Tampere University

PARALLEL COMPUTING

M.Sc. Embedded Systems

Report Part 1

Group Members:

Mehdi Moallemkolaei	153010947	mehdi.moallemkolaei@tuni.fi
Md Ashfak Haider Nehal	153045077	mdashfakhaider.nehal@tuni.fi

OCT 2025

Part 1.

Contents

Tampere University

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`

2. 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). Therefore, the compiler could not vectorize either parallelPhysicsEngine or parallelGraphicsEngine loops due to function-call dependencies and complex loop control.

6.1.4. Best Flags:

We want to test and compare 4 different scenarios:

Base Flags = /Oy /Ob2 /O2 /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 original code'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

/Ox

- 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:**
  - 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:**
  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.

The best performance comes from enabling **/fp:fast** on top of a standard /O2 Release toolchain.

6.1.6. Code Breaker Flags:

We used multiple random flags and they were mostly working properly, but in one case it failed. This combo: "/O2" "/fp:precise" "/arch:AVX2" failed to build. We couldn't find the reason, that might be due to some incompatibility or some issues with precise and AVX2! So we skipped precise and the problem solved.

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.

```
#pragma omp parallel for private(i)
for(i = 0 ;i < SIZE; ++i) {...}
```

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;
}
#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;
}
```

Still no vectorization by the compiler.

6.4. OpenMP Parallelization

6.4.1. Average Frame times

Frame sets	Mode	Physics (moving sat)	Graphics (space coloring)	Total
1	OpenMP	~ 98 ms	~ 43 ms	~ 145 ms
2	OpenMP	~ 98 ms	~ 42 ms	~ 144 ms
3	OpenMP	~ 97 ms	~ 43 ms	~ 143 ms
4	OpenMP	~ 97 ms	~ 44 ms	~ 145 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 ~144 ms, mostly from parallelizing the Graphics loop.

6.4.2. Extra code transformation

A. Switched to sqrtf where variables are float (graphics & init) to avoid implicit double promotions.
B. Kept the original math to preserve visual correctness and pass the error checker.
C. Used /fp:fast to allow contraction and math re-association that MSVC's vectorizer benefits from.

6.4.3. Parallelized loops

A. Physics satellite loop in parallelPhysicsEngine.
Why: trivially data-parallel and race-free. Practical speedup is tiny but harmless.

```
#pragma omp parallel for private
for (idx = 0; idx < SATELLITE_COUNT; ++idx)

#pragma omp parallel for
for (idx2 = 0; idx2 < SATELLITE_COUNT; ++idx2)
```

B. Graphics pixel loop in parallelGraphicsEngine.
Pixels are independent; each iteration writes a distinct pixels[i].

```
#pragma omp parallel for private
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 and are bandwidth-bound.

6.4.6. Machine and Environment

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.