Sheet 06

PS Parallel Programming

Patrick Wintner

April 28, 2025

1 Approximation of Pi

The execution times of different implementations of the Monte Carlo method are measured.

1.1 Source Code

1.1.1 Serial Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define _USE_MATH_DEFINES
#include <math.h>

int main() {
    long n = 700000000;
    long i, count = 0;
    double x, y, pi;
    double startTime, endTime;

    startTime = omp_get_wtime();

    srand((unsigned) time(NULL));
```

```
    for (i = 0; i < n; i++) {
        x = (double) rand() / RAND_MAX;
        y = (double) rand() / RAND_MAX;

        if (x * x + y * y <= 1) count++;
    }

    endTime = omp_get_wtime();

    pi = 4.0 * count / n;
    if(pi<0.99*M_PI || 1.01*M_PI<pi) {
        fprintf(stderr, "Error: estimated value deviates significantly: %f\n", pi);
        return 1;
    }
        printf("%2.4f\n", endTime-startTime);
    return 0;
}
```

1.1.2 Parallel Implementation using a Critical Section

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define _USE_MATH_DEFINES
#include <math.h>


int main() {
    long n = 700000000;
    long i, count = 0;
    double x, y, pi;
    double startTime, endTime;

    startTime = omp_get_wtime();

    #pragma omp parallel private(x, y)
    {
        unsigned seed = (unsigned) time(NULL)+13*omp_get_thread_num();
        #pragma omp for schedule (static)
        for (i = 0; i < n; i++) {
            x = (double) rand_r(&seed) / RAND_MAX;
```

```
            y = (double) rand_r(&seed) / RAND_MAX;

            if (x * x + y * y <= 1){
                #pragma omp critical
                {
                    count++;
                }
            }
        }
    }

    endTime = omp_get_wtime();

    pi = 4.0 * count / n;

    if(pi<0.99*M_PI|| 1.01*M_PI<pi) {
        fprintf(stderr, "Error: estimated value deviates significantly: %f\n", pi);
        return 1;
    }
        printf("%2.4f\n", endTime-startTime);
    return 0;
}
```

1.1.3 Parallel Implementation using an Atomic Statement

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define _USE_MATH_DEFINES
#include <math.h>

int main() {
    long n = 700000000;
    long i, count = 0;
    double x, y, pi;
    double startTime, endTime;

    startTime = omp_get_wtime();

    #pragma omp parallel private(x,y)
    {
```

```
18          unsigned seed = (unsigned) time(NULL) + 13*omp_get_thread_num();
19          #pragma omp for schedule (static)
20          for (i = 0; i < n; i++) {
21              x = (double) rand_r(&seed) / RAND_MAX;
22              y = (double) rand_r(&seed) / RAND_MAX;
23
24              if (x * x + y * y <= 1) {
25                  #pragma omp atomic
26                  count++;
27              }
28          }
29      }
30
31      endTime = omp_get_wtime();
32
33      pi = 4.0 * count / n;
34      if(pi<0.99*M_PI|| 1.01*M_PI<pi) {
35          fprintf(stderr, "Error: estimated value deviates significantly: %f\n", pi);
36          return 1;
37      }
38          printf("%2.4f\n", endTime-startTime);
39      return 0;
40  }
```

1.1.4 Parallel Implementation using a Reduction Clause

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include <omp.h>
5   #define _USE_MATH_DEFINES
6   #include <math.h>
7
8   int main() {
9       long n = 700000000;
10      long i, count = 0;
11      double x, y, pi;
12      double startTime, endTime;
13
14      startTime = omp_get_wtime();
15
16      #pragma omp parallel private(x,y)
```

```
17      {
18          unsigned seed = (unsigned) time(NULL)+13*omp_get_thread_num();
19          #pragma omp for reduction(+: count) schedule (static)
20          for (i = 0; i < n; i++) {
21              x = (double) rand_r(&seed) / RAND_MAX;
22              y = (double) rand_r(&seed) / RAND_MAX;
23
24              if (x * x + y * y <= 1) count++;
25          }
26      }
27
28      endTime = omp_get_wtime();
29
30      pi = 4.0 * count / n;
31      if(pi<0.99*M_PI|| 1.01*M_PI<pi) {
32          fprintf(stderr, "Error: estimated value deviates significantly: %f\n", pi);
33          return 1;
34      }
35          printf("%2.4f\n", endTime-startTime);
36      return 0;
37  }
```

1.2 Measurement Method

The measurement was done on the LCC3 cluster by calling `sbatch job.sh serial 3`, `sbatch job.sh critical 3`, `sbatch job.sh atomic 3` and `sbatch job.sh reduction 3`.

The following scripts are involved in the experiment.

1.2.1 SLURM Script

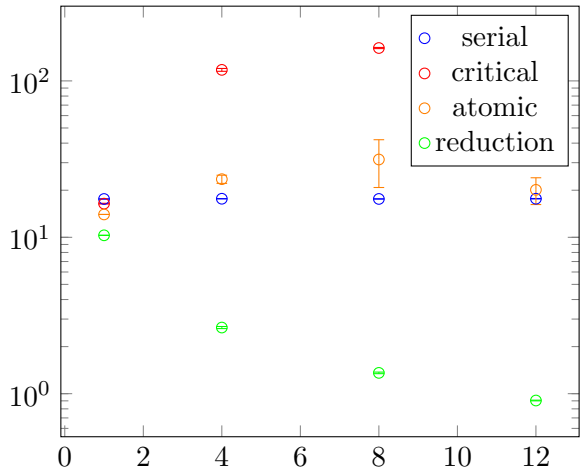
```
1  #!/bin/bash
2  # usage: sbatch [slurm_options] <executable> <number_of_measurements>
3
4  # Execute job in the partition "lva" unless you have special requirements.
5  #SBATCH --partition=lva
6  # Name your job to be able to identify it later
7  #SBATCH --job-name csba4017
8  # Redirect output stream to this file
9  #SBATCH --output=output.log
```

```
# Maximum number of tasks (=processes) to start in total
#SBATCH --ntasks=1
# Maximum number of tasks (=processes) to start per node
#SBATCH --ntasks-per-node=1
# Enforce exclusive node allocation, do not share with other jobs
#SBATCH --exclusive

./benchmark.sh $1 $2
```

1.2.2 Benchmark Script

```
#!/bin/bash
# Usage: ./benchmark.sh <executable> <number_of_measurements>

results=$1".dat"
echo "x y ey" > $results # create header
make
for i in {1,4,8,12} # number of threads
do
        measurements=$i"_"$1".log"
        export OMP_NUM_THREADS=$i
        for j in $(seq 1 $2) # repeat measurement £2 times
        do
                ./$1 >> $measurements # store measurement results in <executable>.log
        done
        ./toTable $measurements $results $2 $i #store table in <executable>.dat
        rm $measurements
done
make clean
```

1.3 Experiment Results



1.4 Discussion

Unsurpisingly, there is no performance impact when using different numbers of threads for the serial implementation. The performance of the parallel implementation with an atomic statement is roughly equal to the performance of the parallel implementation and slightly detoriates with an increasing number of threads. Considering that only the variable assignments to x and y (calling rand_r) runs in parallel, while incrementing the counter is done serial, this is not really surprising. The performance of the parallel implementation with a critical section detoriates siginficantly when increasing the number of threads. Only the implementation using a reduction clause gains performance by increasing the number of threads. This mets the expections, because each thread has its own count, thus not slowing each other when incrementing the counter.

2 Mandelbrot

The execution time of an parallel implementation using openOMP of the mandelbrot set is measured.

2.1 Source Code

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

// Include that allows to print result as an image
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

// Default size of image
#define X 1280
#define Y 720
#define MAX_ITER 10000

void calc_mandelbrot(uint8_t image[Y][X]) {
        size_t i, j;
        #pragma omp parallel for schedule(guided) private(j)
        for(i=0; i<Y; ++i) {
                for(j=0; j<X; ++j) {
                        double x=0;
                        double y=0;
                        double cx=(double)j/(X-1)*3.5-2.5; // scale j to [-2.5, 1]
                        double cy=(double)i/(Y-1)*2-1; // scale i to [-1. 1]
                        size_t iteration=0;
                        while(x*x+y*y<2*2 && iteration<MAX_ITER) {
                                double x_tmp=x*x-y*y+cx;
                                y=2*x*y+cy;
                                x=x_tmp;
                                iteration=iteration+1;
                        }
                        // scale iteration to [0,255]
                        char norm_iteration=iteration*255/MAX_ITER;
                        image[i][j]=norm_iteration;
                }
        }
}

int main() {
        uint8_t image[Y][X];

```

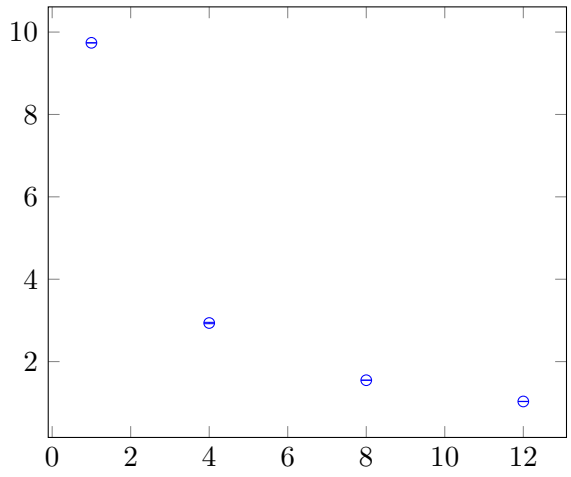
```
42      double startTime = omp_get_wtime();
43          calc_mandelbrot(image);
44          double endTime = omp_get_wtime();
45          printf("%2.4f\n", endTime-startTime);
46
47          const int channel_nr = 1, stride_bytes = 0;
48          stbi_write_png("mandelbrot.png", X, Y, channel_nr, image, stride_bytes);
49          return EXIT_SUCCESS;
50  }
```

2.2 Measurement Method

The measurement was done on the LCC3 cluster by calling `sbatch job.sh mandelbrot 3`.

The same scripts as in exercise 1 are involved in the experiment.

2.3 Experiment Results



2.4 Discussion

The execution time drops significantly when the number of threads is increased. Considering that threads are completely independent here, this mets the expections.