TMA4280 Super Computers

Jakob Hovland
Jørgen Grimnes
Assignment 4

Spring 2015

1 Introduction

This report presents a solution to homework project number four in TMA4280 Super Computers, spring 2015, at Norwegian University of Science and Technology. The assignment is a introduction to implementing efficient and parallel programs.

2 The summation program

The solution should be able to generate a vector v. Then the program should compute the sum S_n in double precision on a single processor and print out the difference $S - S_n$ in double precision for $n = 2^k$, with $k = 3, .., 14$.

The implemented program was written in C++ on Mac OSX 10.10 and tested on an Intel compiler. The lightweight program of only about 25 SLOC uses the build-in C++ vector data structure.

Improving the code The program can easily be improved by skipping the precalculation of the vector v. This will save us a both a for-loop and require less memory.

k	$S - S_n$	**k**	$S - S_n$
3	0.11751	9	0.0019512
4	0.060588	10	0.00097609
5	0.030767	11	0.00048816
6	0.015504	12	0.00024411
7	0.0077821	13	0.00012206
8	0.0038986	14	6.1033e-05

Table 1: Error measurements for the linear program

3 Incorporating OpenMP

In order to familiarize ourself with OpenMP, we exploited the parallelization capabilities of OpenMP by using the simply compiler pragma *omp parallel for*. Please see refer to the file *summation openmp.cpp* for the specific implementation.

4 Incorporating MPI

In the MPI implementation, the zero ranked process is responsible for generating both it's own vector of digits and the vectors which the "slave" processes sum up. The P_0 process distribute these vectors by a call to *MPI::COMM_WORLD.Send*. Each processor is then responsible for summing up its own vector of elements. Finally, MPI add up the partial sums and forward the sum to process P_0 for an error printout. Please see the file *summation mpi.cpp* for the implemented solution.

5 OpenMP and MPI in combination

By combining the OpenMP functionality with MPI, we achieve a higher degree of multiprocessing. Each process will then be able to calculate its sum in a multithreaded fashion. Please see the file *summation mpi openmp.cpp* for compiling example of using MPI and OpenMP in combination.

6 Featured MPI calls

We have been operating with the *MPI::COMM_WORLD* communication group

in this program. The calls listed in Table 2 were used in our MPI implementations.

7 Error comparison

The error rate of our calculation $S-S_n$ will increase as we distribute our program over an increasing number of processors due to the floats and rounding error. This is clearly visible in Table 3. The table compares the error measurments between the linear and the dual/octo processed programs as we increase k.

In Figure 1 we may that the error of a $|P| = 8$ execution converges to the error of a $|P| = 2$ execution as we increase the k. This is due to the fact that the round-off accuracy in all processes overshadows the floating point error in passing the values among the processors.

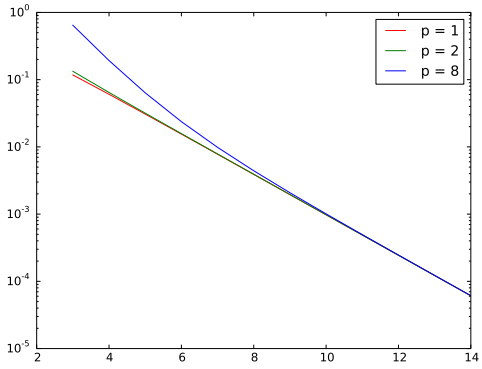

Figure 1: Comparison of the error reduction

8 Memory requirements

For both the single-processor and multi-processor programs the number of vector elements that each process needs to store is $\frac{n}{P}$. In simple terms this means that with p processes the multi-processor program can handle a P times larger problem size than the single-processor program. As an example, if we assume that there are 8 processes, every process has 1 GiB RAM available for storing vector elements, and vector elements are double precision floating points, the maximum value of n for the single-processor program is 2^{27} and the maximum value of n for the multi-processor program is 2^{30}.

k	$error_{P=8}$	$error_{P=2}$	$error_{P=1}$
3	0.64493	0.13314	0.11751
4	0.19244	0.064494	0.060588
5	0.063731	0.031743	0.030767
6	0.023745	0.015748	0.015504
7	0.0098423	0.0078431	0.0077821
8	0.0044137	0.0039139	0.0038986
9	0.00208	0.0019512	0.001955
10	0.0010083	0.00097609	0.00097704
11	0.00049621	0.00048816	0.0004884
12	0.00024612	0.00024411	0.00024417
13	0.00012257	0.00012206	0.00012208
14	6.1159e-05	6.1033e-05	6.1037e-05

Table 3: Error measurements for $P = 8$ and $P = 2$

9 Floating points

Calculating vector v

The number of floating-point operations needed to generate the vector v are:

$$2n$$

Calculating sum S_n

If the vector v is given, the number of floating point operations needed to compute the sum S_n are:

$$n - 1$$

However, if the vector v is not given, the number of floating point operations needed to compute the sum S_n are:

$$3n - 1$$

Load-balancing

Our multi-processor program is as load balanced as it can be, given that only processor P_0 can generate the vector elements. If the problem size n is not divisible by the number of processors P, the program distributes the remaining elements evenly among the processors. Thus each processor performs $\left\lfloor \frac{n}{p} \right\rfloor - 1$ or $\left\lfloor \frac{n}{p} \right\rfloor$ additions. However, the

Call	Description
MPI::Init	Initializes the MPI execution environment
MPI::COMM_WORLD.Get_size	Returns the size of the group associated with COMM_WORLD.
MPI::COMM_WORLD.Get_rank	Determines the rank of the calling process in COMM_WORLD.
MPI::COMM_WORLD.Send	Performs a standard-mode blocking send.
MPI::COMM_WORLD.Recv	Performs a standard-mode blocking receive.
MPI::COMM_WORLD.Reduce	Reduces values on all processes within COMM_WORLD
MPI::Finalize	Terminates MPI execution environment.

Table 2: Executed MPI calls

program would be more load balanced if each processor was also responsible for generating its own vector elements. When only processor P_0 is responsible for generating the vector elements, it performs $2n + \frac{n}{p} - 1$ floating-point operations in total, while the other processors only perform $\frac{n}{p} - 1$ floating-point operations. If all the processors were responsible for generating their own vector elements, every processor would perform $\frac{3n}{p} - 1$ floating-point operations.

10 Parallel processing

Parallel processing is definitely attractive to use to solve this problem, as it is straightforward to implement, can be load-balanced, and requires very little synchronization between processes and threads. On the other hand, since this problem is $O(n)$ it would probably only be required for very large problem sizes n.

11 Conclusion

We have become acquainted with both the MPI and the OpenMP library. We have seen both how to exploit their processing power both individually and in combination.