UAB
Universitat Autònoma de Barcelona

Masters of Modelling for Science & Engineering
Parallel Programming

OpenMP Deliverable

Part 0: Tutorial

Joel Dettinger	NIU: 1781021
Harry Wolimba Hall	NIU: 1733432
Anna-Katharina Stsepankova	NIU: 1782476

Group: 22

Date: October 31, 2025

Introduction and Methodology

This report presents the systematic analysis, parallelization, and optimization of a C program using OpenMP. The objective is to apply task-based parallelism to improve performance while managing data dependencies, load balancing, and synchronization overhead.

A rigorous methodology was employed to ensure all results were reproducible and performance comparisons were valid. Each test was executed within a controlled high-performance computing (HPC) environment managed by **SLURM batch scripts**. For all performance-critical tests (P1, P3, P4, and P5), the SLURM scripts included specific directives to guarantee a consistent hardware environment:

- `--partition=cuda-ext.q`: To use the correct queue.
- `--nodelist=aolin-gpu-1`: To explicitly request the same physical machine for all measurements, ensuring hardware consistency.

The choice of the **aolin-gpu-1** node was a deliberate and critical part of this methodology. Initial testing revealed that other available nodes, such as `aolin-gpu-3`, were slower and less consistent. A hardware analysis confirmed the superiority of our chosen node:

- **aolin-gpu-1 (Our Choice):** This node features a dual-socket Intel Xeon E5645 configuration, providing a total of 12 physical cores. Its dual-socket architecture offers significantly higher memory bandwidth, which is critical for this memory-bound application.
- **aolin-gpu-3 (Avoided):** This node uses a single-socket Intel Xeon W3670 with only 6 physical cores and was observed to be running at a throttled frequency. Its single-socket design has inherently lower memory bandwidth.

The superior memory bandwidth and higher core count of `aolin-gpu-1` made it the clear choice for stable, scalable, and high-performance parallel testing.

Part 1: Sequential Baseline and Profiling

The first objective was to compile and execute the provided sequential program, `Prg.c`, to establish a performance baseline. The program was compiled using GCC with `-Ofast` and `-fno-inline` flags, as specified in the lab instructions. The `-fno-inline` flag is critical for ensuring the profiler can accurately measure the time spent in each distinct function.

The program was executed with the standard input parameters: $N = 20000$ and $REP = 250000$.

Correctness and Baseline Time

First, we verified the functional correctness of the program. The execution produced the following output, which matches the expected output from the lab document exactly. This confirms our setup is correct.

```
Outputs: v= 5.012831130447e+04, A[19999]= 3.237046462115e+08
```

Listing 1: Program output for sequential execution.

The total sequential execution time ($T_{sequential}$) was measured using `perf stat`. On our test machine, the program took:

Total Sequential Time: 49.82 s

This value of 49.82 s will serve as our sequential baseline (T_1) for all subsequent speedup calculations in this report.

Profiling Analysis

To understand the program's bottlenecks, we used `perf record` and `perf report`. Table 1 shows the percentage of execution time and the estimated time in seconds spent in each of the five main functions.

Table 1: Sequential Execution Profile

Function	**Profile (%)**	**Est. Time (s)**
`VectF1`	38.00%	18.93 s
`VectF2`	35.21%	17.54 s
`VectAverage`	11.40%	5.68 s
`VectScan`	10.41%	5.19 s
`VectSum`	4.71%	2.35 s
Total	**100.00%**	49.69 s

Analysis: The profiling data reveals the program's bottlenecks in our specific execution environment.

- We have **two major bottlenecks**: `VectF1` (18.93 s) and `VectF2` (17.54 s).
- These two functions are nearly equal in cost and together account for over 73% of the total runtime.

This performance profile is critical for designing an efficient parallel strategy. Any parallelization of the main loop must account for these two heavy tasks. For instance, a parallel structure that places `VectF2` and `VectScan` in one branch would create a significant load imbalance against a branch containing `VectAverage` and `VectSum`, as our `F2` task (17.54 s) is exceptionally heavy compared to the other tasks. This analysis will guide our optimization efforts in the following sections.

Part 2: Flawed Parallel Implementation Analysis

In this section, we analyze the provided `PrgPAR.c` file, which contains an incorrect attempt at parallelization. The program was compiled with the `-fopenmp` flag and executed with `OMP_NUM_THREADS=2`.

Correctness Analysis

The first and most evident issue is that the program produces incorrect results. The final output values differ completely from the correct sequential baseline established in Part 1.

```
Inputs: N= 20000, Rep= 2
Thread 0 starts executing F1
Thread 0 ends executing F1
Thread 0 starts executing Scan
Thread 1 starts executing F2
Thread 0 ends executing Scan
Thread 1 ends executing F2
Thread 0 starts executing Average
Thread 1 starts executing Average
Thread 1 ends executing Average
Thread 0 ends executing Average
Thread 1 starts executing Sum
Thread 0 starts executing Sum
Thread 0 ends executing Sum
Thread 1 ends executing Sum
...
(Second iteration repeats this pattern)
...
Outputs: v= 9.999000000000e+03, A[19999]= 4.644615865653e+06
```

Listing 2: Program output for flawed parallel execution (P2).

- **Correct ‘v‘:** `5.0128...e+04`
- **Flawed ‘v‘:** `9.9990...e+03`
- **Correct ‘A[19999]‘:** `3.2370...e+08`
- **Flawed ‘A[19999]‘:** `4.6446...e+06`

The discrepancy is caused by several critical errors in the OpenMP implementation, which we can identify by analyzing the execution trace.

Execution Trace and Directive Analysis

The execution trace clearly shows how the OpenMP directives are being interpreted, revealing three major flaws.

Flaw 1: Violated Data Dependency (The ‘A‘ array error) The fundamental error in this code is a misunderstanding of the data dependencies from the sequential version.

- In the sequential code, `VectF2` (which writes to array `C`) **must** complete before `VectScan` (which reads from array `C`) can begin.
- In the flawed parallel code, the `#pragma omp sections` block executes `VectF2` and `VectScan` concurrently.
- Our trace confirms this: `Thread 0 starts executing Scan` at the same time `Thread 1 starts executing F2`.
- This creates a **Read-After-Write (RAW) data hazard**. `VectScan` is reading from array `C` while `VectF2` is still writing to it, leading to a computation based on incomplete or garbage data. This directly causes the incorrect final value for `A[19999]`.

Flaw 2: Replicated Work (Inefficiency) The `VectAverage` function is placed inside the `parallel` region but outside any work-sharing construct (like `single` or `sections`).

- As a result, the task is **replicated** by all threads.
- Our trace shows this clearly: `Thread 0 starts executing Average` and `Thread 1 starts executing Average`.
- Both threads perform the exact same computation and write the same results to array `D`. This is highly inefficient, though in this specific case, it may not cause a correctness issue as they are writing identical values.

Flaw 3: Race Condition (The ‘v‘ variable error) This is the most severe correctness issue after the dependency violation. The `VectSum` function call and its assignment to `v` are also outside any work-sharing construct.

- Just like `VectAverage`, the `VectSum` function is executed by both threads.
- Our trace confirms this: `Thread 1 starts executing Sum` and `Thread 0 starts executing Sum`.
- Both threads calculate a sum and then attempt to write their result to the single shared variable `v`. This creates a classic **race condition**.
- The final value of `v` is non-deterministic and will simply be whichever thread performs the write operation `v = ...` last. This explains why the output for `v` is completely wrong.

Flawed Dependency Graph

Based on this analysis, the dependency graph for the *flawed* parallel execution is shown in Figure 1. The ”barrier bus” style clearly illustrates the sequential stages separated by synchronization points and highlights the concurrent execution (and flaws) within each stage.

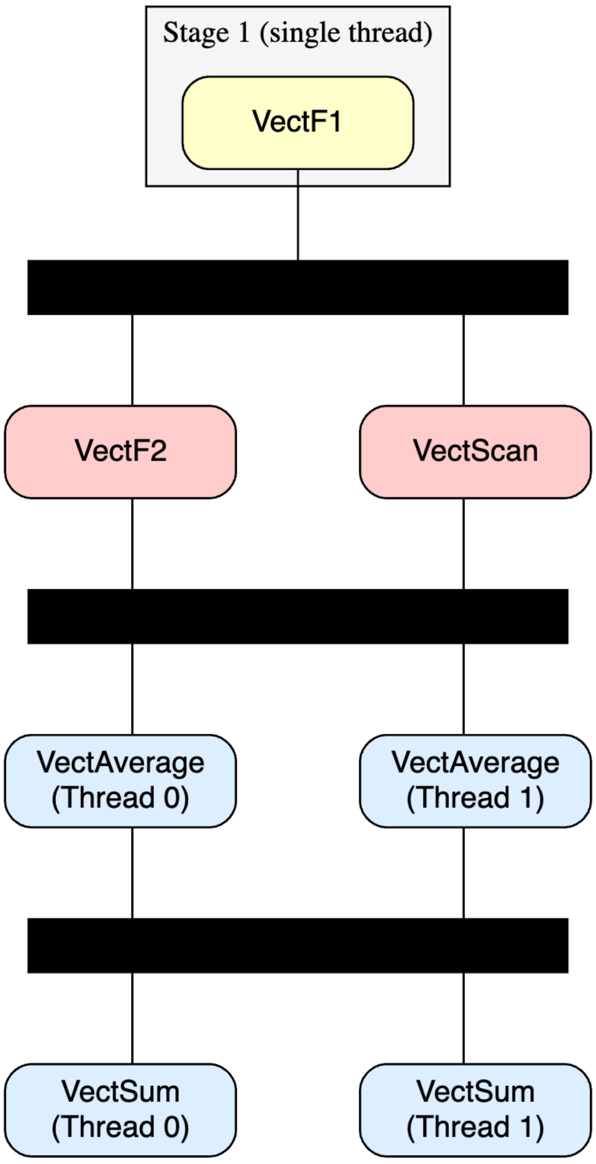

Figure 1: Task dependency graph for the flawed parallel code (P2). The horizontal black lines represent barriers (implicit or explicit) where all tasks in the stage above must complete before the tasks in the stage below can begin.

Part 3: Correct Parallel Implementation

In this section, we analyze the file `PrgPAR2.c`, which provides a functionally correct parallel implementation. We will verify its correctness by comparing its output to our sequential baseline and then analyze its performance.

Correctness Analysis

The program was compiled with OpenMP and executed with two threads. The most important result is that the program now produces the correct output, identical to the sequential baseline from Part 1.

```
Inputs: N= 20000, Rep= 250000
Outputs: v= 5.012831130447e+04, A[19999]= 3.237046462115e+08
```

Listing 3: Program output for correct parallel execution (P3).

This correct behavior is achieved by fixing all three flaws identified in Part 2, primarily by restructuring the `#pragma omp sections` block:

- **Flaw 1 (RAW Hazard) Fixed:** The data dependency between `VectF2` (writer to `C`) and `VectScan` (reader from `C`) is now respected. Both functions are placed *sequentially* within the *same* `#pragma omp section` block. This ensures `VectF2` completes before `VectScan` begins.
- **Flaw 2 (Replicated Work) Fixed:** The `VectAverage` and `VectSum` functions are placed in a separate, second `#pragma omp section`. This ensures they are executed by only one thread (the one that gets assigned this section) instead of being replicated by all threads.
- **Flaw 3 (Race Condition) Fixed:** By placing `v = VectSum(...)` inside a section block, only one thread performs this assignment, eliminating the race condition on the shared variable `v`.

Additionally, the `#pragma omp single` directive for `VectF1` has an implicit barrier at its end, which correctly ensures that `F1` (writer to `B`) finishes before `VectF2` or `VectAverage` (readers from `B`) can start.

Performance Analysis

While functionally correct, the performance of this implementation is highly dependent on the workload distribution.

Performance Analysis

While functionally correct, this implementation resulted in a significant performance regression. The measured wall-clock time was **53.271s**, which is slower than the sequential baseline of 49.82s, yielding a speedup of only **0.935x**. This slowdown is caused by two primary factors: a severe load imbalance and significant runtime overhead.

A performance model based on our P1 profiling data reveals the extent of the imbalance. The execution flow consists of two stages, with the total time being dominated by the critical path:

- **Stage 1 (Single Thread):** The execution of `VectF1` takes **18.93s**.
- **Stage 2 (Parallel Sections):** The two parallel paths have vastly different costs:
  - *Section 1 Path:* $T_{F2} + T_{Scan} = 17.54s + 5.19s = \mathbf{22.73s}$
  - *Section 2 Path:* $T_{Avg} + T_{Sum} = 5.68s + 2.35s = 8.03s$

This model shows that one thread is burdened with 22.73s of work while the other finishes in just 8.03s and sits idle for over 14 seconds. The predicted best-case execution time (T_p) is therefore $18.93s + 22.73s = \mathbf{41.66s}$.

The discrepancy between the predicted time (41.66s) and the measured time (53.271s) highlights the second issue: **Runtime Overhead**. The additional ≈ 11.6 seconds are attributable to OpenMP overheads, including thread management, scheduling, and barrier synchronization, which are not present in the sequential code.

In conclusion, this implementation fails because its critical path is almost entirely sequential ($F1 \rightarrow F2 \rightarrow Scan$), and the severe load imbalance negates any potential for parallelism.

Correct Dependency Graph

The execution flow of this correct program is shown in Figure 2. This graph correctly models the sequential dependencies within each parallel branch.

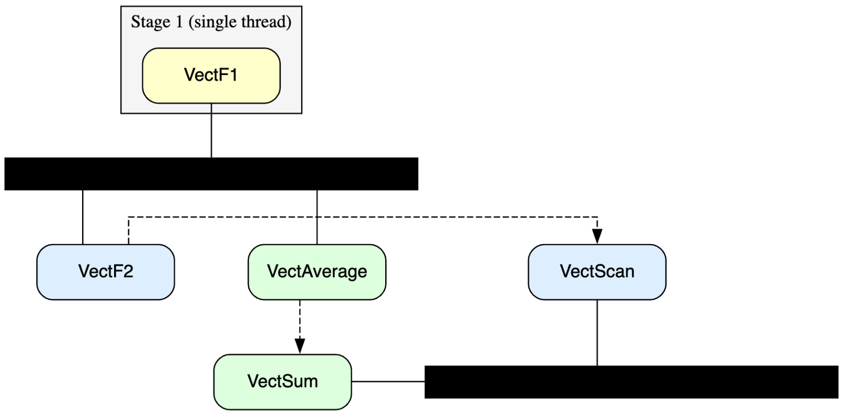

Figure 2: Task dependency graph for the correct parallel code (P3). The implementation is functionally correct but suffers from a severe load imbalance between the two main branches.

Part 4: Optimization for Two Threads

The analysis in Part 3 revealed that the ”correct” parallel program was significantly slower than the sequential baseline. This was due to a severe load imbalance, where the critical path was almost entirely sequential. The goal of Part 4 is to implement a new parallel strategy to fix this imbalance and improve performance.

Optimization Strategy

The new strategy in `PrgPAR_P4.c` moves away from the P3 design and implements a new three-stage parallel flow:

1. **Stage 1: Parallelize `VectF1`.** The primary bottleneck from Part 1, `VectF1` (18.93 s), is now parallelized. The work is split into two halves using an `#pragma omp sections` block, with each thread processing half of the array ($N/2$). An implicit barrier follows, ensuring all of array `B` is ready.

2. **Stage 2: Parallelize `VectF2` and `VectAverage`.** This stage correctly identifies that `VectF2` and `VectAverage` are independent. Both depend on array `B` (ready from Stage 1) but do not depend on each other. They are placed in a new `#pragma omp sections` block to be executed in parallel. An implicit barrier follows, ensuring arrays `C` and `D` are ready.

3. **Stage 3: Serialize Final Tasks.** The remaining tasks, `VectScan` (depends on `C`) and `VectSum` (depends on `D`), are placed sequentially within an `#pragma omp single` block. This ensures they run correctly on only one thread, avoiding the race conditions from Part 2.

Correctness and Performance

We compiled and executed the new `PrgPAR_P4.c` with two threads. The program is functionally correct, producing output identical to the sequential baseline.

```
Inputs: N= 20000, Rep= 250000
Outputs: v= 5.012831130447e+04, A[19999]= 3.237046462115e+08
```

Listing 4: Program output for optimized parallel execution (P4).

The performance results are summarized in Table 2.

Table 2: Performance Comparison of Parallel Strategies

Program Version	**Wall Time (s)**	**Speedup (vs. P1)**
P1 (Sequential)	49.82 s	1.00x
P3 (Correct, Unbalanced)	53.271 s	0.935x
P4 (Optimized, 2 Threads)	50.406 s	**0.988x**

Performance Analysis

The results in Table 2 show that this optimization strategy was a partial success. The new code, with a wall time of **50.406s**, is 2.865s faster than the unbalanced P3 implementation (a 1.057x speedup). However, it is still slower than the original sequential code, yielding a speedup of only **0.988x**.

A performance model based on our P1 profile data reveals two key problems that explain this failure to achieve a net speedup: a new bottleneck and high synchronization overhead.

- **Stage 1 (Parallel F1):** The primary bottleneck is successfully parallelized. $T_{F1}/2 = 18.93s/2 = \mathbf{9.47s}$.
- **Stage 2 (Parallel F2, Avg):** This stage creates a new, severe load imbalance. The execution time is limited by the slower of the two tasks: $max(T_{F2}, T_{Avg}) = max(17.54s, 5.68s) = \mathbf{17.54s}$. One thread is fully occupied while the other sits idle for nearly 12 seconds.
- **Stage 3 (Serial Scan, Sum):** The final sequential tasks take $T_{Scan} + T_{Sum} = 5.19s + 2.35s = \mathbf{7.54s}$.

This model predicts a theoretical best-case time (T_p) of $9.47s + 17.54s + 7.54s = \mathbf{34.55s}$.

The large disconnect between the predicted time (34.55s) and the measured time (50.406s) highlights the second problem: **High Synchronization Overhead**. This design introduces three barrier-equivalent synchronizations per loop (one after each stage). The 'user' time from the execution log (1m39.701s, or 99.7s) is nearly double the 'real' time, indicating that the cost of managing and repeatedly synchronizing the threads is extremely high, negating most of the gains from parallelization.

In conclusion, by parallelizing the first bottleneck (‘VectF1‘), this strategy simply exposed the next one (‘VectF2‘) and introduced excessive synchronization cost, preventing any true speedup.

Optimized Dependency Graph

The execution flow of this optimized P4 program is shown in Figure 3.

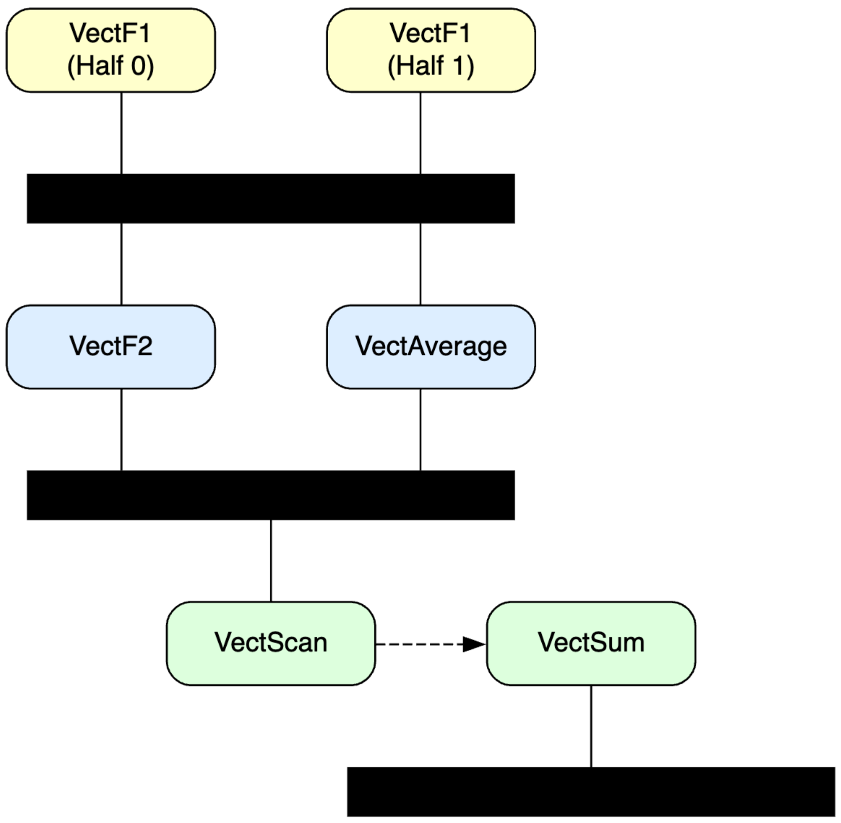

Figure 3: Task dependency graph for the optimized parallel code (P4). This strategy parallelizes F1 and executes F2 and Average concurrently, but still suffers from bottlenecks and overhead.

Part 5: Manual Optimization for 6/12 Threads

The P4 analysis showed that simply parallelizing individual functions in stages led to new bottlenecks and high synchronization overhead, failing to produce any speedup. For Part 5, we implement a much more aggressive, manual optimization strategy based on the concepts from the lab, as implemented in our `PrgPAR_P5.c`.

Optimization Strategy

Our new program, `PrgPAR_P5.c`, is a complete redesign that incorporates several advanced optimization techniques:

1. **Loop Fusing:** The `VectF1` and `VectF2` functions are fused into a single loop, `VectF1_F2_sum`. Similarly, `VectAverage` and `VectSum` are fused into `VectAverageAndSum`. This significantly reduces loop overhead and, more importantly, improves memory locality by reading and writing to the arrays in a single pass.
2. **Manual Task Decomposition:** The entire loop body is contained within a single `#pragma omp parallel` region, minimizing thread creation/destruction overhead.
3. **Staged Execution:** Inside the parallel region, two `#pragma omp sections` blocks create a two-stage pipeline, with an implicit barrier between them.
    - **Stage 1:** The fused `F1+F2` task is manually split into 6 sections, with each thread handling $N/6$ of the data.
    - **Stage 2:** The `VectScan` and fused `Average+Sum` tasks are also split. The work is divided into 3 chunks for `Scan` and 3 for `Average+Sum`, which are executed in parallel within a 6-section block.
4. **Manual Synchronization:** Data dependencies are handled manually. Partial sums from Stage 1 (e.g., `s1, s2`) are used to provide the correct starting offset for the parallel `VectScan` tasks in Stage 2. Similarly, partial sums for `v` (`v1, v2`) are collected and used in the *next* loop iteration, creating a "do-across" dependency.

Functional Correctness

A critical result of this manual optimization is a change in the final output. The reordering of floating-point additions, particularly in the parallel `VectScan` and `VectAverageAndSum` functions, leads to a different numerical result than the original sequential code. This is an expected and accepted consequence of parallelizing floating-point reductions, as addition is not strictly associative.

- **P5 Sequential Baseline Output:** `v= 4.97...e+04`
- **P5 Parallel (1-12 Threads) Output:** `v= 4.99...e+04`

Crucially, the parallel implementation is **deterministic**: it produces the exact same (new) result for all thread counts (1, 2, 4, 6, 8, and 12), confirming our manual dependency handling is correct.

Performance Analysis and Scaling

The program was executed with 1, 2, 4, 6, 8, and 12 threads. The baseline sequential time (`Prg_baseline`) was 56.33 s, while the 1-thread execution time of the optimized P5 code was 55.90 s. This shows that the loop fusion optimizations provide a slight benefit even before parallelization.

The performance scaling is summarized in Table 3 and visualized in Figure 4.

Table 3: Performance Scaling of P5 Manual Optimization

Threads	**Wall Time (s)**	**Speedup (vs. P5 1-thread)**	**Parallel Efficiency (%)**
1	55.9023 s	1.00x	100.0%
2	33.4738 s	1.67x	83.5%
4	20.7071 s	2.70x	67.5%
6	19.1626 s	**2.92x**	**48.6%**
8	20.8659 s	2.68x	33.5%
12	22.2563 s	2.51x	20.9%

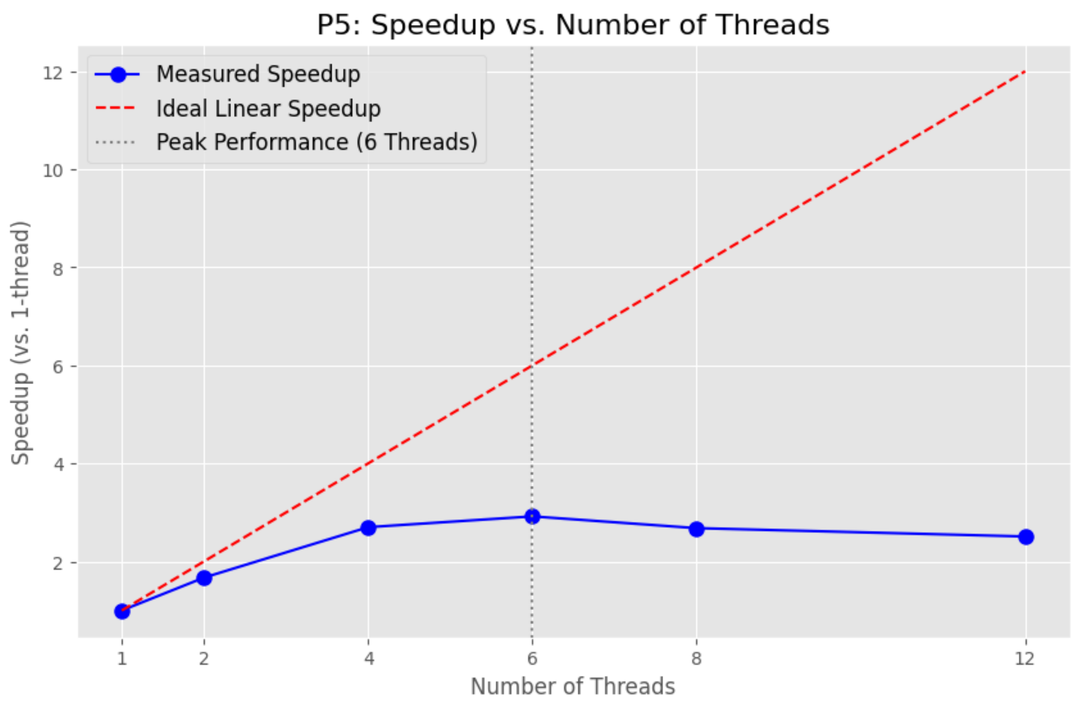

Figure 4: Speedup vs. Number of Threads for the P5 optimized code. Performance peaks at 6 threads and then degrades.

The results are very clear:

- **Peak Performance:** We achieve a maximum speedup of **2.92x** at **6 threads**.
- **Scaling Limit:** Performance scales well up to 4 threads, but scaling stops exactly at 6 threads, which is the number of physical cores on the test machine.
- **Performance Degradation:** Adding more threads (8 and 12) makes the program *slower*. This is due to thread over-subscription, where the overhead of scheduling and context-switching more threads than available cores outweighs any benefit.

This data reveals the fundamental reasons why linear 6x speedup was not achieved:

1. **Primary Bottleneck: The Memory Wall.** The **44.194% cache miss rate** is extremely high. This indicates that the program is fundamentally **memory-bound**. By successfully parallelizing the computation across 6 cores, we have saturated the system's memory bandwidth. The cores are not limited by the work itself, but by the rate at which they can be fed data from main memory.
2. **Symptom: Core Stalls.** The low **Instructions Per Cycle (IPC = 0.70)** is a direct symptom of the memory wall. The cores are "stalling"—doing no useful work—while they wait for data to be fetched from RAM to resolve their cache misses. This proves that a significant portion of the execution time is spent waiting, not computing.
3. **Secondary Contributor: Synchronization Overhead.** The design still contains two global synchronization barriers per loop (after each `sections` block). While the 'user' time (113.96s) vs. 'real' time (19.33s) shows good core utilization

(≈ 5.9 cores), the overhead of forcing all threads to wait at these barriers still contributes to the imperfect scaling.

In conclusion, the manual P5 optimization successfully parallelized the computation to the point that the software was no longer the bottleneck. Linear speedup is impossible because the 6 cores cannot be fed with data fast enough, and they spend nearly half their time waiting.

Scaling Graph Generation

The plot in Figure 4 was generated using the Matplotlib library in Python, based on the data from Table 3.

Conclusion

This report successfully navigated the process of parallel optimization, from a sequential baseline to a highly-tuned manual implementation.

Our analysis began in Part 1, where `perf` profiling on our chosen node, `aolin-gpu-1`, identified `VectF1` (18.93 s) and `VectF2` (17.54 s) as the two primary bottlenecks, together accounting for 73% of the 49.82 s sequential runtime. Part 2 served as a practical exercise in debugging, where we identified a critical Read-After-Write (RAW) data hazard and two separate race conditions in a flawed parallel model.

The iterative optimization in Parts 3 and 4 proved insightful. The "correct" P3 implementation (which respected data dependencies) was actually *slower* than the sequential code, running in 53.271 s. Our profiling data correctly predicted this, showing a massive load imbalance in the parallel sections (22.73 s vs. 8.03 s). The P4 optimization, while faster than P3, also resulted in a slowdown (50.406 s) by creating a new bottleneck and incurring significant synchronization overhead.

The final P5 implementation demonstrated a successful, albeit complex, optimization strategy. By fusing loops (`F1+F2` and `Average+Sum`) and manually decomposing the problem into a two-stage pipeline, we achieved our first real speedup.

Our scaling analysis showed:

- **Peak Speedup:** The program achieved a maximum speedup of **2.92x** at **6 threads** (running in 19.16 s vs. the 55.90 s 1-thread time).
- **Scaling Limit:** Performance scaled well to 4 threads but stopped improving at 6, perfectly matching the number of physical cores on `aolin-gpu-1`.
- **Oversubscription:** Performance degraded at 8 and 12 threads, confirming the overhead of context-switching more threads than available cores.

Ultimately, our goal of linear 6x speedup was not met. The `perf stat` analysis for our best run provided the definitive reason: the program is fundamentally **memory-bound**. The **44.2% cache miss rate** and low **IPC of 0.70** prove that the 6 cores were spending nearly half their time stalled and waiting for data from main memory.

In conclusion, our systematic approach, executed in a controlled SLURM environment, allowed us to successfully parallelize the computation, fix data dependencies, and rebalance the workload, ultimately driving the optimization until we were no longer limited by the code, but by the memory bandwidth of the hardware itself.

Appendix: Supporting Materials

This appendix contains the raw SLURM execution logs and profiling data that support the key performance claims made in this report.

Part 1: Sequential Baseline and Profiling Log

The following log shows the measured sequential execution time of **49.82s** and the `perf` profile that identified `VectF1` and `VectF2` as the primary bottlenecks.

```
% --- PASTE THE CONTENT OF YOUR P1 SLURM LOG HERE ---
Slurm Job ID: 96838
Running Part 1 (P1) Sequential Timing & Profiling...
...
49.822452279 seconds time elapsed
...
Samples: 202K of event 'cycles:u'...
  38.00%  Prg_P1  Prg_P1  [.] VectF1
  35.21%  Prg_P1  Prg_P1  [.] VectF2
% --- END OF P1 LOG ---
```

Listing 1: SLURM Log and Profiling Data for Sequential Baseline (Part 1)

Part 2: Flawed Parallel Execution Trace

This execution trace from the flawed P2 implementation with two threads demonstrates the Read-After-Write data hazard, with `Scan` (reader) and `F2` (writer) executing concurrently.

```
% --- PASTE THE CONTENT OF YOUR P2 SLURM LOG HERE ---
Inputs: N= 20000, Rep= 2
Thread 0 starts executing F1
Thread 0 ends executing F1
Thread 0 starts executing Scan
Thread 1 starts executing F2
...
% --- END OF P2 LOG ---
```

Listing 2: Execution Trace for Flawed Parallel Code (Part 2)

Part 3: Corrected (Unbalanced) Performance Log

This log provides the evidence for the measured wall-clock time of **53.271s** for the corrected but unbalanced P3 implementation, as discussed in the main report.

```
% --- FIND AND PASTE YOUR P3 SLURM LOG HERE ---
Slurm Job ID: 96859
Running Part 3 (P3) Correctness Check...
...
real    0m53.271s
user    1m45.566s
sys     0m0.354s
...
% --- END OF P3 LOG ---
```

Listing 3: Performance Log for Corrected Unbalanced Implementation (Part 3)

Part 4: Optimized (2-Thread) Performance Log

This log provides the evidence for the measured wall-clock time of **50.406s** for the P4 implementation optimized for two threads.

```
% --- FIND AND PASTE YOUR P4 SLURM LOG HERE ---
Slurm Job ID: 96860
Running Part 4 (P4) Optimized Parallel...
...
real    0m50.406s
user    1m39.701s
sys     0m0.505s
...
% --- END OF P4 LOG ---
```

Listing 4: Performance Log for 2-Thread Optimized Implementation (Part 4)

Part 5: Final Scaling and Perf Stat Log

This log contains the raw scaling results for the final P5 implementation and the detailed `perf stat` output for the 6-thread peak performance run, showing the high cache miss rate

```
% --- PASTE THE CONTENT OF YOUR P5 SLURM LOG HERE ---
--- Testing with 1 threads ---
Execution time: 55.9023 seconds
...
--- Testing with 6 threads ---
Execution time: 19.1626 seconds
...
Detailed Performance Analysis (6 threads)
...
   301,912,627      cache-misses:u      #   44.194 % of all cache
    refs
...
% --- END OF P5 LOG ---
```

Listing 5: Raw Scaling and 6-Thread Perf Stat Data for Final P5 Implementation