

## Sheet 06

# PS Parallel Programming

Patrick Wintner

April 27, 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

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

```

17     for (i = 0; i < n; i++) {
18         x = (double) rand() / RAND_MAX;
19         y = (double) rand() / RAND_MAX;
20
21         if (x * x + y * y <= 1) count++;
22     }
23
24     endTime = omp_get_wtime();
25
26     pi = 4.0 * count / n;
27     if(pi<0.99*M_PI || 1.01*M_PI<pi) {
28         fprintf(stderr, "Error: estimated value deviates significantly: %f\n", pi);
29         return 1;
30     }
31     printf("%.2.4f\n", endTime-startTime);
32     return 0;
33 }

```

### 1.1.2 Parallel Implementation using a Critical Section

```

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
9  int main() {
10     long n = 700000000;
11     long i, count = 0;
12     double x, y, pi;
13     double startTime, endTime;
14
15     startTime = omp_get_wtime();
16
17     #pragma omp parallel private(x, y)
18     {
19         unsigned seed = (unsigned) time(NULL)+13*omp_get_thread_num();
20         #pragma omp for schedule (static)
21         for (i = 0; i < n; i++) {
22             x = (double) rand_r(&seed) / RAND_MAX;

```

```

23         y = (double) rand_r(&seed) / RAND_MAX;
24
25         #pragma omp critical
26         {
27             if (x * x + y * y <= 1) count++;
28         }
29     }
30 }
31
32 endTime = omp_get_wtime();
33
34 pi = 4.0 * count / n;
35
36 if(pi<0.99*M_PI|| 1.01*M_PI<pi) {
37     fprintf(stderr, "Error: estimated value deviates significantly: %f\n", pi);
38     return 1;
39 }
40 printf("%.4f\n", endTime-startTime);
41 return 0;
42 }

```

### 1.1.3 Parallel Implementation using an Atomic Statement

```

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 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("%.24f\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 `./job.sh serial 3`, `./job.sh critical 3`, `./job.sh atomic 3` and `./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
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 ./benchmark.sh $1 $2

```

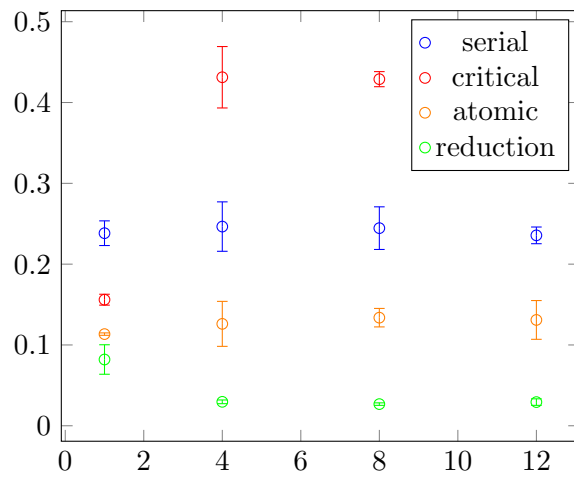
## 1.2.2 Benchmark Script

```

1  #!/bin/bash
2  # Usage: ./benchmark.sh <executable> <number_of_measurements>
3
4  results=$1".dat"
5  echo "x y ey" > $results # create header
6  make
7  for i in {1,4,8,12} # number of threads
8  do
9      measurements=$i"_"$1".log"
10     export OMP_NUM_THREADS=$i
11     for j in $(seq 1 $2) # repeat measurement l2 times
12     do
13         ./ $1 >> $measurements # store measurement results in <executable>.log
14     done
15     ./toTable $measurements $results $2 $i #store table in <executable>.dat
16     rm $measurements
17 done
18 make clean

```

### 1.3 Experiment Results



### 1.4 Discussion