DynoGrid

Dynamically Load-Balanced PIC Code with an Adaptive Grid

Scott Luedtke, Max Porter, Joel Iventosch, Mark Sholte

May 17, 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.

We have developed a 3D PIC code with a standard Boris particle pusher [2], added an adaptive grid, implemented data passing routines among nodes, and load balanced the work based on the changing grid. Our code is not physically accurate, ommiting 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 our load balancing to behave similarly on a production code.

2 Code Description

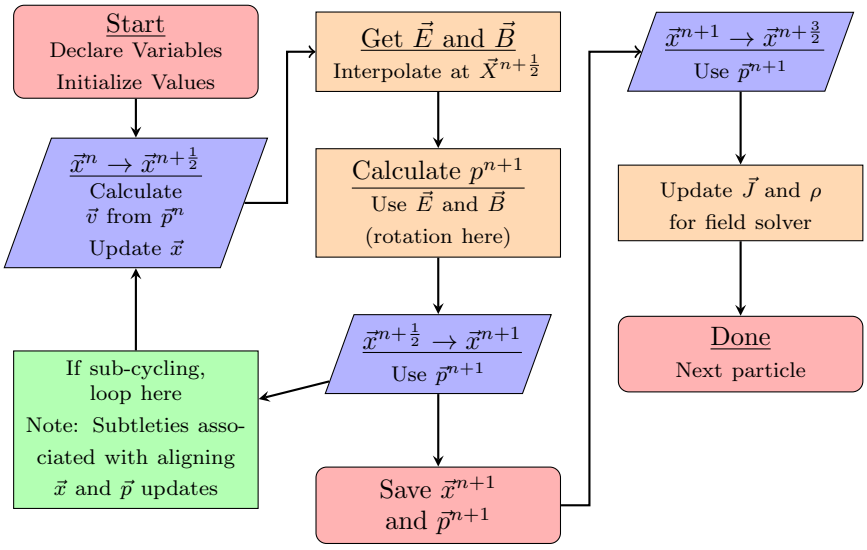
PIC codes are commonly used in computational physics. A PIC code typically consists of an array of grid points that define electromagnetic field values, and particles that move continuously throughout the grid. We created a fairly simple 3D laser-plasma code with externally defined fields (the laser), 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. 1. 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.

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).

Figure 1: Schematic of the Boris pusher. The sub-cycling and current and density calculations were not implemented.



2.2 Particle Lists

2.3 Grid Trees

2.4 Grid Refinement

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 hove moved into it. All of the particles that need to be passed are thus in the second list of the ghost cells 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 compressed to a convenient, simplified form to minimize the message size. 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 uncompressed and restored to full cell form, and one cell at a time is added to the grid based on its global spatial position. Care is taken to update the local grid's size and, as may occasionally be necessary, check whether the new cells stray out of bounds of the previously initialized grid, and if so, to re-initialize the grid so that it is centered around the new dimensions with ample padding to avoid another re-initialization in the near future. Finally we ensure that all sends have finished (which they surely have) and exit the cell passer.

3.3 Load Balancer

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.

5 Performance Results

6 Discussion

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 worked on the grid refinement and stuff.

Joel Iventosch wrote the output and visualization routines and helped with MPI passing. He also helped with the grid refinement.

Mark Sholte wrote much of the list functions and tree structure. He also managed any git repository issues that arose and proved a masterful debugger.

8 Conclusions

[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.