SANDIA REPORT

SAND2000-0183
Unlimited Release
Printed January 2000

Load-Balancing Techniques for a Parallel Electromagnetic Particle-in-Cell Code

Steven J. Plimpton, David B. Seidel, Michael F. Pasik, and Rebecca S. Coats

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2000-0183
Unlimited Release
Printed January 2000

Load-Balancing Techniques for a Parallel Electromagnetic Particle-in-Cell Code

Steven J. Plimpton
Parallel Computational Science Department

David B. Seidel, Michael F. Pasik, Rebecca S. Coats
Electromagnetics and Plasma Physics Analysis Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1111

Abstract

QUICKSILVER is a 3-d electromagnetic particle-in-cell simulation code developed and used at Sandia to model relativistic charged particle transport. It models the time-response of electromagnetic fields and low-density plasmas in a self-consistent manner: the fields push the plasma particles and the plasma current modifies the fields.

Through an LDRD project a new parallel version of QUICKSILVER was created to enable large-scale plasma simulations to be run on massively-parallel distributed-memory supercomputers with thousands of processors, such as the Intel Tflops and DEC CPlant machines at Sandia. The new parallel code implements nearly all the features of the original serial QUICKSILVER and can be run on any platform which supports the message-passing interface (MPI) standard as well as on single-processor workstations.

This report describes basic strategies useful for parallelizing and load-balancing particle-in-cell codes, outlines the parallel algorithms used in this implementation, and provides a summary of the modifications made to QUICKSILVER. It also highlights a series of benchmark simulations which have been run with the new code that illustrate its performance and parallel efficiency. These calculations have up to a billion grid cells and particles and were run on thousands of processors. This report also serves as a user manual for people wishing to run parallel QUICKSILVER.

Acknowledgements

As discussed in Sections 1 and 6, Gary Montry contributed in several ways to the key ideas and successful code implementations described in this report. We enjoyed and benefited from our many discussions with him about parallel PIC codes.

We appreciate the many other Sandians who have contributed over the years to QUICKSILVER's development and have been supportive of this parallelization effort. These include Mark Kiefer, Paul Mix, Tim Pointon, Jeff Quintenz, Doug Riley, Joe Kotulski, and Ray Lemke. The QUICKSILVER team at Sandia has 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. Finally, we thank our management, including our center directors Bill Camp and Don Cook, who have been supportive of this effort from its inception.

Contents

1 Introduction **7**

2 QUICKSILVER Overview **9**
- 2.1 Geometry . . . 9
- 2.2 Timestep . . . 10
- 2.3 Data Structures . . . 11

3 Parallel Strategy **12**

4 User Instructions for Parallel QUICKSILVER **14**
- 4.1 Setting up a Simulation . . . 14
- 4.2 Running Parallel QUICKSILVER . . . 17
- 4.3 Output . . . 17
- 4.4 Accuracy of Parallel Results . . . 17

5 Implementation Details **18**
- 5.1 MERCURY . . . 18
  - 5.1.1 Decomposition Algorithms . . . 18
  - 5.1.2 Output . . . 20
  - 5.1.3 Decomposition Strategies . . . 21
- 5.2 Input and Setup . . . 21
  - 5.2.1 Blocks and Grids . . . 21
  - 5.2.2 Parallel Initialization . . . 22
  - 5.2.3 Boundary Conditions . . . 22
  - 5.2.4 Output Commands . . . 24
  - 5.2.5 Final Setup . . . 24
- 5.3 Fields . . . 25
  - 5.3.1 Setup . . . 25
  - 5.3.2 Connection . . . 29
  - 5.3.3 Block Surface Instabilities . . . 30
- 5.4 Particles . . . 32
  - 5.4.1 Serial vs Parallel . . . 32
  - 5.4.2 Modifications for Parallel QUICKSILVER . . . 32
- 5.5 Output . . . 34
  - 5.5.1 HISTORY Commands . . . 34
  - 5.5.2 SNAPSHOT and KPWRITE/KPSAVE Commands . . . 35
- 5.6 Unsupported Features . . . 36

6 Load Balancing **37**
- 6.1 Other Ideas . . . 37
- 6.2 Our Solution . . . 38
- 6.3 Details of Initiation . . . 39
- 6.4 Details of Operation . . . 42

7 Benchmark Calculations **44**
- 7.1 Performance and Scalability . . . 45
- 7.2 Static Load-Balancing . . . 48
- 7.3 Dynamic Load-Balancing . . . 51

8 Conclusions **54**

A Appendix **59**
A.1 Modified QUICKSILVER Commands 59
A.2 New QUICKSILVER Commands 60
A.3 MERCURY-Generated QUICKSILVER Commands 62
A.4 PDS2PFF File Conversion Utility 63

List of Figures

1 An idealized 2-d cross-section of a QUICKSILVER geometry. 10
2 Computational stages of a QUICKSILVER simulation. 10
3 A hexahedral QUICKSILVER grid cell with its associated field quantities. 12
4 The indexing convention for a 2-d QUICKSILVER grid block. 13
5 A 2-d schematic of MERCURY's recursive decomposition procedure. 20
6 Stages of creating a "plan" for field connections between all blocks. 26
7 A 2-d diagram of edge-centered E_x field components in two blocks. 27
8 Load-balancing of a block decomposition for three processors. 39
9 Steps to initialize particle load-balancing in a parallel QUICKSILVER simulation. 40
10 QUICKSILVER timestep with load-balancing enabled. 43
11 Parallel efficiencies for the data in Tables I-IV 50
12 Parallel efficiencies for the data in Tables V-VIII 55

List of Tables

I CPU time and parallel efficiency for a fields-only simulation of fixed size. 45
II CPU time and parallel efficiency for a fields-only simulation of scaled size. 46
III CPU time and parallel efficiency for a particle simulation of fixed size. 48
IV CPU time and parallel efficiency for a particle simulation of scaled size. 49
V Performance of a fixed-size particle simulation with static imbalance. 51
VI Performance of scaled-size particle simulations with static imbalance. 52
VII Performance of a fixed-size particle simulation with dynamic imbalance. 53
VIII Performance of scaled-size particle simulations with dynamic imbalance. 54

1 Introduction

Plasma simulation via particle-in-cell (PIC) methods has a long history extending back nearly 40 years to the beginning of scientific computing. Over the ensuing decades many practitioners have developed a rich set of numerical and computational techniques useful for simulating a variety of plasma phenomena [2, 5].

At Sandia, an interest in modeling plasma effects and understanding experiments within the pulsed-power group led to the development of the QUICKSILVER package [3, 15]. QUICKSILVER (QS) is a 3-d, finite-difference, fully-relativistic, particle-in-cell code[1] which has been used both inside and outside Sandia to simulate ion and electron diodes, magnetically insulated transmission lines, microwave devices, electron beam propagation, and high-current plasma devices. It represents 15-20 person-years of development effort over the last 14 years. 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.

PIC codes such as QS can be extremely compute-intensive, employing a few million particles and grid cells simulated for thousands of timesteps to capture necessary physical effects. On high-end workstations and traditional vector supercomputers (the platform for which QS was originally designed) such simulations often run for many hours or days. This naturally motivates a need for a parallel computing capability. This has two beneficial effects. First, computations of this scale can be performed more quickly. More importantly, particle and grid-cell counts can be expanded dramatically so that 3-d complexity in novel geometries can be modeled with increased fidelity.

As an example, within DOE's Accelerated Strategic Computing Initiative (ASCI), Sandia is tasked with simulating plasma effects in neutron generators, a key weapon component. Other QS applications include the understanding of system-generated electromagnetic pulse (SGEMP) effects on weapon components and the understanding of power-flow physics in Z-pinch accelerators, a device for high-yield inertial-confinement fusion. Back-of-the-envelope estimates predict that full 3-d PIC models of plasma effects in these devices could easily require 100 million particles and 10 million grid cells, simulated for 100,000 timesteps. The ASCI program has sited massively parallel supercomputers at Sandia, Los Alamos, and Livermore for performing such simulations, which will require parallel PIC codes capable of running scalably on thousands of processors.

With this background, the goal of this LDRD project was two-fold: (a) to produce a fully-functional and highly-scalable parallel implementation of QS suitable for running very large PIC simulations on 1000s of processors, and (b) to implement a strategy in the parallel code to address a fundamental parallel performance bottleneck with PIC codes – that of load-imbalance due to spatial and temporal inhomogeneities in particle densities within the simulation domain. When combined with the need to statically balance field updates across processors for the stationary grid and the fact that particles must exchange field information with nearby grid points twice each timestep (gather/scatter), these particle density fluctuations can pose a serious performance challenge to parallel implementation of a code such as QS.

The work of the LDRD was broken into two pieces. One effort, led by Gary Montry, a contractor with Southwest Parallel Software, Inc. (http://www.spsoft.com), implemented a parallel version of QS for single-block geometries (blocks are discussed in the next section). This allowed rapid experimentation with field and particle kernels and with different parallelization strategies. It also led to new load-balancing ideas which are discussed later in this report. The second thrust of the LDRD was to create a true multi-block parallel QS; the resulting code is the subject of this report.

[1]The QUICKSILVER "package" is really a suite of codes which includes QUICKSILVER itself as well as pre- and post-processing tools.

Other researchers have also long recognized the advantages parallel computation offers to PIC simulations, since the pushing of particles and the advancing of field quantities are inherently parallelizable operations. Notable implementations of structured-grid parallel PIC algorithms and codes include the following:

(1) General Concurrent PIC (GCPIC) algorithm in a 1-d electrostatic PIC code [9]: Uses FFT-based field solves and a standard particle-push formulation. Employs only a single 1-d block with each processor owning a spatial sub-domain of particles and grid cells. Dynamically compensates for unequal particle distribution by adjusting the sizes of each processor's sub-domain. Achieved excellent speed-ups on early Intel hypercube machines for up to 32 processors.

(2) GCPIC algorithm in a 3-d electrostatic PIC code [10]: Uses FFT-based field solves and standard particle pushes. Employs only a single block, spatially decomposed in a 1-d, 2-d, or 3-d fashion across processors. Only tested with uniform distributions of particles. Achieved good speed-ups on up to 512 processors of the Intel Delta (predecessor to the Intel Paragon).

(3) Skeleton codes (kernels) for 3-d electrostatic PIC [4]: Uses FFT-based field solves and standard particle pushes. Employs only a single block, spatially decomposed across processors. Load-imbalance was not tested since particle densities only varied by 10%. Achieved good speed-ups on several distributed memory parallel machines, including the Intel Paragon, Cray T3D, and TMC CM-5.

(4) 3-d electromagnetic PIC code [21]: Uses a field-solve and particle-push formulation similar to QUICKSILVER. Employs only a single block with uniform particle distribution, spatially decomposed across processors. Achieved excellent speed-ups on up to 512 processors of the Intel Delta.

(5) Kernel of SOS electromagnetic PIC code [20]: Uses a field-solve and particle-push formulation similar to QUICKSILVER. Employs only a single block. Handles load-imbalance in particle distribution by assigning particles to processors separately from field grid cells. Load balances particles dynamically using orthogonal recursive bisectioning (ORB).

(6) 3DPIC electromagnetic PIC code [5]: Uses a field-solve and particle-push formulation similar to QUICKSILVER. Employs multiple body-fitted blocks, each composed of a topologically regular grid of hexahedral finite elements. Is parallelized by assigning one or more blocks with their particles to each processor. Has the potential for load-balancing via reassigning grid blocks and their particles to different processors, though it has not been implemented (to our knowledge). Achieved reasonable speed-ups on 1800 processors of the Intel Paragon for problems with uniform particle distributions.

Of these implementations, only the last one was for a true multi-block production-scale code, similar in spirit and scope to QS. Thus the decomposition and load-balancing methods employed in efforts (1)-(5), while educational for us, were not directly applicable to our work with QS. As will be discussed in Section 2, we adopted a philosophy similar to that of effort (6) of parallelizing at the block level, assigning different grid blocks with their particles to processors. For reasons that will be discussed in Section 6 the idea proposed in effort (6) for achieving better load-balance by migrating entire blocks to new processors did not seem to be the best option for QS; we opted instead for a novel idea whereby only particles within a "window" region of a grid block migrate to another processor, but not the grid cells themselves. By dynamically creating and destroying multiple windows of various sizes, we are able to balance the particle push separately from the field update. The net effect is better load balance and parallel performance, as will be illustrated in Section 7.

There has also been work at Sandia on PIC algorithms for unstructured grid geometries (and elsewhere, see [17, 7] for example). The VOLMAX code authored by Doug Riley and David Turner encapsulated many of the algorithms needed for performing field updates on hybrid structured/unstructured grids [13]. This code was used in an LDRD effort led by Dave Seidel to create a hybrid PIC capability that led to the QS/VOLMAX code [16]. These projects developed single-processor and shared-memory parallel codes. Recently, an ASCI-funded effort to create a new parallel unstructured-grid electromagnetic PIC code has been undertaken by Joe Kotulski and others on the VOLMAX team. To date, they have successfully implemented parallel field-solution algorithms in a new version of VOLMAX.

The remainder of this report is structured as follows. In the next section a brief overview of the computational kernels and data structures used in parallel QS is given. In Section 3, our basic strategy for parallelizing QS is outlined. Section 4 describes how to formulate problems for and run the parallel QS code; this section may be the only one that users of the code wish to read. Section 5 is the most detailed of this report; it provides a concise explanation of all the parallel algorithms used in the new code as well as a summary of all the changes made in both QS and the MERCURY pre-processor to enable a parallel implementation. This section is intended to be a reference for current and future QS developers, so that with the section as a guide and by reading the source code and its comments, they can (hopefully!) deduce what the parallel modifications to the code are designed to do. In section 6 we discuss the load-balancing strategies that were implemented in parallel QS. In section 7 we highlight several benchmark calculations we performed on the Intel Tflops and CPlant machines at Sandia to test the new code's accuracy, performance, and scalability. In Section 8 we offer some conclusions and plans for future work. Finally, an Appendix is included which lists additions and changes to the set of valid QS input commands that were made for parallel QS.

2 QUICKSILVER Overview

In this section we highlight the computational features and basic data structures of the original serial QUICKSILVER (QS) code that are relevant to understanding the parallelization effort described in this report. More detailed descriptions of serial QS can be found in [15].

2.1 Geometry

QS performs its computations within a simulation geometry conceptually similar to that shown in Figure 1. Note that although the sketches in this report are typically 2-d for simplicity; all the attributes discussed extend in the obvious way to the 3-d QS code.

A QS geometry consists of one or more user-defined grid "blocks", outlined with thick borders in the figure. The blocks may be connected arbitrarily, but must be conformal in the sense that they adjoin each other perfectly with no overlap. Each block contains a topologically regular 3-d mesh (thin lines), which is aligned with the coordinate axes, though the grid spacings may be non-uniform in x, y, or z. (QS also supports cylindrical and spherical coordinate systems.) Thus a grid "cell" is a small hexahedral element. An important restriction on the grids in each block is that grid lines must be continuous across block boundaries, as shown in the figure. For computational efficiency, regions outside the block volumes are not treated by the QS simulation.

Each grid cell has 6 surfaces, and each of these surfaces is of one of 3 types: it adjoins another grid cell in the same block, a grid cell in a different block, or an external surface. All external surfaces must have boundary conditions assigned to them. QS supports a wide variety of these conditions which impose different effects on fields and particles: absorptive and reflective surfaces, periodic boundaries, TEM inlet

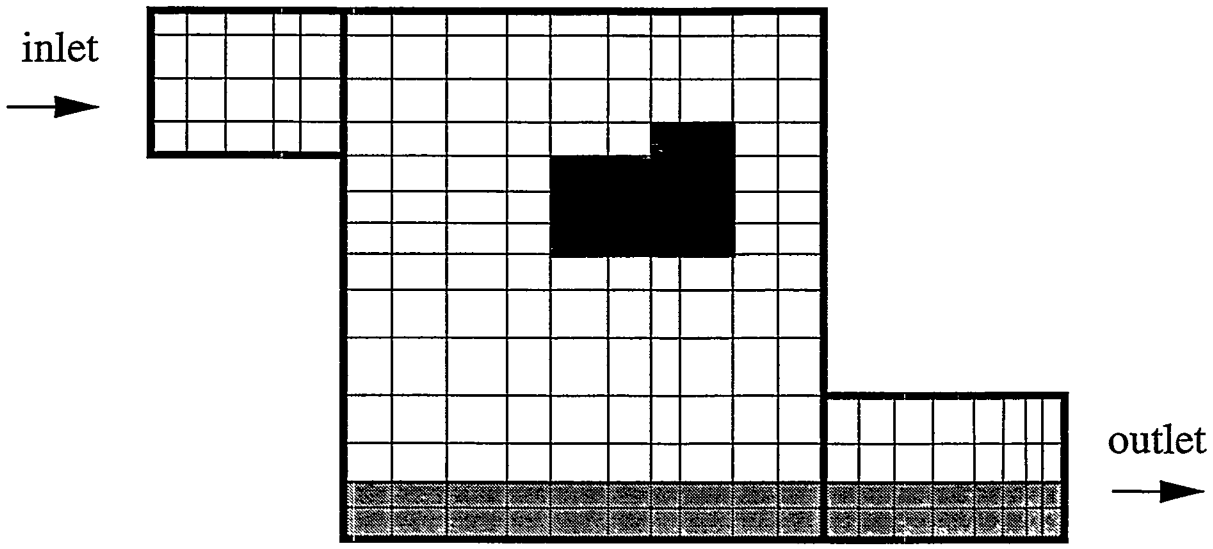


Figure 1: *An idealized 2-d cross-section of a* QUICKSILVER *geometry. Dielectric regions are shaded; conductor regions are black.*

planes, transmission line ports, etc. Inside a block, individual grid cells may also be assigned material properties, such as being part of a conductor or dielectric medium.

Particles can be pre-loaded or created within the QS 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 but can always be located uniquely within a particular grid cell. Thus a particle can move transparently across a block boundary to another block, but cannot cross an external boundary. Particles can be deleted due to interactions with conductive surfaces or external boundary conditions.

2.2 Timestep

Logically, a QS simulation proceeds through the stages listed in Figure 2.

(1) Problem initialization and setup

(2) Loop over timesteps:

- (2a) Leapfrog update of $\vec{E}$,$\vec{B}$ fields on grid
- (2b) Create new particles
- (2c) Advance particle positions
- (2d) Delete particles as required
- (2e) Accumulate particle charge Q and current $\vec{J}$ on grid
- (2f) Output of desired diagnostics

Figure 2: *Computational stages of a* QUICKSILVER *simulation.*

In step (1), the problem geometry, boundary and initial conditions, and requested outputs are defined. Step (2) is the computational heart of the code — the timestep loop.

In step (2a), Maxwell's equations

$$\frac{\partial \vec{E}}{\partial t} = \frac{1}{\mu\epsilon}(\nabla \times \vec{B}) - \frac{\vec{J}}{\epsilon} \tag{1}$$

$$\frac{\partial \vec{B}}{\partial t} = -(\nabla \times \vec{E}) \tag{2}$$

for the electric $\vec{E}$ and magnetic $\vec{B}$ fields are solved, where μ and ϵ are the permeability and permittivity of free space, respectively. QS uses a finite-difference time-domain (FDTD) method (explicit [22] or implicit [6]) to advance the fields 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 adjacent grid cells. The implicit solver iterates several times on the same operation.

In steps (2b-2d) particles are created, pushed, and deleted (as necessary). The particle push involves a "gather" operation where the average $\vec{E}$ and $\vec{B}$ fields from the 8 corner points of the particle's cell are used to interpolate a field value at the particle's current position. Then the particle's position and velocity are 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

$$\vec{F} = q\vec{E} + q(\vec{v} \times \vec{B}) \tag{3}$$

Following the particle push, in step (2e) the final particle position is used to "scatter" charge density Q back to the 8 corner points of the particle's cell. Similarly, the path the particle traveled from its beginning to final position during the timestep is used to scatter current density J to surrounding grid points in one or more cells.

Finally, in step (2f), various diagnostic quantities can be computed and output to files as desired. These include snapshots of subsets or all of the particle and field arrays, as well as time- or spatial-averaged quantities, such as line or surface integrals over specified field components.

2.3 Data Structures

Each grid cell in the QS simulation has a unique i,j,k index within a block m. Several field quantities are associated with each cell as illustrated in Figure 3. Each component of each field resides at a particular point within the cell volume. As shown in the figure, electric-field ($\vec{E}$) and current-density ($\vec{J}$) components are edge-centered quantities, while magnetic-field ($\vec{B}$) components are face-centered. Scalar charge-density (Q) and average-field components ($\vec{E}_{ave}$ and $\vec{B}_{ave}$) used for interpolating fields to individual particles, are located at cell corners. Other quantities associated with the cell itself, such as its conductor or dielectric status, apply to the entire cell volume and are treated as cell-centered quantities.

QS stores each grid-based quantity (e.g., a field component) as a collection of 3-d arrays, one per grid block.[2] As shown in Figure 4, the size of a 3-d array in a single block is determined by 3 quantities: *imax,jmax,kmax*. These specify the number of grid *points* (or lines) in each dimension, including the surface faces that bound the block. Thus the number of grid *cells* in a dimension is one less; there are $(imax - 1) \times$

[2] Actually, for memory management purposes, a field component is stored as one long linear array encompassing all blocks. But it is accessed by block index and 3-d spatial grid location as discussed here.

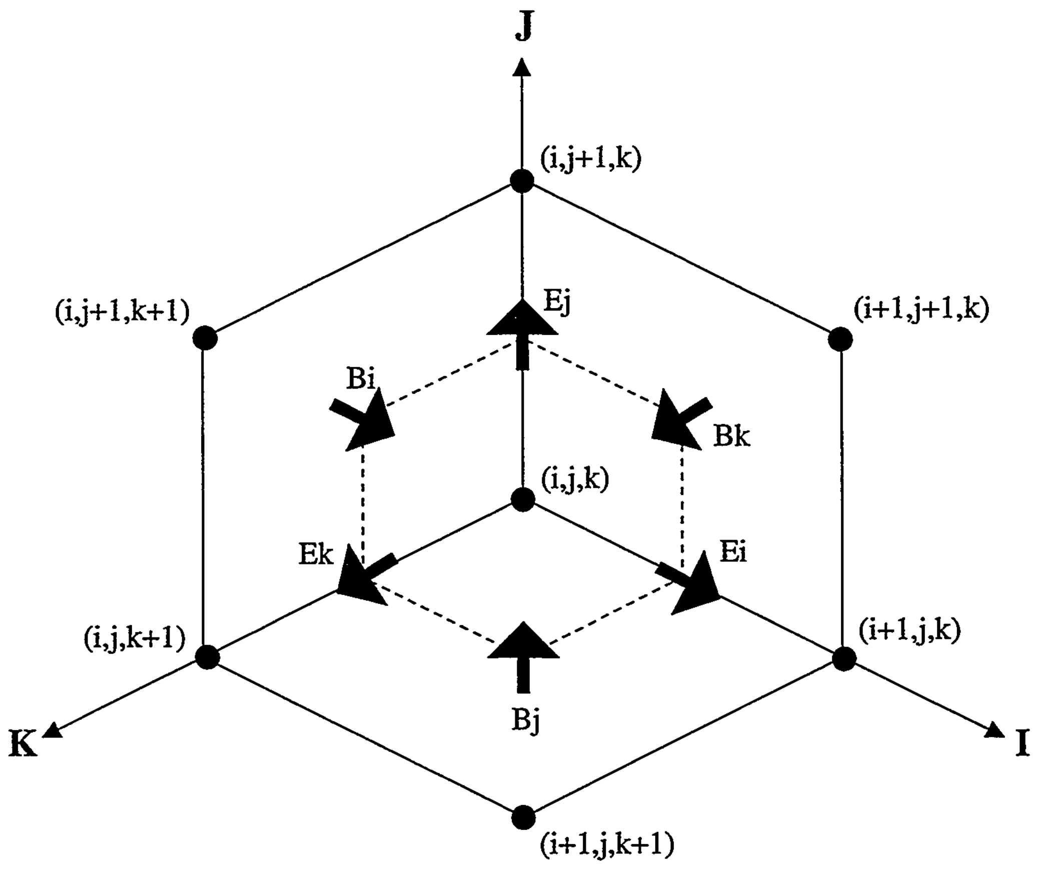


Figure 3: *A hexahedral* QUICKSILVER *grid cell with its associated field quantities. Grid point values are shown in parentheses; the dotted lines lie along half-grid spacings. All 6 field values are associated with the ijk grid cell. $\vec{E}$ field components are edge-centered within the cell; $\vec{B}$ field components are face-centered.*

$(jmax - 1) \times (kmax - 1)$ grid cells within the block (unshaded region in the figure). For convenience in the field update and particle push, the storage for each block also includes a layer of surrounding ghost cells. Thus the grid points of the extended block are indexed from 0 to $imax + 1$ (as shown in the figure) and the grid cells from 0 to $imax$.

Particle information (position, momentum, charge, ijk cell index, block number) within QS is stored separately from the grid arrays as a one-dimensional list. For flexibility in creating and destroying particles, this list is organized as a collection of fixed-length "caches" which can be allocated as needed. One "entry" in a particular cache contains all the information about a single particle.

3 Parallel Strategy

The key question that must be addressed for implementation of any PIC code on a distributed-memory parallel machine is how the field and particle data will be decomposed across processors. Before answering this question for QUICKSILVER (QS), it is worth noting several points.

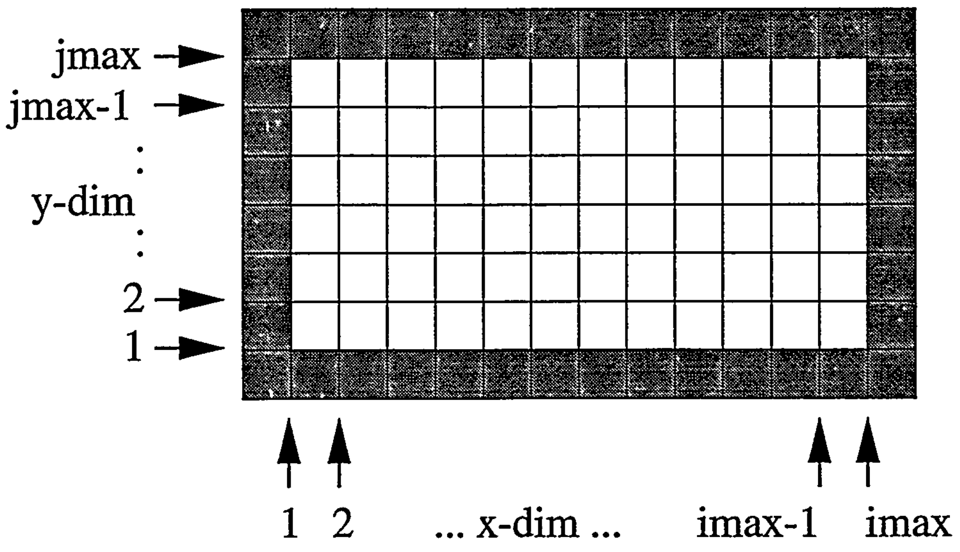


Figure 4: *The indexing convention for grid points of a 2-d* QUICKSILVER *grid block with its surrounding ghost cells (shaded).*

First, all of the major computational stages outlined in Figure 2 of the previous section involve either fields, particles, or interactions between them. The field update and particle push are inherently parallelizable since each datum (grid cell or particle) can be computed 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 particle-field interactions. Similarly, the gather operation (interpolation from fields to particles) is parallelizable over particles since the field arrays are only read from (not written to) during this computation. The scatter operation (interpolation from particles to fields) 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 not an issue, since each block has its own ghost cells that serve as duplicates of memory locations that could otherwise be simultaneously overwritten.

Second, to run QS 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 stages within a QS timestep are computed sequentially, one after the other. For example, it is not possible to have some processors updating all the fields at the same time other processors are pushing all the particles.

Third, we re-emphasize that the grid operations in QS are all block-based; for example, the field update routine is structured as a loop over blocks, with the *ijk* cell values within the block being updated as inner loops. Serial QS then invokes a sequence of routines that update ghost cell field values using connection information for block pairs that adjoin at faces, edges, and corners. The key point is that serial QS is already structured so that it can efficiently run a simulation containing multiple blocks of field values and all the particles inside those grid blocks.

Fourth, a QS simulation geometry can be partitioned into an arbitrary number of blocks and still represent the same physical model. (The question of whether two simulations using different block partitionings produce identical answers is discussed in the next section.) For example, the geometry of Figure 1 is illustrated as a 3-block simulation, but if each block were further sub-divided, it could have been formulated as a 10-block or 100-block simulation (so long as no block dimension is made smaller than 3 grid cells).

With these facts in mind, a natural strategy for parallel QS is to assign one or more blocks of grid cells to each processor along with all the particles that reside in those blocks. If the initial problem specification contains unequal-sized blocks or fewer blocks than processors, we can sub-divide the user blocks into smaller blocks as a pre-processing step. If this is done in such a way that each processor can be assigned (nearly) equal numbers of grid cells, then QS field updates will be load-balanced for the duration of the simulation, since the grids are static. If this grid partitioning also assigns equal numbers of particles to each processor, then the entire QS simulation is load-balanced. If not, then we still have a particle load-imbalance problem; this issue is addressed in Section 6.

The great advantage of this strategy is that the vast majority of serial QS does not have to be modified to work in parallel. On a single processor, the parallel QS code simulates all the grid blocks and all the particles in those blocks, i.e. the entire problem. Running on a parallel machine, each processor is still computing on a collection of grid blocks and the particles in those blocks, but they now comprise only a portion of the global geometry. So long as each processor can acquire boundary-condition information for its blocks from neighboring processors (e.g. ghost cell field values and incoming particles), then it can treat its blocks and particles as if they comprised the entire simulation domain.

Since our target architecture for parallel QS was distributed-memory parallel machines, we programmed in a message-passing paradigm. For portability we used the message-passing interface (MPI) standard [8]. This allows parallel QS to be run on any parallel machine which compiles standard Fortran (F77) and C and provides an MPI library. This includes all current-generation distributed-memory parallel machines (e.g. Intel Tflops, Cray T3E, IBM SP-2, workstation clusters) as well as shared-memory platforms (e.g. SGI Origin and DEC 8400). We also emphasize that parallel QS runs on any number of processors, including a single processor. Thus the parallel version of QS is also a serial code, which can be run on any Unix workstation, in which case it operates essentially identically to the original serial QS.

Our starting point for this parallelization effort was version 3.0 of serial QS, which contains approximately 100,000 lines of mostly F77 code (including comments). The existing code required small modifications in selected places for parallelization. For example, error checking had to be enhanced to allow for the (now) legitimate case of applied boundary conditions having no overlap with a particular processor's block(s). We also added about 10,000 lines of F77 and C code to the new parallel QS. These were primarily routines that implement new capabilities needed for parallel execution, such as the communication of ghost cell field values to different processors or the migration of particles from one processor to another. All of these changes and additions are detailed in Section 5. First, however we describe how to run the new parallel QS code, from a user's perspective.

4 User Instructions for Parallel QUICKSILVER

4.1 Setting up a Simulation

Instructions for creating serial QUICKSILVER (QS) input files and running serial QS are given in [3]. The normal procedure is to first run the pre-processor MERCURY. MERCURY enables the user to setup the problem geometry, define boundary conditions, specify outputs, etc. When MERCURY finishes it produces a "qcks.in" and "pvlx" file, both of which are inputs to serial QS.[3] The former contains a list of QS input

[3]More accurately, MERCURY produces a single output deck which is typically run through the *splitf* utility or *xqcks* script to produce the two QS input files.

commands that run the desired problem. The latter has a list of array bounds that QS uses for dynamic memory allocation. As serial QS runs it produces a variety of output files, many of them in a portable file format called PFF [14]. Post-processing analysis and visualization tools can then be run using the PFF files as input.

The steps for running parallel QS are similar with a few additional options. MERCURY is still used to create the desired problem geometry and simulation settings. There are two new MERCURY commands[4] which are used to tell MERCURY how to partition the grid blocks for a parallel run:

```
CUSTOM PROCESSORS P [assign]
CUSTOM DECOMPOSE N M
CUSTOM DECOMPOSE N MX MY MZ
```

For a parallel run, the "CUSTOM PROCESSORS P [assign]" command is mandatory, where $P \geq 1$ specifies the number of processors the problem will be run on. The optional [assign] argument specifies how the blocks will be assigned to processors. If used it must be one of 3 values: "sorted", "clumped", or "strided". The default value is "sorted", which will generally produce good results. For the interested user, Section 5.1.1 provides more details about the 3 options.

When the CUSTOM PROCESSORS command is used, MERCURY will decompose the user-defined blocks into sub-blocks, and assign them to the P processors. The decomposition procedure can be guided by the optional "CUSTOM DECOMPOSE" commands. If used, one must be specified for each user block. CUSTOM DECOMPOSE N M means chop user block N into M sub-blocks. CUSTOM DECOMPOSE N MX MY MZ means chop user block N with planar cuts along each of the 3 dimensions into MX by MY by MZ sub-blocks. If no CUSTOM DECOMPOSE commands are used, MERCURY will decompose the user blocks as best it can into P equal-sized sub-blocks, assigning one to each processor. For example, if 3 user blocks are decomposed for 100 processors, and one is twice as large as each of the other two, then MERCURY will chop the large block into 50 sub-blocks and the two smaller blocks into 25 sub-blocks each. If there are more user blocks than processors (and no CUSTOM DECOMPOSE commands are used), MERCURY will simply assign the blocks to the processors. More details on how these operations are performed are discussed in Section 5.1.1.

When finished, the MERCURY output in *qcks.in* for a parallel QS run will contain several new and altered QS commands (UBLOCK, UGRID, PROCESSOR, BLOCK). These are discussed in Section 5.1.2, but do not have to be understood to simply use parallel QS. MERCURY will also adjust the QS data array bounds so as to be appropriate for running on P processors; these new values are part of the *pvlx* output that MERCURY creates. For arrays that are distributed across processors, the corresponding bound will be the maximum value needed by any single processor, based on the computed decomposition.

There are also several new QS commands which can be manually added to the *qcks.in* file, prior to running parallel QS. The ones with a "CUSTOM" prefix can be specified in MERCURY; the PARALLEL command (if needed) must be added manually to the final *qcks.in* file.

```
CUSTOM SCREEN 10
CUSTOM EBJCHECK 20 1
CUSTOM LOADBALANCE 2.0 1.01
PARALLEL 1
```

[4] All new and modified QS and MERCURY commands are summarized in Appendix A

The "CUSTOM SCREEN N" command tells QS to write a few run statistics for the current timestep to the screen every $N >= 0$ timesteps. $N = 0$ means never write which is the default. These statistics include summations of the $\vec{E}$, $\vec{B}$, $\vec{J}$, and Q field components across the entire grid, and total particle counts for creating, pushing, and deleting particles. These values are useful in determining whether a parallel QS run is producing the same answer as a serial QS run.

The "CUSTOM EBJCHECK N M" command invokes a consistency check for $\vec{E}$, $\vec{B}$, $\vec{J}$ field components that lie on the shared surfaces between blocks. If $N > 0$ the check is performed only on $\vec{E}$ and $\vec{B}$ field components. If $N < 0$ then $\vec{J}$ field components are included in the check. The check is performed every $abs(N)$ timesteps; $N = 0$ means never perform it (the default). If the same component exists on the surface of two (or more) blocks, but the value is not bit-wise identical in both blocks, an error is flagged and diagnostic information is printed to the screen. If the 2nd parameter is $M = 0$, just a total count of errors if printed; if $M = 1$ then more detailed information is printed. This check is made across all block boundaries regardless of whether an adjacent block is owned by the same or a different processor. As discussed in more detail in Section 5.3.3, this error often gives rise to instabilities in a serial or parallel QS run, and should not occur if the code is functioning properly.

The "CUSTOM LOADBALANCE TOL1 TOL2" command controls how dynamic load-balancing is performed during a parallel QS run. The first parameter says to trigger a re-balance operation when imbalance is greater than $TOL1 \geq 1.0$, where perfect balance = 1.0. The 2nd parameter $TOL2 \geq 1.0$ controls what level of load-balance the operation attempts to achieve. Again perfect balance is a value of 1.0. Good settings to use in a typical QS simulation are $TOL1 = 1.5$ and $TOL2 = 1.1$. The effect and implementation of this command, including a precise definition of "imbalance", are explained in Section 6.

There are also two array limits modified by MERCURY when a CUSTOM LOADBALANCE command is encountered, and which can be further adjusted by the user. These are *wbsca1* = N and *wbsca3* = M. The first parameter extends the memory allocated for 1-d grid arrays by a factor of N; the second extends a few of the 3-d grid arrays by a factor of M. N and M can be expressed as integer or real factors, e.g. $N = 2.5$. As discussed in 6, the extra memory is used for new grid blocks created during the load-balance procedure. For typical problems where load-balancing is used, setting *wbsca1* = 3.0 (since it consumes little memory) and *wbsca3* = 2.0 is adequate; this is what MERCURY outputs by default. If QS runs out of memory when attempting to load-balance particles on a particular timestep, it will issue a warning which means these *pvlx* settings should be boosted by the user.

When parallel QS is run on a single processor, it normally executes as if it were the original serial QS code. For example, inter-block field connections are performed using the original QS routines. Optionally, the field connections (and other operations) can use the parallel algorithms described in the next section. This is invoked using the "PARALLEL 1" command in *qcks.in*. "PARALLEL 0" is the default which means to run in serial QS mode. Using the command on a single processor can be a useful debugging exercise to compare parallel QS output with original serial QS output. The command must be placed in *qcks.in* before or immediately after the BLOCK, GRID, and PERIODIC commands. When running on multiple processors, this command is ignored, since only the "PARALLEL 1" option makes sense.

Finally, the syntax of one QS command in *qcks.in* was changed for parallel QS. The PERIODIC command now has the syntax, "PERIODIC N LO HI". The first parameter N is the dimension I, J, or K. The LO/HI parameters are the coordinates in that dimension that are the periodic boundaries of the simulation domain. For example, "PERIODIC J 0.0 10.0" means the xz plane at $y = 0.0$ is conceptually the same as the plane at $y = 10.0$, and all grid cells in the simulation are assumed to lie between $y = 0.0$ to $y = 10.0$.

Note that all PERIODIC commands should be included in the same section of the *qcks.in* file where the BLOCK and GRID commands are listed. The old serial QS syntax for the PERIODIC command should no longer be used. The motivation for this change to PERIODIC is discussed Section 5.2.

4.2 Running Parallel QUICKSILVER

The only change needed when running parallel QS as compared to serial QS is when using POISSON boundary conditions. As discussed in the next section, parallel QS does not have the ability to generate Poisson solutions on-the-fly for block surface inlet conditions.

Instead the user should generate the *qsp2d.pff* file which contains the solution by running parallel QS on one processor on the original user-block description of the problem geometry for a timestep or two. This file is created automatically by parallel QS (when running on one processor) if it does not already exist. Parallel QS (on multiple processors) can then be run using *qsp2d.pff* as an additional input file.

4.3 Output

When parallel QS runs, it produces many of the same PFF and text files as serial QS. These files can be post-processed and visualized in the usual way. It also produces new files in a PDS (parallel data set) format [19] that can be converted into PFF files. This is discussed in more detail in Section 5.5.

At the end of a parallel QS run, various performance statistics will be printed to the screen. These include a breakdown of CPU timings for different portions of the timestep, particle counts, and load-balance information. The numbers include averages of various quantities across all processors, as well as histogramming by processor. For example, each processor keeps track of the CPU time it spends in particle pushing. For a run on P processors, the average, minimum, and maximum of this collection of P times is printed out, along with a histogram where the range (from minimum to maximum time) is divided into 10 bins and the time for each of the P processors is tallied into one of the bins. The histogram data can be useful in determining if load-imbalance occurred during the run.

4.4 Accuracy of Parallel Results

We now address the question of whether the user should expect parallel QS to produce the same answers (bit-wise identical) as serial QS. There are several questions to consider:

Is serial QS deterministic?
Is parallel QS deterministic?
Does serial QS give the same answer on different machines?
Does serial QS give the same answer no matter how many blocks are used?
Does parallel QS give the same answer no matter how many processors are used?

The answers are one "yes" and four "no's"., but since this isn't a quiz, we should explain further!

First, serial QS is deterministic. Running the same input files on the same machine will produce bit-wise identical answers (despite the use of random numbers, see Section 5.6). However, serial QS will not give identical answers when run on two different machines. This is due to round-off differences in computed quantities which can propagate onward to the next timestep, causing the two sets of results to diverge over time. In QS this effect may not only produce slightly different field values, but can alter whether a particular particle is created or destroyed. This will cause future timesteps to be fundamentally different. Clearly, such round-off problems are to be expected when running on two machines which treat floating-point operations differently. Less obviously, they can also occur if a problem is decomposed into blocks in two different ways. For example, if the geometry of Figure 1 were formulated as a 6-block simulation instead of 3-block, serial QS would sum $\vec{J}$ field values in different orders near block surfaces, which could produce round-off differences, and thus (eventually) lead to two different results.

Parallel QS suffers from these same limitations. Running the same physical problem on 100 versus 200 processors will typically be done with different numbers of blocks. Thus the simulation results will not agree precisely with each other or with a serial QS run. Even if the same number of blocks are used (running 2 blocks/proc on 100 procs versus 1 block/proc on 200 procs), there is an additional source of round-off differences when comparing two parallel QS runs. As will be discussed in Section 5.3, communication of $\vec{J}$ field values for grid locations shared between three (or more) blocks that reside on different processors is done asynchronously. Because the messages can arrive in random orders, the data are summed in different orders, and two runs on differing numbers of processors may not agree. This same effect can cause a repeat run on the same number of processors to disagree as well. Hence parallel QS is not deterministic, though in practice it may often turn out to be.

Notwithstanding these caveats, it is important to note that any two QS simulations of the same problem geometry should still produce answers that agree in a "statistical" sense, i.e. the two simulations should compute the same physical effects to within some statistical error bar, just as if a Monte Carlo simulation were run twice with a different initial random number seed. Parallel QS adheres to this looser standard; a parallel QS run should produce the same statistical answer as a serial QS run, independent of how many blocks and how many processors it is run on.

5 Implementation Details

In this section, we describe the changes and additions made to Version 3.0 of serial QUICKSILVER (QS) to create parallel QS. The modifications can be broken into several categories, based on what portion of the code they affect. We discuss each of these in turn: MERCURY (pre-processor), QS input and setup, QS fields, QS particles, and QS output. At the end of the section we discuss unsupported features in the current version of parallel QS.

For QS developers, this section (along with 6) serves as a detailed overview of the changes made for parallel QS. When references are made to specific QS routines, there are typically comments included in the code, prefixed by "c SJP", that will correspond to the overview given here. New files were also added to parallel QS; the majority are F77, C, and header files with the prefix *parallel_*. Most of the routines in those files are also discussed in this section. Much of the new code has additional useful documentation (variable definitions, routine overviews, etc) included in those files.

5.1 MERCURY

5.1.1 Decomposition Algorithms

As discussed in the previous section, a user can specify "CUSTOM DECOMPOSE" commands to tell MERCURY how to decompose each user block into sub-blocks. This operation takes place in MERCURY's *decompose* routine which performs two operations. It sub-divides user blocks into smaller blocks and it assigns the new blocks to individual processors.

The procedure for chopping a block into MX by MY by MZ pieces is straightforward. The procedure for chopping a block into an arbitrary number of sub-blocks, where each sub-block is roughly the same size and as cubic in shape as possible, is more involved. The latter goal is desirable to minimize the surface area of the sub-block, since the surface area represents field values that must be communicated to other processors. This procedure is invoked when a CUSTOM DECOMPOSE N M command is used or when no CUSTOM DECOMPOSE commands are specified. In the latter case, MERCURY performs the following heuristic, where P is the number of processors being decomposed for:

```
(1) Compute NTOT = the total number of grid cells in all blocks
(2) For each block $n$:
    (2a) FRAC(n) = cells in block $n$ / NTOT
    (2b) TARGET = FRAC(n) * $P$
    (2c) Chop block $n$ into TARGET sub-blocks (same as CUSTOM DECOMPOSE n TARGET)
```

The operation of step (2c) is performed in a recursive fashion. Consider the task of chopping a 10x13x15 block of cells into 5 (roughly) equal-sized pieces. First, the routine finds the longest dimension, in this case the z=15 direction. It then chooses to make an xy-planar cut (perpendicular to this dimension) at the z-location that comes the closest to leaving 2/5 of the grid cells on one side of the cut and 3/5 of the cells on the other side of the cut $(2 + 3 = 5)$. In this case the cut would create one block of size 10x13x6 and one of size 10x13x9. We have now broken the original problem into two self-similar new problems: chop a 10x13x6 block into 2 pieces, and a 10x13x9 block into 3 pieces. The routine recurses on this sequence of steps until each sub-block is a single piece. A 2-d example of a recursive decomposition is shown in Figure 5. Note that the nature of the algorithm often creates sub-blocks which border neighboring blocks in an irregular fashion.

Once all the user blocks have been sub-divided, the *decompose* routine assigns one or more sub-blocks to each processor. For load-balance purposes the goal is to give each processor as equal a number of grid cells as possible. This is done in one of 3 ways depending on the whether the optional "assign" argument in CUSTOM PROCESSORS is "sorted", "clumped", or "strided". Consider a list of M sub-blocks, each with a (possibly) different number of grid cells, to be assigned to P processors.

The "clumped" option assigns the 1st few sub-blocks in the list (a clump of blocks) to processor 0, the next few to processor 1, and so forth. How many sub-blocks are given to each processor depends on the block sizes; the clump size is adjusted so as to give each processor an equal number of grid cells. This option will tend to put sub-blocks that are geometrically close to each other on the same processor, since sub-blocks from the same original user block are grouped together in the list of M sub-blocks.

The "strided" option simply assigns every Pth block in the list to the same processor. For example, for 10 blocks assigned to 3 processors: processor 0 gets blocks 1,4,7,10; processor 1 gets blocks 2,5,8; processor 2 gets blocks 3,6,9. This method is a poor choice for load-balancing since it doesn't take into account block sizes, but is useful for debugging purposes.

The "sorted" option (the default) does the best job at load-balancing, but does not keep nearby blocks on the same processor. First the list of M sub-blocks is sorted by size, largest to smallest. The largest block is assigned to the processor with the least cells (initially all processors have 0 cells). Then the next largest block is assigned to whatever new processor has the least cells, and so forth until all blocks are assigned to processors. This is an implementation of the well-known bin-packing algorithm.

When the *decompose* routine finishes these two operations MERCURY prints a summary of the results to the screen. This data can be examined to see if a reasonable decomposition was created.

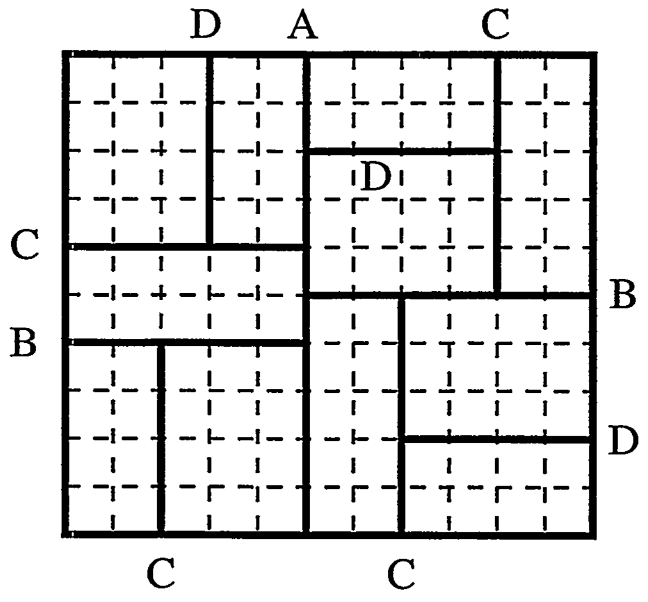


Figure 5: *A 2-d schematic of one grid block chopped into 11 sub-blocks by MERCURY's recursive decomposition option. The original grid is shown with dotted lines; the sub-block boundaries are solid lines. The letters represent cuts at various levels of the recursive algorithm. Initially a single cut A is made, then two B cuts, etc.*

5.1.2 Output

The *pvlx* file produced by MERCURY contains array bounds which every processor will use to allocate its local memory. These bounds are set to the maximum value any processor needs for the portion of the global problem (blocks, grids, boundary conditions, etc.) that it is assigned. If parallel QS generates a memory-overflow error when reading the *qcks.in* file due to insufficient *pvlx* settings, this is a bug, which should be reported to the QS developers.

The *qcks.in* file will contain several commands new to serial QS users, for example

```
UBLOCK 0.0 -8.0 -8.0 100.0 8.0 8.0
UGRID 1 I 0.0 100 1.0 0.0 0.0
PROCESSOR 3
BLOCK 53.0 -8.0 -8.0 69.0 8.0 0.0 1 54 1 1 70 17 9
```

The UBLOCK and UGRID commands list the block bounds and grid spacings for the original user blocks which were specified for the simulation using the standard BLOCK and GRID commands in serial QS. Every processor stores a copy of these settings which it will use to create its local grids. The PROCESSOR N command indicates that the next set of BLOCK commands are only relevant to processor N. This is in effect until the next PROCESSOR command is read. In other words, all processors except N ignore these commands. The BLOCK commands have additional appended arguments which specify the location of this (smaller) block within the original (larger) user block it was derived from.

5.1.3 Decomposition Strategies

QS users may be wondering how to best use the new MERCURY options so as to decompose a problem to run the fastest on a given number of processors. While the CUSTOM DECOMPOSE commands give considerable flexibility in this choice, a safe strategy is the following. The best decomposition for field updates is one block/processor, with all blocks being the same (roughly cubical) size. This balances the field computation, while minimizing inter-processor communication. If the CUSTOM DECOMPOSE command is not used, MERCURY will attempt to do this by default.

For some problem geometries, chopping into P blocks may not be a good choice. For example if there are more blocks than processors to begin with, or the blocks are of radically different size, then significant load-imbalance may result. In these cases, the CUSTOM DECOMPOSE command should be used with the goal of "over-decomposing" the problem into 2*P or 3*P blocks of as equal size as possible. Note that it is better to assign one (or a few) processors significantly less work (grid cells) than the average, than it is to assign one (or a few) processors significantly more. This is because during the field-update operation all processors will have to wait for the slowest one (most work) to finish.

When particle effects on load-balance are included, the question of a "best" decomposition strategy is more difficult. This issue is discussed further in Section 6 of the report.

5.2 Input and Setup

The majority of commands that parallel QS reads from the *qcks.in* input script are not changed from serial QS, either in syntax or meaning. To read this file in parallel, the *opread* routine in the QS *iopack* library was modified so that only processor 0 reads a line from the file, then broadcasts it to all the other processors.

Typically the first section of *qcks.in* contains global settings (e.g. timestep count and size). These set global variables in the code which every processor stores a copy of. This is a natural location for the user to add new parallel QS commands such as "CUSTOM SCREEN", "CUSTOM EBJCHECK", "CUSTOM LOADBALANCE", and "PARALLEL" which were discussed in Section 4.

5.2.1 Blocks and Grids

The next section of the file defines the problem geometry via BLOCK and GRID commands. In the previous section we described how new PROCESSOR commands are interspersed with the BLOCK commands to cause each one to be interpreted by only a single processor. During this process each block is assigned a unique global ID from 1 to *nblk_total* (the total number of blocks on all processors). The subset of blocks stored locally by a particular processor are numbered in the usual QS fashion from 0 to nblk by the processor itself. Each processor stores the global IDs for its blocks in an auxiliary *blocktag* array (see *parallel.inc*).

After BLOCK and GRID commands are read, QS calls its *mkgrd* routine to create the 1-d grid arrays that store the grid coordinates for each dimension of each block. Special care must be taken to insure that the end points (at the surface and ghost cells) of each blocks's 1-d arrays match up exactly with the opposite end points in adjoining blocks, including periodic blocks. This is to insure that the field-differencing equations which rely on grid spacings are consistent across block boundaries.

In serial QS end-point matching is relatively straightforward since the code knows about all blocks. In parallel QS it is more troublesome, since blocks are distributed across processors. Initially we built grids for each block, then communicated the grid arrays to other processors to match end point information. This can cause small inaccuracies when one user-specified grid region (linear or quadratic) is chopped into pieces for each of several sub-blocks. Instead, we now use the UBLOCK and UGRID commands to generate the entire set of global grids (still only 1-d arrays) in duplicate on each processor for the original user-block description of the problem geometry. End-point matching for the user blocks is done by each processor in the usual serial way using these global grid arrays. Then, each processor can extract the subset of values it needs for the 1-d grid arrays in its local blocks. The end points of these arrays will now be guaranteed to match since all processors extract identical values from copies of the same global arrays. All of this logic is encoded in the *mkgrd* routine and the *grdbuf* and *zoner* routines it calls.

5.2.2 Parallel Initialization

At the end of the *mkgrd* routine a new parallel function is called, *parallel_setup. Parallel_setup* initializes several variables which are stored in duplicate on every processor. These are documented in *parallel.inc.* They include *block2proc* which stores what processor owns every block in the simulation, and *global2local* which stores the local block index (from 0 to *nblk* on its owning processor) of every block in the simulation. The routine also initializes the new *bgcell* array. This is a 3-d integer array for all the cells (interior and ghost) of a processor's blocks. It stores the global ID number of the block that owns the cell. Thus for cells interior to a block it is set to the ID of the block itself. For ghost regions, the cell either corresponds to an external boundary or is an image of a real cell in the interior of another block. In the former case, *bgcell* for the ghost cell is set to 0; in the latter case it is set to the global ID of the other block. Additionally, the sign of the *bgcell* value is set negative if the ghost cell lies across a periodic boundary in any of the three dimensions (see discussion of periodicity below). Setting ghost-cell values in *bgcell* requires inter-processor communication. The details of this operation are discussed in 5.3.

5.2.3 Boundary Conditions

The next section of the *qcks.in* file typically defines various external boundary conditions (e.g. PEC, PMC, INLET, POISSON, OUTLET, BEAM_EMIT, CUSTOM TLINE) and internal material properties (e.g. CONDUCTOR, DIELECTRIC, FIELD_EMIT) for the QS simulation. PERIODIC boundaries are also allowed; though the syntax of this command and its placement in the *qcks.in* file have changed as discussed below.

After each command is read, the region (e.g. a 2-d surface) over which it is applied is checked against the block extents. In serial QS it is an error if the applied condition does not coincide with any block surface. This was changed in parallel QS to allow for the possibility that a particular processor's blocks will not have any overlap with a prescribed region. It is still an error if no processor's blocks have any overlap. This necessitated many (usually minor) changes in the error-checking logic for several commands in *dtread.F* and the lower-level routines it calls.

Poisson Inlets The setup code for a few boundary conditions had to be modified more extensively. One was the POISSON command. This option allows a 2-d solution to Poisson's equation for $\vec{E}$ fields to be applied at a surface as a time-dependent initiator of field flux entering one or more blocks. In serial QS, the user can either solve the 2-d Poisson's equation at start-up or read in a previously computed solution from a *qsp2d.pff* file. For parallel QS, we limit the choice to reading in a solution from a file.

The reason for this is that parallelizing the 2-d Poisson solve across a limited set of processors that own blocks adjoining the Poisson surface patch would be difficult. Since the solve itself is a one-time 2-d calculation and thus not costly, it made more sense to require the user to create the Poisson input file before running parallel QS. This is a portable PFF file which can be created via a serial QS run (one or more timesteps with Poisson output enabled) on a workstation using the original user-block geometry. Eventually, the QS developers will enhance MERCURY to produce the *qsp2d.pff* file if needed, so that the user of parallel QS will not be required to perform this extra run.

A *parallel_poisson* routine was added to parallel QS to enable reading of the PFF file and distribution of the solution data to multiple processors. A single processor reads the surface solutions, one user block at a time from the file, and broadcasts them to every processor. Each processor determines if any of its block surfaces overlap with the Poisson solution surface, and extracts the appropriate subset of the solution data.

Applied B-Fields Another input command that needed similar I/O modification was APPLIED-B READ. This QS option is used to define an external $\vec{B}$ field that is added to the average $\vec{B}$ fields used to push particles. A PFF file defining a (typically) 2-d azimuthally symmetric $\vec{B}$ field is read-in by QS and the $\vec{B}$ field is interpolated to all grid points in the simulation geometry.

For parallel QS, we modified *arzbrd.f* to read this file on a single processor and broadcast the data to all others. Each processor can then independently perform the interpolation to create field values appropriate for only its blocks and grid cells.

Transmission Lines Transmission line models are typically used in QS to model source/load impedance mismatches. For example, a series of transmission lines may be used to represent the pulsed power section (generators, pulse forming lines, impedance transitions, etc.) of an accelerator. One-dimensional transmission lines connect to the 3-d simulation geometry at a plane on the external boundary of a simulation. Serial QS has a restriction that a transmission line must only connect to a single block. Currently, this requirement still exists in parallel QS, where it is now more restrictive since an original user block is typically decomposed into numerous smaller sub-blocks. This means the user must insure that MERCURY performs its decomposition in such a way that the transmission line surface connection is not bisected by a new sub-block boundary. This should not be overly restrictive since transmission line cross-sections tend to be small, but it does require user attention.

The bulk of the coding changes for transmission lines in parallel QS occur in *tlinit.F* and *tline.inc*. User definitions of transmission lines and generators are stored on all processors. Processors with block(s) that contain transmission line ports build an index *tlmap* that maps from their local list to the global data structures. The *ebc* and *fldslv* routines that apply the transmission line model were modified to use this mapping.

Periodic Boundaries Another boundary condition that was modified for parallel QS was the PERIODIC command. Serial QS allows the specification of multiple periodic surface "patches" in any dimension, that effectively serve as conduits for particles and fields from some portion of the simulation geometry to another. This capability can be used (or mis-used!) to create simple or arbitrarily complex connections. For simplicity, we decided not to support this full generality in parallel QS. Rather we implemented the usual style of global periodic boundaries. These are specified in parallel QS using a new form of the command, "PERIODIC N LO HI" where n is a dimension index (I, J, or K), and LO/HI are the coordinates in that dimension that are connected. An additional change is that these PERIODIC specifications must now be included in the first section of the *qcks.in* file with the UBLOCK and UGRID commands

An example of this command's usage was given in Section 4. Parallel QS treats periodic boundaries as transparent to particle motion and as simply another kind of block connection for field updates (see the next section 5.3).

5.2.4 Output Commands

The final section of a typical *qcks.in* file contains HISTORY and SNAPSHOT commands for specifying simulation diagnostics. The syntax does not change for parallel QS, except for a few additional optional arguments. These arguments and the parallel implementation of the output commands are discussed in Section 5.5.

5.2.5 Final Setup

After the *qcks.in* file is processed, QS performs additional setup in *qsinit.F* before beginning the main timestep loop. In parallel QS, some new and modified tasks are performed:

(1) The *parallel_field_setup* routine is called to create the block connection "plans" used to communicate $\vec{E}$ and $\vec{B}$ fields between blocks on different processors (see Section 5.3 for a discussion of plans).

(2) If particles are to be used in the simulation, the *parallel_particle_setup* routine is called. Similar to (1), a separate plan is formed for communicating $\vec{J}$ and Q fields, as well as $\vec{J}$ fields that lie on block surfaces. Neighbor lists of which processors will be exchanging particles are also constructed – see Section 5.4. A new 3-d array, *bgijk*, is also constructed. This array stores in compact form (a single integer) the ijk indices of each cell in a processor's blocks. Similar to what was done with the *bgcell* array, inter-processor communication is performed using the *bgijk* array. This sets the ghost-cell values of the array to the ijk indices of the corresponding interior cell in another block. The *bgijk* array is used when a particle migrates to a new block on another processor to update the particle's ijk indices – see Section 5.4.

It is worth noting that taken together, the *bgcell* and *bgijk* arrays give a processor all the information it needs regarding how its blocks connect to all other blocks in the simulation. In serial QS this information was stored in the *bgcim* arrays (block ID number) and *bcm* arrays (offsets for how each block adjoined to every other). With a parallel simulation using potentially 1000s of blocks, the *bgcim* array could no longer store large-enough ID numbers. And as will be discussed in the next section, the all-to-all connectivity of the 2-d *bcm* arrays was not a scalable memory option for storing inter-block connectivity information.

(3) The 3-d *bgcim* arrays store various cell-wise flags which describe each cell's material and boundary properties. In serial QS, each block set CIM values for its interior cells, then did inter-block connections to initialize ghost-cell CIM values. In parallel QS, this would require communication. Instead, the CIM initialization code was modified so that each processor applies the various material and boundary condition commands to entire blocks, including ghost cells. Ghost-cell values are thus initialized explicitly without the need for inter-block information.

(4) When the *bgcim* array is initialized, each processor sets flags for each of its cells that are indices into tables of various properties (e.g. dielectric, conductor, boundary conditions). Since *bgcim* values will sometimes be exchanged between processors (see Section 6 on load-balancing), it is important that each processor construct its tables and CIM indices with identical orderings. Thus the code for processing the DIELECTRIC (and other) commands which affect the CIM indices was modified to insure identical results on all processors, even if a particular processor's blocks did not overlap with a given DIELECTRIC region. This occurs in *infinl* and related routines. Similar care is taken with the masks used to pack/unpack particle indices to insure that all processors construct identical masks; this occurs in the *mskdef* routine.

5.3 Fields

Field "connections" between adjoining blocks take place at two points during a QS timestep. The first is after $\vec{E}$ and $\vec{B}$ fields have been updated in the *fldslv* routine. The second is in the *jqdnsy* routine after $\vec{J}$ and Q fields have been created by scattering particle current and charge. In both cases, fields have been computed within individual blocks. Before the timestep can proceed field values at or near block boundaries must be exchanged between blocks. In the case of $\vec{E}$ and $\vec{B}$ fields, this exchange serves to update ghost-cell values. In the case of $\vec{J}$ and Q fields, the exchange sums values near block surfaces that have only been partially computed in individual blocks. Serial QS accomplishes these tasks via a series of surface- and edge-based operations (*blkcnn*, *fledge*, *jqsurf*, *jqedge* routines) using the pre-computed *bcm* arrays that store information on how every possible pair of blocks are connected. These routines and data structures were not designed for parallel operation or to be scalable to problems with 1000s of blocks, so we opted for a different approach in parallel QS.

In a generic sense, a one-way "connection" between two blocks involves the mapping of one set of field values in a "source" block to another set of values in a "destination" block. The connection itself may be thought of as "sending" the set from the source block, followed by "receiving" the set into the destination block. The destination block may sum the received values with pre-existing ones, or simply overwrite them. Regardless of which field quantity is being treated, the "overlap set" of mapped values is a sub-section of the 3-d array of field values in each block. Depending on the extent of the overlap between the two blocks, a sub-section may be a 2-d plane (or several planes) of values, a 1-d line, or even a single point, but can always be represented as a set of 3-d array indices for the block, i.e. field(ilo:ihi,jlo:jhi,klo:khi) in array syntax.

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 "send/receive" operation requires a message be sent by one processor and received by the other. Second, one routine can handle multiple kinds of field connections ($\vec{E}$, $\vec{B}$, $\vec{J}$, $\vec{Q}$, and others), so long as it knows the "mapping rules" appropriate to each kind of field.

In parallel QS, this block connection operation is implemented in two parts: (1) a setup routine that computes the overlaps for all pairs of blocks, and (2) 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 operation is called every timestep. The latter operation takes the place of the original serial block connection code in *fldslv* and *jqdnsy*. We describe both the setup and connection routines in some detail as they are a key kernel of parallel QS.

5.3.1 Setup

Grid connections are initialized using the *parallel_connect_create* routine. Called by each processor, its inputs include the list of blocks owned by that processor and the 1-d grid arrays for each of those blocks. The local starting address of each component of each 3-d field array is also passed in as is information about periodicities defined for the global simulation geometry. The function of this routine is to create a data structure that will represent all of the block overlaps a processor needs to exchange (send and receive) with other processors. The routine is written in C to enable easy creation of this fairly complex data structure (see the *cplan* struct in *parallel.h*). The routine returns a pointer to this data structure, called a "plan", to the F77 calling routine. The sequence of operations used to create a plan within *parallel_connect_create* is listed in Figure 6.

The first step is to acquire the 1-d grid arrays for all blocks from all processors; they will be used

(1) Acquire 1-d grid arrays for all blocks.
(2) Count overlap sets for sends from my blocks to all others.
(3) Allocate memory for storing the overlap info.
(4) Generate overlap info for sends from my blocks to all others.
(5) Count overlap sets for receives into my blocks from all others.
(6) Allocate memory for storing the overlap info.
(7) Generate overlap info for receives into my blocks from all others.
(8) Allocate memory for all send and receive messages.

Figure 6: *Stages of creating a "plan" for field connections between all blocks.*

numerous times in the remainder of the routine. Since they do not require much storage, a one-time global communication (MPI_Allgatherv calls in *parallel_grid_combine*) is done to acquire a local copy on each processor of the entire concatenated set of 1-d grid arrays.

The next step (2) is a double loop: for each of my blocks check for an overlap with every other block in the global simulation. This is a local computation which is performed by calling the *poverlap* routine with two sets of 1-d grids, one for my block as a "sender" and one for the other block as a "receiver". Because periodic boundary conditions can affect how two blocks overlap (or even enable a block to overlap with itself), the *poverlap* routine checks if the sending block adjoins a periodic boundary on any of its 6 faces. If it does, a periodic image of the 1-d grid array in a particular dimension is generated by adding or subtracting the appropriate periodic length to each grid coordinate. The *poverlap* routine then loops over all possible periodic images and calls the lower-level *overlap* routine with a particular instance of a send block image grid and the original receive block grid.

The *overlap* routine does a quick check to see if the corner points of the two blocks overlap in each dimension. If they do not (the predominant case), there can be no overlap and the routine exits. If this test is passed, then a detailed check for overlap ensues. This check is specific to the field component being communicated; we will list the overlap mapping rules for each field in a moment. In a geometric sense, the task of finding an overlap can be viewed as in Figure 7.

As discussed in Section 2, within a grid cell a particular field component exists at a point in 3-d space. In Figure 7, we consider the x-component of the $\vec{E}$ field. The "sending" block has E_x values defined at a 2-d array of points represented by circles (a 3-d array of points in 3-d QS). Note that E_x values are on half-grid spacings in x and full-grid spacings in y. As will be discussed below, the extent of the 2-d array for the sender includes only E_x values inside and on the surface of the block. The "receiving" block only needs E_x values that are a half-grid spacing outside the block, in its ghost cells whose extent is bordered by dashed lines in the figure. These E_x values are represented by the square points in the figure.

The overlap routine superimposes (see the *superpose* routine) these two sets of points (circles and squares). If the two blocks adjoin, then the two sets have a subset of points in common. This subset of overlapping points, shown as triangles in the figure, is the "overlap set", stored in a *set* struct defined in *parallel.h*. As indicated in the figure, depending on how the send and receive blocks adjoin, the overlap set can be a contiguous set of triangles (Overlap 1) or be disjoint (Overlap 2). In the former case, the overlap set can be represented by a set of 6 indices which bound the sub-array of triangular points. In the latter case, which

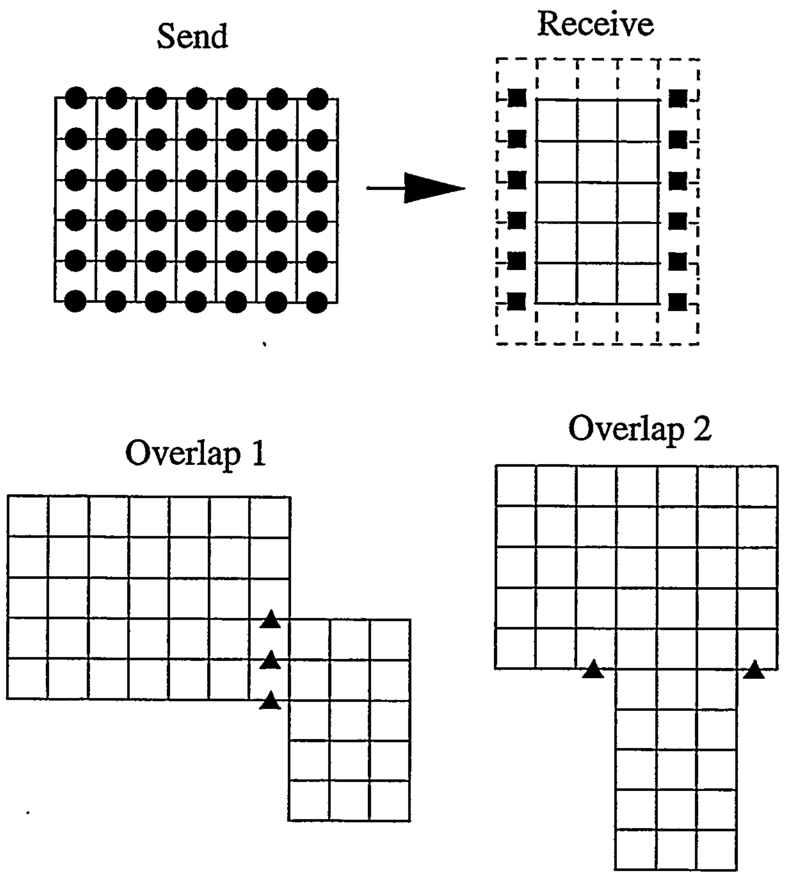


Figure 7: *A 2-d diagram of edge-centered E_x field components in two blocks. If the two blocks adjoin, then a few circular points in the "send" block overlap with some square points in the ghost cells of the "receive" block. The triangular points represent the overlap "set". Two kinds of possible overlap are shown. In the first, the overlap set is contiguous; in the second, it is disjoint.*

will be discussed below, the disjoint set is broken into multiple contiguous sets before being stored.

The low-level test for superposition of two points is handled carefully in parallel QS; the two points are declared "identical" if they are within a small distance "epsilon" of each other. This is because the computation of (periodically shifted) grid arrays for different blocks can result in small round-off differences in the grid values themselves. When the superposition test is performed, a conservative epsilon is computed as a function of the grid spacing (in each dimension) of the grid arrays for each block; see the *eps* routine in *parallel_connect_create.c.*

At the end of step (2) in Figure 6 the total number of overlap sets that this processor has with all blocks in the simulation has been tallied. In step (3) memory is allocated in the plan data structure for storing detailed information about these overlap sets. This information is organized so that the processor can send a single message to each partner processor. The message will contain all the overlap sets needed by the partner, which may result from several of the sending processor's blocks overlapping with several of the receiver's blocks.

Step (4) is similar to step (2) except that this time the results of the overlap tests are stored in the plan data structure in an array of *pplan* data structs (defined in *parallel.h*). This includes the extent of each 3-d overlap set, i.e., the bounds of the array of triangles in Figure 7. When the message is sent, the sending processor will first extract the field values for the overlap set from a particular field array and pack them into a contiguous message buffer. This packing operation is a triply nested loop that strides thru memory to extract the appropriate field values. The looping bounds and offset for this operation are what is stored in the plan as a *pplan* struct (see *parallel.h*), along with the starting address for where the overlap set begins in memory. The computation of this address uses the F77 field array addresses passed into *parallel_connect_create*.

Steps (5)-(7) are identical to (2)-(4) except that each processor now computes what information it will *receive* from all other processor's blocks. Since the send/receive operation is asymmetric as can be inferred from Figure 7, we compute the receive information explicitly from the receiver's perspective. Instead of a pack operation, now an unpacking of each received message will be performed to scatter the values from the message buffer into their appropriate field array locations. The loop bounds and offsets appropriate for this unpacking are again stored in a *pplan* struct along with a flag that indicates whether the unpacked values should be summed to their destination locations or overwrite them.

Finally, in step (8) the total volume of field data that will be sent and received as part of this block connection operation is now known. Each processor allocates a receive buffer, large enough to hold all its incoming messages. It also allocates a send buffer with enough memory to hold the largest message it will send to any other single processor. These buffers will be used in the actual message exchanges described in the next section 5.3.2.

To this point, all of the block-connection discussion has been independent of what field component is actually being exchanged. We now explain the specific overlap mapping rules that are applied to each kind of exchange. First, we state the goal that the exchange is designed to accomplish. We then specify the portions of the send block grid and receive block grid that are passed to the overlap routine to achieve this goal. This is typically done once for each field component (e.g. E_x, E_y, and E_z) since each component resides on different points in 3-d space. However, as shown in Figure 7, in some cases the overlap set is not a contiguous 3-d array of points (see Overlap 2). Representing the triangular points as a 3-d sub-array would include intermediate points which are not part of the overlap set. We solve this problem by breaking the receive grid into smaller pieces (e.g. each vertical column of square points in Figure 7) and calling the overlap routine multiple times for one field component exchange.

The *parallel_connect_create* routine is called with a "which" flag that specifies which field (or fields) is to be exchanged. All of the logic we will describe for overlap calculations with different types of field exchanges is encoded (with comments) inside the *overlap* routine in *parallel_connect_create.c*. We describe each option in turn from the perspective of an individual processor.

(1) Cell-wise quantities (*bgcell, bgijk, bgcim* arrays): The goal is to acquire only the ghost-cell values of my blocks. The send grid is all my interior cells. The receive grid is all my interior and ghost cells. Since no send grid contains a point that corresponds to any interior cell of a receive block, the *overlap* routine will only compute overlaps that include receiver ghost cells and I will thus receive only ghost-cell values.

(2) $\vec{E}$ fields (*bgei, bgej, bgek* arrays): The goal is to acquire only the ghost-cell values that are a half-grid spacing outside my blocks. The send grid is all points inside and on the surface of my blocks. The receive grid is only points a half-grid spacing outside my blocks, but NOT on the surface. For each $\vec{E}$ field component there are 2 possible connections between a send and receive block which requires two calls to the overlap routine. Connections where one or both of the blocks

is a PML (perfectly matched layer [1]) are handled as a special case, since this requires two field values be exchanged instead of one. This is done by generating two overlap sets (identical except for their starting addresses in memory) for a single grid overlap.

(3) $\vec{B}$ fields (*bgbi*, *bgbj*, *bgbk* arrays): The goal and send and receive grids are exactly the same as for $\vec{E}$ field connections. However, because $\vec{B}$ field components are face-centered quantities the receive grid must be broken into 4 contiguous sub-arrays and the *overlap* routine called 4 times for each component.

(4) $\vec{J}$ fields (*bgji*, *bgjj*, *bgjk* arrays): Because the movement of a particle can deposit current density a full grid spacing outside my block (see Section 5.4), the goal is to obtain fully-summed grid values up to a full-grid spacing inside my blocks and up to a half-grid spacing outside my blocks. The send grid is thus all points inside and on the surface of my blocks as well as ghost points a half-grid and full-grid outside my blocks. The receive grid is all points inside and on the surface of my blocks as well as ghost points a half-grid outside my blocks (but not a full-grid outside). Note that this overlap rule implies that a single point in one of my blocks may overlap with multiple images in other blocks. This is correct since it must receive values from each of those blocks to form a fully-summed $\vec{J}$ component for all particles that may have contributed to it. We note that our field-connection paradigm handles this potentially complex overlap logic straightforwardly even for arbitrary block connections.

(5) Q fields (*bgq* array): The goal is the same as for $\vec{J}$ fields except that Q fields are corner-centered quantities. Thus fully-summed grid values are only needed for points a full-grid spacing inside and on the surface of my blocks. The corresponding send grid is all points on the inside, surface, and full-grid spacing outside my blocks. The receive grid is all points inside and on the surface of my blocks. The charge density in QS is stored by charge groups for different species of particles. This is essentially a 4th storage dimension over charge groups in the *bgq* array. This is treated similarly to PML blocks for $\vec{E}$ fields; one overlap set is created for each charge group, each with a different starting address in memory.

(6) Average $\vec{E}$ and $\vec{B}$ fields (*bgeai*, *bgeaj*, *bgeak*, *bgbai*, *bgbaj*, *bgbak* arrays): Spatially-averaged $\vec{E}$ and $\vec{B}$ fields are needed by QS to perform sub-cycling of particle motion within a single timestep. These are corner-centered field quantities that are computed from the edge- and face-centered $\vec{E}$ and $\vec{B}$ field components within each block. The connection goal is to acquire only ghost cell values a full-grid spacing outside my blocks. The send grid is all points inside and on the surface of my blocks. The receive grid is only points a full-grid spacing outside my blocks, but NOT on the surface. This means the receive grid must be broken into 6 contiguous pieces (planes of ghost cell values) and the *overlap* routine is thus called 6 times.

For options (1), (2), (3), and (6), the received values overwrite existing ghost cell values (except for option (2) with PML blocks where summing of received values can also take place). For the $\vec{J}$ and Q fields of options (4) and (5), the received values are summed to existing field values in the receiving block.

Finally, the $\vec{E}$ and $\vec{B}$ fields are communicated at the same time in a parallel QS timestep (in the *fldslv* routine). Thus there is one plan, created at setup time, which stores both $\vec{E}$ and $\vec{B}$ field inter-block connection information. Similarly, there is one plan created for both $\vec{J}$ and Q field connections which is used within the *jqdnsy* routine.

5.3.2 Connection

The actual communication of field values is straightforward once block connections have been pre-computed and stored in a plan. The operation is performed by the *parallel_connect* routine. Given parallel QS's block-decomposition strategies and arbitrary assignment of blocks to processors, these connections can (in general) require any processor to send/receive a message to/from any other processor. On a distributed memory parallel machine, such message passing is most efficiently done in an asynchronous fashion. The plan data structure is designed so as to enable this irregular pattern of message passing to be performed as quickly as possible each timestep.

First, each processor posts receives (MPI_Irecv) for all the messages it expects to receive. This is to avoid unnecessary message copying by the underlying MPI library. The processor then sends all of its outgoing messages. For each processor it is sending to, a message buffer is packed using the overlap information stored in the plan, and the message is sent (MPI_Send). The processor then performs all block connections for cases where it owns both the "sending" and "receiving" block. The overlap sets for these connections are stored in the plan identically to how they are stored for off-processor connections, but in-memory copies can be performed rather than message sending/receiving. These strided memory copies are accomplished by first packing into a buffer, then unpacking from the buffer, the same as if a message were actually sent.

This strategy of treating on-processor connections the same as off-processor connections has two advantages. First, it means that parallel QS will run as-is on a single processor. All block connections will simply be handled by this on-processor portion of the *parallel_connect* routine. Second, on a parallel machine, a processor can potentially do useful work while waiting for incoming messages to arrive.

Once the on-processor connections (if any) have been completed, each processor waits for incoming field data. The MPI_Waitany routine will return when any of its incoming messages have arrived, at which point the processor immediately unpacks the field data in that message. When all messages have been received and unpacked, the block connection operation is complete.

5.3.3 Block Surface Instabilities

Implicit in the asynchronous nature of the block connection algorithm described in the previous section, is the fact that incoming messages from other processors may arrive in random order. The order may be different from one machine to the next, when a simulation is repeated on the same machine, or even from one timestep to the next. In the case of $\vec{J}$ field communication, this means that field values may be summed in different orders, producing slight differences (round-off in the last digit) in the results. Normally this is not an issue; it only produces small statistical differences between two QS runs, as discussed in Section 4. However, in the case of $\vec{J}$ field values that lie on the surface between two blocks it can cause a subtle instability.

$\vec{J}$ fields are used in the next timestep by each block to update new $\vec{E}$ fields. If the same surface $\vec{J}$ field value is different in two blocks, then so will the corresponding $\vec{E}$ field value be different. Because $\vec{E}$ field values at block surfaces are never shared between blocks, this small difference can propagate to the next timestep. The signature of this instability is that over the course of 100s or 1000s of timesteps, there is a growing difference in the same surface $\vec{E}$ and $\vec{B}$ component as stored by two blocks that share the common surface.

Our solution to this problem was to force the $\vec{J}$ values at block surfaces to be bit-wise identical across all blocks that own images of the same physical edge-centered point. Since the normal $\vec{J}$ and Q field communication involves significantly more data than just surface $\vec{J}$ field values, we implemented this with a second communication step, called from *jqdnsy*. As with the other block connections we first setup a plan via a call to *parallel_connect_create*. The overlap mapping rule for this style of block connection is a new seventh option, different from the six rules listed in the previous section:

(7) Block surface $\vec{J}$ fields (*bgji, bgjj, bgjk* arrays): The goal is to overwrite only the surface values

of my blocks. The send grid is all points inside and on the surface of my blocks. The receive grid is also all points inside and on the surface of my blocks. Since no send grid contains a point inside a receive block, the overlap routine will only compute overlaps that involve my surface values.

These overlap sets are stored in the plan in the usual way with one significant change. If a particular $\vec{J}$ field value has images on the surface of several blocks, we want exactly one of those image values to overwrite all the others. Consider all the blocks in the simulation to be numbered globally from 1 to N. If a surface point has images in several blocks, we want the value in the lowest-numbered block to overwrite the value in all higher-numbered blocks. We force this to occur by only storing an overlap set in the plan if it involves a send from a lower-numbered block to a higher-numbered block. In other words, half of the overlap sets generated by rule (7) above are discarded.

This modified plan will insure a processor only overwrites its surface $\vec{J}$ field components with values from lower-numbered blocks. However, high-numbered blocks may still receive two (or more) of such values, so the value from the lowest-numbered block must be used in the last overwrite. This is enforced by changing the MPI_Waitany call used in *parallel_connect* to an MPI_Wait for a specific message. The loop over expected messages is ordered so that values from the lowest-numbered block overwrite all previously overwritten values. This altered logic is encoded in a separate *parallel_connect_ordered* routine which is the one called from *jqdnsy* for the $\vec{J}$ field surface communication.

It is worth noting that this instability is not a parallel issue, but a result of the new block-connection algorithm itself. Once we knew what to look for, we were able to trigger the instability when running on one processor. This is because the round-off differences in $\vec{J}$ can also occur at the surface of two blocks owned by the same processor, since the same surface field value is stored by both blocks and the two values are computed by summing contributions in different orders. The original serial QS code did not suffer from this instability (to our knowledge) because it used a different (non-parallelizable) block connection scheme. The old algorithm which mapped one block's faces/edges to another block's, actually copied more field data than was needed to satisfy the mapping rules outlined in the previous section. An unexpected benefit of the over-copying was that block surface $\vec{E}$ and $\vec{B}$ values were overwritten, forcing them to be the same on both blocks.

In the process of finding and fixing this instability problem, we added an option to the block connection routines that will flag an error whenever field components on shared block surfaces are not bit-wise identical. This test is invoked by specifying a CUSTOM EBJCHECK command in *qcks.in*; see Section 4 for instructions on using this command. Though it involves extra communication it is a useful check to perform whenever a user suspects that parallel QS may be producing incorrect answers.

It is implemented as (yet) another option for a new style of block connection. $\vec{E}$ and $\vec{B}$ fields on the shared surfaces between blocks are communicated and tested for equality. If desired (see Section 4), $\vec{J}$ fields can also be included in the test. The setup of such a communication plan requires a new eighth overlap rule in *parallel_connect_create*:

(8) Block surface $\vec{E}$, $\vec{B}$, and $\vec{J}$ fields (*bgei, bgej, bgek, bgbi, bgbj, bgbk, bgji, bgjj, bgjk* arrays): The goal is to communicate only the surface values of my blocks. The send grid is all points inside and on the surface of my blocks. The receive grid is also all points inside and on the surface of my blocks. As in rule (7), since no send grid contains a point inside a receive block, the overlap routine will only compute overlaps that include surface values.

The check itself is performed from *qcks.F* via a call to *parallel_connect*. When messages are unpacked, the received field values do not overwrite existing ones, rather they are compared to existing field values and an error count is incremented (stored inside the plan) if they are not bit-wise identical. A subsequent call from *qcks.F* to *parallel_query_errors* will print diagnostic information to the screen if any such errors were found by any processors.

5.4 Particles

5.4.1 Serial vs Parallel

Before highlighting the changes/additions made for parallel QS, we first explain how particles are pushed in serial QS. As discussed at the end of Section 2, each particle stores 4 indices corresponding to its location within cell ijk in block m in addition to its position, momentum, and charge. The timestep size in QS is bounded so that a particle can move at most one grid cell in a single timestep. After the move is computed, the particle's initial and final coordinates and initial and final ijk indices are known. If the particle ends up in a cell inside the same block this stage of the push is done (though it could have been killed by entering an internal conductor cell). If it ends up in a ghost cell of block m, then the status of the ghost cell is checked in the *bgcim* array. There are several possibilities. Either the particle is killed (e.g. it is a conductor, outlet, inlet), or put back into the block (e.g. reflection boundary), or it passes into another block. In the latter case, the particle's ijk and block m indices are immediately updated to reflect its new block location.

Thus in serial QS there are three possible outcomes of a particle move: the particle stays in the same block, moves to a new block, or is killed. The final stage of the particle push is to scatter its charge Q and current density $\vec{J}$ to the appropriate field arrays. Since each particle already has valid final ijk and m indices, the field arrays associated with the particle's final block are the ones that are updated.

For parallel QS, we alter this sequence of steps, so that charge and current density are scattered to the field arrays associated with the particle's initial block. The reason is that there is now a 4th possible outcome of a particle move: the particle moves to a new block owned by another processor. We call this particle "migration". Migrating particles have to be communicated to their new processors every timestep as part of the particle push. If we waited to scatter Q and $\vec{J}$ until after a particle arrived at its final block, two complications arise. First, we would need to send extra information with the particle regarding its initial location at the beginning of the move. This is so $\vec{J}$ could be scattered along the particle's entire path. Second, the code for performing the scatter would have to be invoked twice, once in the usual way for particles staying on a processor, and once after new particles arrived from other processors.

5.4.2 Modifications for Parallel QUICKSILVER

With this change in mind, these are the modifications made to particle handling for parallel QS:

(1) In the QS setup phase, several variables and lists are pre-computed by each processor for later use in particle migration. The variables themselves are listed in *parallel.inc*; they are initialized in *parallel_particle_setup.F* by a call to *neighbor_init*. The most important of these are *nneighsend* and *nneighrecv*. These are the number of "neighbor" processors that this processor will potentially send particles to and receive particles from. A neighbor processor is one who owns one or more grid cells adjacent to a processor's blocks, i.e. that overlap with any of the processor's ghost cells. A processor can quickly generate its list of neighbor processors by looping over all its ghost cells and checking their *bgcell* values. The utility of the neighbor list is that it identifies the only other processors that a processor need communicate with when migrating particles.

(2) New migrate caches (see *caches.INC*) were added in addition to the survive caches already used. Recall that "caches" are the QS data structure used to store lists of particles while allowing their number

to grow and shrink from timestep to timestep. The new caches store particles tagged for migration during the push; the particles will later be sent to other processors.

(3) A new *parallel_partbc.f* routine was added (to replace serial *partbc.f*), which is called after particles have moved to their new positions, to check the outcome of the move. The new routine determines which of four possible outcomes has occurred and places the particles in appropriate caches. The outcome options for each particle are (a) killed, (b) stay in same block, (c) move to another block owned by this processor, or (d) migrate to a new block on another processor. *Parallel_partbc* uses the *bgcell* value for the particle's final cell to determine which outcome has occurred. Recall that *bgcell* was initialized with the global block number that is the owner of each of a block's interior and ghost cells.

(4) Scattering of particle charge Q and current density $\vec{J}$ is done in each particle's initial block, even if it moves to another block on the same or a different processor. This operation occurs in the *jandro* and *jandr1* routines. To enable initial-block scattering, the updating of the block index m and grid indices ijk for each particle to reflect its final position is delayed until after the calls to these routines. The change from final to initial block scattering also means particle charge now accumulates in ghost cells of the Q field arrays. This effect is accounted for in the inter-block field connection rules for Q described in Section 5.3.

(5) The *vechdr* routine was modified to pack and otherwise manipulate the new migrate caches. For particles that have moved to a new block owned by the same processor, *vechdr* also now updates the particles ijk and block m indices appropriately for the new block. (Recall that this operation was removed from *partbc* so that particle charge and current could be scattered based on the particle's initial block.)

(6) At the end of the particle push, a new *parallel_migrate* routine is called (from *parhdr.F*). Its purpose is to have each processor send old particles that have left its blocks and receive new particles whose final coordinates now reside inside its blocks. The first task is for each processor to count how many particles need to be sent to each of its neighbors. A linear pointer (integer) list is allocated and the migrate cache is scanned. For each particle, the new processor which will be sent that particle is determined and a counter (*neighsendcount*) for that processor is incremented. The ijk and block m indices for the particle are set to new values appropriate for the receiving processor using information in the *bgijk* array. As the scan proceeds, the linear pointer list is used to store the sub-list of particles that will be sent to each neighbor processor.

The next step of *parallel_migrate* is for each processor to tell its neighbors how many particles to expect. Each processor sends and receives these counts, even if the count is zero. It can then check its memory to insure it has sufficient space for the incoming particles and allocate the needed message buffers.

Similar to the communication algorithm for field connections between blocks, each processor now posts a receive (MPI_Irecv) for each incoming message it expects. It then packs up and sends (MPI_Send) a list of particles to each of its neighbors. The sub-list mentioned above is used to efficiently extract the appropriate particles from the migrate cache during this operation. The processor then waits for its incoming messages to arrive and adds the received particles to its caches.

We note that the asynchronous nature of this communication procedure allows for particle exchanges between any pair of neighboring processors. Typically for a QS run on large numbers of processors, each processor will have only a few neighbors. Due to the irregular nature of MERCURY's block decompositions and assignment of blocks to processors, these neighbors could be any other random processor. This scheme allows each processor to communicate with only its neighbors. It is also efficient since, other than the handshaking exchange of particle counts, all particles migrating between a pair of processors are sent/received in a single message.

(7) The *parhdr* routine was modified to log new statistics on particle movement and migration, both for individual processors and across processors. The *parcnt.INC* file contains the new data structures. Some of

these statistics can be printed to the screen via the CUSTOM SCREEN command and are also summarized in the run's final output. They can be useful for debugging purposes as well as analyzing the performance and load-balance characteristics of a simulation.

(8) Finally, we note that QS has several commands that govern how particles are created including BEAM_EMIT, FIELD_EMIT, and CUSTOM PRELOAD . These options work the same in parallel QS as they did in serial QS, with each processor creating the appropriate particles in its own blocks each timestep. For beam emission this required correct computation of weighting factors for emission regions spread across multiple processors; this takes place in *creini.f* and *crebmi.f.* For field emission, there is a subtle difference between serial and parallel QS, which is discussed in Section 5.6.

5.5 Output

Simulation output from QS is specified via "HISTORY", "SNAPSHOT", "CUSTOM KPSAVE", and "CUSTOM KPWRITE" commands. There are no syntax changes in these commands, except for an additional optional argument for the ones that control particle snapshots: SNAPSHOT PARTICLE, SNAPSHOT MAX_PARTICLE, and CUSTOM KPWRITE. In serial QS these commands take an (optional) argument for the maximum number of particles to include in a single snapshot. In parallel QS this argument refers to a global count of particles across all processors. An additional local maximum can be specified which will be used to limit the number of particles any single processor will output.

The following sections outline how the operation of the output routines has been modified in some cases for parallel QS and describe a new output file format used for snapshot quantities.

5.5.1 HISTORY Commands

HISTORY commands are used for integrating field strength or energy over lines/planes/volumes and for summing particle and performance statistics, all as a function of time in the simulation. They typically create a relatively small volume of output, which in parallel QS is still written into a file in the native QS PFF format.

When the *dtread* routine reads a HISTORY FIELD command from *qcks.in*, its geometric extent is checked against the simulation geometry for possible errors. In parallel QS we must now consider cases where none of a processor's blocks overlap with the HISTORY geometry or where two or more processors share an overlap. In the former case, we simply allow this to not be an error, unless no processors have any overlap. In the latter case, special care must be taken to insure each processor computes a valid fraction of the HISTORY quantity. This is so that when the contributions are summed across processors, an accurate total will result. Consider the case where a line integral is to be performed ($\vec{E} \cdot d\vec{l}$) along a series of geometry segments defined in a HISTORY FIELD command. In serial QS, this would be computed by stepping along the line segments one grid cell at a time, assigning the segment to one unique block when it bordered two or more blocks. In parallel QS, portions of these segments may run along the surfaces or edges of blocks owned by different processors. The key question is how to coordinate all processor's efforts to correctly compute the contribution of each cell to the overall integral.

Our solution to this problem was to insure that only one processor computes each cell-wise contribution to the HISTORY quantity. First, in a pre-processing operation, the *addseg* routine was modified to break up each HISTORY segment into sub-segments, where each sub-segment has the same set of "neighbors". In this context, a set of neighbors are the blocks that own grid cells that border the line segment. Each one-cell-length portion of the line segment will have 4 grid cells that border it (or less if it lies along a global external boundary).

The break-up operation is performed by the *line2many* routine in *parallel_seg.f.* It first checks the global block owners (via the *bgcell* array) of the 4 cells that surround the initial portion of the line segment. It then "walks" along the segment, one cell-length at a time, checking each set of 4 neighbors. Whenever the set-of-4 changes, the segment is truncated and a new segment is begun. The routine returns a new list of shorter segments that comprise the original segment. Conceptually, each processor now has an expanded list of segments that describe the overlap of its blocks with the global HISTORY geometry. The *addseg* routine then decides for each new segment, whether this processor will compute it or not. The decision is made by the *seg_decide* routine (also in *parallel_seg.f*) which checks the 4 neighbor blocks and masks out all but the lowest-numbered one. Processors which own the other 3 blocks discard this segment from their HISTORY list. Thus the contribution from each HISTORY segment will be computed by exactly one processor.

The same logic is applied to HISTORY FIELD commands for planar geometries (e.g. surface integrations). In this case, the *plane2many* routine chops a planar patch in two dimensions into sub-patches where every cell face in the sub-patch has the same two neighbor blocks on either side. Exactly one processor keeps each sub-patch in its HISTORY list. HISTORY FIELD commands for volume geometries (e.g. energy summation) do not require any special pre-processing. This is because the volume naturally breaks up into sub-volumes within blocks that are already uniquely owned by a particular processor.

The above description implies that a processor may only contribute to a subset of the total number of HISTORY outputs. The mapping of local histories to global ones is done with new variables defined in *chist.inc.* At each timestep, a processor computes the values for its local histories. In the *psthis.F* routine these contributions are summed across processors (via MPI_Allreduce) and written by processor 0 into the *qshis.pff* file. There is special logic in this routine to insure that every history quantity is accurately computed when summed across all processors. For some quantities this means that only one processor should contribute to the sum since the quantity is already stored in duplicate on multiple processors.

One final note about HISTORY command output is in order. At the end of a run, QS performs one final data manipulation (essentially a transposition of the 2-d data set) on all history quantities before writing out the *qshis.pff* file in its final form. The field arrays are used as temporary memory to hold all the history quantities. In parallel QS this operation is done by a single processor. If a small-geometry problem with many histories is run on a large number of processors, it is possible that the field arrays on a single processor will be too small to perform the data manipulation. Parallel QS will issue an error message if that occurs.

5.5.2 SNAPSHOT and KPWRITE/KPSAVE Commands

Particle and field snapshots generated by SNAPSHOT and CUSTOM KPWRITE/KPSAVE commands can produce large volumes of output. In parallel this would be a problem if we throttled all output through a single processor. Instead, these commands take advantage of the PDS (parallel data set) I/O library recently developed at Sandia [19, 18]. Similar to the QS PFF library, PDS creates portable binary files that can be moved transparently from one machine to another. The files can contain multiple kinds of scalar, array, and time-dependent data. On machines with parallel disk systems (e.g. the Intel Tflops), PDS performs two important tasks that maximize I/O throughput. First, it multiplexes a parallel code's reads and writes (from every compute processor) through the system's multiple I/O nodes. Second, it buffers small I/O operations so that the actual file reads and writes are done in large blocks.

Both of these tasks are performed invisibly by the library, so there is no special coding needed in the application code to make I/O work in parallel. However, for parallel QS, we did have to make numerous modifications to output routines such as *pstsnp* that now call the PDS library in order to conform to the library's API.

Similar to the HISTORY commands, the SNAPSHOT and KPWRITE/KPSAVE geometries specified in *qcks.in* are intersected with each processor's blocks during parallel QS's setup phase. On a timestep when a snapshot is computed, each processor extracts the relevant information from its blocks and does a PDS "write" simultaneously with all other processors. The PDS library routines aggregate this information into one parallel output file. For field snapshots, this file also contains mappings from each processor's local blocks to the original user blocks. This map information is computed and written into the PDS file in the *pstsnp* routine. This enables post-processing tools such as *pds2pff* to re-map the parallel field values back to the original user-block geometry. As its name implies, *pds2pff* is used to convert PDS files into QS PFF files suitable for further analysis. The syntax for use of this command is listed in the Appendix.

Finally, we note that when running parallel QS on one processor, the user can choose whether to use the old snapshot file format (PFF) or the new default format (PDS). This is specified using the "CUSTOM USE_PDS" command described in the Appendix.

5.6 Unsupported Features

There are a few capabilities of serial QS that have not yet been fully implemented in parallel QS. We discuss each in turn.

(1) Random numbers (RNs) are used within QS in several ways. First, they can be used to create a particle at a randomized location within a grid cell. A new particle is then assigned a random number which it uses throughout its simulation lifetime. This particle RN is used to select particles for diagnostic output. RN's are also used when a particle moves to a new cell. Technically, it should deposit current density in all the cells it crosses. This can involve an expensive trajectory computation if the particle moves diagonally, cutting across the corners of one or more cells. To avoid this, QS uses a RN to choose a random path that deposits current density along a subset of the possible cell crossings. Averaged over many particles, the results should be statistically the same as if exact trajectories were computed.

The problem for parallel QS is how to have each processor compute RNs independently. This could be done rigorously using parallel RN generation techniques, but was deemed not important enough to pursue. Instead we let every processor use the same RN generator and initial seed. Since each processor uses its stream of RNs in a different way for different particles, this should not affect the statistical quality of the results. We note that the use of RNs in this way is an additional source of statistical discrepancy between parallel QS runs on different numbers of processors, as discussed in Section 4.4. This would still be the case even if parallel RN generators were used. To avoid this problem (e.g. for debugging purposes) the RNs used in parallel QS can be set to a specific value (e.g., 0.5) using the "CUSTOM FIXEDRANF value" command. This effectively turns off RNs for everything except the selection of diagnostic particle output.

(2) Restart files are used in serial QS to checkpoint a lengthy calculation so it can be resumed in a new run from precisely the point in simulation time that the restart file was written. The restart file contains all the particle and field data, as well as all other state information needed to restart the simulation. This file write is performed by the *rstart* routine. *Rstart.F* is the single largest file (over 4000 lines) in QS because of all the detailed output that must be done. In principle, parallel QS could write a similar file from multiple processors, using the PDS library described in the previous section, but this is not yet fully implemented. Instead, parallel restarts are done by having each processor write a separate file in the original serial QS format. This is not as scalable an operation as the PDS method would be, nor does it allow parallel QS to be restarted on any number of processors.

(3) Field emission of particles from conductor surfaces can occur slightly differently in a serial versus

parallel QS run. Normally, emission occurs when the electric field normal to an emission surface exceeds a breakdown threshold. The breakdown event causes a CIM value in *bgcim* to be set to insure the normal electric field is zeroed. In parallel QS, if that grid cell and emission surface is at a block boundary between two processors, the CIM setting is not communicated to the adjacent processor (see code comment in *crefeq.f*). This means that particles in the other processor's adjacent grid cell will experience a slightly different force from the electric field than they should. However, unless there is a corner in the emission surface right at the block boundary, this condition will self-correct within a few timesteps. In any case, such differences have a negligible effect on the simulation dynamics. Thus we have not written specialized communication routines to adjust the CIM setting on the adjacent processor.

6 Load Balancing

As discussed in Section 3, load imbalance often occurs in parallel PIC simulations. This is because the field update and particle push are separate expensive computations which are difficult to independently spread uniformly across all processors. There are also computations needed to gather/scatter information between the field grids and the particle positions. This imposes the additional constraint that a processor should own grid cells and particles in the same geometric region.

In parallel QS, as was discussed in Section 4, it is straightforward (via MERCURY) to decompose the field grids evenly across processors. However, because particle densities in the simulation domain can vary by orders of magnitude both spatially and temporally, this by itself can lead to huge particle imbalances for some problems. Conversely, we could attempt to bias the MERCURY decomposition so processors in regions of high particle density own less grid cells. Even if the density variations were static in time (which they aren't for most problems) and this scheme was able to balance the particle load perfectly, we could still have significant imbalance in the field update computation. Although particles often account for 90% or more of the total computational time in QS, the field imbalance would have a dramatic effect on the overall scalability of parallel QS running on 100s or 1000s of processors.

6.1 Other Ideas

As we thought about this problem for parallel QS, several strategies were considered. One idea for large problems was to *statically* over-decompose (via MERCURY) the grid into more blocks than processors. Thus each processor would be assigned several blocks. The block-to-processor assignment could be made randomly so that the blocks on one processor would be scattered throughout the simulation domain. Then even if there were large particle density variations, one processor should own some blocks with few particles and others with many. This scattered over-decomposition would hopefully produce a rough overall load-balancing of the particle load.

This option requires no additional coding in QS, as it can be experimented with by proper MERCURY usage. The drawbacks are that the inter-block communication cost for fields is increased due to using more and smaller blocks, i.e. increasing the surface-to-volume ratio of individual blocks. Another drawback is that there is no guarantee the random assignment of blocks to processors will not result in some processors doing considerably more work for a particular simulation.

Another option that was considered was to *dynamically* load balance by moving entire grid blocks with their interior particles to other processors. The same kind of initial over-decomposition would be used as described above. At some point during the simulation, if load-imbalance was detected, a heavily loaded processor would send one or more of its blocks to lightly loaded processors. As the particle densities varied in time, blocks could shift back and forth between processors to keep the load balanced. This is similar to the idea that was proposed in [5] for their parallel PIC code, though to our knowledge it was never actually implemented.

One disadvantage of this approach is that it would be difficult to implement in QS, without considerable reworking of the code and data structures. There is no single data structure containing a block and its particles that could be bundled up and sent to another processor. The boundary conditions and diagnostic (output) requests apply to multiple blocks and are stored in separate lists which would have to be recomputed and restructured if blocks moved between processors. Also, all the arrays which hold 1-d grid and 3-d cell values are densely packed block-by-block, due to the F77 usage of allocated memory. Deleting and inserting new blocks in this packed structure would require considerable data copying. An additional drawback of this approach is that the load-balancing is coarse-grained. Unless the field grids are chopped into hundreds of blocks/processor (which would be very costly for inter-block communication), the smallest "unit" of work which can be passed to another processor is an entire block of grid cells and particles. Thus in practice it might not be effective at insuring load balance.

Another idea that was considered was to dynamically adjust block sizes. For example, if a block with too many particles adjoined one with few particles, the boundary between them could be shifted to make the first block smaller and the second block larger. This is similar to the approach used in [20] where density variations in particles trigger a complete repartitioning of the (single block) grid into new varying-sized sub-blocks. In QS it would be difficult to completely re-partition a general (many block) geometry on-the-fly into a new set of balanced sub-blocks. It would incur the same data structure and code complexity issues discussed above. Even incrementally adjusting a few block boundaries so as to "grow" underworked blocks at the expense of overworked ones, would be a coding challenge. If MERCURY's recursive option were used to partition a block, then irregularly joined sub-blocks result. It would thus be difficult to grow some blocks while shrinking others. If MERCURY were used to chop a block into a regular 3-d array of sub-blocks, then block growth would require entire planes (boundaries for several blocks) to shift so that an entire row of blocks grew at the expense of the next row. This many-block effect could adversely impact the hoped-for load-balancing benefits.

6.2 Our Solution

The approach we finally decided to implement was based on an idea originally proposed by Gary Montry, a contractor working with the Sandia QS group. Under the LDRD, Gary experimented with various parallelization strategies in a version of parallel QS he and Mike Pasik developed. This version was restricted to one initial user block. The new load-balancing strategy worked well in their code, so we adopted and modified the ideas for use in the full multi-block parallel QS described in this report. So far as we know, the idea is a novel one in parallel PIC code development; a journal article emphasizing the load-balancing aspects of this work has been submitted for publication [12].

Gary's idea was to continue to use a *static* (balanced) decomposition of the field grids, but to dynamically migrate particles from overworked processors to underworked ones via "windows" within a processor's blocks. A "window" is a contiguous sub-region of grid cells within a heavily-loaded block that is mapped to a new block on a lightly-loaded processor, as illustrated in Figure 8.

Particles within the window migrate from the "parent" block (on the heavily-loaded processor) to a new "child" block (on the lightly-loaded processor). The child processor pushes those particles so long as they remain inside the window region. Particles that enter/exit the window migrate between the parent and

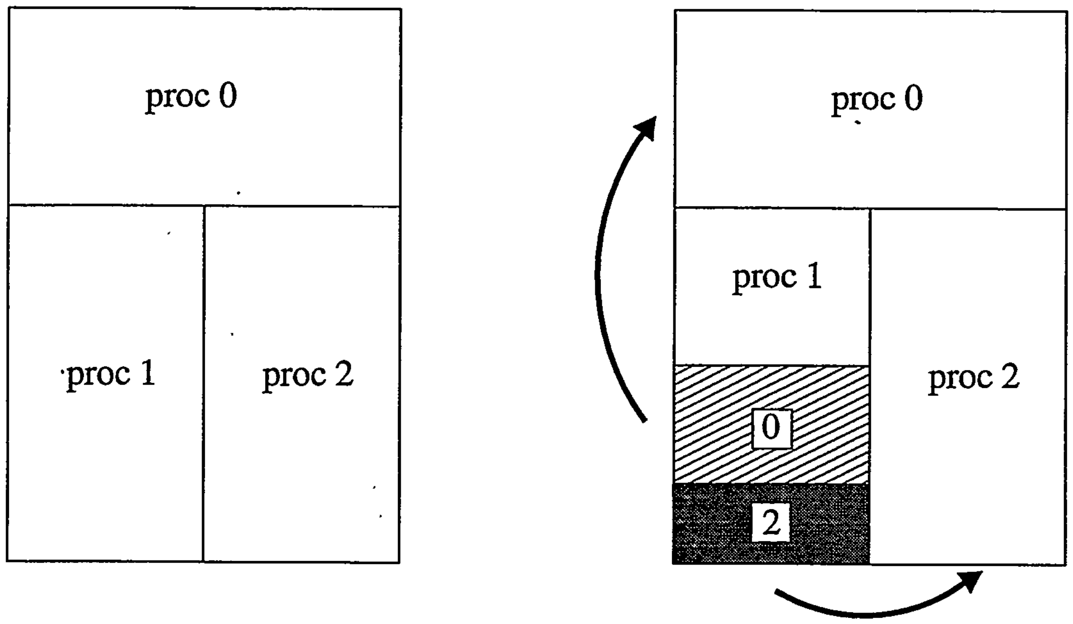


Figure 8: *A three-processor decomposition with one block assigned to each processor (left side). Shaded regions of processor 1's heavily-loaded block are designated as "windows" and assigned to processors 0 and 2 (right side). Processors 0 and 2 push particles in the shaded regions to achieve better load-balance.*

child processors. For example, on the left side of the figure, each of 3 processors initially owns one block. If processor 1's block (the parent) has too many particles, two (shaded) window regions (the child blocks) are created, one each for processors 0 and 2. Thus processor 1 will only push particles in the remaining (unshaded) region of its block. Processors 0 and 2 will each push particles in two blocks, the original block they owned and a new child block.

Note that within the window regions, the child processor will only push particles; the parent processor will continue to compute $\vec{E}$ and $\vec{B}$ field updates in these regions. This is to maintain load balance in the field computations. Since the child's new particles actually reside (in a geometric sense) in the parent block, this will require communication of additional field information between the parent and child processors. The hope is that the extra overhead of this particle and field communication will be more than compensated for by achieving load-balance in both the particle push and field update. As we shall see, parallel QS attempts to maintain this balance by dynamically adjusting the number of windows and their sizes.

6.3 Details of Initiation

The load-balancing option in parallel QS is enabled by use of the "CUSTOM LOADBALANCE TOL1 TOL2" command in *qcks.in*, where $TOL1, TOL2 \geq 1.0$ are real numbers explained below. This invokes a one-time call to the *parallel_balance_setup* routine from *qsinit.F* and a call to the *parallel_balance* routine every timestep from *qcks.F*. The setup routine pre-computes various static quantities that will be useful in the balancing procedure. These include copies of the global 1-d grid arrays for all user blocks which are used to create grid arrays for the child blocks. The setup routine also allocates memory for various arrays used by the balancer to store particle counts and block sizes; these are documented in *parallel.inc.*

The *parallel_balance* routine creates window-block connections between parent/child pairs of processors if it detects imbalance in the current timestep. It does this by computing the average particle count across all processors and the maximum particle count on any processor. The ratio of max/ave is a measure of load-imbalance in the particle push operation, where a value of 1.0 represents perfect balance. If this ratio is smaller than TOL1, the first parameter in the CUSTOM LOADBALANCE command, then balance is adequate and the routine simply exits. When it decides to turn load balancing ON, the routine proceeds with the operations listed in Figure 9.

(1)	Count particles in planes of all blocks.
(2)	Perform serial balancing to determine parent/child pairings.
(3)	Build 1-d grids for child blocks.
(4)	Set 3-d bgcell and bgijk arrays for parent/child connections.
(5)	Communicate 3-d CIM array from parent to child blocks.
(6)	Update particle migration neighbor lists.
(7)	Create E/B and J/Q field connections between parent/child blocks.

Figure 9: *Steps to initialize particle load-balancing in a parallel* QUICKSILVER *simulation.*

In principle, window regions within a block could be of any size and shape. For simplicity, however, we restrict their shape by only making 1-d partitions perpendicular to the longest dimension of a parent block. For example, if a particular block is $10x20x15$ grid cells, then all window regions within that block will be $10xNx15$ in size, where N is some number of xz planes. Choosing the longest dimension (independently in each block) allows for the finest granularity in this style of partitioning. To determine the optimal value(s) of N, we first must count the number of particles in each plane of every block. This is done in step (1). Each processor loops over all its particles, extracts a current ijk cell index, and increments a counter associated with the appropriate plane. This 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 stores a count of the total number of particles on each processor (accumulated via MPI_Allgather).

These two lists are used in step (2) by the *balance_serial* routine, along with static information (precomputed in *parallel_balance_setup*) that describes the number and sizes of blocks on all processors. *Balance_serial* computes the optimal sizes of window regions for each parent block and assigns corresponding child blocks to specific processors. It is a serial routine in the sense that every processor performs the entire balance operation without communication; each processor receives the same global inputs and computes the entire list of parent/child pairings for all processors. This means this portion of the load-balance creation is not scalable in a parallel sense, but in practice it is a quick operation to produce these pairings (even for 1000s of windows on 1000s of processors), and it would also be difficult to achieve as near-optimal a result if each processor did not have global information about the state of imbalance.

The serial balancer works via an iterative process to reduce imbalance. Each time through its main loop, one window region is created in a block on a parent processor and a corresponding block is assigned to a child processor. This operation reduces the global imbalance before the next iteration of the loop. The loop continues until the imbalance is less than TOL2, the 2nd parameter in the CUSTOM LOADBALANCE command, or until no further progress can be made.

At any iteration, the processor with the most particles is chosen as the parent and the processor with the least as the child. The optimal number of particles to migrate from one to the other (the *itarget* variable in the code) is set equal to the smaller of either processor's particle count variation from perfect balance. The routine then checks all possible partition placements (or "cuts") for each of the parent's blocks. Starting from each end of the block, the cut position is incremented one plane at a time. The total number of particles in the proposed window region is tallied (by summing over plane counts) and compared to the desired *itarget*. The "optimal" cut is the one which will create a window block that migrates a number of particles closest to *itarget* without exceeding it. This may be an entire parent block or a fraction thereof. In either case, the window planes are masked out in the parent block so they will not be considered again as a window candidate. The parent and child processor IDs are stored along with the extent of the window region in the parent (see outputs of the *balance_serial* routine). The particle counts of the parent and child processors are then adjusted to reflect the one-way migration, and the routine proceeds to the next iteration.

In step (3) of Figure 9, each processor uses the lists of child blocks returned by *balance_serial* to update its copy of the *block2proc* and *global2local* vectors. As discussed in Section 5.2, these store a global mapping of blocks to processors (see *parallel.inc*). Each processor also scans the output to determine if it is participating in the load-balance operation and if so, whether it is a parent or child and which of its blocks are affected. If it owns new child blocks, each processor augments its 1-d block arrays (e.g. *imax, jmax, kmax, locb, lenblk*) to reflect the new sizes. It also initializes 1-d grid arrays for these blocks using the global grid information stored for the original user block definitions. Six 1-d arrays are needed to push particles in the new child blocks; these are the full grid and reciprocal grid values in *bgxif, bgxjf, bgxkf, bgrdif, bgrdjf, bgrdkf.*

During this operation, the child processors also check that they have sufficient memory for storing the extended 1-d arrays and the new 3-d field arrays that will be associated with their new blocks. When the CUSTOM LOADBALANCE command is used in *qcks.in*, the *memalloc* routine allocates extra space at the end of the appropriate 1-d and 3-d arrays in anticipation of child-block creation. As discussed in Section 4, the amount of extra memory is governed by use of two new parameters that must be specified in the *pvlx* file. If the extra memory is insufficient on any processor for the newly defined parent/child blocks, then a warning message is printed and the routine exits without turning the load balancer on. If this occurs frequently in a particular run, the user should boost the settings for the *wbsca1* and *wbsca3* parameters in the *pvlx* file.

In step (4), the 3-d *bgcell* and *bgijk* arrays are modified as needed in parent blocks and are initialized in child blocks. Recall that these arrays are used in the particle push to determine when a particle needs to migrate to which processor and what new cell it will reside in on the receiving processor. Within a parent's window region, *bgcell* and *bgijk* are set to point to the new child block. The usual inter-block communication operation is then performed without including child blocks. This is the same communication discussed in Sections 5.2 and 5.3 for the initial setup of the *bgcell* and *bgijk* arrays. This operation updates all ghost cells (except in child blocks) so that they point at any newly created blocks. A second communication is then performed directly from parent processors to their partner child processors. Each parent sends *bgcell* and *bgijk* values for the window region and the cells that immediately surround it. The child processors use these values to initialize the interior and ghost cells of each child block. Note that the child processor has no knowledge of the blocks that border its corresponding window region in the parent block. This window region could be adjacent to other windows or to other parent blocks (which could also contain windows). This 2-step communication operation (parent$\leftrightarrow$parent followed by parent$\rightarrow$child) allows the child processor to receive that information indirectly from its parent processor without having to communicate with the (unknown) owners of the other blocks.

In step (5), cell-wise CIM values are sent from each parent block to the corresponding child block. This operation sets the interior and ghost cell values for the 3-d *bgcim* array on the child processor in its new block(s). This array stores conductor and dielectric information and is the final 3-d array needed by the child processor to enable it to accurately push its new particles.

The creation of window blocks means that a processor may now need to exchange particles with new neighbor processors. Parent processors will be sending particles to child processors (and vice versa) who may not previously have been an exchange partner. Two window regions within a parent block may border each other (as in Figure 8) which will require child processors to communicate with each other. Parent and child processors associated with a window region in one block may also have new exchange partners due to window regions in a second block that borders the first. In step (6), the particle migration neighbor lists (discussed in Section 5.4) are recomputed to reflect these new partners. This is done via a call to *neighbor_init* which scans the modified *bgcell* values computed in step (4) for both parent and child blocks.

Finally, in step (7), new connection plans are formed for parent and child blocks which need to exchange field data. Child processors need average $\vec{E}$ and $\vec{B}$ field values to push particles. These corner-centered average quantities are computed by parent blocks after the normal $\vec{E}$ and $\vec{B}$ field update. Each child processor needs values at all points inside and on the surface of its child block(s). These correspond to points in the interior and on the surface of the window region in the parent block. Also, after the child processor pushes its particles, it will accumulate $\vec{J}$ and Q field values in its 3-d arrays. These need to be sent back to the parent processor from each child. Similar to the discussion in Section 5.3, this encompasses all $\vec{J}$ and Q values inside and on the surface of the child block as well as those a half-grid and full-grid cell outside. These will be summed by the parent processor to cells in its window region and immediately surrounding it, including ghost cells of the parent block if necessary.

The plans for storing these one-sided communication patterns (parent-to-child, or child-to-parent) are the same data structures used for other kinds of inter-block connections (see Section 5.3 and *parallel.h*). However, because the overlap of parent/child grids is defined explicitly by the window region, it is not necessary to use all the logic of the overlap calculation in *parallel_connect_create* to create the new plans. Instead we call a simpler routine, *parallel_connect_window_create*, which constructs each processor's local plan directly from the global list of window-block connections computed by *balance_serial*. The same routine is used to create plans for communicating cell-centered 3-d arrays (*bgcell*, *bgijk*, and *bgcim*) from parent blocks to children. These plans were used in the load-balancing initiation phase in steps (4) and (5).

6.4 Details of Operation

We now describe how load balancing is performed within the context of a normal parallel QS timestep. Figure 10 outlines the extra operations that occur each timestep when the CUSTOM LOADBALANCE command is used.

Steps (1,2,5,7) are the normal parallel QS operations that occur every timestep; see Figure 2 for a comparison. Steps (3,4,6) are new balancing operations, which may occur depending on whether load balancing is turned ON or OFF in the current timestep.

The point in the QS timestep at which load-balancing is turned ON is at step (4a), after $\vec{E}$ and $\vec{B}$ fields have been updated and communicated, but before the particle push. The number of particles on each processor is tallied and the imbalance criterion described in the previous section is applied. If balancing is currently OFF and there is sufficient imbalance, then balancing is turned ON and window block sizes and connections are computed and initialized as described in detail in the previous section.

The particle push then proceeds in step (5) and each parent processor will migrate a fraction of its

(1)	Leapfrog update of $\vec{E}$,$\vec{B}$ fields on grid
(2)	Communicate $\vec{E}$, $\vec{B}$ fields between blocks
(3*)	If BALANCE ON, communicate $\vec{E}$, $\vec{B}$ fields in windows
(4a*)	If imbalanced and BALANCE OFF, create windows and turn BALANCE ON
(4b*)	If imbalanced and BALANCE ON, turn BALANCE OFF
(5)	Create, advance, delete, and migrate particles
(6*)	If BALANCE ON, communicate $\vec{J}$, Q fields in windows
(7)	Communicate $\vec{J}$, Q fields between blocks

Figure 10: QUICKSILVER *timestep with load-balancing enabled. Starred steps are the additional load-balancing operations.*

particles to its partner child processor(s). Note that during this first timestep with the balancer ON, the parent processors still perform the (imbalanced) push operation. This includes the accumulation of $\vec{J}$ and Q fields in the parent blocks (steps 5 and 7), so actually there is no need to send them in step (6) from child to parent at the end of the first timestep.

At the beginning of the next timestep, $\vec{E}$ and $\vec{B}$ fields are updated. Child blocks do not participate in this operation, so it remains load-balanced across the original (static) block decomposition. After the usual ghost-cell exchange of these field quantities in step (2) between parent processors, an additional call is made in step (3) to *parallel_connect* with the new plan (*plan_eb_window*) that sends average $\vec{E}$ and $\vec{B}$ fields in a one-way fashion from parent blocks to child blocks. Since child blocks are now populated with particles and fields, the ensuing particle push is now balanced. Child processors accumulate $\vec{J}$ and Q field values in the usual way in the child block's 3-d arrays. At the end of the timestep, a call is made in step (6) to *parallel_connect* with the other new plan (*plan_jq_window*) that sends $\vec{J}$ and Q fields directly from child blocks to parent blocks. This is done prior to the usual $\vec{J}$ and Q exchanges between parent blocks in step (7), so that parent processors will have fully-summed field values in their blocks (interior and ghost cells) before participating in that operation.

QS continues calling *parallel_balance* in step (4b) every timestep until the routine detects that particles are again imbalanced (greater than TOL1). When this occurs, load-balancing is turned OFF. This is accomplished by setting all the 3-d *bgcell* and *bgijk* values in the parent and child blocks to point only to parent blocks. This will insure that all particles in any child block migrate back to a parent block on this timestep. The timestep ends with one final communication of $\vec{J}$ and Q fields from child blocks to parent blocks in step (6). On the next timestep the load balancer cleans up after itself by destroying all the plan data structures used for load-balance communication and restoring the particle neighbor lists to their original pre-load-balancing values.

Performance results for parallel QS runs using this load-balancing technique are presented in Section 7. We conclude this section with several observations about the method and its implementation.

(1) A key advantage of this algorithm as implemented in parallel QS is that it required only minor modifications to existing code. In particular, the particle push and field update coding did not change at all. Additional communication calls were added (in *fldslv* and *jqdnsy*) for $\vec{E}/\vec{B}$ and $\vec{J}/Q$ field exchanges between parent and child blocks, but the algorithm for inter-processor particle migration also did not need

to be changed.

(2) As outlined above, because of the way that load-balancing is turned ON and OFF, there is a one-timestep delay between when imbalance is initially detected and when the particles are actually pushed by the new child processors. Also, when load balancing is turned off, particles are always returned to their parent processors before a re-balancing can be performed (e.g. on the next timestep). This means that if load-balancing is switching ON and OFF rapidly every N steps due to fast-varying particle densities, then even in the best-case scenario, there is always one step out of N where particles are wholly imbalanced (pushed only by parent processors).

(3) There is an overhead cost associated with turning load balancing ON and OFF. This is 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 ON for several steps or more, this cost is also amortized over the duration of the balancing. This will certainly be the case if particle densities vary only slowly in time (relative to a timestep), which is the case for many problems.

(4) The per-timestep cost of load-balancing is the extra field communication and particle migration that must be done 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 need to migrate on subsequent timesteps. Also, in contrast to the more general and irregular inter-block communication between parent blocks, the extra field communication for load-balancing is of a one-to-one nature: one parent block sends (or receives) field quantities to (or from) one child block. There is also an additional memory overhead incurred by load balancing. This is the additional 1-d and 3-d array storage that must be set aside by the *memalloc* routine to allow for new child block creation.

(5) The relative costs/benefits of using the CUSTOM LOADBALANCE option can be analyzed by examining the balance statistics printed by parallel QS at the end of a run. The "actual" particle imbalance (max/ave averaged over all timesteps) will be printed whether load balancing is used or not. This can be compared between runs with load-balancing enabled versus disabled, as can the change in particle-push and field-communication timings. The "ideal" imbalance that is printed is a measure of the best load-balance QS could hope to achieve given the 1-d granularity of the window-block partitioning. It is computed from the final imbalance remaining after the *balance_serial* routine has created all the windows it can. Additional statistics are also printed for the average duration that the load-balancer is on and the number and aggregate sizes of window blocks.

7 Benchmark Calculations

In the course of developing parallel QS, we performed a variety of small test runs to debug various changes and features we were adding to the code and to see if everything worked on varying numbers of blocks and processors. In this section we describe a series of full-scale benchmark runs we ran with the finished code to test its overall performance and parallel scalability, as well as its load-balancing capabilities.

The tests were performed on two parallel machines at Sandia. The first is the Intel Tflops machine, which is a conventional massively parallel machine built by Intel for Sandia's ASCI program. It consists of 333 MHz Intel Pentium processors interconnected by an Intel-proprietary backplane and network interface chips. Some of the tests were also run on Sandia's new Computational Plant (CPlant) machine which is a Beowulf-style [11] cluster of workstations built in-house by Sandia. It consists of 500 MHz DEC Alpha workstations connected by Myrinet.

We first discuss the code's performance on large-scale problems with a uniform spatial distribution of work. In the final two subsections, problems that require static and dynamic load-balancing are benchmarked.

7.1 Performance and Scalability

The first benchmark problem is a fields-only calculation (no particles) on a single 80x100x96 grid block of 768,000 grid cells run for 2000 timesteps with an explicit time integration scheme. A Poisson inlet boundary condition is applied to one face of the block, with perfect electric conductor (PEC), perfect magnetic conductor (PMC), and outlet conditions applied to the other faces. The block interior contains three conducting strips. Several HISTORY FIELD commands are defined in the input script for diagnostic purposes.

The CPU time for running this test problem on various numbers of processors of both the Tflops and CPlant machines is shown in Table I. The resulting parallel efficiency is computed by dividing the one-processor 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 Tflops benchmark would run optimally (100% efficient) in 2754.2/1024 = 2.69 seconds. Since it actually ran in 9.08 seconds, it is 2.69/9.08 = 29.6% efficient, which is 0.296*1024 = 303 times faster than it ran on a single processor.

	Tflops		CPlant	
Procs	CPU time	Parallel Eff	CPU time	Parallel Eff
1	2754	100.0	1113	100.0
2	1337	103.0	604.5	92.0
4	643.9	106.9	309.2	90.0
8	326.4	105.4	161.0	86.4
16	172.5	99.8	88.7	78.4
32	90.8	94.8	57.2	60.8
64	44.7	96.3	42.2	41.2
128	26.8	80.3	34.1	25.5
256	17.9	60.1		
512	12.4	43.4		
1024	9.08	29.6		

Table I: *CPU time (seconds) and parallel efficiency for a fields-only simulation of fixed-size run on varying numbers of processors on the Intel Tflops and Alpha-based CPlant machines. The problem had 768,000 grid cells and was run for 2000 timesteps.*

This problem was designed to be a large calculation that an analyst might reasonably perform on a single-processor workstation. It illustrates the speed-up offered by the parallel version of QS even when the number of grid cells per processor becomes small (a few hundred for 1024 processors). The reduced efficiencies are due to the increased cost of field communication versus computation as the surface-to-volume ratio of each processor's block increases. When the blocks are too small, the communication cost of exchanging field information with neighboring processors dominates.

The super-linear performance (efficiencies greater than 100%) on a few processors of Tflops is due (we believe) to cache effects. When the problem size per processor is reduced enough that significant portions of the field arrays fit in cache, the field update computations actually speed-up. The CPlant machine has slower message-passing software and communication hardware than Tflops; hence the parallel efficiencies for CPlant fall off much more quickly than for Tflops. However, the one-processor timing on the DEC Alpha processor is about 2.5 times faster than on the Intel Pentiums. Thus the code's raw speed on CPlant is still competitive with Tflops out to 128 processors.

The second benchmark problem is also a fields-only calculation, but the number of grid cells is scaled with the number of processors used. This benchmark illustrates the very large size of fields-only problems that can be run on a large parallel machine. One large user block is specified for each run, so that when partitioned for P processors, each processor owns a 30x30x30 block of 27,000 grid cells. As before, explicit timestepping is used, a mixture of Poisson, PEC, PMC, and outlet boundary conditions were applied to the user block, and several HISTORY diagnostics were specified. This time the problem was run for 10,000 timesteps to allow the waveform incident at the Poisson inlet to travel throughout the simulation domain.

The CPU time for running the second test problem on Tflops and CPlant is shown in Table II. The total number of grid cells is also listed, from 27,000 on one processor to 86.4 million on 3200 processors. Because this is a scaled-size problem, the parallel efficiencies are much better than in the previous fixed-size case. On Tflops there is still some degradation in performance on very large numbers of processors, presumably due to message contention effects with each processor exchanging field data for its block with 26 surrounding blocks (processors).

		Tflops		CPlant	
Procs	Grid Cells	CPU time	Parallel Eff	CPU time	Parallel Eff
1	27000	412.0	100.0	129.6	100.0
2	54000	414.8	99.3	147.3	88.0
4	108000	419.6	98.2	169.0	76.7
8	216000	416.4	98.9	194.4	66.7
16	432000	417.8	98.6	214.3	60.5
32	864000	425.2	96.9	247.0	52.5
64	1,728,000	430.0	95.8	283.1	45.8
128	3,456,000	433.8	95.0	293.7	44.1
256	6,912,000	442.4	93.1		
512	13,824,000	452.1	91.1		
1024	27,648,000	480.4	85.8		
2048	55,296,000	558.8	73.7		
3200	86,400,000	610.0	67.5		

Table II: *CPU time (seconds) and parallel efficiency for a fields-only simulation of scaled-size run on varying numbers of processors on the Tflops and CPlant machines. The problem size is 27,000 grid cells per processor and was run for 10,000 timesteps.*

We also ran one much larger billion-grid-cell fields-only calculation on 3200 processors of the Intel Tflops with similar boundary conditions and diagnostic settings. A run of 1000 timesteps required 545.8 seconds. We estimated its parallel efficiency at 87.3% using the one-processor timings in Table II as a reference point. It is interesting to note that each grid cell in a fields-only calculation uses 25 single-precision words or 100 bytes of memory. Thus the billion-cell calculation required about 100 Gbytes of storage. However, each of the 3200 processors on Tflops has 256 Mbytes of memory for an aggregate memory of 800 Gbytes. Thus this very large problem required less than 15% of the Tflops memory to run.

The one-processor timing data in Table II can also be used to compute a "grind" time for fields-only calculations with parallel QS. On Tflops the explicit timestepping integrator requires 1.5 microseconds per grid-cell per timestep. On CPlant it is approximately 0.5 microseconds per grid-cell per timestep. An explicit update in a single grid cell requires about 60 floating-point operations (flop). Hence a Tflops processor is running this benchmark at about 40 Mflops (million flop/sec) and a CPlant processor at about 120 Mflops. The billion-cell benchmark on 3200 Tflops processors runs at about 110 Gflops (billion flop/sec).

The third benchmark problem is a fixed-size particle calculation. A one-block grid of 64x64x64 = 262,144 grid cells is populated with 3.15 million particles. This particle/cell count of 12 is typical of many QS problems. Each grid cell is pre-loaded with 6 electrons and 6 positrons, each of which is 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. PMC (mirror) boundaries were applied to all 6 faces of the user block. Because the number density associated with the particles is set to a small value and pairs of oppositely charged particles move in the same direction (no net current), the particles in this problem are essentially non-interacting. Over time they stream back and forth within the user 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 HISTORY FIELD and PARTICLE settings were specified in the input script so as to generate a variety of diagnostic outputs.

The timing results for running this problem on various numbers of Intel Tflops processors are shown in Table III.[5] These results exhibit better scalability than their fields-only counterparts in Table III, because there is more computational work required to push particles on a per-grid-cell basis. On one processor the code is spending 92% of its time in the particle push routines, and 7% in the field update. Even with high-velocity particles (1/2 grid cell per timestep) causing a relatively large fraction of particles to migrate to new blocks (and processors) each timestep, the particle migration time remains only a small fraction of the overall run time even for very large processor counts.

As before, this benchmark was designed to be at the high end of the problem size an analyst might run on a fast desktop workstation. The timings indicate that the Tflops machine is able to run this problem in a highly scalable fashion out to many hundreds of processors.

A fourth benchmark problem is a scaled-size particle simulation with each processor owning a 30x30x30 block of 27,000 grid cells. As in the previous particle problem, each grid cell is 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.

Timing results for this problem are shown in Table IV. 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, the code exhibits excellent scalability of over 90% parallel efficiency for all numbers of processors.

The memory requirements of this benchmark were 9 words (36 bytes) per particle and 100 bytes per grid cell. Thus the largest problem required about 46 Gbytes or roughly 6% of the memory available on 3200 Tflops processors.

[5] We were unable to run our particle benchmarks successfully on CPlant (as of December 1999) due to software bugs in the CPlant system software. Hence no CPlant timings are included in the particle benchmark tables.

	Intel Tflops	
Procs	CPU time	Parallel Eff
1	7486	100.0
2	3783	98.9
4	1890	99.0
8	972.8	96.2
16	483.7	96.7
32	240.8	97.2
64	124.5	94.0
128	64.1	91.2
256	34.5	84.8
512	19.1	76.6
1024	11.8	62.0

Table III: *CPU time (seconds) and parallel efficiency for a particle simulation of fixed-size run on varying numbers of processors on the Intel Tflops. The problem had 262,144 grid cells and 3.15 million particles and was run for 256 timesteps.*

As before, the "grind" times for particle pushing can be computed from the one-processor timings in Table IV. Since particle pushing consumes 92% of the time, it is requiring 8.6 microseconds to push one 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 update takes 280 flops. Thus a Tflops processor is pushing particles at a rate of 44 Mflops and doing implicit field updates at a rate of 36 Mflops. The billion-particle problem is running at an aggregate speed of 118 Gflops on 3200 processors.

Finally, for comparison purposes, we combine the parallel efficiency data from the previous four tables in one plot, shown in Figure 11. All one-processor timings are shown as 100% efficient. Hence the figure disguises the raw performance difference between the Tflops Pentium and DEC Alpha CPlant processors. As expected, the figure shows that scaled-size problems outperform fixed-size ones on both machines as processor counts increase. This is true for both fields-only and particle problems. The faster computational and slower communication rates on CPlant (relative to Tflops) degrade its parallel efficiency much more quickly than occurs on Tflops.

7.2 Static Load-Balancing

In this section, benchmark results are presented for fixed- and scaled-size problems that require static load-balancing. By "static", we mean that the particle load is spatially inhomogeneous, but does not vary in time.

The first benchmark problem is similar to the fixed-size particle benchmark of the previous section. A one-block domain of 64x64x64 = 262,144 grid cells with mirror boundaries is populated with 3.15 million particles. Recall that in the previous uniform-load problem, each cell in the simulation was pre-loaded with 12 particles (6 sets of 2 each). Each set was given a velocity in a different coordinate direction ($\pm x, \pm y, \pm z$) so that they moved approximately 1/2 grid cell per timestep. Because each set filled the entire domain, as

			Intel Tflops	
Procs	Grid Cells	Particles	CPU time	Parallel Eff
1	27000	324,000	604.6	100.0
2	54000	648,000	608.8	99.3
4	108000	1,296,000	612.7	98.7
8	216000	2,592,000	635.4	95.2
16	432000	5,184,000	637.0	94.9
32	864000	10,368,000	639.0	94.6
64	1,728,000	20,736,000	646.8	93.4
128	3,456,000	41,472,000	649.0	93.2
256	6,912,000	82,944,000	650.3	93.0
512	13,824,000	165,888,000	655.3	92.3
1024	27,648,000	331,776,000	655.7	92.2
2048	55,296,000	663,552,000	656.2	92.1
3200	86,400,000	1,036,800,000	662.8	91.2

Table IV: *CPU time (seconds) and parallel efficiency for a particle simulation of scaled-size run on varying numbers of processors on the Intel Tflops. The problem had 27,000 grid cells and 324,000 particles per processor and was run for 200 timesteps.*

the simulation progressed, particle density stayed essentially constant throughout the simulation box.

In this benchmark, each of the 6 sets of particles is only loaded in 1/4 of the simulation domain (at a 4 times higher density). For example, the set of particles that moves in the $+x$ direction fills a 64x16x64 slab (thin in the y dimension) of grid cells with 8 particles/cell. Similarly, the sets that move in the $\pm y$ or $\pm z$ dimensions fill 64x64x16 and 16x64x64 slabs respectively.

The net effect of this strategy is three-fold. First, the total number of particles in the problem (3.15 million) is the same as in the uniform case. Second, the initial distribution of particles is very inhomogeneous. Approximately 42% (27/64) of the cells in the simulation have no particles; another 27/64 of the cells have 16 particles/cell; another 9/64 have 32 particles/cell; and 1/64 of the cells have 48 particles/cell. Finally, because the particles fill the entire box dimension in the direction they move, the particle count in each cell stays essentially constant for the duration of the simulation.

The benchmark timings listed in Table V are for a simulation run on the Intel Tflops of 256 timesteps with a 3-stage implicit integration scheme for the field solves. As before, this means that an individual particle traverses the simulation domain twice to return (roughly) to its initial position.

The first three columns of data are for runs with load-balancing disabled. As expected, the relatively poor parallel efficiencies are due to particle imbalance across processors. As discussed in Section 6, "imbalance" is defined as the ratio of the *maximum* particle count on any processor to the *average* particle count across all processors. The maximum imbalance in this problem is 4.0 which occurs on 64 or more processors when one processor owns a region where all cells have 48 particles (versus the global average of 12).

The second set of 3 columns are results for runs with load-balancing enabled. Because the particle distribution is essentially static, on the first timestep the balancer computes an initial set of window blocks that cause an equal redistribution of particles. This re-balanced state persists until the end of the simulation.

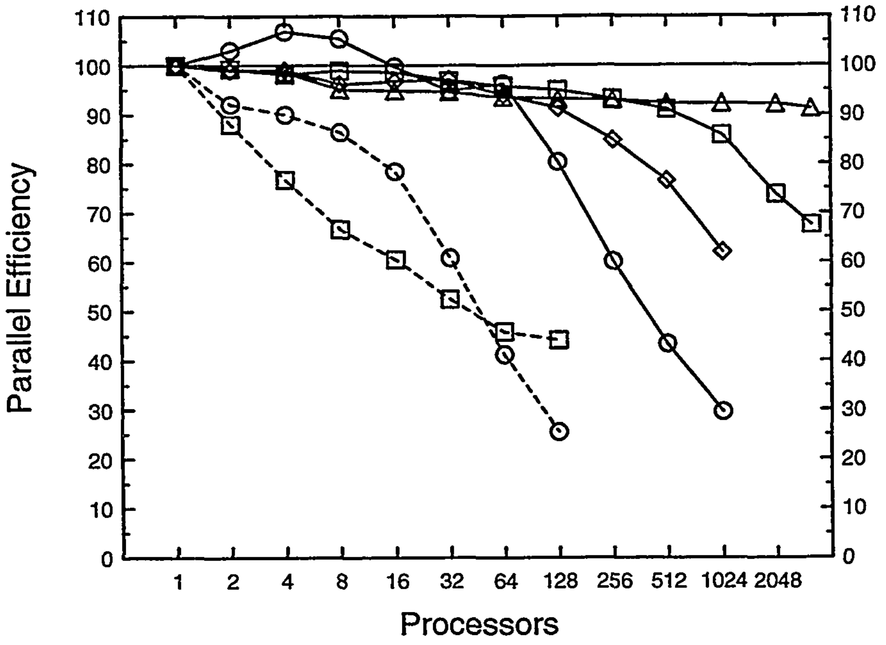


Figure 11: *Parallel efficiencies for the data in Tables I-IV. Solid lines are runs on the Intel Tflops; dotted lines are for CPlant. Data for 4 benchmarks are shown: fixed-size fields-only (circles), scaled-size fields-only (squares), fixed-size particles (diamonds), and scaled-size particles (triangles).*

As the efficiency results indicate, this rebalancing is quite good until there are so many processors that each one owns only a very small (e.g. 8x8x8) sub-domain of grid cells. The imbalance figures with load-balancing ON indicate the window-block redistribution is quite effective at equalizing particle counts across processors.

The last 3 columns are results from Table III for the same size problem (262,144 grid cells, 3.15 million particles) where all cells have 12 particles — a uniform load. These timings represent the "best" result that the load-balancer could hope to achieve if it were 100% successful at balancing the particle load and incurred no overhead in its re-distribution operations. The small speed-up for this problem on one processor on the non-uniform problems (7210 versus 7486 seconds) is due (we believe) to cache effects in gathering/scattering data from the particles to the grid. When particles only interact with a small fraction of the grid arrays, more of the grid arrays can remain in fast-access cache memory, resulting in a small net speed-up. We note that the efficiencies listed in the table for load-balancing ON are computed using the one-processor timing for balancing OFF as a baseline, not from the slower one-processor uniform timing.

The next benchmark calculation is for a scaled-size problem requiring static load-balancing. Similar to the scaled-size particle benchmark of the previous section, each processor owns a 30x30x30 block of grid cells. The global domain is pre-loaded with 6 sets of particles. As in the fixed-size static load-balancing benchmark, each set fills only a fraction of the global domain, but at a higher density. For this problem a compression factor of 10x was used instead of 4x. Thus on 8 processors, the global simulation box is 60x60x60 and each set of particles is a 6x60x60 slab of particles. As before each of the 6 slabs is oriented differently within the box and its particles are given initial velocities in different coordinate directions. In these runs, about 73% ($9^3/10^3$) of the global box is devoid of particles, 1/1000 of the grid cells have 120 particles/cell (10x the average), and the particle density again stays roughly constant for the duration of the

	Balance OFF			Balance ON			Uniform Problem		
Procs	CPU	Eff	Imbal	CPU	Eff	Imbal	CPU	Eff	Imbal
1	7210	100.0	1.0	7281	99.0	1.0	7486	100.0	1.0
2	4723	76.3	1.33	3668	98.3	1.006	3783	98.9	1.0
4	2916	61.8	1.67	1882	95.8	1.003	1890	99.0	1.0
8	1784	50.5	2.0	949.6	94.9	1.004	972.8	96.2	1.0
16	1169	38.5	2.67	494.8	91.1	1.048	483.7	96.7	1.0
32	726	31.0	3.33	247.0	91.2	1.009	240.8	97.2	1.0
64	434.4	25.9	4.0	127.5	88.4	1.012	124.5	94.0	1.0
128	218.6	25.8	4.0	66.9	84.1	1.012	64.1	91.2	1.0
256	112.4	25.1	4.0	38.1	73.9	1.012	34.5	84.8	1.0
512	57.6	24.4	4.0	20.5	68.7	1.012	19.1	76.6	1.0
1024	32.9	21.4	4.0	16.0	44.0	1.012	11.8	62.0	1.0

Table V: *Performance (CPU-time, parallel-efficiency, load-imbalance) for a fixed-size particle simulation with a static imbalance in particle load on the Intel Tflops. Results with the load-balancer turned OFF and ON are shown as well as for a problem of the same size with uniform particle load.*

simulation.

Results for running this benchmark problem for 200 timesteps are given in Table VI for varying numbers of processors on the Intel Tflops. The biggest simulation was on 1024 processors which had about 27.6 million grid cells and 332 million particles. As in the previous table, results for load-balancing disabled versus enabled are shown, as well as results for runs of the same-size uniform-load problems from Table IV.

As before, the parallel efficiency for runs without load-balancing quickly degrade as particle imbalances near a peak of 10x. The runs with load-balancing ON are much more efficient though there is some extra overhead when compared to the uniform-load runs. Certain processor configurations (e.g. 32 procs) also do less well. This is probably due to the limited set of possible window-block sizes that the load balancer has to choose from for a given grid and processor configuration. This effect is likely exacerbated for this problem which contains some cells with 120 particles and others with none. Overall however, the timing results in this and the previous table are evidence that our load-balancing technique of reassigning particles to lightly-loaded processors (via window blocks) is very beneficial for simulations with static variations in particle densities.

7.3 Dynamic Load-Balancing

We now benchmark problems that require dynamic load-balancing. By "dynamic" we mean that particle densities vary not only spatially but also in time.

The first benchmark is similar to the previous fixed-size particle benchmarks. It has 64x64x64 = 262,144 grid cells and 3.15 million particles. The particles are pre-loaded in 3 sets (not 6 as before), each of which is a slab that fills 1/4 of the global domain. The first set is a 16x64x64 grid-cell slab (thin in x) with 16 particles/cell and each particle is given a velocity in the $+x$ direction. Similarly, the other 2 slabs are thin in y and z and their particles move in the y and z directions respectively.

This particle assignment strategy gives the same initial particle distribution as in the static load-balancing

	Balance OFF			Balance ON			Uniform Problem		
Procs	CPU	Eff	Imbal	CPU	Eff	Imbal	CPU	Eff	Imbal
1	582.5	100.0	1.0	578.1	100.1	1.0	604.6	100.0	1.0
2	758.9	76.8	1.33	593.0	98.2	1.002	608.8	99.3	1.0
4	934.6	62.3	1.67	599.1	97.2	1.003	612.7	98.7	1.0
8	1135	51.3	2.0	613.1	95.0	1.03	635.4	95.2	1.0
16	1491	39.1	2.67	654.3	89.0	1.10	637.0	94.9	1.0
32	1860	31.3	3.33	737.8	79.0	1.25	639.0	94.6	1.0
64	2226	26.2	4.0	672.0	86.7	1.10	646.8	93.4	1.0
128	2935	19.8	5.33	675.5	86.2	1.09	649.0	93.2	1.0
256	3669	15.9	6.67	754.7	77.2	1.23	650.3	93.0	1.0
512	4388	13.2	8.0	710.5	82.0	1.10	655.3	92.3	1.0
1024	4724	12.3	8.67	755.0	77.2	1.19	655.7	92.2	1.0

Table VI: *Performance (CPU-time, parallel-efficiency, load-imbalance) for scaled-size particle simulations with a static imbalance in particle load. Results with the load-balancer turned OFF and ON are shown and for problems of the same size with uniform particle load.*

benchmark. The 16x16x16 grid-cell region of highest density (48 particles/cell) is in the lower-left front corner of the global simulation box. As time advances, this region of high density migrates diagonally toward the upper-right back corner of the box. All particles are given velocities such that they require about 2 timesteps to traverse a grid cell. Thus in a 256-timestep simulation, the region of highest density moves from the lower-left corner of the box to the upper-right and back to the lower-left. This is a quite rapid fluctuation in particle density throughout the simulation domain for which the load-balancer must attempt to compensate.

In Table VII we show timing results for this fixed-size dynamic problem running on varying numbers of Intel Tflops processors. As before, results with the load-balancer turned off show significant degradation in parallel efficiency. With the load-balance turned ON, the fast-moving particles trigger the balancer to re-balance the particle load every few timesteps. The net result is an improved efficiency though not as dramatic an improvement as in the static case.

As more and more processors are used, a processor's sub-domain becomes smaller and the fast-moving regions of high particle density cause the relative imbalances between processors to fluctuate more rapidly. For example, on 32 processors this 256-timestep run re-balanced 21 times; on 512 processors re-balancing was performed 66 times. This contributes to a degradation in parallel efficiency in two ways. First, there is the overhead cost of setting up a new window-block decomposition each time re-balancing is done. Second, as described in Section 6, between re-balancings all the particles migrate back to their original processors and are pushed in an unbalanced fashion for one timestep. This effect is reflected in the imbalance column (for load balancing ON) in the table, since it is an average over the imbalance present at every timestep. Thus on 512 processors, the 26% of the timesteps (66 out of 256) where particles are wholly imbalanced are a significant slow-down factor for the overall simulation.

However, even with these caveats, the overall effect of load-balancing for this fixed-size problem is a speed-up factor (versus no load-balancing) of roughly 2x on moderate numbers (16-256) of processors. It is worth noting that this is a problem with very fast-moving particles (velocities of 4 to 8 timesteps per

	Balance OFF			Balance ON			Uniform Problem		
Procs	CPU	Eff	Imbal	CPU	Eff	Imbal	CPU	Eff	Imbal
1	7128	100.0	1.0	7211	98.8	1.0	7486	100.0	1.0
2	4545	78.4	1.29	3962	90.0	1.10	3783	98.9	1.0
4	2757	64.6	1.58	2056	86.7	1.13	1890	99.0	1.0
8	1651	54.0	1.88	1176	75.8	1.28	972.8	96.2	1.0
16	1023	43.5	2.33	626.5	71.1	1.34	483.7	96.7	1.0
32	603.5	36.9	2.79	337.6	66.0	1.42	240.8	97.2	1.0
64	353.4	31.5	3.25	186.3	59.8	1.51	124.5	94.0	1.0
128	202.0	27.6	3.67	110.3	50.5	1.71	64.1	91.2	1.0
256	112.7	24.7	4.08	63.7	43.7	1.84	34.5	84.8	1.0
512	66.6	20.9	4.50	44.3	31.4	2.35	19.1	76.6	1.0
1024	34.4	20.2	4.58	32.0	21.8	2.55	11.8	62.0	1.0

Table VII: *Performance (CPU-time, parallel-efficiency, load-imbalance) for a fixed-size particle simulation with a dynamic imbalance in particle load on the Intel Tflops. Results with the load-balancer turned OFF and ON are shown as is a problem of the same size with uniform particle load.*

cell-crossing are more typical of QS problems); slower dynamic variation in the particle loads would cause the load-balancer to be triggered less often and result in better parallel efficiencies.

Finally, we benchmark a scaled-size problem requiring dynamic load-balancing. As before, each processor owns a 30x30x30 block of grid cells. The global domain is pre-loaded with 3 sets of particles, each of which is a slab that fills 1/10 the domain at a 10x higher density than in the case of the uniform-load problems. As in the fixed-size dynamic problem, particles in the 3 sets are given initial velocities so that they move in the direction of the thin dimension of the slab. For example, on 512 processors the global domain is a 240x240x240 grid and the first set of particles (density of 40 particles/cell) fills a 24x240x240 slab and moves in the $+x$ direction (thin dimension of the slab). This means that 1/1000 of the domain is a high-density region of particles (120/cell) that moves over time from the lower-left front corner to the upper-right back corner of the box. As before, this occurs quickly as all particles cross a cell width in two timesteps.

Timing results for a 200-timestep run of this problem are shown in Table VIII. As in the previous table, the results with the load-balancer turned ON show a speed-up of roughly 2x on 16 or more processors versus the non-load-balanced runs. In the 128-processor simulation, the balancer was invoked 26 times so that the average lifetime of a set of created window blocks was only 7 timesteps. As previously discussed, the fast-varying particle load in this test problem limits the effective parallel efficiency due to the relatively high fraction of timesteps (1 out of 8 in this case) that the code spends in an unbalanced state. The largest problem in the table was a run with 27 million grid cells and 324 million particles. Despite the highly dynamic nature of the load variation, an overall speed-up of 373 (out of 1024) was still obtained.

Finally, in Figure 12, the parallel efficiency results from the load-balancing timings in Tables V-VIII are plotted. The upper plot shows the results for the fixed-size problems; the lower plot is for the scaled-size simulations. In both plots, square data points are results for running with the load-balancer turned OFF; triangular data points are with the balancer ON. Similarly, the shaded symbols (squares or triangles) are for problems with static spatial imbalance in particle load; open symbols are for simulations where the "hot

	Balance OFF			Balance ON			Uniform Problem		
Procs	CPU	Eff	Imbal	CPU	Eff	Imbal	CPU	Eff	Imbal
1	576.7	100.0	1.0	573.1	100.2	1.0	604.6	100.0	1.0
2	744.6	77.4	1.31	625.6	92.2	1.07	608.8	99.3	1.0
4	911.4	63.3	1.63	725.6	79.5	1.25	612.7	98.7	1.0
8	1089	52.9	1.94	757.1	76.2	1.27	635.4	95.2	1.0
16	1390	41.5	2.51	873.8	66.0	1.43	637.0	94.9	1.0
32	1696	34.0	3.07	957.3	60.2	1.54	639.0	94.6	1.0
64	2008	28.7	3.64	1004	57.4	1.58	646.8	93.4	1.0
128	2530	22.8	4.61	1115	51.7	1.74	649.0	93.2	1.0
256	3045	18.9	5.57	1281	45.0	1.96	650.3	93.0	1.0
512	3572	16.1	6.54	1412	40.8	2.12	655.3	92.3	1.0
1024	4089	14.1	7.53	1584	36.4	2.34	655.7	92.2	1.0

Table VIII: *Performance (CPU-time, parallel-efficiency, load-imbalance) for scaled-size particle simulations with a dynamic imbalance in particle load. Results with the load-balancer turned OFF and ON are shown along with results for same-size simulations with uniform particle load.*

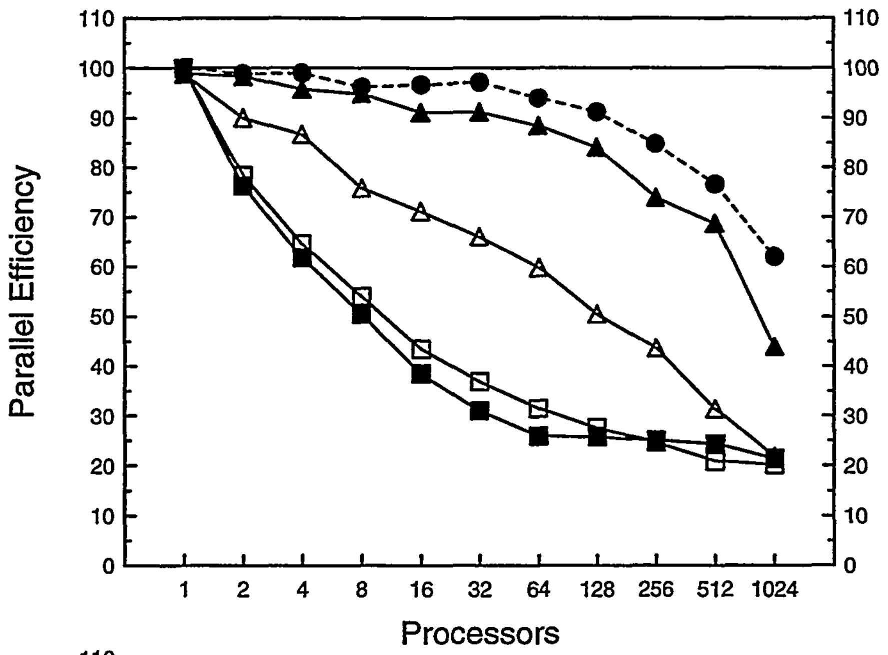
spots" of imbalance moved rapidly across the simulation domain.

In both plots, the circular data points (dotted lines) are efficiencies for perfectly-balanced (uniform load) problems with the same total number of grid cell and particle counts. These circular data are effectively the highest efficiency that could be achieved on these problems, if the balancer were working perfectly. As the plots indicate, for statically-imbalanced problems, the balancer comes close to achieving maximum performance. For problems requiring dynamic load-balancing, the balancer is not as effective, but still typically offers a marked improvement over running with no re-balancing of particle load.

8 Conclusions

In this report we have described the algorithms and performance of a new parallel version of the QUICKSILVER (QS) electromagnetic PIC code. The new code retains most of the original features that have made serial QS an attractive and powerful simulation tool for the electromagnetics and plasma physics group here at Sandia. 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 efficient and 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 evenly across processors. As highlighted in the previous section, the result is a code that can effectively run very large PIC simulations on thousands of processors with a billion or more grid cells and particles.

As discussed at the end of Section 5 there is still some work that needs to be done on parallel QS. The specification of transmission lines needs to be made more flexible. A robust restart-file capability needs to be added, which is a challenge for very large simulations because of the size of the data sets involved. There are also load-balancing enhancements that could be considered. One drawback of the current scheme is that when load balancing is turned ON, particles are pushed for one timestep in an unbalanced state, as



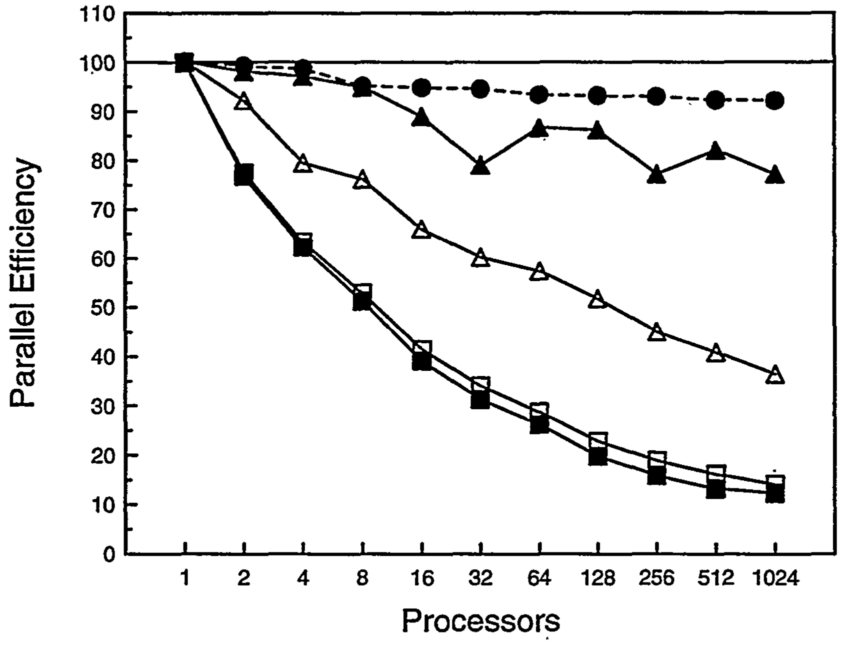


Figure 12: *The upper plot is parallel efficiencies for the fixed-size problems simulated in Tables V and VII. The lower plot is for the scaled-size problems of Tables VI and VIII. The lower curves (squares) are for load-balancing turned OFF; the intermediate results (triangles) are for load-balancing turned ON. Shaded symbols are for problems with static imbalance in particle load; open symbols are for simulations with dynamic imbalance. The dotted lines (circles) are the efficiencies of the corresponding uniform-load problems.*

they migrate to new processors. For simulations with rapidly changing particle densities, this means that 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 discussed ideas for migrating particles "instantly" to new processors to avoid this one-step delay, but it involves other trade-offs whose effects are hard to predict.

Finally, the electromagnetics and plasma physics group at Sandia is actively investigating what the next steps are in the evolution of plasma simulation capability. The "holy grail" of PIC techniques for Sandia (and others) would be to have a unified code that allows for hybrid structured/unstructured grids, is easily maintainable and extensible, and has the potential to run in tandem with other simulation modules (e.g. radiation transport) to model multi-physics effects. And, of course, this must all run in parallel on large-scale machines, as well as on high-end workstations, and have a variety of user-friendly pre- and post-processing tools. It remains to be seen whether such an ambitious goal will lead to a re-writing of (structured grid) QS and (unstructured grid) VOLMAX in an object-oriented style, or an encoding of their basic algorithms in a high-level framework such as SIERRA or ALEGRA, or some other ultimate solution.

References

[1] J. P. Berenson. *J. Comp. Phys.*, 114:185, 1994.

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

[3] R. S. Coats, M. L. Kiefer, T. D. Pointon, and D. B. Seidel. QuickSilver user's guide. Available from authors, May, 1997.

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

[5] 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.

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

[7] F. Kazeminezhad, S. Zalesak, and D. Spicer. A particle model on an unstructured mesh. *Comp. Phys. Comm.*, 90:267–292, 1995.

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

[9] 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.

[10] 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.

[11] NASA. http://www.beowulf.org.

[12] S. J. Plimpton, D. B. Seidel, M. F. Pasik, and G. R. Montry. Load-balancing a parallel electromagnetic PIC code. *to be submitted to Comp. Phys. Comm.*, 2000.

[13] D. J. Riley and C. D. Turner. VOLMAX: A solid-model-based, transient volumetric maxwell solver using hybrid grids. *IEEE Antennas and Propagation Mag.*, 39:20–33, 1997.

[14] D. B. Seidel, R. S. Coats, M. L. Kiefer, T. D. Pointon, and L. P. Mix. PFF - a compact machine-independent file format for simulation data, 1990. presented at 9th Biennial Cube Symposium, Santa Fe, NM.

[15] 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.

[16] D. B. Seidel, M. F. Pasik, M. L. Kiefer, D. J. Riley, and C. D. Turner. Advanced 3D electromagnetic and particle-in-cell modeling on structured/unstructured hybrid grids. Technical Report SAND97-3190, Sandia National Laboratories, Albuquerque, NM, January, 1998.

[17] E. Sonnendrucker, J. J. Ambrosiano, and S. T. Brandon. A finite-element formulation of the darwin PIC model for use on unstructured grids. *J. Comp. Phys.*, 110:310–319, 1994.

[18] J. Sturtevant. http://sasg829.sandia.gov/pds/index.htm.

[19] J. Sturtevant, M. Christon, P. Heerman, and P. Chen. PDS/PIO: Lightweight libraries for collective parallel I/O. In *Proc. SC98*. IEEE Computer Society Press, 1998.

[20] D. W. Walker. The parallel implementation of a large-scale particle-in-cell plasma simulation code. *Concurrency*, 2:257–288, 1990.

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

[22] K. S. Yee. *IEEE Transactions on Antennae Propagation*, 14:2155–2163, 1966.

A Appendix

This appendix provides a concise listing of all new and modified QUICKSILVER input commands supported by the new parallel version of QS and the MERCURY pre-processor. They are listed in a format similar to the QS Users Guide [3], so that these pages can be simply be added to the existing Users Guide if desired.

A.1 Modified QUICKSILVER Commands

The following QUICKSILVER (QS) commands have been modified:

PERIODIC

Define a periodic boundary condition. Boundary orientation is specified by its normal direction. The locations of the two periodic planes are specified by their normal coordinate values.

format:
PERIODIC *ijk x1 x2*
where:
ijk - coordinate direction of periodic plane normals: I, J, or K
x1 x2 - normal coordinate ordinate value for the two periodic planes

SNAPSHOT

Save field or particle spatial data at specific times for postprocessing.

MAX_PARTICLE
Specify the default maximum number of particles to write for particle snapshots. If the number of processors equals one and both parameters are provided, each is set to the maximum of the two provided values.

format:
SNAPSHOT MAX_PARTICLE *[max max_global]*
where:
max - default number of particles for local particle snapshot storage (default is 3000)
max_global - default maximum number of particles in particle snapshots (default is *max*)

PARTICLE
Enter parameters to write particle spatial, momentum, and charge data for postprocessing. The SNAPSHOT MAX_PARTICLE command controls the default maximum number of particles to save. The particle fraction is adjusted if the specified fraction exceeds the maximum number of particles. If no volume is entered, the simulation limits are used. If the number of processors equals one and both *max* and *max_global* are provided, each is set to the maximum of the two provided values.

format:
SNAPSHOT PARTICLE *'title' ktbeg ktend ktinc species[-data_type] fraction [max [max_global]] & [xibeg xjbeg xkbeg xiend xjend xkend]*
where:
'title' - title for particle snapshot (up to 32 characters)
ktbeg - beginning timestep number for particle snapshot
ktend - ending timestep number for particle snapshot
ktinc - timestep increment for particle snapshot
species - name of species to be saved in snapshot (ALL for all)

data_type - flag that particle momentum, charge, or both are to be saved in addition to location; valid types are p, q, and pq (default is location only)
fraction - fraction of species particles to be saved in snapshot
max - number of particles for local particle snapshot storage (default given by SNAPSHOT MAX_PARTICLE)
max_global - maximum number of particles in particle snapshots (default is *max*)
xibeg xjbeg xkbeg - beginning (i,j,k) ordinate of volume
xiend xjend xkend - ending (i,j,k) ordinate of volume

CUSTOM KPWRITE
Write saved killed particles out to the particle PFF file. (See CUSTOM KPSAVE command). If the number of processors equals one and both *pbufsize* and *lbufsize* are provided, each is set to the maximum of the two provided values.

format:
CUSTOM KPWRITE *spe[.tagname] pbufsize [lbufsize] pfflbl*
where:
spe[.tagname] - Species/Tagname label (see CUSTOM KPSTAG command)
pbufsize - maximum number of particles in a KPS dataset
lbufsize - number of particles for local KPWRITE storage (default is *pbufsize*)
pfflbl - label for KPS dataset

A.2 New QUICKSILVER Commands

The following new QUICKSILVER (QS) commands have been added:

CUSTOM PROCESSORS

Direct MERCURY to prepare a QS input deck for use with multiple processors.

format:
CUSTOM PROCESSORS *P [assign]*
where:
P - number of processors to be used
assign - specifies how the blocks are to be assigned to processors: sorted (default), clumped, or strided

CUSTOM DECOMPOSE

Optional commands to guide MERCURY in subdividing the blocks of the problem domain into new blocks.

format:
CUSTOM DECOMPOSE *n m* or
CUSTOM DECOMPOSE *n mx my mz*
where:
n - user-block number
m - subdivide block *n* into *m* sub-blocks
mx my mz - subdivide block *n* with planar cuts along each of the 3 dimensions into *mx* by *my* by *mz* sub-blocks

PARALLEL

When parallel QS is running on just one processor, tell it whether to run in original serial mode or use its new parallel algorithms.

format:
PARALLEL *n*
where:
n - 0 for serial (the default) or 1 for parallel

CUSTOM SCREEN

Instructs QS to display run statistics for parallel performance to stdout.

format:
CUSTOM SCREEN *n*
where:
n - display statistics every *n* timesteps (default is 0, indicating never)

CUSTOM EBJCHECK

This command invokes a consistency check for $\vec{E}$, $\vec{B}$, and $\vec{J}$ field components that lie on the shared surfaces between blocks.

format:
CUSTOM EBJCHECK *n m*
where:
n - if $n > 0$, the check is performed only on $\vec{E}$ and $\vec{B}$ components, if $n < 0$, the $\vec{J}$ components are also included. The check is performed every $|n|$ timesteps. (default is 0, indicating never)
m - $m = 0$: only a total count of errors is printed; $m = 1$: more detailed information is also displayed

CUSTOM LOADBALANCE

This command controls how dynamic load-balancing is performed during a parallel QS run.

format:
CUSTOM LOADBALANCE *tol1 tol2*
where:
tol1 - value (>= 1.0) of imbalance required to trigger a re-balance operation (perfect balance = 1.0)
tol2 - value (>= 1.0) of imbalance that the balancing algorithm attempts to achieve (perfect balance = 1.0)

CUSTOM USE_PDS

Force QS to write field and particle (including KPWRITE) snapshot data in PDS format when there is only a single processor. By default, PFF format will be used if there is only one processor.

CUSTOM FIXEDRANF

Force QS to use a fixed value for all random numbers involved in particle creation and advancement algorithms. Note that this does not modify the use of random numbers for limiting particle fractions in snapshot diagnostics.

format:
  CUSTOM FIXEDRANF *value*
where:
  value - value (between 0.0 and 1.0) to be used for random number calls

A.3 MERCURY-Generated QUICKSILVER Commands

The following new and modified QUICKSILVER (QS) commands are automatically generated by MERCURY.

UBLOCK

Provides a description of the original block for the problem description provided to MERCURY. WARNING: this command should be modified only by experienced users who understand what they are doing!

format:
  UBLOCK *xibeg xjbeg xkbeg xiend xjend xkend*
where:
  xibeg xjbeg xkbeg - beginning (i,j,k) ordinate
  xiend xjend xkend - ending (i,j,k) ordinate

UGRID

Provides a description of the original grid for the problem description provided to MERCURY. The block number refers to the blocks provided via the UBLOCK command. WARNING: this command should be modified only by experienced users who understand what they are doing!

format:
  UGRID *m ijk x0 nc a [b [c]]*
where:
  m - block number where mesh region is located
  ijk - coordinate direction of region: I, J, or K
  x0 - beginning ordinate of mesh region
  nc - number of cells in mesh region
  a b c - linear, quadratic and cubic coefficient in mesh-generating function (*b* and *c* default to 0.0)

PROCESSOR

Toggles the processor number that is currently taking ownership of BLOCK commands.

format:
  PROCESSOR *n*
where:
  n - processor number (0 to nprocs-1)

BLOCK

Modified form of the block command for parallel runs. It provides additional information to locate the block on the original block structure (supplied via the UBLOCK command).

format:

BLOCK *xibeg xjbeg xkbeg xiend xjend xkend lcl2ublk ib jb kb ie je ke*
where:
xibeg xjbeg xkbeg - beginning (i,j,k) ordinate
xiend xjend xkend - ending (i,j,k) ordinate
lcl2ublk - user block which contains this block
ib jb kb - grid index in user block describing beginning position of this block
ie je ke - grid index in user block describing ending position of this block

A.4 PDS2PFF File Conversion Utility

QUICKSILVER now writes its field and particle snapshot data to PDS-formatted files when running in parallel. The *pds2pff* utility has been developed to allow the user to convert these PDS files to PFF-formatted files. The utility automatically senses the type of data in the input file (field or particle) and converts the data to the corresponding PFF dataset types.

format:
pds2pff *file*
where:
file - file name (without extension) of PDS file to be converted; *pds2pff* will convert the data in *file.pds* to PFF format and write the resulting data to the file *file.pff*.

DISTRIBUTION:

1 Gary Montry
Southwest Parallel Software
11812 Persimmon NE
Albuquerque, NM 87111

1 Robert Peterkin
AFRL/DEHE
Kirtland AFB, NM 87117-5776

1 Paul Steen
Maxwell Technologies
8888 Balboa Ave.
San Diego, CA 92123-1506

1 Rene Vezinet
Centre D'Etudes de Gramat
Departement Electromagnetisme et
Rayonnements Ionisants
46500 Gramat
FRANCE

1	MS 0188	LDRD Office, 4001
1	0321	W. J. Camp, 9200
1	0323	D. L. Cook, 1900
1	0820	P. Yarrington, 9232
1	0836	R. O. Griffith, 9117
1	0847	R. W. Leland, 9226
1	1111	S. S. Dosanjh, 9221
20	1111	S. J. Plimpton, 9221
1	1111	S. A. Hutchinson, 9221
1	1111	J. N. Shadid, 9221
1	1111	N. D. Pundit, 9223
1	1111	K. D. Devine, 9226
1	1111	B. A. Hendrickson, 9226
5	1152	M. L. Kiefer, 1642
1	1152	J. D. Kotulski, 1642
1	1152	K. O. Merewether, 1642
1	1152	D. J. Riley, 1642
1	1152	C. D. Turner, 1642
1	1166	G. J. Scrivner, 15332
5	1186	R. C. Coats, 1642
1	1186	R. W. Lemke, 1642
1	1186	L. P. Mix, 1642
5	1186	M. F. Pasik, 1642
5	1186	D. B. Seidel, 1642
1	1186	T. A. Mehlhorn, 1674
1	1186	T. D. Pointon, 1674
1	1186	S. A. Slutz, 1674
1	1186	R. A. Vesey, 1674
1	1190	J. P. Quintenz, 1600
1	1193	J. E. Maenchen, 1645
1	1194	D. H. McDaniel, 1640
1	1194	S. E. Rosenthal, 1644
1	1194	R. B. Spielman, 1644
1	9018	Central Technical Files, 8940-2
2	0899	Technical Library, 4916
1	0612	Review & Approval Desk, 4912 For DOE/OSTI