Sheet 06

PS Parallel Programming

Patrick Wintner

April 28, 2025

1 Monte Carlo

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

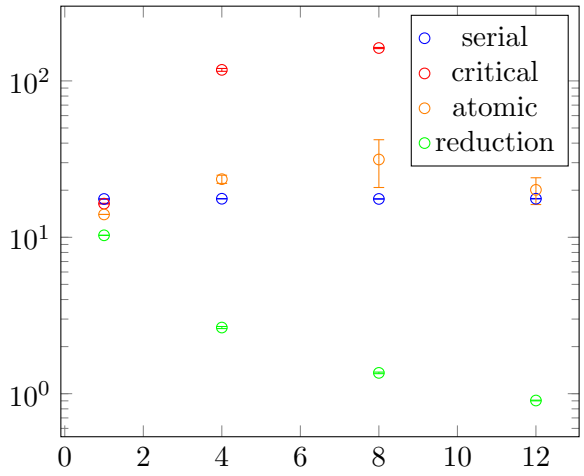
```
1  #!/bin/bash
2  # usage: sbatch [slurm_options] <executable> <number_of_measurements>
3
4  # Execute job in the partition "lva" unless you have special requirements.
5  #SBATCH --partition=lva
6  # Name your job to be able to identify it later
7  #SBATCH --job-name csba4017
8  # Redirect output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

```

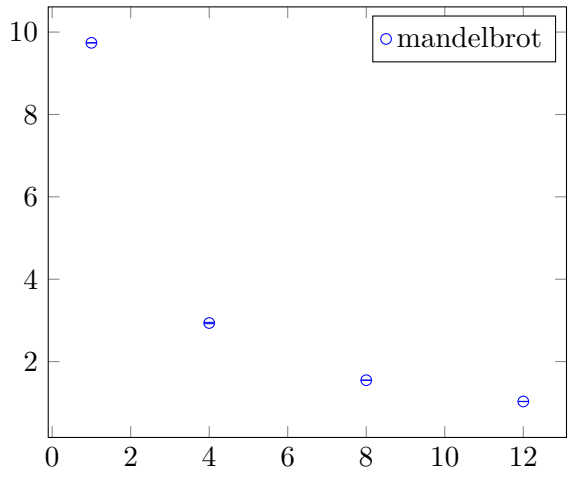
```
42      double startTime = omp_get_wtime();
43          calc_mandelbrot(image);
44          double endTime = omp_get_wtime();
45          printf("%2.4f\n", endTime-startTime);
46
47          const int channel_nr = 1, stride_bytes = 0;
48          stbi_write_png("mandelbrot.png", X, Y, channel_nr,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

3 Loop Scheduling Methods

The impact of using different scheduling methods on openOMP implementations of the Hadamard product and the Mandelbrot set and the impact of increasing the size of the matrix on the execution time of the Hadamard product are observed.

3.1 Examples of Loop Scheduling Methods

3.1.1 static

Divides the loop into equal-sized chunks.

3.1.2 dynamic

The iterations are broken into chunks of the specified size. When a thread finishes the execution of a chunk, the next chunk is assigned to that thread.

3.1.3 guided

Similar to dynamic, but decreases chunk sizes over time to reduce overhead during runtime.

3.1.4 runtime

The scheduling type and the chunk size is taken from the OMP_SCHEDULE environment variable.

3.1.5 auto

The scheduling type is left up the system.

3.2 Source Code

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

#ifndef SCHEDULING
    #define SCHEDULING auto
#endif

void matrix_free(size_t n, double **m) {
    for(size_t i=0; i<n; ++i) {
        free(m[i]);
    }
    free(m);
}
double **matrix_init(size_t n) {
    double** m=malloc(n*sizeof(*m));
    if(!m) {
        perror("malloc");
        return NULL;
    }
    srand(time(NULL));
    for(size_t i=0; i<n; ++i) {
        m[i] = malloc(n*sizeof(**m));
        if(!m[i]) {
            matrix_free(i, m);
            perror("malloc");
            return NULL;
        }
        for(size_t j=0; j<n; ++j) {
            m[i][j]=rand()/RAND_MAX;
        }
    }
    return m;
}
void hadamard(size_t n, double **a, double **b, double **c) {
    size_t i, j;
    #pragma omp parallel for schedule(SCHEDULING) private(j)
    for(i=0; i<n; ++i) {
        for(j=0; j<n; ++j) {
```

```
42              c[i][j] = a[i][j]*b[i][j];
43          }
44      }
45  }
46
47  int main(int argc, char** argv) {
48      if(argc!=2) {
49          fprintf(stderr, "Usage: %s <n>\n", *argv);
50          return EXIT_FAILURE;
51      }
52
53      char* end;
54      size_t n=(size_t) strtoll(*(argv+1), &end, 10);
55      if(*end || errno) {
56          perror("strtol");
57          return EXIT_FAILURE;
58      }
59
60      double **a=NULL, **b=NULL, **c=NULL;
61      a=matrix_init(n);
62      if(!a) {
63          return EXIT_FAILURE;
64      }
65      b=matrix_init(n);
66      if(!b) {
67          matrix_free(n, a);
68          return EXIT_FAILURE;
69      }
70      c=matrix_init(n);
71      if(!c) {
72          matrix_free(n, a);
73          matrix_free(n, b);
74          return EXIT_FAILURE;
75      }
76      double startTime = omp_get_wtime();
77      hadamard(n, a, b, c);
78          double endTime = omp_get_wtime();
79          printf("%2.4f\n", endTime-startTime);
80      matrix_free(n, a);
81      matrix_free(n, b);
82      matrix_free(n, c);
83      return EXIT_SUCCESS;
84  }
```

```c
#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

#ifndef SCHEDULING
    #define SCHEDULING auto
#endif

void calc_mandelbrot(uint8_t image[Y][X]) {
        size_t i, j;
        #pragma omp parallel for schedule(SCHEDULING) 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];
```

```
45
46      double startTime = omp_get_wtime();
47          calc_mandelbrot(image);
48          double endTime = omp_get_wtime();
49          printf("%2.4f\n", endTime-startTime);
50
51          const int channel_nr = 1, stride_bytes = 0;
52          stbi_write_png("mandelbrot.png", X, Y, channel_nr, image, stride_bytes);
53          return EXIT_SUCCESS;
54  }
```

3.3 Measurement Method

The measurement was done on the LCC3 cluster by calling `sbatch job.sh benchmark_scheduling mandelbrot` `sbatch job.sh benchmark_scheduling hadamard 3`, `sbatch job.sh benchmark_threads hadamard 3` and `sbatch job.sh benchmark_matrix_size hadamard 3`.

The following scripts are involved in the experiment.

3.3.1 SLURM Script

```
1   #!/bin/bash
2   # usage: sbatch [slurm_options] <skript> <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
10  # Maximum number of tasks (=processes) to start in total
11  #SBATCH --ntasks=1
12  # Maximum number of tasks (=processes) to start per node
13  #SBATCH --ntasks-per-node=1
14  # Enforce exclusive node allocation, do not share with other jobs
15  #SBATCH --exclusive
16
17  ./$1 $2 $3
```

3.3.2 Benchmark Scripts

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

export OMP_NUM_THREADS=12
dim=10000
results=$1"_scheduling.dat"
echo "x y ey" > $results # create header
for i in {"static","dynamic","guided","auto","runtime"} # loop scheduling method
do
        make OPT=$i
        measurements=$i"_"$1".log"
        for j in $(seq 1 $2) # repeat measurement £2 times
        do
                ./$1 $dim >> $measurements # store measurement results in <executable>.l
        done
        ./toTable $measurements $results $2 $i #store table in <executable>.dat
        rm $measurements
        make clean
done
```

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

make OPT="static"
dim=10000
results=$1"_threads.dat"
echo "x y ey" > $results # create header
for i in {1,4,8,12} # number of threads
do
        measurements=$i"_"$1".log"
        export OMP_NUM_THREADS=$i
        for j in $(seq 1 $2) # repeat measurement £2 times
        do
                ./$1 $dim >> $measurements # store measurement results in <executable>.l
        done
        ./toTable $measurements $results $2 $i #store table in <executable>.dat
        rm $measurements
done
make clean
```

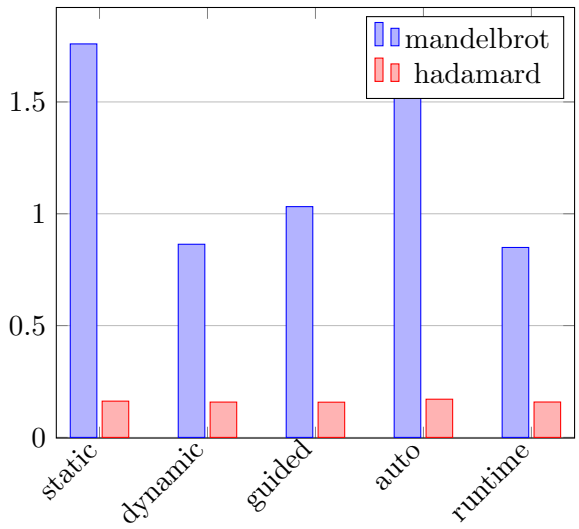
```
#!/bin/bash
```

```
# Usage: ./benchmark_matrix_size.sh <executable> <number_of_measurements>

make OPT="static"
export OMP_NUM_THREADS=12
results=$1"_matrix_size.dat"
echo "x y ey" > $results # create header
for i in {10000,20000,30000} # size of matrix
do
        measurements=$i"_"$1".log"
        for j in $(seq 1 $2) # repeat measurement £2 times
        do
                ./$1 $i >> $measurements # store measurement results in <executable>.log
        done
        ./toTable $measurements $results $2 $i #store table in <executable>.dat
        rm $measurements
done
make clean
```

3.4 Experiment Results

3.5 Scheduling Method

The number of threads is set to 12, the size of the matrix is 10.000^2

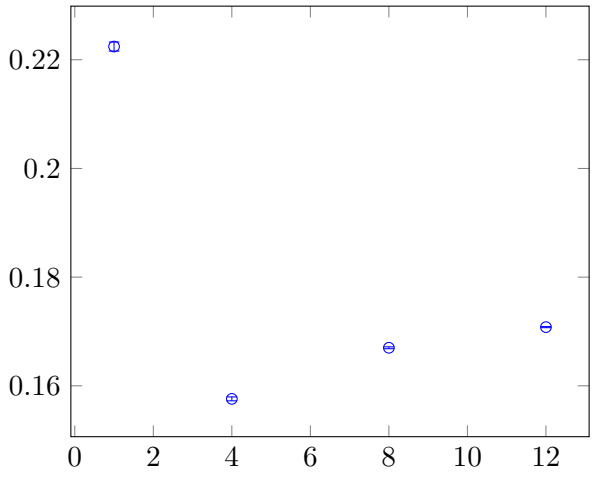


The scheduling method has no impact on the hadamard product. The performance of

the implementation of the mandelbrot set is improved by dynamic and similar scheduling methods.

3.6 Number of Threads

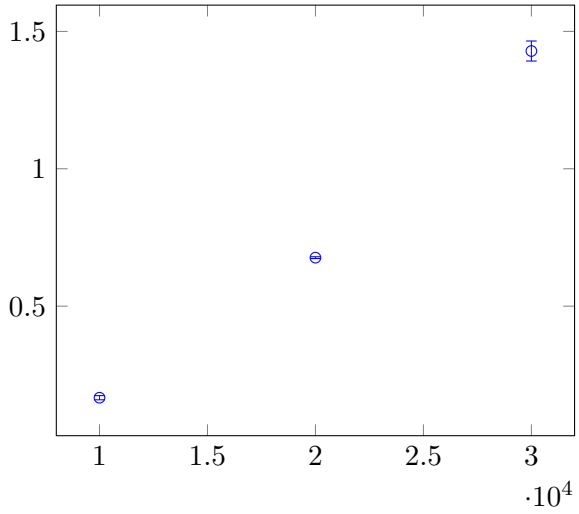
The used scheduling method is static, the size of the matrix is set to 10.000^2.



The best performance can be achieved by setting the number of threads to four.

3.7 Size of Matrix

The used scheduling method is static, the number of threads is set to 12.



The execution time increases linearily wwith the length of the matrix.