OpenMP Parallelization of ZPIC Particle-In-Cell Simulation Code

João Alves
Master's Degree in Advanced Computing
University of Minho
Braga, Portugal
pg61439@alunos.uminho.pt

Luís Carmo
Master's Degree in Advanced Computing
University of Minho
Braga, Portugal
pg61440@alunos.uminho.pt

***Abstract*—This work parallelizes the ZPIC Particle-In-Cell (PIC) simulation code using OpenMP to reduce execution time. The main goal was to profile it, find its most significant hotspots, and attempt to optimize them by utilizing various parallelization techniques. The solution was to be ran as efficiently as possible in the Deucalion supercomputer, specifically in nodes containing ARM-based Fujitsu A64FX processors.**

***Index Terms*—Particle-In-Cell, OpenMP, Parallel computing, Plasma physics simulation, Performance optimization**

I. Introduction

The main goal of this work is to parallelize the ZPIC 1D Particle-In-Cell simulation code [1] using OpenMP to reduce its execution time on the Deucalion supercomputer, specifically targeting ARM-based Fujitsu A64FX processors (48 cores @ 2.0 GHz, 32GB HBM2).

To achieve this goal, we followed a structured methodology: profiling to identify computational hotspots, analyzing data dependencies and parallelization opportunities, selecting strategies based on expected scalability, implementing optimizations, and measuring performance through scalability analysis.

Our implementation achieved 7.17× speedup with 32 threads, reducing execution time from 56.61s to 7.90s. Analysis revealed L2 cache contention as the primary scaling bottleneck beyond 24 threads.

II. Profiling and Hotspot Identification

To profile this application and our successive changes to it, we used the **perf** profiling tool, with the data visualized in **Firefox Profiler**[2].

Total (samples)	Self	
43%	3,137	3,137	spec_advance /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
22%	1,638	1,638	kernel_x /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
8.5%	617	617	yee_e /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
7.7%	559	559	dep_current_zamb /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
7.2%	528	528	spec_set_u /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
4.6%	337	337	emf_advance /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
2.9%	211	211	interpolate_fld /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
2.3%	169	169	__memcpy_generic /usr/lib64/libc-2.28.so
0.6%	45	45	__GI_memset inlined
0.3%	25	25	spec_sort /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
0.1%	4	4	perf_event_exec [kernel.kallsyms]
0.1%	4	4	finish_task_switch [kernel.kallsyms]
0.0%	2	2	calloc inlined
0.0%	1	1	flush_signal_handlers [kernel.kallsyms]
0.0%	1	1	cl_lock_release /lib/modules/4.18.0-348.23.1.el8_5.aarch64/extra/kernel/fs/lustre/obdclass.ko
0.0%	1	1	emf_move_window /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
0.0%	1	1	emf_update_gc /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
0.0%	1	1	__softirqentry_text_start [kernel.kallsyms]
0.0%	1	1	__kernel_gettimeofday [vdso]
0.0%	1	1	current_update /projects/F202500010HPCVLABUMINHO/uminhocp010/zpic_CompParalela/zpic
0.0%	1	1	_int_malloc /usr/lib64/libc-2.28.so
0.0%	1	1	free inlined
0.0%	1	1	rcu_core [kernel.kallsyms]
0.0%	1	1	rcu_report_qs_rnp [kernel.kallsyms]

Fig. 1. Hotspots detected with *Firefox Profiler*

As seen in **Figure 1**, the main hotspots are **spec_advance** (43% of runtime, ~31 seconds) and **kernel_x** (22% of runtime, ~16 seconds), accounting for 65% of total runtime. These two functions will be our main focus, although we also applied several smaller optimizations that are described later.

III. Optimizations

A. Parallelism

We parallelized the two main hotspots using OpenMP with `#pragma omp parallel for`.

a) *`spec_advance`:*

Inside this function, there's a particle loop with two parallelization challenges:

- It writes to a variable energy, which is external to the loop. Running it in parallel would cause a race condition. To fix this, each thread would need a private copy of energy, which they would write to. When all threads finish, all the copies are summed into the original variable. This is possible with OpenMP by adding `reduction(+:energy)`.
- The nested function `dep_current_zamb`, called inside the loop, contains multiple RAW dependencies, meaning `#pragma omp atomic` instructions had to be used. Alternative solutions like thread-local grids introduced greater overhead than just using atomics.

b) *`kernel_x`:*

The original implementation has **read-after-write (RAW) dependencies**—each iteration reads values modified by the previous one, preventing direct parallelization.

- **Double buffering solution:** We use separate input and output (`J_temp`) buffers. All threads read from `J` and write to `J_temp`, then swap pointers. This eliminates race conditions and enables us to use OpenMP on it, with the only downside being that it requires double the memory.

  We also switched from Array-of-Structures (`float3* J`) to Structure-of-Arrays (separate `float* Jx, Jy, Jz` arrays) for better cache locality and vectorization.

c) *Others:*

spec_set_u: We parallelized 3 of 4 loops in this function. The second loop (momentum accumulation) was kept sequential, as atomic operations for concurrent grid writes added more overhead than sequential execution.

OMP_PROC_BIND: Changing this OpenMP setting [3] to `close` helps a lot specifically for the A64FX, because threads are divided into four groups [4], each having their own L2 cache. This setting tells OpenMP to keep its threads within the same group, so cache access is more efficient, saving us a few seconds of time.

No scheduling?: For every loop that we made parallel, all scheduling options were tested, but we found that keeping it static (in this case not specified at all) had the best performance.

B. Vectorization

In addition to parallelization, we decided to improve performance even more by modifying some sections of the code, so the compiler would vectorize as many loops as possible, taking advantage of the A64FX's support for SVE instructions. The main change we made was switching from a `float3` structure, which stores a single instance of `x, y, z`, to a float3_soa layout, where each variable is a separate array. As in: `B[i].x -> B.x[i]`. This ended up significantly reducing cache misses for loops that traversed, for example, all X values in a `float3` structure, due to them being close together in memory.

Note: Vectorized loops contain `#pragma omp simd simdlen(16)`, which is a hint added to tell the compiler that it should use SVE vectorization for that loop, instead of defaulting to NEON.

C. Scalability Analysis

We evaluated performance on the Deucalion supercomputer using Fujitsu A64FX processors (48 cores @ 2.0 GHz, 32GB HBM2).

Figure 2 shows speedup vs thread count. Our implementation achieved maximum speedup of 7.17× with 32 threads, reducing execution time from 56.61s to 7.90s. Performance scales well up to 24 threads, plateauing at 32 threads and degrading slightly at 48 threads (6.87×, –4.1% vs best).

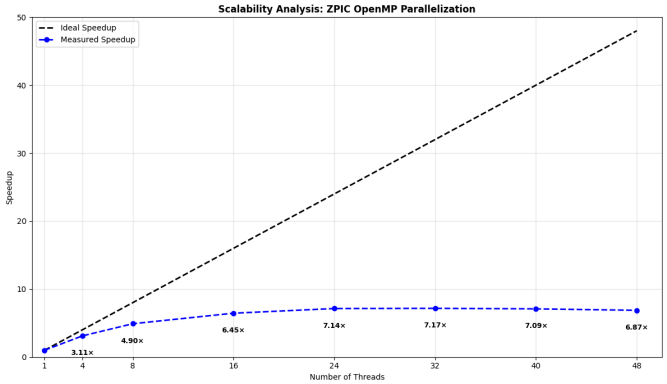

Fig. 2. Scalability Analysis

CPI analysis (**Figure 5**) shows efficient instruction execution with CPI decreasing from 1.36 to 0.69, a 2× improvement demonstrating successful SIMD vectorization. Despite this increased efficiency, speedup stagnates, revealing that L2 cache contention is the real constraint.

Figure 3 quantifies this bottleneck. L2 cache refills go from 208K (1 thread) to 219.7M (48 threads), a 1000× increase. The critical threshold is reached between 8 and 16 threads, at which point L2 refills increase from 116K (8 threads) to 71.9M, a 620× increase as more threads start competing for the same L2 cache.

Performance decrease is directly caused by this conflict. Memory wait cycles (`ld_comp_wait_l2_miss`) increased by 1270× from 11.4M (1 thread) to 14.5B (48 threads) in **Figure 4**. Instead of processing data, threads spend exponentially more time waiting for L2 refills.

The speedup limitation can be explained by the combination of higher IPC and saturated L2 bandwidth: the CPU performs instructions effectively when data is available, but the availability of data becomes a major constraint.

IV. Conclusion

This study effectively used OpenMP to parallelize the ZPIC PIC simulation code, resulting in a 7.17× speedup with 32 threads on ARM-based Fujitsu A64FX processors. Our parallelization strategy combined reduction clauses, double buffering to eliminate RAW dependencies, and vectorization using SVE instructions, successfully addressing both correctness and performance challenges.

The main bottleneck, according to performance studies, was L2 cache contention. Beyond 16 threads, L2 refills increased by 1000× and wait cycles by 1270×. Even with effective CPU utilization (CPI decreasing from 1.36 to 0.69), additional speedup is limited by thread competition for shared L2 cache. This demonstrates that **memory bandwidth**, rather than computational capacity, limits scaling on modern HPC architectures.

Performance scales efficiently up to 24 threads (7.14× speedup, 29.75% efficiency), with diminishing returns beyond this point. While 32 threads achieves peak speedup (7.17×), we recommend **24 threads** for production workloads as the optimal balance between performance and resource efficiency - the marginal 0.4% gain does not justify the 33% increase in resource consumption and significantly elevated cache pressure.

References

[1] BMalaca, "zpic_CompParalela." 2025.

[2] Mozilla, "Profiling with Linux perf - Firefox Profiler." [Online]. Available: https://profiler.firefox.com/docs/#/./guide-perf-profiling

[3] OpenMP Architecture Review Board, "OpenMP Specification: OMP_PROC_BIND (Section 6.4)." 2018.

[4] Fujitsu Limited, "FUJITSU Processor A64FX Datasheet (English)." 2020. [Online]. Available: https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf

A. Appendix A

A.1. L2 Cache Refills vs Threads

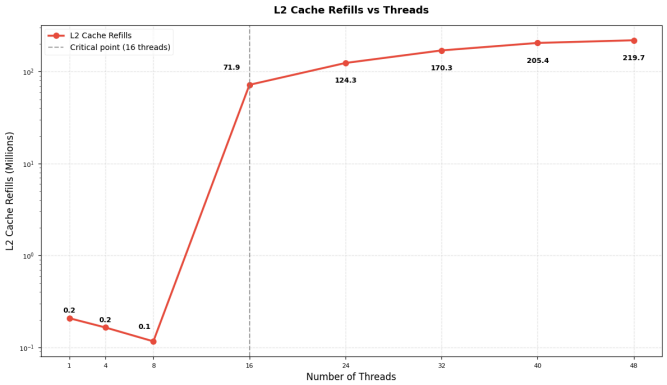

Fig. 3. L2 Cache Refills vs Threads

A.2. L2 Wait Cycles vs Threads

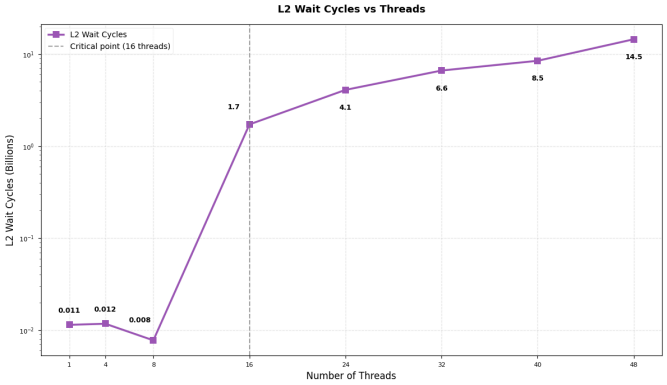

Fig. 4. L2 Wait Cycles vs Threads

A.3. CPI Evolution vs Threads

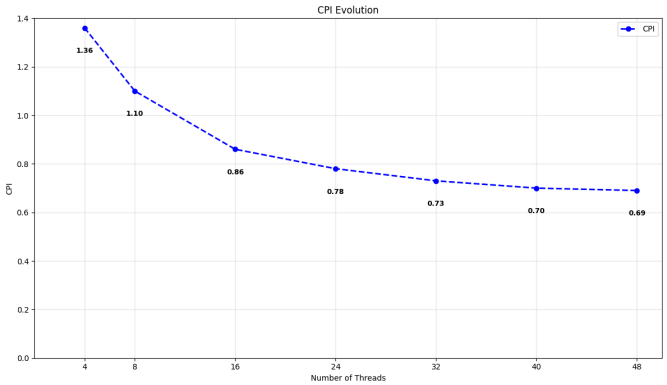

Fig. 5. CPI Evolution vs Threads