CPA Lab-Report
Lab 2 Prime Numbers

Simon Birrer, Dominic Schürmann

November 21, 2015

Date Performed:	October 27, 2015
Students:	Simon Birrer
	Dominic Schürmann
Instructor:	Francisco Javier Piris Ruano

Contents

1 Biggest prime storable in 8 bytes

The Source for the solution is in the file *primo_grande.c*. In the figure 1 you will see the changes applied to function primo of the original code.

```
1    int numberOfThreads;
2    int offset;
3    #ifdef _OPENMP
4    #pragma omp parallel
5    numberOfThreads = omp_get_num_threads();
6    offset = 2 * numberOfThreads;
7    #else
8    numberOfThreads = 1;
9    offset = 2;
10   #endif
11
12    #pragma omp parallel private(i)
13    {
14      #ifdef _OPENMP
15      int threadIndex =  omp_get_thread_num();
16      int startIndex = (2 * threadIndex) + 3;
17      #else
18      int startIndex = 3;
19      #endif
20      for (i = startIndex; p && i <= s; i += offset)
21        if (n % i == 0) p = 0;
22    }
```

Figure 1: code changes primo grande

1.1 Compiling without OpenMP

To use the program in both ways, either with or without OpenMP , we used the preprocessor directives. Now the compiler decides upon the arguments if the code will use OpenMP or not for the passages where OpenMP function calls will happen.

This can be seen in figure 1 in lines 3-10 and 14-19.

1.2 Time measurement of parallelized version

In figure 2 are the measured times of executing the program with different numbers of threads using kahan.

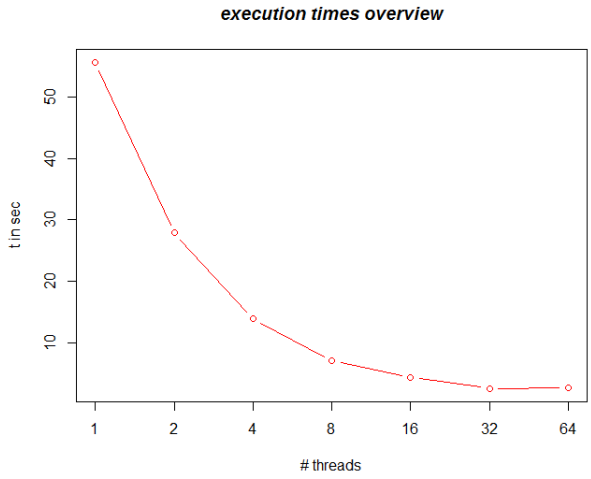

Figure 2: execution times for exercise 1.2

Since a node of kahan has 32 cores, the execution with 32 threads was the fastest. In addition the performance decreases if the number of threads will be increased to more than 32 threads. This is shown in figure 2 and the overhead is even more visible in figure 3 . As a result of this, the best speedup will be achieved with one thread for each processor.

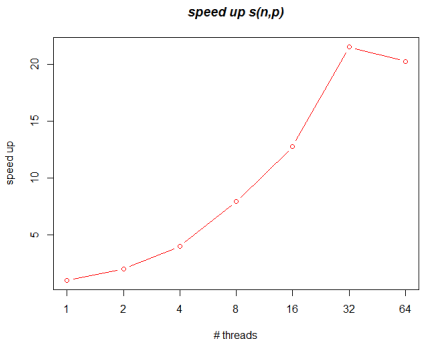

Figure 3: speed up for exercise 1.2

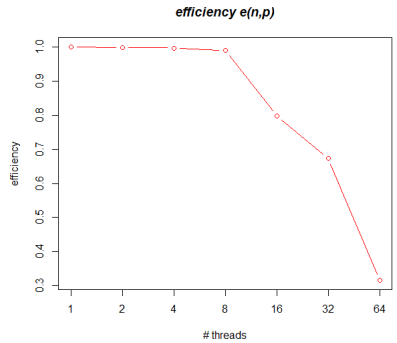

Figure 4: efficiency graph of exercise 1.2

2 Count primes in a range

In all code examples, where a time measurement was needed, following code was used to measure the time in seconds:

```
#ifdef _OPENMP
double t1 = omp_get_wtime();
#endif

//code for algorithm here

#ifdef _OPENMP
double t2 = omp_get_wtime();
printf("looptime: %f seconds \n", t2-t1);
#endif
```

Figure 5: execution time measurement

2.1 Measurement sequential execution

To measure the sequential time for the *primo_numeros_.c* code, we compiled the code with OpenMP. To achieve a sequential execution we defined the environment variable $OMP\_NUM\_THREADS = 1$.
The sequential execution took 289.24 seconds.

2.2 Measurement parallel most inner loop

Based on the exercise a measurement for the most inner loop does not make much sense, because it is really inefficient. Every execution without an if clause in the pragma omp definiton was terminated because of the walltime exceeding.

2.3 Measurement parallel most outer loop

The most efficient way to speedup the given exercise code is by parallelize the most outer loop. The source is in the file *primo_numeros_reduction.c*. The following code changes where made to achieve this:

```
1  int numberOfThreads;
2  #pragma omp parallel
3  numberOfThreads = omp_get_num_threads();
4  n = 2; /* Por el 1 y el 2 */
5  #pragma omp parallel for reduction(+:n) schedule(runtime)
6  for (i = 3; i <= N; i += 2){
7      if (primo(i))
8      {
9          n++;
10     }
11 }
```

Figure 6: parallelization of the most outer loop

The environment variable *OMP_NUM_THREADS* was set to 32 Threads to achieve the best speedup.

2.3.1 scheduling distribution

We checked the execution time for the same code with different scheduling distributions. Therefore we changed the environment variable *OMP_SCHEDULE* .

schedule type	chunk size	execution time in [s]
static	0	17.58
static	1	13.58
dynamic	0	14.18
dynamic	1	14.23
dynamic	2	13.52

Table 1: measurement results for different schedule distribution

2.4 Exercise 1: Without reduction clause

The whole parallelization can be done without using a reduction for the variable n. In this code segment we used an OpenMP atomic clause to achieve the same behaviour. The source is in the file *primo_numeros_without_reduction.c*.

```
1  int numberOfThreads;
2  #pragma omp parallel
3  numberOfThreads = omp_get_num_threads();
4  n = 2; /* Por el 1 y el 2 */
5  #pragma omp parallel for schedule(runtime)
6  for (i = 3; i <= N; i += 2){
7      if (primo(i))
8      {
9          #pragma omp atomic
10         n++;
11     }
12 }
```

Figure 7: parallelization without reduction

2.5 Exercise 2: Printing found primes for each thread

To find out which thread found how many prime numbers, we had to save the information for each thread in a vector. We use a vector based on the index of each thread and therefore we do not have to care about race conditions or any other problem based on public variables. The source is in the file *primo_numeros_threadinfo.c*.

```
int numberOfThreads;
#pragma omp parallel
numberOfThreads = omp_get_num_threads();
n = 2; /* Por el 1 y el 2 */

int* primesFoundPerThread =
  (int*) calloc(numberOfThreads, sizeof(int));

for(i = 0; i < numberOfThreads; i++){
  primesFoundPerThread[i] = 0;
}

#pragma omp parallel for
for (i = 3; i <= N; i += 2){

    if (primo(i))
    {
      int threadId = omp_get_thread_num();
      primesFoundPerThread[threadId]++;
       #pragma omp atomic
       n++;
    }
}

for(i = 0; i < numberOfThreads; i++){
  printf("thread %i: %i primes\n",i,primesFoundPerThread[i]);
}
```

Figure 8: printing found primes by each thread

If we run this code with a static distribution we see, that thread zero computes the most prime numbers and thread 31 computes the least amount of numbers. Based on the prime number theory ,the bigger the numbers to look for primes are, the less prime numbers can be found. This reflects in the results of out measurements.

3 Appendix

3.1 Formulas

Speed Up

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

Efficiency

$$E(n,p) = \frac{S(n,p)}{p}$$

3.2 Measurements of exercise 1

number or threads	1	2	4	8	16	32	64
time in seconds	55.571	27.850	13.931	7.019	4.349	2.580	2.741

Table 2: execution time exercise 1

number or threads	1	2	4	8	16	32	64
speed up S(n,p)	1.000	1.995	3.989	7.917	12.779	21.539	20.273

Table 3: speed up exercise 1

number or threads	1	2	4	8	16	32	64
efficiency E(n,p)	1.000	0.998	0.997	0.990	0.799	0.673	0.317

Table 4: efficiency exercise 1

List of Figures

List of Tables