Parallel Project

# 6 First part: CPU parallelization

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

What are the average frame times of the physics routine (“satellite moving”), graphics routine (“space coloring”), and total frame time (milliseconds) after the first few frames:

1. **With the Original C Version without any compiler optimization flags (in Debug-mode)?**

|  |  |  |
| --- | --- | --- |
| Time spent on moving satellites | Time spent on space coloring | Total time in milliseconds between frames |
| 217 ms | 2444 ms | 2668 ms |

1. **With the Original C version using most compiler optimizations (in Release-mode)?**

|  |  |  |
| --- | --- | --- |
| Time spent on moving satellites | Time spent on space coloring | Total time in milliseconds between frames |
| 155 ms | 996 ms | 1154 ms |

1. **Did the compiler tell it managed to vectorize any of the loops in physics Engine or graphics Engine-functions?**

No.

* If the loops of ParallelPhysicsEngine and ParallelGraphicsEngine-functions failed to vectorize, report the reason?

For **ParallelPhysicsEngine**:

\Project\Satellites\parallel.c(129) : info C5002: loop not vectorized due to reason '1300'  
\Project\Satellites\parallel.c(142) : info C5002: loop not vectorized due to reason '1203'  
\Project\Satellites\parallel.c(138) : info C5002: loop not vectorized due to reason '1106'  
\Project\Satellites\parallel.c(177) : info C5002: loop not vectorized due to reason '1300'

For **ParallelGraphicsEngine**:

\Project\Satellites\parallel.c(226) : info C5002: loop not vectorized due to reason '500'  
\Project\Satellites\parallel.c(251) : info C5002: loop not vectorized due to reason '1104'  
\Project\Satellites\parallel.c(196) : info C5002: loop not vectorized due to reason '1106'

1. **Experiment with the SIMD instruction set and FP relaxation related optimization flags. Which are the compilation flags you found to give the best performance?**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions used** | **Time spent on moving satellites** | **Time spent on space coloring** | **Total time in ms between frames** |
| /fp:fast | 110 ms | 574 ms | 688 ms |
| /arch:AVX2 | 154 ms | 983 ms | 1142 ms |
| /fp:fast /arch:AVX2 | 97 ms | 567 ms | 668 ms |
| /fp:fast /arch:AVX | 108 ms | 567 ms | 679 ms |
| /arch:AVX | 153 ms | 981 ms | 1138 ms |
| /arch:sse4 | 154 ms | 988 ms | 1147 ms |
| /arch:avx512 | 153 ms | 977 ms | 1133 ms |
| /arch:sse4 /arch:avx512 | 153 ms | 976 ms | 1132 ms |
| /fp:fast /arch:avx512 | 110 ms | 573 ms | 686 ms |
| /fp:fast /arch:sse4 | 111 ms | 574 ms | 689 ms |
| /arch:AVX2 /arch:sse4 | 152 ms | 984 ms | 1141 ms |
| /arch:AVX2 /arch:avx512 | 154 ms | 980 ms | 1139 ms |

The best compilation flag was /fp:fast /arch:AVX2.

1. **Can you explain what each of the optimization flags you found to give the best performance does?**

* /fp:fast - Relaxes strict IEEE-754 rules so the compiler can reorder and "contract" FP ops (e.g., form FMAs), use approximate recip/rsqrt, assume no NaNs/Infs, and reassociate expressions. This often unlocks SIMD for sqrt/div heavy code and reduces dependencies. Result: faster but slightly different numerics.
* /arch:AVX - Allows the autovectorizer to target AVX (256-bit YMM registers) for float/double SIMD. Wider vectors than SSE, but no integer AVX2 ops.
* /arch:AVX2 - Enables AVX2 (still 256-bit, but with rich integer/vector ops and gathers). On FP workloads, also lets the compiler use FMA3 where profitable when contraction is allowed (e.g., with /fp:fast). Often the sweet spot on modern CPUs.
* /arch:AVX512 - Permits 512-bit vectors, mask registers, and wide loads/stores. Only used if the CPU/OS support it; many consumer CPUs don't, or the compiler may avoid it due to potential down-clocking. If unsupported, the compiler falls back to a lower ISA.
* /arch:SSE4 - Targets 128-bit SSE4.x. Narrower vectors; useful for older CPUs, typically slower than AVX/AVX2 on modern chips.

1. **Did you find some compiler flags which cause broken code to be generated, and if so, can you think why?**

No breakage occurred.

## 6.2 Generic algorithm optimization

* **Can you find any ways to change the code to either get rid of unnecessary calculations, or allow the compiler to vectorize it better, to make it faster. If yes, what did you do and what is the performance with your optimized version?**

1. **Inside parallelGraphicsEngine(), replacing the sqrt and merging two satellite loops**

**Why it’s faster**

* Removes 2× sqrtf per pixel (black hole & first satellite loop).
* Eliminates the second satellite loop entirely (we accumulate numerator+weights on the first pass and normalize once).
* Replaces divisions by using the reciprocal once (invW) and squared-distance comparisons (SIMD-friendly).
* Keeps byte-for-byte semantics vs your current sequential engine (we preserved the “nearest baseline + 3·weighted average” behavior and the inside-radius early white).

**Vectorization angle**

* Squared distance + branch consolidation turns the inner loop into straight-line, division-light code. Compilers are far more willing to autovectorize dx\*dx + dy\*dy, reciprocal, FMAs, and min-tracking than a path with sqrt + break + two passes.

1. **Inside parallelGraphicsEngine(), removing % and /**

**Why it’s faster**

* Eliminates one integer division and one modulo per pixel (both costly).
* Access patterns become perfectly linear, which the compiler and CPU prefetcher love.

**Vectorization angle**

Regular 2D loops + linear idx often trigger better auto vectorization and improved cache behavior.

**Performance**:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Time spent on moving satellites | Time spent on space coloring | Total time in milliseconds between frames |
| Before | 97 ms | 567 ms | 668 ms |
| After | 97 ms | 544 ms | 643 ms |

## 6.3 Code analysis for multi-thread parallelization

* **Which of the loops are allowed to be parallelized to multiple threads?**
  + **Physics iteration** (outer) - Not parallelizable. Loop-carried dependency across time steps (each step uses the state from the previous step).
  + **Physics satellite** (inner) - Parallelizable for a fixed time step (each satellite i updates its own state).
  + **Graphics pixel** (outer) - Parallelizable. Each pixel writes pixels[idx] once and reads shared, read-only data.
  + **Graphics satellite** (inner) - Parallelizable in theory (per-pixel, independent iterations), but not beneficial (trip count 64, early break, and reductions → thread overhead dominates).
* **Are there loops which are allowed to be parallelized to multiple threads, but which do not benefit from parallelization to multiple threads? If yes, which and why?**
  + **Physics satellite** (in the original nesting) - Allowed, but not beneficial: putting #pragma omp for inside the time-step loop causes PHYSICSUPDATESPERFRAME (=100 000) barriers per frame; overhead kills scaling.
  + **Graphics satellite** - Allowed, but not beneficial: only 64 iterations, branchy (break) and reduction-like pattern → threads add overhead; better target is SIMD.
* **Can you transform the code in some way (change the code without affecting the end results) which either allows parallelization of a loop which originally was not parallelizable, or makes a loop which originally was not initially beneficial to parallelize with OpenMP beneficial to parallelize with OpenMP? If yes, explain your code transformation?**

**Transformation 1 - Physics: loop interchange + register temporaries**

* **What changed**: Swapped the loop order so each thread owns a block of satellites and advances each one through all-time steps; kept x,y,vx,vy in registers and wrote back once at the end; used schedule(static).
* **Why it helps**: Removes the 100 000 per-step barriers and avoids false sharing from repeatedly writing adjacent tmpPosition[i]/tmpVelocity[i]. Coarse grains per thread ⇒ good scaling.
* **Side math cleanup**: Hoisted dt = DELTATIME/PHYSICSUPDATESPERFRAME, replaced two divides by |r| with one sqrt plus multiplies (invd, invd2).

**Transformation 2 - Graphics: parallelize rows, keep one-pass inner loop (no sqrt)**

* **What changed**: Parallelized the Graphics pixel loop by rows (#pragma omp parallel for schedule(static) over y); turned the pixel loop into a 2-D sweep (no %// per pixel); fused the two satellite passes into one pass that (a) early-outs to white if inside radius, (b) accumulates the weighted sums and weights, and (c) tracks the nearest using squared distance; normalized once.
* **Why it helps**: Big, cache-friendly chunks (rows) minimize scheduling overhead and keep writes linear; removing sqrtf and extra pass slashes scalar cost and makes the inner loop SIMD-friendlier.
* **Does your code transformation have any effect on vectorization performed by the compiler?**
  + Yes, replacing sqrtf/two-pass logic with squared-distance + one pass reduces divisions and branches, which helps the compiler's autovectorizer (and at least improves instruction-level parallelism even if SIMD is limited). The physics inner loop still has a time dependency, so SIMD across steps is not expected; the cleaned math reduces stalls.

**6.4 OpenMP Parallelization**

**Average frametimes (x64-Release, /fp:fast /arch:AVX2 + OpenMP):**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Time spent on moving satellites | Time spent on space coloring | Total time in milliseconds between frames |
| Before | 97 ms | 544 ms | 643 ms |
| After | 11 ms | 84 ms | 97 ms |

**Extra code transformations / optimizations**

* **Physics (T1):** Loop interchange so that each thread owns a subset of **satellites** and advances them through **all time steps** (removes 100k step barriers). Kept x,y,vx,vy in **registers** and wrote back **once** to avoid false sharing. Hoisted dt=DELTATIME/PHYSICSUPDATESPERFRAME and replaced two divides-by-‖r‖ with one sqrt + multiplies.
* **Graphics (T2):** Parallelized **rows** (outer pixel loop), changed to a **2-D sweep** (no per-pixel %//), removed sqrtf (squared-distance tests), and **fused the two satellite passes into one** (accumulate weights & nearest in one loop, then normalize once).

**Which loops were parallelized**

* **Physics iteration (outer):** **Not** parallelized (true loop-carried dependency across time).
* **Physics satellite (inner):** **Yes**, after loop interchange (OpenMP over satellites, one write-back per satellite).
* **Graphics pixel (outer):** **Yes**, OpenMP over **rows** (schedule(static)).
* **Graphics satellite (inner):** Left **single-threaded** (tiny trip count, early break, reductions). We optimized it for SIMD/ILP by removing sqrt and doing a single pass.

**Did any OpenMP parallelization break or slow things down?**

* **Naïve Physics satellite inside the time-step loop:** **Slowed down** due to **~100,000 barriers per frame**; also, more false sharing from per-step writes to adjacent tmpPosition[i]/tmpVelocity[i].  
  **Fix:** loop interchange + keep per-satellite state in registers, write back once, schedule(static).
* **Graphics pixel with collapse (2) and aggressive simd on MSVC:** Slight regression (scheduling overhead + conservative SIMD); the **row-parallel** version with one-pass inner loop was consistently faster.

**Scaling with core / hardware threads**

* Performance improved substantially when OpenMP was enabled and loops were coarse-grained (rows / satellites). On typical 6-16 logical-thread CPUs, **Graphics pixel** dominates scaling; **Physics** scales well after loop-interchange. If scaling plateaus, the reasons are:
  + 1. **Graphics-bound:** once graphics time dominates, adding threads to physics yields little change.
    2. **Runtime overhead:** too-fine task slicing (fixed by schedule(static) and coarse grains).
    3. **Memory bandwidth / cache effects:** especially at high thread counts.

**System & toolchain**

|  |  |
| --- | --- |
| **Setup** | **Configuration** |
| Ashfak PC | Processor: Intel(R) Core (TM) i5-10400F CPU @ 2.90GHz RAM: 16GB  Compiler: Virtual Studio x64 Version: 4.8.09032 |

**Effort**

* **Time spent on Part 1:** 43 hours