Lab Class High Performance Computing
Prof. Tobias Kreilos

Instruction sheet for lab class 6

Exercise 1: Parallel Jacobi

Download the file jacobi_benchmark.cpp from moodle. Make some tests to see if you can observe the effects of the parallelization when running with a different number of threads.

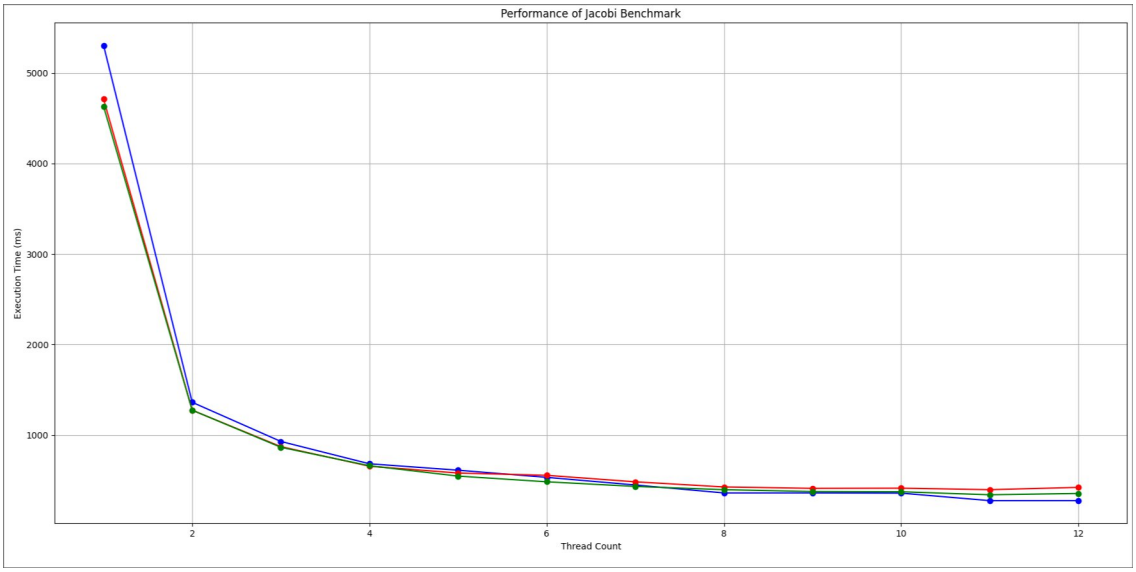
Run the program without changing anything and store the result in the file “ref.asc”.
Imporant: disable storing in “ref.asc” before you start modifying the code!

Activate the verification against the matrix stored in “ref.asc” to make sure you do not break anything when you change the code.

Implement the optimization discussed in the lecture – creating the threads once and not forking and joining for every iteration. Caution, this is not straightforward! Be careful as new variables are becoming shared variables and hence susceptible to data races.
If you get constantly stuck in deadlocks, in a first step try running for a fixed number of iterations, instead of checking the max distance.

Benchmark your solution on the cluster and create a plot of performance and efficiency as a function of thread count, using the original parameters. Try different scheduling strategies.

Insert the plot here.



blue: static, 32 red: dynamic, 32 green: guided

You should be able to observe that performance scales almost linearly up to 6 threads and flattens for a higher thread count. Can you explain the observation?

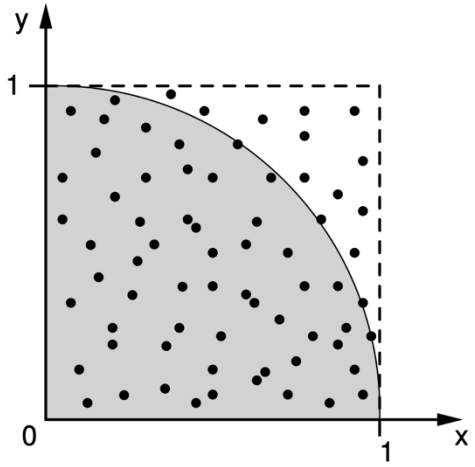
Lab Class High Performance Computing
Prof. Tobias Kreilos

HOCH SCHULE OFFEN BURG

Exercise 2: Computing π with Monte Carlo methods

Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.

For example, consider a quadrant inscribed in a unit square. Given that the ratio of their areas is π/4, the value of π can be approximated using a Monte Carlo method:

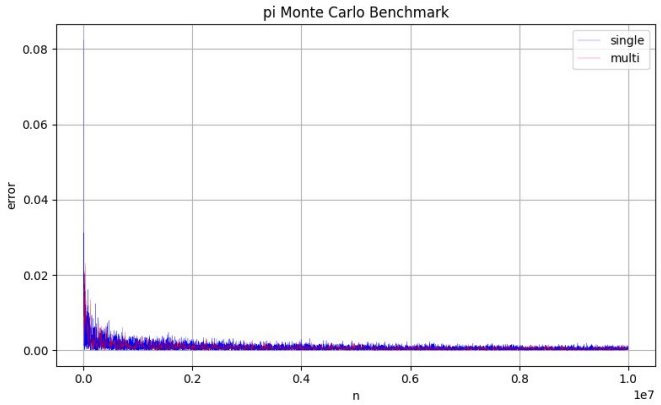


1. Draw a square, then inscribe a quadrant within it.
2. Uniformly scatter a given number of points over the square
3. Count the number of points inside the quadrant, i.e. having a distance from the origin of less than 1.
4. The ratio of the inside-count and the total-sample-count is an estimate of the ratio of the two areas, π/4. Multiply the result by 4 to estimate π. [Wikipedia]

To convert this to an algorithm, generate N pairs of random numbers (x, y) and count the number of pairs r where $x^2 + y^2 < 1$. Then you obtain an approximation $\pi \simeq \frac{4r}{N}$.

Write a program that computes π with this method (on a single core). Generate a plot of the computed approximation of π as a function of N, with N going up to 1e7.
Caution: 10^7 is larger than INT_MAX, so use a ~~int64_t~~ as type for N.

Insert the plot here.

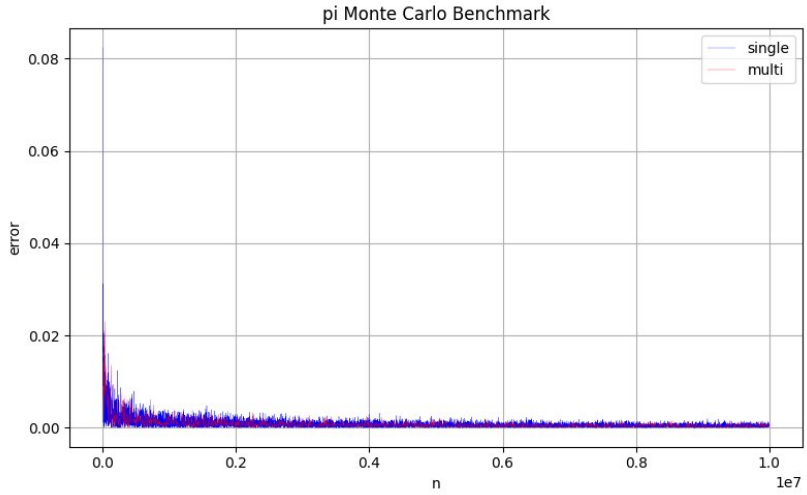


Now, parallelize the code with OpenMP. Take care: which variables need to be shared with write access and how do you prevent race conditions?
Caution: the function rand() from stdlib.h is not reentrant safe (and therefore not thread safe). As an alternative, you might for example instantiate an instance of std::mt19937 for each thread.

The next parts should be executed on the cluster.

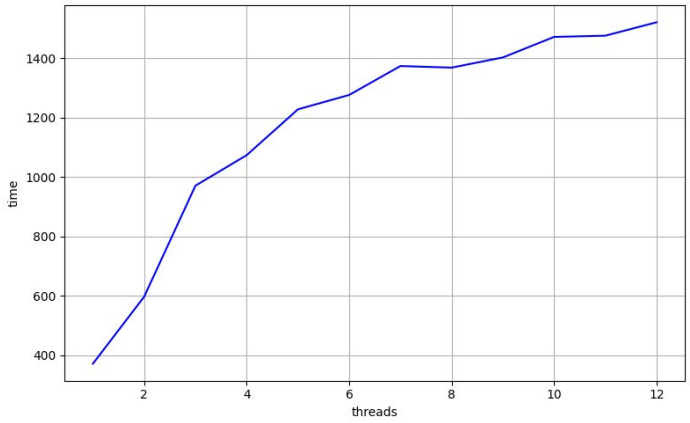
Plot the convergence to π as a function of N as your program is executed with 12 threads. If the convergence is much worse than before – are your random numbers independent on each thread?

Insert the plot here.



How well does your program scale? Perform a strong scaling study, increasing the number of threads from 1 to 12, with $N = 1e7$.

Generate a plot of performance vs. thread count and one plot of efficiency vs. thread count and insert them here. A sensible measure for performance might be points per second.



Now, do a weak scaling study. Fix the number of points N to 10^6 per thread (e.g. $N = 5 \cdot 10^6$ for 5 threads). Plot the performance and as a function of thread count.