

**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](#_Toc211505266)

[**6.1. Benchmarking the original Code and Improving Performance via Compiler Settings** 3](#_Toc211505267)

[**6.1.1 & 6.1.2:** 3](#_Toc211505268)

[**6.1.3. Auto Vectorization Report:** 3](#_Toc211505269)

[**6.1.4. Best Flags:** 4](#_Toc211505270)

[**6.1.5. Each Flag’s Performance** 4](#_Toc211505271)

[**6.1.6. Code Breaker Flags:** 5](#_Toc211505272)

[**6.3. Multi-Thread Parallelization** 6](#_Toc211505273)

[**6.4. OpenMP Parallelization** 7](#_Toc211505274)

[**6.4.1. & 6.4.2. Average Frame times and code transformation** 7](#_Toc211505275)

[**6.4.3. Parallelized loops** 7](#_Toc211505276)

[**6.4.4. Performance scale with multi-cores** 7](#_Toc211505277)

[**6.4.5. Bonus Question** 7](#_Toc211505278)

[**6.4.6. Machine and Environment** 7](#_Toc211505279)

[**6.4.7. Hours to complete Part 1** 7](#_Toc211505280)

## **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** | **~ 189 ms** | **~ 1409 ms** | **~ 1609 ms** |
| 2 | **Original C – Release Mode** | **~ 132 ms** | **~ 637 ms** | **~ 760 ms** |

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

Averaged over the printed console output after the first 3 frames.

The commands to build the project:

1. To rebuild the Original C file in Debug mode:

* Remove-Item -Recurse -Force .\build
* cmake -S . -B build -G "Visual Studio 17 2022" -A x64
* cmake --build build --config Debug
* .\build\Debug\parallel.exe 123

1. To rebuild the Original C file in Release mode:

* Remove-Item -Recurse -Force .\build
* cmake -S . -B build -G "Visual Studio 17 2022" -A x64
* cmake --build build --config Release
* .\build\Release\parallel.exe 123

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

Enabling the flags in the CMake text file:

* target\_compile\_options(parallel PRIVATE $<$<CONFIG:Release>:/Qvec-report:2>)
* target\_compile\_options(parallel PRIVATE $<$<CONFIG:Debug>:/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.

MSVC’s vectorizer 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).
* **1203:** function calls inside the loop (sqrt/sqrtf) inhibit autovectorization unless inlined/contracted with /fp:fast.
* **1300/1305:** loop too small or complex control flow (early break, reductions intertwined).
* **500:** unknown aliasing or pointer escape patterns (access through global arrays).

These exactly match the structure of Physics (time-step dependence) and Graphics (two nested scans with an early exit), so the compiler largely leaves them scalar.

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

We want to test and compare 4 different scenarios:  
**Base Flags** = /Ox /Qvec-report:2

|  |  |
| --- | --- |
| Scenarios | Flags |
| A. Base: | set(FLAGS\_THIS\_RUN "${FLAGS\_BASE}") |
| B. + AVX2: | set(FLAGS\_THIS\_RUN "${FLAGS\_BASE};${FLAGS\_SIMD}") |
| C. + fp:fast: | set(FLAGS\_THIS\_RUN "${FLAGS\_BASE};${FLAGS\_FPFAST}") |
| D. + AVX2 + fp:fast | set(FLAGS\_THIS\_RUN "${FLAGS\_BASE};${FLAGS\_SIMD};${FLAGS\_FPFAST}") |

Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Scenarios | Physics  (moving sat) | Graphics  (space coloring) | Total |
| A | **Base Flags** | **~ 129 ms** | **~ 630 ms** | **~ 760 ms** |
| B | **Base + AVX2** | **~ 133 ms** | **~ 619 ms** | **~ 753 ms** |
| C | **Base + fp:fast (WOW!)** | **~ 88 ms** | **~ 368 ms** | **~ 459 ms** |
| D | **Base + AVX2 + fp:fast** | **~ 90 ms** | **~ 414 ms** | **~ 506 ms** |

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

- The major speedup comes from **/fp:fast** (scenario C) (~40% faster graphics and ~32% faster physics).

- Adding **/arch:AVX2** alone (scenario B) has only a small effect because the originalcode’s loop shapes aren’t SIMD-friendly.

- Combining **AVX2 + fp:fast** (scenario D) helps vs A/B but still trails C here.

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

**/fp:fast (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:**
  + The hot 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), which slashes scalar FLOPs.
  + It can also reorder operations in the weighted-sum/coloring loops and physics integrator, cutting temporary values and memory traffic.

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

* **What it does:**

Prints MSVC’s vectorizer diagnostics (C5001 vectorized / C5002 not vectorized + reasons).

* **Why does it help here:**

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

The best performance comes from enabling **/fp:fast** on top of a standard /O2 Release toolchain. /fp:fast allowed aggressive FP reordering and faster sqrt/div sequences in the physics and pixel-coloring loops, yielding large scalar speedups even though the loops still did not auto-vectorize.

But why **/arch:AVX2** is not in the winners list?

**/arch:AVX2** enables AVX2 instruction selection and makes vectorization possible where legal. The build logs showed the key loops still didn’t vectorize (C5002 with reasons like 1300/1106/500/1104), so AVX2 had little to latch onto. Hence, **fp:fast** dominated the gains by improving scalar math codegen, and adding AVX2 on top didn’t beat the fp:fast-only result.

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

We guess the only flag in the experiments that is realistically risky for correctness here is **/fp:fast**. Everything else we tried mainly changes performance, not semantics.

**/fp:fast**

It can produce wrong and unstable results.

* **What it does that is risky**
  + Reassociates floating-point expressions (changes math order).
  + Contracts mul+add into FMA, uses reciprocal/rsqrt sequences, flushes subnormals, relaxes NaN/sign-zero/exception rules.
* **Why can that break this program?**
  + My graphics loop does huge, weighted reductions and divides by sums of weights. Reordering + approximate 1/sqrt can change small weights a lot → different color sums; with some seeds it can push errors beyond the "ALLOWED\_ERROR" threshold or exceed the "ALLOWED\_NUMBER\_OF\_ERRORS".
  + Physics uses Euler integration with many tiny steps. Reassociation + approximate inverses can accumulate drift, so satellites diverge more from the sequential checker in the first frames.
* **Typical symptoms**
  + errorCheck() prints many mismatched pixels in frames 0–1 and aborts.
  + In extreme cases, satellites may spiral differently (still “looks OK” but fails the reference check).

**/arch:AVX2**

* Generally safe by itself. It just allows wider instructions; it doesn’t change math semantics unless combined with /fp:fast. In my logs, the hot loops didn’t vectorize anyway, so it mostly had no effect. Not a correctness risk in this project.

/fp:fast offered the best speed-up (~ 40 %), though at a risk of minor numeric drift.

## **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 and beneficial:**

Each pixel is independent. No shared writes, no loop-carried dependencies. In our code, we parallelized this with #pragma omp parallel for over i.

* **Physics satellite — Conditionally allowed:**

The update of satellite i at time t+1 depends only on satellite i at time t and the mouse state. There are no inter-satellite forces. Therefore, satellites are independent within a given time step and can be updated in parallel.

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

This loop carries dependencies through tmpPosition[i], tmpVelocity[i] from step t to step t+1. Parallelizing the time loop directly would break correctness.

* **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.

In summary, only the per-pixel graphics loop and per-satellite physics updates show meaningful speed-up potential with OpenMP

**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.**

#pragma omp parallel for private(i)

for(i = 0 ;i < SIZE; ++i) {...}

This line can parallelize the main loop to run in parallel in the parallel graphics engine pixel loop.

**2.**

#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;

}

This line parallelizes the initialization of arrays.

#pragma omp parallel for //private(idx2)

(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;

}

This loop parallelizes the final copy-back of calculated positions and velocities.

Still no vectorization by the compiler.

## **6.4. OpenMP Parallelization**

### **6.4.1. & 6.4.2. Average Frame times and code transformation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Frame sets | Mode | Physics  (moving sat) | Graphics  (space coloring) | Total |
| 1 | OpenMP | ~ 98 ms | ~ 43 ms | ~ 148 ms |
| 2 | OpenMP | ~ 98 ms | ~ 42 ms | ~ 147 ms |
| 3 | OpenMP | ~ 97 ms | ~ 43 ms | ~ 148 ms |
| 4 | OpenMP | ~ 97 ms | ~ 44 ms | ~ 147 ms |

\* These benchmarks are from “.\build\Release\parallel.exe 123”, so they are averages over all frames.

We just parallelized some loops along with the compiler parallelization

The OpenMP version reduced the total frametime to ~147 ms, mostly from parallelizing the Graphics loop.

### **6.4.3. Parallelized loops**

* Physics satellite loop (inside the Physics iteration loop) in parallelPhysicsEngine.

#pragma omp parallel for private

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

#pragma omp parallel for

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

* Graphics pixel loop in parallelGraphicsEngine.

#pragma omp parallel for private

for(i = 0 ;i < SIZE; ++i)

### **6.4.4. Performance scale with multi-cores**

We did the exercise on a single computer on TC303. Which configuration is core i7. So, maybe the performance could be scaled with the number of CPU cores, but we did not try on the other pc.

### **6.4.5. Bonus Question**

**(Bonus Point)**

**What we optimized (the final code)**

* **Multithreading (OpenMP):**
  + **parallelGraphicsEngine**: #pragma omp parallel for over the Graphics pixel loop (each pixel independent).
  + **parallelPhysicsEngine**: OpenMP over the copy-in (tmpPosition/tmpVelocity) and copy-out (writeback to satellites) loops.
  + Kept the Physics iteration (time) loop sequential to avoid 100k fork–join overhead.
* **Math precision:** Use of sqrtf in float paths for the graphics stage (physics uses double accumulators as in the original).
* **Compiler:** Aggressive optimizations /Ox, relaxed FP rules /fp:fast, AVX2 /arch:AVX2, OpenMP enabled.i

**Why is this the limit for this code?**

* The Graphics pass scales well with OpenMP, now ~45 ms.
* Physics is dominated by the 100,000-iteration time loop. Parallelizing the inner satellite loop inside that time loop would trigger 100k parallel regions, large scheduling overhead and cache churn. Keeping time sequential avoids that but leaves Physics mostly single-threaded (~100 ms).

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

We did the project on our own laptops. Asus ROG Strix SCAR 15, Intel Core i7-12700H.

We did the project on a single computer on TC303.

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

We spent approximately a week finishing part 1.

~2 hr/day, total of around 10 hr.