DynoGrid

Dynamically Load-Balanced PIC Code with an Adaptive Grid

Scott Luedtke, Max Porter, Joel Iventosch, Mark Sholte
https://github.com/mgsholte/dynogrid/

May 19, 2015

1 Introduction

Particle in Cell (PIC) codes are commonly used to simulate plasma physics. Existing codes, such as EPOCH [1], allow physicists to simulate many experiments, including laser experiments. Larger problems, such as 3D simulations of high-intensity laser experiments like those performed on the Texas Petawatt Laser (TPW), are too computationally expensive to run on modern supercomputers. One way to bring these problems down to size is to adaptively change the grid size. Simulations can require a very high resolution at certain sites (for example, where a laser hits a target), and orders of magnitude coarser resolution throughout the rest of the simulation.

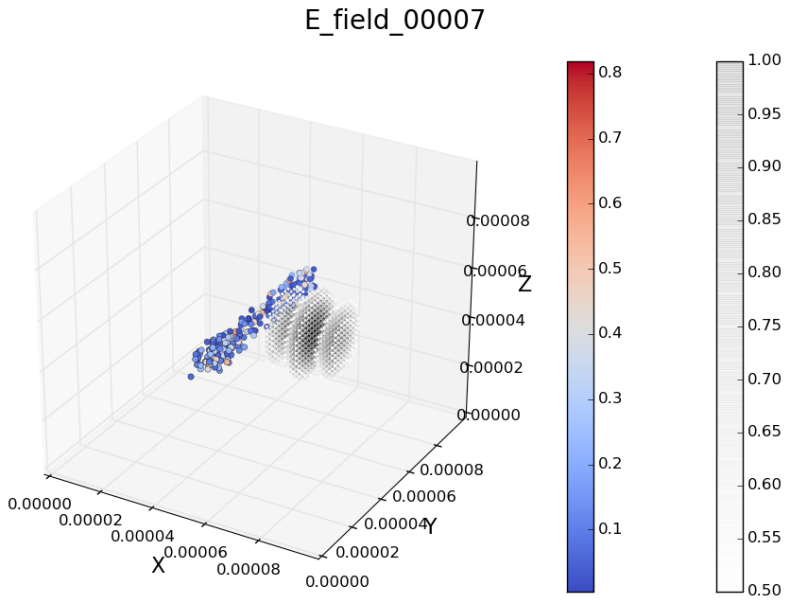


Figure 1: Visualization of our Simulation, showing the particles dispersing after the laser hits them.

We have developed a 3D PIC code with a standard Boris particle pusher [2], added an adaptive grid, and implemented data passing routines among nodes. Additionally, we developed a load balancer, however we were unable to finish debugging it. Our code is not physically accurate, omitting a field solver and ignoring

many physically relevant effects, but we believe it is an accurate representation of a production code in performance characteristics and expect it to scale similarly to a production code.

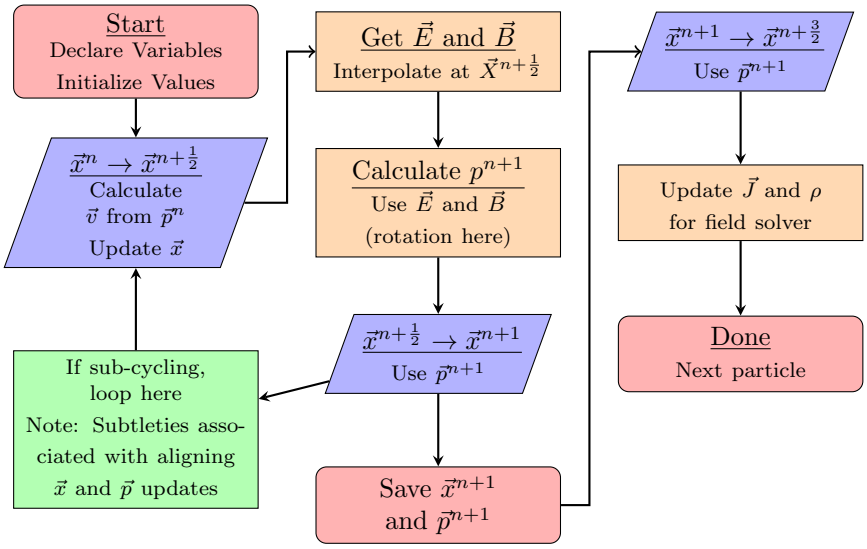
2 Code Description

PIC codes are commonly used in computational physics. A PIC code typically consists of an array of grid points that define electromagnetic field values, and particles that move continuously throughout the grid. We created a fairly simple 3D laser-plasma code with externally defined fields (the laser), and physical effects such as particle collision and photon pair production omitted. We did however add an adaptive grid which makes load balancing more complex.

2.1 Particle Pusher

Particle-in-cell (PIC) codes used to simulate laser-plasma interactions commonly use the Boris pusher [2], a second order accurate pusher for charged particles in electromagnetic fields. The pusher is outlined in Fig. 2. Essentially, the particle is advanced a half time-step, the momentum is updated with $\vec{E}$ and rotated with $\vec{B}$, and the particle is moved a final half time-step. With a properly set time step, a particle will never move more than one cell.

Figure 2: Schematic of the Boris pusher. The sub-cycling and current and density calculations were not implemented.



Our pusher follows the implementation in EPOCH [1], but with some modifications. We use our own linear interpolation method (for simplicity), and omit the steps necessary for the field solver, which we did not implement. We have different logic for finding the field points nearest a particle, and recursive logic to find the finest grid cell a particle is in. In addition, we have a particle list for each base cell, and implemented logic to pass particles between lists (this makes the load balancing easier).

2.2 Particle Lists

Each cell may contain any arbitrary number of particles during the course of the simulation. This means each cell needs a variable length data structure to keep track of the particles within its boundaries. For this purpose, we created a singly-linked list structure which holds pointers to particles in a special node struct.

Each node holds a pointer to a single particle and a pointer to the node containing the next particle in the list.

Midway through the project, we decided to generalize the lists to hold anything, such as neighboring processors or cells to send. The lists proved very robust and the implementation can be viewed in `list.c` and `list.h`.

2.3 Grid Trees

In order to simulate electromagnetic fields, the area of the simulation must be discretized, with each point on its grid holding 3D values for the electric and magnetic fields $\vec{E}$ and $\vec{B}$. Since we are implementing an adaptive grid, we chose to represent our grid cells with a data type called a tree. A tree holds various information only at the coarsest cell size, such as the particles located in that cell, the cell position relative to the entire grid over all processors (a.k.a. its global position), and the ID of the processor that owns it. It also holds a pointer to a tree node, which contains the eight corner grid points (their $\vec{E}$ and $\vec{B}$ fields) and 8 recursive pointers to (potential) tree node children. This allows for an efficient adaptive grid by separating data which is only needed at the coarsest grid level from data which needs to recurse, and therefore removing any unnecessarily repeated information.

As an additional note, it happens that treating each grid cell as an object (rather than, say, having a global array of grid points) leads to the possibility of repeated grid points since each point is "owned" by eight different cells. However we took care to prevent these repeats by having each cell be in charge of a single grid point, and simply point to the grid points of seven other neighbor cells. In order to not miss any grid points in this fashion, we then added a layer of "ghost cells" to each processor that each contribute one grid point without otherwise affecting the simulation.

It may also be helpful to explain the purpose of having each grid cell keep track of its owner. With our previously described particle pusher, it is necessary for each processor to keep a copy of every cell belonging to other processors which is adjacent (or diagonal) to a cell it owns, so that it can accurately simulate its particles moving into those cells (and therefore being transferred to other processors). In order to distinguish between these neighbor cells (or "ghost cells" as we refer to them) and its own cells, every cell simply remembers who its real owner is.

2.4 Grid Refinement

Grid refinement is how we execute the idea of an adaptive grid. An adaptive grid is a grid which starts at a "base" grid spacing which is relatively low, then dynamically becomes finer only where deemed necessary. The condition we use to determine how refined a given grid cell needs to be is the variation of the electric and magnetic fields across that cell. If the variation is large, we conclude that the cell does not have enough resolution to properly represent that cell and therefore we choose to refine it. If a cell has been refined, but the variation across its cell (and the cells of its children) becomes relatively small, it concludes that the extra resolution is no longer necessary and chooses to coarsen. Refinement can also be done recursively on the generated children, and therefore, up to physical limits, refinement can occur as many times as necessary.

Since our tree data type is recursive, our grid refinement algorithm is also, of course, recursive. Our algorithm consists chiefly of two functions: refine() and coarsen(). As the names would suggest, refine() causes a tree node to generate 8 children, each half of its size in each dimension, and coarsen() causes a tree node to destroy its children, establishing itself as the most refined cell size at that location. At each time step, every grid cell is asked three questions: Does it need to coarsen? Does it need to refine? And do its children need to coarsen or refine? Since this acts recursively at each time step, any amount of refinement or coarsening can happen in a single step, so the refinement level of the grid can never fall behind the current demand.

3 Parallelization

3.1 Particle List Passer

Particles move throughout the simulation volume and often end up in a different cell at the end of any given time step. Since different cells can be on different processors, all processors must stop at the end of every time step to pass particles and reconcile their differences with neighbors. To facilitate this, each base grid cell keeps a list of the particles in it and a separate list of the particles that have moved into it. All of the particles that need to be passed are thus in the second list of the "ghost" cells (the surface cells which belong to neighboring processors) and there is no need to traverse a list of all particles to find those that need to be passed. This is implemented in three steps in functions in `push.c`, `mpicomm.c`, and `mpi_dyno.c`.

Before the communication, each processor assembles a list of its neighbors by checking the owner of each of its ghost cells. The neighbor relationship is one-to-one, i.e., if A is B's neighbor, B is A's neighbor. Thus each processor can communicate directly with only the processors it needs to.

First, each processor communicates to each neighbor how many particle lists it will pass, so the neighbors know how many to expect. This is followed by an `MPI_WaitAll` command to ensure synchronization. Next, the neighbors communicate the size of each list to be sent, followed by another `MPI_WaitAll`.

In the third step, the particles are themselves passed. First, the singly-linked lists of particles are packed in to buffers that are continuous arrays of particles. These buffers are then sent. Memory for the receive is allocated based on the previous two communications and the receive is executed, followed by a final `MPI_WaitAll`. Finally, the particles are unpacked into their respective lists.

3.2 Cell Passer

During load balancing (explained in the next section) it is necessary to pass cells between processors in order to equilibrate the amount of work required of each processor. This is carried out with the cell passer. The cell passer executes `MPI_Isend` and `MPI_Irecv` with each of its neighbors along the axis of load balancing, although only up to one of these sends will be non-empty. Since we have chosen the y-axis to be our single axis of load balancing, only processors directly above or below each other trade cells.

The process of trading cells goes as follows: first the lists to be sent to neighbors are packed to a convenient, sendable buffer. Since each cell actually contains both a cell (grid points plus a global spatial location) and a list of particles, these elements are compressed individually and sent sequentially. After all messages are sent, the cells which are being given away are removed from the local grid. Then all receives are called and waited for. Upon completion, the received data is unpacked and restored to full cell form, and one cell at a time is added to the grid based on its global spatial position. Finally we ensure that all sends have finished and exit the cell passer.

3.3 Load Balancer

The load balancer exchanges cells and particles between processors so that each processor has approximately the same amount of work to do. The cell passer does not currently work, so the balancer has not been fully tested.

The balancer first calculates how much work is on each processor based on the number of particles and cells each processor controls. An all reduce computes the total work from which the target work is calculated. The balancer then enters a loop that runs until no processor has more work than 1.1 times the average work.

In each step of the balancer, each processor communicates with its neighbors and tells it how much work it wants to give or receive to become balanced. Based on the responses of its neighbors, each overloaded processor gives a layer of cells in one direction. If a processor and it's neighbors are all overloaded, there is a heuristic that gives cells outwards. The heuristic prevents deadlocking and matches where the particles typically go.

4 Theoretical Performance

For sequential performance, performance should depend on the number of particles and grid points in the simulation. Essentially, computation time should be

$$\mathcal{O}(N + M), \tag{1}$$

where N is the number of particles and M is the number of grid points. Since we are using an adaptive grid, M is not constant, however, we will treat it as a constant here. I (Scott) expect that full, physics-accurate simulations will refine in one area and simultaneously coarsen in another, keeping the total number (but not the processor distribution) of grid points generally constant. This is especially true for our test problem.

In the parallel case, we expect the performance to be

$$\mathcal{O}\left(\frac{N+M}{p} + \left[L + \frac{M}{p^{\frac{2}{3}}b}\right] + \left\{L\log p + (nL + \frac{p^{\frac{2}{3}}}{b})p^{\frac{1}{3}}\right\}\right), \tag{2}$$

where p is the number of processors, b is the bandwidth, and L is the latency. The terms in brackets are for the particle passing and depend on how much boundary area there is between processors. The terms in braces are for the load balancer and include reductions and broadcasts on a hypercube topology.

5 Performance Results

Although limitations in our cell passing and load balancing routines did not enable us to fully implement the load balancing schema that we'd envisioned, our parallelized code was still able to achieve satisfactory results, as is evidenced in the table and plots below.

Time	**S**	**E**	**p**	**Proc Divs**			**# of Cells**			**Tot Cells**	**Tot Particles**
				x	**y**	**z**	**x**	**y**	**z**		
731.9	1.00	1.00	1	1	1	1	96	96	96	2,097,152	20,971,520
209.0	3.50	0.29	12	1	4	3	96	96	96	2,097,152	20,971,520
124.6	5.87	0.24	24	1	6	4	96	96	96	2,097,152	20,971,520
80.0	9.15	0.19	48	1	8	6	96	96	96	2,097,152	20,971,520
55.6	13.16	0.14	96	1	12	8	96	96	96	2,097,152	20,971,520
37.8	19.36	0.10	192	1	16	12	96	96	96	2,097,152	20,971,520

Table 1: **Strong Scaling Results**. S = speedup, E = efficiency, p = number of processors

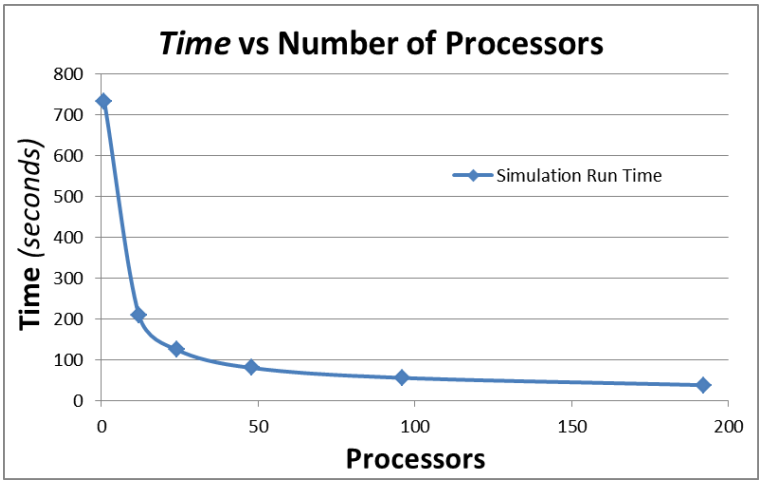


Figure 3: Strong Scaling: ***Time*** **vs Number of Processors**

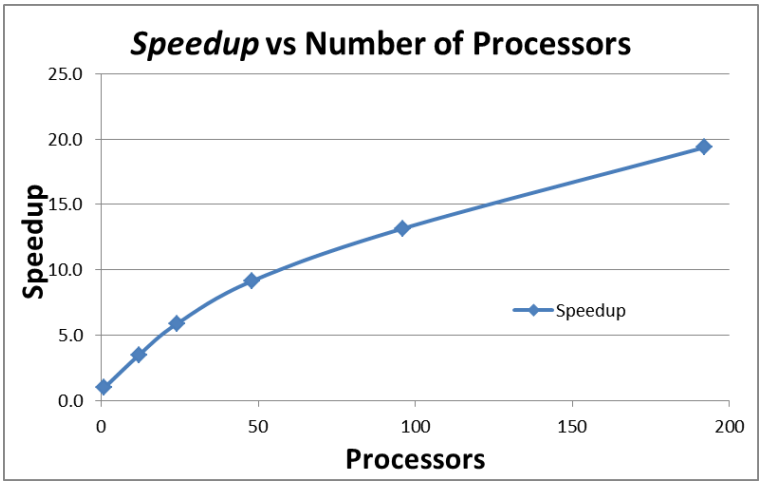


Figure 4: Strong Scaling: ***Speedup*** **vs Number of Processors**

Time	S	E	p	Proc Divs			# of Cells			Tot Cells	Tot Particles
				x	y	z	x	y	z		
689.6	1.0	1.0	1	1	1	1	96	96	96	884,736	8,847,360
2215.2	0.3	0.002	16	1	4	4	192	192	192	7,077,888	70,778,880

Table 2: **Weak Scaling Results**. S = speedup, E = efficiency, p = number of processors

6 Conclusions

Despite all that we accomplished in this project, the size of what we aimed to achieve ended up greater than we could manage. Most notably, the logic for a 3D code proved very convoluted and difficult to debug. We were able to figure everything out with the exception of the load balancer and cell passer. We made significant progress toward a 3D load balancer, but we opted for a (still challenging) one-dimensional balancer

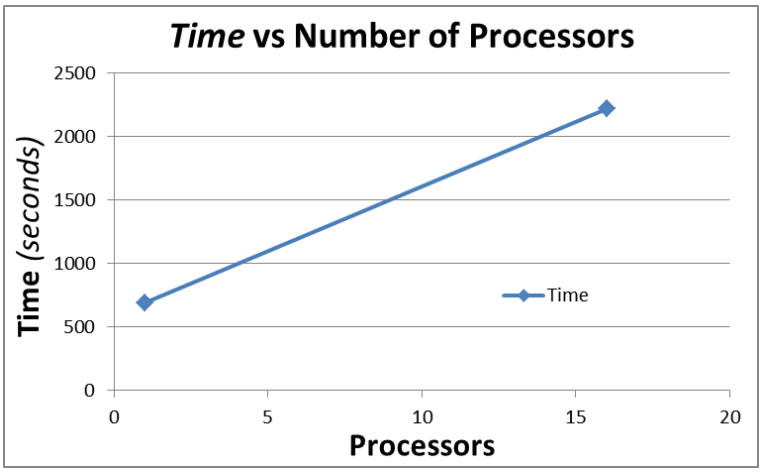


Figure 5: Weak Scaling: ***Time*** **vs Number of Processors**

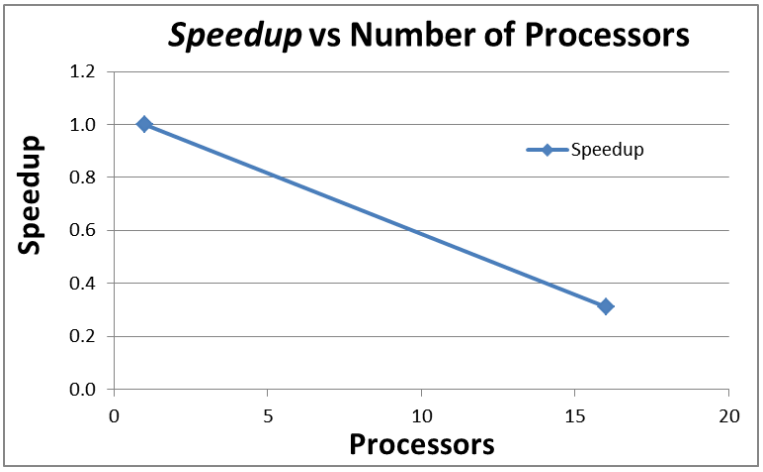


Figure 6: Weak Scaling: ***Speedup*** **vs Number of Processors**

instead. In the end we were not able to fully debug the balancer and cell passer and instead ran a parallel, but unbalanced, code.

Our code did scale, though not particularly well. Without the load balancer, the initial processor divisions were in place for the whole simulation. The initial divisions are naively set based on the number of base grid cells, and do not reflect the fact that the particles are densely packed in the center and the laser is at one edge. As the simulation progresses, the particles and laser move, but not in a way to significantly improve the balancing. As we increased the number of processors, there were many more divisions and thus much more particle passing to be done. This surely hurt the efficiency results.

Since full-scale laser-plasma simulations will be much larger than what we ran, the weak scaling is arguably more important. We estimated that doubling the resolution in each dimension would result in 16 times as much work. There would be 2^3 times as many base cells and 2 times as many time steps. Unfortunately, our larger runs are waiting in the queue and probably won't get started for quite a while. As a result, our weak scaling results are somewhat inconclusive.

In working on this project we have all learned valuable lessons. Developing and debugging complex codes is hard. Things that are conceptually simple (e.g. adaptive grid, load balancing) may prove very complex to implement. MPI can be fussy. Some of us took this class with the desire to learn MPI, and this project

has taught us quite a bit. We even discovered that sometimes compilers pad structs in an undefined way and custom MPI data types must reflect that.

7 Whodunit

Scott Luedtke was primarily responsible for the particle pusher and load balancer. Since this project is closely related to Scott's research, he oversaw the physics and provided general guidance for what the code should accomplish.

Max Porter was most frequently responsible for grid-related functions. He wrote the grid (and particle) initializations, the laser, and much of the (rather complicated) grid cell passer. He also worked out bitwise logic necessary for grid implementation.

Joel Iventosch wrote the output and visualization routines and helped with the MPI passing routines. He also contributed to the grid refinement and load balancing functions, and helped with the recursive laser implementation.

Mark Sholte wrote much of the list functions and tree structure. He also wrote much of the code for sending particles between processors in the pusher. In addition, he managed any git repository issues that arose and proved a masterful debugger.

[1] The EPOCH code was developed as part of the UK EPSRC funded projects EP/G054940/1

[2] Birdsall, C.K. and Langdon, A.B. (1985). *Plasma Physics via Computer Simulation.* McGraw-Hill.