Parallel Particle-in-Cell Algorithm

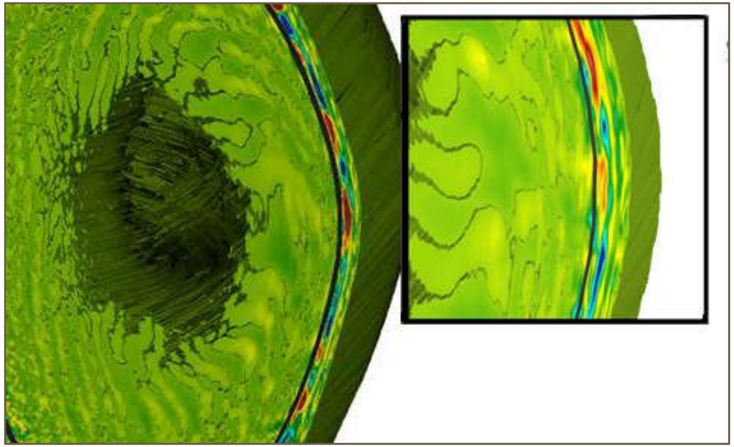
CS 205 - Final Presentation

Kevin Howarth, Aditi Memani, Hari Raval, Taro Spirig

Motivation

Background

- **Particle in Cell (PIC) is an attractive approach for simulating particle trajectories**
  - **N-body** → **$O(N^2)$** due to considering binary interactions between all particles
  - **PIC** → **$O(N)$** due to performing work on each particle only once
- **Princeton Plasma Physics Laboratory develops XGC**
  - PIC research code
  - Informs design of new fusion systems
  - Planned for use on exascale computing architectures



A property heatmap inside a tokamak simulated by XGC
Source: insidehpc.com

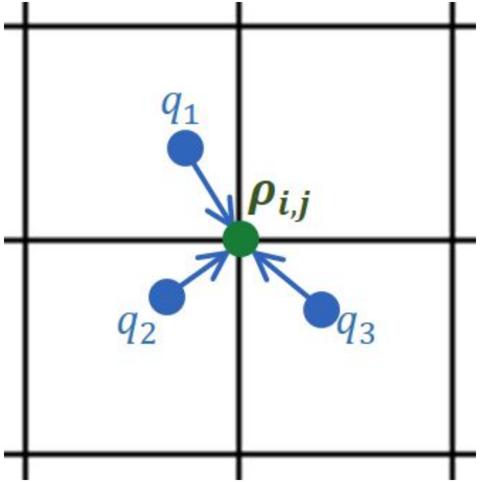
Scientific Goals

- **Demonstrate capability of our PIC code to simulate plasma behavior at realistically large input values**
    - Handle a large number of gridpoints → reduce spatial discretization error
    - Simulate at small timesteps → accurately resolve high frequency plasma behavior
- **Serve as a proof of concept for accurately representing plasma phenomena**
    - 10^{18} particles / m^3 in real-world fusion problems

Algorithm

PIC Algorithm Overview

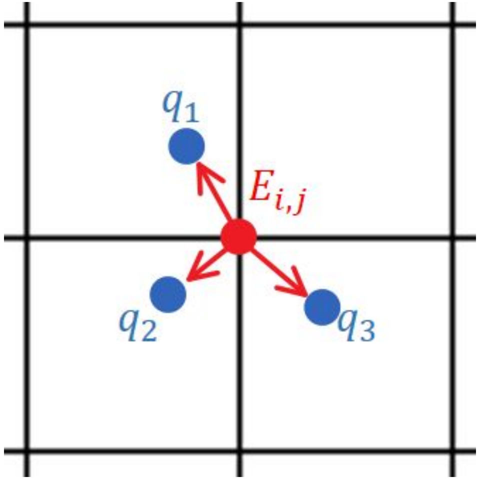
1. Interpolate from Particles to Mesh



2. Solve Discrete Poisson Equation on Mesh

$$\nabla^2 \phi = -\rho, \qquad \nabla \phi = E$$

3. Interpolate from Mesh back to Particles



4. Time-step Particle Locations

$$\frac{d\overline{v}}{dt} = qE(\overline{x}), \qquad \frac{d\overline{x}}{dt} = \overline{v}$$

Data Structures

Particle: **Not NUMA aware**

```
Particle() {

        std :: vector < float > xx;
        std:: vector < float > xy;
        std:: vector < float > vx;
        std:: vector < float > vy;
        std:: vector < float > Ex;
        std:: vector < float > Ey;
}
```

P_1	P_2	P_3	P_4	P_5	...	P_n

$\in \mathbb{R}^{N_p}$

Particle Particles = new Particle[N_p]();*

Data Structures

Particle: Not NUMA aware

```
Particle() {

        std :: vector < float > xx;
        std:: vector < float > xy;
        std:: vector < float > vx;
        std:: vector < float > vy;
        std:: vector < float > Ex;
        std:: vector < float > Ey;
}
```

Particle: NUMA aware

```
Particle(int N_tin) {

        Nt = N_tin;
        xx = new float [Nt];
        xy = new float [Nt];
        vx = new float [Nt];
        vy = new float [Nt];
        Ex = new float [Nt];
        Ey = new float [Nt];
}
```

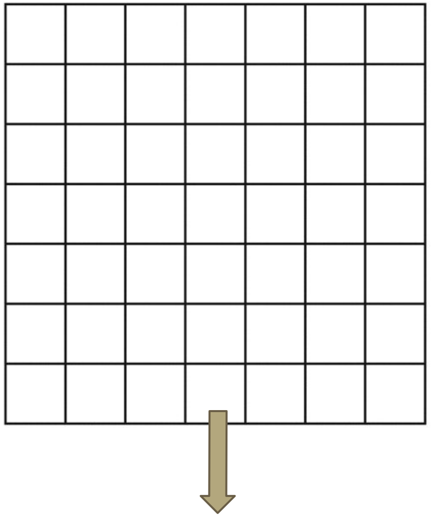
P_1	P_2	P_3	P_4	P_5	...	P_n

$\in \mathbb{R}^{N_p}$

Particle Particles = new Particle[N_p]();*

Data Structures

Computational Mesh



Charge	**Potential**	**Electric Field**

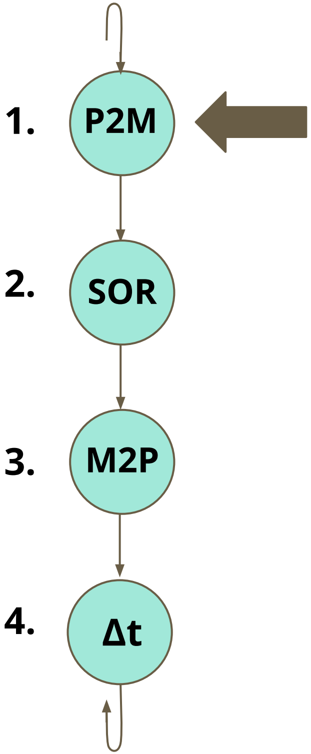
$\in \mathbb{R}^{N_g \times N_g}$

Eigen:: Matrix qGrid, pGrid, eGrid;

Parallel Implementation

Step 1: Particle-to-Mesh Interpolation

Algo. Steps



Parallel Implementation of P2M

- P2M reads from particles and writes to (shared) mesh
  - Implement thread-private grids and custom reduction
  - Parallelize over number of particles

$$\phi_{M4}(x) = \begin{cases} 1 - \frac{5}{2}|x|^2 + \frac{3}{2}|x|^2 & \text{if} \quad |x| \leq 1 \\ \frac{1}{2}(2 - |x|)^2(1 - |x|) & \text{if} \quad 1 < |x| \leq 2 \\ 0 \quad 2 \leq |x| \end{cases}$$

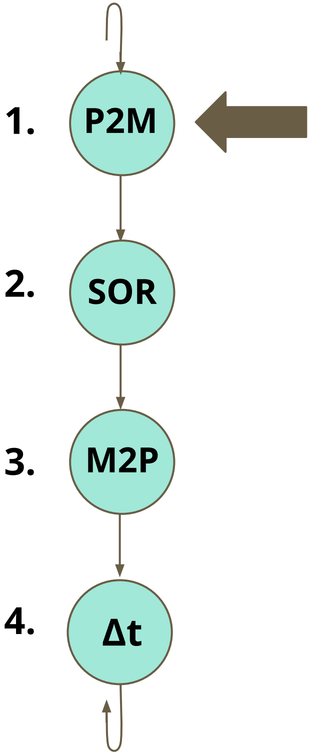
M4 Interpolation Kernel

$$\rho_d = q_p \times \phi_{M4}(x_{p,d}) \times \phi_{M4}(y_{p,d})$$

Rule to update grid point "d" with signature from particle "p"

Step 1: Particle-to-Mesh Interpolation

Algo. Steps



Parallel Implementation of P2M

- P2M reads from particles and writes to (shared) mesh
  - Implement thread-private grids and custom reduction
  - Parallelize over number of particles

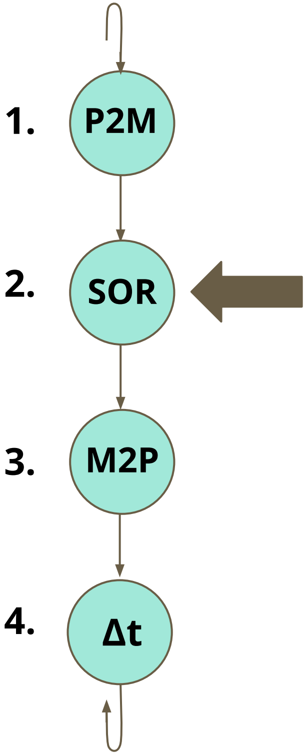
```
/*--------------------------------------------------------------------------------------------*/
// declare a custom reduction which performs addition of Eigen::Matrix objects
#pragma omp declare reduction(MatrixPlus: Eigen::Matrix: omp_out=omp_out+omp_in) \
                    initializer(omp_priv=Eigen::Matrix::Zero(omp_orig.rows(),omp_orig.cols()))
/*--------------------------------------------------------------------------------------------*/
```

```
// perform the custom reduction to add Eigen::Matrix objects
#pragma omp parallel for reduction(MatrixPlus: chargeGrid)
    for (unsigned int p = 0; p < N_p; p++) {
        P2M_M4_kernel(particles[p], chargeGrid);
    }
}
```

Custom reduction definition and usage

Step 2: Successive Over-Relaxation

Algo. Steps



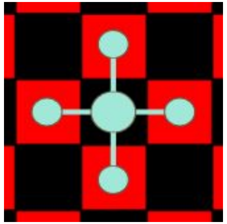
Parallel Implementation of SOR

- SOR corresponds to stencil:
  - Read from Self and Neighbors, Write to Self
- Parallelization over number of grid points in Red-Black sweeps

$$V_{i,j}^{k+1} = (1-\omega^k)V_{i,j}^k + \omega^k \frac{V_{i-1,j}^k + V_{i+1,j}^k + V_{i,j-1}^k + V_{i,j+1}^k + h^2\rho_{i,j}}{4}$$

SOR update rule for grid point (i, j) in iteration "k"

Example stencil for update during black sweep



Step 2: Successive Over-Relaxation

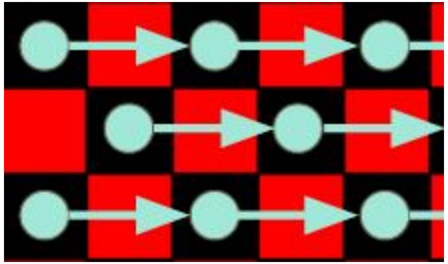
Algo. Steps

1. P2M
2. SOR
3. M2P
4. Δt

Parallel Implementation of SOR

```
// update "BLACK" nodes in "red-black" ordering scheme
#pragma omp parallel for schedule(static)
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n - 1; j += 2) {
                V[i][j] = ... // SOR Stencil Here
            }
        }
```

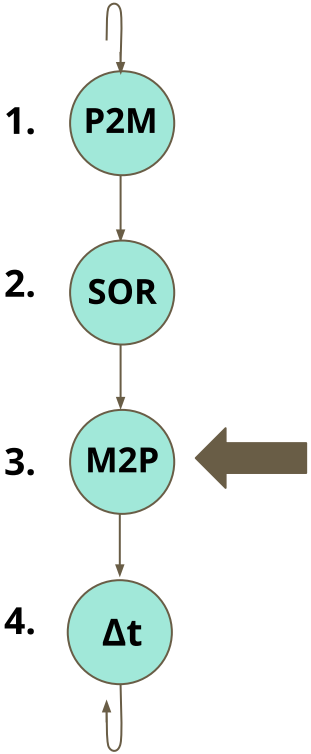
Parallelization of black sweep



Visualization of parallel black sweep + importance of memory layout!

Step 3: Mesh-to-Particle Interpolation

Algo. Steps



Parallel Design of M2P

- M2P reads from shared mesh and writes to particles
- Parallelize over the number of particles

$$\vec{E}_p = \sum_{d \in \mathbb{D}_{M4}} \vec{E}_d \times \phi_{M4}(x_{p,d}) \times \phi_{M4}(y_{p,d})$$

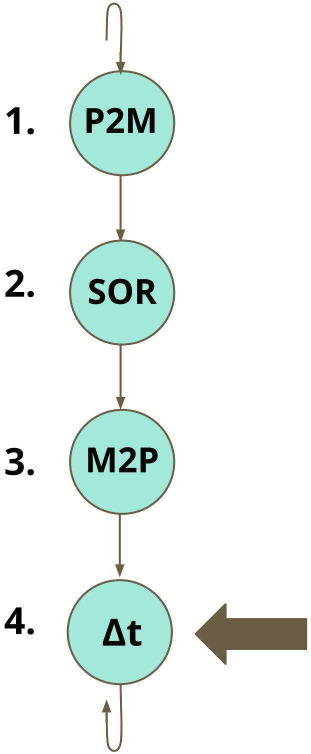
Rule to update particle "p" with signature from surrounding grid points $d \in \mathbb{D}_{M4}$

```
#pragma omp parallel for schedule(static)
    for (unsigned int p = 0; p < N_p; p++) {
            M2P_M4_kernel(particles[p], electricFieldGrid);
        }
    }
}
```

Parallelization of M2P interpolation

Step 4: Time-stepping

Algo. Steps



Parallel Design of Time-stepping

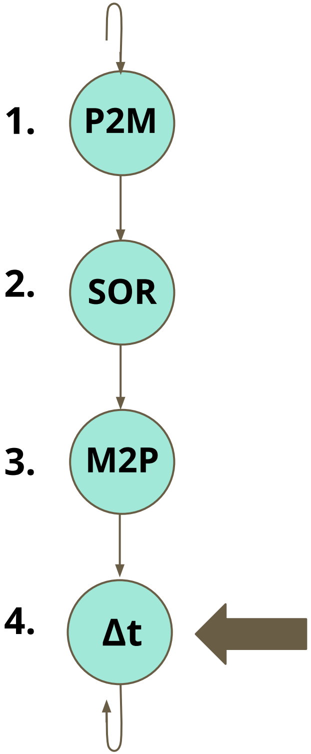
- Time-step using Leapfrog Method
  - Read from and write to same particle
  - Independent per particle →parallelize over number of particles

$$\vec{v}_{t+1/2} = \vec{v}_{t-1/2} + \Delta t \frac{q}{m} \vec{E}(\vec{x}_t)$$
$$\vec{x}_{t+1} = \vec{x}_t + \Delta t \vec{v}_{t+1/2}$$

Leapfrog Method rule to update a given particle's kinematic data

Step 4: Time-stepping

Algo. Steps



Parallel Design of Time-stepping

- Possible load imbalance when enforcing periodic wraparounds
- Address via dynamic scheduling with large chunk size

```
#pragma omp parallel for schedule(dynamic, 500)
    for (unsigned int p = 0; p < N_p; p++) {
        // timestep the velocity
        particles[p].vx[ti + 1] = particles[p].vx[ti] + constantFactorV * particles[p].Ex[ti];
        particles[p].vy[ti + 1] = particles[p].vy[ti] + constantFactorV * particles[p].Ey[ti];

        // timestep the position
        float xNew = particles[p].x[ti] + constantFactorX * particles[p].vx[ti + 1];
        float yNew = particles[p].y[ti] + constantFactorX * particles[p].vy[ti + 1];

        // enforce periodic boundary condition
        while (particleOutsideGrid) {
            periodicWraparound_x(xNew, particleOutsideGrid);
            periodicWraparound_y(yNew, particleOutsideGrid);
        }

        // write the new particle position
        particles[p].x[ti + 1] = xNew;
        particles[p].y[ti + 1] = yNew;
    }
```

Parallel Results

Problem Size

Small Case

10^6 particles (N_p)
400 grid points (N_g)
100 timesteps (N_t)

Large Case

3×10^7 particles (N_p)
1500 grid points (N_g)
100 timesteps (N_t)

Memory Requirement

$$\mathrm{Mem}(N_p, N_g, N_t) = 4 \times (N_p \times N_t \times (2 + 2 + 2) + N_g^{\,2} \times (1 + 1 + 2)) \times 10^{-9}$$

Note: All performance benchmarks and scaling analysis run on 1 Intel Broadwell node (~126 GB memory) with the large problem size

Problem Size

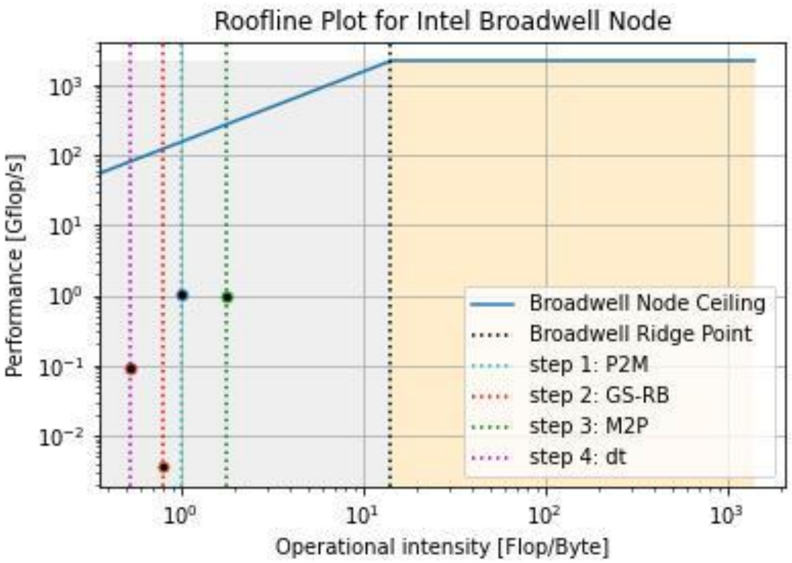
Problem Size	“Small” Case	“Large” Case
Wall clock time [hours]	0.03	1.5
Memory per node [GB]	2.4	72.1

Our “large” problem size requires 72.1 GB which is less than the Intel Broadwell node capacity, and is still computationally reasonable

Roofline Analysis: Sequential

- All four kernels are memory bound
- Gauss-Seidel - largest bottleneck with performance < 0.01 Gflop/s
- Focus on analyzing *M2P,* because it has highest ceiling for improvement

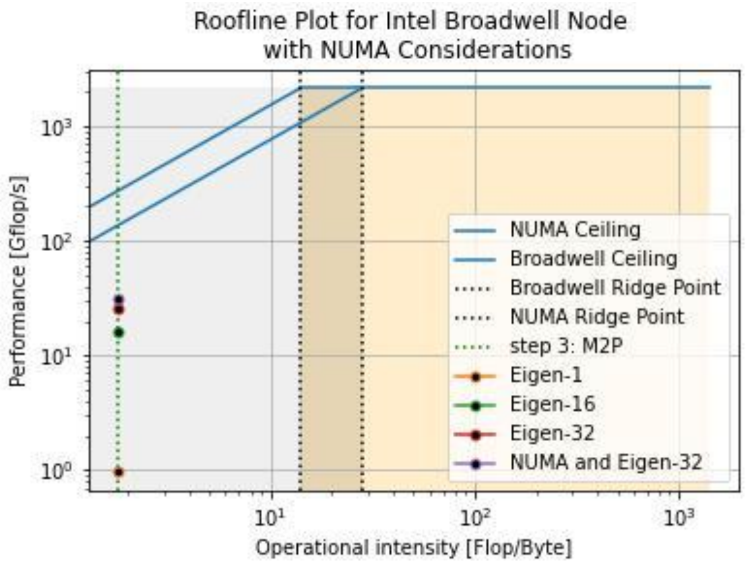
Roofline for Sequential Code



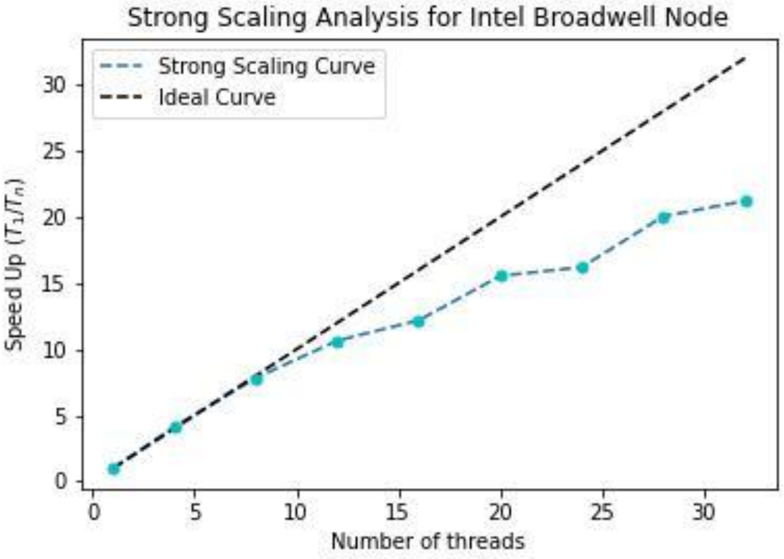
Roofline Analysis: M2P

- **Improves from 1.0 Gflop/s (sequentially) to 32.7 Gflop/s (32 threads, NUMA)**
- **Difference between NUMA aware and unaware is small**
  - M2P infrequently interacts with the *Particle* data structure
- **Other kernels (e.g. Timestepping) with more *Particle* memory accesses benefit significantly from NUMA**

Roofline for Parallel M2P Code



Strong Scaling Analysis



Threads	*Time* (T_n)	*Speed Up* (T_1/T_n)
1	5214.2856 s.	1.0
4	1260.3978 s.	4.1
8	665.9816 s.	7.8
12	489.5262 s.	10.7
16	427.8973 s.	12.2
20	335.5052 s.	15.5
24	321.7918 s.	16.2
28	260.1357 s.	20.0
32	245.8093 s.	21.2

Implementing shared memory parallelism with OpenMP, leveraging the Eigen library, and including NUMA first touch considerations produced a speed-up of ~21.2X with 32 threads

Conclusions

Key Takeaways

- **Implement and optimize high-performance code using:**
  - Tools: OpenMP, Eigen, Pybind11
  - Techniques: Data layout/access patterns, scheduling, effective data structures
  - Software-Hardware Interplay: Roofline Analysis, NUMA first-touch policy
- **Kevin will conduct his master's thesis implementing a new style of PIC algorithm with a group at the Princeton Plasma Physics Laboratory**

Future Work: Holistic Improvements

- **Implement efficient I/O pipeline**
  - Increase usability and efficiency
  - Explore capabilities available with HDF5 library
- **Switch to SoA data structures to enable SIMD vectorization**
- **Migrate to MPI + OpenMP Hybrid memory model**
  - Enables much larger problem sizes

Future Work: Kernel-Specific Improvements

- **Particle to Mesh Interpolation:**
  - Leverage partially shared grids or cell list data structures (*reduce memory footprint*)
  - Implement a higher order interpolation kernel (*increase accuracy and operational intensity*)
- **Discrete Poisson Equation:**
  - Use a better algorithm such as multigrid or FFT (*increase accuracy and efficiency)*
- **Mesh to Particle Interpolation:**
  - Implement a higher order interpolation kernel (*increase accuracy and operational intensity*)
- **Timestepping:**
  - Utilize a higher order ODE Solver (*increase accuracy and operational intensity*)

Thank You