Introduction to Supercomputers: Problem Set 4

Sindre Magnussen

January 2014

Introduction and results

I will just start with explaining a bug in my code. Since MPI was acting strange on my computer, I had to test my code at my friends computer. As a result, the value of n during printing is wrong. I hope this does not trouble the reader to much when testing my code.
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. I have used two preprocessor-flags (HAVE_MPI and HAVE_OPENMP) to enable/disable either OpenMP-code or MPI-code. The two flags are set when we run *cmake*. 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.

I make use of the following MPI calls in my solution:

MPI_Init() Initializes MPI.

MPI_Comm_size() Saves number of processors.

MPI_Comm_rank() Saves this nodes id.

MPI_Send() Sends a message, used when sending vectors to other nodes. Blocking call.

MPI_Recv() Receives message, used by other nodes when receiving from vector 0. Blocking call.

MPI_Reduce() Gathers information at a node. Used when we want to report the answer.

All of these functions are required.

Memory requirements As the other variables is of constant size, it is only the vector that determine the memory needed. I will only discuss the memory requirements from that point of view with $N \gg 1$. In the serial code we have to store N double values of 64 bits. That is, we are required to store all the vector elements in one memory.

With multiprocessor systems we still need to store the N double values. So the memory needed is still dependent of the vector size N. Even though we run on P processors where each allocates O($\frac{N}{P}$) pieces of memory. One positive thing, when running on a cluster we evenly distribute the memory across the nodes. So at each node there is allocated O($\frac{N}{P}$) memory, but we need in total O(N) memory on the cluster.

Floating point operations and work load The fill_vector()-function does two additions and one multiplication when calculating the value to be passed to the function v(). The function v() does two multiplications and one division. This gives us a total of 6 floating point operations. Since K_MAX is 14, N is 16384 which in total gives us 98304 (6*N) floating point operations.

When summing up elements we do one floating point operation per element in the vector. The next part depends if we are running with MPI. If we are not, we simply sum up. This requires N floating point operations. If MPI is enabled we also do some floating point operations when calling MPI_Reduce().

Let P be the number of processing-nodes. Then we need $\frac{N}{P}$ floating point operations per vector to calculate the partial sum in each node. This is done for each node, so in total we get N operations. We have P partial sums. We then do (P-1) floating point operations to gather the information at node 0. So in total:

$$FPO_{tot} = N + (P - 1) \tag{4}$$

I will say that this system do not have a balanced work load, since node 0 is responsible of generating all the vectors. It would be balanced if all nodes generated it's own vector.

Parallel processing I don't think it is necessary to use parallel processing to solve this particular problem. The problem size is to small to benefit from parallel processing.

If we increase the problem-size we see that we benefit of using OpenMP. As an example on my machine enabling only serial code, 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, a big value of K_MAX that is, one should expect the OpenMP code to perform better given enough memory and multicore-chip.

The same reasoning works for MPI. The MPI-code is even slower than the OpenMP-code, due to the overhead of sending vectors from node 0.

Some results The following table shows some results of $S - S_N$, running with both OpenMP and MPI enabled (P is the number of processes).

k	P=2	P=4	P=8
3	1.175120e-01	1.175120e-01	1.175120e-01
4	6.058753e-02	6.058753e-02	6.058753e-02
5	3.076680e-02	3.076680e-02	3.076680e-02
6	1.550357e-02	1.550357e-02	1.550357e-02
7	7.782062e-03	7.782062e-03	7.782062e-03
8	3.898631e-03	3.898631e-03	3.898631e-03
9	1.951219e-03	1.951219e-03	1.951219e-03
10	9.760858e-04	9.760858e-04	9.760858e-04
11	4.881621e-04	4.881621e-04	4.881621e-04
12	2.441108e-04	2.441108e-04	2.441108e-04
13	1.220629e-04	1.220629e-04	1.220629e-04
14	6.103329e-05	6.103329e-05	6.103329e-05

As we can see there is no difference in running on 2, 4 or 8 nodes, as expected.

Explanation of solution: MPI

A short explanation of the MPI-code. Even though it's not required in the answers, it's nice to see how I think regardless of the code.
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. 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.