Lab Report to: Homework 3 - Matrix Multiplication Parallelization with OpenMP

Harald Höller, 0104177

11.04.2011

1 OpenMP Parallelization of Matrix Multiplication

1.1 Ex. 1: Implementation

The implementation of the OpenMP parallelization is simply based on the matrix multiplication algorithm ($\mathbf{AB} = \mathbf{C}$) from exercise 1. The main alterations concern the outermost for-loop with running index i where I added the OpenMP parallelization pragma. In order to get the maximum available number of threads as well as the actually used number of threads, I print out both omp_get_max_threads() and omp_get_num_threads(). Since I don't want that for the whole for-loop, I splitted the pragmas.

```
  /* MATRIX MULTIPLICATION */

   printf("Max number of threads: %i \n",omp_get_max_threads());
#pragma omp parallel
   printf("Number of threads: %i \n",omp_get_num_threads());
   c_1=time(NULL);  // time measure: start mm
#pragma omp parallel for private(m,j)
  for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
      C[i][j]=0.; // set initial value of resulting matrix C = 0
      for(m=0;m<N;m++) {
      C[i][j]=A[i][m]*B[m][j]+C[i][j];
      }
    }
  }
```

I tested this algorithm for small problem sizes with exemplary simple entries in the matrices **A** and **B**.

1.2 Ex. 2: Execution on o3800 with different number of threads

For really big problem sizes one would use a script like

```
#BSUB -q lva
#BSUB -o std_8.out
#BSUB -e std_8.err
export OMP_NUM_THREADS=8
/home/cb01/cb01298/hw_3/a.out
```

and send it as a job to the engine via `bsub > script`, however as we should use the `time` command to also measure the total execution, I chose as problem size $N = 1000$ in order to get reasonable job times on the frontend.

1 thread

```
$ export OMP_NUM_THREADS=1
$ time ./a.out
Max number of threads: 1
Number of threads: 1
Execution time: 91.000000

real    1m30.784s
user    1m30.462s
sys     0m0.092s
```

2 threads

```
$ export OMP_NUM_THREADS=2
b$ time ./a.out
Max number of threads: 2
Number of threads: 2
Number of threads: 2
Execution time: 45.000000

real    0m45.464s
user    1m30.448s
sys     0m0.061s
```

4 threads

```
$ export OMP_NUM_THREADS=4
$ time ./a.out
Max number of threads: 4
Number of threads: 4
Number of threads: 4
Number of threads: 4
Number of threads: 4
Execution time: 23.000000

real    0m22.905s
user    1m30.562s
sys     0m0.062s
```

8 threads

```
$ export OMP_NUM_THREADS=4
$ time ./a.out
Max number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Number of threads: 8
Execution time: 11.000000

real    0m11.687s
user    1m31.150s
sys     0m0.086s
```

Discussion

As one can see, the measures of the execution time itself $(1, 91), (2, 45), (4, 23), (8, 11)$ implemented directly in the code show very clearly a practically linear speedup. The time measurements using the `time` command are $(1, 90.5), (2, 45.5), (4, 22.9), (8, 11.7)$ supports this result perfectly. I plotted the inverse execution time against the number of threads to show this also graphically. We could norm this plot to the definition of the speedup

$$S_p = \frac{t_1}{t_p}$$

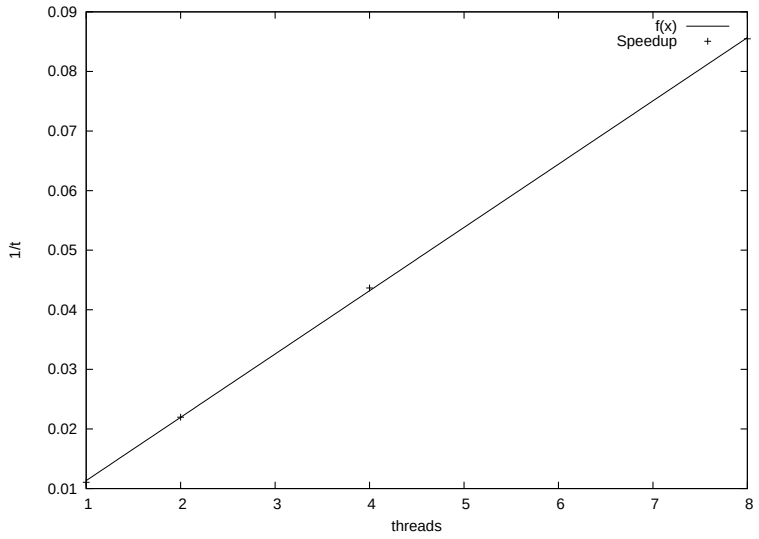

Abbildung 1: Speedup

by simply multiplying it with 90.5 of course. The speedup for 8 threads for example is then $S_8 = 7.735$. The efficicy is given by

$$e = \frac{1}{p}\frac{t_1}{t_p}$$

which is $(2, 0.994), (4, 0.987)$ and $(8, 0.966)$.

The results of this toy model show, that when practically no serial code consumes time, the efficiency of the overall execution time is very high.

Appendix

```
//
//  parallel+optim SS11
//  hw3: matrix multiplication parallelization
//  last change: 2011/04/11
// author: harald hoeller
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#define N 1000

  float A[N][N], B[N][N], C[N][N]; // declaring matrices of NxN size

int main ()
{

  /* DECLARING VARIABLES */
  int i, j, m; // indices for matrix multiplication
  float t_1; // Execution time measures
  clock_t c_1, c_2;

  /* FILLING MATRICES WITH RANDOM NUMBERS */
  srand ( time(NULL) );
  for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
      A[i][j]= (rand()%10);
      B[i][j]= (rand()%10);
    }
  }

  /* MATRIX MULTIPLICATION */

   printf("Max number of threads: %i \n",omp_get_max_threads());
#pragma omp parallel
   printf("Number of threads: %i \n",omp_get_num_threads());
   c_1=time(NULL);  // time measure: start mm
#pragma omp parallel for private(m,j)
// #pragma omp_set_num_threads(8)
  for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
      C[i][j]=0.; // set initial value of resulting matrix C = 0
      for(m=0;m<N;m++) {
      C[i][j]=A[i][m]*B[m][j]+C[i][j];
      }
//         printf("C: %f \n",C[i][j]);
    }
  }

  /* TIME MEASURE + OUTPUT */
  c_2=time(NULL);  // time measure: end mm
  t_1 = (float)(c_2-c_1); // time elapsed for job row-wise
  printf("Execution time: %f \n",t_1);

  /* TERMINATE PROGRAM */
  return 0;
}
```