Università della Svizzera italiana

Institute of Computing CI

**High-Performance Computing** **2022**

Student: Fabio Marchesi Discussed with: Gioele De Pianto, Raffaele Morganti

## Solution for Project 2

Due date: 26.10.2022 (midnight)

# 1. Parallel reduction operations using OpenMP [10 points]

In this section, we are required to use the OpenMP directives to speed up the computation of the dot product of two vectors: $\alpha = a^T \cdot b$, where $a, b \in R^N$. In particular, we are required to compare the performance of the sequential version of the algorithm to two parallel versions that use respectively the directives `reduction` and `critical`.

- **Reduction**: This directive allows us to specify a variable and an operation. Each thread will receive its own private copy of the specified variable, the operation will be used to understand the default value of that variable. When all the threads will be done, then all the private copies will be merged using the operation specified in the directive. Our code looks like this:

```
alpha_parallel = 0.0;
#pragma omp parallel for reduction (+:alpha_parallel)
for (int i = 0; i < N; i++) {
        alpha_parallel += a[i] * b[i];
}
```

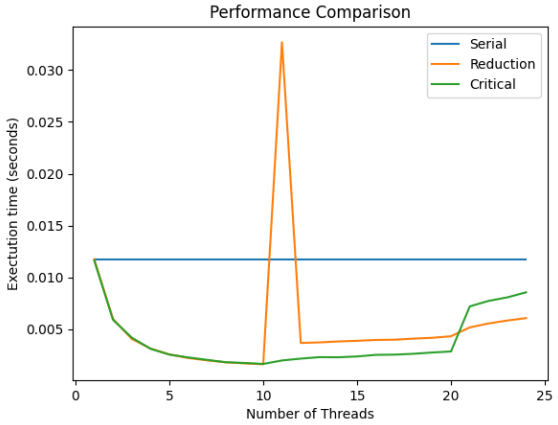
  Since the used operation is `+` all the private copies of `alpha_parallel` will be initialized to 0.0. After the for loop will remain only one copy of `alpha_parallel` which will contain the correct result.

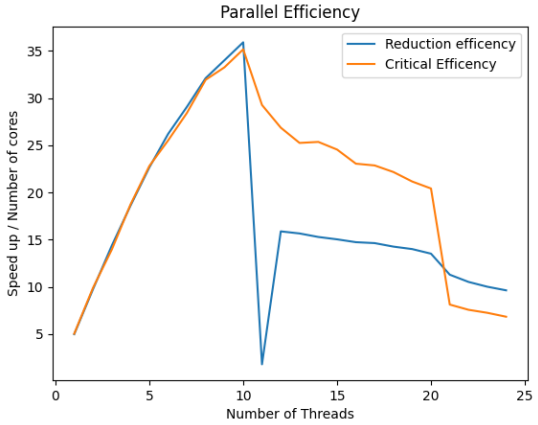
- **Critical**: This directive needs to be inserted into a parallel region, and all the instructions inside its scope will be executed in a serial way. It's important that we don't just update a shared copy of the sum in the foor-loop inside the critical scope because in this way we will have a computation that is serial but with the added overhead of the threads' management. In order to correctly use this directive, it's necessary to make a private variable for each thread that will contain the partial sum computed by each thread, and, only in the end, update the shared sum using the directive `critical`.

```
alpha_parallel = 0.0;
#pragma omp parallel
{
        double thread_sum = 0.0;
        #pragma omp for
        for (int i = 0; i < N; i++) {
                thread_sum += a[i] * b[i];
        }
        #pragma omp critical
        alpha_parallel += thread_sum;
}
```

In order to compare the parallel versions to the serial one, we computed the executions time for each version using $N = 10^5, 10^6, 10^7, 10^8, 10^9$, and for each value of $N$ we computed 100 iterations. The parallel versions were tested using $t$ threads, with $t = 1, 2, \ldots, 24$. We tried up to 24 threads because it was required from the project, but the CPU on the cluster has only 20 cores, for this reason having more than 20 threads will only increase the overhead and not the performance. Indeed, looking at all the plots, we see that the performance gets worse having more than 20 threads.

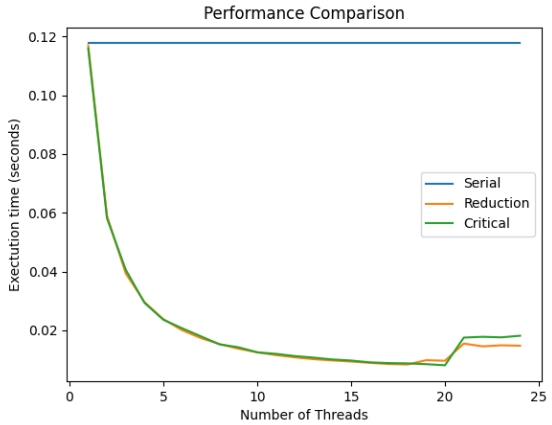
- $N = 10^5$

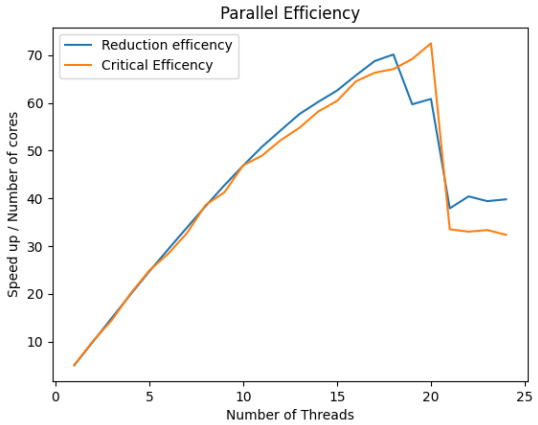


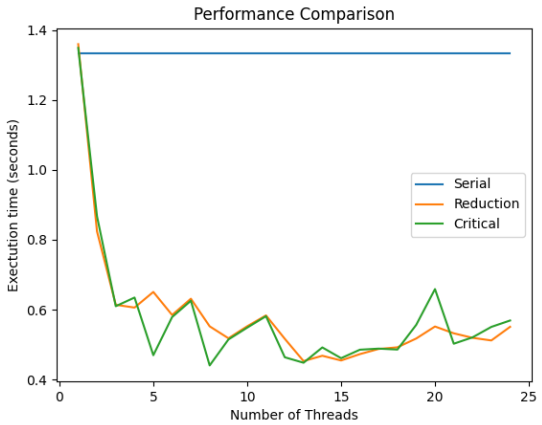

This value of $N$ is the only one where the lines relative to the parallelized versions are higher than the sequential version, also if only for one value of $t$. In particular, the version that uses the `reduction` directive appears to be slower than the sequential version for $t = 11$. The most probable reason is that the cluster was doing other computations at that moment, or that the overhead generated by the need to manage all the threads was really high.

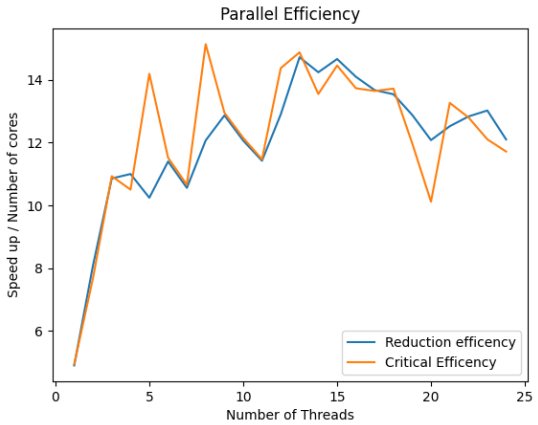
- $N = 10^6$



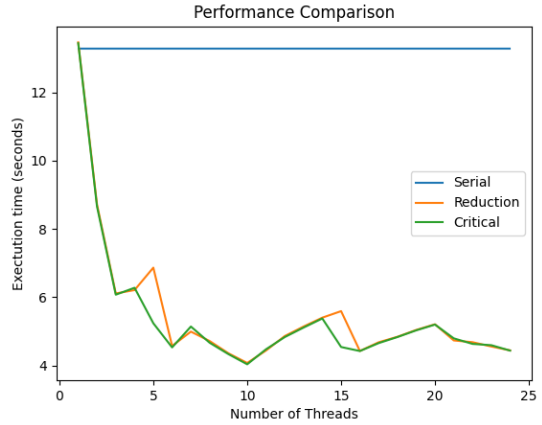



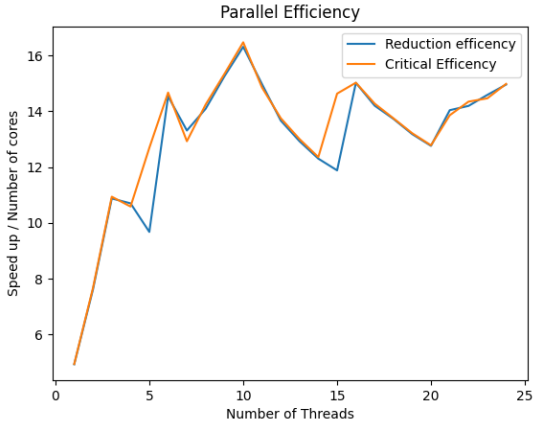

- $N = 10^7$



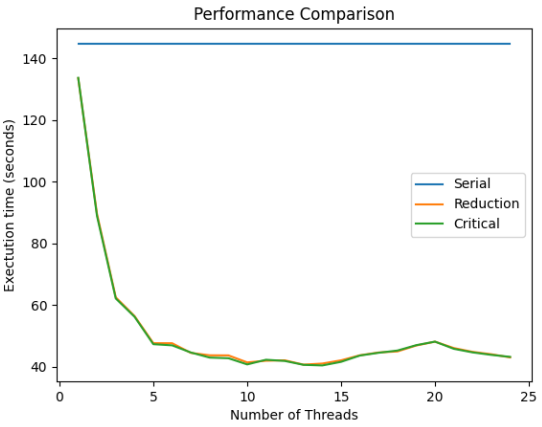



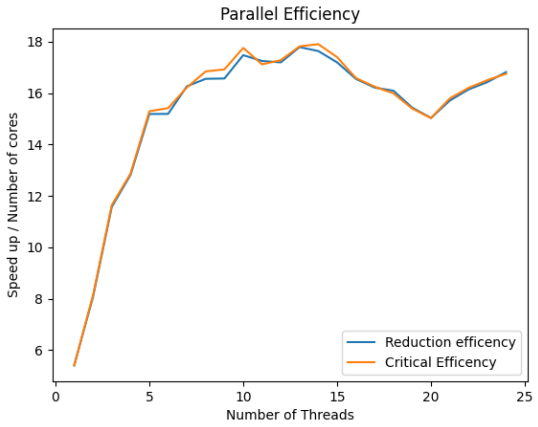

- $N = 10^8$




- $N = 10^9$




If we look at all this plots, we can see that when we only have one thread the performance is almost identical to the serial version of the algorithm, but when the number of threads starts to increase the execution time drops drastically. At the same time, we can see that the improvement of each added thread decreases, indeed, the first four added threads (so in total we have 5 threads) make a much bigger difference than the other 20. This is due to the fact that each thread execution's needs to be managed, adding overhead. Indeed, we can also see that (for example) with $N = 10^9$ using 15 threads, the execution time is lower than using 20 threads. For each array size and for each number of threads, it's also been plotted the parallel efficiency, which is computed as $\frac{speed\_up}{\#cores}$.

This metric can be used to estimate how much of the computing capacity is actually used to carry out a calculation. From those plots we can deduce that the highest parallel efficiency is obtained when the number of threads is high but always a bit less than 24. Indeed, for the smallest array, due to the high overhead the peak is at around 10 threads, while in the other instances it comes with more threads.

## 2. The Mandelbrot set using OpenMP [30 points]

We are asked to compute the Mandelbrot set, in particular we need to compute an image, where these images are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each number as image coordinates, pixels are colored according to how rapidly the sequence diverges, if at all. The sequential code used to compute the Mandelbrot set looks like this.

```
  for (j = 0; j < IMAGE_HEIGHT; j++) {
    cx = MIN_X;
    for (i = 0; i < IMAGE_WIDTH; i++) {
          n = 0;
          x2 = 0.0, y2 = 0.0;
          while (x2 * x2 + y2 * y2 <= 2 * 2 && n < MAX_ITERS) {
                  double xtemp = x2 * x2 - y2 * y2 + cx;
                  y2 = 2 * x2 * y2 + cy;
                  x2 = xtemp;
                  n++;
          }
          nTotalIterationsCount += n;
      int c = ((long)n * 255) / MAX_ITERS;
      png_plot(pPng, i, j, c, c, c);
      cx += fDeltaX;
    }
    cy += fDeltaY;
  }
```

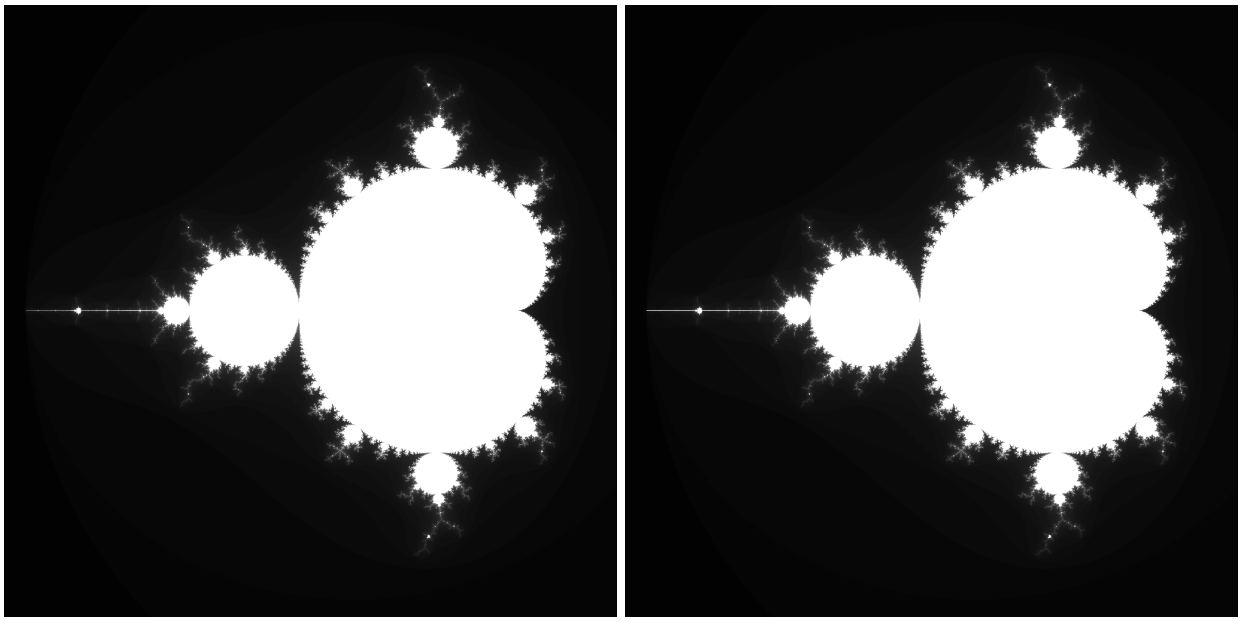
The number of pixels and the number of iterations are parameter, increasing the number of iterations we get a more accurate gradient. In the code we have that for each pixel $(i, j)$ is mapped into a complex number, and then we check how fast its sequence diverges with a while-loop. The number of iterations before the sequence diverges is used to choose the color of the pixel $(i, j)$. We can exploit the fact that the computation of each pixel is independent.

```
#pragma omp parallel private(i, j, n, cx, cy, x2, y2)
{
  long my_iterations = 0;
#pragma omp for
  for (j = 0; j < IMAGE_HEIGHT; j++) {
    cx = MIN_X;
    cy = MIN_Y + j * fDeltaY;
    for (i = 0; i < IMAGE_WIDTH; i++) {
      n = 0;
      x2 = 0.0, y2 = 0.0;
      while (x2 * x2 + y2 * y2 <= 2 * 2 && n < MAX_ITERS) {
        double xtemp = x2 * x2 - y2 * y2 + cx;
        y2 = 2 * x2 * y2 + cy;
        x2 = xtemp;
        n++;
      }
      my_iterations += n;
      int c = ((long)n * 255) / MAX_ITERS;
      png_plot(pPng, i, j, c, c, c);
      cx += fDeltaX;
    }
  }
#pragma omp critical
```

```
  nTotalIterationsCount += my_iterations;
}
```

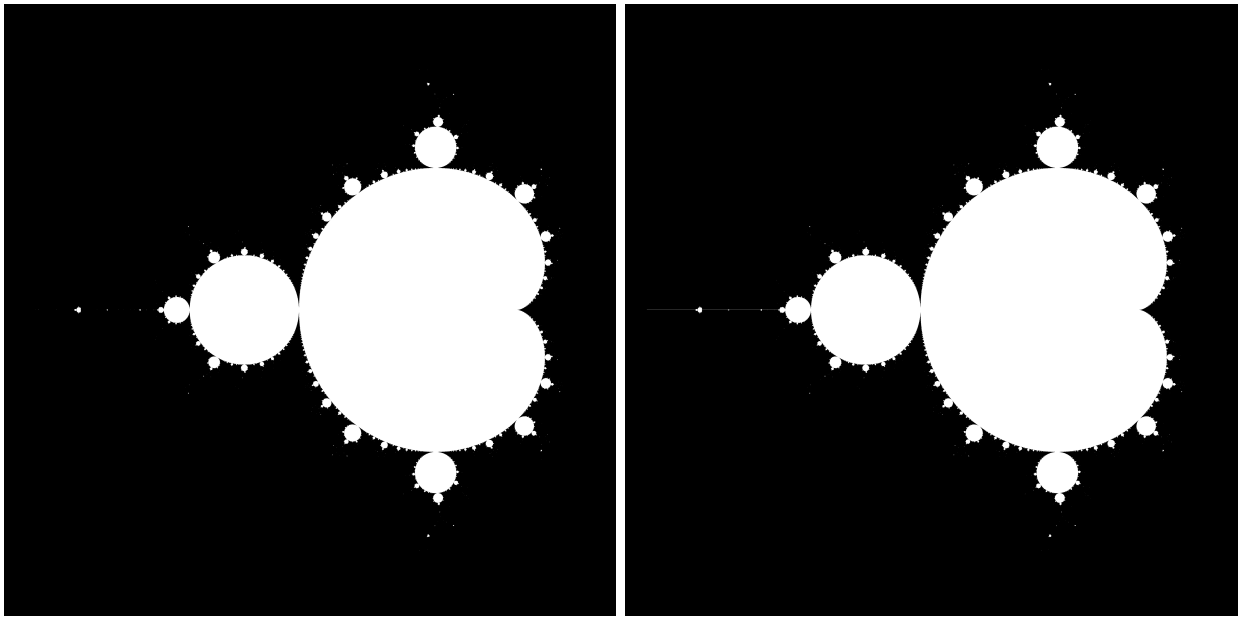
What happens here is that the computation of each pixel is done independently, so we need to change the way that $cy$ is calculated, indeed, we can't increment it each time by $fDeltaY$ because we don't know in which order the iterations will be computed. Instead, we can compute $cy$ looking at the current value of $j$, doing this multiplication instead of successive sums will lead to a different approximation error of $cy$ so in total there will be a minimal difference between the total number of iterations of this code and the sequential one. Here the parallelization is really strong because each row will be assigned to a thread, allowing us to work on different rows simultaneously. Another modification that we need to do is to change the way we update $nTotalIterationsCount$, being a shared variable we decided to give to each thread its own partial counter and in the end each thread will update $nTotalIterationsCount$ using the directive `critical`.
Before looking at the improvement given by the parallelization, let's see the image that we produced. These are the fractals produced by the sequential and parallel versions (the sequential version is on the left and the parallel on the right).

Figure 1: 1024x1024, 100 iterations

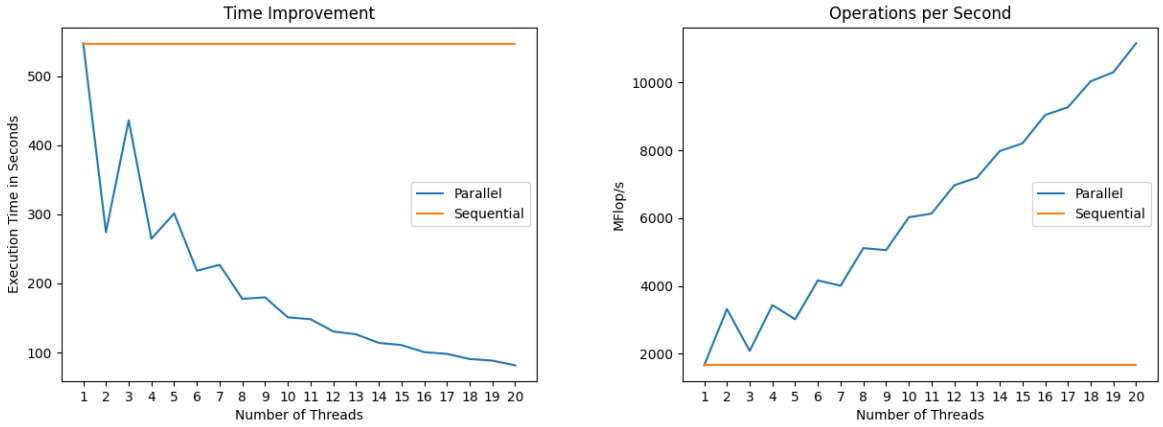


Instead, these are the results for bigger parameters.

Figure 2: 4096x4096, 32507 iterations



Now that we know that the parallel version computes the correct image, let's look at the performance. We computed the execution time and the number of operations per second for the parallel version using a different number of threads. These tests were run using 4096x4096 pixels and 32507 iterations.




We can see that using parallelization we have a much better use of the machine capabilities and the total time is reduced by a factor greater than five.

## 3. Bug hunt [15 points]

Let's look individually at each one of the 5 programs proposed:

### 3.1. Bug 1

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50
#define CHUNKSIZE 5

int main(int argc, char *argv[]) {
```

```
  int i, chunk, tid;
  float a[N], b[N], c[N];

  /* Some initializations */
  for (i = 0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;

#pragma omp parallel for shared(a, b, c, chunk) private(i, tid)                \
    schedule(static, chunk)
  {
    tid = omp_get_thread_num();
    for (i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
      printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
  } /* end of parallel for construct */
}
```

If we try to compile this code, we'll get a compilation error. The reason behind this error is that when we use the compiler directive `pragma omp parallel for` we need to follow it with a for-loop, while in this case we have the assignment of **tid**. The best way to solve this compilation error is to split the directives `parallel` and `for`, putting `for` after the assignment of `tid`.

## 3.2. Bug 2

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

int main(int argc, char *argv[]) {
  int nthreads, i, tid;
  float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d is starting...\n", tid);

#pragma omp barrier

    /* do some work */
    total = 0.0;
#pragma omp for schedule(dynamic, 10)
    for (i = 0; i < 1000000; i++)
      total = total + i * 1.0;

    printf("Thread %d is done! Total= %e\n", tid, total);

  } /*** End of parallel region ***/
}
```

The main problem here is that `tid` in declared outside the parallel region, in this way all the threads are sharing the same copy of `tid`. To correct this bug, we need to add `private(tid)` to the directive `pragma omp parallel`. In the code there wasn't specified what were we actually trying to compute, but it's likely that we also want `total` differently: in particular we can declare

a local variable for each thread that stores the partial sum computed by each thread, and then using the directive `critical` update the toal sum.

### 3.3. Bug 3

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50

int main(int argc, char *argv[]) {
  int i, nthreads, tid, section;
  float a[N], b[N], c[N];
  void print_results(float array[N], int tid, int section);

  /* Some initializations */
  for (i = 0; i < N; i++)
    a[i] = b[i] = i * 1.0;

#pragma omp parallel private(c, i, tid, section)
  {
    tid = omp_get_thread_num();
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }

/*** Use barriers for clean output ***/
#pragma omp barrier
    printf("Thread %d starting...\n", tid);
#pragma omp barrier

#pragma omp sections nowait
    {
#pragma omp section
      {
        section = 1;
        for (i = 0; i < N; i++)
          c[i] = a[i] * b[i];
        print_results(c, tid, section);
      }

#pragma omp section
      {
        section = 2;
        for (i = 0; i < N; i++)
          c[i] = a[i] + b[i];
        print_results(c, tid, section);
      }

    } /* end of sections */

/*** Use barrier for clean output ***/
#pragma omp barrier
    printf("Thread %d exiting...\n", tid);

  } /* end of parallel section */
}

void print_results(float array[N], int tid, int section) {
  int i, j;

  j = 1;
/*** use critical for clean output ***/
#pragma omp critical
```

```
  {
    printf("\nThread %d did section %d. The results are:\n", tid, section);
    for (i = 0; i < N; i++) {
      printf("%e  ", array[i]);
      j++;
      if (j == 6) {
        printf("\n");
        j = 1;
      }
    }
    printf("\n");
  } /*** end of critical ***/

#pragma omp barrier
  printf("Thread %d done and synchronized.\n", tid);
}
```

If we look at this code we can see that in the parallel region there are two sections (the directive `nowait` is used with `sections`, but it's useless because there is a `barrier` at the end of the sections). This means that two of the spawned threads will do those two sections that also call the function `print_results()`. The problem is that in the function `print_results()` there is another barrier, and what happens with barriers is that, a thread that is waiting at a barrier can continue only if also all the other threads are waiting at a barrier (the barrier doesn't have to be the same one for each thread). So when the threads that execute `print_result()` get to the barrier inside that function and all the other reach the barrier after the sections, all the thread continue their execution. But when the threads which executed `print_result()` reach the next barrier, all the other threads aren't at a barrier, and so they remain locked there. In general, if the number of barriers isn't the same for each thread, the threads with more barriers will remain stuck.

## 3.4. Bug 4

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1048

int main(int argc, char *argv[]) {
  int nthreads, tid, i, j;
  double a[N][N];

/* Fork a team of threads with explicit variable scoping */
#pragma omp parallel shared(nthreads) private(i, j, tid, a)
  {

    /* Obtain/print thread info */
    tid = omp_get_thread_num();
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);

    /* Each thread works on its own private copy of the array */
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        a[i][j] = tid + i + j;

    /* For confirmation */
    printf("Thread %d done. Last element= %f\n", tid, a[N - 1][N - 1]);

  } /* All threads join master thread and disband */
}
```

The problem here is that `a` (which size is 8MB) is too big to fit on the stack, so, also before the use of the threads the allocation causes a segmentation faults. Firstly, we need to increase the stack size of the process itself. After that, each thread will receive its copy of the matrix, but like before, that matrix doesn't fit on the stack frame allocated for each thread, so we need to increase the size of each stack frame with the command **export OMP_STACKSIZE=size**.

### 3.5. Bug 5

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main(int argc, char *argv[]) {
  int nthreads, tid, i;
  float a[N], b[N];
  omp_lock_t locka, lockb;

  /* Initialize the locks */
  omp_init_lock(&locka);
  omp_init_lock(&lockb);

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid)
  {

    /* Obtain thread number and number of threads */
    tid = omp_get_thread_num();
#pragma omp master
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);
#pragma omp barrier

#pragma omp sections nowait
    {
#pragma omp section
      {
        printf("Thread %d initializing a[]\n", tid);
        omp_set_lock(&locka);
        for (i = 0; i < N; i++)
          a[i] = i * DELTA;
        omp_set_lock(&lockb);
        printf("Thread %d adding a[] to b[]\n", tid);
        for (i = 0; i < N; i++)
          b[i] += a[i];
        omp_unset_lock(&lockb);
        omp_unset_lock(&locka);
      }

#pragma omp section
      {
        printf("Thread %d initializing b[]\n", tid);
        omp_set_lock(&lockb);
        for (i = 0; i < N; i++)
          b[i] = i * PI;
        omp_set_lock(&locka);
        printf("Thread %d adding b[] to a[]\n", tid);
        for (i = 0; i < N; i++)
          a[i] += b[i];
```

```
        omp_unset_lock(&locka);
        omp_unset_lock(&lockb);
      }
    } /* end of sections */
  }   /* end of parallel region */
}
```

The problem of this code is the management of concurrency. We can see that there are two locks: `locka` and `lockb`. Two of the spawned threads will compute the 2 sections in the parallel region. The problem is that in the first section we have the lock of `locka` and then the lock of `lockb`, instead, in the second section we have the lock of `lockb` and then the lock of `locka`. This means that, if the thread in the first region gets `locka` and the thread in the second region gets `lockb` then there will be a deadlock situation. In order to fix this situation, we should know what is the expected behavior of the program, here isn't clear if $a$ should be processed before or after $b$.

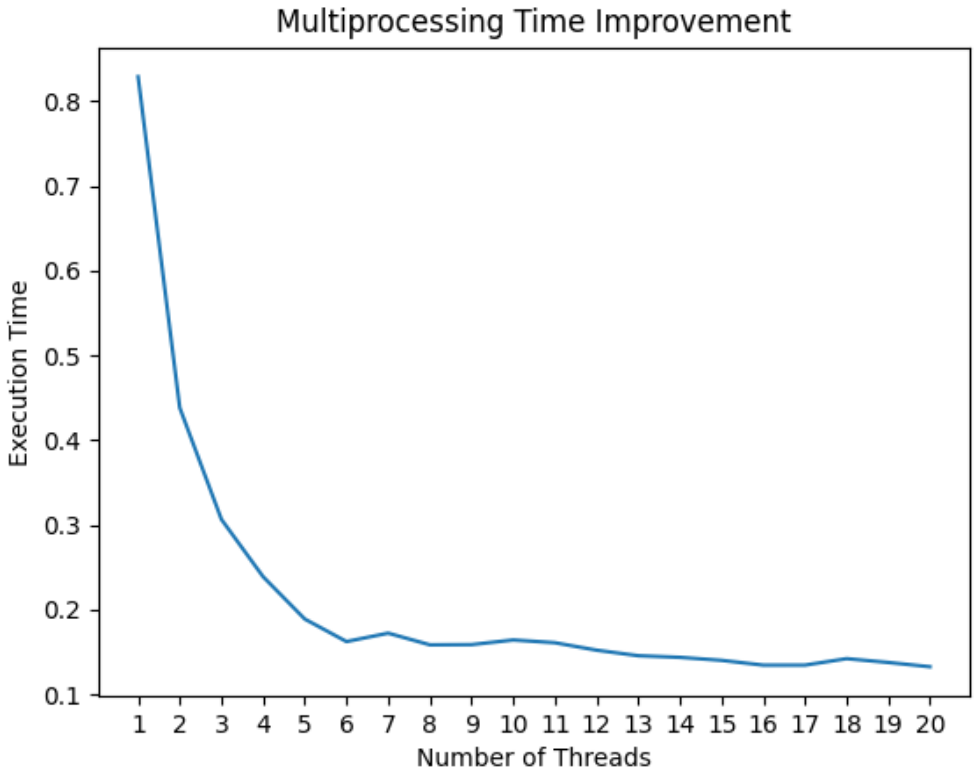
## 4. Parallelhistogram calculation using OpenMP [15 points]

The given program visualize the generation of numbers using a normal distribution. In order to do that, $N$ values are generated with a normal distribution and then are collocated into $k$ bins ($k$ is equal to 16, so it's pretty small) that split the range of the generate numbers in $k$ continuous segments. Being the elements generated with a normal distribution, we expect that the bins in the middle will be the most used, while the first and the last bins will be the least used. This process is divided into 2 parts, (1) before $N$ values are generated and then (2) are collocated in the bins. We want to parallelize the second part, this means that given a vector $vec$ of size $N$ containing the generated elements we want $t$ threads to count the number of elements that goes in each bin. In order to do that, using the directive `omp for`, we divide the iteration over $vec$ into different parts for the threads. Now each thread will look at some positions on $vec$ and should place those elements into the bins. The best way to do that is to give at each thread its set of $k$ bins to update, because if we wanted to use only a set of bins for all the threads we would need to manage the concurrent access to it and the performances would be similar to the sequential version of the algorithm. When all the threads will be done counting the elements that go in each bin, then we can update the global bins sequentially (this part doesn't use too much time, because the number of threads and the number of bins is small).

The final code looks like this

```
#pragma omp parallel shared(dist)
        {
                long my_dist[BINS];
                for (int i = 0; i < BINS; i++) {
                        my_dist[i] = 0;
                }
#pragma omp for
                for (long i = 0; i < VEC_SIZE; ++i) {
                        my_dist[vec[i]]++;
                }
#pragma omp critical
                for (int i = 0; i < BINS; i++) {
                        dist[i] += my_dist[i];
                }
        }
```

This code produce the correct output (the bins respect the normal distribution) and this is the improvement in terms of performance.




We can see that the sequential version uses 0.8 seconds, while using more threads allows us to perform the task in less than 0.2 seconds.

## 5. Parallel loop dependencies with OpenMP [15 points]

In this section, we need to optimize a for loop that carries loop dependencies. The loop looks like this:

```
int N = 2000000000;
double up = 1.00000001;
double Sn = 1.00000001;
int n;
double *opt = (double *)malloc((N + 1) * sizeof(double));
for (n = 0; n <= N; ++n) {
  opt[n] = Sn;
  Sn *= up;
}
```

It's not possible to just use the directive `omp parallel for` because each iteration would be computed by some thread not in a linear order, but in order to compute the correct value of $opt[i]$ we need to guarantee the correct order of the iterations. Another problem would be the update of $S_n$ which should be done into a `critical` scope, slowing down the performance.

Looking at the code we see that in general $opt[i] = S_n * (up)^i$, so we implemented a parallel version in this way: we divide the array $opt[]$ into $t$ logical chunks of the same size, where $t$ is the number of threads. In this way, each thread will compute a continuous part of $opt[]$ using its own for-loop. Let's suppose that thread $k$ has to compute $opt[l:r]$, it can compute $opt[l]$ as $S_n * (up)^l$ and then iterate from $l+1$ to $r$ computing $opt[i]$ as $opt[i-1] * up$. Each thread should also update the value of $S_n$ (multiplying it by $up$ at each iteration), but like we said before, to update a variable that is shared between the threads it's slow. So, instead of updating $S_n$ at each iteration, each thread will compute its own multiplicative factor based on the size of its chunks, and then update $S_n$ only one time in a serial way (using `critical`).

```
#pragma omp parallel
{
        int i, id, nths, istart, iend;
        // Id of the current thread
        id = omp_get_thread_num();
        // Number of threads
        nths = omp_get_num_threads();
        // Start and end of the interval that thread id needs to compute
        istart = id * N / nths;
        iend = (id + 1) * N / nths;
        // Used to update Sn
        double factor = 1.0;
        if (id == nths - 1) {
                iend = N;
        }
        // Computing opt[l]
        opt[istart] = Sn * fastexp(up, istart);
        factor *= up;
        for (i = istart + 1; i < iend; i++) {
                opt[i] = opt[i - 1] * up;
                factor *= up;
        }
// Updating Sn
#pragma omp critical
        Sn *= factor;
}
```

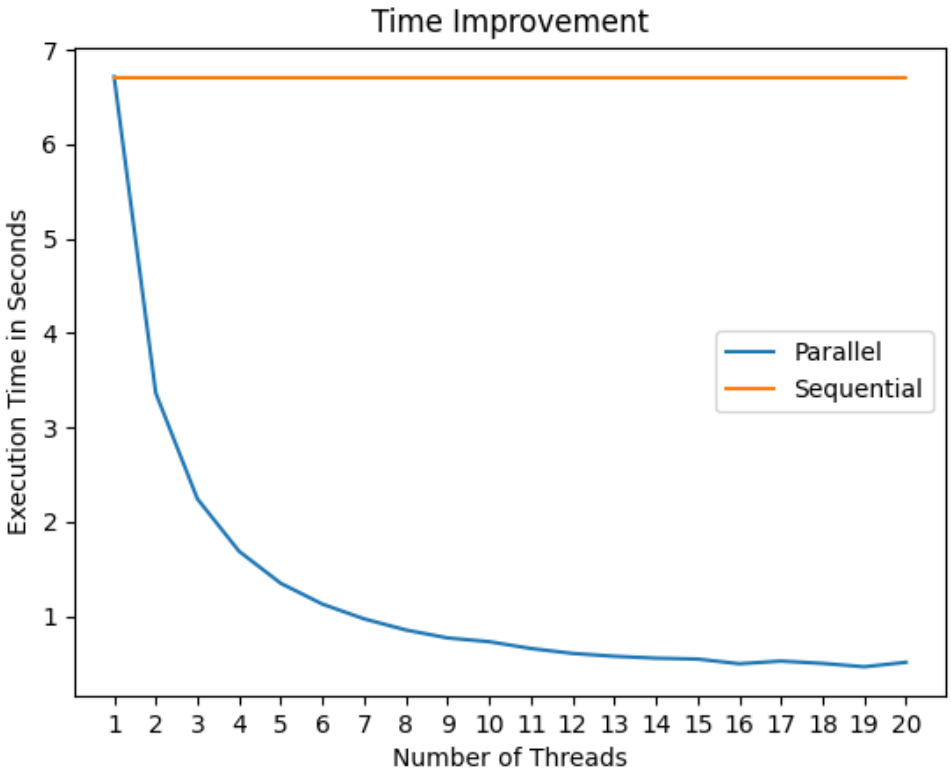
Where the function $fastexp(a, b)$ computes $a^b$ in $O(log(b))$ , and it looks like this

```
double fastexp(double base, int esp) {
        double res = 1.0;
        while (esp) {
                if (esp % 2) {
                        res *= base;
                }
                esp /= 2;
                base *= base;
        }
        return res;
}
```

Looking at results we see that the sequential version, with $N = 2000000000$, takes 6.716s to run while the parallelized version with 20 threads takes 0.4542s to compute the same result. The speed-up is really impressive, but at the same time, using the exponential function instead of the consecutive multiplications we have a slightly higher approximation error, for example looking at the difference between the serial version and the parallel version with 20 threads:

| Version | Time | Final $S_n$ | $\|opt\|^2$ |
|---|---|---|---|
| Sequential | 6.716s | 485165097.62511122 | 5884629305179574.00 |
| Parallel | 0.4542s | 485165092.76982492 | 5884628212411730.00 |

In terms of time scaling factor using the parallel version, we have a factor of 14.78, trying different numbers of threads we obtain this plot.


Time Improvement
Execution Time in Seconds
Number of Threads
Parallel
Sequential
1
2
3
4
5
6
7
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20