Introduction to Supercomputers: Problem Set 4

Sindre Magnussen

January 2014

Introduction

In this assignment we were supposed to write a program that calculates the sum in (1) with a range of n-values and compares the result with equation (3):

$$S_N = \sum_{i=1}^{N} v(i) \tag{1}$$

$$v(i) = \frac{1}{i^2}, \qquad i = 1, \ldots, N \tag{2}$$

$$S = \lim_{N \to \infty} S_N = \frac{\pi^2}{6} \tag{3}$$

My solution is written in C. The code I have written makes it possible to enable both serial, OpenMP-code and MPI-code. OpenMP and MPI should work together. The code has been tested on a local machine. See the following explanation of how to enable the different run-options:

Serial program
The serial program can be run by setting OPENMP_EN and MPI_EN to OFF in the CMakeCache.txt.

OpenMP-code
Enable OPENMP_EN, by setting it ON in CMakeCache.txt.

MPI-code
Enable MPI_EN, by setting it ON in CMakeCache.txt.

Explanation of solution: MPI

The MPI-code needs some explanation. The code does the following. First we initialize MPI and set up the variables needed. After that, node 0 is responsible for handing of work to the other nodes. Node 0 in total generates the total problem size, that is, it generates the vectors of length max N (2^{14} in our case) divided by the number of nodes. A vector in a node starts at value of the *rank* and each element has an offset of *rank*. I figured it was better to hand of the vectors needed to solve the sum in (1) with N equal to 2^{14}, as this minimizes the number of messages passed and gives easier code to grasp. To calculate (1) for smaller N, we simply pick out the elements needed in each of the vectors to calculate the sum.

Node 0 then simply sends of the vectors to the right nodes. This is done with MPI_Send(...). The other nodes receives the message by calling MPI_Recv(...). Both of these calls is blocking. Each time we are done processing a sum at a given value of N, we send the partial sum of to node 0 again to print out the solution. This is accomplished by calling MPI_Reduce(...) (see the code for the parameters to each of the functions).

I also make use of MPI_Wtime() to measure computation-time if MPI is enabled by calling the WallTime()-function (taken from the lecturers common.c library).

Floating point operations and work load

When generating the vector(s) we need two additions and one multiplication. That is three floating point operations per vector element.
When summing up elements, with OpenMP enabled, we do one floating point operation per element in the vector.

I will say that this system has do not have a balanced work load, since node 0 is responsible of generating all the vectors.

I don't think it is necessary to use parallel processing to solve this particular problem. This conclusion is based on the computation-time in the different run-modes. The serial code-runs is almost constant in the computation-time. It's run in a couple of microseconds. When I enable OpenMP, the computation-time is fluctuating more. This is expected since we do fork and join operations when filling and summing the vector(s). This makes the code to run in milliseconds in some runs, but it can also be as good as the serial code.

The problem size is to small to benefit from parallel processing when using OpenMP. But if we increase the value of k, it is a benefit. This can easily be

seen if we increase the value of K_MAX (see source code). As an example on my machine, a K_MAX value of 26 gives a serial computation-time of about 2.1 seconds, but when I run with OpenMP I get a computation time of 1.8 seconds. As the problem size gets bigger, that is a big value of K_MAX, one should expect the OpenMP code to perform better given a enough memory and multicore-chip.

Something about MPI on when the bug is fixed....