Università della Svizzera italiana | Institute of Computing CI

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

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 2

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP

1.1. Dot Product

The implementation of the parallel versions of the Dot Product are given in Listing 1 and 2. Important to note here is that the critical construct is not used to compute `alpha_local` directly, which would lead to a serial code with an additional overhead. Rather compute the `alpha` locally on each thread and then use the critical construct to compute the final `alpha`. Both methods essentially perform a reduction operation, but is there a difference in performance?

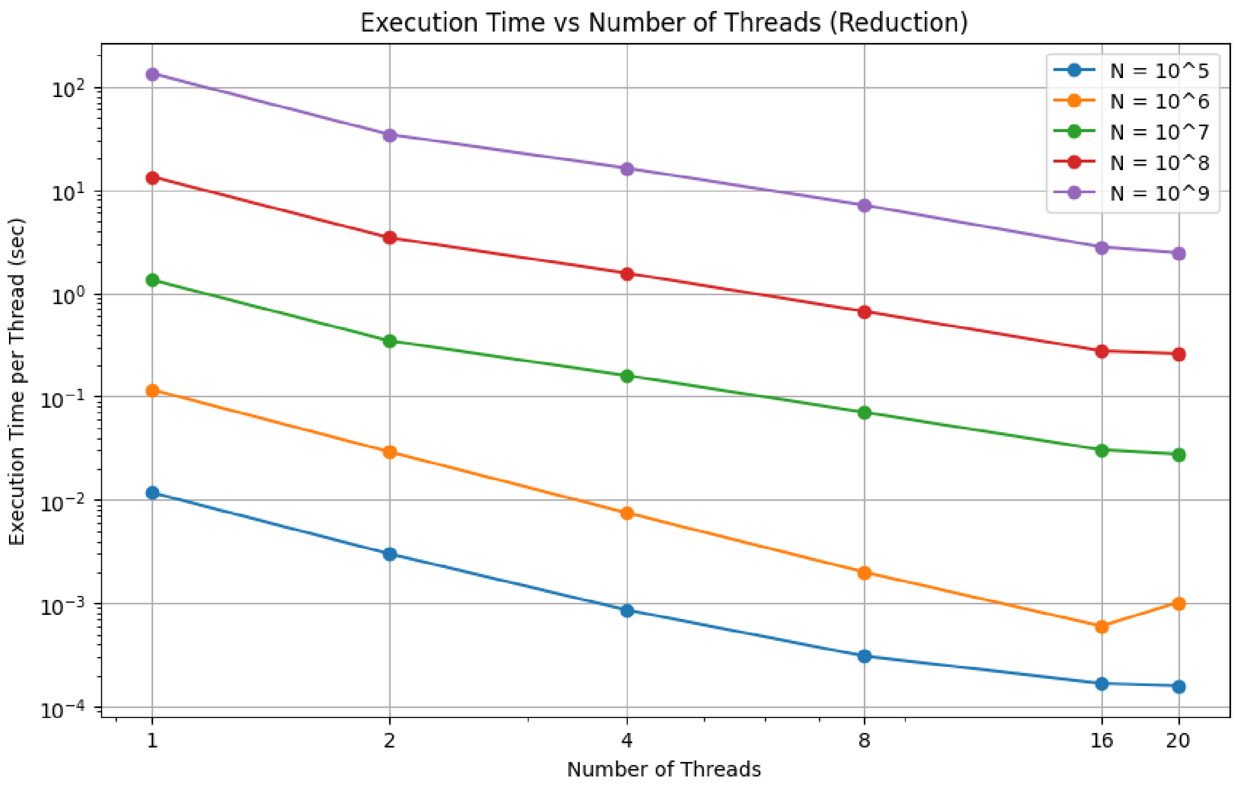
```
  for (int iterations = 0; iterations <
     NUM_ITERATIONS; iterations++) {
    alpha_parallel_red = 0.0;
#pragma omp parallel for reduction(+ :
    alpha_parallel_red)
    for (int i = 0; i < N; i++) {
      alpha_parallel_red += a[i] * b[i
    ];
    }
  }
```

Listing 1: Reduction Method

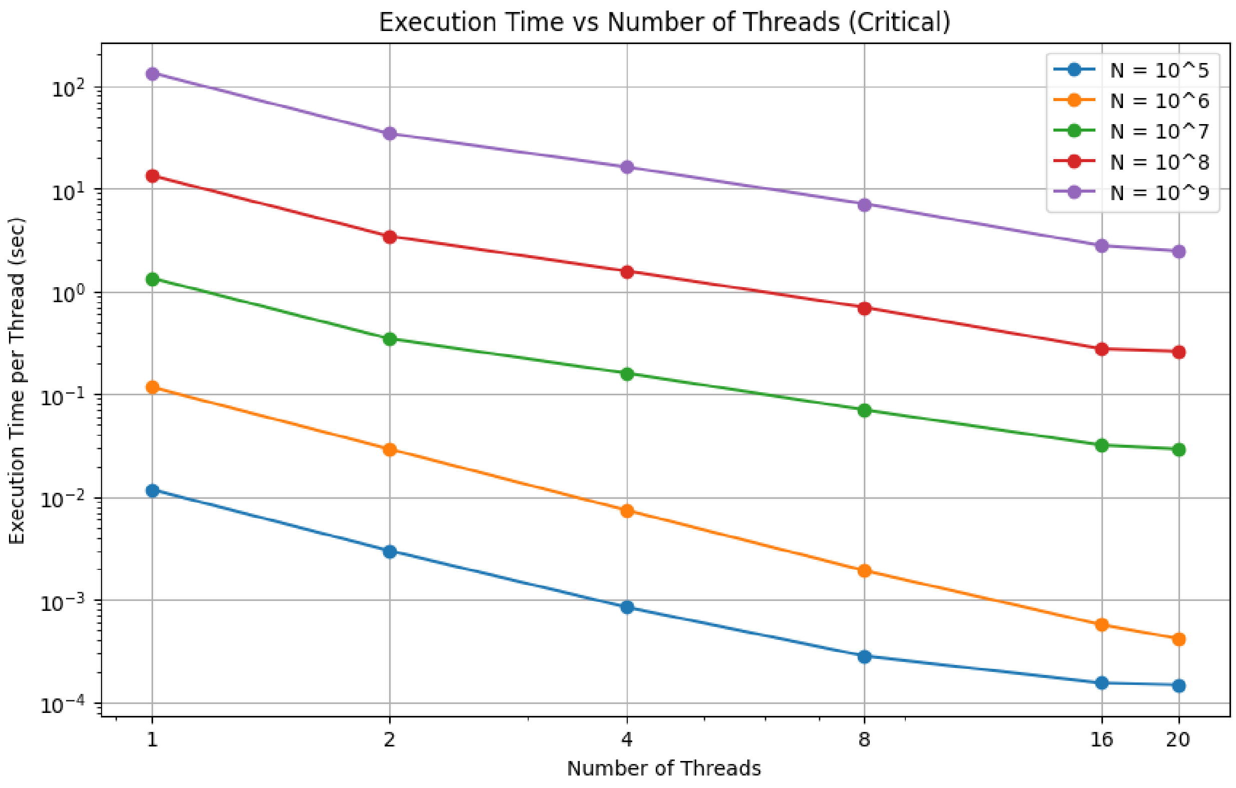
```
for (int iterations = 0; iterations <
    NUM_ITERATIONS; iterations++) {
  alpha_parallel_crit = 0.0;
  alpha_local = 0.0;
#pragma omp parallel firstprivate(
    alpha_local)
{
#pragma omp for
  for (int i = 0; i < N; i++) {
    alpha_local += a[i] * b[i];
  }
#pragma omp critical
  {
    alpha_parallel_crit += alpha_local;
  }
}
}
```

Listing 2: Critical Section Method

In order to figure this out we use strong scaling analysis, where the amount of work stays constant (fixed size of the vector length) [1]. For this analysis we introduce a plot that compares the execution time per thread to the number of threads 1. Looking at the values of the plot, they appear to be identical,yet having a closer look at the actual values, we can see that the reduction method is a tiny bit faster, which is due to having a lower thread overhead compared to the critical implementation. For both methods we can clearly deduct that when increasing the number of threads from 1 to 16, we can see an significant decrease in execution time, while increasing the number of threads from 16 to 20 we either get a very small decrease in execution time or the execution even increases. This is mostly likely caused by the overhead caused the synchronization and the increased competition of resources, such as cache coherency and memory bandwidth. Comparing the slope of the curves, in both cases we can clearly see that for vector sizes 10^5 and 10^6 the execution time decreases more rapidly, meaning when we add more threads to this vector size for small number of threads the speedup is a lot better and faster.

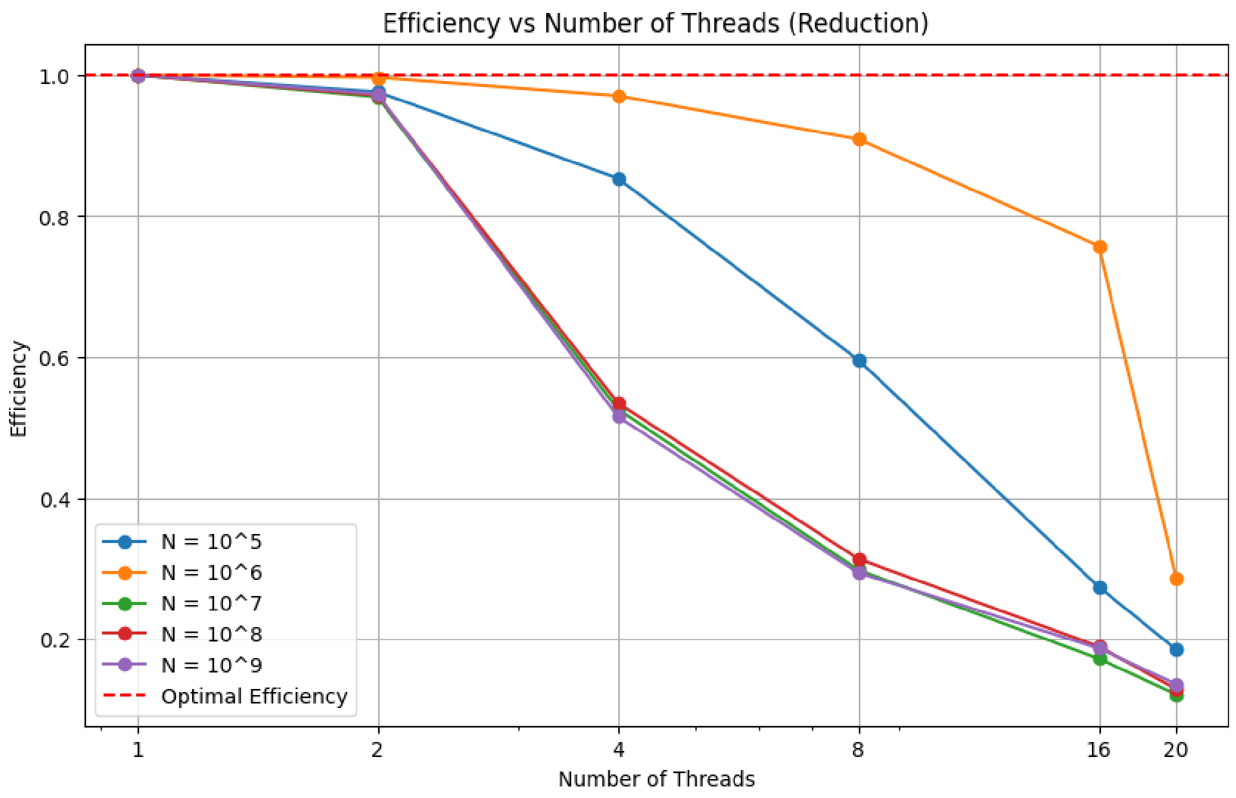


(a) DotProduct using the Reduction clause

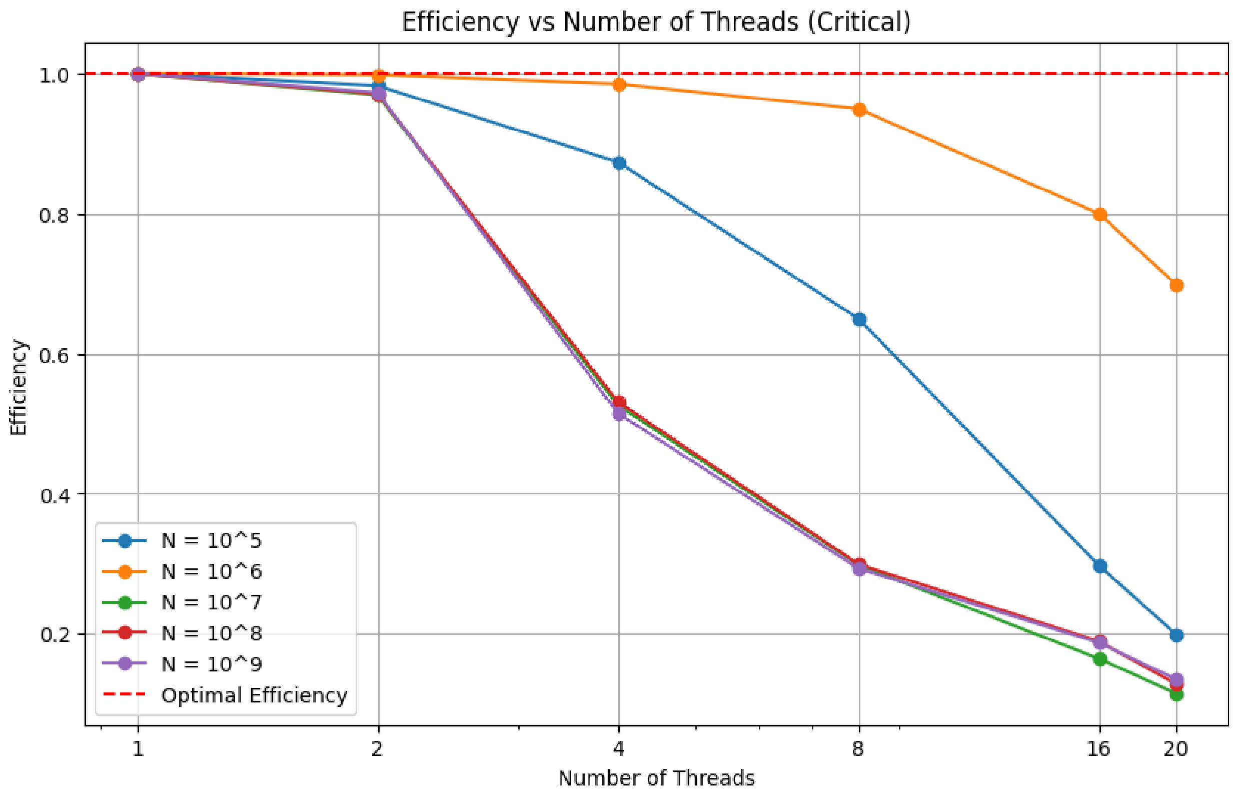


(b) DotProduct using the Critical section

Figure 1: Strong scaling comparison of the Reduction method and the Critical Method using the execution time per thread



(a) Efficiency of DotProduct using the Reduction clause



(b) Efficiency using the Critical section

Figure 2: Efficiency comparison of the Reduction method and the Critical Method

Another question posed in scalability is how effectively can a given resource be used in a parallel program [1]. We can define parallel efficiency as

$$E = \frac{T_1}{nT_n} \tag{1}$$

where T_n is the execution time of thread n-th thread. Looking at the efficiency plots in Figure 2, we can observe once more that both methods critical and reduction behave almost identical. The plot also clearly shows that for vector sizes 10^5 and 10^6 can use the small number of threads more efficiently, while for a higher number of threads they experience a sharp drop in efficiency. This suggests that for higher number of threads the managing of threads is high in comparison to the computational work. For a larger vector size initially there's a large decline of efficiency, but with increasing number of threads the decline is rather gradual, which makes sense, since large arrays can distribute their elements more effectively over a large number of threads. In addition for a larger array, the time OpenMP needs to create and manage threads is lower in relation to the time the threads needs to compute the dot product. If we compare the different vector sizes with the optimal efficiency then the for the two smallest vector sizes are the most efficient or in other words closest to the optimal efficiency.

At what point the multi-threaded version of the dot product becomes beneficial depends on the balancing of the overhead versus the amount of work per thread. This is usually indicated by a sharp drop in the efficiency plot. In our case beneficial for all N because the multi-threaded version is faster than the serial, but it is important not to use too many threads, depending on the vector size.

1.2. Approximating π

For my parallel version of pi simply use `omp parallel for` directive to distribute the work over the available threads. In addition I use the reduction clause to ensure the safe combination of the individual sums coming from each thread, avoiding race conditions when multiple threads write to a shared variable. The load balancing is done automatically by OpenMP, which should be well optimized for such a simple operation.

```
  /*  Serial Version of Computing Pi*/
  for (long long int i = 0; i < N; i++) {
    x = dx * (i + 0.5);
    sum += 4 / (1 + x * x);
  }
  pi = sum * dx;

  /*  Parallel Version of Computing Pi*/
  sum = 0.;
#pragma omp parallel for reduction(+ : sum)
  for (long long int i = 0; i < N; i++) {
    x = dx * (i + 0.5);
    sum += 4 / (1 + x * x);
  }
  pi = sum * dx;
```

Listing 3: Serial and Parallel implementation of Pi

The speedup is defined by the ratio

$$S_n = \frac{T_1}{T_n} \tag{2}$$

where T_1 and T_n are the execution time of the serial and parallel version with n threads. It shows us how much faster a parallelized program runs in comparison to its serial counterpart. The optimal (theoretical) speedup is only achievable with perfect parallelization, where no overhead or bottlenecks exist and is proportion to the number of threads $S_n = n$. In realty this not achievable, because in order to run a program in parallel we always have an overhead for communication,

synchronization, load balancing among others. There are different types of speedup plots we can look at in this case we look at strong scaling problem, which means the input size stays fixed at vector length 10^{10}.
The efficiency is closely related to the speedup and as seen in equation 1, it is the speedup divided by the number of threads.

Looking at the results of the speedup plot in Figure 3, this implementation of computing Pi comes very close to the optimal speedup, which means that the overhead is being held to a minimum and adding an additional thread halves the execution time. In terms of efficiency for this program it is very close to 1 and creates approximately a constant line.

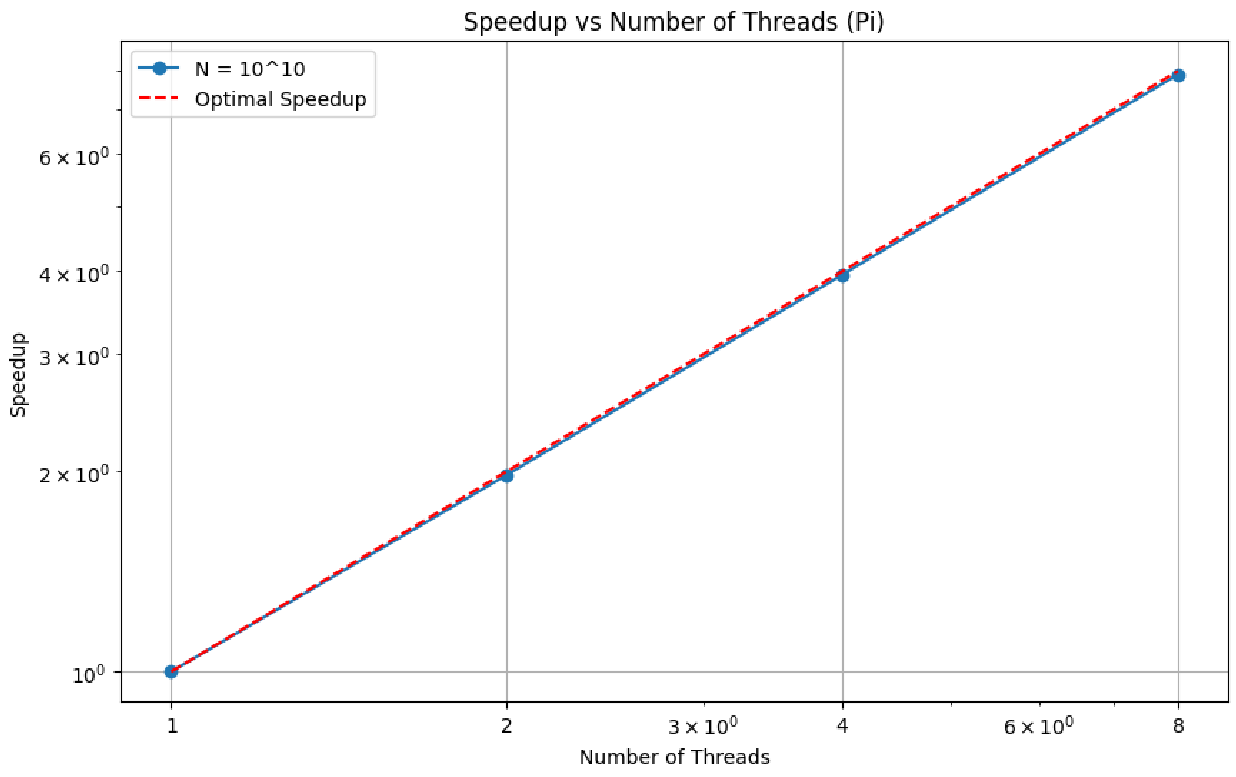


Figure 3: Speedup plot for computing pi

2. The Mandelbrot set using OpenMP

For the Mandelbrot using OpenMp we base our strong scale analysis on the closely related speedup and efficiency plot in Figure 5. For reference the dotted curves represent in both plots theoretical optimum. The first point that stands out is that the speedup and the efficiency does not depend on the resolution. This suggests that when increase the resolution the proportion of workload remains the same. Looking at the observed speedup, from single to 2 threads the speedup almost ideal, but from 2 to 4 threads the speedup stays almost identical, following that, the speedup increases steadily. A key observation is that for higher number of threads the speedup moves farther away from the ideal curve due to increased costs, such as synchronization time, memory bandwidth limitations and cache contention. This behavior can directly translated to the efficiency plot as well, where the efficiency decreases quickly and then deceases at a lower rate until from 16 to 20 threads the efficiency is almost identical. This is a common behavior in strong scaling experiments, because at a certain point the same factor as mentioned in the speedup plot start to weigh in and slow the implementation down, particularly with a higher number of threads.

Threads	**Iterations/second**	**MFlop/s:**	**Total time:**
1	2.075e+08	1660.67	155.782
2	4.144e+08	3315.96	78.036
4	4.291e+08	3433.23	75.370
8	6.393e+08	5115.16	50.587
16	1.130e+09	9045.01	28.608
20	1.397e+09	11177.4	23.150

Table 1: Subset of benchmark key values for the resolution 4096 × 4096 with roughly 32.34 billion iterations

The benchmarking used in this exercise also evaluates other fascinating key values, in Table 1 we can see an example for a specific resolution, where we can clearly see that when increasing the number of threads more iterations per second can be executed, which is also reflected in the MFlop/s and the decreasing total execution time. This shows the effective distribution of the workloads across multiple threads, but the gains made by adding more threads, as seen in the efficiency and speedup plot, will at some point have diminishing returns. Overall the performance gains are significant, especially up to 16 threads.

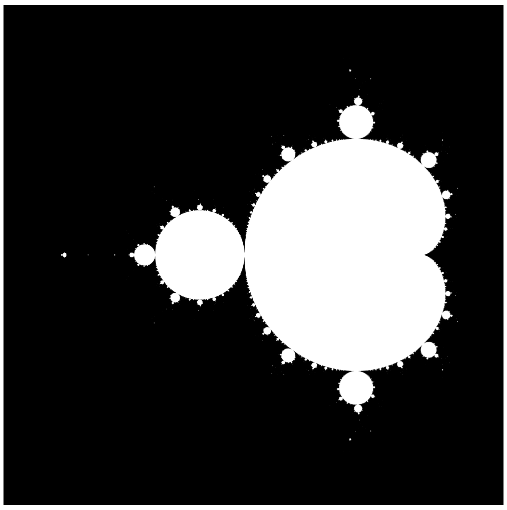
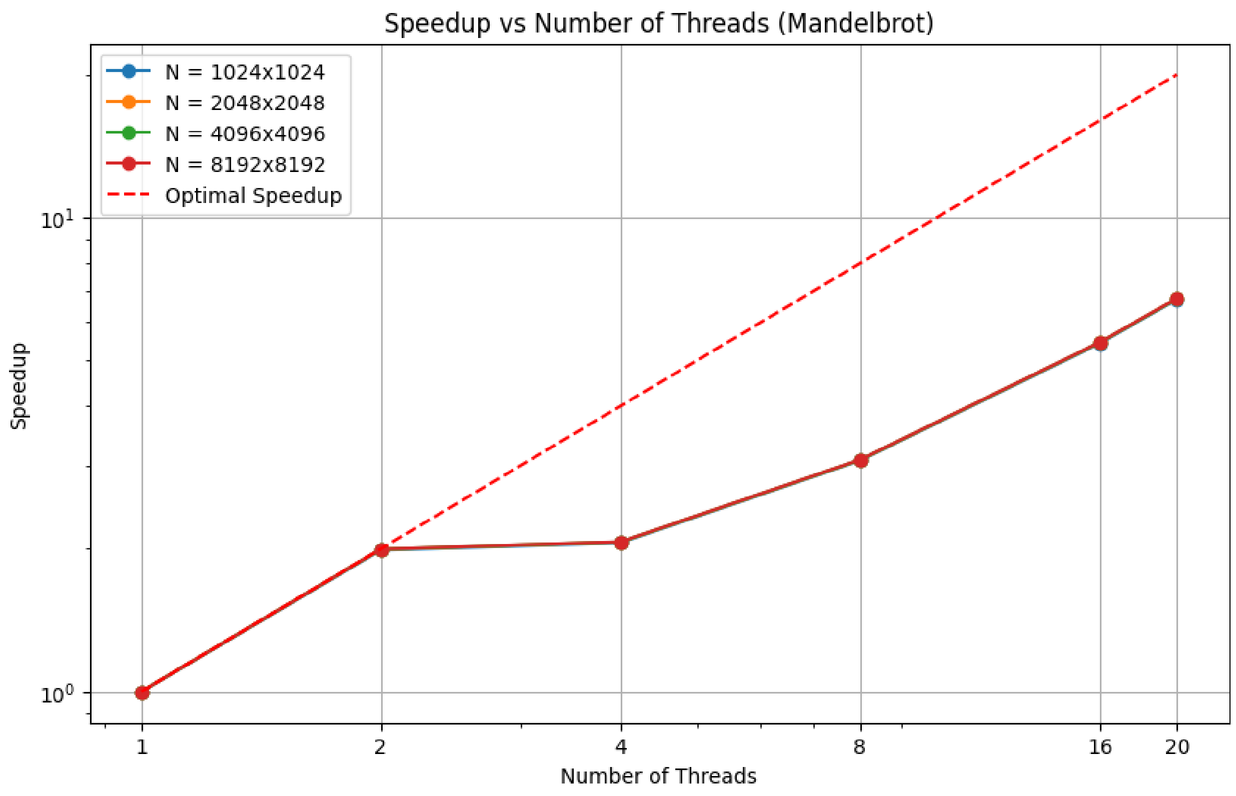
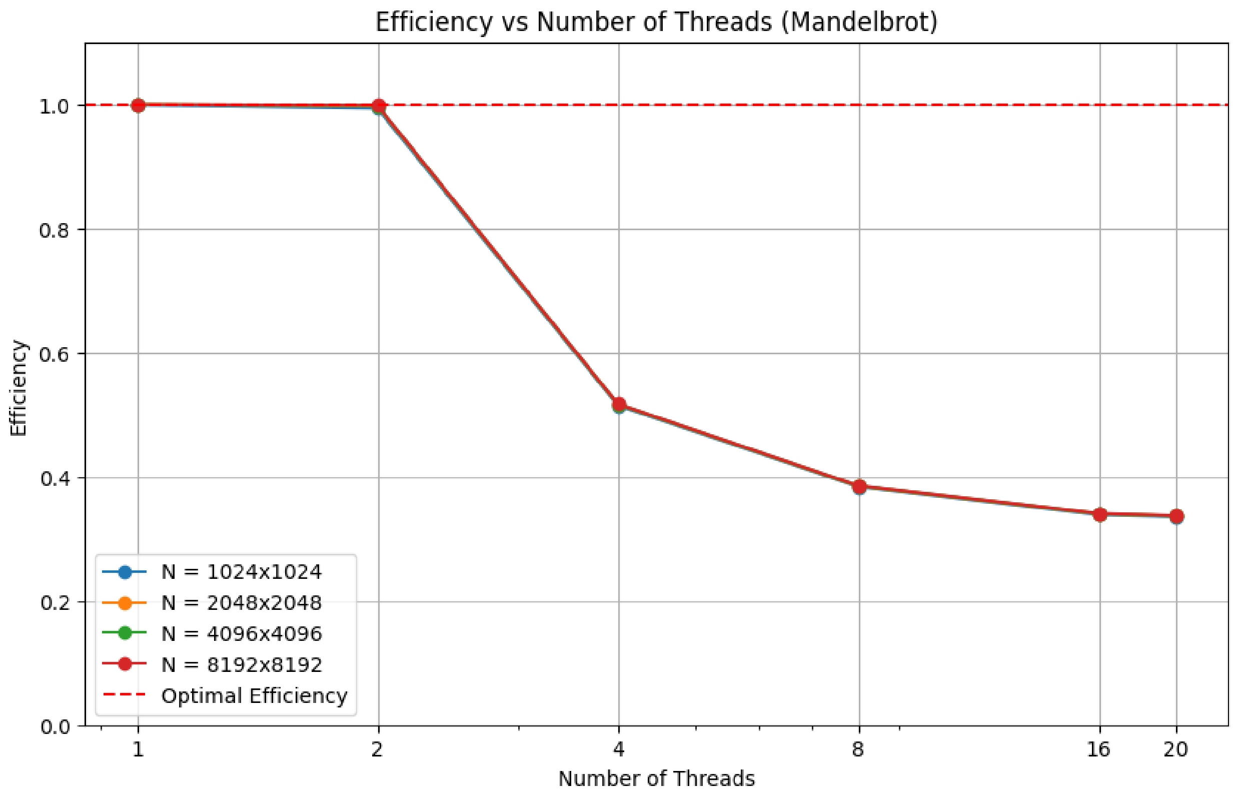


Figure 4: The Mandelbrot set with resolution 4096 × 4096 using parallel OpenMP implementation

Also important to mention is that in my implementation, I use `omp for collapse(2)` statement, in order to parallelize over both outer loops, which iterate over the x and y coordinates. I used this with the premise to allow multiple loops to be collapsed into a single loop and parallelize them a single entity, hence trying to have the iteration space evenly spread out over all processes. Comparing my results with two of my fellow students, which use a standard `omp for` loop, showed no significant improved or deterioration in execution time. This leads me to believe that OpenMP performs very similar optimization based on both statements.



(a) Speedup



(b) Efficiency

Figure 5: Speedup and Efficiency for the Mandelbrot implementation

3. Bug Hunt

3.1. Bug 1: Check tid

For the first bug, we can find the issue in the `#pragma` statement, see Listing 4, which is caused by already including the `for` and the `schedule(static, chunk)` statement, both of which can only be used right before a for-loop.

```
#pragma omp parallel for shared(a, b, c, chunk) private(i, tid) \
              schedule(static, chunk)
  {
    tid = omp_get_thread_num();
    /* CODE */
  }
```

Listing 4: Bug in bug1.c

In order to fix this bug, we need to split the `#pragma` statement in two distinct ones. The first one indicating that we are in a parallel region, shown on Line 1 of Listing 5 and the second `#pragma omp for` statement on Line 4 right before the loop we want to parallelize.

```
1 #pragma omp parallel shared(a,b,c,chunk) private(i, tid)
2 {
3     tid = omp_get_thread_num();
4 #pragma omp for schedule(static, chunk)
5     for (i = 0; i < N; i++) {
6       c[i] = a[i] + b[i];
7       printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
8     }
9 }
```

Listing 5: Fix for bug1.c

3.2. Bug 2: Check shared vs private

The second bug can be fixed by changing the variables `tid` and `i` to private variables.

```
#pragma omp parallel private(tid, i)
```

Listing 6: Fix bug2.c

By doing this every thread is now assigned its proper thread Id (`tid`) and the number of threads is now only printed by thread with index 0. Furthermore by setting `i` to private, each thread can now process its chunk of the `total`, otherwise `i` gets incremented by every thread, which could lead to certain `i` being skipped in the `total` computation.

3.3. Bug 3: Check barrier

For this bug it is critical to understand what the `sections` constructs does. Consulting the documentation [2], sections distribute work among the threads in a parallel region. This means in our case one of the threads executes section 1, another section 2 and the rest are preceding to the `omp barrier` after the `omp sections nowait` region. Important to note here is that the `nowait` option is used which removes the default barrier when exciting the `sections` construct.
In addition both `omp section` call the `print_results` function, which includes another `omp barrier` statement at the end of the function.
In OpenMP, threads cannot differentiate between different barriers in parallel region, which means that when the two threads executing the code in the section 1 and 2 reach the `omp barrier` statement in the `print_results` function, all the other threads are continuing executing the code, because every thread reached a barrier. When the threads that executed the sections reach the barrier after the section they halt there and wait until all the other threads reach barrier, which never happens, hence the program does not halt.

The solution to fix this bug is simply to remove the `omp barrier` statement at the end of the `print_results` function. Then the threads not entering a section, are waiting until the other two threads are finished with their section and all the threads are printing the exciting together.

3.4. Bug 4: Stacksize

In OpenMP there exists a stack size limit for each thread in a parallel region. On Unix like systems you can run the command

```
ulimit -a
# Example output in KB
8176
```

Listing 7: Find current stacksize on Unix-like system

In our case we a 1048×1048 matrix of type double, for which every thread gets its own copy. This Matrix therefore requires $8 \cdot 1048 \cdot 10480.001 = 8786.432KB$ of storage. In our example the matrix requires more storage than the stack size given in 7. In order to store further variables, you have to increase the storage size even further. This is possible by using the following command

```
ulimit -s 9000
```

Listing 8: Setting new stacksize on Unix-like system

3.5. Bug 5: Locking order

In our final bug we found ourselves in a deadlock situation in each of the `omp section` which are execute in parallel, where a thread locks `locka` and simultaneously another thread locks `lockb`, as a next step both threads want to lock each others `omp_lock_t`, but because its already locked they cannot do that, resulting in a deadlock. The solution can be seen in the simplified code in Listing 9, where each thread only locks one of the variable and then as soon as the lock is not needed anymore it is unlocked.

```
#pragma omp sections nowait
    {
#pragma omp section
      {
        omp_set_lock(&locka);
          /* MORE CODE */
        omp_unset_lock(&locka);
        omp_set_lock(&lockb);
          /* MORE CODE */
        omp_unset_lock(&lockb);
      }

#pragma omp section
      {
        omp_set_lock(&lockb);
          /* MORE CODE */
        omp_unset_lock(&lockb);
        omp_set_lock(&locka);
          /* MORE CODE */
        omp_unset_lock(&locka);
      }
    } /* end of sections */
```

Listing 9: Non-Deadlock fix for omp_bug-5.c

4. Parallel histogram calculation using OpenMP

In this exercise we will be analyzing the strong scaling of the parallel implementation of the histogram computation. The measured timings can be found in Table 2. Because this is strong scaling

analysis we kept the array size of the vector constant at $2 \cdot 10^9$ throughout the tests.
The serial execution provides us with a baseline for comparison with the parallelized versions. The single threaded execution shows a minor increase in performance, which is unexpected, due to the fact that there should be an overhead due to the thread management. This indicates that the overhead was somehow less significant and it could be based on the workload of the cluster. As we increase the number of threads, between 2 and 8 threads, we can observe a significant reduction in execution time, which indicates efficient scaling, when the workload is distributed over an increasing number of threads.
When comparing the timing from 8 to 16 threads, the execution time still slightly decreases, but at much lower rate than before. This leads to the conclusion that the efficiency decreases and the thread management overhead and potential contention between threads for shared resources.
From 32 onwards the timing increases, which means that the overhead outweighs the benefits gained from parallelizing the code. Another important factor that plays into the slower execution time is that the node only has 20 cores, resulting in more threads than cores, which means that certain threads potentially have to wait their turn until they can execute their chunk of code.

Threads	**Timing**
Serial	0.839258s
1	0.830452s
2	0.418288s
4	0.213519s
8	0.110719s
16	0.112852s
32	0.120083s
64	0.127492s
128	0.129270s

Table 2: Execution time for strong scaling parallel histogram calculation

5. Parallel loop dependencies with OpenMP

Two major changes were made in order to parallelize the code and keep it working independently of the schedule pragma. Because of the latter point, we are not allowed to use the scheduling to our advantage, therefore we have to compute the `Sn` using the power function, this guarantees that `Sn` is properly computed no matter if it is serial, dynamic or static scheduled.
Furthermore we introduce the `firstprivate` and `lastprivate` for `Sn`, therefore the predefined value before the loop is copy into a private variable for each thread and that after the parallel for loop `Sn` is assigned to the value computed of the last iteration of the loop when $n = N$. `Sn` can then be used for the print statement, which follows afterwards.

```
#pragma omp parallel for firstprivate(Sn) lastprivate(Sn)
  for (n = 0; n <= N; ++n) {
    Sn = pow(up, n);
    opt[n] = Sn;
  }
```

Listing 10: Parallelized section of recur_omp.c

References

[1] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. en. Google-Books-ID: rkWPojgfeM8C. CRC Press, July 2010. ISBN: 978-1-4398-1193-1.

[2] *IBM OMP Sections*. en-US. June 2024. URL: `https://www.ibm.com/docs/en/zos/3.1.0?topic=processing-pragma-omp-section-pragma-omp-sections` (visited on 10/20/2024).