Pic-Vert: A Particle-in-Cell Implementation for Multi-Core Architectures

Yann Barsamian

Université de Strasbourg
Laboratoire des sciences de l’Ingénieur, de l’Informatique et de l’Imagerie (ICube), CNRS UMR 7357
INRIA Nancy

October 2018
Today’s Outline

- Introduction
  - The context of this thesis.
  - Some computer architecture.
  - The Particle-in-Cell model.

- (Some) contributions
  - In the context of the standard algorithm.
  - With the help of a data structure crafted for our needs.
  - Comparison to the state-of-the-art.

- Conclusion
  - Other contributions.
  - Summary.
  - Perspectives.
Introduction
General Context: Controlled Thermonuclear Fusion

Step 1.

Sun image
Step 2.
Step 3. ITER\(^1\) tokamak\(^2\)

(also applicable in other contexts, e.g., astrophysics, where we have to model different particles / planets / ... that interact)

\(^1\)“The way” (in Latin) to produce energy (Cadarache, France)

\(^2\)Токамак: тороида́льная камера с магнитными катушками (toroidal chamber with magnetic coils)
## General Tool: Supercomputers

<table>
<thead>
<tr>
<th>Rank</th>
<th>Country</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>🇺🇸</td>
<td>2,282,544</td>
</tr>
<tr>
<td>2</td>
<td>🇨🇳</td>
<td>10,649,600</td>
</tr>
<tr>
<td>5</td>
<td>🇯🇵</td>
<td>391,680</td>
</tr>
<tr>
<td>6</td>
<td>🇨🇭</td>
<td>361,760</td>
</tr>
<tr>
<td>11</td>
<td>🇰🇷</td>
<td>570,020</td>
</tr>
<tr>
<td>13</td>
<td>🇮🇹</td>
<td>253,600</td>
</tr>
<tr>
<td>14</td>
<td>🇫🇷</td>
<td>561,408</td>
</tr>
</tbody>
</table>

During this thesis, we used:

- Occigen (ставил, 85,824 cores, Rank 70),
- Marconi (ставил, 54,432 cores, Rank 98),
- Curie (ставил, 77,184 cores, Rank 145), and
- Icps-gc-6 (2 × 10 cores) and this laptop (2 cores).

Source: June 2018 list of https://www.top500.org.
Kinetic Modeling with Particle-in-Cell (PIC) Methods

\[
\begin{align*}
\frac{\partial f}{\partial t} + \nabla \cdot \nabla_x f - \vec{E} \cdot \nabla \vec{v} f &= 0 \\
\nabla_x \vec{E} &= \rho = 1 - \int f \ d\vec{v} 
\end{align*}
\]

Vlasov

Poisson

- Distribution function \( f \): \( N \) numerical particles (red)
- Electric field \( \vec{E} \) and charge density \( \rho \): 3d grids (black)
\[
\begin{align*}
\frac{d \vec{x}}{dt} &= \vec{v} \quad \text{and} \quad \frac{d \vec{v}}{dt} = -\vec{E} \\
\nabla_x \vec{E} &= \rho = 1 - \int f \, d\vec{v} \\
\end{align*}
\]

- Vlasov characteristics
- Poisson

- Distribution function $f$: $N$ numerical particles (red)
- Electric field $\vec{E}$ and charge density $\rho$: 3d grids (black)
\[
\begin{align*}
\frac{d\mathbf{x}}{dt} &= \mathbf{v} \quad \text{and} \quad \frac{d\mathbf{v}}{dt} = -\mathbf{E} \\
\nabla_x \mathbf{E} &= \rho = 1 - \int f \, d\mathbf{v}
\end{align*}
\]

- Vlasov characteristics
- Poisson

- Distribution function \( f \): \( N \) numerical particles (red)
- Electric field \( \mathbf{E} \) and charge density \( \rho \): 3d grids (black)

- Physical effects on small scale (+ large scale)
- Noise (numerical errors when \( N \) is small)
- Frequent particle motion
Kinetic Modeling with Particle-in-Cell (PIC) Methods

\[
\begin{align*}
\frac{d\vec{x}}{dt} &= \vec{v} \\
\frac{d\vec{v}}{dt} &= -\vec{E} \\
\nabla \times \vec{E} &= \rho = 1 - \int f \, d\vec{v}
\end{align*}
\]

- Vlasov characteristics
- Poisson

- Distribution function \( f \): \( N \) numerical particles (red)
- Electric field \( \vec{E} \) and charge density \( \rho \): 3d grids (black)

- Physical effects on small scale (+ large scale)
  \( \Rightarrow \) increase \( ncx \times ncy \times ncz \)
  \( (1 \, 000 \times 1 \, 000 \times 1 \, 000) \)

- Noise (numerical errors when \( N \) is small)

- Frequent particle motion
Kinetic Modeling with Particle-in-Cell (PIC) Methods

\[
\begin{align*}
\frac{d\vec{x}}{dt} &= \vec{v} \\
\frac{d\vec{v}}{dt} &= -\vec{E}
\end{align*}
\]

Vlasov characteristics

\[
\nabla_x \vec{E} = \rho = 1 - \int f \ d\vec{v}
\]

Poisson

- Distribution function \( f \): \( N \) numerical particles (red)
- Electric field \( \vec{E} \) and charge density \( \rho \): 3d grids (black)

- Physical effects on small scale (+ large scale)
  \( \Rightarrow \) increase \( ncx \times ncy \times ncz \)
  \((1000 \times 1000 \times 1000)\)

- Noise (numerical errors when \( N \) is small)
  \( \Rightarrow \) increase \( \frac{N}{ncx \times ncy \times ncz} \)
  \((10000 \text{ to } 10000000)\)

- Frequent particle motion
Particle-in-Cell (PIC) Pseudo-Code

Initialization:
1. Initialize $N$ particles $\text{icell, dx, dy, dz, vx, vy, vz}$ of size $[N]$
2. Compute $\rho$ and $E$ $\text{rho, Ex, ey, ez}$ of size $[ncx][ncy][ncz]$

Algorithm:
3. Foreach time iteration do
4. If (condition) then
5. Sort the particles$^3$ $\mathcal{O}(N)$ counting sort
6. End If
7. Set all cells of $\rho$ to 0
8. Foreach particle do
9. Update the velocity $v^+ = -E \Delta t$
10. Update the position $x^+ = v \Delta t$
11. Deposit the charge on the nearest $\rho$ cells
12. End Foreach
13. Compute $E$ from $\rho$ $\text{FFT Poisson solver}$
14. End Foreach

$^3$Decyk, Karmesin, de Boer, & Liewer (1996)
Particle-in-Cell (PIC) Pseudo-Code

**Initialization:**
1. Initialize $N$ particles $\text{icell, d\{x,y,z\}, v\{x,y,z\}}$ of size $[N]$
2. Compute $\rho$ and $E$ $\text{rho, E\{x,y,z\}}$ of size $[ncx][ncy][ncz]$

**Algorithm:**
3. **Foreach** time iteration **do**
4.  **If** (condition) **then**
5.       Sort the particles$^3$
6.  **End If**
7.  Set all cells of $\rho$ to 0
8. **Foreach** particle **do**
9.       Update the velocity $^4$
10.      Update the position $^4$
11. Deposit the charge on the nearest $\rho$ cells $^4$
12. **End Foreach**
13. Compute $E$ from $\rho$ $^4$
14. **End Foreach**

$^3$Decyk, Karmesin, de Boer, & Liewer (1996)

$^4$Any difference in system hardware or software design or configuration may affect actual performance (-:
The Cloud-in-Cell\textsuperscript{5} method:

\[ \mathbf{v} = -E \Delta t \]

for (i = 0; i < N; i++) {
    vx[i] -= delta_t * (dx[i] * dy[i] * E[i_cell[i]].x_ne + (1.-dx[i]) * dy[i] * E[i_cell[i]].x_nw + (dx[i]) * (1.-dy[i]) * E[i_cell[i]].x_se + (1.-dx[i]) * (1.-dy[i]) * E[i_cell[i]].x_sw);
    vy[i] -= delta_t * (dx[i] * dy[i] * E[i_cell[i]].y_ne + (1.-dx[i]) * dy[i] * E[i_cell[i]].y_nw + (dx[i]) * (1.-dy[i]) * E[i_cell[i]].y_se + (1.-dx[i]) * (1.-dy[i]) * E[i_cell[i]].y_sw);
}

\textsuperscript{5}Birdsall & Fuss (1969)
Contributions (part I)


To sort or not to sort?

<table>
<thead>
<tr>
<th>Sort</th>
<th>Upd. v</th>
<th>Upd. x</th>
<th>Deposit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not sort</td>
<td>0.0</td>
<td>98.0</td>
<td>64.6</td>
<td>35.9</td>
</tr>
<tr>
<td>Sort every 100</td>
<td>3.6</td>
<td>78.3</td>
<td>64.4</td>
<td>25.6</td>
</tr>
<tr>
<td>Always sort</td>
<td>209.0</td>
<td>66.3</td>
<td>64.2</td>
<td>13.4</td>
</tr>
</tbody>
</table>

Execution time (in s). Test case: 200 000 000 particles, 128 × 128 grid, \( \Delta t = 0.1 \), 500 iterations. Architecture: Intel Broadwell, 18 cores, 76.8 GB/s.

Periodic sorting: better data locality, and shorter overall time: find the best frequency\(^6\).

Sorting at each iteration\(^7\): enhancement of the data locality & vectorization of the update velocities loop, but too costly.

Magic sorting that lasts longer\(^8\): needs less frequent sorting.

---

\(^6\)Marin, Jin, & Mellor-Crummey (2008)
\(^7\)Lanti, Tran, Jocksch, Hariri, Brunner, Gheller, & Villard (2016)
\(^8\)Barsamian, Hirstoaga, & Violard (2017); Barsamian, Hirstoaga, & Violard (2018)
## Overall Optimization Gains

<table>
<thead>
<tr>
<th></th>
<th>$T$ (s)</th>
<th>%</th>
<th>Acc. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline$^9$</td>
<td>120.4</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>+ Loop not-so-invariant-code-motion</td>
<td>113.4</td>
<td>5.8</td>
<td>5.8</td>
</tr>
<tr>
<td>+ Loop Fission</td>
<td>97.9</td>
<td>13.7</td>
<td>18.7</td>
</tr>
<tr>
<td>+ Redundant arrays $(E, \rho)^{10,11}$</td>
<td>94.0</td>
<td>4.0</td>
<td>21.9</td>
</tr>
<tr>
<td>+ Structure of Arrays $(particles)$</td>
<td>76.0</td>
<td>19.1</td>
<td>36.9</td>
</tr>
<tr>
<td>+ Space-filling curves $(E, \rho)$</td>
<td>72.6</td>
<td>4.5</td>
<td>39.7</td>
</tr>
<tr>
<td>+ Optimized update-positions</td>
<td>68.8</td>
<td>5.2</td>
<td>42.8</td>
</tr>
</tbody>
</table>

Total execution time, gains and accumulated gains, for a $128 \times 128$ grid, 50 million particles, 100 iterations simulation (sorting every 20 iterations). Architecture: Intel Haswell.

---

$^{10}$Bowers (2003)
$^{11}$Vincenti, Lobet, Lehe, Sasanka, & Vay (2016)
## Overall Optimization Gains

<table>
<thead>
<tr>
<th></th>
<th>T (s)</th>
<th>%</th>
<th>Acc. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline(^9)</td>
<td>120.4</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>+ Loop not-so-invariant-code-motion</td>
<td>113.4</td>
<td>5.8</td>
<td>5.8</td>
</tr>
<tr>
<td>+ Loop Fission</td>
<td>97.9</td>
<td>13.7</td>
<td>18.7</td>
</tr>
<tr>
<td>+ Redundant arrays ((E, \rho)^{10,11})</td>
<td>94.0</td>
<td>4.0</td>
<td>21.9</td>
</tr>
<tr>
<td>+ Structure of Arrays ((particles))</td>
<td>76.0</td>
<td>19.1</td>
<td>36.9</td>
</tr>
<tr>
<td>+ Space-filling curves ((E, \rho))</td>
<td>72.6</td>
<td>4.5</td>
<td>39.7</td>
</tr>
<tr>
<td>+ Optimized update-positions</td>
<td>68.8</td>
<td>5.2</td>
<td>42.8</td>
</tr>
</tbody>
</table>

Total execution time, gains and accumulated gains, for a 128 x 128 grid, 50 million particles, 100 iterations simulation (sorting every 20 iterations). Architecture: Intel Haswell.

**75 million** particles processed/second on one core.

---


\(^{10}\)Bowers (2003)

\(^{11}\)Vincenti, Lobet, Lehe, Sasanka, & Vay (2016)
Space-Filling Curves for $E$ and $\rho$: in 2d

- **Row-Major (Standard layout in C)**
  - L4D curve (Chatterjee, Jain, Lebeck, Mundhra, & Thottethodi, 1999)

- **Morton curve (Morton, 1966)**
  - Hilbert curve (Hilbert, 1891)
Space-Filling Curves for $E$ and $\rho$: in 3d

Row-Major (Standard layout in C)

L6D curve
(Barsamian, Hirstoaga, & Violard, 2018)

Morton curve (Morton, 1966)

Hilbert curve (Hilbert, 1891)
These layout functions are inappropriate for programs that access array elements randomly. [...] Even more than for the 4D layout, the Morton layout function is expensive to compute naively.
Chatterjee et al., 1999

This paper investigates using data and computation reorderings to improve [...] irregular applications.
Mellor-Crummey, Whalley, & Kennedy, 2001

As Morton-order representation [...] attracts more users because of its excellent block locality, the efficiency of these conversions becomes important.
Raman & Wise, 2007
Space-Filling Curves for $E$ and $\rho$: Results

<table>
<thead>
<tr>
<th></th>
<th>Update $v$</th>
<th>Update $x$</th>
<th>Deposit</th>
<th>Sort</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row-major</td>
<td>63.6</td>
<td>39.7</td>
<td>42.8</td>
<td>28.6</td>
<td>177</td>
</tr>
<tr>
<td>L4D</td>
<td>57.5</td>
<td>40.0</td>
<td>32.0</td>
<td>28.6</td>
<td>161</td>
</tr>
<tr>
<td>Morton</td>
<td>59.3</td>
<td>39.8</td>
<td>29.8</td>
<td>28.4</td>
<td>160</td>
</tr>
<tr>
<td>Hilbert</td>
<td>59.0</td>
<td>323.7</td>
<td>33.6</td>
<td>28.6</td>
<td>452</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Update $v$</th>
<th>Update $x$</th>
<th>Deposit</th>
<th>Sort</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row-major</td>
<td>92.6</td>
<td>55.3</td>
<td>31.5</td>
<td>21.4</td>
<td>202</td>
</tr>
<tr>
<td>L6D</td>
<td>85.5</td>
<td>55.5</td>
<td>29.9</td>
<td>20.9</td>
<td>193</td>
</tr>
<tr>
<td>Morton</td>
<td>89.4</td>
<td>56.7</td>
<td>33.5</td>
<td>19.8</td>
<td>200</td>
</tr>
<tr>
<td>Hilbert</td>
<td>87.3</td>
<td>244.4</td>
<td>29.2</td>
<td>20.3</td>
<td>382</td>
</tr>
</tbody>
</table>

Time spent in the different loops (in seconds) when using several space-filling curves.

Top: 2d, 512 x 512 grid, 1 billion particles, 100 iterations (sorting every 20 iterations).

Bottom: 3d, 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations).
// Update-velocities
for (size_t i = 0; i < num_particle; i++) {
    vx[i] -= delta_t * E_x_part;  // Reads array Ex
    vy[i] -= delta_t * E_y_part;  // Reads array Ey
}
void poisson_solver([...] double** Ex, double** Ey) {
    [...] 
    for (size_t i = 0; i < ncx; i++) {
        for (size_t j = 0; j < ncy; j++) {
            Ex[i][j] *= delta_t;
            Ey[i][j] *= delta_t;
        }
    }
}

// Update-velocities
for (size_t i = 0; i < num_particle; i++) {
    vx[i] -= E_x_part; // Reads array Ex
    vy[i] -= E_y_part; // Reads array Ey
}
Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.
Strong Scaling on 24 Cores, With Loop Fission

Memory Bandwidth (GB / s)

- Update v
- Update x
- Deposit
- Sort
- Stream

Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.

---

Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake.

Strong scaling: 64 x 64 x 64 grid, 1 billion particles, 100 iterations (sorting every 10 iterations). Architecture: Intel Skylake. Strip-mined is 6.7% slower on 1 core but 12% faster on 24 cores.
Roofline Model\textsuperscript{14} on Intel Haswell (12 Cores)

- Baseline (1): missing computational efficiency.
- Loop fission (2): missing memory efficiency.
- Loop strip-mining (3): the best of two worlds.

\textsuperscript{14}Williams, Waterman, \& Patterson (2009)
\textsuperscript{15}McCalpin (1995) - Code v5.10 (2013)
Roofline Model on Intel Haswell (12 Cores)

- Baseline (1): missing computational efficiency.
- Loop fission (2): missing memory efficiency.
- Loop strip-mining (3): the best of two worlds.

Can we do better?

---

Williams, Waterman, & Patterson (2009)

Roofline Model\textsuperscript{14} on Intel Haswell (12 Cores)

- Baseline (1): missing computational efficiency.
- Loop fission (2): missing memory efficiency.
- Loop strip-mining (3): the best of two worlds.

Can we do better?

\textsuperscript{14}Williams, Waterman, & Patterson (2009)
\textsuperscript{15}McCalpin (1995) - Code v5.10 (2013)
Contributions (part II)

DOI: 10.1007/978-3-319-78024-5_13

DOI: 10.1007/978-3-319-96983-1_53

URL: https://doi.org/10.6084/m9.figshare.6391796.
To sort or not to sort?

<table>
<thead>
<tr>
<th></th>
<th>Sort</th>
<th>Upd. v</th>
<th>Upd. x</th>
<th>Deposit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not sort</td>
<td>0.0</td>
<td>98.0</td>
<td>64.6</td>
<td>35.9</td>
<td>199.0</td>
</tr>
<tr>
<td>Sort every 100</td>
<td>3.6</td>
<td>78.3</td>
<td>64.4</td>
<td>25.6</td>
<td>177.0</td>
</tr>
<tr>
<td>Always sort</td>
<td>209.0</td>
<td>66.3</td>
<td>64.2</td>
<td>13.4</td>
<td>353.0</td>
</tr>
</tbody>
</table>

Execution time (in s). Test case: 200 000 000 particles, 128 × 128 grid, \( \Delta t = 0.1 \), 500 iterations. Architecture: Intel Broadwell, 18 cores, 76.8 GB/s.

Periodic sorting: better data locality, and shorter overall time: find the best frequency.

Sorting at each iteration: enhancement of the data locality & vectorization of the update velocities loop, but too costly.

Magic sorting that lasts longer: needs less frequent sorting. Efficient data structure to keep particles sorted\(^{16}\): avoid the sorting step.

\(^{16}\)Durand, Raffin, & Faure (2012); Nakashima, Summura, Kikura, & Miyake (2017); Barsamian, Charguéraud, & Ketterlin (2017)
struct chunk {
    struct chunk* next;  // 0<=size<=K
    int size;
    float dx[K], dy[K], dz[K];
    double vx[K], vy[K], vz[K];
} chunk;

struct { chunk* front, back; } bag;
The Eight-Colors Algorithm

8 phases to tame the number of data races when moving particles.

Kong, Huang, Ren, & Decyk (2011)
Particles moving more than half a tile away require special care.

Kong, Huang, Ren, & Decyk (2011)
Chunk Bags: Particle Arrays

chunkbag particles[nbCells] // nbCells = ncx*ncy*ncz

\[\begin{array}{cccccc}
X & X & X & X & \cdots & X \\
\end{array}\]

particles with cell identifier 1

\[\begin{array}{cccccc}
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & X \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & X \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & X \\
\end{array}\]

particles with cell identifier 0

chunkbag particlesNextPrivate[nbCells],
particlesNextShared[nbCells]

- particlesNextPrivate[i] receives particles moving to a nearby cell \(i\): no atomic operation required.
- particlesNextShared[i] receives particles moving to a remote cell \(i\): atomic push used.
- particles[i] at the next time step is obtained by merging the two.
void bag_push_serial([...]) {
    chunk* c;
    int id;

    c = b->front; // The front chunk.

    id = c->size++; // The id of the last free cell.

    c->dx[id] = dx; c->dy[id] = dy; c->dz[id] = dz;
    c->vx[id] = vx; c->vy[id] = vy; c->vz[id] = vz;
    if (id == CHUNK_SIZE - 1) // The chunk is now full.
        add_front_chunk(b, thread_id);
}

chunk* c;
int id;
while (true) {
    // Until success.
    c = b->front; // The front chunk.
    #pragma omp atomic capture
    id = c->size++; // The id of the last free cell.
    if (id < CHUNK_SIZE) {
        // The chunk was not full.
        c->dx[id] = dx; c->dy[id] = dy; c->dz[id] = dz;
        c->vx[id] = vx; c->vy[id] = vy; c->vz[id] = vz;
        if (id == CHUNK_SIZE - 1) // The chunk is now full.
            add_front_chunk(b, thread_id);
        return;
    } else {
        // The chunk was full.
        #pragma omp atomic write
        c->size = CHUNK_SIZE;
        while (atomic_read(&b->front) == c) {}
    }
}
Chunk Bags: Merge Operation

![Diagram of Chunk Bags Merger]

- **Chunk Bags**: Merge Operation
- **Pic-Vert**: Parallel and Optimized Particle-in-Cell Implementation
- **Y. Barsamian (Strasbourg)**
Chunk Bags: Merge Operation

Y. Barsamian (Strasbourg)

Pic-Vert: Parallel and Optimized Particle-in-Cell Implementation

31/10/2018
Upper bound on the number of chunks: \( \lceil N/K \rceil + 4 \cdot \text{nbCells} \).

All chunks allocated at initialization (no dynamic malloc/free).
Roofline Model\textsuperscript{19} on Intel Skylake (24 Cores)

- Peak floating-point performance (1.612 TFlops/s)
- Peak memory bandwidth (theoretical: 128 GB/s)
- Peak memory bandwidth (Stream\textsuperscript{18}: 98.2 GB/s)

Operational Intensity (Flops/byte)

- Attainable GFlops/s

\begin{itemize}
  \item 3d Pic-Vert
  \item 2d3v Pic-Vert
\end{itemize}

\textsuperscript{18}McCcalpin (1995) - Code v5.10 (2013)
\textsuperscript{19}Williams, Waterman, & Patterson (2009)
Comparison of Pic-Vert to Other Implementations

Different implementations on different architectures: cores, memory bandwidth in GB/s, number of particles processed by second (absolute and normalized w.r.t. memory bandwidth). Top: CPUs. Bottom: accelerators (GPUs, MICs).

<table>
<thead>
<tr>
<th>Implem.</th>
<th>Architecture</th>
<th>Cores</th>
<th>M.B.</th>
<th>Part./s</th>
<th>Norm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPIC</td>
<td>IBM PowerXCell 8i</td>
<td>9</td>
<td>204.8</td>
<td>$173 \cdot 10^6$</td>
<td>0.85</td>
</tr>
<tr>
<td>OSIRIS</td>
<td>Intel Xeon E5-2680</td>
<td>8</td>
<td>51.2</td>
<td>$134 \cdot 10^6$</td>
<td>2.62</td>
</tr>
<tr>
<td>ORB5</td>
<td>Intel Xeon E5-2670</td>
<td>8</td>
<td>51.2</td>
<td>$69 \cdot 10^6$</td>
<td>1.35</td>
</tr>
<tr>
<td>PICADeOR</td>
<td>Intel Xeon E5-2697 v3</td>
<td>14</td>
<td>68</td>
<td>$127 \cdot 10^6$</td>
<td>1.87</td>
</tr>
<tr>
<td>GTC-P</td>
<td>Intel Xeon E5 2692 v2</td>
<td>12</td>
<td>59.7</td>
<td>$100 \cdot 10^6$</td>
<td>1.68</td>
</tr>
<tr>
<td>PIConGPU</td>
<td>Intel Xeon E5-2698 v3</td>
<td>16</td>
<td>68</td>
<td>$111 \cdot 10^6$</td>
<td>1.63</td>
</tr>
<tr>
<td>Pic-Vert</td>
<td>Intel Xeon Platinum 8160</td>
<td>24</td>
<td>128</td>
<td>$740 \cdot 10^6$</td>
<td>5.78</td>
</tr>
<tr>
<td>Pic-Vert</td>
<td>Intel Xeon E5-2690 v3</td>
<td>12</td>
<td>68</td>
<td>$374 \cdot 10^6$</td>
<td>5.49</td>
</tr>
<tr>
<td>PIConGPU</td>
<td>NVIDIA Tesla GK210</td>
<td>2496</td>
<td>480</td>
<td>$336 \cdot 10^6$</td>
<td>0.70</td>
</tr>
<tr>
<td>ORB5</td>
<td>NVIDIA Tesla K20X</td>
<td>2688</td>
<td>250.0</td>
<td>$177 \cdot 10^6$</td>
<td>0.71</td>
</tr>
<tr>
<td>PICADeOR</td>
<td>Intel Xeon Phi 7250 (KNL)</td>
<td>68</td>
<td>115.2</td>
<td>$298 \cdot 10^6$</td>
<td>2.59</td>
</tr>
<tr>
<td>EMSES</td>
<td>Intel Xeon Phi 7250 (KNL)</td>
<td>68</td>
<td>115.2</td>
<td>$1300 \cdot 10^6$</td>
<td>11.3</td>
</tr>
</tbody>
</table>
A parameter that affects the efficiency of any PIC simulation is $p$, the fraction of particles that cross cell boundaries.

Most previous work focus on simulations with a low value of $p$.

- **VPIC**: the deposit step is only vectorized on 4 particles when none of those particles cross cell boundaries\(^{20}\).
- **UPIC**: results are shown for $p$ up to 12%.
- **EMSES**: the mechanism is shown to be efficient only when $p$ is low (1–2% in this paper).

In our test cases, $p$ reaches up to 99% (and particles move to arbitrarily far away cells).

By design, $p$ has only little impact on our performance.

\[^{20}\text{With } p = 5\%, \text{ this happens only } 0.95^4 = 81\% \text{ of the time.}\]
Fast-Moving Particles

Experiments with different particle velocities, where the initial velocities follow the sum of two Gaussian distributions, like in the bump-on-tail instability:

\[ f_0(x, vx, vy, vz) = g(vx) \cdot g(vy) \cdot g(vz), \]

with

\[ g(w) = \frac{1}{\sqrt{2\pi} v_{th}} \left( p_{drift} \exp \left( -\frac{(w-v_{drift})^2}{2v_{th}^2} \right) + (1 - p_{drift}) \exp \left( -\frac{w^2}{2v_{th}^2} \right) \right). \]

- up to 4.4% of “fast-moving particles” (more than 2 cells away),
- up to 3.7% of possible conflicts\(^{21}\),
- if processed sequentially (EMSES): 85% slowdown on 24 cores\(^{22}\),
- when processed with our shared bags: only 7.0% slowdown.

\(^{21}\)Not all fast-moving particles go out of the “extended tile” — consider a particle on the far left of the tile moving 3 cells to the right.

\(^{22}\)Let \( t \) denote the single-core execution time. Assume 3.7% of sequential execution, and 96.3% using 24 cores. The parallel execution time is:

\[ 0.037t + 0.963t/24 = 1.85t/24. \]
Conclusion
Validation: 2d3v Electron Hole Test Case

- 64 billion particles,
- grid of size $512 \times 512$,
- time step 0.1,
- spatial domain $[0, 32]^2$,
- magnetic field 0.2.

Snapshots of $\rho$ at $t=0$ (top right), 20 (bottom left), and 40 (bottom right).

---

$^{23}$Muschietti, Roth, Carlson, & Ergun (2000)
Other Contributions

- Full advantage of vectorization (SIMD)
- Design of other variants for our algorithm with chunk bags
- Efficient 2d semi-Lagrangian implementation
- A common framework for Particle-in-Cell and semi-Lagrangian methods
- New 2d test cases with their theoretical analyses


Contributions

- Particles sorted at all time with low memory overhead (4 \cdot \text{nbCells} extra chunks, lowest in the state-of-the-art)
- Optimal memory bandwidth usage: each particle is loaded from/written to main memory only once per iteration
- Full advantage of SIMD
- Efficient handling of fast particles

Results on Intel Skylake, 24 cores, 128 GB/s

- 740 million particles / second in 3d
- 55% of the maximum bandwidth

Comparison to state-of-the-art implementations thanks to a new metric
Future Outlooks

- Test our ideas when solving the Vlasov–Maxwell equations
- Test Pic-Vert on Many Integrated Core (MIC) architecture (more cores)
- Investigate the use of chunks in domain decomposition (distributed memory parallelism)
- Collaborate with other teams
http://www.barsamian.am/Pic-Vert/