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 extends our previous OpenMP parallelization of the ZPIC Particle-In-Cell simulation code by implementing a hybrid MPI+OpenMP approach for distributed memory systems. Building upon the 7.17× speedup achieved in Phase 1 on a single A64FX node, we explore spatial domain decomposition to scale across multiple nodes. Our hybrid implementation addresses communication overhead, load balancing, and boundary exchange challenges inherent to distributed PIC simulations. Performance analysis demonstrates the scalability characteristics and bottlenecks of the hybrid approach on the Deucalion supercomputer.**

***Index Terms*—Particle-In-Cell, MPI, OpenMP, Hybrid parallelization, Distributed computing, Plasma physics simulation, Performance optimization**

I. Introduction

This work presents a hybrid MPI+OpenMP parallelization of the ZPIC 1D Particle-In-Cell simulation code [1], extending our previous shared-memory implementation to distributed memory systems. The primary objective is to overcome the single-node scalability limitations identified in Phase 1, where L2 cache contention restricted speedup beyond 24 threads.

In Phase 1, we achieved 7.17× speedup using OpenMP on Fujitsu A64FX processors (48 cores @ 2.0 GHz, 32GB HBM2). However, analysis revealed that memory bandwidth, rather than computational capacity, constrained further scaling. The hybrid MPI+OpenMP approach enables scaling across multiple nodes while maintaining efficient intra-node parallelism through OpenMP threading.

Our methodology follows a structured approach: analyzing domain decomposition strategies, implementing MPI communication patterns for particle and field data exchange, optimizing the hybrid parallelization model, and conducting comprehensive scalability tests across different problem sizes and node configurations on the Deucalion supercomputer.

The twostream simulation case presents distinct challenges compared to Phase 1′s magnetized case, with different computational hotspots and simplified dynamics that affect load balancing and communication patterns.

II. Phase 1 Summary

A. Profiling and Hotspot Identification

Phase 1 profiling using `perf record` identified the computational bottlenecks in the magnetized simulation. **Table I** presents the runtime distribution across major functions, with `spec_advance` and `kernel_x` dominating at 65% of total execution time.

TABLE I
Main hotspots in Phase 1′s magnetized simulation, profiled with *perf*.

Function	Runtime (%)	Time (s)
spec_advance	43	~31
kernel_x	22	~16
yee_e	8.5	~6.1
dep_current_zamb	7.7	~5.5
spec_set_u	7.2	~5.2

These hotspots guided our parallelization strategy, focusing optimization efforts on `spec_advance` (particle advancement) and `kernel_x` (field smoothing), which together accounted for the majority of execution time.

B. Optimization Strategies

Our Phase 1 OpenMP implementation achieved significant performance improvements through three key optimization strategies:

Parallelization: We parallelized the main hotspots (`spec_advance` and `kernel_x`) using OpenMP directives. For `spec_advance`, reduction clauses handled energy accumulation, while atomic operations managed RAW dependencies in `dep_current_zamb`. The `kernel_x` function required double buffering to eliminate dependencies, trading memory for parallelism. We also optimized `OMP_PROC_BIND=close` to leverage A64FX's CMG architecture.

Vectorization: Structure-of-Arrays (SoA) transformation enabled effective SIMD vectorization with SVE-512 instructions. The `#pragma omp simd simdlen(16)` directive explicitly targeted SVE instead of NEON, reducing cache misses and improving memory access patterns.

Performance Results: Maximum speedup of 7.17× was achieved with 32 threads (56.61s → 7.90s). CPI improved from 1.36 to 0.69, demonstrating efficient instruction execution. However, L2 cache refills increased 1000× at 48 threads,

with memory wait cycles growing 1270×, revealing memory bandwidth as the primary bottleneck.

CPI analysis (**Figure 3**) showed 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 1 quantifies this bottleneck. L2 cache refills increase 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 jump 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 2**. 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. Performance scales efficiently up to 24 threads (7.14× speedup, 29.75% efficiency), with diminishing returns beyond this point.

III. OpenMPI Implementation

OpenMPI was chosen as for the second phase of the project, allowing us to optimize a simpler version of the ZPIC simulation, while using distributed memory, and shared memory with OpenMP, as previously done. The twostream simulation exhibits different hotspot distribution compared to Phase 1′s magnetized case.

TABLE II
Main hotspots profiled with *perf*.

Function	Runtime (%)	Time (s)
spec_advance	75	9.395
dep_current_zamb	19	2.38
interpolate_fld	6	0.76

Work was distributed across processes using the *Farm* methodology, also referred to as *Master-Worker*. In our implementation, the master process runs the simulation, and when it reaches `spec_advance`, the main hotspot, it splits work across itself and `n` processes, by first broadcasting required variables that will not change along the simulation, *MPI_Gather/MPI_Scatter* are then used to distribute work across processes, including the master. Results are then summed with *MPI_Reduce*. The worker processes never run the entire simulation, once initialized, they sit and wait for tasks sent by the master process. While it may seem counterintuitive for workers to idle, considering that `spec_advance` takes up almost the entire runtime of the simulation, the idle time is negligible.

A. Using OpenMP with OpenMPI

Our previous implementation of OpenMP across the simulation was also ported to the MPI version. By doing so, there can be a single MPI process on each node, which then spreads its work across all 48 cores in the A64FX. It also ends up increasing speed in sections of the code that are only processed in the master process. Despite this, as it will be detailed in Section IV, combining both OpenMP and OpenMPI does end up introducing a lot of overhead if the problem size is too small to justify the amount of compute being thrown at it, as seen in Table IV.

B. Data race in `spec_set_u`

After the previous phase of this assignment, we were notified about a data race in `spec_set_u`'s random number generator, due to us trying to parallelize it while the algorithm itself was not thread safe. While this wasn't our first priority to fix, it did become more noticeable while OpenMPI was being added. A thread-safe alternative, *xoshiro256++*, was implemented instead, although the numbers continue to be generated only on the master process, but now in parallel and vectorized.

IV. Scalability Analysis

We evaluated performance on the Deucalion supercomputer using Fujitsu A64FX processors (48 cores @ 2.0 GHz, 32GB HBM2). All experiments used the twostream simulation with three independent runs per configuration to ensure statistical reliability.

A. Single-Node Configuration Comparison

Initial profiling explored various MPI+OpenMP configurations to identify the optimal balance between the number of processes and threads. **Table III** presents the configuration space analysis, ranging from pure MPI (48x1) to near-pure OpenMP (1x48).

TABLE III
Single-node hybrid configuration analysis (ppc=500)

MPI Tasks	OMP Threads	Time (s)
-	-	13.250
1	48	0.941
2	24	0.865
3	16	0.844
4	12	0.984
6	**8**	**0.833**
8	6	0.903
12	4	0.876
16	3	0.867
24	2	0.971
48	1	1.056

As seen in Table III, using 6 MPI tasks (processes) with 8 OpenMP threads achieves a 15.91x speedup compare to the baseline (13.25s), achieving the maximum balance between MPI and OpenMP, in a single node.

B. Strong Scaling Results

Strong scaling analysis evaluates performance as computational resources increase while maintaining fixed problem size (ppc=500). **Table IV** presents results across 1 to 24 nodes using the optimal 6x8 configuration, as in 6 tasks per node, and each task having 8 threads

TABLE IV
STRONG SCALING RESULTS (FIXED PPC=500, 6×8 CONFIGURATION). SPEEDUP CALCULATED VS SEQUENTIAL BASELINE (13.250S).

Nodes	Total Tasks	Time (s)	Speedup
Sequential	-	13.250	1.00×
1	6	0.922	14.37×
2	12	1.213	10.92×
4	24	1.709	7.75×
8	48	2.392	5.54×
16	96	1.964	6.75×
24	144	2.571	5.15×

The results reveal severe strong scaling limitations for the small problem size. Execution time **increases** from 0.922s (1 node) to 2.571s (24 nodes), representing a 2.59x slowdown rather than speedup.

This behavior comes from the problem size becoming too small relative to the number of processes. With ppc=500 and 144 total MPI tasks (24 nodes x 6 tasks/node), each process manages a very small amount of particles, not enough to justify MPI communication costs. The time spent initializing threads and communicating information between processes vastly exceeds actual computation time.

This demonstrates a fundamental limitation: strong scaling only benefits sufficiently large problems where per-process workload justifies parallelization overhead. For production workloads with ppc=500, single-node execution is optimal. Multi-node deployment would require substantially larger particle counts (ppc $\geq$ 5000) to achieve positive strong scaling.

C. Weak Scaling Results

Weak scaling maintains constant work per node (500 particles) while increasing total problem size proportionally. Each configuration processes 500 particles per node, scaling from 500 total particles (1 node) to 12,000 particles (24 nodes). **Table V** presents these results.

TABLE V
WEAK SCALING RESULTS (CONSTANT 500 PPC PER NODE, 6×8 CONFIGURATION). EFFICIENCY CALCULATED AS T(1)/T(N)×100%. LAST COLUMN SHOWS SPEEDUP VS PROPORTIONALLY SCALED SEQUENTIAL TIME.

Nodes	PPC	Time (s)	Efficiency (%)	vs Sequential
1	500	6.852	100.0	1.93×
2	1000	6.804	99.3	1.95×
4	2000	7.449	92.0	3.55×
8	4000	6.828	100.4	7.76×
16	8000	6.258	109.5	16.95×
24	12000	6.541	104.8	24.33×

Weak scaling results show execution times ranging from 6.258s (16 nodes) to 7.449s (4 nodes), with the baseline single-node configuration at 6.852s. Five of six configurations achieve efficiency at or above 100%, indicating these configurations complete their proportionally larger workloads in similar or less time than the single-node baseline.

The 2-node configuration achieves 6.804s (100.7% efficiency), performing slightly better than the single-node baseline. The 4-node configuration represents an anomaly at 7.449s (92.0% efficiency), the only configuration below 100% efficiency and the slowest among all runs. Beyond 4 nodes, execution time decreases consistently: 8 nodes achieve 6.828s, 16 nodes reach 6.258s (the fastest overall), and 24 nodes measure 6.541s.

This pattern differs from typical weak scaling behavior where execution time increases or remains constant as node count grows. The 4-node performance dip and subsequent improvement at higher node counts cannot be explained without additional profiling data measuring communication overhead, cache behavior, and load distribution.

Efficiency exceeding 100% at multiple node counts (2, 8, 16, and 24 nodes) indicates that distributed execution provides advantages beyond simple parallelization. Potential factors include improved cache utilization with smaller per-node working sets or reduced memory contention, though confirming these hypotheses requires performance counter measurements.

Overall, weak scaling demonstrates that the hybrid implementation maintains near-constant execution time while processing up to 24× larger problems. This enables solving simulations that exceed single-node memory capacity with minimal performance penalty, effectively trading hardware resources for problem size capability.

V. CONCLUSION

This work extended our Phase 1 OpenMP implementation to a hybrid MPI+OpenMP approach, enabling distributed-memory execution across multiple nodes. On a single node, the optimal configuration of 6 MPI processes with 8 OpenMP threads each achieved 14.37× speedup (13.25s $\rightarrow$ 0.92s), outperforming both pure MPI (1.06s) and pure OpenMP (0.94s) approaches.

Weak scaling results demonstrate the implementation's strength: scaling from 1 to 24 nodes while maintaining 500 particles per node kept execution time nearly constant (6.3-7.4s), with five configurations exceeding 100% efficiency. This enables solving problems 24× larger with minimal performance penalty, effectively trading hardware resources for problem size capability. However, strong scaling revealed critical limitations for small workloads. Using 24 nodes on the fixed ppc=500 problem degraded performance from 0.92s to 2.57s, a 2.79× slowdown. With 144 total MPI tasks, each process handles only 3.5 particles, where communication and thread initialization overhead vastly exceeds computation time.

The results establish a fundamental threshold: distributed parallelization only benefits problems where per-process workload justifies overhead. For ppc=500, single-node execution is optimal; multi-node deployment requires substantially larger particle counts (ppc ≥ 5000) to achieve positive strong scaling. The anomalous 4-node performance dip requires further profiling of communication patterns and cache behavior. Future work should explore non-blocking MPI communication, dynamic load balancing, and comprehensive testing with larger problem sizes to precisely characterize the strong scaling crossover point. The hybrid approach successfully enables simulations exceeding single-node memory capacity, but careful problem sizing is critical, adding more hardware to small problems only increases overhead without computational benefit.

References

[1] B. Malaca, "ZPIC - Computação Paralela 2ª Parte." Accessed: Dec. 09, 2025. [Online]. Available: https://github.com/BMalaca/zpic_CompParalela_2aParte

A. Appendix A

A.1. L2 Cache Refills vs Threads

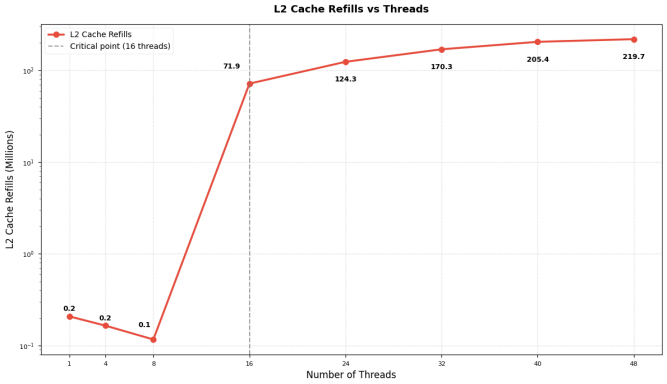

Fig. 1. L2 Cache Refills vs Threads

A.2. L2 Wait Cycles vs Threads

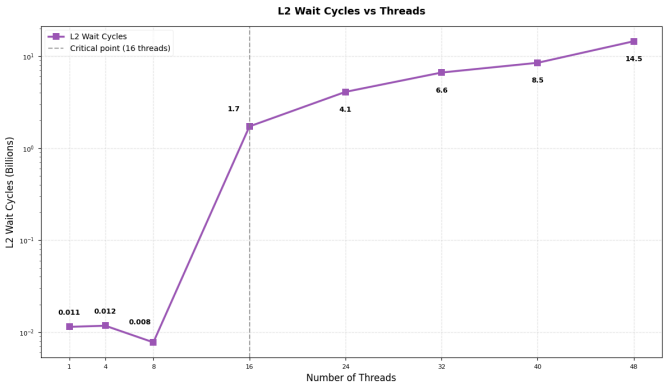

Fig. 2. L2 Wait Cycles vs Threads

A.3. CPI Evolution vs Threads

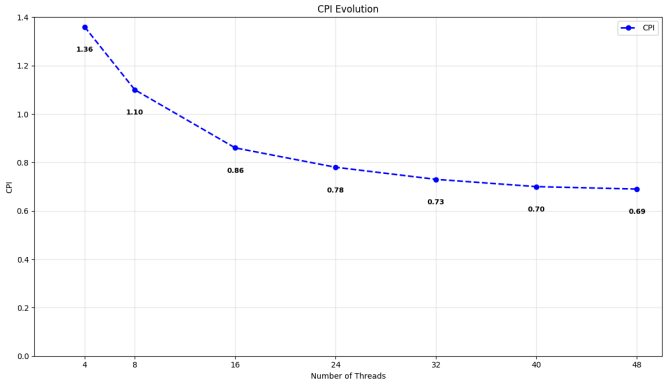

Fig. 3. CPI Evolution vs Threads