Scalability of Monte Carlo Pi Calculation Using HPX

Damond Howard, (Hartmut Kaiser)
Louisiana State University, Center for Computation and Technology, Baton Rouge, LA 70803.

STE||AR
stellar.cct.lsu.edu

Abstract

HPC is becoming increasingly popular in solving scientific problems that require massive amounts of processing power. In order to take advantage of HPC, software that allows the interconnected nodes, memory and processors communicate between each other is needed. That's where HPX comes in. Although Calculating pi using the Monte Carlo Method is a very simple problem in itself, it describes a set of problems that can also be solved using the Monte Carlo Method, and the data collected from such a calculation can describe how similar applications will scale under similar conditions with HPX. Upon analysis of the data, I found that when a certain amount of work was done per thread, it allowed for a dramatic speedup in execution time. I also saw that after reaching a certain amount of cores, the execution speed began to slow down as expected due to too much parallelization. The Monte Carlo Pi program ended up scaling quite well under certain conditions producing execution times far less than the serial execution.

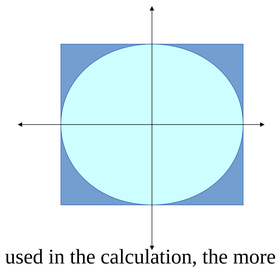
Calculating Pi Using the Monte Carlo Method

Pi is a mathematical constant commonly defined as the ratio of a circles circumference to its diameter. Here's how to calculate Pi using the Monte Carlo Method.

1.Inscribe a unit circle within a square.

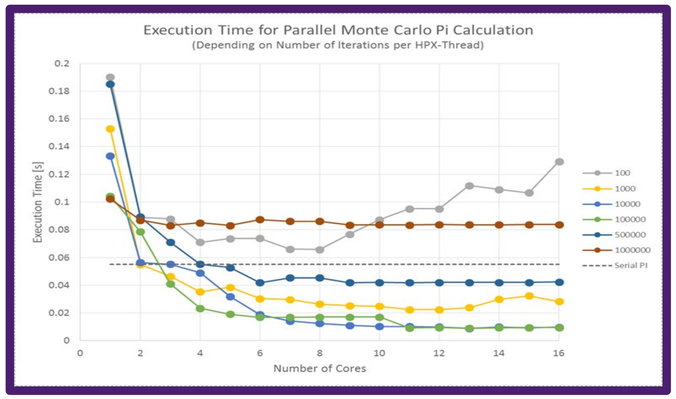
2.Choose a set amount of random points inside the square.

3.The approximation of Pi is going to be the number of points inside the circle divided by the total number of random points multiplied by 4.



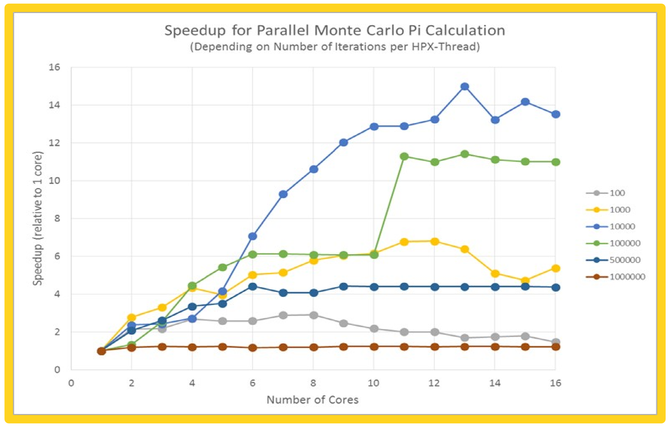
The more random points used in the calculation, the more accurate pi becomes.
If a random point *(x,y)* satisfies *x^2+y^2=1* then it lies inside of the square.

Data, Analysis and Results



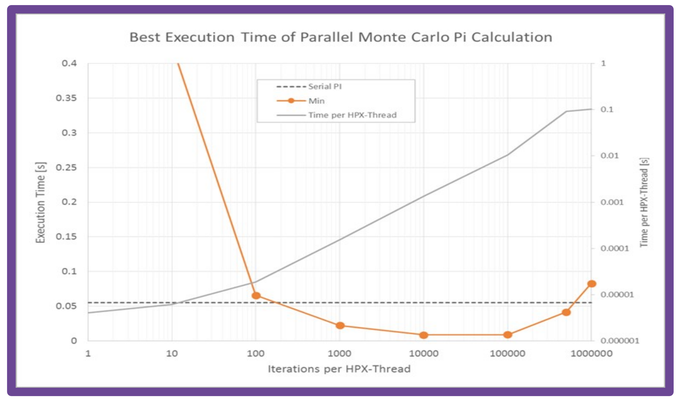
Analysis of Pi calculations

The simulation takes 0.0552536 seconds to execute serially and the fastest times are achieved at 10000 iterations per HPX-Thread. Execution time for 100 and for 1000000 iterations per HPX-thread fare exceed the serial time. This is because 100 iterations per HPX-threads is not enough work to justify the overhead while with 100000 iterations there is no parallelization due to the fact that the number of iterations per HPX-Thread is the same as the total number of iterations leaving it to execute on one thread.



Scaling and Efficiency

Speedup is the most important thing to consider when measuring scalability in HPC, because it shows how well resources are used by the machine and the runtime system.
speedup in this case depends on the number of iterations per HPX-Thread, and is measured by the time for 1 core to execute the program over the time for N cores to execute the program. There was 0 speedup for 1000000 iterations per HPX-Thread because there was only one thread for each run. The most speedup came from 10000 iterations per HPX-Thread with the greatest speedup at 13 cores.



Fastest Execution Time

The best execution times for the Monte Carlo Pi calculations occurred at 10,000 iterations per HPX-Thread as well as 100,000 iterations per HPX-Thread. These are the optimal conditions because each thread does a reasonable amount of work to account for overhead and not too much to where there is not enough parallelization.

What is HPX ?

HPX stands for High Performance ParalleX and is an open source and freely available C++ runtime system. HPX provides a uniform, standard-oriented way to program parallel and distributed applications for many different platforms. HPX works on a variety of architectures such as 32-bit, 64-bit, and ARM processors as well as several different operating systems such as Linux, Windows, and OS X. HPX can also be used on several different types of clusters. HPX manages parallel execution on several threads while minimizing the causes of a slow system.

Conclusion

When running tests with the Monte Carlo Pi program results were very close to what was expected. There was a certain amount of cores and work per thread that allowed for the fastest possible execution of the calculation as well as a considerable amount of speedup as number of cores increased. Not enough parallelization and the program was too slow, and too much parallelization and the program becomes serial again. Also there must be sufficient work within each thread in order to justify the overhead of actually creating each thread .

Acknowledgments

LA-STEM NSF MPS 0651345

Louisiana State University, Center for Computation and Technology

STE||AR group