A Load-Balancing Algorithm for a Parallel Electromagnetic Particle-in-Cell Code

Steven J. Plimpton* David B. Seidel* Michael F. Pasik* Rebecca S. Coats*
Gary R. Montry†

Keywords: particle-in-cell, electromagnetics, plasma simulation, load-balancing, parallel computing
PACS codes: 52.65.-y 52.65.Rr

Abstract

Particle-in-cell simulations often suffer from load-imbalance on parallel machines due to the competing requirements of the field-solve and particle-push computations. We propose a new algorithm that balances the two computations independently. The grid for the field-solve computation is statically partitioned. The particles within a processor's sub-domain(s) are dynamically balanced by migrating spatially-compact groups of particles from heavily loaded processors to lightly loaded ones as needed. The algorithm has been implemented in the QUICKSILVER electromagnetic particle-in-cell code. We provide details of the implementation and present performance results for QUICKSILVER running models with up to a billion grid cells and particles on thousands of processors of a large distributed-memory parallel machine.

1 Introduction

Plasma simulation via particle-in-cell (PIC) methods has a long history extending back to the beginning of scientific computing. Over the ensuing decades practitioners have developed a rich set of numerical and computational techniques useful for simulating a variety of plasma phenomena [1, 2]. The basic PIC idea is to model the reponse of electromagnetic fields and low-density plasmas (charged particles) to each other in a self-consistent manner: the fields push the plasma particles and the plasma current modifies the fields. PIC codes can be extremely compute-intensive, which naturally motivates a need for parallel computing.

The two basic operations within a PIC code, field solves and particle pushes, are inherently parallel computations. However spatial and temporal inhomogeneities in the particle density, coupled with the need to interpolate back and forth between the grid and particles every timestep, can create a fundamental parallel performance bottleneck. This paper addresses this well-known problem – how to balance the two competing computations of a PIC simulation across the processors of a distributed-memory parallel machine.

*Sandia National Laboratories, Albuquerque, NM 87185. {sjplimp, dbseide, mfpasik, rscoats}@sandia.gov.
†Southwest Software, Albuquerque, NM 871111. montry@spsoft.com.

At Sandia, our interest in modeling plasma effects and understanding pulsed-power experiments led to the development of the serial QUICKSILVER (QS) package over the last 15 years [3, 4]. QS is a 3d multi-block, finite-difference, fully-relativistic, particle-in-cell code, which has been used to simulate ion and electron diodes, magnetically insulated transmission lines, microwave devices, electron beam propagation, and high-current plasma devices. QS is an electromagnetic PIC code which means it solves Maxwell's equations for the time-dependent speed-of-light propagation of electric and magnetic fields, rather than an electrostatic PIC code, which captures field effects via solutions to Poisson's equation. For computational efficiency, the field and particle computations within QS are performed on regular (structured) grids.

The kinds of effects that QS will model over the next few years, such as power-flow physics in Z-pinch accelerators (a device used in high-yield inertial-confinement fusion experiments) will require simulations with 100 million particles, 10 million grid cells, and hundreds of thousands of timesteps. The desire to run such models scalably on large parallel machines with thousands of processors has led us to develop the parallel version of QS discussed in this paper and address the load-balancing issues that naturally arise.

Other researchers have also long recognized the advantages parallel computation offers to PIC simulations. Notable implementations of parallel PIC algorithms and codes for distributed-memory machines include the following efforts. The GCPIC algorithm of Liewer and Decyk [5] has been widely cited for recognizing the potential need to partition particles and fields separately on a parallel machine to achieve load balance. The original implementation [5] was for an electrostatic PIC model with uniform particle distributions, partitioned in 1d within a single 1d block. No dynamic load balancing was needed but field values were redistributed to different processors for the FFT-based field solves.

Later implementations of GCPIC with 3d partitionings of a single-block problem were benchmarked with uniform particle distributions [6, 7]. The skeleton parallel PIC benchmarks of Decyk [8] also used the GCPIC algorithm for problems that did not require load balancing. A dynamically-balanced GCPIC implementation was presented by Ferraro et.al. [9] for an electrostatic PIC model with a single-block 2d domain. The particles were partitioned in 1d and the dynamic balancer adjusted the thickness of the 1d slices assigned to each processor as particle densities varied. Walker also load-balanced particles separately from grid cells via orthogonal recursive bisectioning [10] in single-block geometries using a kernel of the SOS electromagnetic PIC code as a test-bed [11].

Carmona et.al. [12] outlined a comprehensive taxonomy for partitioning and load-balancing strategies for parallel PIC simulations, with some options implemented in their single-block PICARD PIC code. In their nomenclature, the work we present here uses "semi-explicit" partitioning: the field grid is statically decomposed and the dynamic allocation of particles to processors is influenced by the grid partitions.

Parallelization strategies for at least two 3d electromagnetic PIC codes with production-scale capabilities similar in scope to QS have been discussed in the literature. Eastwood et.al. [13] described 3DPIC which uses multiple 3d body-fitted blocks of finite elements to discretize the simulation geometry. Large blocks are sub-divided, sub-blocks assigned to processors, and inter-block connections (glue patches) created and exchanged between processors, similar to what we describe in the next section. 3DPIC's performance was benchmarked on up to 1840 processors of the Intel Paragon with good scalability for problems with uniform particle distributions. Dynamic load-balancing strategies were not addressed in this work.

Recently, Blahovec et.al. [14] described ICEPIC which uses a single 3d structured-grid block

into which complex simulation geometries can be embedded. ICEPIC has a unique dynamic load-balancing capability which re-partitions the block as needed. Grid cells and particles are partitioned together in a weighted manner (a "semi-explicit hybrid" method in the Carmona taxonomy [12]) in 2d slices, which could normally cause either the cell or particle computation to become imbalanced. However, ICEPIC formulates its timestep in such a way that the field solve and particle push need not be synchronized [15]. Thus each of the two stages can be imbalanced so long as the total computation on each processor is balanced. The asynchronicity is achieved by overlapping communication and computation and by computing on a block's interior cells separately from its boundary cells (the ones that must be exchanged with other processors). This novel strategy balances the PIC computation effectively, at least for problems where particle computation is not dominant [14].

The load-balancing algorithm we present here is more in the spirit of GCPIC, with grid cells and particles balanced independently, but in such a way as to minimize extra communication. Preliminary versions of this work were presented in [16, 17]. Here we provide concise details of how a multi-block PIC code such as QS was effectively parallelized in Section 2, along with benchmark results for problems with uniform particle distributions. In Section 3 we present our new load-balancing idea. When the particle load on one or more processors becomes heavy, particles within sub-regions (windows) of those processor's domains are migrated to lightly loaded processors, but the field grid partitioning is not altered. By dynamically creating and destroying windows of various sizes, the particle push can be balanced independently from the field solve. The net effect is better overall load balance and parallel performance, as illustrated in Section 4.

2 Parallel Strategy

A QUICKSILVER (QS) simulation geometry consists of one or more 3d grid blocks aligned with the xyz axes. (QS also supports cylindrical and spherical coordinates.) Blocks can connect arbitrarily, but must adjoin each other perfectly with no overlap. Each block contains a topologically regular 3d mesh, which is also aligned with the coordinate axes, though the grid spacings may be non-uniform in x, y, or z. A grid cell is thus a small hexahedral element.

An example geometry with two blocks (solid lines) is illustrated (in 2d) at the left of Figure 1. As shown in the figure, an important restriction on the grids within each block (dotted lines) is that grid lines must be continuous across block boundaries. This means that the 6 faces of each grid cell always adjoin either another grid cell (in the same or an adjacent block) or an external boundary. QS supports a variety of external boundary conditions which impose various effects on fields and particles: absorptive and reflective surfaces, periodic boundaries, transverse electromagnetic inlet planes, perfectly matched layers, transmission line ports, etc. Individual grid cells can also be assigned material properties, treating them as a conductor or dielectric medium.

As in all PIC codes, field quantities (electric field $\vec{E}$, magnetic flux density $\vec{B}$) are stored and solved for on the grid, as are particle-averaged quantities (current density $\vec{J}$, charge density ρ). Particles can be pre-loaded or created within the geometry due to boundary conditions or physical effects such as beam injection or space-charge-limited field emission. Each particle moves in a continuous fashion through the geometry and transparently across block boundaries, but does not cross external boundaries. Particles can be deleted due to interactions with internal conductive surfaces or external boundaries.

The two major computations within each timestep of a PIC simulation are the field solve and

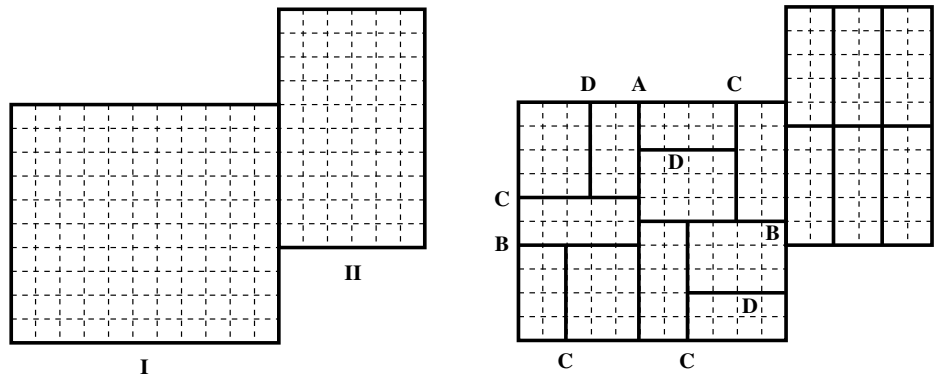

Figure 1: (Left) A 2d schematic of a simulation geometry with two adjoining blocks. (Right) For parallel execution, each block is divided into sub-blocks (solid lines). The subdivision is performed in a regular manner in the smaller block and recursively within the larger; initially a single cut A is made, then two B cuts, etc.

particle push. QS uses a finite-difference time-domain (FDTD) method (explicit [18] or implicit [19]) to solve Maxwell's equations and advance $\vec{E}$ and $\vec{B}$ as a function of their previous-timestep values and the previous-timestep particle current density $\vec{J}$. In the explicit case each grid cell updates its field values using information from its 6 adjacent grid cells. The implicit solver iterates several times on the same operation.

The particle push first performs a "gather" operation where the average $\vec{E}$ and $\vec{B}$ fields from the 8 corner points of the cell the particle is inside are used to interpolate a field value at the particle's current position. The particle's position and velocity are then updated via the relativistic form of Newton's second law where the Lorentz force $\vec{F}$ on the particle with charge q and velocity $\vec{v}$ is given by $\vec{F} = q\vec{E} + q(\vec{v} \times \vec{B})$. The final stage of the push is a "scatter" of the particle's charge back to the 8 corner points of the surrounding cell to form the scalar charge density ρ. Similarly, the path the particle traveled from its initial to final position during the timestep is used to scatter current density $\vec{J}$ to surrounding grid points in one or more cells.

Note that both the field-solve and particle-push operations are inherently parallel since each grid cell or particle can be computed on independently of all others. This is an attribute of collisionless PIC codes such as QS where particles do not interact with each other directly, but only indirectly through the particle-field formulation. Similarly, the gather operation is parallelizable over particles since the field arrays are only read from (not written to) during this computation. The scatter operation is also parallelizable over particles with the caveat that two (or more) particles cannot update the same grid array location simultaneously. In QS this caveat is avoided by each block having its own ghost cells that duplicate memory locations that could otherwise be simultaneously overwritten.

It is also important to note that for QS (and other PIC codes) to run with high parallel efficiency, all processors must own (nearly) equal numbers of grid cells and also own (nearly) equal numbers of particles. This is because the field-solve and particle-push operations within a timestep are computed sequentially, one after the other (see [14] for an exception). If only the sum of grid cells

plus particles is balanced across processors then parallel performance can suffer as some processors wait to push their particles while other processors finish their field solve and vice versa. The load-balancing algorithm we present in Section 3 addresses this issue.

The basic strategy used to parallelize QS builds on the multi-block nature of the original serial code. Each processor is assigned one or more blocks of grid cells along with all the particles that reside in those blocks. Since in a typical problem the original user-defined blocks will be of unequal size and there will be far fewer blocks than processors, a pre-processing step is performed to partition the original blocks into smaller sub-blocks, as illustrated at the right of Figure 1. Each block is split independently either into a regular N_1 by N_2 by N_3 array of sub-blocks (as in the smaller of the two blocks in the figure) or recursively into an arbitrary user-specified number of sub-blocks (as in the larger block into 11 sub-blocks). Cut A is first made perpendicular to the block's longest dimension to split it into 2 pieces, then each sub-piece is divided with cuts B, then cuts C are made, and so on recursively. By default, the partitioner attempts to create one sub-block per processor, each as cubic in shape as possible, as this will minimize inter-processor communication. But a larger number of smaller sub-blocks can also be created. If particle density varies over the simulation geometry this may help with load-balancing even without using the ideas presented in the next section.

Following partitioning, the assignment of sub-blocks to processors is made with the goal of giving each processor as equal a number of grid cells as possible. Depending on the partitioning each processor may be assigned one or more sub-blocks. For example the partitioned version of Figure 1 could be run on 17 processors (one sub-block each) or on 4 processors (4 or 5 sub-blocks on each). Both the partitioning and assignment operations are pre-computations performed only once since the QS grids do not change during a simulation. This procedure balances the field-solve computation once and for all.

Note that the collection of sub-blocks adjoin each other in conceptually the same way that the original blocks adjoined. The 17-block formulation of Figure 1 is as equally valid as the original 2-block formulation. The additional complication that parallel QS must handle is the exchange of field and particle information between two adjoining blocks owned by different processors. For field values, this is handled by forming inter-block connections as illustrated in Figure 2. Each component of each field resides at a particular point in a grid cell volume. Electric-field ($\vec{E}$) and current-density ($\vec{J}$) components are edge-centered quantities; magnetic-field ($\vec{B}$) components are face-centered; charge-density (ρ) resides at cell corners.

Once a field solve is performed, each block has new values defined on a 3d grid of points; the new points for edge-centered E_x values are shown as circles in the "send" block of the figure (illustrated in 2d). Similarly, each block needs to receive updated field values for a set of points it did not update itself. Typically these are "ghost" values (dotted cells) owned internally by other blocks as in the squares of the "receive" block of the figure. For the case of $\vec{J}$ values, a block will have computed both internal and ghost values (due to particles moving into its ghost cells) and will need to receive both internal and ghost values from another block and sum them to its corresponding values.

When the send and receive blocks adjoin in a particular way, a "connection" between them is defined as the set of overlapping circle and square points. The overlap values are those the processor owning the send block needs to send to the processor owning the receive block. The connection set is shown in the figure as triangular points for two different ways the send and receive block might adjoin. In the first case (overlap 1), the connection is a contiguous set of points; in the second case

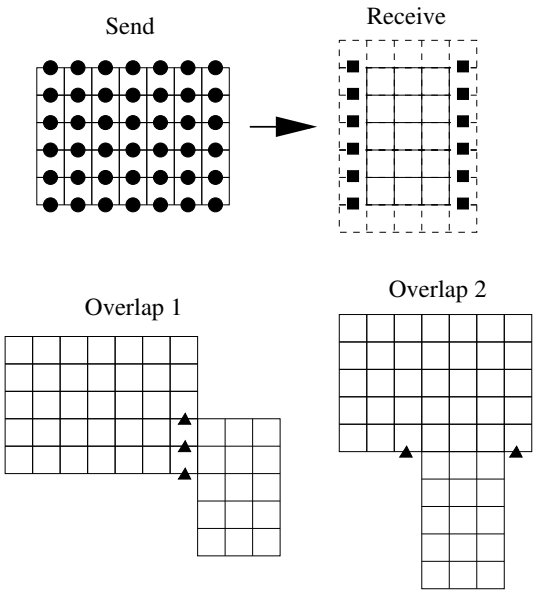

Figure 2: A 2d diagram of edge-centered E_x field components in two blocks. If the two blocks adjoin, then some circular points internal to the send block overlap with some square points in the dotted ghost cells of the receive block. The triangular points represent the overlap set. Two kinds of possible overlap are shown, one contiguous, the other disjoint.

(overlap 2) it is not. For 3d blocks the connections may take the form of one or more 2d planes or 1d lines or even single points. The sending and receiving processors both store the connection as one or more subsets of 3d indices - i.e., field(ilo:ihi,jlo:jhi,klo:khi) - that define what values to send and where to put (or sum) the received values.

This paradigm for block-connection has two nice features. First, on-processor and off-processor connections can be treated essentially the same. If both blocks reside on the same processor, the "send/receive" operation is simply an in-memory copy. If the blocks are on different processors, the operation requires a message be sent by one processor and received by another. Second, the same "connection" routine can handle multiple kinds of field values so long as it knows where in the grid cell the fields reside (edge, face, corner) and the kind of connection needed for each type of field (overwriting or summing).

In parallel QS, the block connection operation is implemented in two parts: a setup routine that computes the overlap connections for all pairs of blocks, and a communication routine that actually connects the blocks via sends and receives. The former operation is called only once since the grids in QS are static; the latter is called every timestep. The setup routine checks all pairs of sub-blocks, most of which do not adjoin. For those that do, the connection is computed and stored as sets of 3d indices. Each processor i then organizes its full list of block connections so that all the connections with another processor j can be packed into one message. Processor j will also compute its own connections for data to send to processor i.

The communication of field values is straightforward once block connections have been pre-computed. The communication takes place twice per timestep within QS. $\vec{E}$ and $\vec{B}$ field data are

communicated together after the field solve. $\vec{J}$ and ρ data are communicated together after the particle push. Note that due to the strategies described above for block-partitioning and assignment of sub-blocks to processors, each processor will typically be sending (and receiving) data from a few random other processors. On a distributed-memory parallel machine using the MPI message passing library [20], such unstructured communication is most efficiently done in an asynchronous fashion. First, each processor posts receives (MPI_Irecv) for all the messages it expects to receive. The processor then packs and sends all its outgoing messages, one per neighbor (MPI_Send). The processor can then perform all in-memory block connections for cases where it owns both the sending and receiving block. Finally, the processor waits for incoming field data. The MPI_Waitany routine will return when any incoming messages have arrived, at which point the processor can unpack the field data in that message. When all messages have been received, the block connection operation is complete.

The second issue that parallel QS must treat is particle migration between processors each timestep. During the particle push all particles that move into a block's ghost cells are flagged for migration if the ghost cell is owned by another processor. After the push is complete (gather, move, scatter), each processor counts the number of particles it needs to send to each of its neighboring processors. As with field communication, this is a small subset of the total number of processors. Each processor sends the count (which may be zero on a particular timestep) to its neighbors, which allows each processor to allocate memory for the particles it will receive. The communication of particles is now done asynchronously similar to how field values were communicated. Each processor posts a receive for each packet of incoming particles it expects. It then packs up and sends a single message containing all the appropriate particles to each of its neighbors. The processor waits for incoming messages to arrive and adds the received particles to its data structures.

We now present benchmark results for how QS performs on the Intel Tflops (ASCI Red) machine at Sandia. It is a distributed-memory massively parallel machine with 333 MHz Intel Pentium processors interconnected by an Intel-proprietary backplane and network interface chips. Figure 3 shows parallel efficiencies for four problems, two with field-only computations (squares), and two with particles and fields (circles). For the latter problems, the particle distribution was uniform across the simulation geometry, so these benchmarks do not test the load-balancing issues that will be discussed in the next section. For each case, a fixed-size problem was run on varying numbers of processors (solid symbols), as well as a scaled-size problem (open symbols) where the number of grid cells and particles doubled each time the number of processors doubled. In all 4 problems a single user-defined block was partitioned into one sub-block per processor.

The fixed-size fields-only benchmark (solid squares) used a single 80x100x96 grid block of 768,000 grid cells run for 2000 timesteps with an explicit time integration scheme. It is prototypical of a calculation that an analyst might perform on a single-processor workstation. A Poisson inlet boundary condition was applied to one face of the block, with perfect electric conductor, perfect magnetic conductor, and outlet conditions applied to the other faces. The block interior contained three conducting strips and several diagnostic outputs were also computed on the fly.

The parallel efficiencies for this problem were computed by dividing the one-processor run-time by the quantity (P times the P-processor run-time), where P is the number of processors. Parallel speed-up is simply P times the parallel efficiency. Thus, the 1024-processor run at 29.6% efficiency ran 303 times faster than on a single processor. The data show super-linear performance (efficiencies greater than 100%) on a few processors which is due to cache effects. On large numbers of processors the efficiency eventually decreases dramatically when the number of grid cells per processor becomes

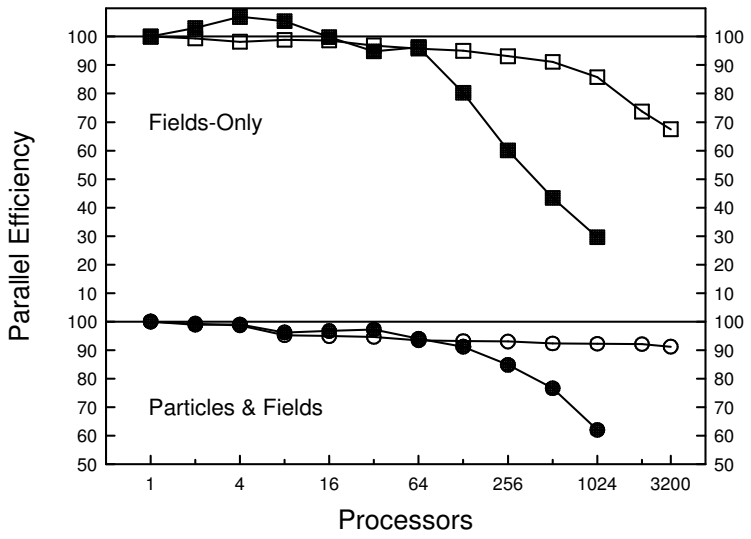

Figure 3: Parallel efficiencies for 4 benchmark simulations on the Intel Tflops: fixed-size fields-only (solid squares), scaled-size fields-only (open squares), fixed-size particles-and-fields (solid circles), and scaled-size particles-and-fields (open circles). The particle simulations have uniform particle distributions.

small (a few hundred for 1024 processors), due to the increased cost of field communication versus computation as the surface-to-volume ratio of each processor's sub-block increases.

The second fields-only benchmark (open squares) was a scaled-size problem. One large user block was specified for each run, so that when partitioned for P processors, each processor owned a 30x30x30 block of 27000 grid cells. Thus on one processor the global problem size was 27000 grid cells; on 3200 processors it was 86.4 million grid cells. The time integration, boundary, and diagnostic specifications were the same as in the fixed-size problem. This benchmark was run for 10000 timesteps to allow the waveform incident at the Poisson inlet to travel throughout the simulation domain. The parallel efficiencies for this problem only degrade to 70% on 3200 processors.

The one-processor timing data for these runs can be used to compute a "grind" time for fields-only calculations within QS. On a Tflops processor the explicit timestepping integrator required 1.5 microseconds per grid-cell per timestep. An explicit update in a single grid cell required about 60 floating-point operations. Hence a Tflops processor ran these benchmarks at about 40 Mflops (million single-precision flop/sec).

We also ran one larger billion-grid-cell fields-only calculation on 3200 processors of the Tflops machine. We estimated its parallel efficiency at 87.3% using the one-processor timings of the smaller fields-only runs as a reference point. In QS each grid cell in a fields-only calculation uses 25 single-precision words or 100 bytes of memory. The billion-cell calculation thus required about 100 Gbytes of storage. Each of the 3200 processors has 256 Mbytes of memory for an aggregate memory of 800 Gbytes. Thus this very large problem used about 13% of the Tflops memory and ran at a speed of 0.55 CPU-sec/timestep and a rate of roughly 110 Gflops.

The third set of efficiency data plotted in Figure 3 is for a fixed-size particle calculation. As

before, this benchmark was designed to be a problem size an analyst might run on a fast desktop workstation. A one-block grid of 64x64x64 = 262,144 grid cells was populated with 3.15 million particles. This particle/cell count of 12 is typical of many QS problems. Each grid cell was pre-loaded with 6 electrons and 6 positrons, each of which was given an initial velocity in a different coordinate direction $(\pm x, \pm y, \pm z)$ so that they moved approximately 1/2 grid cell per timestep. Mirror boundary conditions were applied to all 6 faces of the global domain. Because the number density associated with the particles was set to a small value and pairs of oppositely charged particles moved in the same direction (no net current), the particles in this problem were essentially non-interacting. Over time they stream back and forth within the block, reflecting off the mirror boundaries. The simulation was run for 256 timesteps with a 3-stage implicit integration scheme for the field solver. This means that an individual particle traverses the simulation domain twice to return (roughly) to its initial position. As in the fields-only problems, various diagnostic outputs were also computed on the fly.

The efficiency data for this fixed-size problem are plotted as solid circles in the figure. This run scaled better than its fields-only counterpart because more computational work is required to push particles on a per-grid-cell basis. On one processor the code spent 92% of its time in the particle push routines, and 7% in the field solve. In parallel, even with high-velocity particles causing a relatively large fraction of particles to migrate to new blocks and processors each timestep, the particle migration time was only a small fraction of the overall run time.

The fourth benchmark problem was a scaled-size particle simulation with each processor owning a 30x30x30 block of 27000 grid cells. As in the fixed-size problem, each grid cell was populated with 12 particles of two species, moving in all 6 coordinate directions. This problem was run for 200 timesteps with the same 3-stage implicit field solver as before. Parallel efficiencies for these runs are shown as open circles in the figure. Total grid cell counts ranged from 27,000 on one processor, to 86.4 million on 3200 processors. Similarly, total particle counts ranged from 324,000 on one processor to over one billion on all 3200 processors. With particle push costs dominating the run time, parallel QS exhibits excellent scalability with over 90% parallel efficiency for all numbers of processors.

In these benchmarks, QS uses 9 words (36 bytes) per particle and the same 100 bytes per grid cell. Thus the largest problem required about 46 Gbytes or roughly 6% of the memory available on 3200 Tflops processors. As before, the "grind" time for particle pushing can be computed from one-processor timings and is 8.6 microseconds per particle per timestep. Similarly, the 3-stage implicit field solve requires 7.8 microseconds per grid cell per timestep. A particle push requires approximately 355 floating-point operations; the implicit field solve takes 280 flops per grid cell. Thus a Tflops processor pushes particles in QS at a rate of 44 Mflops and performs implicit field solves at a rate of 36 Mflops. The billion-particle problem ran at an aggregate speed of 118 Gflops on 3200 processors.

3 Load Balancing

As discussed in the previous sections, load imbalance often occurs in parallel PIC simulations. This is because the field solve and particle push are separate expensive computations which are difficult to balance together. Efficient gather/scatter operations between the field grids and the particle positions impose the additional constraint that a processor should own grid cells and particles in the same geometric region. In parallel QS the field grids are partitioned evenly across processors.

However, particle densities can vary by orders of magnitude both spatially and temporally. Since particles often account for 90% of the total computational time in QS, this can have a serious impact on the code's overall scalability.

Since QS partitions the simulation domain into sub-blocks, we considered load-balancing by migrating entire grid blocks to new processors (an idea discussed in [13]) or repartitioning the global domain when imbalance occurs in some weighted fashion [14]. However, our judgement was that both these approaches would require significant coding effort and overhead within QS. Instead, we investigated a simpler idea, which was less difficult to implement yet effective.

The idea is to never change the balanced partitioning of the field grids, but to dynamically migrate particles from processors with too many particles to ones with too few via "windows" within a processor's sub-blocks. A "window" is a contiguous set of grid cells that is assigned to a new processor. Field solves in the window region continue to be performed by the original (parent) processor. Particles within the window region are pushed by the new (child) processor so long as they remain inside the window. Particles can enter and exit the window and migrate between parent and child in the usual way. Since field solves remain balanced across processors in this approach, the hope is that an appropriate choice of windows will also balance the particle push.

An example for a 3-processor single-block problem is shown in Figure 4. Each of the 3 processors initially owns one sub-block. If processor 1 (the parent) has too many particles, two (shaded) window regions are created within its block, one each for processors 0 and 2. Processor 1 will now push particles only in the remaining (unshaded) region of its block. Processors 0 and 2 will each push particles in two blocks, their original block and a new child block. All 3 processors continue to perform field solves in their original blocks.

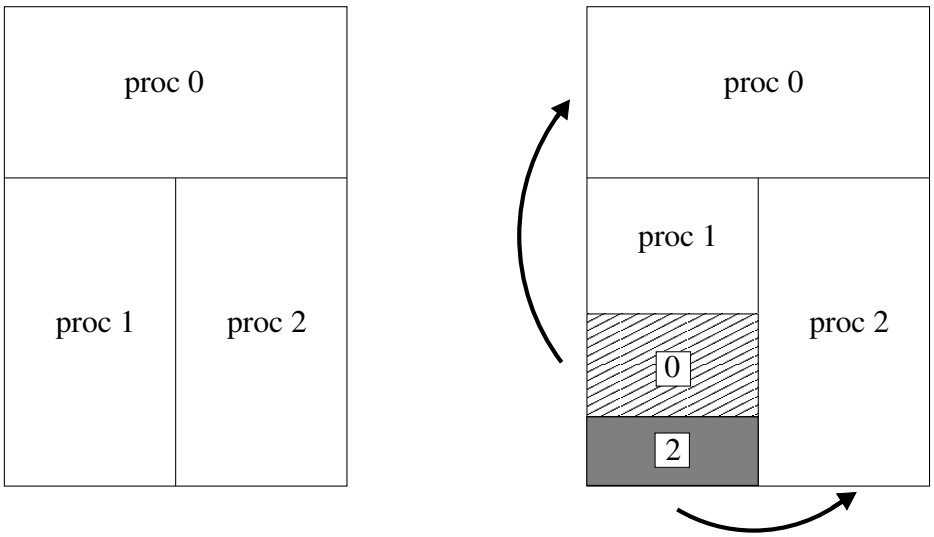

Figure 4: (Left) A three-processor decomposition of a single-block geometry with one sub-block assigned to each processor. (Right) Processor 1 has too many particles, so shaded regions of processor 1's sub-block are designated as "windows" and assigned to processors 0 and 2. Processors 0 and 2 push particles in the shaded regions to achieve better load-balance.

We now provide details as to how this idea is implemented within parallel QS. Before the particle push each timestep, particle imbalance is measured as the ratio of the maximum particle count on any processor to the average particle count. If this ratio exceeds a user-defined threshold, then a balancing routine is invoked and window regions are created.

In principle, window regions within a parent block could be of any size or shape. For simplicity, we restrict their shape to 2d slices perpendicular to one of the dimensions of a block (typically the longest dimension). For example, if a block is $10x20x15$ grid cells, then all window regions within that block will be $10xNx15$ in size, where N is some number of contiguous xz planes. To determine the optimal window sizes, the number of particles in each plane (in the partitioning dimension for that block) of every block are first counted. The list of counts is then concatenated across processors (via MPI_Allgatherv) so that every processor knows the entire set of counts for all block planes. Each processor also knows how many particles are owned by every processor (via MPI_Allgather).

With this information each processor can compute a global set of windows and their associated parent and child processors. It is a quick computation, so for simplicity it is done serially (each processor duplicates the computation) one window at a time. The processor with the most (least) particles is chosen as a parent (child) processor. The target number of particles to migrate from one to the other is the smaller of either processor's variation in particle count from perfect balance. Starting from each end of each of the parent processor's blocks, possible windows are checked, incrementing the window boundary, one plane of grid cells at a time. The total number of particles in the proposed window region is tallied (by summing the plane counts) and compared to the desired target. The optimal window is chosen as the one which migrates a number of particles closest to the target. This may be an entire parent block or a fraction thereof. After adjusting the particle counts of the parent and child processors, this procedure is repeated to create another window between a new pair of processors. Additional windows are created until the overall particle balance is within a user-defined threshold or until further improvements cannot be made.

Note that in this scheme, every processor with too many particles is a potential parent; every processor with too few is a potential child. Each parent processor may create one or more windows in one or more of its blocks. Likewise each child processor may be assigned one or more new child blocks associated with multiple parent processors. The balance routine creates the new communication patterns needed for these parent/child connections. Parent processors send child processors the information needed for them to push particles in the window region (grid geometry, dielectric and conductor flags, etc). Additional flags are set for grid cells in and around the window regions to identify which processor now owns each grid cell.

After windows are created, the particle push for this timestep takes place as usual. Because a parent processor now flags an entire window of grid cells as belonging to a child processor, this causes all particles within the window to migrate to the child. On subsequent timesteps, parent processors continue to compute the $\vec{E}$ and $\vec{B}$ fields for the window regions. Each timestep they must send those values to the appropriate child processors to enable them to push particles whose coordinates are within the windows. Similarly, as window particles are pushed, child processors accumulate $\vec{J}$ and ρ values on the grid of their new blocks. They must send that field data back to the parent processors at the end of each timestep to enable the parents to perform the field solve of the next timestep.

With windows in place, particle pushes occur exactly as in Section 2. Particles migrate freely between parent and child processors, or from child to child (e.g. across the boundary between child

blocks 0 and 2 in Figure 4). So long as overall particle counts stay balanced across processors (within the threshold), the windows persist. When imbalance is again detected, child processors set flags in their new window blocks so that all window particles migrate back to their parent processors during the next particle push. The window data structures are then deleted and the simulation can create new windows (if needed) on the next timestep.

We discuss the effectiveness of this load-balancing strategy in the next section, but first make a few additional comments on the algorithm:

- Parent processors push all particles on the first timestep after balancing is enabled, after which the appropriate particles migrate to child processors. The same one-timestep delay occurs when load-balancing is turned off (particles migrate back to the parent processors) and is then immediately turned on again. This means that if re-balancing occurs every N timesteps, there is one step out of N where particles are imbalanced (pushed only by parent processors), which can limit parallel efficiency. In most problems particle densities do not change so rapidly that this is an issue. Note that the windows described above are geometric regions, not groups of particles. If roughly equal numbers of particles move in and out of the window regions balance is unaffected and N stays reasonably large.

- There are two types of overhead cost associated with this load balancing strategy. There is a start-up cost when balancing is enabled to initialize the various arrays and perform the serial operations that compute window block pairings. In practice this is a small expense compared to the per-timestep cost of actually pushing all the particles. So long as load balancing stays enabled for several steps or more, this cost is also amortized over the duration of the balancing. There is also an every-timestep cost due to particle migration and extra field communication that occurs between parent and child blocks. Note that after the initial migration from parent to child, only particles crossing the boundary of the child block will migrate on subsequent timesteps. However, the entire volume of field values within the child block must be exchanged between parent and child processors each timestep. $\vec{E}$ and $\vec{B}$ values are sent from parent to child; $\vec{J}$ and ρ values from child to parent.

- The advantage of creating windows from contiguous regions of parent block grid cells is two-fold. Particle migration is reduced due to the minimal surface-to-volume ratio of the window region. Likewise the number of field values (per particle) that must be communicated between parent and child processors is also minimized.

4 Results

In this section, two sets of performance data are presented using the load-balancing algorithm within QS. The first set is for idealized fixed- and scaled-size problems that require either static or dynamic load-balancing. These problems use a single-block domain so that grid cell and particle counts can be easily specified. The second set is for a typical production-level QS simulation in a complex multi-block geometry.

Recall that in the particle benchmarks of Section 2, a one-block domain with mirror boundaries was pre-loaded with 6 sets of particles so that each cell contained 12 particles. Because each particle set filled the entire domain and moved in a different direction, particle density stayed uniformly constant throughout the global domain as the simulation progressed.

In the benchmarks of this section a similar domain was used, but the particle conditions were altered to induce load imbalance. Each of 3 sets of particles was loaded into only a fraction of the simulation domain but at a higher density so as to keep the total particle count the same. An illustration is shown (in 2d) in Figure 5. Two slabs of particles (A and B), shown in gray, are loaded into a one-block domain. The remaining white portion of the domain contains no particles; the small black region of overlap between the two gray slabs has the highest density of particles. If the particles in the 2 slabs are assigned velocities in the $\vec{s}$ directions (y direction for slab A, x direction for slab B), then the particle density distribution is static. If the particles are assigned velocities in the $\vec{d}$ directions then the density variation (and load-imbalance in the simulation) becomes dynamic. Slab A moves to the right, slab B moves up, and the small black overlap region moves diagonally through the domain.

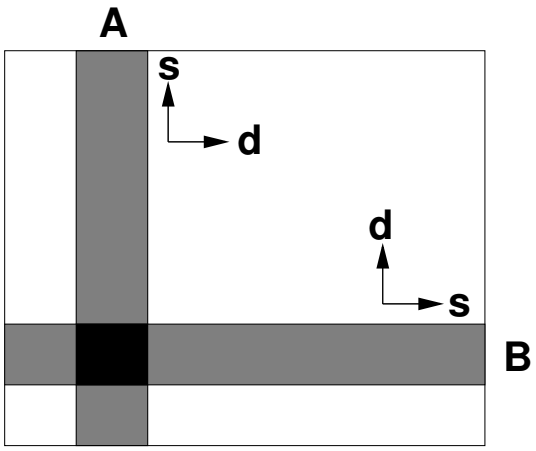

Figure 5: A single 2d block loaded with two gray slabs of particles, A and B, with the highest particle density in the black overlap region. If particles in each slab move in the $\vec{s}$ directions, the particle density remains statically imbalanced. If they move in the $\vec{d}$ directions, the imbalance changes dynamically.

For the QS benchmarks, the 3d analogue of this idea was used. For a fixed-size problem (same 64x64x64 grid as before), three slabs of size 16x64x64, 64x16x64, and 64x64x16 were filled initially at a 4x higher density than in the uniform case. For scaled-size problems, each processor owned a 30x30x30 block of grid cells and a particle compression factor of 10x was used. Thus on 8 processors, the global domain was a 60x60x60 grid and the shapes of the 3 particle slabs were 6x60x60, 60x6x60, and 60x60x6, with each slab loaded at a 10x higher density than in the corresponding uniform-density benchmark. The particles in each slab were assigned velocities in the appropriate xyz directions so that either a static or dynamic load imbalance resulted. In the dynamic cases the small cube of highest density (6x6x6 in the scaled-size problem on 8 processors) moves along the xyz diagonal of the 3d domain. As before, the velocities were set so that all particles moved approximately 1/2 grid cell per timestep.

The net effect of this strategy is two-fold. First, the total number of particles and grid cells in these problems is exactly the same as in the corresponding fixed- and scaled-size benchmarks of Section 2, so we can compare these performance results to those. Second, the distribution of

particles is very inhomogeneous. For the fixed-size problem, about 42% (27/64) of the cells in the simulation have no particles; another 42% of the cells have 16 particles/cell; another 9/64 have 32 particles/cell; and 1/64 of the cells have 48 particles/cell, for a total of 3.15 million particles. In the scaled-size case the imbalance is even greater. About 73% ($9^3/10^3$) of the global domain is devoid of particles, while 1/1000 of the grid cells have 120 particles/cell (10x the average).

In Figure 6 parallel efficiency data is shown for the fixed-size benchmark problems run on varying processor counts of the Intel Tflops machine for 256 timesteps, as before. The topmost curve (circles) is the same curve shown in Figure 3 for a uniform particle density across the entire domain. The lower curves (squares) are for two runs with no load-balancing; the solid squares are for the slabs of particles remaining stationary (static imbalance) while the open squares are for slabs moving through the simulation box (dynamic imbalance). In both cases the parallel efficiency of the QS simulation drops rapidly on increasing numbers of processors, to 20% on 1024 processors, meaning the simulation is only running about 200x faster than the one-processor timing.

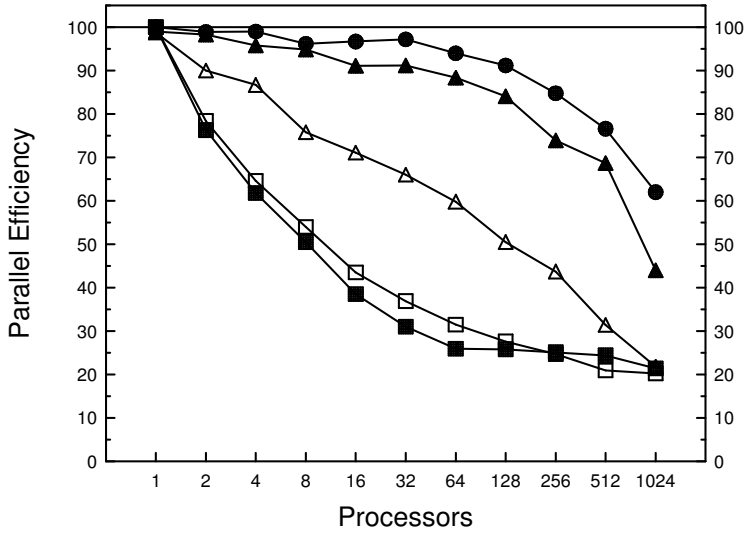

Figure 6: Parallel efficiencies for fixed-size particle simulations with and without load-balancing enabled. The lower curves (squares) are for load-balancing turned off; the intermediate curves (triangles) are for load-balancing turned on; the upper curve (circles) is the efficiency of the corresponding uniform-load problem (see Figure 3). Solid symbols are for problems with static imbalance; open symbols are for simulations with dynamic imbalance.

The middle curves (triangles) are the same runs as the lower curves but with load balancing enabled. The statically imbalanced problem (solid triangles) now runs with nearly the same efficiency as the original uniform particle distribution case (circles). Parent-child windows are created initially and persist for the duration of the simulation since the particle density variation is static. The efficiency of the dynamically imbalanced problem (open triangles) improves on all processor counts except 1024. In the middle of the processor range (8-128 processors) the run-time improvement due to the load balancing is about a factor of two (e.g. 35% versus 70% efficient on 32 processors). At the higher processor counts, processor sub-domains become smaller (e.g. 8x8x8 on 512 processors)

and the fast-moving regions of high particle density cause the imbalances between processors to fluctuate more rapidly. For example, on 32 processors the 256-timestep run re-balanced 21 times; on 512 processors re-balancing occurred 66 times. This has a significant impact on the overall parallel efficiency, since on 512 processors the particle push is thus wholly imbalanced (only parent processors push) on 26% of the timesteps (66 out of 256).

In Figure 7, parallel efficiencies are given for the scaled-size versions of the same benchmark, run for 200 timesteps. The smallest problem (one processor) has 27000 grid cells and 324,000 particles; the largest (1024 processors) has about 27.6 million grid cells and 332 million particles. As before, the curve of circles is from Figure 3 for uniform density problems of the same size. The parallel efficiency for runs without load balancing (solid and open squares) quickly degrades to nearly 10%, reflecting the fact that particle slab widths are only 10% of the domain size in these problems (versus 25% in the fixed-size benchmarks).

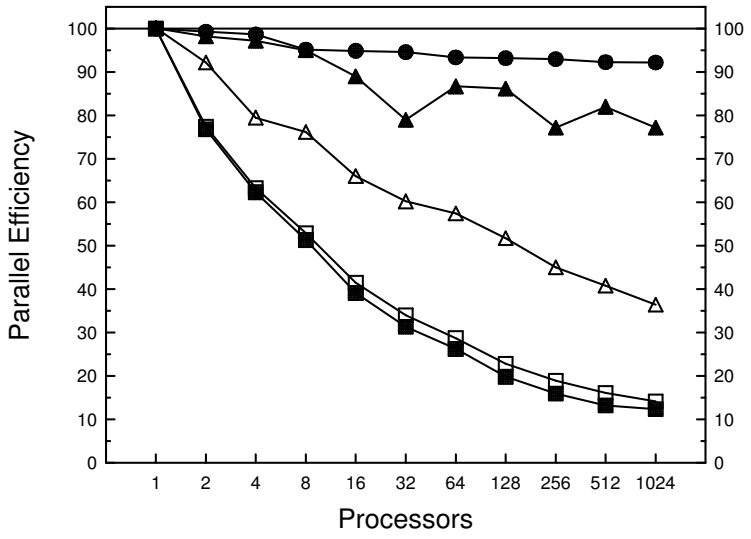

Figure 7: Parallel efficiencies for scaled-size particle simulations with and without load-balancing enabled. The meaning of the curve symbols is identical to that of Figure 6.

The runs with load-balancing enabled (triangles) again show marked improvement. The statically imbalanced problem (solid triangles) now runs nearly as efficiently as the problem with uniform particle distribution. Certain processor counts (32, 256) show a dip in parallel efficiency which is likely due to the limited set of possible window-block sizes that the load balancer had to choose from for particular grid and processor configurations. The dynamically imbalanced runs (open triangles) show improvement versus the unbalanced runs for all processor counts. On 1024 processors the actual run-time is about 2.6x faster (open triangle versus open square). Although the aggregate parallel performance on the largest problem is only 40% on 1024 processors, we emphasize that this benchmark is a stringent test of the load-balancing algorithm due to the high particle velocities and dramatic density variations within the simulation domain. For example, in the 128-processor run, the balancer was invoked 26 times so that the average lifetime of a set of created window blocks was only 7 timesteps.

Finally, we present performance results for a QS simulation of a post-hole convolute [21] whose 3d geometry is illustrated at the left of Figure 8. The post-hole convolute is a vacuum power-flow device used in high-current Z-pinch drivers [22] such as Sandia's Z accelerator. When the accelerator is fired, energy stored in Marx generators at the accelerator's periphery flows radially inward through water dielectric pulse-forming lines into four magnetically insulated transmission lines (MITLs). The double post-hole convolute adds the MITL currents in parallel through an arrangement of 12 anode posts that pass through holes in the cathodes (gray in figure), equally spaced in azimuth. This combined current then drives a Z-pinch at the accelerator's axis. Because of high electric fields at the electrode surfaces in the MITLs, electrons are emitted from the cathode surfaces (red in figure). The current delivered to the load is reduced if these electrons are lost to the anode. However, if the magnetic field generated by the current flowing in the transmission lines is sufficient to inhibit electrons from reaching the anode, the lines become magnetically insulated and the load can be driven efficiently. A 3d angular slice of the convolute is modeled in cylindrical coordinates with mirror boundary conditions to exploit the azimuthal symmetry of the full device. At the right of Figure 8, a snapshot of the QS simulation is shown in 2d cross-section with electrons emitted from the upper cathode in blue and from the lower cathode in red. The dotted lines denote 2d cross-sections of the 11 3d QS blocks defined to encapsulate the complex convolute geometry. Particles fill only portions of the blocks between conductor surfaces (dark lines) leading to large spatial variations in particle densities. This simulation contained 650,000 grid cells and was run for 500,000 timesteps (200 nanoseconds) on 80 processors of the Intel Tflops machine. Particle counts varied from zero at the beginning of the simulation to over 1.7 million.

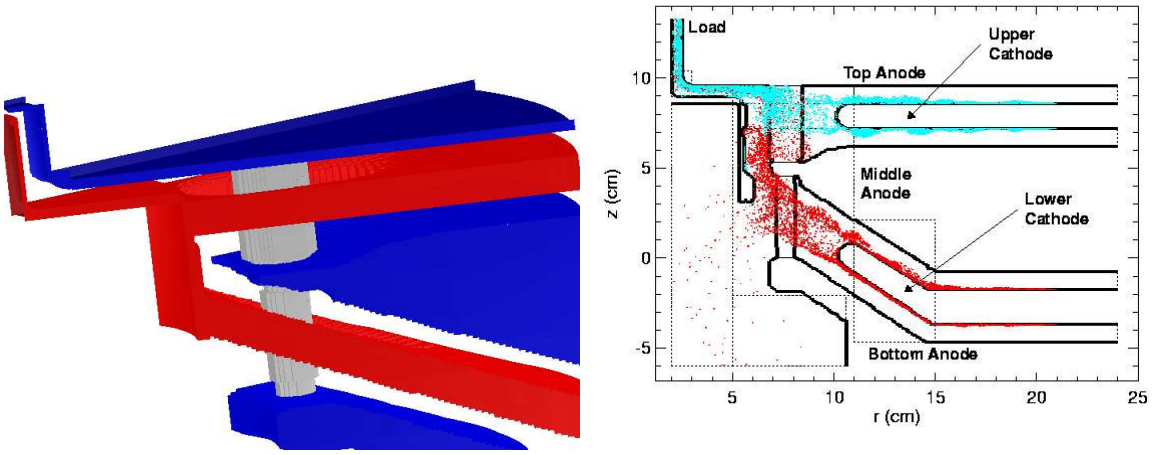

Figure 8: Simulation of an angular slice of a post-hole convolute. At left is the 3d geometry modeled by QS. At right is a 2d cross-sectional snapshot showing particle creation (blue and red) at the two cathode surfaces and mixing inside the convolute. Dotted lines indicate QS block boundaries; dark lines are conductor surfaces.

Figure 9 shows the effect of load-balancing during the simulation. The lower blue curve is total particle count (at each timestep) divided by the number of processors (80). Electron emission from

the cathode surfaces begins at about timestep 180,000. The red curve shows the maximum particle count on any processor at each timestep with load-balancing taking place. The apparent thickness of the red curve is due to oscillations between (nearly) perfect balance followed by growing imbalance until a threshold (1.35) is reached and re-balancing takes place. The red spikes are single timesteps where re-balancing occurred and the imbalance (max particle count divided by mean count) jumped dramatically upward. The time between re-balancings averaged 292 timesteps in this simulation; since data is only plotted for every 25th timestep many spikes are missing, but these data points are representative of the entire run. The green curve shows what the maximum particle count on any processor would have been if load-balancing were not used. This is tracked in the simulation by counting particles in child blocks as if they were owned by the parent processors, which is actually the case on the red-spike timesteps.

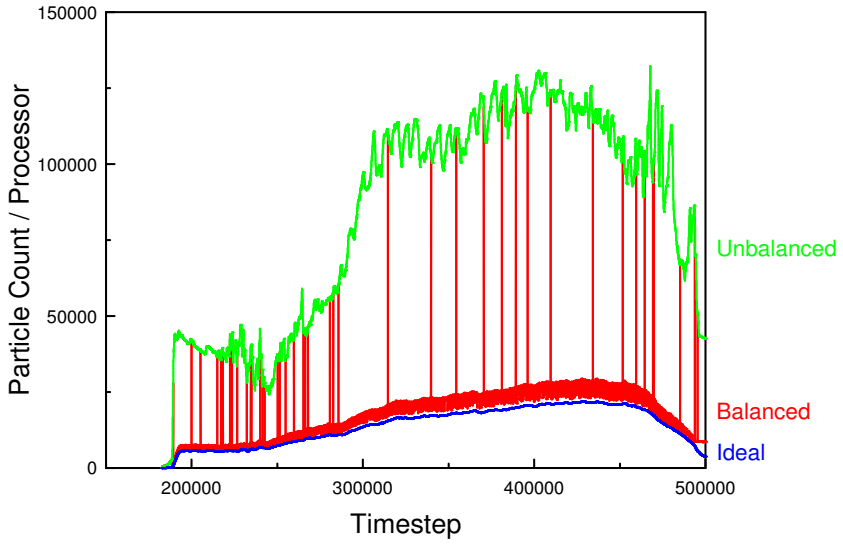

Figure 9: Particle counts in an 80-processor simulation of the convolute. The blue curve is average particle count per processor. The red curve is the maximum particle count on any processor with load-balancing enabled. The green curve is maximum particle count without load-balancing.

The data in Figure 9 indicate the benefit of dynamically redistributing the particles across processors as their density varies spatially and temporally. The small gap between the red and blue data is how close the load-balancer came to achieving perfect balance (max count equal mean). The gap between the green and red curves is proportional to the speed-up that load-balancing enabled. Overall, this problem ran with a cumulative particle load-imbalance (weighted by particle count at each timestep) of 1.23, where 1.0 is perfect balance. By contrast the green curve had a cumulative imbalance of 6.05, meaning the particle-push portion of QS ran about 4.9 times faster with load-balancing enabled than it would have without, assuming the load-balancing overhead costs are small (parent/child field communication each timestep and setup every 292 timesteps). Since the balanced simulation ran for 59 hours this is a significant savings*.

*The grind times for this simulation are larger than for the benchmarks of Section 2 due to more iterations in the implicit field solve and sub-cycling in the particle push.

5 Conclusions

Algorithms enabling a parallel implementation of the QUICKSILVER (QS) electromagnetic PIC code have been described. The new code retains the original features that have made serial QS a powerful simulation tool in the past. Parallel QS uses the same multi-block grid description as serial QS, which enables considerable flexibility in modeling general geometries. This strategy also leads to scalable parallel algorithms for inter-block field connections and particle migration.

Parallel QS also includes a novel load-balancing capability that allows field and particle data to be independently distributed across processors. For static particle imbalances the load-balancer can achieve parallel efficiencies nearly equal to those achieved for problems with uniform particle distributions. For dynamically imbalanced problems the improvements are not as dramatic but still provide a significant performance boost. As highlighted in the previous sections, the resulting parallel code can efficiently run very large PIC simulations on thousands of processors with a billion or more grid cells and particles.

One drawback of the current load-balancing algorithm is that when window regions are created, particles are pushed for one timestep in an unbalanced state, as they migrate to new processors. For simulations with rapidly changing particle densities, as load-balancing is turned on and off at high frequency, there will be lost efficiency due to the fraction of timesteps where particle pushing is (possibly severely) imbalanced. We have considered ideas for migrating particles "instantly" to new processors to avoid this one-step delay, but they involve other trade-offs whose effects are hard to predict.

The load-balancing ideas in this paper could also be applied to PIC codes which use unstructured grids. In such codes a processor's sub-domain is typically a clump of unstructured grid cells (e.g. tetrahedral or hexahedral finite elements). Creating window regions within such a domain to assign to a child processor would require the ability to identify a compact sub-clump of grid cells containing a desired number of particles. This is more complex partitioning problem than in the structured grid case where particle counts in planes of grid cells could be easily tabulated. A processor might invoke a serial dynamic partitioning algorithm on its local grid cells to create such windows [23]. The field data for the cells of the sub-clump would also need to be sent back and forth between parent and child processors. This communication of unstructured grid data is similar to what such a PIC code would already perform across processor boundaries within the global grid.

6 Acknowledgements

Many other Sandians have contributed to QUICKSILVER's development and been supportive of the work described here, including Mark Kiefer, Joe Kotulski, Ray Lemke, Paul Mix, Jeff Quintenz, Doug Riley, and Tim Pointon. Tim also supplied the convolute data discussed in Section 4. We have also collaborated for several years with PIC experts at NASA's Jet Propulsion Laboratory including Victor Decyk, Paulette Liewer, and Joe Wang; they have given us valuable insight into the parallel issues addressed in this report.

References

[1] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation.* Adam Hilger, Bristol, Philadelphia, 1991.

[2] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles.* Adam Hilger, New York, NY, 1988.

[3] D. B. Seidel, M. L. Kiefer, R. S. Coats, T. D. Pointon, J. P. Quintenz, and W. A. Johnson. Load-balancing a parallel electromagnetic PIC code. In *Computational Physics*, page 475. World Scientific, 1991. edited by A. Tenner.

[4] J. P. Quintenz, D. B. Seidel, M. L. Kiefer, T. D. Pointon, R. S. Coats, S. E. Rosenthal, T. A. Mehlhorn, M. P. Desjarlais, and N. A. Krall. Simulation codes for light-ion diode modeling. *Laser and Particle Beams*, 12:283–324, 1994.

[5] P. C. Liewer and V. K. Decyk. A general concurrent algorithm for plasma particle-in-cell simulation codes. *J. Comp. Phys.*, 85:302–322, 1989.

[6] P. M. Lyster, P. C. Liewer, V. K. Decyk, and R. D. Ferraro. Implementation and characterization of three-dimensional particle-in-cell codes on multiple-instuction-multiple-data massively parallel supercomputers. *Computers in Physics*, 9:420–432, 1995.

[7] J. Wang, P. Liewer, and V. Decyk. 3D electromagnetic plasma particle simulations on a MIMD parallel computer. *Comp. Phys. Comm.*, 87:35–53, 1995.

[8] V. Decyk. Skeleton PIC codes for parallel computers. *Comp. Phys. Comm.*, 87:87–94, 1995.

[9] R. D. Ferraro, P. C. Liewer, and V. K. Decyk. Dynamic load balancing for a 2D concurrent plasma PIC code. *J. Comp. Phys.*, 109:329–340, 1993.

[10] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors: Volume 1.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[11] D. W. Walker. Characterizing the parallel performance of a large-scale particle-in-cell plasma simulation code. *Concurrency*, 2:257–288, 1990.

[12] E. A. Carmona and L. J. Chandler. On parallel PIC versatility and the structure of parallel PIC approaches. *Concurrency*, 9:1377–1405, 1997.

[13] J. W. Eastwood, W. Arter, N. J. Brealey, and R. W. Hockney. Body-fitted electromagnetic PIC software for use on parallel computers. *Comp. Phys. Comm.*, 87:155–178, 1995.

[14] J. D. Blahovec, L. A. Bowers, J. W. Luginsland, G. E. Sasser, and J. J. Watrous. 3-D ICEPIC simulations of the relativistic Klystron oscillator. *IEEE Trans on Plasma Science*, 28:821–829, 2000.

[15] G. Sasser, J. Havranek, S. Colella, J. Luginsland, L. Kerkle, and J. Watrous. Modified PIC algorithm for efficient multiprocessor simulations. In *Proc of 16th International Conference on Numerical Simulation of Plasmas*, page 151, 1998.

[16] S. J. Plimpton, D. B. Seidel, M. F. Pasik, and R. S. Coats. Load-balancing techniques for a parallel electromagnetic particle-in-cell code. Technical Report SAND2000-0183, Sandia National Laboratories, Albuquerque, NM, 2000.

[17] D. B. Seidel, S. J. Plimpton, M. F. Pasik, R. S. Coats, and G. R. Montry. Dynamic load-balancing for a parallel electromagnetic particle-in-cell code. In *Proc of 13th Intl Pulsed Power Conf and 28th Intl Conf of Plasma Science*, pages P2–A08. IEEE Computer Society Press, 2001.

[18] K. S. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Transactions on Antennae Propagation*, 14:302–307, 1966.

[19] B. B. Godfrey. Time-based field solver for electromagnetic PIC codes, 1980. presented at 9th Conference on Numerical Simulation of Plasmas, Evanston, IL.

[20] Argonne National Laboratories. http://www-unix.mcs.anl.gov/mpi/index.html.

[21] T. D. Pointon, W. A. Stygar, R. B. Spielman, H. C. Ives, and K. W. Struve. Particle-in-cell simulations of electron flow in the post-hole convolute of the Z accelerator. *Physics of Plasmas*, 8:4534–4544, 2001.

[22] M. K. Matzen. Z pinches as intense x-ray sources for high-energy density physics applications. *Physics of Plasmas*, 4:1519–1527, 1997.

[23] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184:485–500, 2000.