Università della Svizzera italiana | Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student:Kaan Egemen Sen Discussed with: Lorenzo Varese

Solution for Project 2

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

Programming can be structured in two ways: sequentially or in parallel. Parallel programs are more difficult to structure but archive better performances in time which is one of the core points in HPC. OpenMP is a set of compiler directives that allows to creation of multi-threaded blocks called parallel regions. Between two parallel regions only the master thread is executed whereas when the compiler encounters a parallel region it does a fork to spawn the threads which ends at the end of a block with an automatic join operation. In the C language in order to use these compiler instructions it is important to import the omp.h package, then at the beginning of each region we need to type:

pragma omp parallel

In order to parallelize a portion of code it is also important not to forget about race condition situations. In fact, the openMP directive only spawns threads but they do not handle these unexpected situations. Taking as an example the following piece of code:

```
    // serial execution
  // Note that we do extra iterations to reduce relative timing overhead

  time_start = wall_time();
  for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
    alpha = 0.0;
    for (int i = 0; i < N; i++) {
      alpha += a[i] * b[i];
    }
  }
  time_serial = wall_time() - time_start;
  printf("### Output for N = %d\n", N);
  cout << "Serial execution time = " << time_serial << " sec" << endl;
```

it is possible to parallelize the for in order to divide the iteration between the threads. This can be done with the OpenMP constructs. Two of them are the reduction and the critical sections:

- **Reduction pragma**: In this pragma, we are telling the compiler to create parallel regions that support reductions. The reduction syntax requires putting into round brackets the name of that variable, which should be kept private during the iteration and whose value should be merged at the end of the construct. In this case, alpha parallel should be merged to one value after all the threads finish their execution and in particular the partial values calculated by the threads should be summed. This is why the semantics of our case require a plus in round brackets:

(+alpha_parallel)

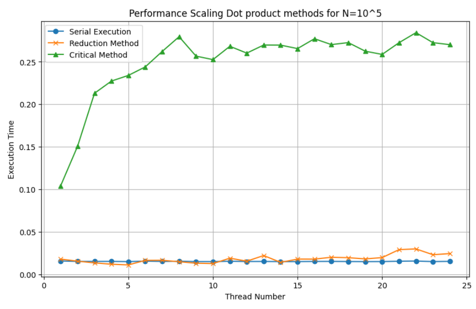
This construct is used in case of multiple threads which should do some simple computation on the same variable in a for loop.

- **Critical pragma**: This construct is instead used when a section should be executed only by a thread at a time. In this other case, the shared variable is substituted by another private variable(it is private because it is defined inside the code block) and used inside the loops rather than the shared one. In this way, the race condition is avoided and its value is merged in a critical section in order not to have a race condition situation. Using these two sections it is possible to obtain the following results:

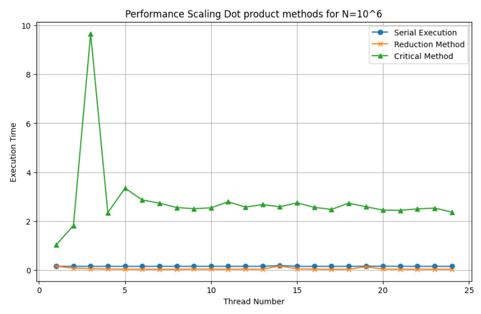
```
 //   i.  Using reduction pragma
time_start = wall_time();
for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++)
{
  alpha_parallel = 0.0;
  #pragma omp parallel for default(none) reduction(+ : alpha_parallel) firstprivate(a, b,N)
  for (int i = 0; i < N; i++)
  {
    alpha_parallel += a[i] * b[i];
  }
}
time_red = wall_time() - time_start;

//   ii. Using  critical pragma
time_start = wall_time();
for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++)
{
  alpha_parallel = 0.0;
  #pragma omp parallel for default(none) shared(alpha_parallel) firstprivate(a, b,N)
  for (int i = 0; i < N; i++)
  {
    #pragma omp critical
    alpha_parallel += a[i] * b[i];
  }
}
time_critical = wall_time() - time_start;
```

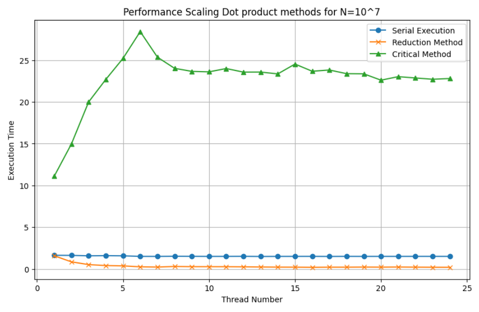
The graphs obtained are the subsequent:



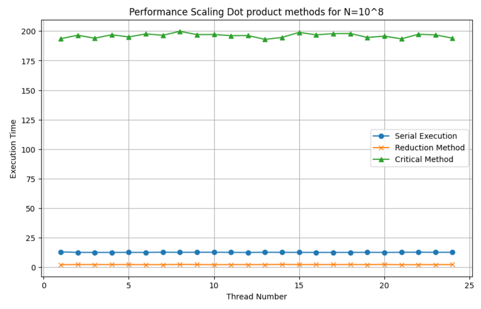
(a)



(b)



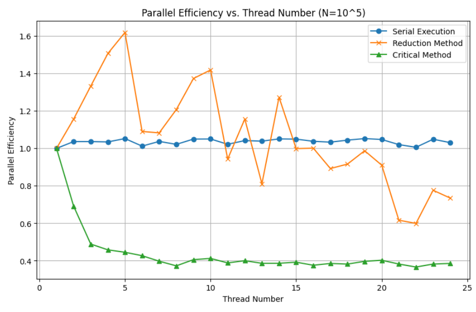
(c)



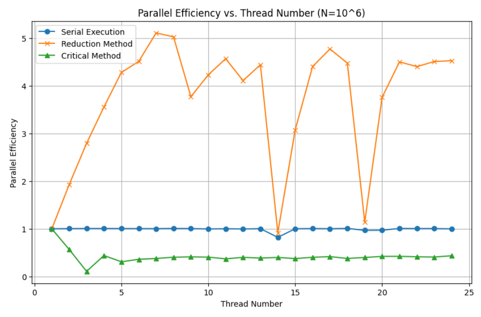
(d)

Figure 1: Resulting Plots Dot Product With Different Methods

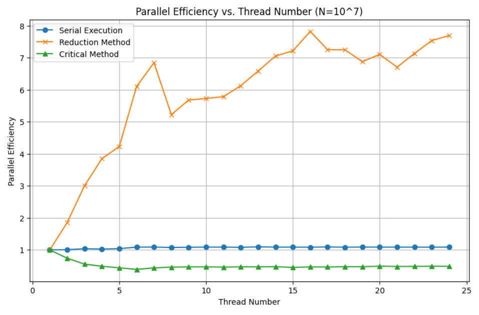
We can clearly see that the Reduction method is way better than the other two methods and we also can see that the worse results we get by the Critical Method. The reduction method always stays near 0 and the gap between Serial execution gets bigger when we try a bigger Vector.



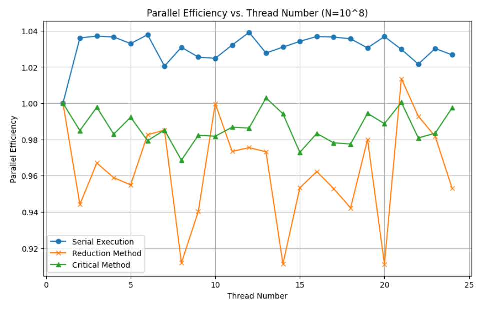
(a)



(b)



(c)



(d)

Figure 2: Efficiency Plots Of The Different Dot Product Methods

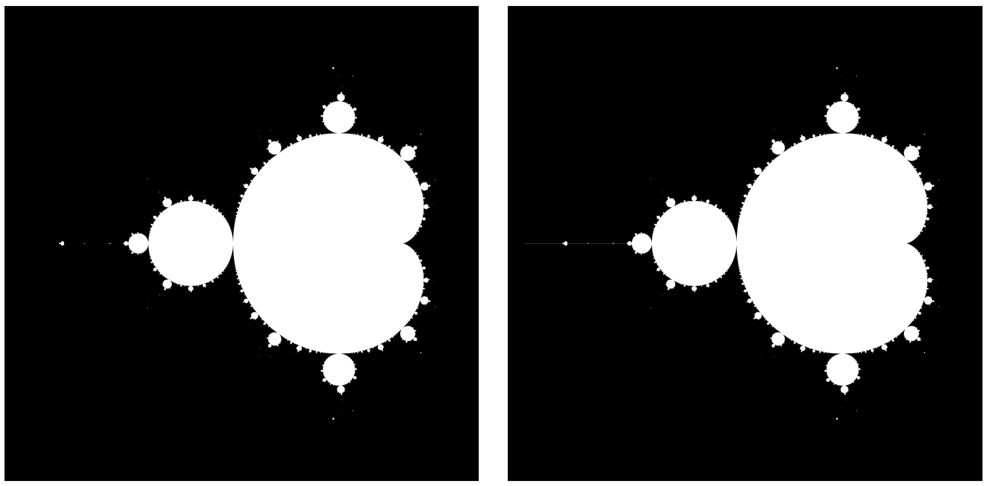
Then we have our efficiency comparisons above and can clearly see which thread number reduces the efficiency of the method.

2. The Mandelbrot set using OpenMP [30 points]

To implement the Mandelbrot set we need to take different values of a complex plane and test if they belong to the Mandelbrot set or not. z is a complex number to which we want to add c after squaring it in this formula:

$$z = z^2 + c$$

As we would normally expect, squaring normal numbers would take an exponential growth of the results, whereas, with some numbers(strictly minor than two) that belong to that set, this does not happen. Indeed, the result of this function will converge to a maximum of 2. In the subsequent implementation, we plot the Mandelbrot set numbers in an image so that we can visualize in white the points that represent the values that belong to the Mandelbrot set and the others in black.



(a) (b)

Figure 3: Resulting images of the Mandelbrot methods for **(a)** sequential **(b)** parallel

The difference between the serial and the parallelized code in terms of performance can be summarized in the subsequent table:

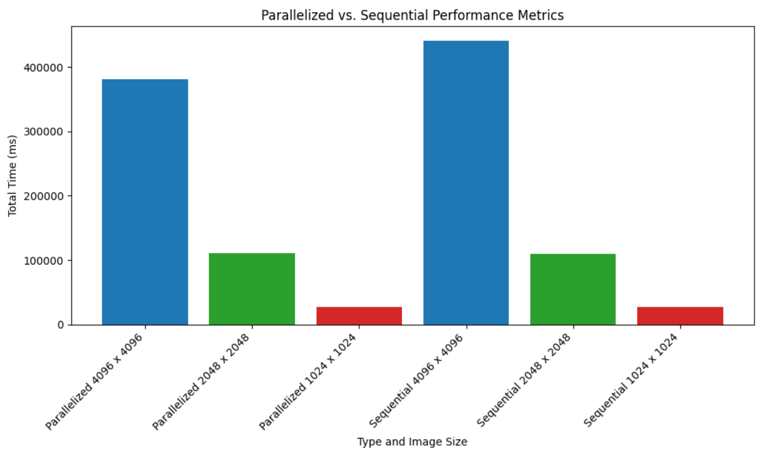


Figure 4: Mandelbrot performances depending on their size and computation methods

To look deeper into the implementation of this graph, the source code can be found in the plot results directory. The results given give a visual measure of how much the results can scale with such a number of threads. Then, running the Mandelbrot code with different image sizes the values will eventually converge when we have small differences in the image pixel size. This is given by the fact that the difference in pixels is not too big whereas the overhead is still high.

Table 1: Parallelized and Sequential Performance Metrics

Type	Total Time (ms)	Image Size	Iterations	Avg. Time per Pixel (μs)	Avg. Time per Iteration (μs)	Iterations/second	MFlop/s
Parallelized	380625	4096 x 4096	113581180678	22.687	0.00335113	2.98407e+08	2387.26
	110319	2048 x 2048	28388754939	26.302	0.003886	2.57334e+08	2058.67
	27482.5	1024 x 1024	7096659743	26.2094	0.0038726	2.58225e+08	2065.8
Sequential	441072	4096 x 4096	113607696015	26.2899	0.00388241	2.57572e+08	2060.57
	110131	2048 x 2048	28402423347	26.2573	0.00387753	2.57896e+08	2063.17
	27467.8	1024 x 1024	7103438305	26.1954	0.00386684	2.58609e+08	2068.87

3. Bug hunt [15 points]

3.1. bug1.c

In the file `bug1.c`, a compile-time error is encountered, indicating that either a `for` loop should directly follow the '#pragma omp parallel for' directive or the 'tid = omp get thread num()' call. This error arises because the '#pragma omp parallel for' directive is designed to be used in conjunction with a preceding 'for' loop. To address this issue, it is necessary to restructure the order of the lines within the code.

3.2. bug2.c

In the source file `bug2.c`, the thread identifier (tid) is expected to be distinct for each thread. However, this is not the case because the tid variable is shared among threads due to its placement outside of the parallel block. To address this problem, it is necessary to make the tid variable private.

3.3. bug3.c

In the code file `bug3.c`, the threads executing section 1 and section 2 encounter a problem where they become trapped within those sections due to a barrier placed just before printing the results. Consequently, the synchronization of threads prior to program termination becomes unattainable, resulting in the program never completing. To address this issue, it is recommended to eliminate the barrier located within the code's result printing section.

3.4. bug4.c

Because the variable 'a' is declared as private among the threads, a separate allocation of an array of size NxN occurs for each thread. This leads to accumulating multiple NxN arrays in memory, exceeding the available stack limit. To address this issue, there are two potential solutions: either determine the stack limit and adjust the value of N accordingly, or increase the stack limit to accommodate the desired array size for each thread.

3.5. bug5.c

A situation of deadlock arises because the threads that acquire locks for the 'a' and 'b' arrays fail to release those locks at the appropriate moments, resulting in a situation where they are indefinitely blocked, waiting for each other's locks. To address this issue, a solution involves having the threads release the locks for the current arrays before attempting to acquire locks for the other arrays, enabling an exchange.

4. Parallel histogram calculation using OpenMP [15 points]

The parallel histogram exercise starts from a sequential implementation of the problem and then it is asked to provide a parallel implementation of it. In order to optimize the histogram I used only a reduction openMP construct which gave me very good results in terms of performance. This is the modified code for the histogram:

```
  # pragma omp parallel for reduction(+:dist[:BINS])
  for (long i = 0; i < VEC_SIZE; ++i) {
    dist[vec[i]]++;
  }
  time_end = wall_time();
```

Now we can see the figures with using $t = 1, ..., 24$ numbers of threads

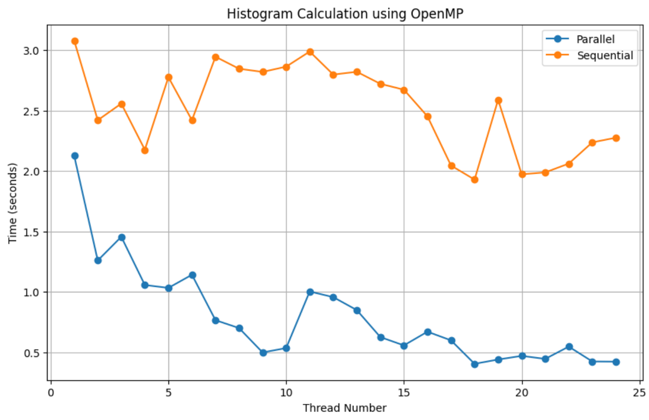


Figure 5

This exercise proves the power of a reduction openMP section that with only a line of code can make huge improvements. Keeping on adding threads, the result of the performance scale tends to stabilize as it is possible to see either from the graph or from the values. To sum up, the usage of OpenMP can boost the performance but the number of threads should be set properly too because after a while the benefits that come from a constant addiction to threads for the parallelization can be overridden by the overhead of the spawn and fork operations.

5. Parallel loop dependencies with OpenMP [15 points]

In this exercise starting from a sequential version of the program we had to obtain a parallel one with the only constraint of parallelizing the cycle with two openMP constructs

- firstprivate: it is a constraint similar to private but with the difference that it initializes all the private instances of a parallel region with the same value, namely the last value which that variable had before the cycle.
- lastprivate: In this case instead, it is still a closer constraint to private but with the difference that propagates the last value that a certain variable had outside the loop. So, in this case, the last iteration of a for loop will decide the outer value of a variable.
- Schedule: Used to control the distribution of loop iterations among the available threads when parallelizing a loop. It allows you to specify how iterations are divided among threads and can influence the workload balancing and data locality. The schedule clause provides several scheduling options.

```
int i = 0;
double og_Sn = Sn;
# pragma omp parallel for firstprivate(Sn, i) lastprivate(Sn) schedule(guided)
for (n = 0; n <= N; ++n) {
    if (i == 0 || i != n) {
        opt[n] = og_Sn * pow(up, n);
        Sn = opt[n] * up;
    } else {
        opt[n] = Sn;
        Sn *= up;
    }
    i = n + 1;
}
```

For each thread, the loop checks if the thread performed the previous iteration, which is kept track of by the variable i. If not, opt[n] is initialized by the power operator. Since the usage of the function pow is costly, the number of times it is called is reduced by checking i in each iteration. If the current thread was the one executing the previous loop, the iteration is carried out the same way as the sequential solution.

The results of parallel and sequential loops can be seen below :

Table 2: **Parallel** Computation(**Guided**) Results

Metric	**Value**
Parallel RunTime	3.895321 seconds
Final Result Sn	485165097.62503749
Result $\|\|opt\|\|_2^2$	5884629305179471.000000

Table 3: **Parallel** Computation(**Auto**) Results

Metric	**Value**
Parallel RunTime	4.632961 seconds
Final Result Sn	485165097.62551695
Result $\|\|opt\|\|_2^2$	5884629305189192.000000

Table 4: **Parallel** Computation(**Static**) Results

Metric	**Value**
Parallel RunTime	3.020055 seconds
Final Result Sn	485165097.62551695
Result $\|\|opt\|\|_2^2$	5884629305189192.000000

Table 5: **Sequential** Computation Results

Metric	**Value**
Sequential RunTime	5.014161 seconds
Final Result Sn	485165097.62511122
Result $\|\|opt\|\|_2^2$	5884629305179574.000000

The logic of the code here is that we need to initialize all the variables in the correct way, since they are parallelized, it is important to initialize $Sn = up^{ncount}$ where count is always 0 for all thread at the first iteration since it is declared first private. Then for the other iteration, we keep on doing the normal operation $Sn = up$ in order to initialize all the other values of Sn and to update opt[n] correctly. Without the first initialization, the final value of Sn produced would be the value of Sn of the last iteration which is not what it is expected. The goal here is to have a merged value of Sn without using pragma reduction and this initialization would make the for work in the same way without having to use the latter construct. The values can be slightly different in the parallelized execution because of the presence of the pow function that causes a small loss of information.