Università della Svizzera italiana

Institute of Computing CI

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

Student: Varese Lorenzo Discussed with: Sen Kaan Egemen

Solution for Project 2

This project will introduce parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

1.1. Implementation Details

For this exercise, we focus on the task of summing up the product of two vectors a and b of length N. The scalar α is calculated as:

$$\alpha = \sum_{i=0}^{N-1} a[i] \times b[i]$$

1.1.1. Caution Regarding a Specific Implementation Choice

In the provided code, multiple iterations of the dot product are performed. Given that there are no specific constraints outlined in the requirements, I opted to focus the parallelization exclusively on the computation of the dot product itself. I chose not to parallelize the outer loop responsible for multiple iterations, as it is not intrinsically related to the dot product computation. Its primary function appears to be extending the runtime for benchmarking purposes.

1.1.2. Serial Implementation

A standard serial implementation is provided as a baseline for comparison.

```
double alpha = 0.0;
for (int i = 0; i < N; ++i) {
    alpha += a[i] * b[i];
}
```

1.1.3. Parallel Implementation with Critical Section

In this parallelized version of the algorithm, a shared variable, denoted as $\alpha_{parallel}$, is updated within a critical section. This is done to guarantee mutual exclusion among the threads, thereby preventing race conditions or inconsistent updates to the variable. This critical section acts as a synchronization point, ensuring that only one thread at a time can modify $\alpha_{parallel}$, preserving the integrity of the computation.

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

1.1.4. Parallel Implementation with Reduction Clause

In this optimized strategy, I employ OpenMP's 'reduction' clause for the efficient aggregation of the α variable. Specifically, the 'reduction' clause automates the process of creating private copies of α for each thread and then accumulating these into a global sum in a thread-safe manner. This eliminates the need for manual synchronization, thereby improving computational performance and reducing the likelihood of concurrency-related errors. By leveraging the 'reduction' clause, we can achieve both computational efficiency and code simplicity.

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

1.2. Performance Evaluation

The execution times were recorded for N sizes of "100,000", "1,000,000" "10,000,000" and "100,000,000", with thread counts ranging from 1 to 24.

For $N = 100,000$, the serial version took 0.053s, the critical section version took 0.285s with 4 threads, and the reduction version took 0.013 with 4 threads. This indicates that parallelization with reduction clause provides a significant speedup.

More in general, the data suggests that parallelizing the operation yields considerable performance gains, especially as the size of N increases. The use of OpenMP's `reduction` clause shows significant advantages in execution time over using a `critical` section.

For a multi-threaded version to be beneficial, N should be sufficiently large to overcome the overheads introduced by parallelization. Based on the performance data, it seems that a multi-threaded version starts to show benefits from N=100,000 and onwards.

From the comprehensive runtime data gathered across all tests, I observe that the Parallel Critical method generally performs poorly in the majority of cases. This suboptimal performance could stem from the parallelization of the loop responsible for calculating the dot product. However, due to assignment constraints, thread synchronization is necessary to modify the dot product value in a thread-safe manner. Consequently, this parallelization strategy, when employing critical sections, incurs a significant overhead. Rather than enhancing performance, this overhead substantially deteriorates it.

1.3. Performance Analysis Through Graphical Representation

In this section, we evaluate and compare the performance of different implementations—both sequential and parallel—by analyzing their respective execution times. The graphical representations offer an in-depth look into the efficacy of the applied strategies under various array sizes and number of threads.

The Figure 1 provides a complete overview of the execution times when applying different strategies. It serves as an initial snapshot for a broad comparison, encompassing both the sequential and parallel methods.

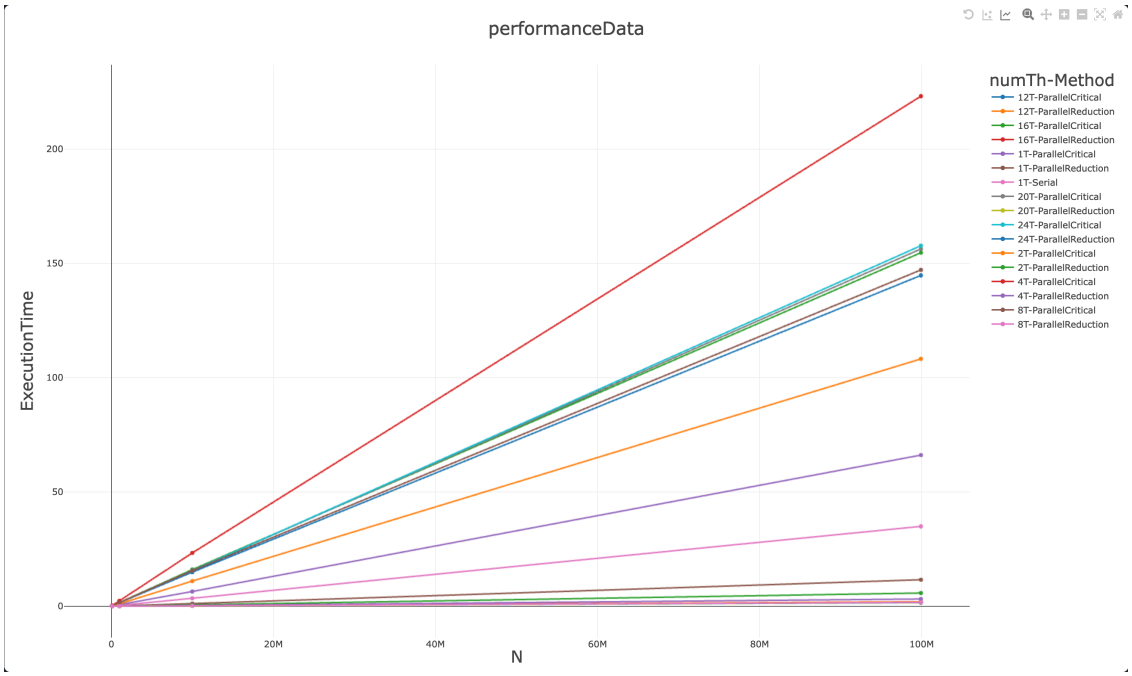

Figure 1: Overview of execution times (seconds) for sequential and multiple parallel strategies.

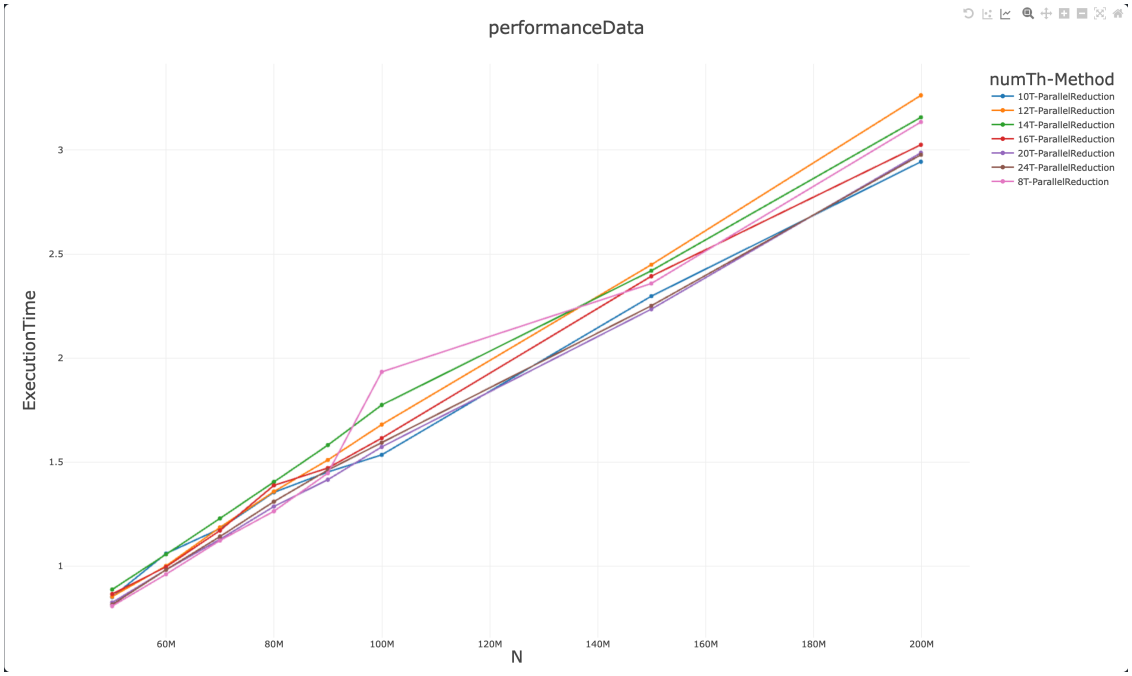

Figure 2: Line graph comparing execution times of different parallel reduction strategies across varying thread counts. The focus is on isolating the best-performing strategy under diverse multi-threading scenarios.

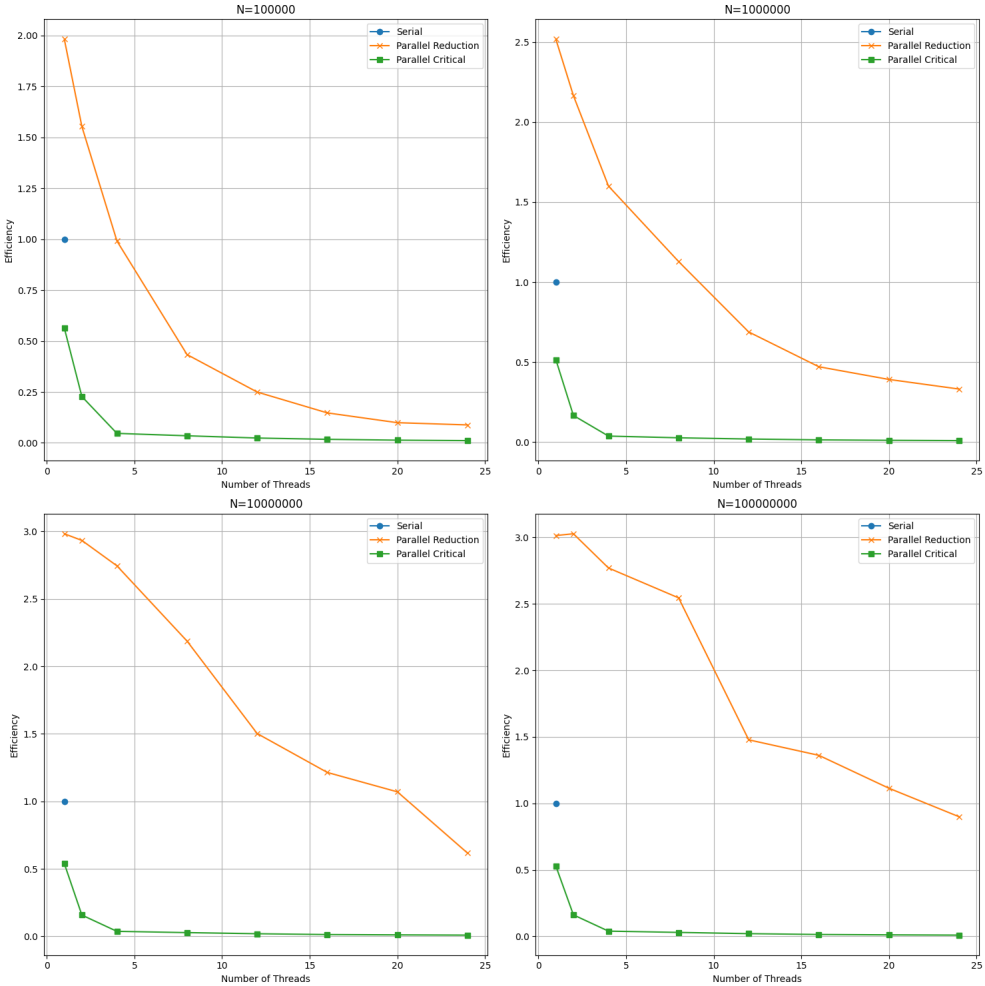

Figure 3: Line graph comparing efficiency.

In Figure 2, attention is focused on the most efficacious parallel strategies. From the perspective of runtime performance, optimality can be achieved with a thread count of 12.

1.3.1. Efficiency Analysis

It's crucial to plot the parallel efficiency of the different dot product benchmarks. The efficiency is typically calculated as:

$$Efficiency(p) = \frac{T_{serial}}{p \times T_{parallel}(p)}$$

Where T_{serial} is the execution time for the serial version, p is the number of threads, and $T_{parallel}(p)$ is the execution time for the parallel version using p threads.

Based on the efficiency data in Figure 3, the efficiency is highest for the Parallel Reduction method, especially as the size N increases. The Parallel Critical method shows significant inefficiency as N and the thread count increase. It is particularly inefficient possibly due to high synchronization costs.

Therefore, Parallel Reduction typically demonstrates greater efficiency and is more aptly suited for tackling large-scale problems. In contrast, both methods exhibit suboptimal performance when applied to smaller problems, especially as the thread count begins to escalate. While the specific thread count at which efficiency starts to deteriorate is contingent upon N, a similar decline is generally observable around 10 threads.

2. The Mandelbrot set using OpenMP [30 points]

2.1. Mandelbrot set parallel implementation

The Mandelbrot set is calculated by iterating through each point in the complex plane to check if it belongs to the set. This inherently parallel task was implemented using OpenMP.

```
long i, j, n;
#pragma omp parallel for default(none)
     shared(fDeltaX, fDeltaY, pPng) collapse(2)
     reduction(+ : nTotalIterationsCount)
for (int j = 0; j < IMAGE_HEIGHT; j++)
{
       for (int i = 0; i < IMAGE_WIDTH; i++)
       {
              double cx = MIN_X + i * fDeltaX;
              double cy = MIN_Y + j * fDeltaY;
              double x = cx;
              double y = cy;
              double x2 = x * x;
              double y2 = y * y;
              int n = 0;
              // Innermost loop remains serial (due to collapse(2))
              for (n = 1; (x2 + y2) < 4 && n < MAX_ITERS; n++)
              {
                     y = 2 * x * y + cy;
                     x = x2 - y2 + cx;
                     x2 = x * x;
                     y2 = y * y;
              }
              nTotalIterationsCount += n;
              int c = ((long)n * 255) / MAX_ITERS;
              png_plot(pPng, i, j, c, c, c);
       }
}
```

The provided code demonstrates the parallel calculation of the Mandelbrot set using OpenMP. The task at hand involves iterating over each point in the complex plane defined by the image dimensions, to ascertain whether or not each point belongs to the Mandelbrot set. OpenMP directives are employed to achieve parallel execution.

The OpenMP ‘pragma omp parallel for‘ directive is used for parallelizing the nested for-loops, with the ‘default(none)‘ clause ensuring that all variables in the parallel region are explicitly scoped, thereby enhancing code safety. The ‘shared(fDeltaX, fDeltaY, pPng)‘ clause dictates that these variables are shared among all threads, as they are read-only within the parallel region and thus safe to share.

Additionally, the ‘collapse(2)‘ clause is employed to flatten the nested for-loops, enabling better load balancing by increasing the number of available loop iterations for scheduling among threads. A reduction operation on ‘nTotalIterationsCount‘ is performed through ‘reduction(+ : nTotalIterationsCount)‘, wherein each thread conducts its own local reduction, and at the end, all local copies are summed up.

The computational core resides in the innermost loop, which remains serial due to the action of the ‘collapse(2)‘ clause. This loop undertakes complex number calculations to determine whether or not a point is part of the Mandelbrot set. Subsequently, the color of each pixel is set based on the number of iterations required for the point to escape, if it does.

In summary, this OpenMP-enabled approach significantly accelerates the computation by effec-

tively leveraging the capabilities of multi-core processors. The parallelism is inherently suited for the task because each point is independent from the others.

2.2. Performance Measurement

The performance analysis unveils a substantial improvement in the parallel implementation compared to the serial version. For a benchmark configuration of 10,000 iterations per pixel at a resolution of 2048×2048 pixels, the observed metrics are as follows:

For the parallel version with 24 threads, the total execution time was 3341.04 milliseconds. With a total of 8,091,743,654 iterations, the average time per pixel was measured to be approximately 0.797 microseconds, and the average time per iteration was approximately 0.000413 microseconds. The performance was gauged to be around 2.42×10^9 iterations per second with a computation rate of 19,375.4 MFlop/s.

In contrast, the serial version took a considerably longer 23,760.6 milliseconds for execution. Despite a similar total number of iterations at 8,087,875,807, the average time per pixel was about 5.665 microseconds, and the average time per iteration was approximately 0.002938 microseconds. The iterations per second were recorded to be 3.40×10^8 with a computation rate of 2723.13 MFlop/s.

These numbers clearly indicate the efficacy of parallelization, with a significant decrease in the average time per pixel and per iteration. Moreover, the parallel version achieved a higher throughput, as evidenced by the iterations-per-second and MFlop/s metrics.

2.3. Benchmarking different image sizes

The algorithm for generating the Mandelbrot set was implemented with a computation kernel that performs 1000 iterations for each pixel to determine if it belongs to the set or not. In both the sequential and parallel versions of the code, this number of iterations is held constant to maintain a significant comparison. The parallelized version of the code employs OpenMP and utilizes 10 threads for the computation. The time taken for computation in both scenarios is measured using a microsecond-precise timer, and the results are presented in terms of both time and MegaFlops per second (MFs).

The algorithm's performance was evaluated using different image resolutions to gauge scalability and understand the implications on runtime and computational throughput. These tests ranged from a modest 256 x 256 resolution to a considerably more resource-intensive 4096 x 4096 pixels. The results are summarized in Table 1.

Resolution	**Sequential (ms)**	**Parallel (ms)**	**Sequential (MFs)**	**Parallel (MFs)**
256 ×256	56.035	20.727	1854.95	5030.91
512 ×512	176.614	60.198	2349.58	6904.83
1024 ×1024	632.655	57.725	2623.74	7200.64
2048 ×2048	2421.72	701.62	2740.96	9464.78
4096 ×4096	9515.63	2746.66	2790.27	9668.78

Table 1: Runtime for Sequential and different Parallel versions

3. Bug hunt [15 points]

This section delves into the identification and remediation of bugs found in a series of OpenMP programs. The programs, ranging from `bug1.c` to `bug5.c`, encompass a variety of issues including, but not limited to, compile-time and run-time errors. Each subsection will succinctly describe the bug, elucidate the underlying problem, and propose a solution.

3.1. bug1.c - tid

The `bug1.c` program aims to demonstrate the parallel for construct in OpenMP. However, it suffers from a compile-time error, attributed to the incorrect placement of the `tid = omp_get_thread_num();` line.

3.1.1. Problem Description

The variable `tid` is assigned a value using `omp_get_thread_num()` outside the loop body. This occurs within the context of the parallelized for-loop. Consequently, the variable `tid` could be uninitialized or incorrect when accessed by multiple threads.

3.1.2. Suggested Fix

To rectify this issue, the assignment of `tid` should be moved inside the loop body. This will ensure that each thread can accurately determine its own thread identifier (`tid`) while executing the parallelized loop. The corrected code snippet is as follows:

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

3.2. bug2.c - shared vs. private

The `bug2.c` program aims to demonstrate OpenMP's parallel region capabilities. It exhibits run-time errors due to race conditions in shared variables.

3.2.1. Problem Description

The issue in this program arises from the incorrect use of shared variables. Specifically, the variables `total`, `tid`, and `i` are shared among all threads, resulting in race conditions. This can lead to inconsistent or incorrect results.

3.2.2. Suggested Fix

To mitigate the issue, the affected variables should be declared as private within the parallel region. This will ensure that each thread has its own copy of these variables, preventing concurrent access issues. The corrected OpenMP pragma directive would be as follows:

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

Additionally, initialize the `total` variable inside the parallel region, before the `#pragma omp for` directive. This ensures that each thread starts with a `total` value of 0.0 before commencing the loop iteration.

The modified code snippet would look like this:

```
#pragma omp parallel private(i, tid, total)
{
  // Existing code for obtaining thread number and other operations
  ...
}
```

3.3. bug3.c - barrier

The `bug3.c` program is designed to demonstrate the usage of OpenMP's parallel sections. The code uses barriers for clean output synchronization among threads. However, the program encounters a runtime error because of incorrect barrier placement.

3.3.1. Problem Description

The issue primarily stems from the use of the `#pragma omp barrier` directive inside the `print_results` function. According to OpenMP specifications, barriers should only be used within the same structured block as the parallel region they are intended to synchronize. The `#pragma omp barrier` directive in the `print_results` function violates this rule, leading to undefined behavior.

3.3.2. Suggested Fix

To resolve this issue, the barrier statement within the `print_results` function should be removed or relocated to a position within the same structured block as its corresponding parallel region in the main function. By removing or relocating the barrier, the program adheres to OpenMP specifications, thereby eliminating the risk of undefined behavior.

3.4. bug4.c - stacksize

The `bug4.c` program attempts to perform array manipulations in a parallel manner using OpenMP. However, the code leads to a segmentation fault.

3.4.1. Problem Description

The main issue can be traced to the stack size requirements for each thread. Specifically, the program defines a large two-dimensional array `double a[N][N]` in the local scope of the `main()` function. According to OpenMP's scoping rules, this array is allocated on the stack. The `#pragma omp parallel` line specifies that each thread should have its private copy of `a` through the use of `private(i, j, tid, a)`. Consequently, the stack size requirement for each thread is multiplied by the number of threads. This combined stack size requirement could far exceed the per-thread stack size limit, leading to a segmentation fault.

3.4.2. Suggested Fix

To mitigate this issue, it is advisable to allocate the array `a` on the heap rather than on the stack. Alternatively, OpenMP environment variables or API calls could be used to increase the per-thread stack size, although this might not be a scalable solution. By moving the array to the heap, the program can avoid exceeding the per-thread stack size limit, thus resolving the segmentation fault issue.

3.5. bug5.c - deadlock

The `bug5.c` code aims to demonstrate OpenMP's capabilities in handling parallel sections. Specifically, two threads are intended to initialize their respective arrays and then add the values to the other's array. However, the code suffers from a deadlock condition.

3.5.1. Problem Description

The deadlock occurs due to the way the locks are acquired and released in the parallel sections. To elaborate, in the first section, the code acquires `locka` and then attempts to acquire `lockb`. Conversely, in the second section, the code first acquires `lockb` and then attempts to acquire `locka`. If both sections are executed concurrently, each section holds one lock while waiting indefinitely for the other to release its lock, thereby causing a deadlock.

3.5.2. Suggested Fix

One possible solution to avoid the deadlock is to enforce a strict locking hierarchy. This implies that all locks are always acquired in a pre-determined, hierarchical order, thus preventing cyclic dependencies between the locks. For example, always acquire `locka` before `lockb` in both sections.

4. Parallel histogram calculation using OpenMP [15 points]

4.1. Parallel implementation of histogram calculation

The parallel implementation uses OpenMP for enhancing the histogram calculation. Each thread in the parallelized region initializes its own local histogram (`local_dist`). This local histogram is updated in a loop, parallelized with `#pragma omp for`.

```
// Parallelize the histogram computation
#pragma omp parallel
{
  long local_dist[BINS] = {0}; // Thread-local histogram

  #pragma omp for
  for (long i = 0; i < VEC_SIZE; ++i) {
    local_dist[vec[i]]++;
  }

  // Merge thread-local histograms into global histogram
  #pragma omp critical
  {
    for (int i = 0; i < BINS; ++i) {
      dist[i] += local_dist[i];
    }
  }
}
```

Note: After the parallelized loop, a critical section (`#pragma omp critical`) is utilized to merge the local histograms into the global one (`dist`). The merging step is significantly less computationally intensive compared to the histogram computation, thereby minimizing the impact of the critical section.

4.2. Runtimes for the original (serial) code, the 1-thread and the N-thread parallel versions

The runtimes for the different configurations are as follows:

In a performance comparison, it is observed that the sequential algorithm executed with a runtime of approximately 0.8653 seconds. In contrast, employing OpenMP with a single thread introduced computational overhead, resulting in a suboptimal performance characterized by a runtime of 1.17515 seconds. Nevertheless, a marked improvement in execution time was observed when the number of threads was increased to two, resulting in a runtime of 0.498723 seconds. Further experiments involving varying numbers of threads demonstrated a consistent improvement in performance, culminating in a runtime of 0.104944 seconds with 11 threads. Beyond this point, however, the performance began to deteriorate.

As seen in Figure 4, the parallel code scales well with the increasing number of threads up to a point. The initial improvement in runtime with an increasing number of threads demonstrates the benefits of parallelization. However, when the number of threads surpasses 12, there is no substantial gain or even a slight increase in runtime. This plateauing and occasional increase are likely attributable to the hardware limitation of my local machine, which has a maximum of 12 physical threads. Deploying more threads than the hardware can support results in thread switching and scheduling overhead, thus negating the benefits of additional parallelization.

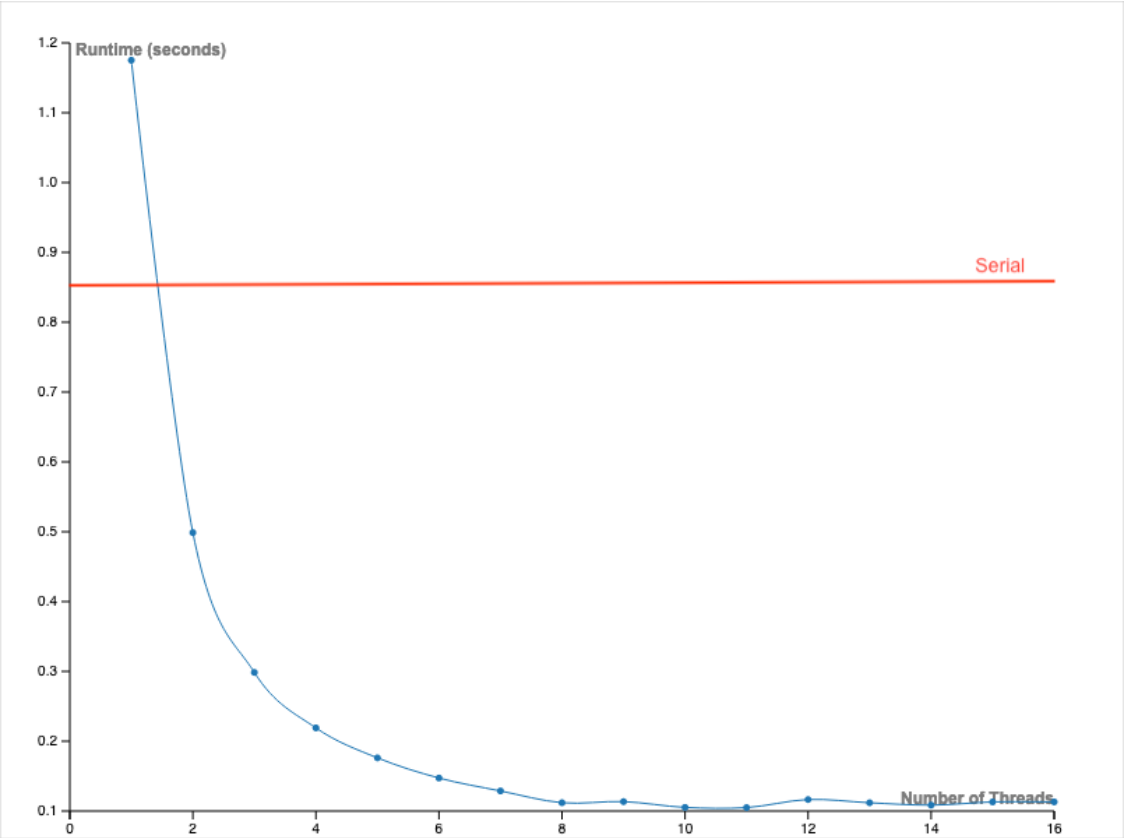

Figure 4: Runtime for Serial and different Parallel versions

5. Parallel loop dependencies with OpenMP [15 points]

OpenMP is commonly employed to parallelize loops for accelerating computational tasks. However, it is crucial to comprehend how loop dependencies can affect the execution time and correctness of the code. The scope of this exercise focuses on parallelizing a numerical calculation that entails recursion.

5.1. Pragma parallel for in case of loop dependencies

The sequential version of the code computes an array *opt* based on a constant *up* and an initial value *Sn*. The *opt* array is filled using a recursive relation that depends on the previous state of *Sn*, which is then multiplied by *up*. However, parallelizing this loop with OpenMP could potentially introduce errors due to the dependencies across iterations. In the parallel version, pragma directives are employed to handle these dependencies carefully.

```
  double Sn_local = Sn;
  int k = 0, n = 0;
#pragma omp parallel for default(none) firstprivate(Sn_local, k)
    lastprivate(Sn) shared(n, opt, up, N) schedule(guided, 150000)
  for (n = 0; n <= N; ++n)
  {
    if (k == 0 || k != n) {
      opt[n] = Sn_local * pow(up, n);
      Sn = opt[n] * up;
    } else {
      opt[n] = Sn;
      Sn *= up;
    }
  }
```

5.2. Comparison with different scheduling

The parallel code was run using different scheduling strategies for comparison. The OpenMP schedule clause was altered between "dynamic", "static", "auto", and "guided". The runtimes for each are as described in Table 2.

In the comparison of different scheduling strategies, it's noteworthy that the parallelized code works independently of the OpenMP schedule pragma used, in accordance with the guidelines. This was achieved through utilization of the 'firstprivate' and 'lastprivate' clauses. Specifically, 'firstprivate' ensured that the initial values of 'Sn_local' and 'k' were appropriately initialized for each thread, and 'lastprivate' made sure that the 'Sn' value from the last loop iteration was transferred back to the global variable.

Scheduling Strategy	Runtime (seconds)
Dynamic	3.442832
Static	3.246387
Auto	3.305259
Guided	3.538744
Sequential	4.205177

Table 2: Runtime for different Scheduling Strategy

In terms of performance, each parallel configuration was effective in reducing the runtime compared to the sequential version. The static scheduling strategy exhibited the lowest runtime among the parallel scheduling options.