

**parallel Computing**

**M.Sc. Embedded Systems**



**Report Part 1**

**Group Members:**

**Oct 2025**

|  |  |  |
| --- | --- | --- |
| **Mehdi Moallemkolaei** | **153010947** | [**mehdi.moallemkolaei@tuni.fi**](mailto:mehdi.moallemkolaei@tuni.fi) |
| **Md Ashfak Haider Nehal** | **153045077** | [**mdashfakhaider.nehal@tuni.fi**](mailto:mdashfakhaider.nehal@tuni.fi) |

# **Part 1.**

Contents

[**Part 1.** 2](#_Toc212130721)

[**6.1. Benchmarking the original Code and Improving Performance via Compiler Settings** 3](#_Toc212130722)

[**6.1.1 & 6.1.2:** 3](#_Toc212130723)

[**6.1.3. Auto Vectorization Report:** 3](#_Toc212130724)

[**6.1.4. Best Flags:** 4](#_Toc212130725)

[**6.1.5. Each Flag’s Performance** 4](#_Toc212130726)

[**6.1.6. Code Breaker Flags:** 5](#_Toc212130727)

[**6.3. Multi-Thread Parallelization** 5](#_Toc212130728)

[**6.4. OpenMP Parallelization** 6](#_Toc212130729)

[**6.4.1. Average Frame times** 6](#_Toc212130730)

[**6.4.2. Extra code transformation** 6](#_Toc212130731)

[**6.4.3. Parallelized loops** 6](#_Toc212130732)

[**6.4.4. Parallelizations leading to slowdown** 6](#_Toc212130733)

[**6.4.5. Performance scale with the number of cores** 6](#_Toc212130734)

[**6.4.6. Machine and Environment** 6](#_Toc212130735)

[**6.4.7. Hours to complete Part 1** 6](#_Toc212130736)

## **6.1. Benchmarking the original Code and Improving Performance via Compiler Settings**

### **6.1.1 & 6.1.2:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Delays/Versions | Physics  (moving sat) | Graphics  (space coloring) | Total |
| 1 | **Original C – Debug Mode** | **~ 170 ms** | **~ 1257 ms** | **~ 1429 ms** |
|  | | | | **~ 1461 ms** |
| **~ 1407 ms** |
| **~ 1399 ms** |
| **~ 1433 ms** |
| **~ 1465 ms** |
| 2 | **Original C – Release Mode** | **~ 119 ms** | **~ 560 ms** | **~ 681 ms** |
|  | | | | **~ 676 ms** |
| **~ 693 ms** |
| **~ 694 ms** |
| **~ 670 ms** |
| **~ 684 ms** |

The Release build improves performance by ~52% overall versus Debug, mainly due to compiler optimizations.

First line in the table, shows the average over all frames after 10 first frames, and the other lines are sample frames randomly picked from 10 frames.

To run in the terminal (like for using seed):

.\out\build\x64-Release\parallel.exe

or

.\out\build\x64-Debug\parallel.exe

### **6.1.3. Auto Vectorization Report:**

Enabling the flags in the CMake text file:

* target\_compile\_options(parallel PRIVATE "/Qvec-report:2")

The “Debug mode” Log does not contain any vectorization report.

The “Release mode” Log only shows:

* info C5002: loop not vectorized due to reason 'xxxx'

So, it means that no vectorization is done.

It reports like “C5002: loop not vectorized due to reason '1106/1203/1300/…'”.

In our code, the typical blockers are:

* **1106:** data dependence, the compiler can’t disprove (loops read/modify the same arrays).
* **1300/1305:** loop too small or complex control flow (early break, reductions intertwined).
* **500:** unknown aliasing or pointer escape patterns (access through global arrays).

### **6.1.4. Best Flags:**

We want to test and compare 5 different scenarios:

|  |  |
| --- | --- |
| Scenarios | Flags |
| A. Base: | "/Qvec-report:2" "/O2" |
| B. + AVX2: | "/arch:AVX2" |
| C. + fp:precise: | "/fp:precise" |
| D. + fp:fast: | "/fp:fast" |
| E. + AVX2 + fp:fast | "/fp:fast" "/arch:AVX2" |

Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Scenarios | Physics  (moving sat) | Graphics  (space coloring) | Total |
| 0 | No Flag | **~ 121 ms** | **~ 546 ms** | **~ 669 ms** |
| A | **Base Flags** | **~ 117 ms** | **~ 548 ms** | **~ 665 ms** |
| B | **Base + AVX2** | **~ 118 ms** | **~ 550 ms** | **~ 670 ms** |
| C | **Base + fp:precise** | **~ 122 ms** | **~ 547 ms** | **~ 671 ms** |
| D | **Base + fp:fast (WOW!)** | **~ 77 ms** | **~ 325 ms** | **~ 404 ms 😏** |
| E | **Base + AVX2 + fp:fast** | **~ 81 ms** | **~ 369 ms** | **~ 453 ms** |

Scenario D seems to produce a better result than the other options.

- The major speedup comes from **/fp:fast** (scenario D).

- Adding **/arch:AVX2** does not help because the originalcode’s loop shapes aren’t SIMD-friendly.

### **6.1.5. Each Flag’s Performance**

**/O2**

* **What it does:**

It enables a set of speed-oriented passes, such as inlining, common subexpression elimination, loop invariant code motion, strength reduction, constant propagation, some loop unrolling, etc.

* **Why did it help here:**

lowers scalar overhead in both physics and graphics loops, especially useful for the many small expressions and array accesses.

**/fp:fast (fast math and the game-changer)**

* **What it does:**

Relaxes strict IEEE 754 semantics so the compiler can reassociate FP expressions, fuse mul+add, hoist/common-subexpressions, use reciprocal/rsqrt sequences, and otherwise pick faster math code paths.

* **Why did it help here:**

Some loops do lots of divides and sqrt/sqrtf. With fp:fast, the compiler can replace divisions by multiplies and use fast sqrt/rsqrt sequences and reuse them.

**/Qvec-report:2 (reporting only)**

* **What it does:**

Just prints auto vectorization decisions/reasons (those C5002 codes). Useful to explain “why not vectorized”.

**Why did it help here:**

It doesn’t speed up execution; it tells why the hot loops didn’t vectorize.

### **6.1.6. Code Breaker Flags:**

We used multiple random flags and they mostly were working properly. However, some flags like AVX2 not only were not helpful, but caused drawback in the benchmarks.

## **6.3. Multi-Thread Parallelization**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Loop | Legal? | Worth doing? | Reasoning |
| 1 | Physics iteration | No | — | Loop-carried time dependence  (step t feeds step t+1). |
| 2 | Physics satellite | Yes | Yes | Independent per-satellite within a time step. |
| 3 | Graphics pixel | Yes | Definitely! | Massive, independent work per pixel. Ideal scaling. |
| 4 | Graphics satellite | Yes | Almost No | The trip count is small. Expensive to thread. Prefer SIMD inside each pixel. |

**Which loops are allowed to be parallelized?**

* **Graphics pixel — Allowed:**

Each iteration writes to a distinct pixels[i]; reads are from shared, read-only satellite state. This loop is embarrassingly parallel and dominates runtime so is the biggest win with OpenMP.

* **Physics satellite — [Conditionally] allowed:**

Within a single time step, satellite i only reads/writes its own tmpPosition[i] and tmpVelocity[i], so iterations over i are independent (data-parallel).

* **Physics iteration — Not safe to parallelize across iterations:**

Each iteration uses the updated tmpPosition[i] / tmpVelocity[i] from the previous iteration (Euler time stepping). That’s a strict temporal dependency; running different steps concurrently would change physics.

* **Graphics satellite — Not useful to thread per pixel:**

Inside a single pixel, we loop over 64 satellites twice. It is technically parallelizable with reductions (for the weighted sums) and a min-reduction (for the closest satellite), but spawning threads at per-pixel granularity would explode overhead and cache thrash.

**Loops that are legal but don’t benefit? Why?**

The loop for copying tmpPosition and tmpVelocity in parallelPhysicsEngine and the single pixel rendering loop could be parallelized, but they will not benefit because they are simple and have less amount of data.

**Transformations enabling/boosting parallelism?**

Here are two code transformations that enable and improve parallelization:

**1.**

int idx;

#pragma omp parallel for private(idx)

for (idx = 0; idx < SATELLITE\_COUNT; ++idx) {

tmpPosition[idx].x = satellites[idx].position.x;

tmpPosition[idx].y = satellites[idx].position.y;

tmpVelocity[idx].x = satellites[idx].velocity.x;

tmpVelocity[idx].y = satellites[idx].velocity.y;

int idx2;

#pragma omp parallel for

for (idx2 = 0; idx2 < SATELLITE\_COUNT; ++idx2) {

satellites[idx2].position.x = tmpPosition[idx2].x;

satellites[idx2].position.y = tmpPosition[idx2].y;

satellites[idx2].velocity.x = tmpVelocity[idx2].x;

satellites[idx2].velocity.y = tmpVelocity[idx2].y;

}

**2.**

int i;

#pragma omp parallel for private(i)

for(i = 0 ;i < SIZE; ++i) { … }

## **6.4. OpenMP Parallelization**

### **6.4.1. Average Frame times**

Mode: OpenMP and "/O2" "/fp:fast" "/Qvec-report:2"

|  |  |  |  |
| --- | --- | --- | --- |
| Frame | Physics  (moving sat) | Graphics  (space coloring) | Total |
| 1 | ~ 77 ms | ~ 28 ms | ~ 106 ms |
| 2 | ~ 77 ms | ~ 28 ms | ~ 106 ms |
| 3 | ~ 77 ms | ~ 27 ms | ~ 105 ms |
| 4 | ~ 97 ms | ~ 44 ms | ~ 106 ms |
| 5 | ~ 77 ms | ~ 28 ms | ~ 106 ms |
| 6 | ~ 77 ms | ~ 27 ms | ~ 105 ms |
| 7 | ~ 80 ms | ~ 28 ms | ~ 110 ms |
| 8 | ~ 77 ms | ~ 29 ms | ~ 107 ms |
| 9 | ~ 81 ms | ~ 33 ms | ~ 116 ms |
| 10 | ~ 83 ms | ~ 28 ms | ~ 112 ms |
| AVG | **~ 78 ms** | **~ 28 ms** | **~ 107 ms** |

We just parallelized some loops along with the compiler parallelization

The OpenMP version reduced the total frametime to ~107 ms, mostly from parallelizing the Graphics loop.

### **6.4.2. Extra code transformation**

1. Switched to sqrtf where variables are float (graphics & init) to avoid implicit double promotions.
2. Kept the original math.
3. Used /fp:fast to allow contraction and math re-association that MSVC’s vectorizer benefits from.

### **6.4.3. Parallelized loops**

1. Physics satellite loop in parallelPhysicsEngine.

Why: trivially data-parallel and race-free. Practical speedup is tiny but harmless.

int idx;

#pragma omp parallel for private(idx)

for (idx = 0; idx < SATELLITE\_COUNT; ++idx) {…}

int idx2;

#pragma omp parallel for

for (idx2 = 0; idx2 < SATELLITE\_COUNT; ++idx2) {…}

1. Graphics pixel loop in parallelGraphicsEngine.

Pixels are independent; each iteration writes a distinct pixels[i].

int i;

#pragma omp parallel for private(i)

for(i = 0 ;i < SIZE; ++i) { … }

### **6.4.4. Parallelizations leading to slowdown**

We did not observe any breaking or slowdown.

### **6.4.5. Performance scale with the number of cores**

**Scaled well:** the Graphics pixel loop, because it is embarrassingly parallel; scaling flattens as memory bandwidth is saturated and OpenMP overheads become non-negligible.

**Limited scaling:** the Physics part is dominated by the sequential Physics iteration loop; the parallel copy loops barely register.

### **6.4.6. Machine and Environment**

We did the project on a computer at TC303.

### **6.4.7. Hours to complete Part 1**

We spent approximately a week finishing part 1.

~2 hr/day, total of around 15 hr.