Università della Svizzera italiana

Institute of Computational Science ICS

High Performance Computing **2019**

Student: Gabriel Fernandes de Oliveira Discussed with: N/A

Solution for Assignment 3

Due date: 30 October 2019, 11:59pm

Parallel Programming with OpenMP

This assignment begins with the analysis of parallel programs, and will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP *(30 Points)*

1. My implementation of the dot product using OpenMP reduction clause follows:

```
//   i. Using reduction pragma
time_red = walltime(0);
for(int iterations=0;iterations<NUM_ITERATIONS;iterations++) {
  alpha_parallel=0.0;
  #pragma omp parallel for reduction(+ : alpha_parallel)
  for(int i=0; i< NMAX; i ++)
      alpha_parallel += a[i] * b[i];
}
time_red = walltime(time_red);
```

   The reduction was created to sum the various multiplications to the variable *alpha_parallel*.

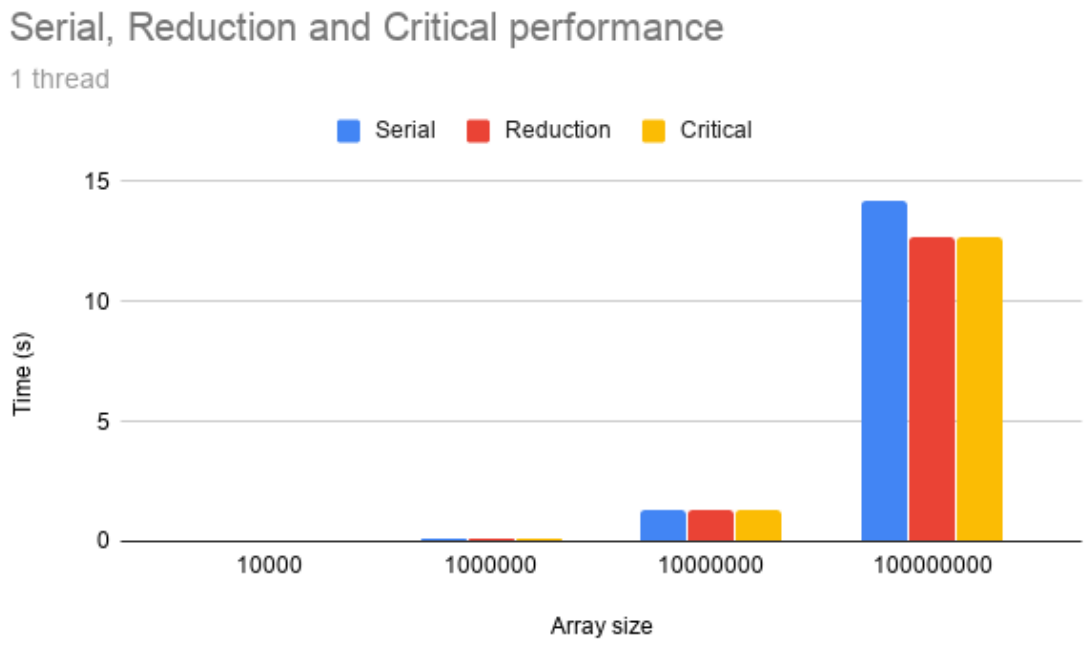
2. My implementation of the dot product using OpenMP parallel and critical clause follows:

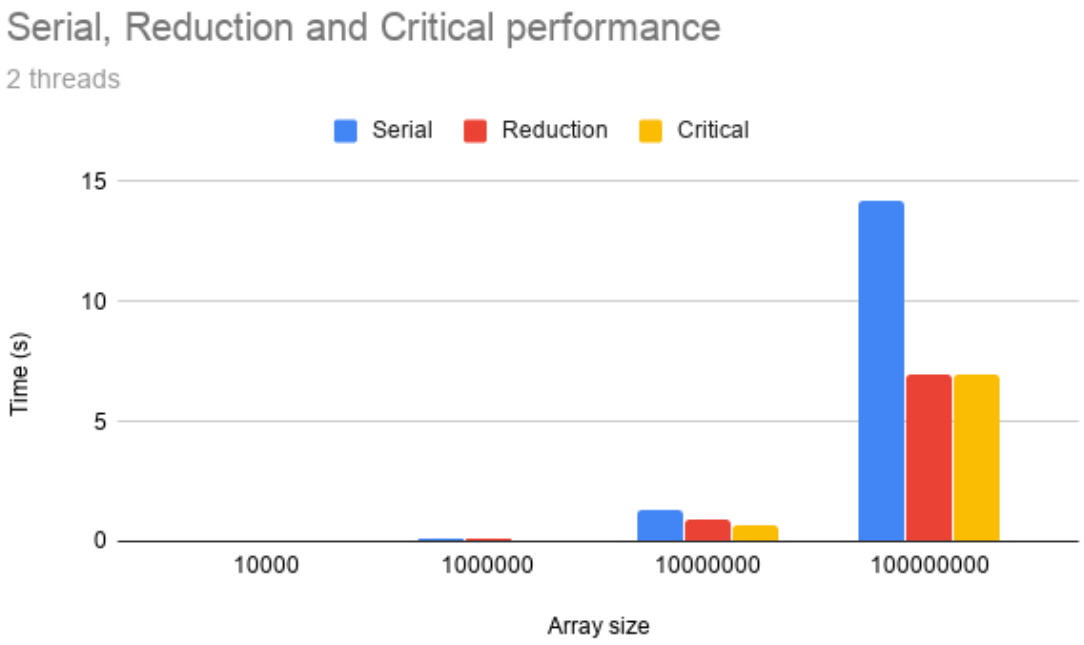
```
//   ii. Using critical pragma
time_critical = walltime(0);
for(int iterations=0;iterations<NUM_ITERATIONS;iterations++) {
  alpha_parallel=0.0;
  long double partial_alpha = 0.0;
  #pragma omp parallel firstprivate(partial_alpha) shared(alpha_parallel, a, b)
  {
      #pragma omp for
      for(int i=0; i< NMAX; i ++)
          partial_alpha += a[i] * b[i];
      #pragma omp critical
      { alpha_parallel += partial_alpha; }
  }
}
time_critical = walltime(time_critical);
```

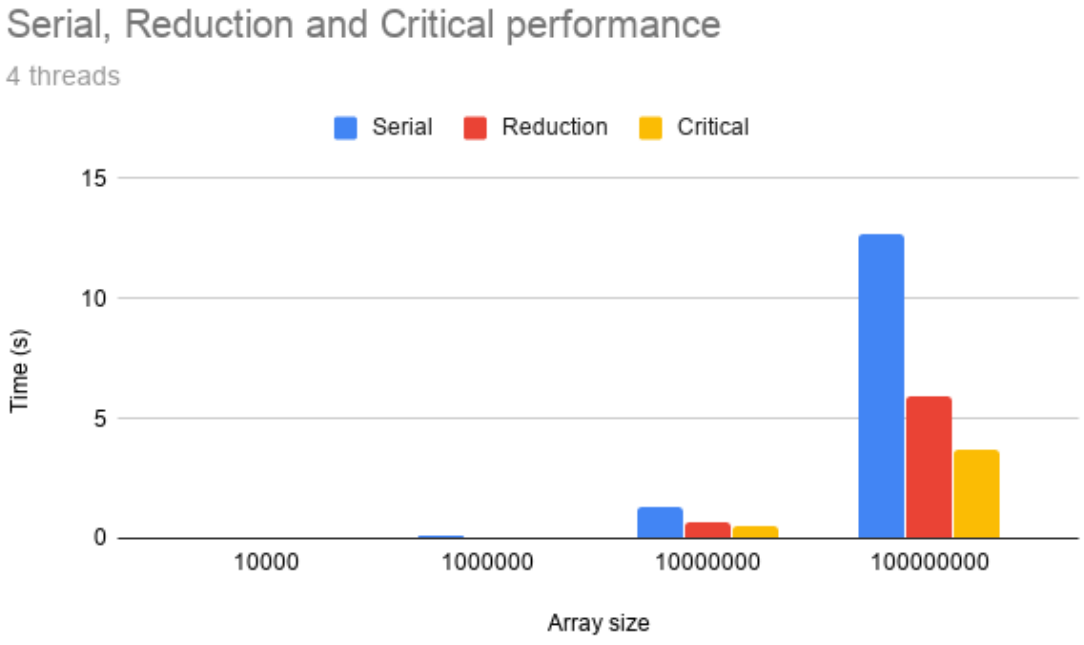
To optimize the code, I created an auxiliary variable *partial_alpha* that is private for every thread, this helps to reduce the amount of critical sections each thread has to execute.

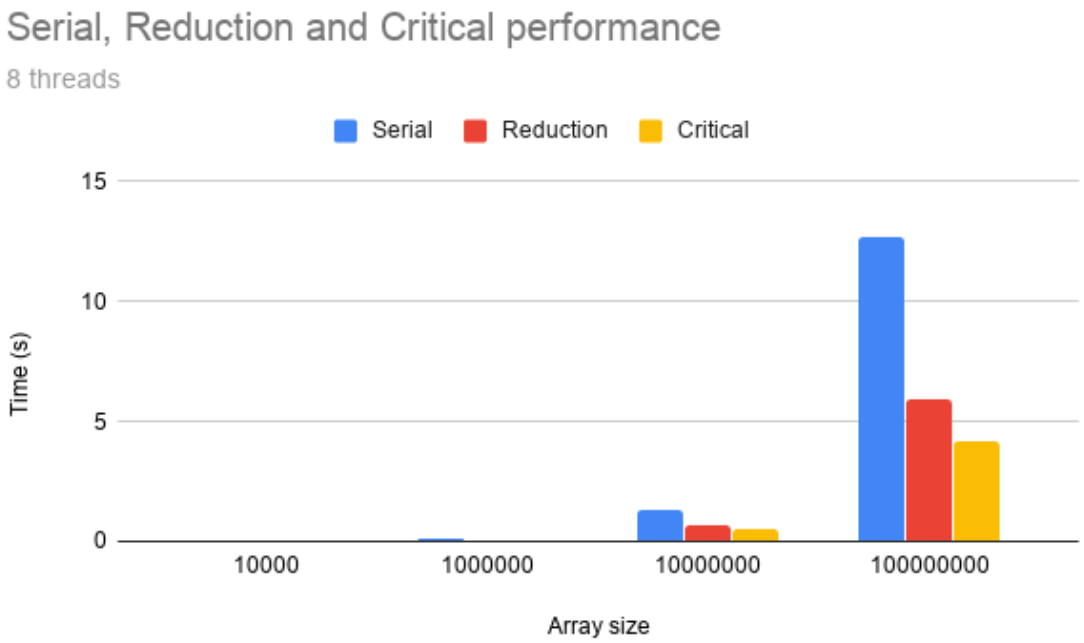
The only critical section created sums the value of *partial_alpha* to the shared variable *alpha_parallel*.

3. The first analysis made was plotting the time the dot product takes versus the array sizes:




Serial, Reduction and Critical performance
2 threads
Serial
Reduction
Critical
15
10
5
0
Time (s)
10000
1000000
10000000
100000000
Array size

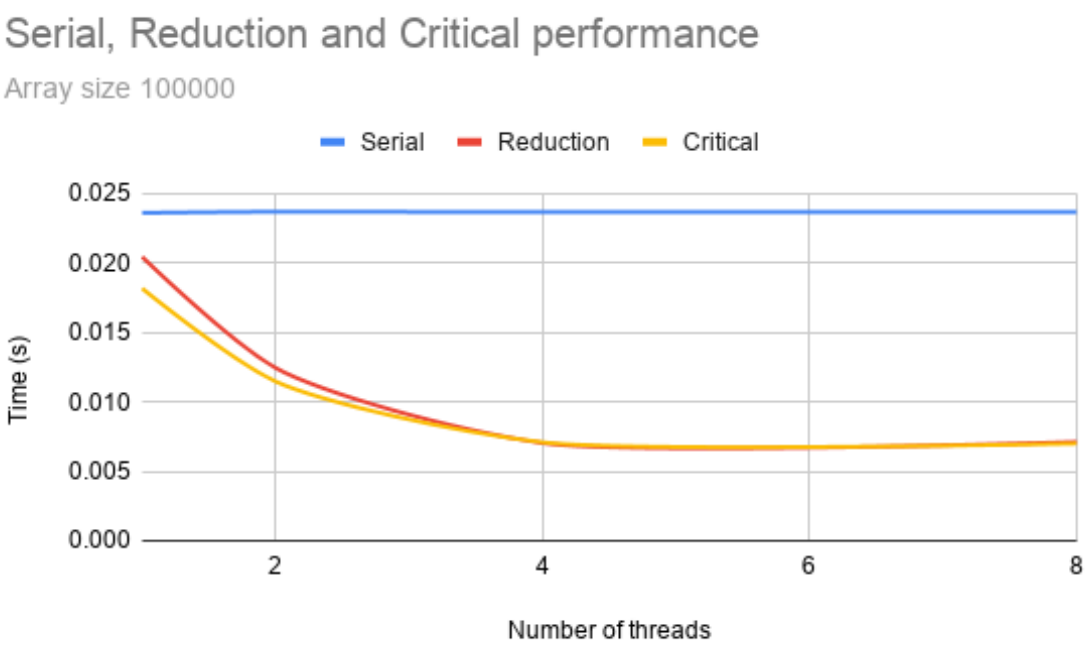

Serial, Reduction and Critical performance
4 threads
Serial
Reduction
Critical
15
10
5
0
Time (s)
10000
1000000
10000000
100000000
Array size

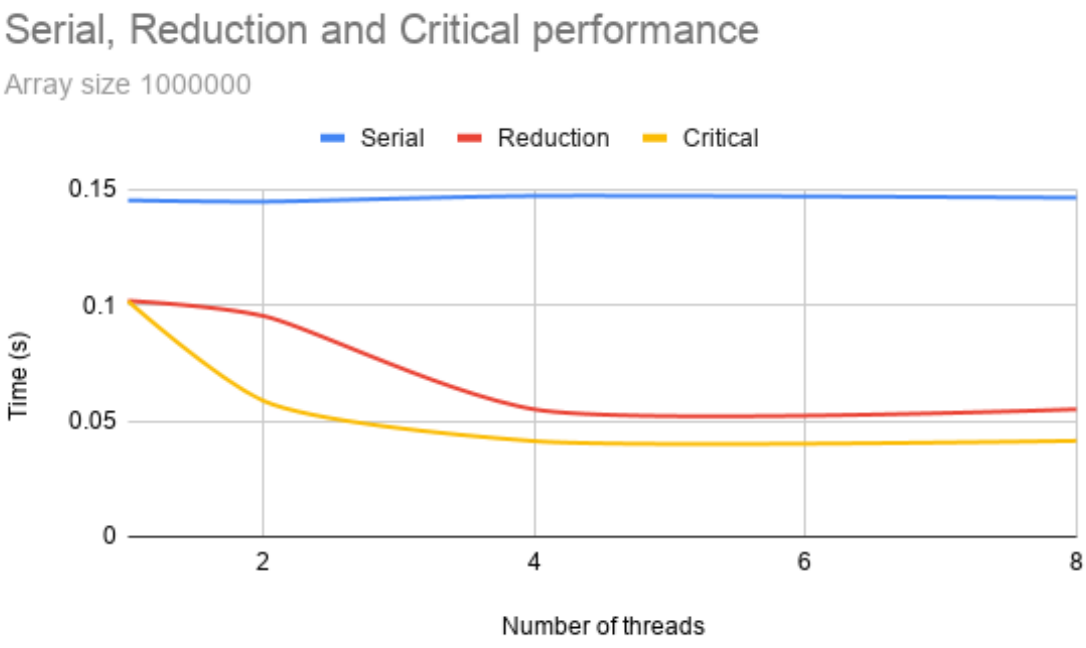


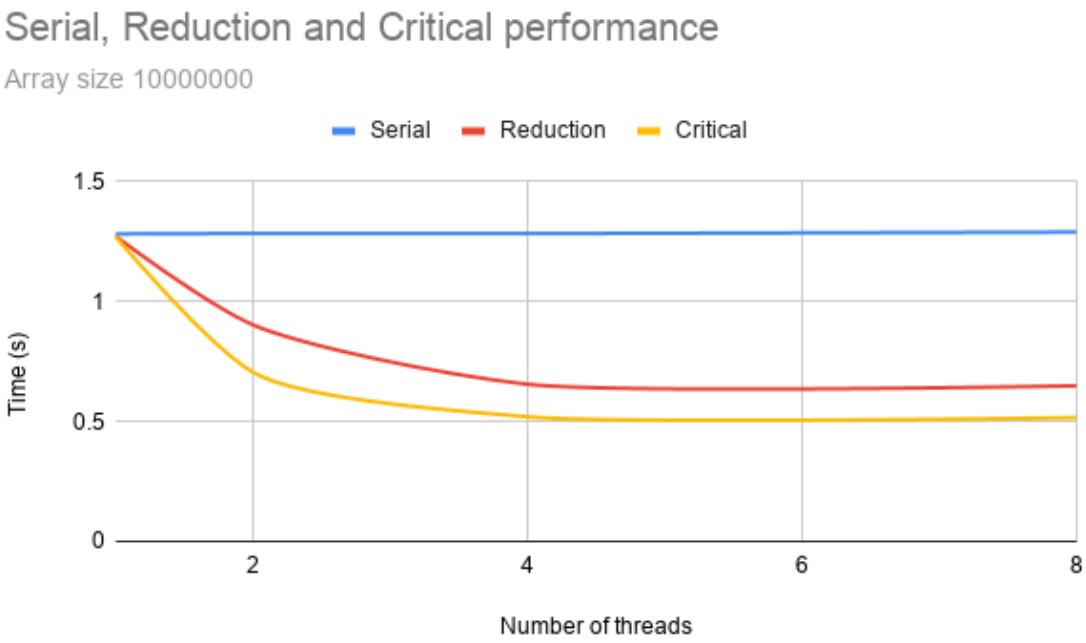
With these graphs it is easy to see the increase in performance the parallel algorithms offer when compared to the serial algorithm.

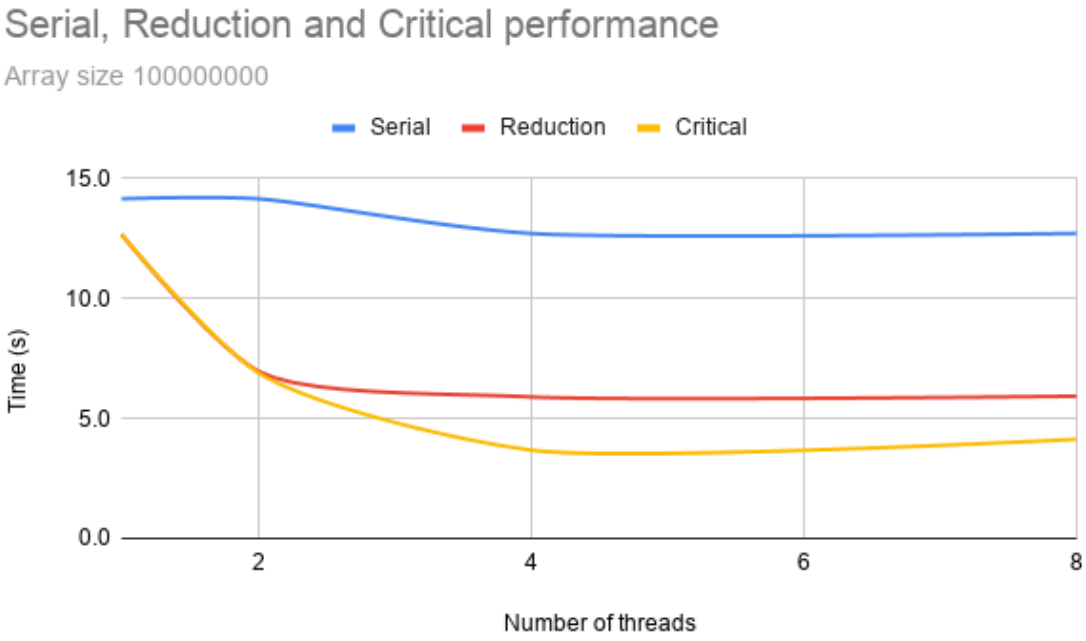
It is also noticeable that the parallel algorithms have a peak in performance at about 4 threads, one can see that there is not much performance gain when using 8 threads.

To better analyse the performance of the algorithms based on the number of threads I also plotted the following graphs:


Serial, Reduction and Critical performance
Array size 100000
Serial
Reduction
Critical
0.025
0.020
0.015
0.010
0.005
0.000
Time (s)
2
4
6
8
Number of threads


Serial, Reduction and Critical performance
Array size 1000000
Serial
Reduction
Critical
0.15
0.1
0.05
0
Time (s)
2
4
6
8
Number of threads





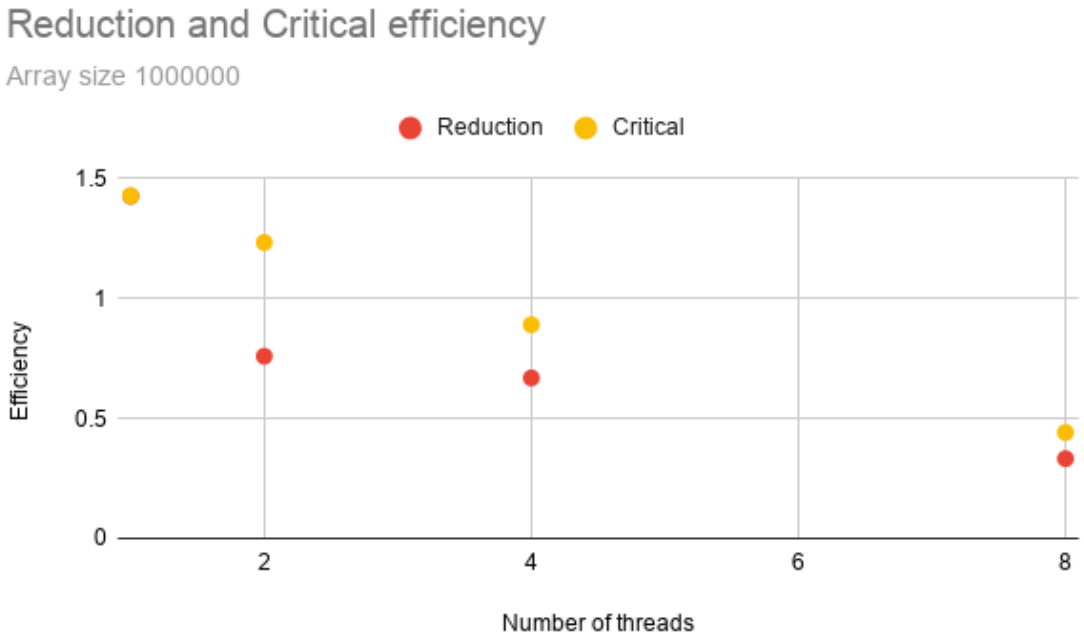
With these graphs it is easier to notice the performance of the parallel algorithms based on the number of threads available.

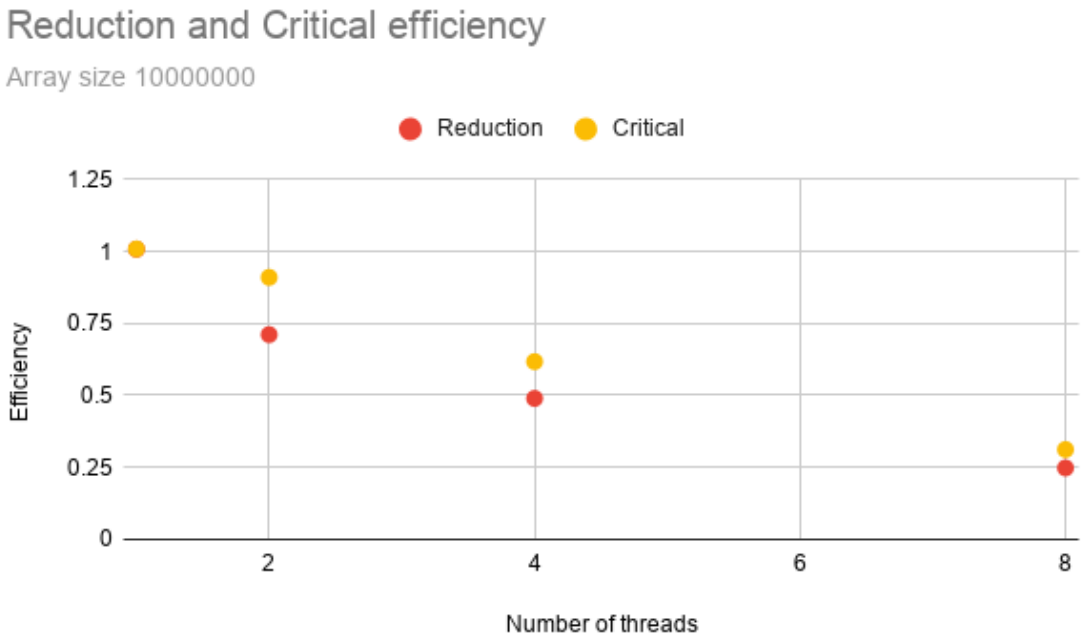
4. The following are the graphs of the parallel efficiency for the experiments made.

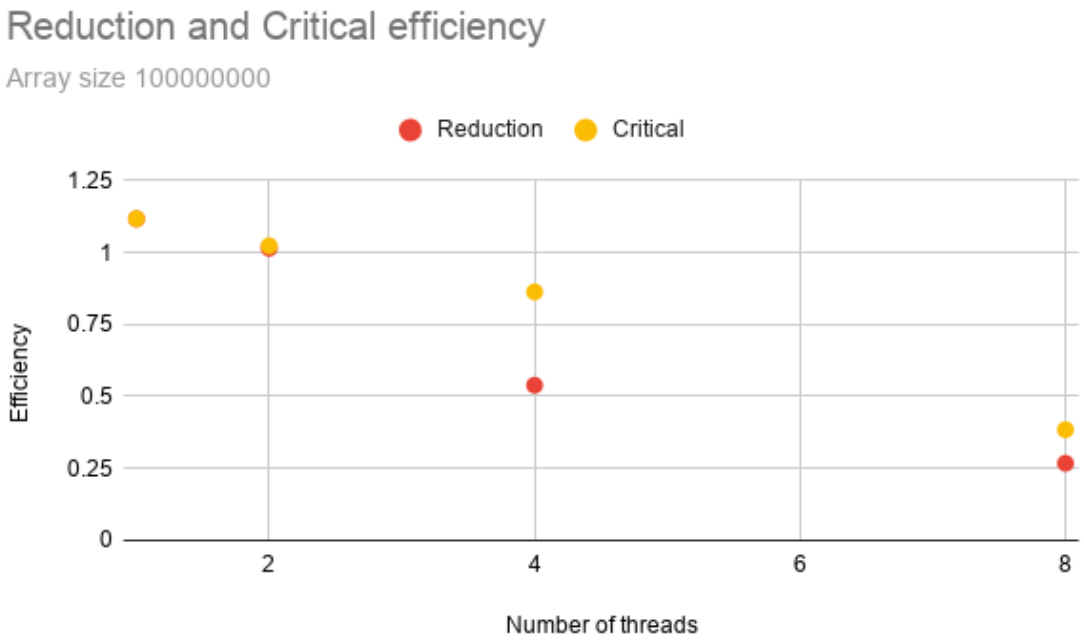
The parallel efficiency E was calculated by the formula:

$$E = \frac{t_{seq}}{t_{par} \times num\_threads}$$

Where t_{seq}, t_{par} are the execution times of the serial and parallel algorithms, repectivelly, and $num\_threads$ is the number of the threads available for the experiment







Although there is a small variance between the results depending of the array size, the trend in all graphs is the same: the more threads used the worst the efficiency.

In the previous question we stated that there was no big performance gain from using 4 threads to using 8 threads, the performance in both cases is almost the same. This ends up creating a

big downgrade in the efficiency value for 8 threads. Hence, in all graphs the efficiency for using 8 threads is between 0.25 and 0.5.

2. Visualizing the Mandelbrot Set *(30 Points)*

The sequential implementation for the computation of the orbit follows:

```
// compute the orbit z, f(z), f²(z), f³(z), ...
// count the iterations until the orbit leaves the circle |z|=2.
// stop if the number of iterations exceeds the bound MAX_ITERS.

double zx = 0, zy = 0;
n = 0;

double ox = zx, oy = zy;
double oxx = zx*zx, oyy = zy*zy, oxy = zx*zy;
while(oxx + oyy < 2. && n < MAX_ITERS){
    zx = oxx - oyy + cx;
    zy = oxy*2 + cy;
    n++;

    oxx = zx*zx;
    oyy = zy*zy;
    oxy = zx*zy;
}
nTotalIterationsCount += n;
// n indicates if the point belongs to the mandelbrot set
// plot the number of iterations at point (i, j)
int c = ((long) n * 255) / MAX_ITERS;
png_plot (pPng, i, j, c, c, c);
```

The counting of the total number of iterations is mantained by the variable $nTotalIterationsCount$ which is always incremented by the amount of iterations made in each orbit calculation (n).

The following is the number of iterations required for each image size (using 35207 as the maximum number of iterations on an orbit computation):

Total number of iterations				
256	512	1024	2048	4096
444108151	1774160381	7100090703	28390590023	113559844217

3. Parallel Mandelbrot *(20 Points)*

The parallelization process consisted in three main changes:

1. The introduction of the pragma statement.

```
#pragma omp parallel for schedule(dynamic) private(i, j, x, y, cx, cy, x2, y2, n) shared(nTotalIterationsCount)
for (j = 0; j < IMAGE_HEIGHT; j++)
{
    cx = MIN_X;
    cy = MIN_Y + fDeltaY*j;
```

Figure 1. Lines 36-40 of the file *mandel_par.c*

   I chose to use the dynamic scheduling in order to better divide the work between threads.

   Since it is quite unpredictable if the computation of the orbits will execute MAX_ITERS iterations or if the loop will be broken in few iterations because the orbit leaves the circle $|z| = 2$, I've decided not to use a static scheduling for the threads.

2. Changing the way *cy* was computed.

   In the sequential code *cy* was initialized with the value MIN_Y and incremented by $fDeltaY$ after every execution of the loop for the image height (first loop in the mandebrot calculation section).

   In order for this to work in a parallel environment I had to change the way this variable was modified: instead of constantly incrementing the variable I added a line, at the beginning of the loop initializing *cy* with the value $MIN\_Y + fDeltaY \times j$, as can be seen in figure 1.

   Initializing the variable this way removes the need of making *cy* a shared variable or including a more complicated way to keep track of its value between threads.

3. Adding a critical section to keep track of the total amount of iterations.

```
#pragma omp critical
{nTotalIterationsCount += partial_n;}
```

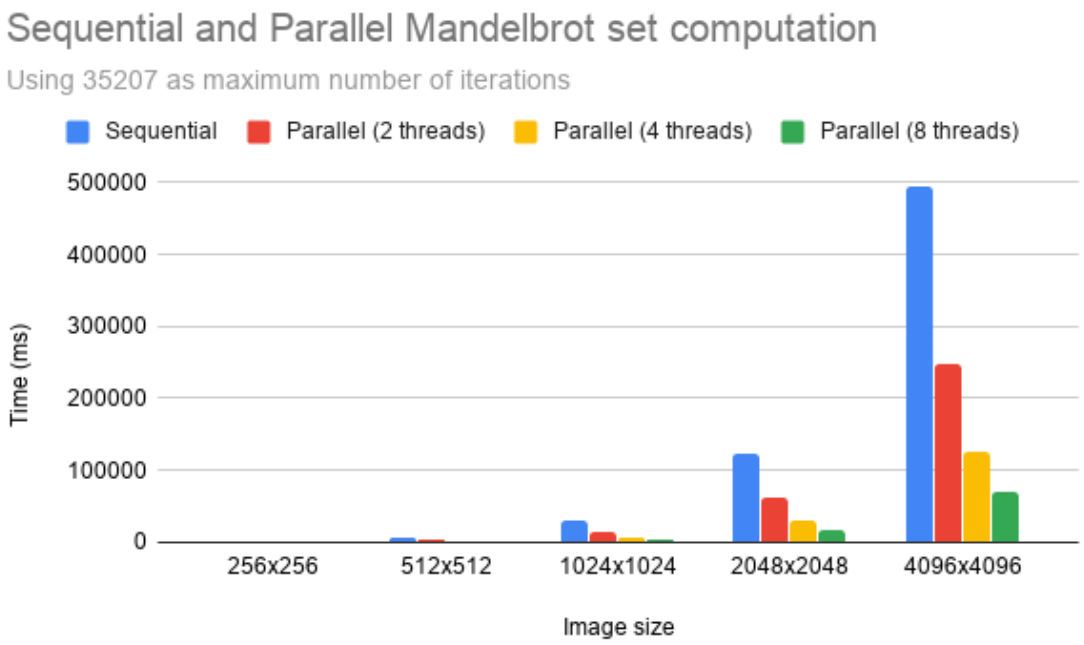
Figure 2. Lines 74-75 of the file *mandel_par.c*

   Since $nTotalIterationsCount$ is a shared variable, its value must be updated by every thread in a critical section.

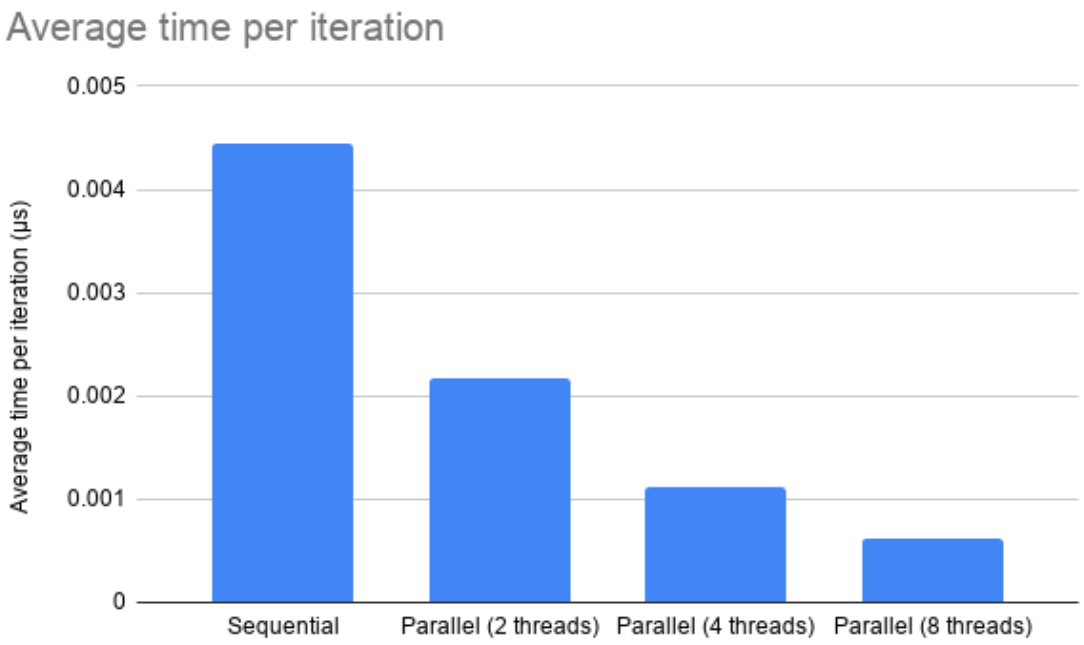
   A private variable $partial\_n$ was also created to reduce the amount of critical sections updating the value of $nTotalIterationsCount$.

I've ran the sequential algorithm and the parallel algorithm with different number of threads and computed the results in the following graphs.

One can see that the program with 8 threads had a better performance in this problem when comparing to the dot product problem.



The following graph shows very well the performance increase based on the number of threads used.



Doubling the number of threads ended up dividing by half the average time for iteration.

4. Bug Hunt *(20 Points)*

omp_bug1 This program presents a compile time error. That happens because the directive *#pragma omp parallel for* should be directly followed by a for loop, and this is not what happens in this program.

In the program, the line following the pragma directive is *tid = omp_get_thread_num()*.

That explains the compile error the program faces.

A way to correct this program is the following:

```
#pragma omp parallel     \
shared(a,b,c,chunk)            \
private(i,tid)
{
tid = omp_get_thread_num();

#pragma omp for \
schedule(static,chunk)
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 */
```

Breaking the pragma in two parts. The first part is just the parallel premisse, without the for and the scheduling part. The second part is the for and the desired scheduling strategy, placed right before the for.

omp_bug2 There are two problems in this implementation.

The first problem is that the messages in the end of the program all claim to be the same thread, this is due to the fact that the variable *tid* is shared through all threads, so the only value displayed at the end is the *id* value of the last thread to execute the line *tid = omp_get_thread_num()*.

The second problem is in the calculation of the *total* variable, since this variable is shared through all threads its changes should be inside a critical section. Instead of using a critical

section, one could also use reduction for this for, as shown in the solution that follows:

```
/*** Spawn parallel region ***/
#pragma omp parallel private(tid) shared(total)
{
    /* 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 */
    #pragma omp for schedule(dynamic,10) reduction(+ : total)
    for (i=0; i<1000000; i++)
        total = total + i*1.0;
```

The only necessary changes to solve this remaining problem are making *tid* a private variable and adding the reduction premise before the loop, so that the shared variable *total* will be correctly calculated.

omp_bug3 The main problem here is that the program doesn't end. There will always be two threads trapped in the last barrier of the main function, these being the threads that executed the function.

Follows an illustration of what happens:

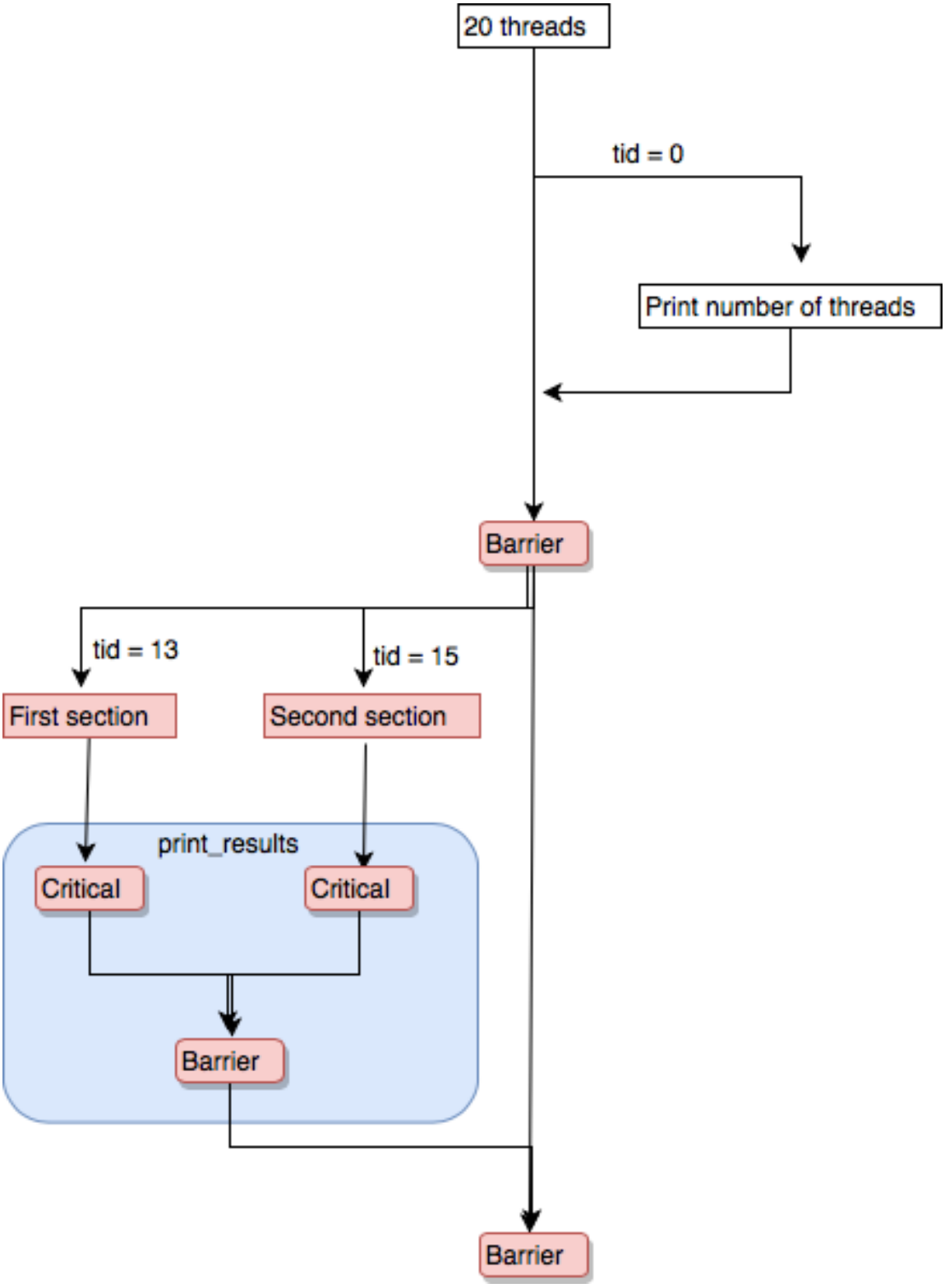


Figure 3. An example of the flow threads may take when executing *omp_ bug3.c*

One can see that two threads, of *tid* 13 and 15 in the example shown in 3, pass through three

barriers in their flow, whilst all other threads pass through only two barriers.

This ends up making the threads 13 and 15 stuck on the last barrier, by themselves. The program never finishes because the remaining 18 threads the last barrier was waiting for have already been killed since they finished the execution of the main function.

One possible way to fix this is by removing the extra barrier found in the *print_results* file. This way, all threads synchronize at the end of the function main and finish their execution, allowing the program to end.

omp_bug4 Using *ulimit -a* one can see the stack size that icsmaster allows for usage. This value is $8196kB \approx 8.2MB$.

In this program the runtime error is caused by a stack memory overflow. The matrix a has 1048^2 doubles, that translates to roughly $8.8MB$. Moreover, since a is a private variable, the whole matrix is being copied for every thread created. That means that if the program executes with 20 threads, for instance, the memory being used is $20 \times 8.8MB = 175.7MB$ which is much more than the stack can store.

There are some solutions to this problem:

- Reducing the dimensions of the matrix to less than $\sqrt{(8.8 \times 10^6)/(num\_threads \times 8)}$. This way, each matrix will have $(8.8 \times 10^6)/num\_threads$ doubles. Where $num\_threads$ is the number of threads that will have the matrix as a private variable.
- The second solution is making a a shared variable, so that it doesn't get copied for every thread. This solves the memory problem, but greatly damages performance, since the updates in the matrix would have to be wrapped by a critical section.
- The final solution, and most clever, would be to simply change the program to remove the matrix a, and simply make every thread print the message "Thread $thread - number$ done. Last element=$tid + 2N - 2$.

  This would make the output of the program be the same, improve performance of the routine and also remove the memory problem from the program. But this solution only works because the only element printed is the last one in the matrix, and its calculation is well defined by the program.

omp_bug5 The deadlock caused is very similar to the metaphor for the Dining Philophers. There are two resources a, b and two sections that need to use both resources.

Follows a description of the actions leading to the deadlock, for two threads, named 1 and 2:

Thread 1 executes the first section and thread 2 executes the second section.

Thread 1 locks the usage of array a for himself.

Thread 2 locks the usage of array b for himself.

Thread 2 tries to lock array a for himself but cannot do it, since it still belongs to thread 1.

Thread 1 tries to lock array b for himself but cannot do it, since it still belongs to thread 2.

Hence no thread may advance in their computation, both threads keep infinitelly waiting for each other to release their locks.

One solution for this problem is changing the order the locks are made. If both threads reserve the usage of a and b, in this order, at the beginning of their sections there won't be ny more deadlocks, as implemented in the file *omp_bug5*.