Exerice 4 Report

TMA4280 – Introduction to Supercomputing

Silas Spence

February 25th 2013

Table of Contents

Introduction

This report details one serial and two parallel versions of a C code that creates a vector, v, where:

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

And then computes the sum, S_n:

$$S_n = \sum_{i=1}^{n} v(i)$$

This sum is compared to the limiting case:

$$\lim_{n\to\infty} S_n = \pi^2/6$$

By printing this difference for various lengths, n, it is possible to see how the error in the solution evolves with changes in vector length. Examining the above equations, we can see that the solution will require 2n floating point operations to generate the vector v, and n-1 floating point operations to generate the sum S_n. A convenient feature of this problem is that the computation and memory required to fill each position in the vector v is identical. This means that as long as the length of each sub-vector assigned to a thread in a parallel program is the same, the program will be load balanced. This requires that n/threads (or n/processes) must be a whole number.

The serial code is parallelized by means of OpenMP and MPI. The results of the parallel codes are compared to each other and results of the serial code for P=4 and P=16 threads/processes. Details of the implementation, including MPI function calls used in the MPI version of the code, are discussed.

In general, as the scope of the problem given is so small, little effort has been made to abstract away details, such as MPI function calls, into subfunctions; in a more realistic usage scenario, this entire problem would be a subfunction in its own right. In both the serial and parallel cases the vector v is in fact superfluous – the sum S_n can be calculated without ever creating or filling the vector. However, the presence of the vector does impose some additional bookkeeping challenges and coding constraints thus it has been created and filled in all three codes (even though the MPI problem statement does not explicitly require this) in order to try and gain the maximum benefit from completing the exercise.

The following sections describe each of the three codes in more detail. Files taken from the course examples, such as common.c and common.h, and used in the solutions presented are not discussed unless modifications are made.

Serial

The serial version of the code is pretty straight forward and a printout of the source code can be found in Appendix A1. A vector is created, with contiguously allocated memory thanks to the createVector function provided in common.c, and then a for loop fills each vector index with the appropriate value. The sum is calculated "on the fly" as each vector index is filled as this saves running a second for loop to get the sum of the vector. The problem statement requires that the difference between the sum, S_n, and the limiting case, S_∞, be printed out for $n=2^k$ where k=4….14. This means a total of 11 differences need to be printed out. Rather than calculating the sum once for each length, the vector and sum are calculated once, for the maximum length of 2^{14}, with a logic statement providing printouts at the required iteration numbers. The modest cost of the additional logical far outweighs the savings in recalculating the first part of the vector 10 times.

OpenMP

In theory, parallelizing this problem with OpenMP should be as simple as putting a #pragma on the for loop with a reduce operation assigned to the sum. This was attempted, and it did provide an answer – an incorrect one.

Rather dangerously, at low threading levels (P=2), the output this provided was not obviously wrong as the values in v drop off very quickly with increasing index number. The error inherent in this approach is that only one of the threads actually contains the index value i. It is only this thread that is capable of evaluating the if statement and thus generating an output of the difference. Depending on whether the loop is split into sections or interleaved, the sum will be in error either from i=2 (interleaved) or i=N/2+1 (partitioned). When the for loop exits, the final reduced sum will be correct. But since the final printout when i=n occurs before exiting the loop, this printed difference also comes out in error.

Due to the lack of granular control over the forking of the threads in OpenMP, I was unable to find a solution that did not require a second for loop. The crudest approach is to loop over all 11 lengths of n, recalculating many many entries in v along the way. A more efficient approach is to use the new outer loop to break the inner loop down into each section of n (i.e. from n=0 to $n<2^4$, n=24 to $n<2^5$, etc.), with the required reduce operation on the sum occurring every time the inner loop exits. This is the approach used in the OpenMP version of the code and can be seen in Appendix A2. As the problem is inherently load balanced, the scheduler used is static in order to minimize overhead due to forking the loop.

MPI

With the much finer control offered by the MPI protocol, it was possible to return to an algorithm with only one loop as in the serial solution. The code can be found in Appendix A3. The same logical operator is used to determine when $i=2^k$ and the only significant difference is that an MPI reduce operation is performed prior to thread 0 printing out the difference.

A significant difference from the OpenMP code is that the vector is divided in an interleaved manner rather than in sections. This ensures that, without any further bookkeeping, the global reduce operation yields a sum that is correct for the current value of i.

In order to ensure general functionality of the interleaving approach as implemented, it was necessary to modify the createVector function provided in common.c to assign any remainder iterations (if n/p did not return a whole number) starting with rank=0 and up, rather than starting with rank=size and working down as it did originally. The error this corrected was skipping indices – if n=3 and p=2 for example, the resulting erroneous sum would have consisted of i=1,2,4 instead of the correct i=1,2,3.

The solution enabled by MPI seems much more elegant than what was possible with OpenMP. Although the common.c code initializes MPI with MPI_Init_thread and sets up more communicators than just MPI_COMM_WORLD, the features this approach enables are not used in the MPI solution presented. For the purposes of the code presented, a simple MPI_Init, with one world communicator, MPI_Reduce, and MPI_Finalize are sufficient. Sharing the variable declarations and initializations via MPI_Bcast is avoided as in the present case the network traffic incurred on a distributed memory machine is much slower than having each node run the startup section of the code.

In practice, with the size of $n=2^{14}$, it is not feasible to benchmark these two codes as the MPI_Init and MPI_Finalize operations drown the calculation itself. Increasing n to a number in the billions yields run times on my local machine for OpenMP and MPI that are essentially identical within the margins of repeatability.

Comparison of Results

The results of the serial code, and the parallel codes with P=4 and P=16 threads/processes, are presented in double precision in Table 1 below.

n	Serial	OMP, P=4	OMP, P=16	MPI, P=4	MPI, P=16
i=2^ 4	0.0605875334032393	0.0605875334032393	0.0605875334032393	0.0605875334032391	0.0605875334032393
i=2^ 5	0.0307668040203022	0.0307668040203020	0.0307668040203020	0.0307668040203020	0.0307668040203020
i=2^ 6	0.0155035654393394	0.0155035654393387	0.0155035654393387	0.0155035654393387	0.0155035654393387
i=2^ 7	0.0077820618937652	0.0077820618937645	0.0077820618937645	0.0077820618937647	0.0077820618937645
i=2^ 8	0.0038986305395463	0.0038986305395459	0.0038986305395459	0.0038986305395465	0.0038986305395459
i=2^ 9	0.0019512188931299	0.0019512188931297	0.0019512188931297	0.0019512188931301	0.0019512188931292
i=2^10	0.0009760858180610	0.0009760858180621	0.0009760858180621	0.0009760858180623	0.0009760858180619
i=2^11	0.0004881620601096	0.0004881620601129	0.0004881620601129	0.0004881620601118	0.0004881620601129
i=2^12	0.0002441108250932	0.0002441108251028	0.0002441108251028	0.0002441108251010	0.0002441108251028
i=2^13	0.0001220628622214	0.0001220628622225	0.0001220628622225	0.0001220628622225	0.0001220628622247
i=2^14	0.0000610332936408	0.0000610332936426	0.0000610332936426	0.0000610332936459	0.0000610332936473

Table 1 – Code Output

Close inspection reveals that the results of each code are not in complete agreement with each other. This is easier to see by subtracting the output of the serial code from the outputs of all the codes, as is shown in Table 2 below.

n	OMP, P=4	OMP, P=16	MPI, P=4	MPI, P=16
i=2^ 4	0.0000000000000000	0.0000000000000000	0.0000000000000002	0.0000000000000000
i=2^ 5	0.0000000000000002	0.0000000000000002	0.0000000000000002	0.0000000000000002
i=2^ 6	0.0000000000000007	0.0000000000000007	0.0000000000000007	0.0000000000000007
i=2^ 7	0.0000000000000007	0.0000000000000007	0.0000000000000005	0.0000000000000007
i=2^ 8	0.0000000000000004	0.0000000000000004	-0.0000000000000002	0.0000000000000004
i=2^ 9	0.0000000000000002	0.0000000000000002	-0.0000000000000002	0.0000000000000007
i=2^10	-0.0000000000000011	-0.0000000000000011	-0.0000000000000013	-0.0000000000000009
i=2^11	-0.0000000000000033	-0.0000000000000033	-0.0000000000000022	-0.0000000000000033
i=2^12	-0.0000000000000096	-0.0000000000000096	-0.0000000000000078	-0.0000000000000096
i=2^13	-0.0000000000000011	-0.0000000000000011	-0.0000000000000011	-0.0000000000000033
i=2^14	-0.0000000000000018	-0.0000000000000018	-0.0000000000000051	-0.0000000000000065

Table 2 –Differences in Output

The differences shown in Table 2 are to be expected and are due to rounding errors from adding two floating point numbers of significantly different magnitudes. The problem starts with the serial code where the vector is summed from i=1 upwards, meaning we start with a large number and add progressively smaller floating point numbers. To minimize this, the vector should be added from i=n downwards. However, this was not implemented as the added algorithmic complication was deemed to be detractive from the main objectives of this exercise.

The above problem with rounding errors potentially gets even worse with parallelization as the above serial issue is still present but now there is also the reduce operation. How the vector is divided has a significant impact on how the rounding errors accumulate.

The OMP code is the worst for this as it partitions the code into serial chunks of n, meaning the first thread will have a number of order 1, while the second thread will have a number of order .001 or so, depending on threading depth, and so on. Control over the order of summation in the reduction operation could limit the effects of this, but no such control is available in OMP.

Due to the structure of the MPI algorithm, which required interleaved division of the vector, the effect of rounding errors should not as bad as with the OMP algorithm, however the problem of summing the vector in the direction of largest to smallest values still remains.

Thus the differences between the serial codes and parallel codes seem to be reasonable as they are all in the last one or two significant digits. In theory, given a similar algorithm, the OMP and MPI rounding errors should be the same, giving the same result. But, as mentioned before, there is no control over the OMP reduce operation and the overhead for taking such control in MPI is not likely to be worthwile; therefore similar algorithms may still produce subtly different results. In reality, as the reduction operation is achieved by double recursion, and the largest sub-sum is held on process/thread 0 in both the OMP and MPI cases, it is unlikely that significant improvements could be made in the rounding error by taking explicit control of the reduction operation.

It is interesting to note that the MPI results show sensitivity to the threading depth while the OMP results do not. This could suggest that the majority of the rounding error is incurred in the first few summations, where the change between each successive value in the vector is at its greatest. This theory is supported by the fact that the OMP results are closer to the serial results than the MPI results. The fact that the MPI results diverge further from the serial results with increasing thread depth seems inconsistent however, and remains somewhat puzzling.

The results in Table 1 were obtained both on a local dual core machine, and on a single 12 core node of kongull. This confirms that the differences between the codes are due to numerical errors and are not architecture dependent.

Conclusions

The details and results of three codes, one serial, one OMP, and one MPI, have been discussed and compared. In general, it was seen that the granular control afforded by MPI could provide some useful benefits while the supposed simplicity of OMP could in fact lead to additional algorithmic complexity.

By retaining the vector, despite it being superfluous for computing the required sum, this problem is memory constrained for large values of n. As the problem is embarrassingly parallel, it can benefit from speedups due to parallelization if n is sufficiently large to drown out the overhead from parallelizing the task. For the give maximum size of $n=2^{14}$, this is not the case for MPI, while OMP yielded fairly similar run times to the serial code on up to 4 physical cores.

Where parallelization becomes most attractive n becomes too big to fit into memory on a single machine. Obviously such a scenario eliminates OMP as an option, since that is for shared memory machines only. As long as $n>>P$, there is negligible additional memory required by the parallel program compared to the serial program; some variables are duplicated in the parallel case, but this is trivial compared to the memory required for the vector and the vector elements are not duplicated.

Appendix A1 – Serial

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "common.h"


int main(int argc, char** argv)
{
    //maximum size of vector - 2^14 for Exercise 4
    int maxN = pow(2,14);

    //value of sum as n->infinity
    double exactsum=pow((4.0*atan(1.0)),2)/6.0;

    //initialize some variables
    double sum=0;
    int k=4;
    int nextN=pow(2,k);

    //make the vector
    Vector v = createVector(maxN);

    //fill the vector
    for (int i=0;i<maxN;++i) {
          v->data[i] = 1.0/((double)(i+1)*(double)(i+1));

          //calculate the sum on the fly - saves a second for loop
          sum += v->data[i];

          //print difference at 2^k, k=4...14
          //for the price of a logical every i, only have to run through
          //vector once instead of length(k) times
          if (i+1==nextN) {
                printf("difference at i=2^%2i: %1.16f\n", k, exactsum-sum);
                k++;
          nextN=pow(2,k);
          }
    }
}

```

Appendix A2 – OMP

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "common.h"


int main(int argc, char** argv)
{
    //value of sum as n->infinity
    double exactsum=pow((4.0*atan(1.0)),2)/6.0;

    //initialize some variables
    double sum=0;
    int lastN=0;

    //make the vector
    Vector v = createVector(pow(2,14));

    //divide the vector filling operation into each 2^k piece
    //in order to parallelize filling task while avoiding calculating
    //vector length(k) times
    for(int k=4;k<15;++k){

          //reset the sum after each 2^k is reached to avoid double
          //counting previously summed numbers
          double isum=0;

          //update number of iterations to complete next section of vector
          int nextN=pow(2,k);
          #pragma omp parallel for schedule(static) reduction(+:isum)
          for (int i=lastN;i<nextN;++i) {
                v->data[i] = 1.0/((double)(i+1)*(double)(i+1));

                //calculate the sum on the fly - saves a second for loop
                isum += v->data[i];
          }
    //add sum from current section of vector to total sum
    sum += isum;
    printf("difference at i=2^%2i: %1.16f\n", k, exactsum-sum);

    //update starting point for next section of vector
    lastN=nextN;
    }
}
```

Appendix A3 - MPI

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "common.h"

int main(int argc, char** argv)
{
int rank, size;
init_app(argc, argv, &rank, &size);

//check number of processors used is a power of two
if(size % 2){
if(rank==0){
printf("please choose a number of processors that is a multiple of 2,
exiting\n");
}
close_app();
return 0;
}

//maximum size of vector - 2^14 for Exercise 4
int maxN = pow(2,14);

//initialize some variables
double mysum = 0;
double sum=0;
int k=4;
int nextN=pow(2,k)/size;

//split whole vector in parts, create subvector on each MPI process
int *ofs, *cols;
splitVector(maxN, size, &cols, &ofs);
Vector v = createVector(cols[rank]);

//fill vector
for (int i=0;i<cols[rank];++i){
//vector filling is interleaved - allows printing the difference
//at each 2^k rather than recalculating entire vector each time
v->data[i] = 1.0/pow((double)(i*size+1+rank),2);
mysum += v->data[i];

//print out difference at every 2^K n
if (i+1==nextN){
MPI_Reduce (&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0) {
double pi = 4.0 * atan(1.0);
printf("difference at i=2^%2i: %1.16f\n", k, pi*pi/6.0-sum);
}
k++;
nextN=pow(2,k)/size;
}
}

//house cleaning. probably not needed (other than MPI_Finalize, but if you are
//going to cut and paste somebody else's code, you might as well go for broke.
freeVector(v);
free(cols);
free(ofs);
close_app();
return 0;
}
```