PiTP Lectures Notes: Problem Set 4

Volker Springel

July 17, 2009

1 Parallelization and the intricacies of floating point math

In this exercise we want to parallelize an extremely simple problem – the summation of a list of numbers. We'll do this both with MPI and OpenMP. The results will prompt us to recall some of the limitations of floating point arithmetic, and to think about implications this has for parallelization in particular. We go through the following steps:

- Sum a provided list of numbers, parallelized with OpenMP.
- Solve the same task with an MPI parallel program.
- Compare the results for different numbers of compute cores.
- Write a program that *exactly* sums the list of numbers.

1.1 Summation with OpenMP

On the School's web-site, you'll find a binary file *numbers.dat*. This contains first a 32-bit integer that gives the number of double-precision values stored in the file (10 million in the provided example), followed by the numbers themselves. (The file is in little-endian. If you happen to work on a big endian processor, you need to swap the endianness.)

Write code that reads in the numbers and simply sums them in a loop. Parallelize this loop with OpenMP, and print the final result. Also print the number of OpenMP threads that were used by the code, via the omp_get_max_threads() function. (Hint: For the gcc compiler, you need the switch *-fopenmp* to active OpenMP.) Note that you can use the environment variable OMP_NUM_THREADS to determine how many threads will be exercised at runtime.

1.2 Summation with MPI

Now implement a version of the code that does the summation in a distributed memory fashion. Do not simply duplicate the data on each MPI-Task, rather implement code that distributes the numbers roughly equally to the processors. Then sum up the partial lists on each MPI-Task, and build the global result with a collective reduction operation. Print the final result to the screen, together with the number of MPI tasks that were in use.

1.3 Testing the methods for different numbers of cores

Run your summation codes for different numbers of CPUs/threads, and record the results. What's going on? Is there a way to decide which result is 'more correct' than the others? What do you make of the fact that the result changes when the number of processors is changed? Think about potential implications for parallel simulation codes. How large do you think the absolute and relative error in your result may be?

1.4 Getting a correct answer

Write code to calculate the *correct* sum of the list of numbers, for example using the GMP library.

Finally, take a look at the article "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg, available on the School's website.