Università della Svizzera italiana | Institute of Computing CI

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

Student: Paolo Deidda Discussed with: –

Solution for Project 2

This project will introduce you to parallel programming using OpenMP.

Contents

1. Parallel reduction operations using OpenMP *(20 Points)*

Dot product

I compiled `Skeleton_codes/dotProduct/dotProduct.cpp` with `g++ -fopenmp` for each $N \in \{10^5, 10^6, 10^7, 10^8\}$ and ran both OpenMP variants.

```cpp
  // Parallel dot product using reduction pragma
  time_start = wall_time();
  for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
    long double alpha_tmp = 0.0;
#pragma omp parallel for reduction(+ : alpha_tmp)
    for (int i = 0; i < N; i++) {
      alpha_tmp += a[i] * b[i];
    }
    alpha_reduction = alpha_tmp;
  }
  time_red = wall_time() - time_start;
```

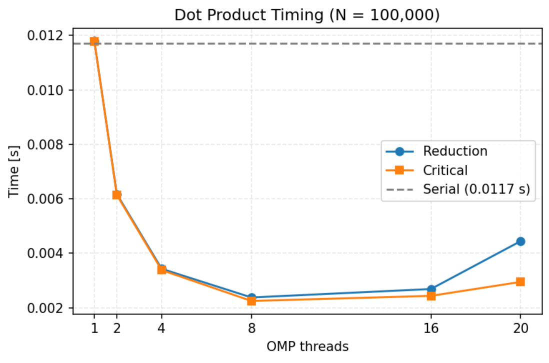
Listing 1: Reduction-based OpenMP dot product

Listing 1 lets OpenMP handle partial sums via the `reduction` clause, avoiding explicit synchronisation while the alternative keeps thread-local buffers and merges them inside a `critical` region, shown in Listing 2.

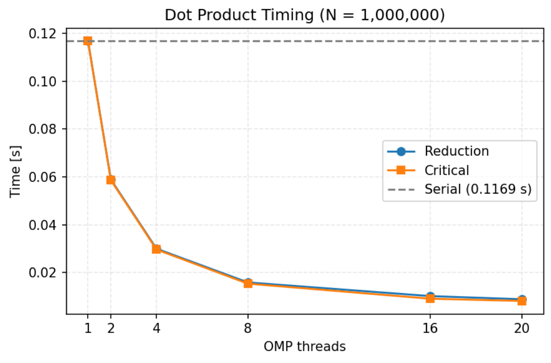
```cpp
  // Parallel dot product using critical pragma
  time_start = wall_time();
  for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
    long double alpha_tmp = 0.0;
#pragma omp parallel
    {
      long double thread_partial = 0.0;
#pragma omp for
      for (int i = 0; i < N; i++) {
        thread_partial += a[i] * b[i];
      }
#pragma omp critical
      { alpha_tmp += thread_partial; }
    }
    alpha_critical = alpha_tmp;
  }
```

Listing 2: Critical-based OpenMP dot product

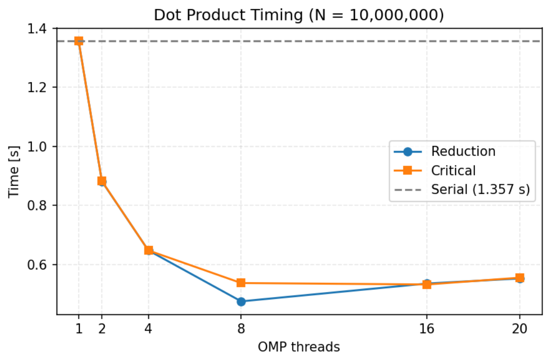
Figures 1a–1d illustrate that even the smallest vector size ($N = 10^5$) benefits from parallel execution: using two threads reduces the runtime from 11.7 ms to 6.2 ms, while eight threads bring it down further to 2.4 ms. The efficiency plot in Figure 2 reveals how OpenMP overhead increases with the number of threads. Efficiency, defined as $E = T_1/(p \cdot T_p)$, decreases whenever speedup grows sub-linearly; for example, a speedup of about 2 on two threads corresponds to $E \approx 1$, whereas a speedup of approximately 4.9 on eight threads results in $E \approx 0.6$. Beyond eight threads, the serial fraction and the cost of `critical` sections dominate, causing efficiencies to drop below 0.3, although the absolute runtime continues to improve (e.g., $T_{16} = 0.54\,\text{s}$ for $N = 10^7$). The reduction-based variant consistently achieves 5–10% higher efficiency because it avoids serialized updates. Overall, multithreading is advantageous when the number of threads remains moderate (up to eight) to limit overhead for smaller vectors; for $N \geq 10^7$, the reduction variant scales well up to sixteen threads.



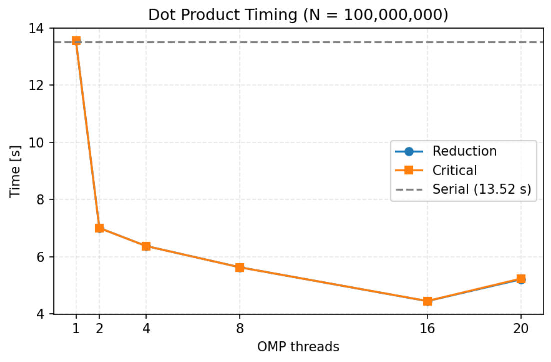
(a) $N = 10^5$



(b) $N = 10^6$



(c) $N = 10^7$



(d) $N = 10^8$

Figure 1: Execution time vs. threads for different vector sizes.

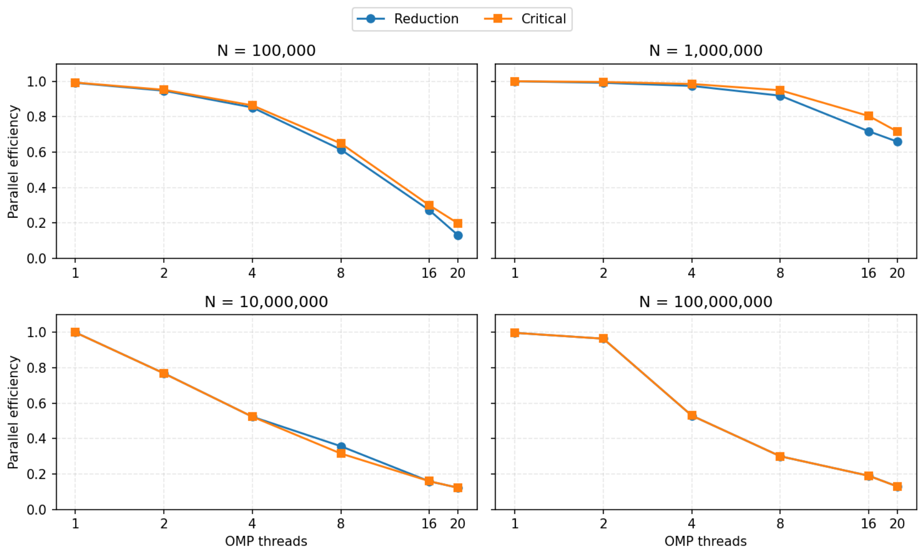


Figure 2: Parallel efficiency $E = T_1/(p \cdot T_p)$ for the two OpenMP variants.

Approximating π

I keep $N = 10^{10}$ as mandated. The serial loop in Listing 3 walks through all indices with a single accumulator, while the OpenMP variant in Listing 4 retains the same structure but distributes iterations across threads using a reduction clause.

```cpp
/* Serial version of computing pi */
sum = 0.0;
time = omp_get_wtime();
for (long long i = 0; i < N; ++i) {
  double xi = (static_cast<double>(i) + 0.5) * dx;
  sum += 4.0 / (1.0 + xi * xi);
}
pi = sum * dx;
```

Listing 3: Serial midpoint integration for π

```cpp
/* Parallel version of computing pi */
time = omp_get_wtime();
sum = 0.;
#pragma omp parallel for reduction(+ : sum)
for (long long i = 0; i < N; ++i) {
  double xi = (static_cast<double>(i) + 0.5) * dx;
  sum += 4.0 / (1.0 + xi * xi);
}
pi = sum * dx;
```

Listing 4: Parallel midpoint integration with OpenMP reduction

The `reduction(+: sum)` clause is the natural choice here: it is an embarrassingly parallel loop, so each thread can integrate its chunk independently and OpenMP combines the private partial integrals at the end. This avoids serial bottlenecks (e.g., critical sections) and preserves the numerical result for the fixed $N = 10^{10}$.

threads	serial	parallel	speedup
1	53.93082	53.93062	1.0000
2	53.94023	26.97132	1.9999
4	53.93210	13.48481	3.9995
8	53.93091	6.74307	7.9980

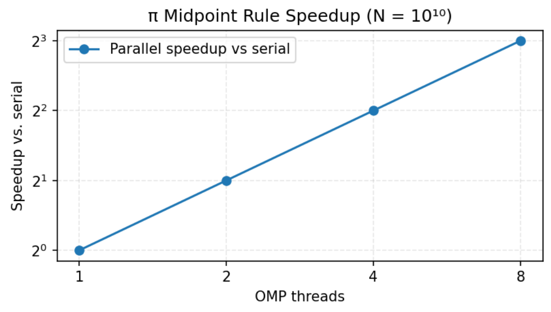


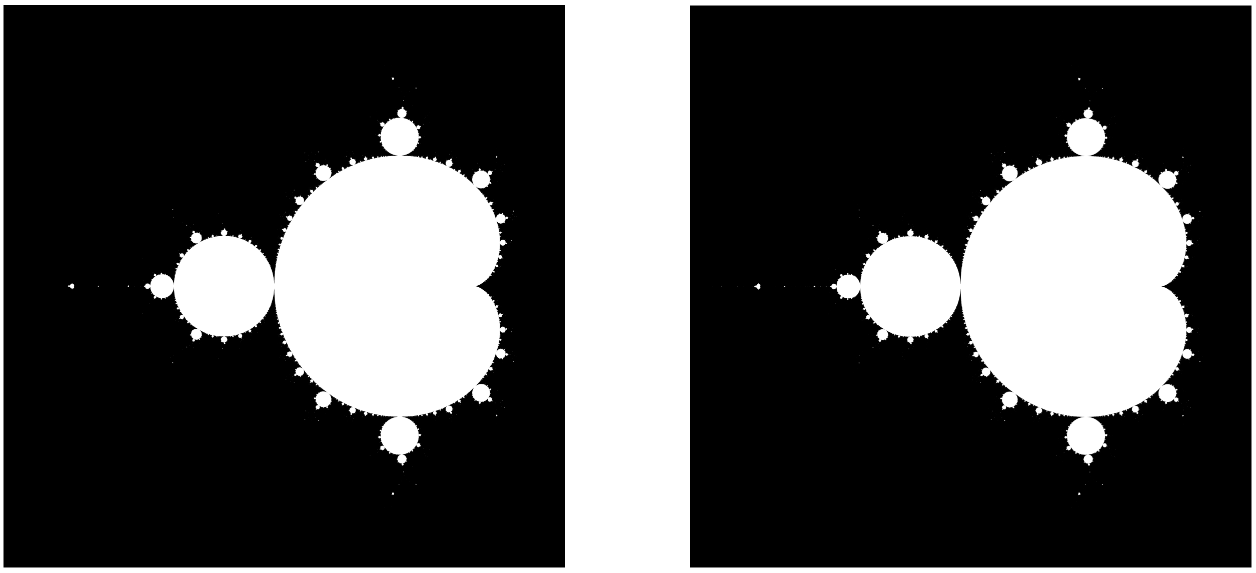
Figure 3: Timings and speedup for the midpoint π approximation with $N = 10^{10}$ (left) parallel speedup vs. serial baseline (right).

The speedup, defined as $S_p = T_1/T_p$, measures how much faster the fixed workload executes as the number of threads increases. As shown in Table 3, every time the number of threads doubles, the runtime is nearly halved, indicating excellent strong scaling. Consequently, the parallel efficiency $E_p = S_p/p$ remains close to one; for example, with eight threads, $E_8 \approx 0.9997$. Since the loop is embarrassingly parallel and the OpenMP reduction clause incurs negligible overhead, the main costs are limited to per-iteration arithmetic and loop control, both of which scale well with increasing thread count. This demonstrates that midpoint integration benefits directly from additional cores without sacrificing accuracy for the prescribed $N = 10^{10}$. An explicit efficiency plot is unnecessary in this case, as the near-ideal speedup already reflects the strong relationship between scaling and efficiency.

2. The Mandelbrot set using OpenMP *(25 Points)*

For this task, I developed two executables. The first, a serial program named `mandel_seq.c`, iterates over every pixel in the complex plane, applying the iteration $z \leftarrow z^2 + c$ until either the magnitude of z exceeds 2 or the maximum number of iterations, `MAX_ITERS`, is reached. Each pixel's iteration count is stored in a temporary buffer, and a global counter `nTotalIterationsCount` accumulates the total iterations. The second executable, `mandel_par.c`, retains the same computational kernel but parallelizes the pixel loop using `#pragma omp parallel for`. It uses private variables for temporary computations and applies a `reduction(+ : nTotalIterationsCount)` clause to safely sum the iteration counts across threads. To avoid race conditions during image writing, the PNG is generated only after the parallel region completes. A helper script, `run_mandel.sh`, automates recompilation of both versions for four image resolutions (1024^2, 2048^2, 4096^2, and 8192^2), runs the serial executable once per size, and then executes the OpenMP version with varying thread counts `OMP_NUM_THREADS` set to $\{1, 2, 4, 8, 16, 20\}$. Each run saves a log file in `Skeleton_codes/mandel/logs/` and the corresponding PNG image in `Skeleton_codes/mandel/png/`.

Example outputs for the two largest resolutions are shown in Figure 4.


(a) 4096×4096, 20 threads

(b) 8192×8192, 20 threads

Figure 4: Mandelbrot renderings generated by the OpenMP version.

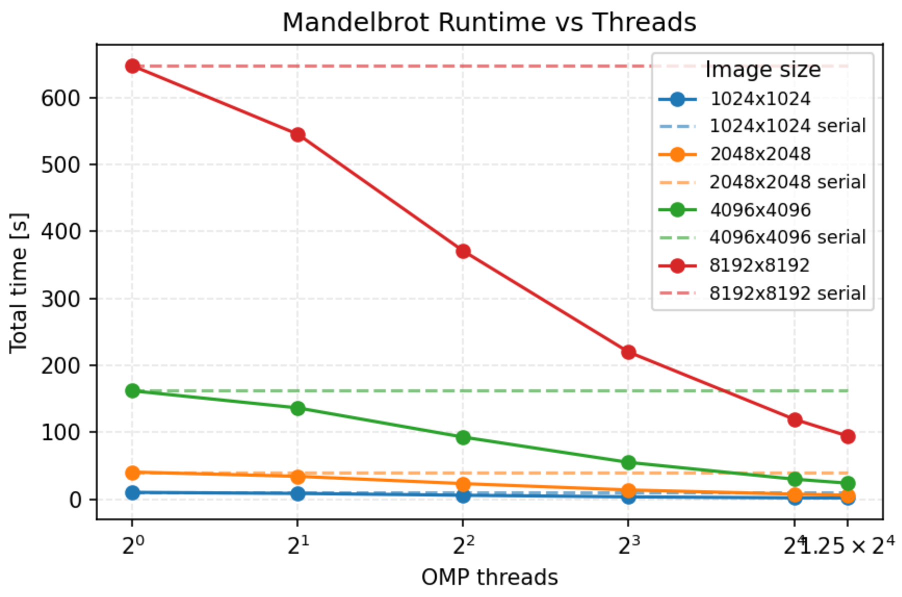


Figure 5: Total runtime versus thread count.

Table 1 summarises the measured times (seconds) for the OpenMP binary; the serial baseline corresponds to the entry with one thread.

Image size	Sequential	2 threads	4 threads	8 threads	16 threads	20 threads
1024×1024	10.13	8.54	5.80	3.44	1.88	1.80
2048×2048	40.48	34.09	23.18	13.74	7.49	5.91
4096×4096	161.92	136.35	92.72	54.93	29.89	23.93
8192×8192	647.71	545.44	370.92	219.81	119.51	94.57

Table 1: Total runtime for the OpenMP version; the iteration count remains constant across threads for each resolution.

Looking at the curves in Figure 5, we see that using 20 threads gives us about a 5.6× speedup for the smallest image, and around 6.8× for larger ones. The dashed baselines line up with the measured points when using one thread, so it shows that the OpenMP setup doesn't add any extra overhead outside of the parallel region. For smaller images, the scaling levels off sooner since there's less work per pixel, and the threads end up competing for memory bandwidth. But when we hit the 8192^2 resolution, the workload for each thread is bigger, and the speedup stays pretty close to linear. In all cases, the iteration counter and the number of pixels logged match what we expected, which confirms that both the numerical kernel and the performance stats we were asked for are valid.

3. Bug Hunt *(15 Points)*

Bug 1 – omp_bug1.c

The code fails to compile because the directive `#pragma omp parallel for` is followed by a brace instead of the loop itself. After fixing the syntax, the call to `omp_get_thread_num()` inside the loop creates unnecessary overhead. **Fix:** I moved the call outside the loop and correctly nest the `for` directive:

```
#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];
}
```

Bug 2 – omp_bug2.c

The variable `total` is shared by default and updated by all threads, causing a race condition. And also printing `tid` and `total` outside the parallel region uses invalid data. **Fix:** either use a reduction for a global total or declare variables as private:

```
#pragma omp parallel for reduction(+:total) schedule(dynamic,10)
for (i=0; i<1000000; i++) total += i * 1.0;
```

Bug 3 – omp_bug3.c

A runtime deadlock occurs because a `#pragma omp barrier` inside `print_results()` is reached only by threads executing the `sections`. Threads without a section never reach the barrier. Sometimes the code terminates, sometimes it hangs. **Fix:** remove the internal barrier and keep synchronization outside.

Bug 4 – omp_bug4.c

Declaring the large matrix `a` as `private` makes each thread allocate its own copy on the stack (∼8.8 MB per thread) and this lead to a segmentation fault at every single run. Also, even if it runs, the main matrix stays unmodified after the parallel region. **Fix:** make `a` shared or allocate it dynamically on the heap:

```
#pragma omp parallel shared(a,nthreads) private(i,j,tid)
```

Bug 5 – omp_bug5.c

Two threads acquire locks in opposite order (`locka` then `lockb`, and vice-versa), producing an occasional deadlock. **Fix:** enforce a consistent locking order or replace explicit locks with OpenMP critical sections:

```
#pragma omp critical
{ /* safe update of a[] and b[] */ }
```

4. Parallel histogram calculation using OpenMP *(15 Points)*

I looked into various synchronization strategies with OpenMP, creating three parallel versions of the histogram computation. There's the `atomic` version, a `Merge Critical` version that merges everything within the parallel region, and then there's the `Merge Out` version, which handles merging sequentially outside that region.

Implementation

```
// with private histograms for each thread ===================
time_start_critical = walltime();
#pragma omp parallel
{
  long dist_private[BINS];
  for (int i = 0; i < BINS; ++i) {
    dist_private[i] = 0;
  }

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

  #pragma omp critical
  {
    for (int i = 0; i < BINS; ++i) {
      dist_crit[i] += dist_private[i];
    }
  }
}
```

Listing 5: Histogram with private histograms and in-region merge (Merge Critical)

```
// with private histograms and merge outside ================
time_start_mergeOut = walltime();

int num_threads = omp_get_max_threads();
std::vector<std::vector<long>> all_private_dists(num_threads, std::vector<long>(BINS, 0));

#pragma omp parallel
{
  int tid = omp_get_thread_num();
  std::vector<long>& dist_private = all_private_dists[tid];
  for (int i = 0; i < BINS; ++i) {
    dist_private[i] = 0;
  }

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

// merge outside serially
for (int i = 0; i < BINS; ++i) {
  dist_mergeOut[i] = 0;
  for (int t = 0; t < num_threads; ++t) {
    dist_mergeOut[i] += all_private_dists[t][i];
  }
}
```

Listing 6: Histogram with private histograms and external merge (Merge Out)

```
// with atomic updates =======================================
time_start_atomic = walltime();

#pragma omp parallel for
for (long i = 0; i < VEC_SIZE; ++i) {
  #pragma omp atomic
  dist_atomic[vec[i]]++;
}
```

Listing 7: Histogram with atomic updates

The `Merge Critical` version was created to allow a thread that finishes its task early to begin merging its partial results right away, even if the other threads are still working. On the flip side, in the `Merge Out` version, all the threads finish their heavy computation first before they go through a single merge process in sequence.

Performance analysis

Figure 6 shows the impressive scaling results we got from the cluster. Up to about 16–20 physical cores, both versions scale pretty well, but the `Merge Out` version actually does a bit better overall. This seems to indicate that steering clear of synchronization during the parallel section leads to better efficiency, even if the merge step itself is all done sequentially at the end. The anticipated benefit of merging within the region earlier just didn't pan out, probably because the costs of locking and competing for cache space outweighed any potential gains.

What's surprising is that when we bumped up the thread count from 8 to 16 in the `Merge Critical` version, the runtime actually got worse, which doesn't really make sense given that 20 cores were available. This is likely due to increased contention over the critical section lock along with the overheads from OpenMP's runtime scheduling and synchronization.

It's also worth mentioning that the `atomic` version didn't perform well at any thread count, even falling short of the sequential baseline. This is mainly because of heavy contention and cache-line conflicts since each atomic increment pretty much serializes access to the shared histogram array.

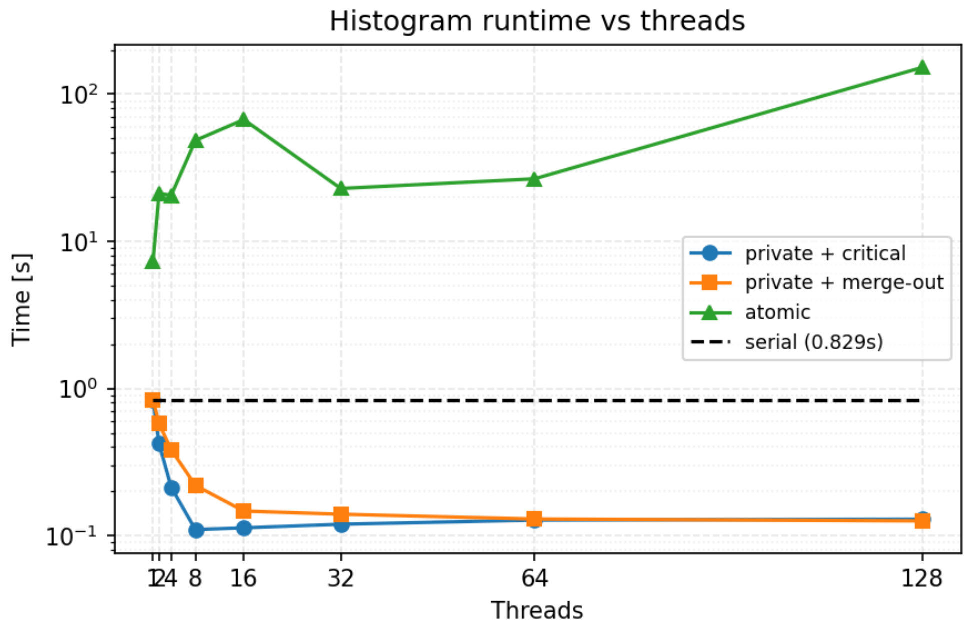


Figure 6: Strong scaling of the three OpenMP histogram versions.

The findings highlight that cutting down on synchronization in the parallel section is crucial for improving performance. Even though both merging methods work, the external merge (`Merge Out`) shows better scalability, whereas the `atomic` method isn't very efficient for this task.

5. Parallel loop dependencies with OpenMP *(15 Points)*

In the first parallel version (Listing 8), we break the index range into separate chunks. Each thread starts off by calculating the initial value with a single `pow(up, start)` call and then moves on to do local multiplications. This approach uses `firstprivate(Sn)` to make sure that each thread is working with different indices.

The second version (Listing 9) checks for gaps in the assigned blocks by looking at consecutive indices. This makes it **independent** from the scheduling decision as. Threads **only** recalculate `Sn` with `pow(up, i)` when it's really needed.

```
  // ----------- Chunk Division (Manual) -----------
  double time_start_chunk = walltime();
  #pragma omp parallel firstprivate(Sn)
  {
    int num_threads = omp_get_max_threads();
    int tid = omp_get_thread_num();
    int chunk_size = (N + 1) / num_threads;
    int start = tid * chunk_size;
    int end = (tid == num_threads - 1) ? (N + 1) : ((tid + 1) * chunk_size);
    double Sn_local = Sn * pow(up, start);

    for (int n = start; n < end; ++n) {
      opt_chunk[n] = Sn_local;
      Sn_local *= up;
    }
  }
```

Listing 8: Parallel recurrence with chunk partitioning

```
  // ----------- Schedule with Check -----------
 double time_start_check = walltime();
#pragma omp parallel firstprivate(Sn)
{
  double Sn_local = Sn;
  int last_i = -1; //  do NOT recalculate for thread that starts with i=0
  #pragma omp for
  for (int i = 0; i <= N; ++i) {
    if (i != last_i + 1) {
      Sn_local = Sn * pow(up, i);
    }
    opt_dyn[i] = Sn_local;
    Sn_local *= up;
    last_i = i;
  }
}
```

Listing 9: Parallel recurrence with on-demand recomputation

The scaling study (Figure 7) shows that the chunk-based version achieves nearly linear speedup up to 16 threads but then starts to plateau. In contrast, the version with the discontinuity check is slower due to overhead of constant check but it appears to maintaint linearity from 16 to 20 threads on the contrary of the first one. It would be really interesing to try how it goes on on a machine with more cores.

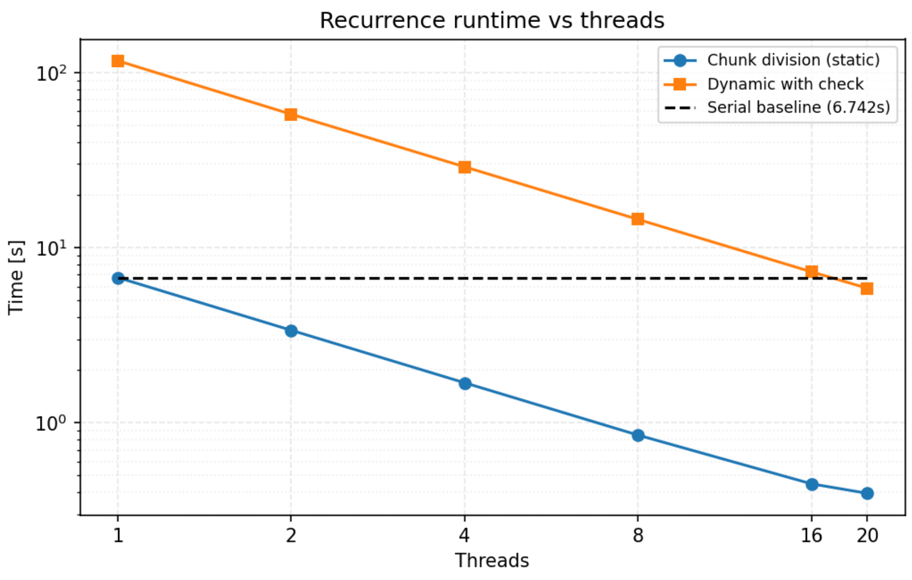


Figure 7: Runtime per thread count for the chunk and dynamic variants, compared with the serial baseline.

6. Quality of the Report *(15 Points)*