Computing Pi on Blue Waters

Mobeen Ludin, Shodor

2015

http://tinyurl.com/acca-cs-pi

If you need help with using Blue Waters, please read the Blue Waters Usage Guide document.

Computing pi is yet another set of simple examples that can be used to learn and teach different parallel programming implementations. These examples include OpenMP, MPI, OpenACC, and CUDA. The pi computation involves splitting a unit circle into quarter circles, dividing the width of a quarter circle into equal parts, using each width as the base of a rectangle, computing the height of each rectangle as it intersects the circle on its left side, and summing the areas of each rectangle. The result is multiplied by 4 to cover the entire circle. Because it is a unit circle, area = pi, so the final result is the computed value of pi.

Log in to Blue Waters:

Type in the following command, replacing **<your username>** with your actual username on Blue Waters (spaces are highlighted):

```
ssh <your username>@bwbay.ncsa.illinois.edu<ENTER>
```

Request interactive job:

```
qsub -I -l nodes=2:ppn=16:xk -l walltime=01:00:00<ENTER>
```

Copy pi code:

In a separate window, log in to Blue Waters again, and type the following command to copy the **pi** code directory into your home directory.

```
cp -r ~ludin/pi ~<ENTER>
```

Enter the following command and confirm you have a directory called **pi** listed:

```
ls ~<ENTER>
```

Close the window.

Run pi serial:

Once the interactive job is ready, change into the **pi serial** directory:

```
cd ~/pi/serial<ENTER>
```

Run the program:

```
time aprun -n 1 ./pi-serial -r <number of rectangles><ENTER>
```

The **-r** is for number of rectangles in a quarter circle. Run the program and change the number of rectangles to 1000, 1000000, 1000000000, and 1000000000000. There are two things to observe. One, as you increase the number of rectangles, it takes more time to solve the problem. Two, as you increase the number of rectangles, the gives you a better approximation of pi.

Run pi OpenMP:

Change into the **pi OpenMP** directory:

```
cd ~/pi/omp<ENTER>
```

Decide how many OpenMP threads you want to use when running the pi program. There are 16 cores on the XK nodes you requested, so the number of threads should be between 1 and 16, inclusive. You can use higher numbers than 16, but then there will be more than 1 thread per core, which will probably hurt performance. Set the number of threads by running this command:

```
export OMP_NUM_THREADS=<number of threads><ENTER>
```

Run the program using this command:

```
time aprun -n 1 -d <number of threads> ./pi-omp -r <number of rectangles><ENTER>
```

Run pi MPI:

Change into the **pi MPI** directory:

```
cd ~/pi/mpi<ENTER>
```

Run the program:

```
time aprun -n <number of processes> ./pi-mpi -r <# of rectangles per process><ENTER>
```

Note that **-r** now specifies the total number of rectangles **per process** as opposed to total.

Run pi OpenACC:

Change into the **pi OpenACC** directory:

```
cd ~/pi/acc<ENTER>
```

Run the program:

```
time aprun -n 1 ./pi-acc -r <number of rectangles><ENTER>
```

OpenACC, like OpenMP, provides the compiler with hints. Sometimes the compiler may not recognize a hint and skip it without letting the user know. To make sure your code was running on the accelerator you can add **CRAY_ACC_DEBUG=1** to the command line before **aprun**, which will cause the program to run in a debugging mode. In order to run debugging mode we have to first load a module:

```
module load craype-accel-nvidia35<ENTER>
```

Run the program:

```
time CRAY_ACC_DEBUG=1 aprun -n 1 ./pi-acc -r <number of
rectangles><ENTER>
```

Run pi CUDA:

Change into the **pi cuda** directory:

```
cd ~/pi/cuda<ENTER>
```

Run the program:

```
time aprun -n 1 ./pi-cuda -r <rectangles per thread> \
-t <threads per block> -b <blocks per grid><ENTER>
```

Note that **-r** now specifies the number of rectangles **per thread**. You can also control the number of threads per block with **-t** (max of 1024, which is the system limit, and must be a power of 2), and the number of blocks per grid with **-b** (must be less than or equal to 65535, which is the system limit).

Exploration questions:

OpenMP:

1. What combination of problem size and number of threads results in the fastest, most accurate approximation of pi?
2. Generate a weak scaling graph using http://shodor.org/interactivate/activities/SimplePlot.

MPI:

For what number of processes does the program consistently take longer to run than for a lower number of process when approximating pi with 1000000000 total rectangles?

All:
Create a comparison graph between the different implementations of pi (serial, OpenMP, MPI, OpenACC, and CUDA). The y-axis should have the time it takes to run the fastest version of each implementation of the program, and the x-axis should have the number of rectangles (use 1000, 1000000, 1000000000, and 1000000000000).