TMA4280
Exercise 4

Vegard Stenhjem Hagen

February 25, 2013

1 Serial

```c
int main(int argc, char** argv){
    double pi = 4.0*atan(1);
    double sum = pi*pi/6;
    double time_init;

    if(argc < 2) {
        printf("Need one parameter, the size of the vector\n");
        return 1;
    }
    int n = atoi(argv[1]);
    printf("Serial:\n");
    double* v = (double*)malloc(n*sizeof(double));
    double sumn = 0;
    time_init = walltime();

    for(long int i=n; i>0; i--){
        //Generate vector
        v[i] = 1.0/((double)i*i);
        //Compute the sum
        sumn += v[i];
    }
    //Compute and print S -S_n
    printf("Error:\t\t %.16e\n",sum-sumn);
    printf("Time Elapsed:\t %f\n",walltime()-time_init);
        return 0;
}
```

2 Parallel - OpenMP

```c
int main(int argc, char** argv){
    double pi = 4.0*atan(1);
    double sum = pi*pi/6;
    double time_init;

    if(argc < 2) {
        printf("Need one parameter, the size of the vector\n");
        return 1;
    }
    long int n = atoi(argv[1]);
```

```
    printf("OpenMP\tThreadcount: %i\n",omp_get_max_threads());
    double* v = (double*)malloc(n*sizeof(double));
    double sumn = 0;
    time_init = walltime();

#pragma omp parallel for schedule(static) reduction(+:sumn)
    for(long int i=n; i>0; i--){
        v[i] = 1.0/((double)i*i);
        sumn += v[i];
    }
    printf("Error:\t\t %.16e\n",sum-sumn);
    printf("Time Elapsed:\t %f\n",walltime()-time_init);
    return 0;
}
```

3 Parallel - MPI

```
int main(int argc, char** argv){
    int rank = 0,size = 1;
    double pi = 4.0*atan(1), sum = pi*pi/6;
    double time_init;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0){
        printf("MPI   \tThreadcount: %i\n",size);
        if(argc < 2) {
            printf("Need one parameter, the size of the vector\n");
            MPI_Finalize();
            return 1;
        }else if(!isPowerOfTwo(size)){
            printf("The number of processors must be a power of 2\n");
            MPI_Finalize();
            return 1;
        }
    }
    int N = atoi(argv[1]), n = N/size;
    double sumn = 0.0;
    double* v = (double*)calloc(n,sizeof(double));
    time_init = walltime();

    int offset = rank*n;
    for(int i=n; i>0; i--){
        v[i] = 1.0/(((double)i+offset)*(i+offset));
        sumn += v[i];
    }
    double s2 = sumn;
    MPI_Reduce(&s2, &sumn, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

    if(rank == 0){
        printf("Error:\t\t %.16e\n",sum-sumn);
        printf("Time Elapsed:\t %f\n",walltime()-time_init);
```

```
37      }
38      MPI_Finalize();
39      return 0;
40  }
```

4 MPI-Calls

MPI_Init() and *MPI_Finalize()* are necessary to use. *MPI_Comm_size()* and *MPI_Comm_rank()* are very convenient to use. The communicator size could in principle be set constant inside the program, but this would limit the versatility of the program. It's possible to make do without finding the rank of each processor, but this would complicate the code quite a bit (if it is even possible to write this program without a call to *MPI_Comm_rank()*. *MPI_Reduce()* is also a convenient call to use to sum up the partial sums stores on each processor. Alternatively calls to *MPI_Send()* and *MPI_Recv()* could have been used.

5 Comparison

The difference $S - S_n$ should in general be the same each time the serial program is run. This is not the case for the parallel version as the resultant S_n depends on the order in which the partial sums on each processor is added together. Doing multiple runs with the parallel program will give up to $P!$ different sums, where P is the number of threads used in the program.

6 Memory

When $n >> 1$ the vector will dominate the memory. For the single processor program the vector will take up $n \cdot sizeof(double)$ of memory space. For the parallel-omp version of the program the vector will be copied for each processor, but for the parallel-mpi version the vector will be split up into equal chunks for each processor, so the memory requirement for each processor will be $n \cdot sizeof(double))/P$.

7 FLOPs

Looking at the line

```
18  v[i] = 1.0/((double)i*i);
```

we count 2 FLOPs, one division and one multiplication, for each element in v, so the total number of FLOPs required to generate v of length n is $2n$ for both the serial and openMP version of the program. For the generation of v in the MPI version the line reads

```
28  v[i] = 1.0/(((double)i+offset)*(i+offset));
```

which contains 4 FLOPs, one division, two additions and one multiplication. The additions can however be avoided by altering the code, but this was not done as addition is asuumed a cheap operation.

Given the elements in v, $n-1$ FLOPs are needed to summarize them in both serial and in parallel.

8 Parallell processing

The use of multiple local CPUs speed up the calculation almost linearly as a function of the number of cores, but on a network the limit factor will be the bandwith as relativley few FLOPs are needed to do each part of the calculation before sending the sum back. The use of more CPUs will also increase the error of the calculation, as there is less control of how the summing is done.