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

```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

```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();

    #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;

            #pragma omp critical
            {
                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;
}
```

1.1.3 Parallel Implementation using an Atomic Statement

```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();

    #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 atomic
                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;
}
```

1.1.4 Parallel Implementation using a Reduction Clause

```
#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();
```

```c
        #pragma omp for reduction(+: count) 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) 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;
}
```

1.2 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

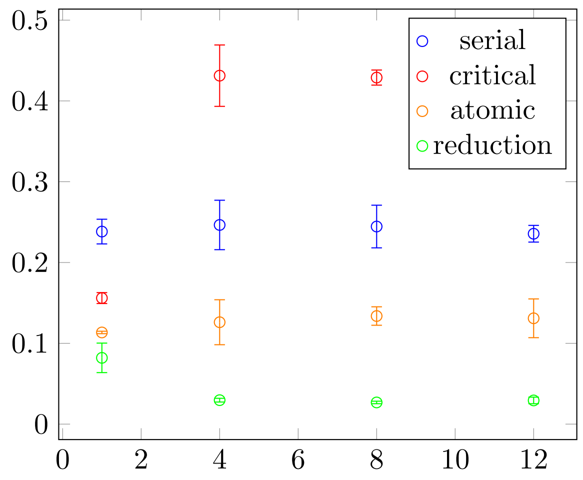
```bash
#!/bin/bash
# usage: sbatch [slurm_options] <executable> <number_of_measurements>

# Execute job in the partition "lva" unless you have special requirements.
#SBATCH --partition=lva
# Name your job to be able to identify it later
#SBATCH --job-name csba4017
# Redirect output stream to this file
#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