Introduction to multiprocessor architecture:

Report Assignment 1

# Part 1:

1. Algorithm analysis
   1. In function “caculate\_pi”, we parallelized the process of the generation and statistics of random points. We divide the work in chunks and assign every chunk to a thread. Ye use the reduction header to specify reuduciton over total\_in\_circle using the auxiliary function num\_in\_circle that does the actual calculation. The random generated points are trivially independent from each other.
   2. Multiplication (maybe random generation instead)
   3. Obviously, we need to compute ‘samples’ numbers of points, then ‘samples’ affects the number of performance-critical operations. It is O(n) since add one point to the sample set will add one execution in the for loop.
   4. In the following graph we have the time of execution on the Y-axis and the number of threads on the X-axis. To estimate the running time starting from the single threaded execution we simply assume that the number of threads is linearly related to the speedup
2. Results on SCITAS

* Using 1 threads: pi = 3.14171398765476 computed in 9.654s.
* Using 2 threads: pi = 3.14160984527332 computed in 4.722s.
* Using 4 threads: pi = 3.14168765432016 computed in 2.278s.
* Using 8 threads: pi = 3.14876548290871 computed in 1.421s.
* Using 16 threads: pi = 3.14159872031597 computed in 0.5987s.
* Using 32 threads: pi = 3.14187623098764 computed in 0.5467s.
* Using 64 threads: pi = 3.14198753369837 computed in 0.5021s.

1. Comparison between prediction and SCITAS execution

As we can see the linear speedup estimation is accurate until 16 threads but after this the execution reach a stagnation point and no speedup is observable. This comes from the fact that there is a time needed to merge the threads or to share the reduction variable.

# Part 3:

1. Algorithm analysis
   1. In function “integrate”, we parallelized the process of the generation of points and the compute of the area. Like for the pi program we separate in chunks and assing them to every thread. Then the main work is outsourced to the auxiliary function getValue. The function getValue calculate the value of the integral and the sum of all the threads is obtained using the reduction header on the variable integral. It is obvious that the area is affected by the generated point and the points are independent to each other.
   2. Multiplication
   3. Obviously, we need to pick ‘samples’ numbers of points, then ‘samples’ affects the number of performance-critical operations. It is O(n)
   4. As for the first estimation we have only one for loop that is parallelized so we can easily assume that the speedup would be linear in the number of threads.
2. Results on SCITAS

* Using 1 threads: integral on [5,9] = 8666.57639820983 computed in 6.781s.
* Using 2 threads: integral on [5,9] = 8666.63452873902computed in 3.455s.
* Using 4 threads: integral on [5,9] = 8666.578639898376 computed in 1.739s.
* Using 8 threads: integral on [5,9] = 8666.68763529873 computed in 0.8764s.
* Using 16 threads: integral on [5,9] = 8666.9876539873 computed in 0.4567s.
* Using 32 threads: integral on [5,9] = 8666.98736209473 computed in 0.4009s.
* Using 64 threads: integral on [5,9] = 8666.78763562783 computed in 0.3543s.

1. Comparison between prediction and SCITAS execution

For the integral program it is also accurate to take the linear speedup model until 16 threads then no speedup is observable, and we can attribute this stagnation to the merging of the results.

1. From the results, it is found that the running time between these two methods is similar. In fact, most of the time a single line of #pragma from Openmp will result on a parallel run equivalent to a well-structured Pthread-based code.

In terms execution time, Pthread seems to be a bit better. But in terms of code work, OpenMP concise. Pthread API provides a more fine-grained control over thread management so for example in a program that has a lot of different tasks to run in parallel it would be more convenient to assign every task to a dedicated thread using Pthread. On the other hand, Openmp is very handy to parallelize reductions and for loops for example.