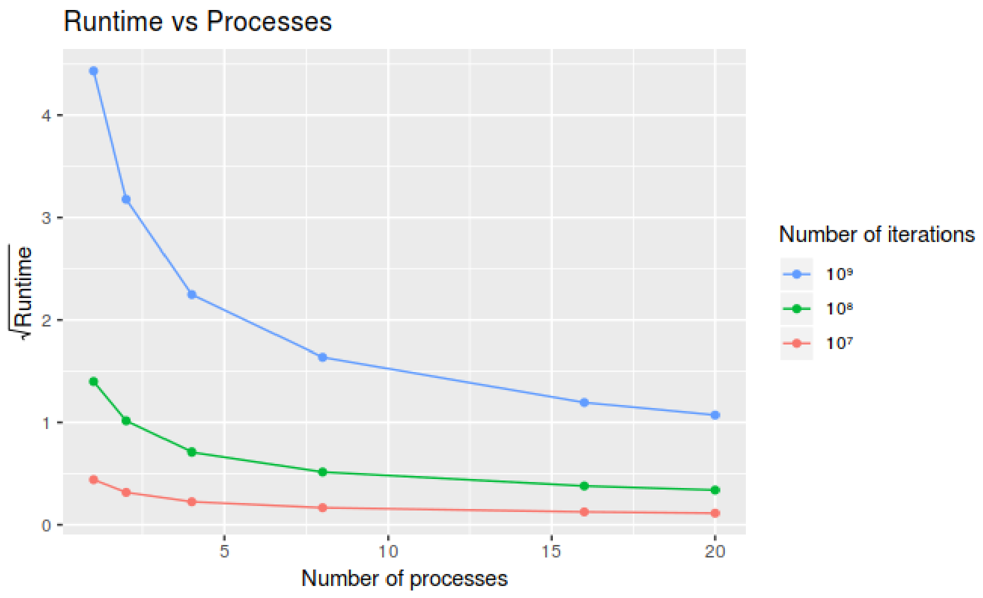


Figure 2: Root of runtime versus number of processes plot shows the benefits of running a parallel application.



3.2 Weak scaling

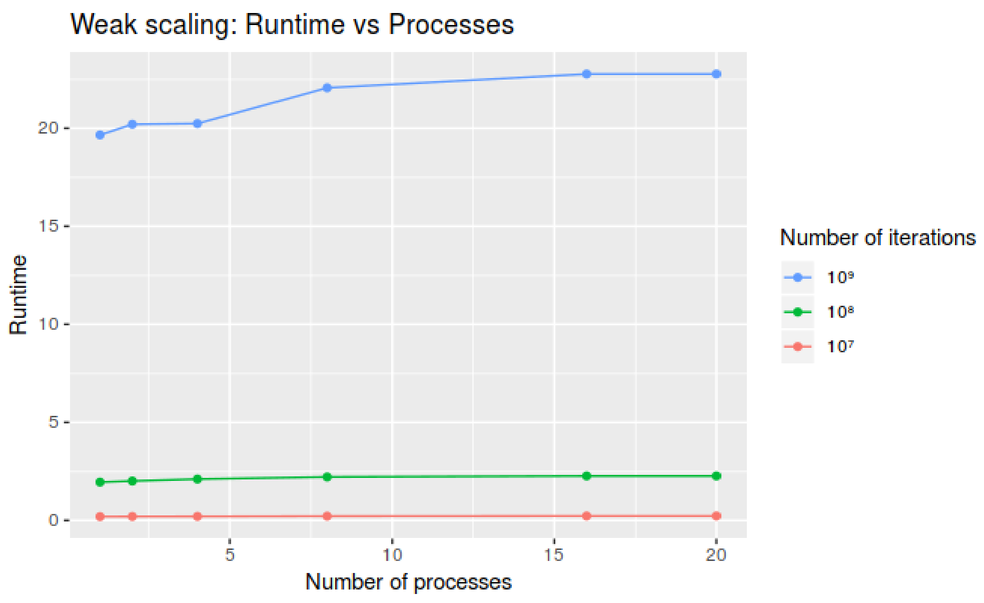
To perform a weak scalability test we increase the both the problem size and the number of processors gradually, to observe if execution time remains constant. As for the previous scalability test, we tested problem sizes of 10^7, 10^8 and 10^9 iterations. We used the same number of MPI processes. We performed five runs for each combination of iterations and MPI processes and calculated the average runtime to reduce the variability error of our results. The execution script is available in Appendix B

Execution results are presented in Table 2. Also, Figure 3 gives us an idea of the weak scalability of our application.

Table 2: Weak scaling. $T(N \times \text{NProc})$ represents the average execution time for the growing problem size N over five runs, without MPI overhead. NProc is the number of MPI processes used for the execution.

NProc	T(10^7×NProc)	T(10^8× NProc)	T(10^8× NProc)
1	0.196s	1.962s	19.656s
2	0.202s	2.020s	20.202s
4	0.206s	2.119s	20.240s
8	0.220s	2.228s	22.071s
16	0.229s	2.279s	22.780s
20	0.230s	2.279s	22.781s

Figure 3: Squared runtime versus number of processes plot shows the benefits of running a parallel application.



For an ideal weak scaling, we should see that the execution time remains constant as the problem size and the number of processes increase. From the results and the graphs presented above we can see that the tested application shows a behavior which is similar to weak scaling, especially for small problem sizes and for large number of processes. We decided not to provide error bounds given the fact we already provided an average value for five runs. As in the previous test, the execution time was the one provided by the program and didn't include the overhead associated with including the MPI library.

4 Efficiency

In order to compare the results between our weak and strong scaling tests, we computed the efficiency E in both cases. The efficiency for strong scaling is defined as

$$E_S = \frac{S}{\text{NProc}}$$

Where S is the speedup and NProc is the number of processes used to run the application. For weak scalability, efficiency instead is simply

$$E_W = \frac{T_1}{T_N}$$

Where T_1 is the execution time obtained using a single processes and T_N the execution time with N processes.

In both cases, efficiency measures the performance of an application in weak or strong scalability testing with respect to the theoretical peak performance of an application run on the same system. Results are presented in Table 3 and Figures 4 and 5.

Table 3: Efficiency for different numbers of iterations in the Monte-Carlo approximation of Pi. E_W is the efficiency for weak scalability, E_S is the one for strong scalability.

NProc	$E_W(10^7)$	$E_S(10^7)$	$E_W(10^8)$	$E_S(10^8)$	$E_W(10^9)$	$E_S(10^9)$
1	1.000	1.000	1.000	1.000	1.000	1.000
2	0.970	0.970	0.971	0.95	0.973	0.975
4	0.952	0.960	0.926	0.968	0.971	0.97
8	0.891	0.875	0.881	0.915	0.891	0.915
16	0.856	0.766	0.861	0.846	0.863	0.861
20	0.852	0.754	0.861	0.846	0.863	0.856

Figure 4: Efficiency for strong scalability test on Monte-Carlo Pi approximation with parallel execution with respect to the number of iterations.

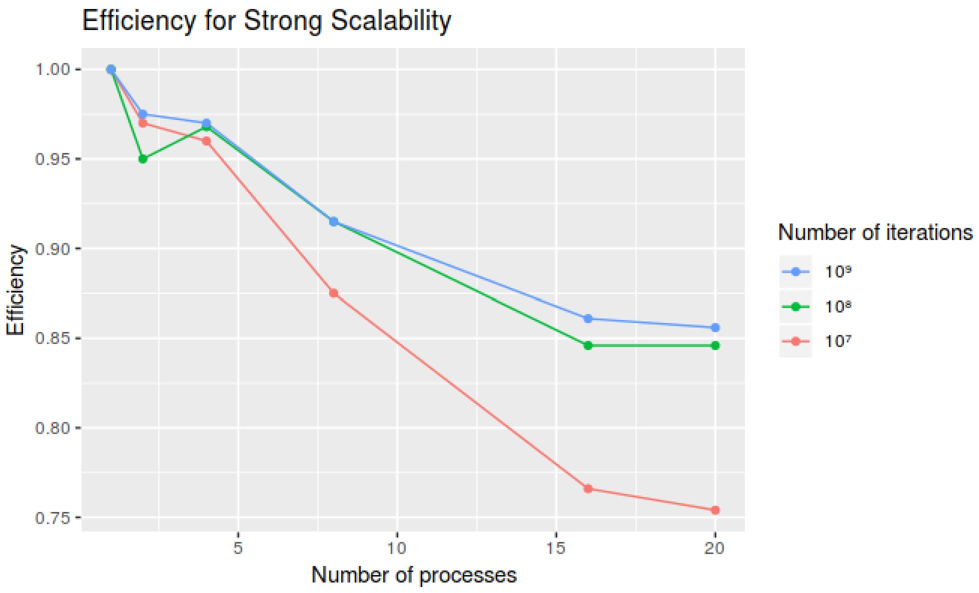
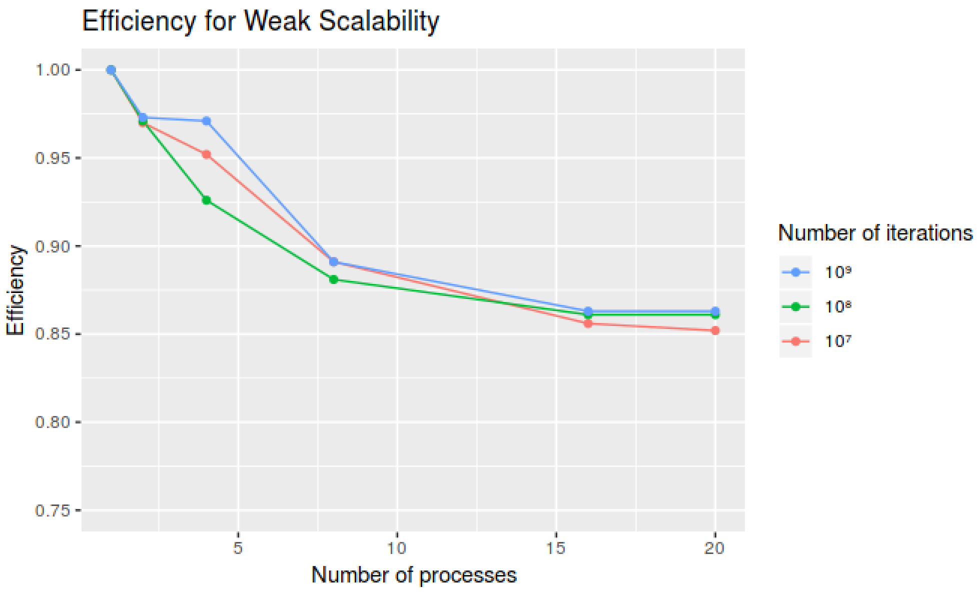


Figure 5: Efficiency for weak scalability test on Monte-Carlo Pi approximation with parallel execution with respect to the number of iterations.



5 Conclusion

5.1 Result Analysis: Weak or Strong Scaling?

From the graphs presented in the previous section, it is evident how we achieve good results for both strong and weak scalability tests performed on the parallel_pi application. Keeping in mind that ideal speedup could not be achieved due to Amdahl's Law, an efficiency of around 85% in most cases can be considered as a very strong indicator of good scalability. Figure 4 gives additional evidence about worse strong scaling performances for small problem sizes, which we already pointed out after performing the test is Section 3.1. For this reason, we can safely assume that our program present a better weak scalability for small numbers of iterations.

References

[1] `https://en.wikipedia.org/wiki/Amdahl's_law`

[2] `https://en.wikipedia.org/wiki/Gustafson's_law`

Appendices

A strong_scaling.sh

```
# Used to compute strong scalability for parallel_pi application
# Moves to the folder containing the application
cd ~/high_performance_computing/exercise1/code/
# Load parallel execution module
module load openmpi
# Execution loop, tests are performed 5 times to obtain an average
for ps in 10000000 100000000 1000000000
do
    for procs in 1 2 4 8 16 20
    do
        for runs in 1 2 3 4 5
        do
            # We divide by procs since the code multiplies the number of
            # iterations by the number of processes used for the execution,
            # and we want to keep problem size fixed instead
          time mpirun -np ${procs} parallel_pi $((ps/procs))
        done
    done
done
exit
```

B weak_scaling.sh

```
# Used to compute weak scalability for parallel_pi application
# Moves to the folder containing the application
cd ~/high_performance_computing/exercise1/code/
# Load parallel execution module
module load openmpi
# Execution loop, tests are performed 5 times to obtain an average
for ps in 10000000 100000000 1000000000
do
    for procs in 1 2 4 8 16 20
    do
        for runs in 1 2 3 4 5
        do
           # The code is already multiplying the number of iterations
           # by the number of processes used for the execution
          time mpirun -np ${procs} parallel_pi ${ps}
        done
    done
done
exit
```