Parallel Particle in Cell Algorithm

Kevin Howarth[1], Aditi Memani[1], Hari Raval[1], and Taro Spirig[1]

[1]Harvard University

May 5, 2022

Abstract

The Particle-In-Cell (PIC) algorithm is widely used to simulate the physics of plasmas because of its $\mathcal{O}(N)$ scaling with the number of particles N compared to the $\mathcal{O}(N^2)$ runtime of a direct N-body simulation. However, as the number of particles in plasmas can exceed 10^{18} particles per cubic meter, a serial implementation of the PIC algorithm becomes exceedingly time consuming and unrealistic. The PIC algorithm functions by interpolating particles to and from a grid on which it solves the Poisson equation. Each of the steps involved in this process are naturally parallelizable. We show that a parallel implementation of the PIC algorithm with 32 threads in a shared memory model can achieve up to a 21× speedup compared to a serial implementation when considering a large number of particles and grid points. We request resources from an HPC architecture to use our parallel implementation to simulate a two-stream instability in a plasma. This computation requires 217.8 core hours.

1 Background and Significance

1.1 Background

The Particle-in-Cell (PIC) algorithm was first introduced in 1955 as a method for simulating fluid mechanics. In the following years, it became popular as a method for simulating the time-evolution of a plasma - a superheated gas whose atoms have split into positively charged ions and negatively charged electrons. The trajectories of these charged particles are affected by Coulombic interactions between the particles as well as externally imposed electromagnetic fields. The resulting physical phenomena - plasma heating/cooling, wave transportation, turbulence, and propagation of instabilities, to name a few - are highly complex and of great interest to plasma physicists.

In particular, these phenomena are of utmost importance to scientists studying nuclear fusion. The field of fusion is concerned with the daunting task of designing and building devices which confine plasmas under extreme conditions in order to generate power. Using only abundant elements like Hydrogen and Helium as fuel, fusion promises to provide an effectively limitless source of clean energy and by extension a means of mitigating the effects of global warming. However, fusion reactors must be designed very carefully in order to achieve net power generation (a feat which has not yet been accomplished after 80 years of research) let alone to achieve economic viability. Thus, the clearer picture scientists have about a fusion plasma's behavior, the better the design of the resulting reactor.

Having sufficiently motivated the reasons for using a particle simulation code, we return to the problem of how to actually write such a code. One could naively attempt to simulate charged particle trajectories as an N-body problem, since Coulomb's law allows for exact calculation of the net force on each charged particle. However, the $\mathcal{O}(N^2)$ scaling of the N-body algorithm is very restrictive. The particle density in interesting plasmas can easily exceed 10^{18} particles per cubic meter, so another algorithm is needed.

The PIC algorithm circumvents the $\mathcal{O}(N^2)$ scaling by solving the Poisson equation on a computational grid to determine the electric potential and force fields arising from a given particle arrangement. Interpolation is used both to assign particle charges to grid points and to transfer electric field strengths back to particle locations, allowing the particles to be time-stepped from their exact positions by means of a numerical ODE solver. Since these operations only do work once on each particle, the PIC algorithm's runtime scaling is $\mathcal{O}(N)$, making it a much more attractive option than a direct N-body algorithm.

1.2 Related Work

The PIC method and its applications for plasma simulation is an active area of research and interest in the plasma physics community. In particular, the Princeton Plasma Physics Lab develops and maintains XGC, a highly optimized implementation of PIC which specializes in the simulation of the edge region of magnetically confined thermonuclear fusion plasmas. XGC was first used to gather research results by C. S. Chang et al [1]. It has since been validated and used for many physical studies [5, 6, 3, 2].

In addition, XGC is instrumented for HPC systems using MPI, OpenMP, CUDA, and AVX512 vectorization, and studies have been done inspecting how to most efficiently parallelize such algorithms [7]. In fact, XGC is one project planned for use on exascale computing architectures once they become available.

2 Scientific Goals and Objectives

The goal of this project is to demonstrate the capability of a PIC code to simulate plasma behavior at realistically large input values (e.g. number of particles). Even accounting for PIC's attractive linear runtime scaling with respect to number of particles, the runtime of a PIC simulation can still become quite large due to the extreme numbers of particles (more than 10^{18} particles per m^3) required to accurately represent fusion plasma phenomenon. Furthermore, a large number of grid points is desirable in order to reduce spatial discretization error. Finally, one may need to simulate at very small timestep sizes (and therefore for a large number of timesteps) in order to accurately resolve high-frequency plasma behavior. So, despite the fact that PIC is quite an efficient algorithm, it is applied to inputs large enough that parallelization becomes necessary to perform simulations in a reasonable amount of time for scientific progress. It is with this reason we would like to request compute hours on an HPC architecture.

3 Algorithms and Code Parallelization

The code used for this project was initially developed by two of the authors (Kevin and Hari) for their AM205 Advanced Scientific Computing: Numerical Methods final project in the Fall 2021 semester. Thus, this project's work involved porting that code from Python to C++, improving the algorithms, and instrumenting it for parallel/high-performance applications. Aside from specifically-mentioned external libraries, all the algorithms and code described in this section were written by the authors.

3.1 Mathematical Model and Non-Dimensionalization

Before discussing algorithmic details, it is appropriate to briefly explain the physical model our code implements. Many varying PIC models are possible depending on the physical phenomena of interest. This is due to inertial differences - an electron is about $1/1600^{\text{th}}$ the mass of a proton and will act at correspondingly higher frequencies. Considering this, our code uses what is known as an electrostatic model which examines only high-frequency electron behavior. In this model, only electrons are explicitly simulated, while ions are assumed never to move on the timescales being simulated and are approximated away as a uniform positive background charge. This approximation effectively halves the number of particles one must simulate to study high-frequency phenomena.

Furthermore, the model considers a two-dimensional square region in space with periodic boundary

conditions, so that an electron which leaves the right (or top) of the grid re-enters at the left (or bottom). This results in charge and mass being conserved over the runtime of a simulation. A two-dimensional model allows for simulation of many interesting plasma effects such as Larmor gyration of particles due to externally imposed magnetic fields, for example.

Finally, an important consideration when simulating individual particles is the extremely small size of relevant physical quantities. For example, the mass of a single electron is 9.11×10^{-31} kg, and its charge is 1.602×10^{-19} C. Additionally, spatial grid spacings and time-step sizes must be extremely small in SI units to accurately resolve particle motions. These numbers are well below the order of floating-point precision, so care must be taken to avoid rounding error dominating physical information. We address this by non-dimensionalizing all physical units involved in the mathematical model before beginning the simulation. This process normalizes all values to ranges which are accurately representable by floating point numbers without concerns relating to rounding error while also removing unnecessary floating point operations from the algorithm. More details on the non-dimensionalization process and electrostatic model are available in [4]. In our code, these operations are performed in the python driver scripts written previously for the AM205 version of the project. A consequence of this is that all C++ code is written assuming physical units are non-dimensionalized away and all discretization parameters (eg. step size, grid spacing) are equal to 1.

3.2 Data Structures

Our code uses an Array-of-Structures (AoS) approach to track the particle data. We implement a custom `Particle` class, where a single `Particle` contains arrays holding that `Particle`'s position and velocity as well as the electric field value at the `Particle`'s location for every timestep. When the simulation begins, an array of `Particle`s is dynamically constructed, and `Particle`s are accessed individually through this array as needed throughout the algorithm.

The AoS approach generally limits one's ability to leverage SIMD instructions, as relevant data for neighboring particles is not ordered in memory. However, it turns out that there are relatively few locations where the PIC Algorithm specifically would benefit from SIMD instructions or particle-to-particle locality gained in a Structure-of-Arrays (SoA) approach. In fact, because each particle tracks its own data at every single timestep of the algorithm, the AoS approach provides a boost to temporal and spatial locality during the time-stepping portion of the algorithm (see Step 4 in Section 3.3 below).

The PIC Algorithm also requires a computational grid which tracks charge density, electric potential, and electric field values at each point in space. As we are considering a two-dimensional region in space, a two-dimensional array of floats representing each physical value of interest is appropriate. For performance reasons discussed in Section 3.4, we implement these arrays as `Eigen::Matrix` objects.

3.3 Algorithms and Parallelization

A single timestep of the PIC algorithm can be broken down into four independent computational kernels. For clarity, both our explanation and implementation are divided into these four steps, for each step from time t to time $t+1$. The four steps of the algorithm repeat until the total simulation time, T, is reached.

Our code employs a shared memory parallelism model facilitated by OpenMP. For many problem sizes of interest, we note that we are able to achieve reasonable results with this model. We are aware that there are limitations to the shared memory approach as the number of particles and grid size increase to industry-level standards. For these much larger cases where the number of particles and grid do not fit into the memory capacity of a single node, MPI distributed parallelism would be more appropriate.

Step 1: Particle-to-Mesh Interpolation

The first step of the PIC algorithm is to interpolate particle signatures (in our case, charge) from the particle locations to the nearby mesh points. This process is often abbreviated as P2M. It can be done with any interpolation kernel, generally notated as $\phi_k(\boldsymbol{x_{p,d}})$, where $\boldsymbol{x_{p,d}}$ represents the distance between particle p and grid point d. For a given particle, we can update the relevant grid point values as:

$$\rho_d = q_p \times \phi_k(\boldsymbol{x_{p,d}}), \quad d \in \mathbb{D} \tag{1}$$

Here, q_p is the charge held by a single particle, ρ_d is the charge density at grid point d, and the set $\mathbb{D}$ of grid points to be updated depends on the interpolation kernel chosen. We implement the smoothing B-spline based M4 kernel proposed by Monaghan [8], which uses the following formula:

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

Here, x is the euclidean distance in a single dimension. It is of third-order local accuracy, and higher-order kernels can easily be substituted in its place if desired. For our two-dimensional use case, we simply apply the kernel once for each dimension, giving the final P2M update rule for a grid point a_d:

$$\rho_d = q_p \times \phi_{M4}(x_{p,d}) \times \phi_{M4}(y_{p,d}), \quad d \in \mathbb{D}_{M4} \tag{3}$$

Here, $x_{p,d}$ and $y_{p,d}$ are the distances between grid point d and particle p in the x and y direction, respectively. Due to the aforementioned non-dimensionalization, all distances are in units of grid spacings, so the set $\mathbb{D}_{M4}$ is simply the sixteen grid points within ± 2 grid spacings of particle p in both dimensions. For points further away, the M4 kernel evaluates to 0, so these grid points can be neglected. The result of this algorithm step is that the charge density ρ is known everywhere on the grid.

This operation is performed independently for each particle, thus (3) should be evaluated in parallel by multiple threads. However, if done naively, this results in a race condition, as multiple threads may attempt to update a single grid point at once. Our solution is to utilize thread-private grids. Each thread initializes its own grid and performs all interpolations to that grid. Then, once all threads have finished, the private grids are reduced (via element-wise addition) to a complete shared grid.

This process has limitations due to the memory required to host each thread-private grid as well as the overhead associated with copying and reducing the grids. However, it is a valid first-pass approach. Other possible approaches to grid sharing include a shared grid with locks on each point and partially shared grids (see Figure 3 in [7]). While some of these methods do result in increased performance, they involve complicated data structures and programming methodologies outside the scope of our work.

Step 2: Solve Discrete Poisson Equation

The second step of the PIC algorithm is to solve the discrete Poisson Equation on the mesh. The Poisson Equation is:

$$\nabla^2 V = \rho \tag{4}$$

Here V is the electric potential. We know ρ everywhere on the grid as a result of *Step 1*, so we can use second-order centered finite differences to discretize (4) as the following:

$$\frac{V_{i-1,j} + V_{i,j-1} - 4V_{i,j} + V_{i+1,j} + V_{i,j+1}}{h^2} = \rho_{i,j} \tag{5}$$

Here i, j represent indices on the grid. Thus, this results in a system of linear equations of dimension equal to the number of grid points (i.e. N_g^2). Solution via matrix inversion is extremely expensive at $\mathcal{O}((N_g^2)^3)$. Instead, we consider iterative methods which are more practical at $\mathcal{O}(I \times N_g^2)$, where I is some large constant number of iterations. Specifically, we will utilize a Successive Over-Relaxation (SOR) method. Details on the mathematical formulation of this method are available in [4]. We skip these details here and focus on the resulting update rule. Beginning with a matrix V^0 initialized to all zeroes, we can update an element (i, j) as:

$$V_{i,j}^{k+1} = (1 - \omega^k)V_{i,j}^k + \omega^k \frac{V_{i-1,j}^k + V_{i+1,j}^k + V_{i,j-1}^k + V_{i,j+1}^k + h^2\rho_{i,j}}{4} \tag{6}$$

Here ω^k is an SOR update parameter described in [4]. Note that this essentially corresponds to passing a plus-sign-shaped stencil over each point (i, j) on the mesh, reading from each point and its orthogonal neighbors, then writing to the point itself. The ordering in which this pass occurs can affect the resulting solution values. A commonly accepted ordering which provides fast convergence and easy parallelization is the red-black ordering, which treats the grid as a checkerboard with alternating red and black squares.

In the red-black ordering, the stencil is passed over all red points first. Next, ω^k is updated. Then, the stencil is passed over all black points, after which ω^k is again updated. This red-black ordering is repeated for a large number of iterations; generally, a few thousand iterations is enough to reach convergence.

The result of SOR iteration is that the electric potential V is known at every point on the mesh. After this, centered difference formulas may again be used to extract the electric field, as shown in (7). The final result is that the electric field is known at every point on the grid:

$$\vec{E} = \nabla V \implies \begin{cases} E_{x,i,j} = \frac{V_{i,j+1} - V_{i,j-1}}{h} \\ E_{y,i,j} = \frac{V_{i+1,j} - V_{i+1,j}}{h} \end{cases} \tag{7}$$

The checkerboard ordering of the red and black sweeps in SOR ensures that race conditions are impossible - a given red point only reads from itself and its four surrounding black points before writing back to itself, and vice versa. Thus, the red sweeps and black sweeps of (6) may be parallelized separately, with necessary synchronization points at the end of each sweep. Furthermore, since (7) only involves reading potential values from one grid and writing electric field values to another grid, there are no race conditions, and (7) can be parallelized over the number of grid points.

A final concern when implementing SOR is the alignment of data layout and data traversal. We implement our SOR stencil sweeps in row-major ordering. Thus, to reduce cache misses, we also order the memory layout of our `Eigen::Matrix` grids in row-major format. This ensures that the three horizontal points in the plus-sign-shaped stencil are almost always hitting in the cache. A possible improvement is to traverse the grid in a Hilbert space-filling curve in order to further increase spatial locality.

Step 3: Mesh-to-Particle Interpolation

The third step of the PIC algorithm is to interpolate the mesh signatures back to each particle (a process abbreviated as M2P). Here, we reuse the M4 interpolation kernel given in (2). The M2P update rule is:

$$\vec{E}_p = \sum_{d \in \mathbb{D}_{M4}} \vec{E}_d \times \phi_{M4}(x_{p,d}) \times \phi_{M4}(y_{p,d}) \tag{8}$$

Here, $\vec{E}_d$ is the electric field at mesh point d, and $\vec{E}_p$ is the electric field at particle p's location. The result of this step is that the electric field at every particle's exact location is known.

Like P2M, this operation is performed independently for each particle, thus (8) can be evaluated in parallel by multiple threads. However, unlike P2M, (8) only involves reading from the shared grid and writing to each particle. Since no two threads operate on the same particle, there is no race condition and thus parallelization is straightforward.

Step 4: Time-step Particle Locations

The final step of the PIC algorithm is to time-step the equation of motion for each particle to calculate its kinematic signature. We do so using the "Leapfrog" method, an ODE solver of second-order accuracy wherein the updated positions and velocities per particle are found by discretizing the equation of motion:

$$\vec{F} = m\frac{d^2\vec{x}}{dt^2} \implies \begin{cases} \vec{x}_{t+1} = \vec{x}_t + \Delta t \vec{v}_{t+1/2} \\ \vec{v}_{t+1/2} = \vec{v}_{t-1/2} + \Delta t \frac{\vec{F}(\vec{x}_t)}{m} \end{cases} \tag{9}$$

Here, $\vec{F}(\vec{x}_t)$ is the force applied to a particle at location $\vec{x}_t$. Note that the particle's new position may need to be updated to enforce the periodic boundary conditions if the particle traveled off the grid. Since we know the electric field at each particle's location, $\vec{E}(\vec{x}_t)$, as well as each particle's charge, q, we can find the force term as $\vec{F}(\vec{x}_t) = q\vec{E}(\vec{x}_t)$. Hence, the final update formula per particle is the following:

$$\vec{v}_{t+1/2} = \vec{v}_{t-1/2} + \Delta t \frac{q}{m}\vec{E}(\vec{x}_t), \qquad \vec{x}_{t+1} = \vec{x}_t + \Delta t \vec{v}_{t+1/2} \tag{10}$$

The result of this step is that the updated position and velocity values have been found for each particle. Thus, upon completing this step, the algorithm can loop back to *Step 1* and repeat as desired.

Parallelization of this step is straightforward as (10) is executed independently for each particle. Each particle reads its own kinematic data from the previous time-step, applies the Leapfrog update, performs data updates to enforce the periodic boundary conditions, and finally writes its updated kinematic data to itself. Hence, we are able to parallelize over the number of particles without any memory conflict. However, we note that each particle may potentially require a varying amount of computation to enforce the periodic boundary condition depending on whether (or by how much) it moved outside the grid. This could lead to an imbalanced work distribution among the threads. To remedy this, we make use of dynamic scheduling with a large chunk size to mitigate the possible work imbalance.

3.4 Scientific Libraries Employed

The primary scientific libraries employed in our code are Eigen and Pybind11. Our use of the Eigen library is motivated by the fact that grid operations can be made both simpler and more efficient. In particular, the use of an `Eigen::Matrix` object greatly simplifies the code needed to define a custom reduction over the thread-private grids in the P2M step. This step requires an extensive amount of matrix copying and addition operations, which Eigen can perform efficiently.

Our use of Pybind11 stems from the fact that we aim to connect our previously written inputs and output scripts in Python [1] to the newly written C++ PIC algorithm. In general, scientific codes may be written and compiled in C++, while Python driver scripts are used to call the codes and perform analysis on outputs. It is good practice to be familiar with this workflow.

3.5 NUMA Considerations

As mentioned in Section 3.2, each Particle contains an array of its relevant kinematic and electric field data at each timestep in the simulation. Using a compiler flag, we implement the option to either use

[1] Kevin and Hari wrote these input and output scripts for the final project "Particle-in-Cell Method for Plasma Simulation" in Harvard University's AM 205 course. The scripts can be found in the project repository for CS 205.

an `std::vector<float>` or a dynamically-allocated array of floats, `float*`. The `std::vector<float>` option makes no guarantees about where in memory the particle data will be stored, while the `float*` option ensures that particle data will be stored near the CPU which initialized it. There is no difference between these options when using a single socket (i.e. 16 or fewer CPUs) on a Broadwell node.

However, when running with more than 16 CPUs, two sockets on a Broadwell node are used, and the overhead of reaching data stored in memory becomes more noticeable, especially if said data is far away from the CPU reaching for it. In this case, the second option enforces a good NUMA first-touch policy. By parallelizing the particle array initialization process, we ensure that all CPUs initialize some particles near themselves. Then, when those same CPUs access data about their particles in memory, there is less overhead required to retrieve that data.

3.6 Validation and Verification

Validation of PIC codes is a challenging and nuanced topic, due mainly to the difficulty of acquiring true data via experimental measurements. It simply isn't possible to experimentally measure the exact, individual trajectories of large numbers of particles. A low-level first approach is to confirm conservation of physical quantities such as charge, mass, and total energy. Another possible approach is to perform an artificial convergence study by running a simulation with extremely fine spatial and temporal discretizations. This high-fidelity simulation constitutes a set of "quasi-true" particle positions. Then, the discretization resolution may be decreased, and one may observe how the position error between corresponding particles in the rough and "quasi-true" solutions converges. A final approach, described in [4], is to formulate tests which involve recreating physically expected behaviors.

Kevin and Hari performed each of these tests at length for their AM205 project last semester. They observed that charge and mass are conserved unconditionally, while total energy is conserved only for small enough timestep sizes. For example, a non-dimensionalized timestep size of 1 conserves energy, while 2.25 does not as the system becomes numerically unstable. Additionally, they observed through their convergence tests that accuracy of the PIC method is dominated by the timestep size for sufficiently large grids ($N_g \gtrsim 100$), with error decreasing as $\mathcal{O}(dt^4)$ due to the interplay between timestepping, interpolation, and finite differencing solution. Finally, they were able to use a one-dimensional version of their code to recreate some physically predicted behaviors, including oscillation at the plasma frequency as well as a streaming instability (both described in [4]).

As the algorithm is fundamentally the same between this semester and last semester, we chose to accept the accuracy of the algorithm based on the above results and focus entirely on optimizing performance for this project. For this semester, we confirmed via visual inspection that the code produces reasonable particle trajectories. One such trajectory is displayed in Figure 1. The red 'X's denote the initial positions of each particle. We observe that nearby particles tend to push each other away in an attempt to reach a stable configuration. We have included a script in our code for reproducing similar figures.

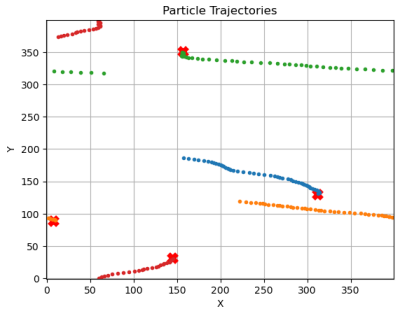


Figure 1: An example simulation result of particle trajectories

4 Performance Benchmarks and Scaling Analysis

4.1 Problem Sizes

To run our performance benchmarks, we first consider a small job size which is compatible with a reasonably short runtime. Our small problem size consists of $N_g = 400$ grid points, $N_p = 10^6$ particles, and $N_t = 100$ timesteps. Our larger problem size, which better highlights performance improvements due to the parallelization of our algorithm, utilizes $N_p = 3 \times 10^7$ particles, $N_g = 1500$ grid points, and $N_t = 100$ timesteps. As outlined in Section 3, for all our problem sizes, we only consider one Intel Broadwell node. Each node has 125 GB of available memory and we compute our required memory usage per job as follows:

$$Mem(N_p, N_t, N_g) = 4\left(N_p \times N_t \times (2+2+2) + N_g^2 \times (1+1+2)\right) \times 10^{-9} \text{ [GB]} \tag{11}$$

Equation 11 comes from the fact that for our algorithm, we need to consider three two-dimensional vectors (position, velocity and electric field) per particle per timestep. Additionally, we track two scalar fields (charge density and potential) and a two-dimensional vector (electric field) at each grid point. Moreover, the factor of four in Equation 11 represents that all computations are performed in single floating point precision. Using this formula to compute our memory requirements leads to the need of about 2.4 GB of memory for our smaller problem size and about 72.1 GB of memory for our larger problem size. We provide more detail about the requirements of each of these problem sizes in Table 1 below [2].

Parameter	*Small*	*Large*
Typical wall clock time (hours)	0.03	1.5
Typical job size (nodes)	1	1
Memory per node (GB)	2.4	72.1
Maximum number of input files in a job	1	1
Maximum number of output files in a job	0	0

Table 1: Workflow parameters of the "small" ($N_g = 400$, $N_p = 10^6$, and $N_t = 100$) and "large" ($N_p = 3 \times 10^7$, $N_g = 1500$, and $N_t = 100$) test cases used during project development and benchmarking, respectively.

4.2 Performance Results

The performance results of our code can be found in Figure 2 below. To produce our roofline analysis, we worked with one Intel Broadwell node and ran our algorithm with an input size of $N_p = 3 \times 10^7$ particles, $N_g = 1500$ grid points, and $N_t = 100$ timesteps, as mentioned in the previous subsection. In the left subplot of Figure 2, we observe the sequential performance of each of the four steps of the PIC algorithm. Notably, we see that all four kernels are memory-bound. The largest performance bottleneck is attributed to the SOR iteration kernel. This makes sense, as SOR requires many memory accesses per operation.

Next, we focused on analyzing the performance improvements for the M2P kernel, since this kernel has the highest ceiling for improvement. This analysis yields the roofline shown in the right subplot of Figure 2. From this plot, we observe that the baseline sequential implementation of this kernel corresponds to about 1.0 Gflop/s. However, after introducing the Eigen library for grid representations and OpenMP shared memory parallelism with 16 threads, we achieve about 16.1 Gflop/s. We further improve to roughly 26.2 Gflop/s when making use of 32 threads. Finally, when applying an intelligent NUMA first-touch policy for our particle data, we find that the performance of the M2P interpolation improves to about 31.7 Gflop/s. We note that the NUMA considerations did not significantly boost the performance of

[2] For the "Maximum number of input files in a job" in Table 1, we write 1 as we have built a capability to read in input data from a CSV. However, when collecting performance metrics, we randomly simulate the data for our measurements.

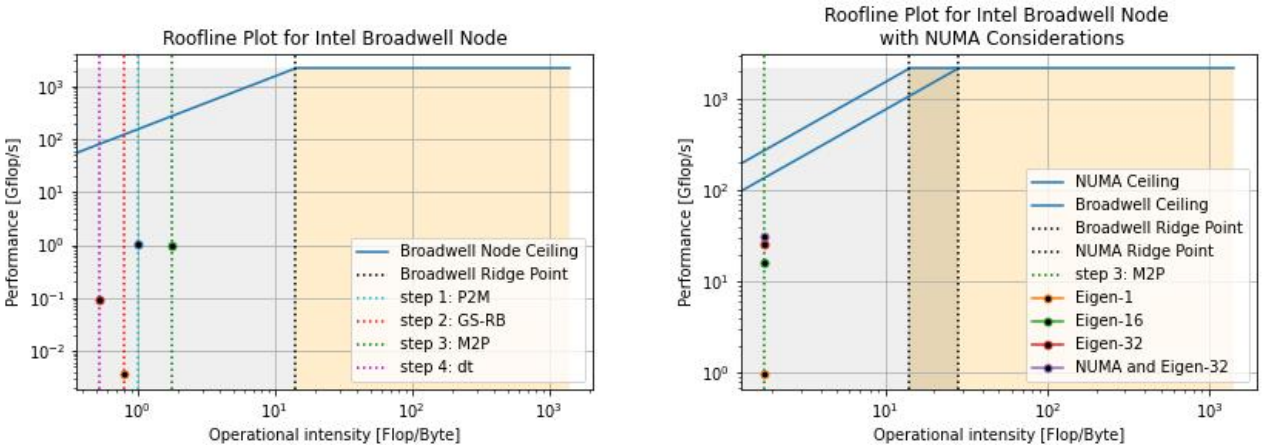


Figure 2: Roofline analysis on one Intel Broadwell node with our "large" problem size ($N_p = 3 \times 10^7$, $N_g = 1500$, and $N_t = 100$). The left subplot shows the performance of the sequential code for each of the four steps of the algorithm. The right subplot shows the performance of "Step 3: Mesh-to-Particle Interpolation" across different parallelization improvements.

this kernel due to the fact that M2P does not involve very many interactions with the `Particle` data structure. In our developmental experiments, we found that the time-stepping kernel benefits much more from NUMA considerations, since this kernel performs more significant work with the `Particle` data structure.

4.3 Strong Scaling Analysis

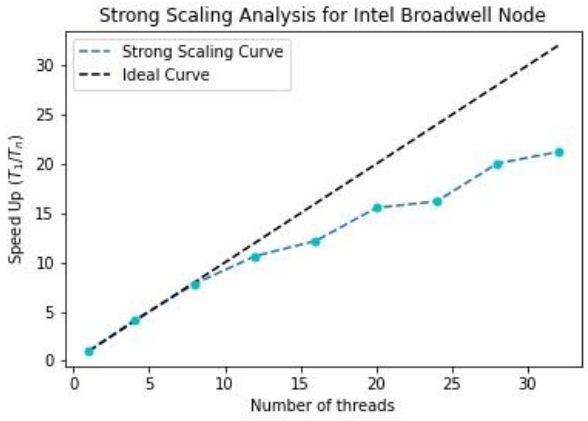
To further understand the improvement in performance from parallelizing the PIC code, we performed a strong scaling analysis as shown in Figure 3. Similar to our roofline analysis, we worked with one Intel Broadwell node and ran our code with an input size of $N_p = 3 \times 10^7$ particles, $N_g = 1500$ grid points, and $N_t = 100$ timesteps. After implementing shared memory parallelism with OpenMP, leveraging the Eigen library, and including NUMA first-touch considerations, we observed a speed-up of around $21.2\times$ when using 32 threads. Figure 3 further shows that for approximately the first twelve threads, our performance aligns nearly perfectly with the ideal curve. Moreover, even after some divergence from the ideal curve when using more than twelve threads, our performance continues to increase at a relatively consistent rate, without plateauing. From this analysis, it becomes evident that using even more threads could be beneficial to our code, as we are likely to see a continued increase in performance.

5 Resource Justification

The wall time for our typical job size ($N_p = 3 \times 10^7$ particles, $N_g = 1500$ grid points, $N_t = 100$ timesteps) using 32 cores is about 245 s. as shown in Figure 3. Our request for compute resources does not require multiple nodes as our implementation is based on a shared memory model. The core hours for a typical production run is therefore computed as follows:

$$2.178 \text{ core hours} = 32 \text{ cores} \times \frac{245s}{3600 \frac{s}{\text{hour}}}$$

In our simulation, we aim to simulate the interactions between two streams of particles (each with 1.5×10^7 particles, leading to a total of 3×10^7 particles) where the particles in one of the stream start with a constant initial velocity in the positive x-direction and the particles in the other stream start with a constant initial velocity in the opposite direction. The phenomenon is known as a two-stream instability. Physically speaking, what we should observe is a transfer of energy within the system. At the beginning,



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

Figure 3: Strong scaling analysis on one Intel Broadwell node with our "large" problem size. The subplots illustrate the scaling of our most optimized parallel code which includes NUMA first-touch considerations.

the entire energy of the system is concentrated in the kinetic energy of the particles. However, as time goes on, the energy is translated into a potential well which forces electrons to accelerate around it.

Metric	*Two-stream Instability Test*
Simulation per task	1
Iterations per simulation	100
core hours per iteration	2.178
Total core hours	217.8

Table 2: Justification of the resource request

To observe the described physical behaviour fully, we require 10^4 timesteps. Table 2 contains the resources requested for this simulation. In summary, we would like to request 217.8 core hours at the supercomputing center to cover our test case.

References

[1] C. S. Chang, S. Ku, and H. Weitzner. Numerical study of neoclassical plasma pedestal in a tokamak geometry. *Physics of Plasmas*, 11(5):2649–2667, 2004.

[2] R. Hager, J. Lang, C. S. Chang, S. Ku, Y. Chen, S. E. Parker, and M. F. Adams. Verification of long wavelength electromagnetic modes with a gyrokinetic-fluid hybrid model in the xgc code. *Physics of Plasmas*, 24(5):054508, 2017.

[3] R. Hager, E. Yoon, S. Ku, E. D'Azevedo, P. Worley, and C. Chang. A fully non-linear multi-species fokker–planck–landau collision operator for simulation of fusion plasma. *Journal of Computational Physics*, 315:644–660, 2016.

[4] R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill Inc., 1981.

[5] S. Ku, C. Chang, and P. Diamond. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion*, 49(11):115021, sep 2009.

[6] S. Ku, R. Hager, C. Chang, J. Kwon, and S. Parker. A new hybrid-lagrangian numerical scheme for gyrokinetic simulation of tokamak edge plasma. *Journal of Computational Physics*, 315:467–475, 2016.

[7] K. Madduri, J. Su, S. Williams, L. Oliker, S. Ethier, and K. Yelick. Optimization of parallel particle-to-grid interpolation on leading multicore platforms. *IEEE Transactions on Parallel and Distributed Systems*, 23(10):1915–1922, 2012.

[8] J. Monaghan. Extrapolating b splines for interpolation. *Journal of Computational Physics*, 60(2):253–262, 1985.