TMA4280 : Problem set #4

Due on Monday, February 25, 2013

Simen Haugerud Granlund & Steffen Pøhner Henriksen

In this assignment we will write a program in C to compute a sum of vectors. More precisely we will consider a vector $v \in \mathbb{R}^n$ defined:

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

The sum of these vectors will converge to $\frac{\pi^2}{6}$ and is mathematically expressed:

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

First we will compute the sum using an serial approach on one processor. Later we will improve the program using multithreading techniques. Common for all the versions of our program is the common.h/common.c files. These files are produced by Arne Morten Kvarving (our lecturer). They create vectors, and matrices continously in memory, and defines structs that contains information about the data and the data itself. This problem set focuses on improving code by use of multithreading and theory, and we don't consider allocating data in memory an important goal here. Therefore we will use these files not written by us to help focus on the main tasks.

The standard serial version and our starting point throughout this problem set is as follows:

Listing 1: Serial version

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

#define M_PI 3.14159265358979323846



Vector genVector(int length)
{
  Vector vec = createVector(length);
  for (int i = 0; i < length; ++i)
  {
    vec->data[i]=1./((i+1)*(i+1));
  }
  return vec;
}

double doSum(Vector vec){
  double sum=0;
  for (int i = 0; i < vec->len; ++i)
  {
    sum += vec->data[i];
  }
  freeVector(vec);
  return sum;
}
```

```
void printDiff(){
  double Sn=(M_PI*M_PI)/6;
  double sum=0;
  double time=0;
  for (int i = 4; i < 15; ++i)
  {
    time = WallTime();
    sum = doSum(genVector(pow(2, i)));
    printf("Diff (n=%f) = %f,",pow(2, i), sum-Sn);
    printf(" Elapsed: %fs\n", WallTime()-time);
  }
}

int main(int argc, char** argv)
{
  printDiff();
  return 0;
}
```

We created a serial program using BLAS in an attempt to make the serial code optimal. We took advantage of the properties of the inner product to do so. Using a vector consisting of pointers to a double of value one, we were able to use BLAS for the summation. It turned out that this was slower due to allocation of memory for the vector of pointers. Allocating and creating the vectors are dominant to the computations needed and kills the potential speedup.

The parallell verison of the program should be suitable for shared memory computers. This favors the use of OpenMP as this library takes advantage of the shared memory. The difference from the serial version is the doSum method. We have added a pragma to the for loop, and peform an reduction on the sum variable. The schedule mode was set to static due to the similar workload of every loop.

Listing 2: OpenMP version

```
double doSum(Vector vec){
  double sum=0;
  #pragma omp parallel for schedule(static) reduction(+:sum)
  for (int i = 0; i < vec->len; ++i)
  {
    sum += vec->data[i];
  }
  freeVector(vec);
  return sum;
}
```

There is also expected of us to create a program suitable for distrubuted memory systems. This smells of MPI. Our MPI-program splits the vector according to the number of processes. Each process then calculate it's part of the sum. An *MPI_Reduce* call then gathers all the parts, and sums them to the final sum. The final sum ends up on the root process which we chose to be the one with rank 0.

Listing 3: MPI version

```
  for (int i = 4; i <15 ; ++i)
  {
```

```
    int n= pow(2, i);
    int *startIndex, *len;
    splitVector(n, size, &len, &startIndex);
    Vector vec = genVector(startIndex[rank],startIndex[rank]+len[rank]);
    sum = doSum(vec);

    #ifdef HAVE_MPI
    double s2=sum;
    MPI_Reduce(&s2, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    #endif

    if (rank == 0)
    {
      printf("Diff (n=%d) = %f,",n, sum-Sn);
      printf(" Elapsed: %fs\n", WallTime()-time);
    }
  }
```

Comparing the results of $S-S_N$ in our MPI program on 4 processes, 16 processes and the serial version resulted in exactly the same results. However, this it not always the case when using the MPI_Reduce with the parameter MPI_SUM. This is becasue this method takes advantage of the associativity nature of sums, and since it can happen that it sums the parts in different order we can get different round-off errors and therefore different results.

Considering the memory requirement our program is pretty smart. All processes need some basic data like number of processes, rank, the correct sum, the part sum and the split vectors. Each process only generates N/P vector elements, therefore the memory consumption per process decreases with more processes. But of course the total memory consumption increases due to the mentioned overhead variables.

When generating a vector element we use the formula $v(i) = \frac{1}{i^2}$. Which requires two floating point operations. Generating the vector v takes two floating point operations per number of elements (n). To compute the total sum S_n using the OpenMP version (for shared memory computers) we use the same method to generate the vectors, each thread sums $\frac{n}{T}$ elements with one floating operation per sum plus the reduction in the end which takes $T-1$ floating point operations, where T is number of threads.

$$F(n) = 2n + \frac{n}{T}T + (T-1) = 3n + T - 1$$

Our multi-processor program is well balanced because each process is responsible for creating its own vector elements, and doing calculation on them. And each process sums exactly the same amount of elements, which takes the same amount of time. One process is responsible for collecting the data in the end and therefore the load on this is slightly larger, but negligble.

We compared the time it took to compute the difference $S - S_n$ on the serial version and the OpenMP version. The OpenMP version was slightly faster when $n > 8192$, but the total time was $0.000401s$ for the serial version. This is such a small trivial problem that the effort going into parallelize it is not worth it. It takes well under one hundred of a second doing the computations to the precision of the double precision datatype on the serial version. The overhead of splitting up the problem is dominating the computation time of the OpenMP version. In addition

we do know the exact value. This makes it only of academic interest to compute an apporoximation.

The source of our code is delivered as seperate files with this report.