Optimizing a Particle-in-Cell Code on Intel Knights Landing

Minhua Wen[*], Min Chen[†], **James Lin**[*‡],

[*]Center for HPC, Shanghai Jiao Tong University(SJTU), Shanghai

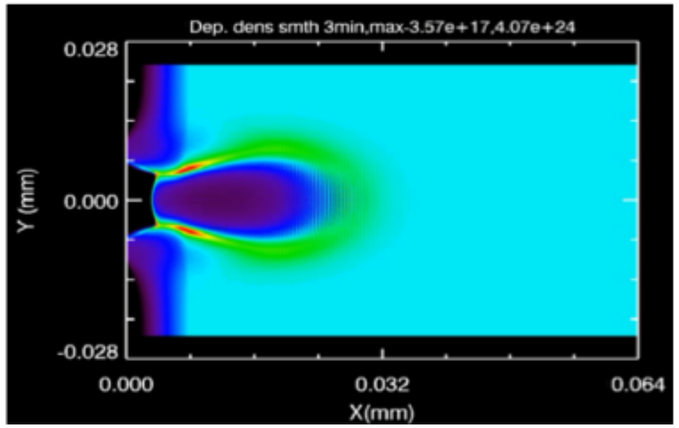
[†]School of Physics and Astronomy, SJTU, Shanghai

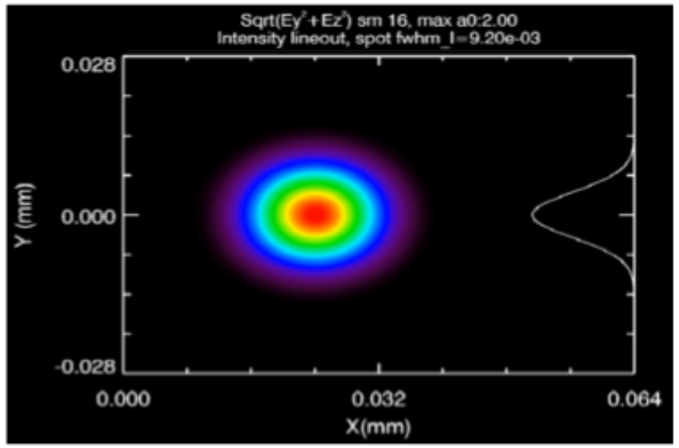
[‡] Tokyo Institute of Technology, Japan

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Outline

Introduction to VLPL-S PIC code

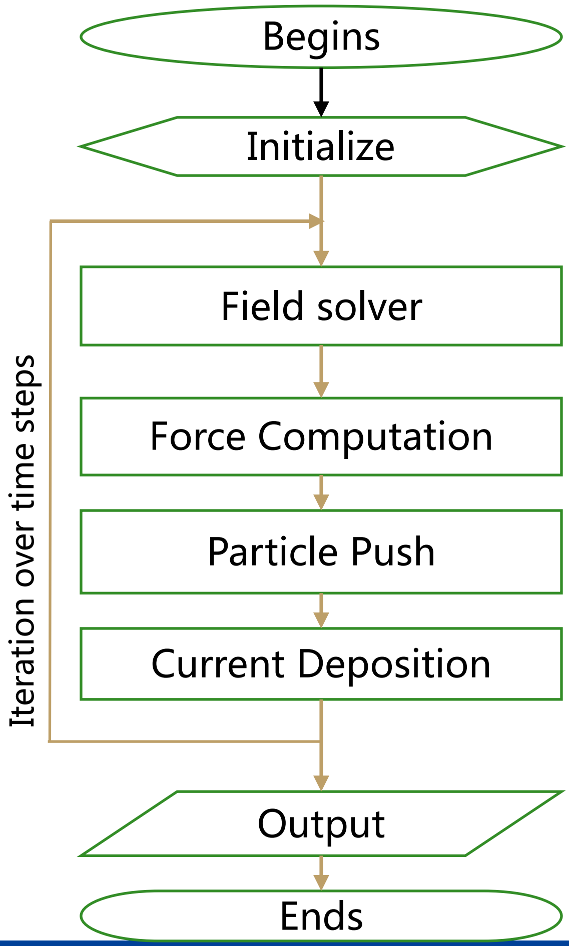




- **Particle-in-cell** (PIC) method is a widely used first-principles model for laser plasmas simulations, with many well-known implementations for different scenarios, such as OSIRIS, EPOCH, VSim, VLPL
- **Virtual laser plasma laboratory** (VLPL) PIC code is originally developed by A. Pukhov in Germany, VLPL-S is a modified In-house code SJTU Laboratory For Laser Plasmas.
- **VLPL-S** focus on the interaction between high intensity ultrashort laser and plasma, used for both theoretical and experimental purpose
  - laser-driven electron acceleration
  - acceleration of ion beams
  - generation of X-ray and gamma ray radiation

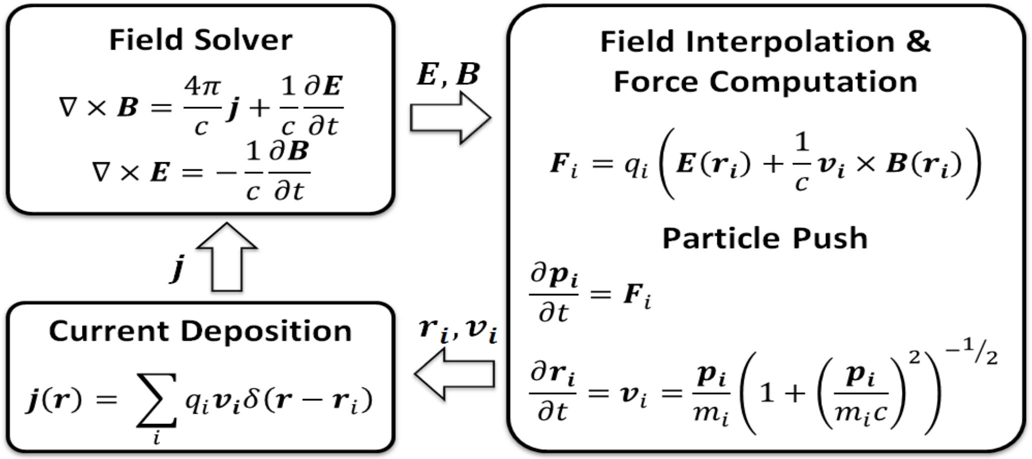
The key algorithm of VLPL-S is as general PIC method

Flow Chart



Equations

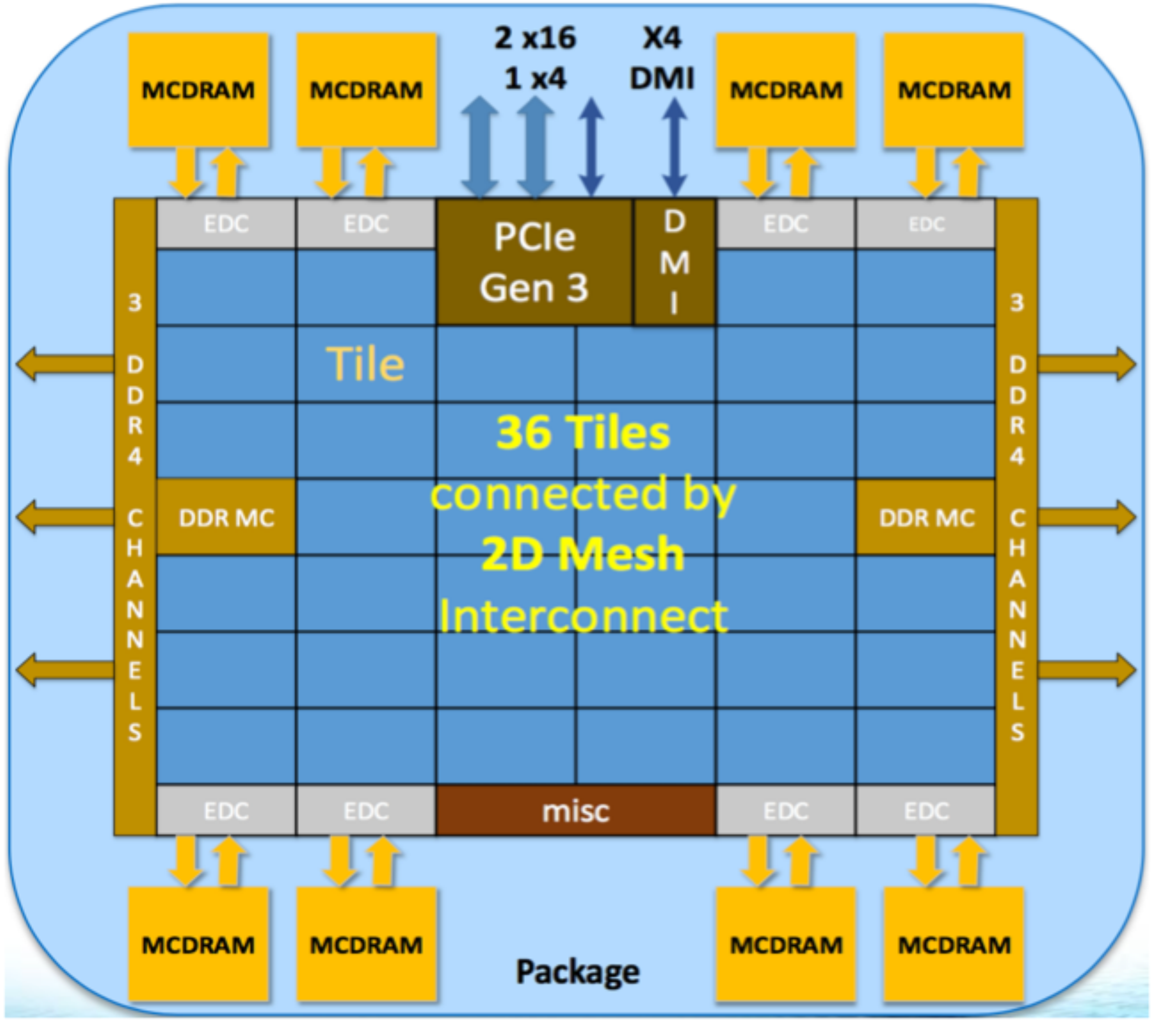
Solving the Maxwell' s equations with the FDTD method

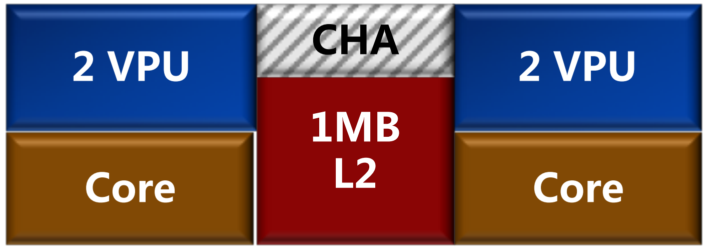

Field Solver
$\nabla \times \boldsymbol{B} = \frac{4\pi}{c}\boldsymbol{j} + \frac{1}{c}\frac{\partial \boldsymbol{E}}{\partial t}$
$\nabla \times \boldsymbol{E} = -\frac{1}{c}\frac{\partial \boldsymbol{B}}{\partial t}$
$\boldsymbol{E}, \boldsymbol{B}$
Field Interpolation & Force Computation
$\boldsymbol{F}_i = q_i\left(\boldsymbol{E}(\boldsymbol{r}_i) + \frac{1}{c}\boldsymbol{v}_i \times \boldsymbol{B}(\boldsymbol{r}_i)\right)$
Particle Push
$\frac{\partial \boldsymbol{p}_i}{\partial t} = \boldsymbol{F}_i$
$\frac{\partial \boldsymbol{r}_i}{\partial t} = \boldsymbol{v}_i = \frac{\boldsymbol{p}_i}{m_i}\left(1 + \left(\frac{\boldsymbol{p}_i}{m_i c}\right)^2\right)^{-1/2}$
$\boldsymbol{r}_i, \boldsymbol{v}_i$
Current Deposition
$\boldsymbol{j}(\boldsymbol{r}) = \sum_i q_i \boldsymbol{v}_i \delta(\boldsymbol{r} - \boldsymbol{r}_i)$
$\boldsymbol{j}$

Workloads

	Features	Number of Cells	Particles per Cell
testA	Load balanced case used for benchmark.	1200x1200	16
testB	A typical case used in researches, load imbalance.	2500x240	9

Intel Knights Landing overview





- 36 Tiles , 2 cores per tile
- 4 Threads per core
- 2 x 512bit Vector Processing Units per core
- 6 channels of DDR4 2400 up to 384GB @ ~81 GB/s Streams Triad
- 8GB/16GB of on-spackage MCDRAM memory @ ~419 GB/s Streams Triad
- 2.7 TF DGEMM.
- OPA (Intel® Omni-Path Architecture) is integrated.

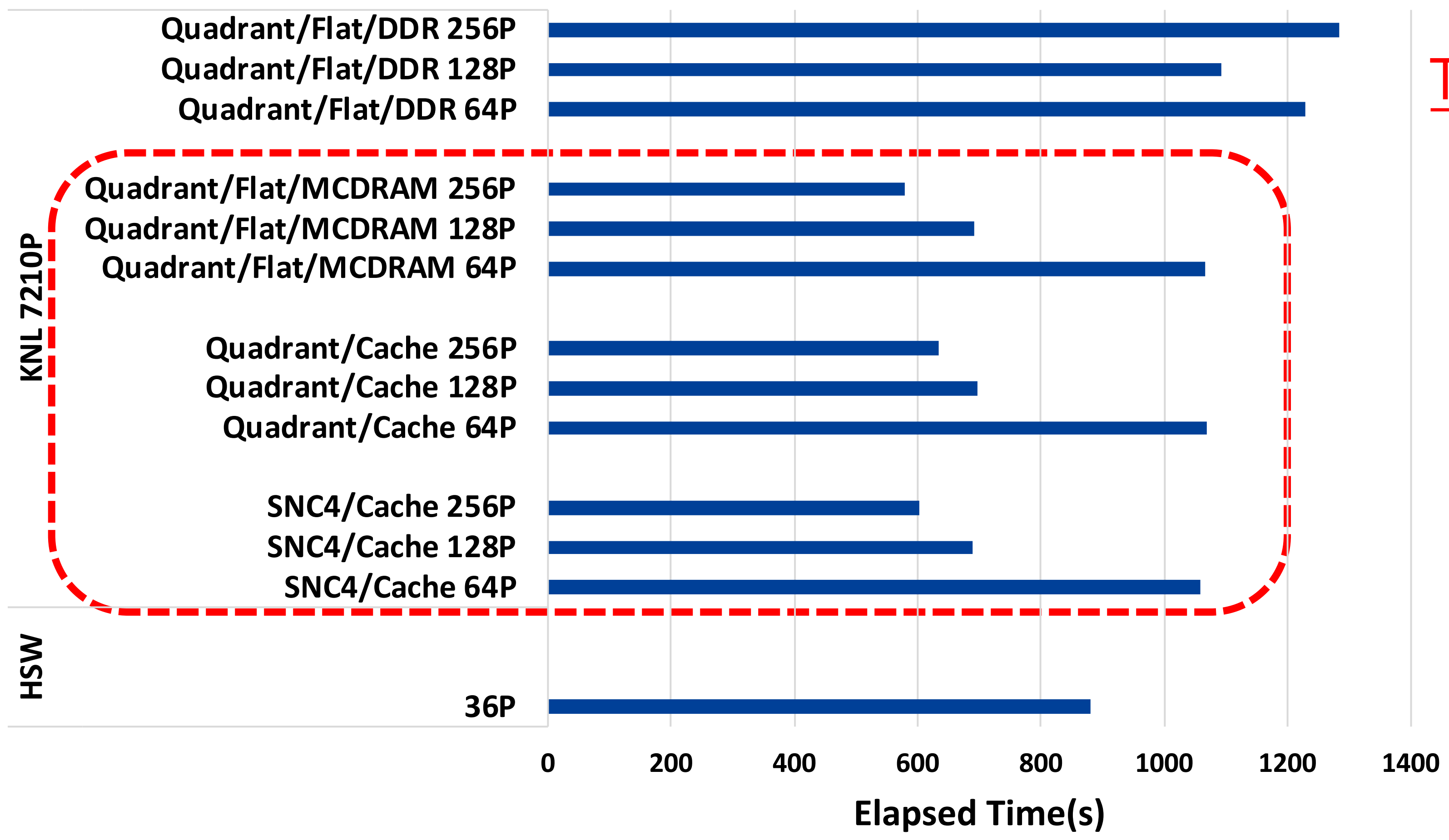
Related work

- Many implementations proved the parallelism of the origin PIC algorithm.
  - Bastrakov et al. reported their implementation of PIC code achieved up to 7x speedup on an 8-core Xeon E5-2690 processor. (Sergey Bastrakov, et al. Particlein-cell plasma simulation on cpus, gpus and xeon phi coprocessors. In ISC, pages 513–514. Springer, 2014.)
  - Surmin et al. achieves 1.6x speedup of their PICADOR code on KNC than a single E5-2600 CPU. (IA Surmin, et al. Particle-in-cell laser-plasma simulation on xeon phi coprocessors. Computer Physics Communications, 202:204– 210, 2016.)
- Ionization is included in the VLPL-S code, thus more memory consumption and more communication cost inevitably happen.
  - Nakashima showed the toughness of manycore and SIMD-aware implementation of PIC. (Hiroshi Nakashima. Manycore challenge in particle-in-cell simulation: How to exploit 1 TFlops peak performance for simulation codes with irregular computation. Computers & Electrical Engineering, 46:81–94, 2015.)

Test machine configurations

Name	Xeon Phi 7210	E5-2699v3
Sockets	1	2
Frequency	1.3GHz	2.3GHz
Cores	64	36
Threads	256	72
DDR4 Memory	6×16GB	8×16GB
MCDRAM	16GB	N/A
Short Name	KNL	HSW

Performance of the initial implementation of VLPL-S code



The lower the better

Code version: original code

testA

Performance analysis of the initial implementation of VLPL-S

- Due to the first-principle nature, PIC simulations generally require intensive computation
- AOS data structure causes inefficient memory access
- Vectorization is not used
- IO takes almost 10% of overall time in production cases
- Load imbalance exists in many production cases
    - Distribution of particles is not uniform in initial condition
    - New particles are generated during iterations

Outline

- Introduction
- Optimizations
- Results
- Conclusion

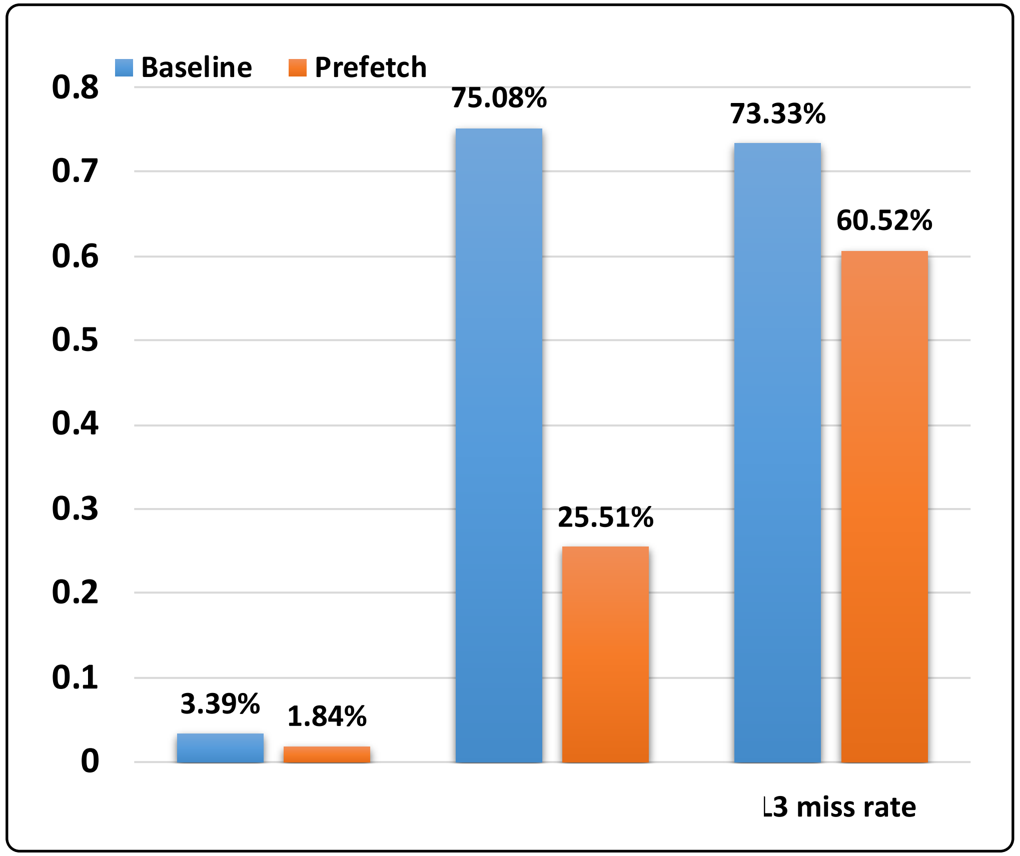
Optimizations

- Compute-oriented Optimizations
  - Memory access optimization
  - Thread level parallelization
  - Vectorization
- Parallel IO
- Dynamic load balancing optimization

Memory access optimization

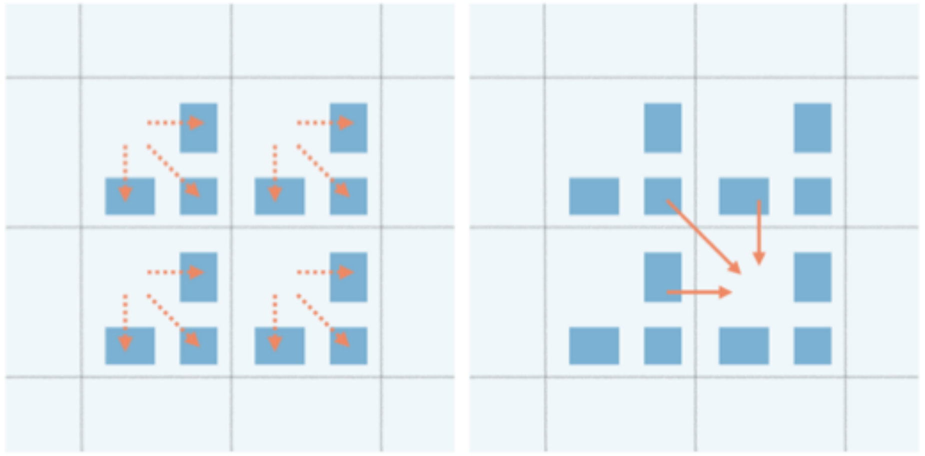
- The hotspot is traversing over linked list.
  - Irregular memory access hurts the performance.
- Prefetching with intrinsics
  - Prefetch next particle into L1 cache.
  - Prefetch next cell into LLC.

Cache miss rate reduced



Thread level optimization

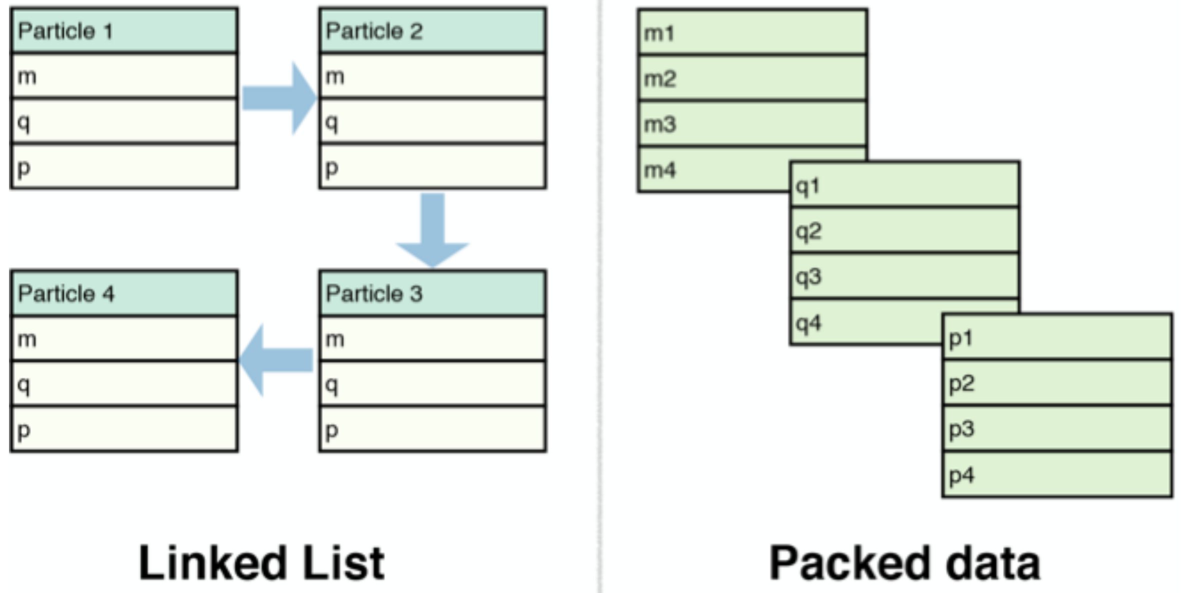
- Applying OpenMP results in data hazard.
- Atomic operation ensures the result, but hurts the performance.
- We propose a passive contribution (buffering then reduction) method to avoid the data hazard.


(a) Step 1. Write to the buffers within cell itself.

(b) Step 2. Pull the data from surrounding cells and then reduction.

Interim method of vectorization

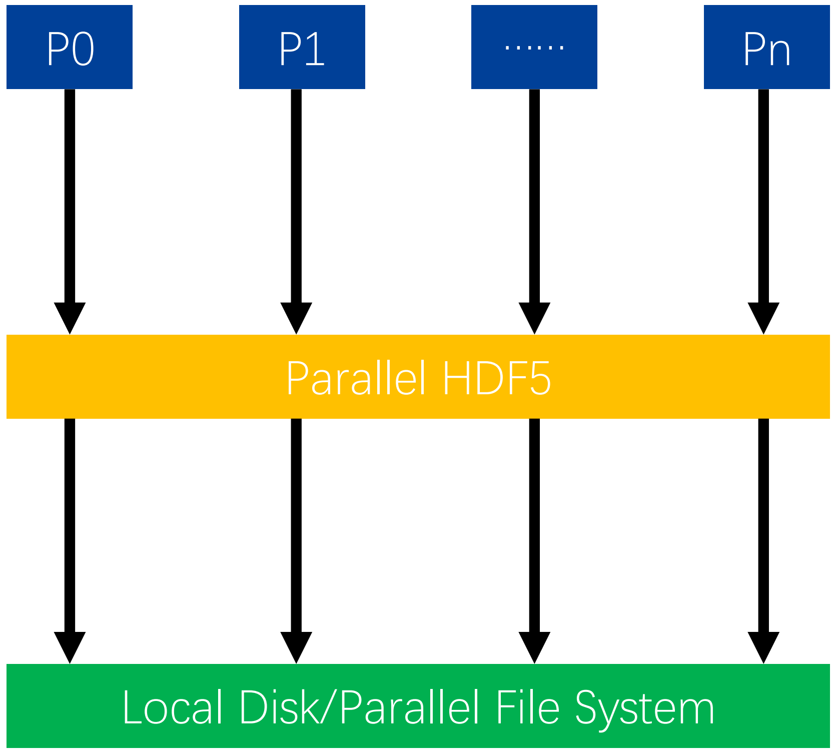
- Vectorization requires the AOS (array of structure) data structure.
- Packing 8 particles from AOS to SOA (structure of array) temporarily.



Parallel IO

- Rebuild the HDF5 library with

  “CC=mpiicpc ./configure --enable-parallel”
- Modifications on Save function
  - All MPI Processes write to the same datasheet
  - Only one write operation for one process



Dynamic load balancing optimization (Early stage)

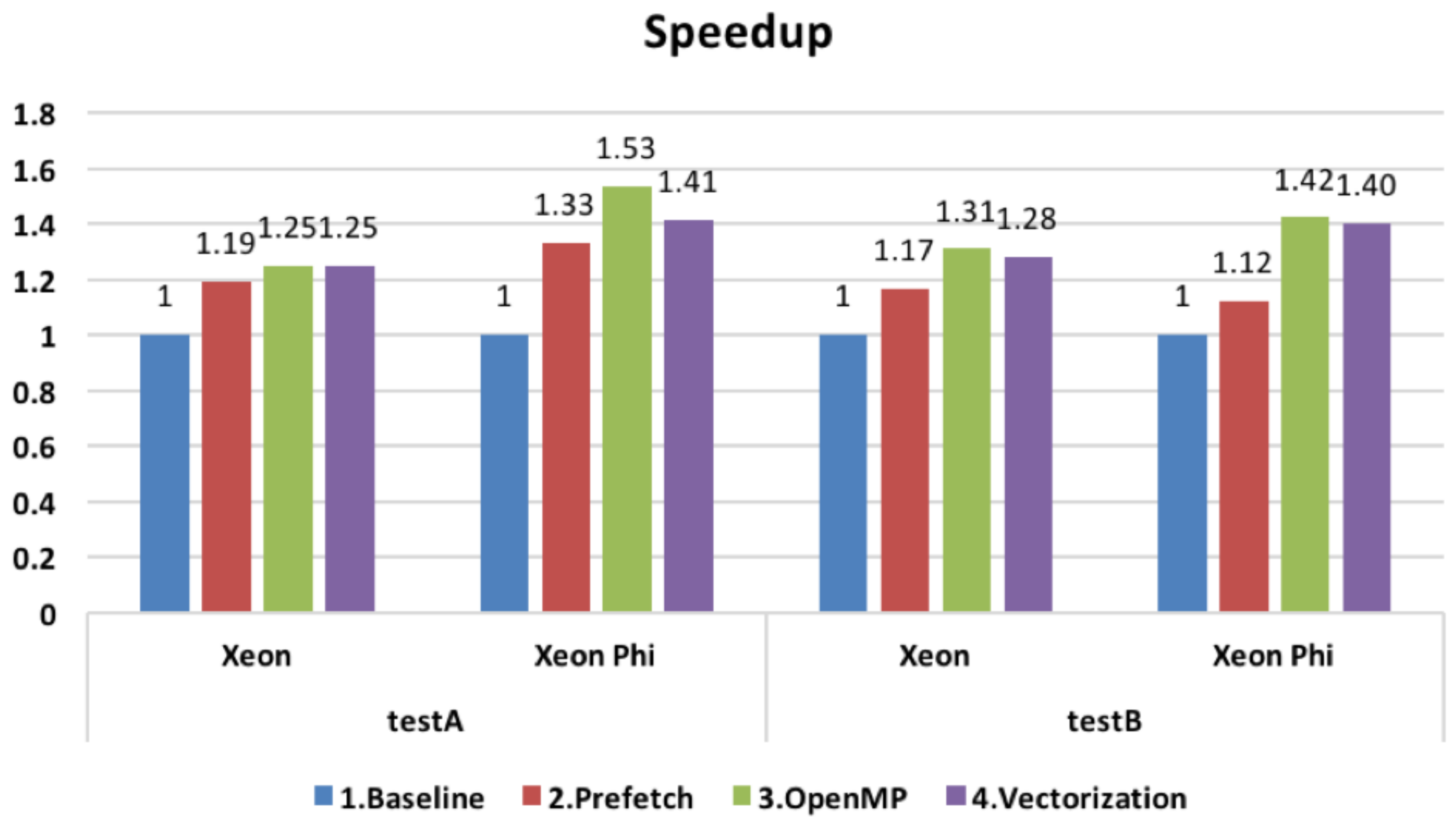
- The distribution of particles in the whole space is not uniform and changing during the iterations in most real test cases
- Evaluate the process load according to the computational time of a whole row/column as a reference
- Tune the computational load by exchanging the cells of whole rows/columns if the load imbalance threshold value is achieved

Outline

- Introduction
- Optimizations
- Results
- Conclusion

Performance is improved up to 1.53x on KNL by Compute-oriented Optimizations

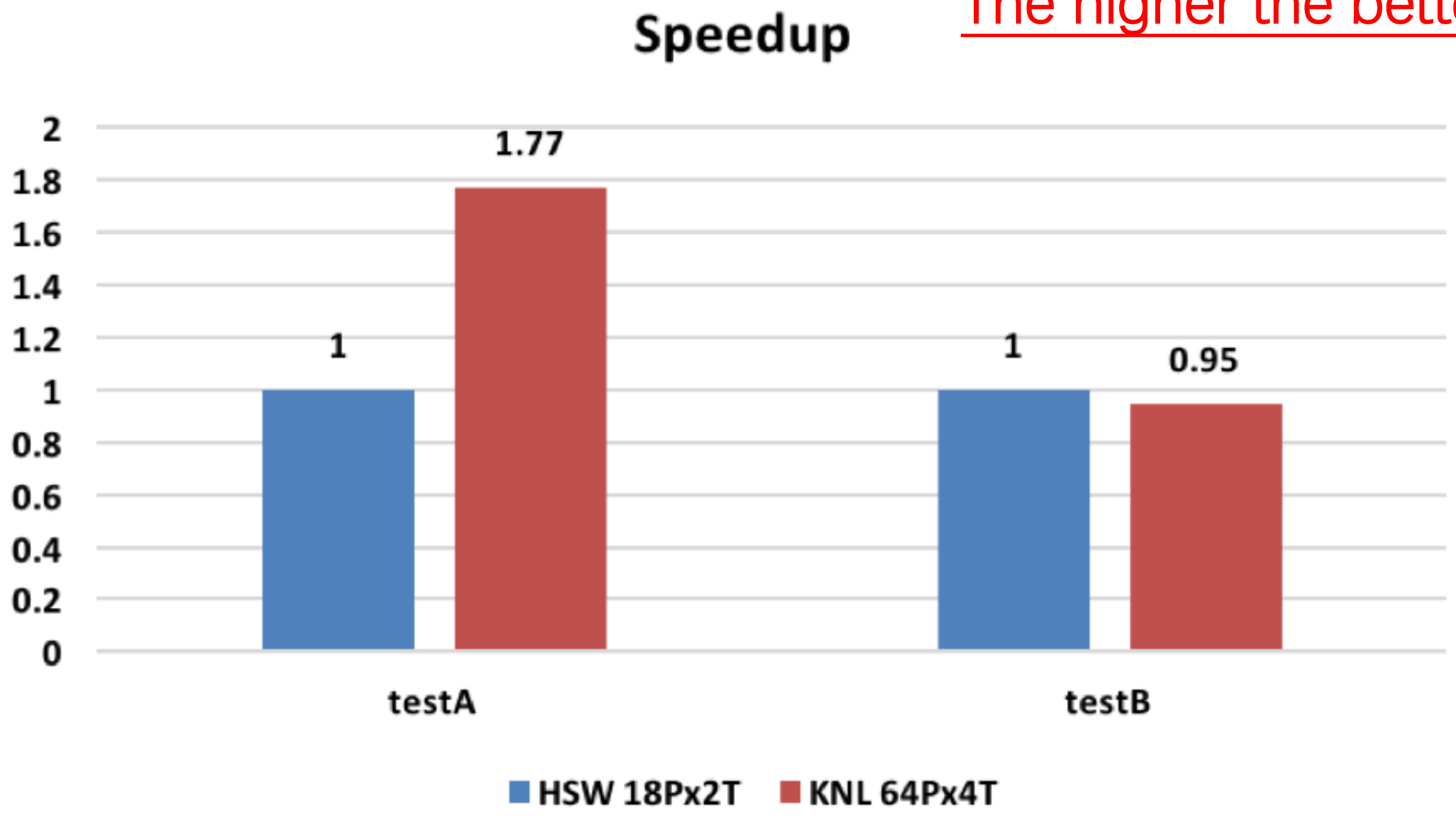
The higher the better



KNL : 7210P Flat Quadrant

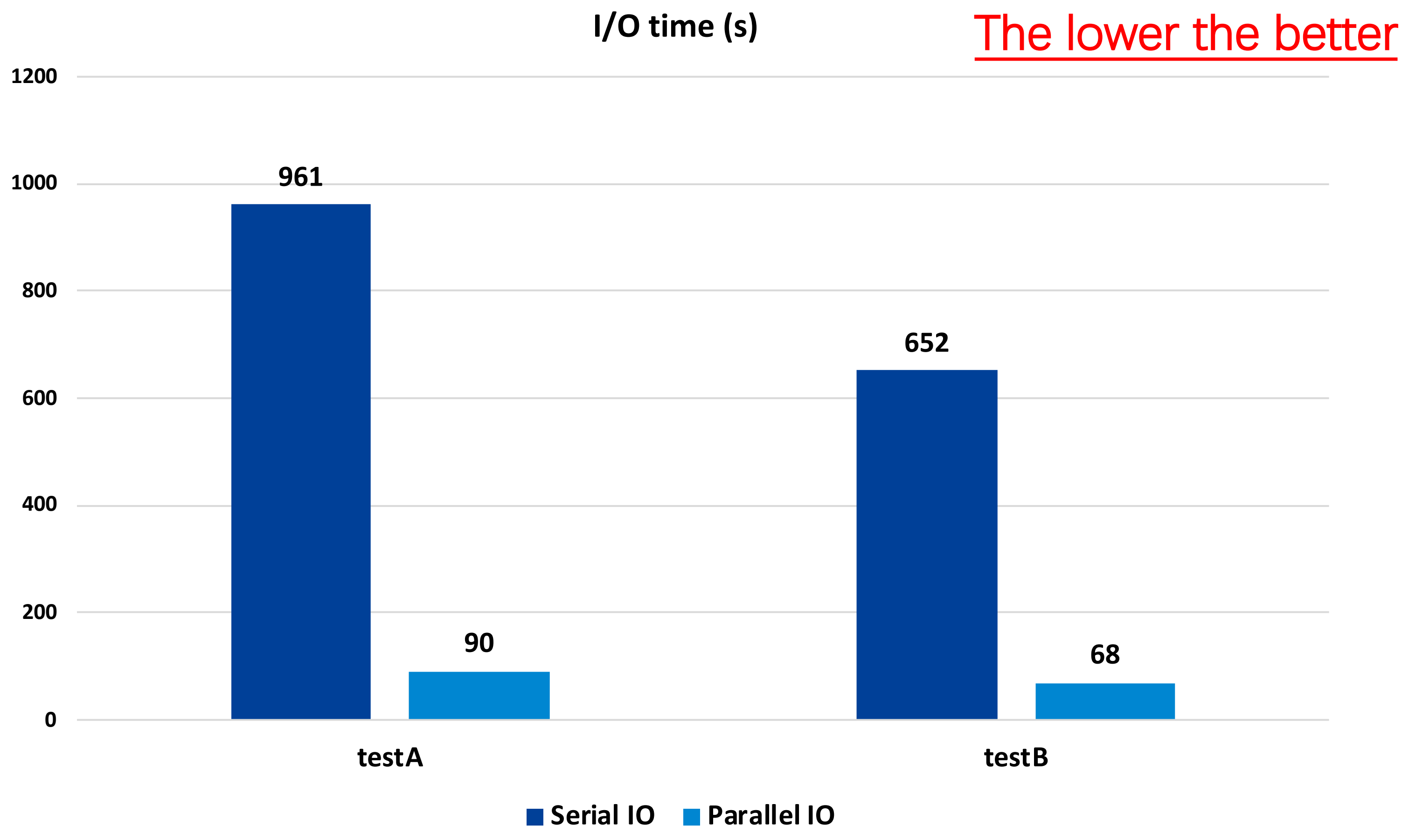
VLPL-S is 1.77X faster on KNL to two Haswell CPU

The higher the better



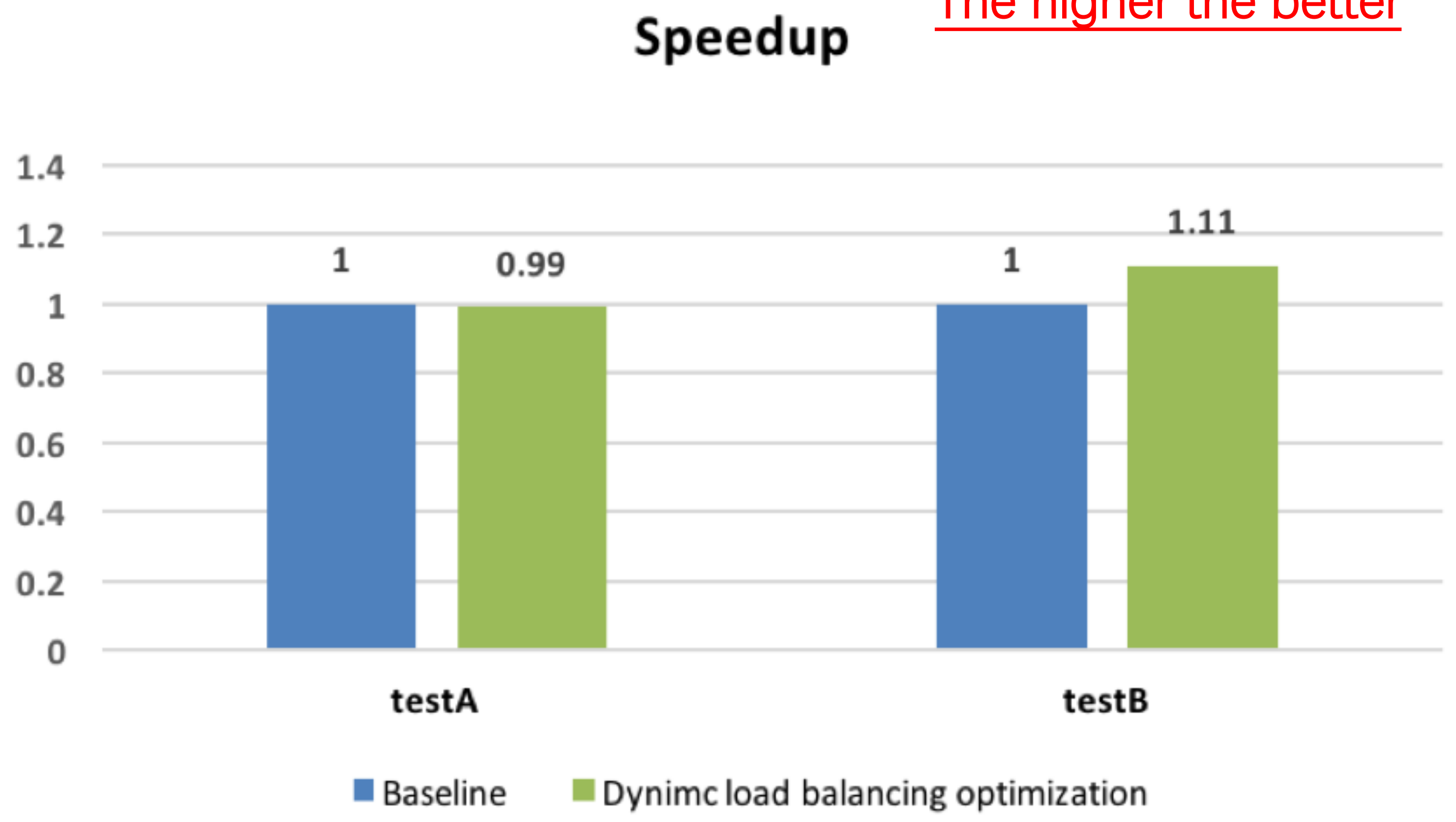
KNL : 7210P Flat Quadrant

I/O is boosted up to 10X



KNL　：7210P Flat Quadrant

Dynamic load balancing optimization improves 11% performance on KNL



The higher the better

KNL : 7210P Flat Quadrant

Outline

- Introduction
- Optimizations
- Results
- Conclusion

Conclusion

- The performance of VLPL-S on KNL 7210 is up to **1.77X faster** than it on a two-socket E5-2699v3 node

- Optimizations of VLPL-S mentioned above work both on Xeon and Xeon Phi

- MCDRAM greatly improves the performance for memory-bond applications

- Thread level parallelism helps reducing the communication overhead and improves the load balancing among threads, thus improves the performance, especially on KNL Clusters

Future work

- Completely rewrite the linked list to SOA and evaluate the performance of vectorization

- Improve workload optimization to further avoid load imbalance