V-PIC

Speed Optimal Implementation of a Fully Relativistic Particle Push with Charge Conserving Current Accumulation on Modern Processors

Kevin J. Bowers, Ph.D.
Plasma Physics Group (X-1)
Los Alamos National Lab
March 9, 2004
Revised April 12, 2004
(Based on LA-UR-03-3359)

NNSA

Los Alamos

Overview

- Introduction
- Hardware considerations
- Implementation considerations
- Implemented algorithms
- Current status

The Big Picture

- This talk is about efficient use of computer resources.
- Algorithmic / numerical limitations of computing get all the attention. Hardware limitations are seldom considered.
  - These limits are physical and / or economic.
  - Scientific codes are often written in ways that barely touch the potential performance of mainstream computer technology.
- Ignoring hardware limitations can lead to dubious outcomes.
  - Ex: Is it worthwhile to develop a method that can take a 5x larger time step but takes 5x longer to compute that step?
- There are high performance scientific-computing-like areas where hardware limitations are addressed in great detail.
  - Signal processing, real-time data acquisition, …
  - Ex: MPEG-2 decoders have same structure as 2d PDE integrator (DVD, PVRs, Cable and Satellite Decoders, Video Cards, Digital Camcorders …)
  - If they can tap the potential, why don't we?

The Little Picture (Code Goals)

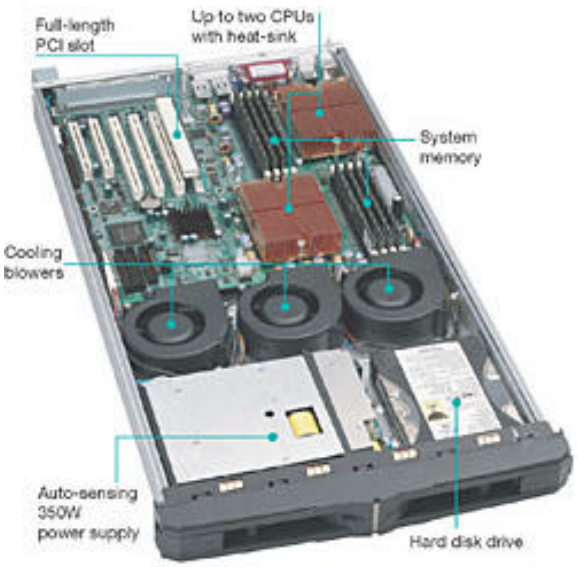
$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}, \nabla \times \vec{H} = \vec{J} + \frac{\partial \vec{D}}{\partial t}$$

$$\frac{\partial f_s}{\partial t} + \frac{\vec{p}}{m_s g_s} \cdot \nabla f_s + q_s \left(\vec{E} + \frac{\vec{p}}{m_s g_s} \times \vec{B} \right) \cdot \nabla_p f_s = \left. \frac{\partial f_s}{\partial t} \right|_c \Leftrightarrow \begin{cases} \frac{d\vec{r}_n}{dt} = \frac{\vec{p}_n}{m_n g_n} \\ \frac{d\vec{p}_n}{dt} = q_n \left(\vec{E} + \frac{\vec{p}_n}{m_n g_n} \times \vec{B} \right)_{r_n} \end{cases}$$

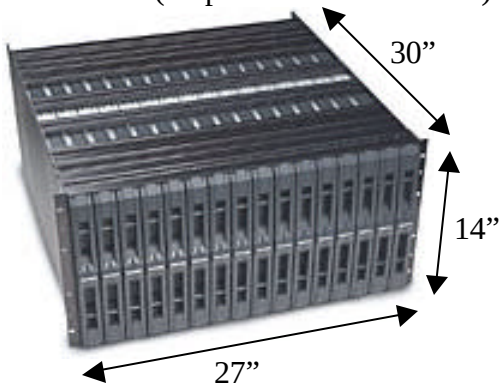
- *Science:* Develop a state-of-the art first-principles code for modeling the relativistic Maxwell-Vlasov system in three dimensions (beam- and laser-plasma interactions, astrophysics, …)
- *Implementation:* Develop techniques for ultra-high-performance PIC simulation on low cost clusters
- *Numerical:* Develop techniques applicable to PIC simulation on domain decomposed unstructured curvilinear meshes

Low Cost Clustering

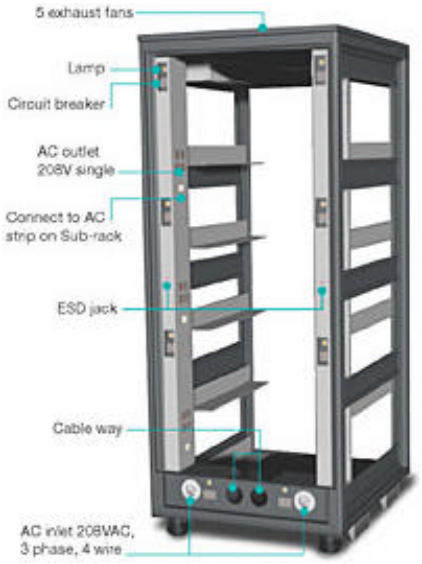
Blade (dual 64-bit Opteron w/ 16 GB)



Subrack (32 processors w/ 256GB)



Full Cabinet (5 subracks)



- Googling “opteron blade” yielded the above (from Appro.com)
- Thousands of processors is an institutional resource
  - Shared with many users (seldom can use all the nodes)
- 32 processors w/ 256GB can hide under your desk
  - Dual 1U blades from Dell ~$2.5K (~100s Gflops for <$50K)
  - 4 compute subracks + head / interconnect subrack is a file cabinet

Amdahl's Law

- How much faster will a code be if it is parallelized?

$$S \sim \frac{1}{1 - p\ (1 - 1/N)}$$

S: Speedup Factor

p: Parallel Frac

N: Num Proc

  - The parallel fraction is the relative fraction of the code that can be executed in parallel
  - Asymptotic speedup is 1/(1-p)

Heresy

- Amdahl's law is misapplied.
  - Which is better, a code that can scale to ~10 processors (p=0.90) or ~100 processors (p=0.99)?
  - What if the per node performance of the 100 processor code was 5% of theoretical and the 10 processor code was 75%?
  - ***"Amdahl's Paradox":*** Improving the efficiency of the parallel section of a code will decrease its scalability. Should you increase scalability by calling "sleep()"?
  - Usually only have access to tens to hundreds processors for finite duration of time anyway
  - Perversely, people profusely parallelize programs prematurely.
    - Node efficiency is usually a fraction of what can be attained.
    - Scalability without efficiency is not a valid performance metric.
    - Maximizing node efficiency should be done in tandem with parallelizing (if not first).
    - ***This is as much an issue of algorithm choice as of implementation.***

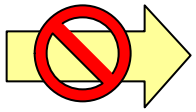
The Bottleneck: $x_n = x_n + v_n \, d_t$

- Benchmark of a 1.733GHz AMD Athlon MP with DDR333 DRAM (aka “DiscoStu.lanl.gov”):
  - “double” loads from main memory: 330M per sec
  - “double” stores to main memory: 85M per sec
- The AXPY operation, $x_n = x_n + v_n \, \delta_t$, performs two flop per every store
  - A large double precision AXPY will never run faster than ~170 Mflops (less than 10% of theoretical).
  - Codes based around these types of operations can ***never*** achieve high efficiency on modern commodity hardware.
- How do you fix it?
  - The issue is the operation’s compute-data ratio is mismatched with the computer’s capabilities.
  - Use compute-heavy (dense-matrix-like) algorithms and/or different data structures and loop organizations.

The Compiler Will Not Save You

- The golden rule of compilers: Correctness before efficiency (as well it should be).
  - However, compilers follow stricter standards than code developers (usually). For example, this optimization is illegal in IEEE-754 floating point:

```
y = b + c;
z = (a + b) + c;
```



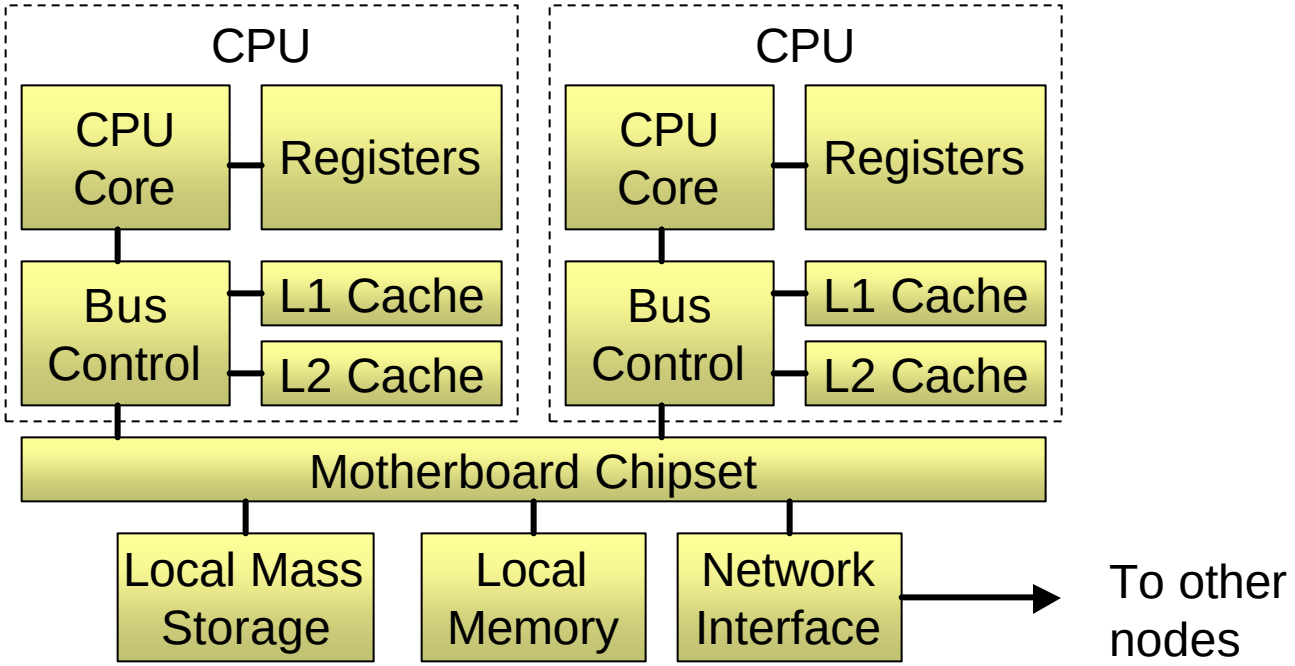
```
y = b + c;
z = a + y;
```

    - Floating point addition is not associative (try a = -1e30, b = 1e30, c=1). Conformant behavior is necessary for some classes of algorithms (ex. interval arithmetic) and compiler writers fear the raging hordes of applied mathematicians who will descend on them if the compiler does not follow standards.
- If seemingly simple optimizations are troublesome, it is unrealistic to expect the compiler to massively reorganize critical data structures and rewrite complicated loops.

Hardware Considerations

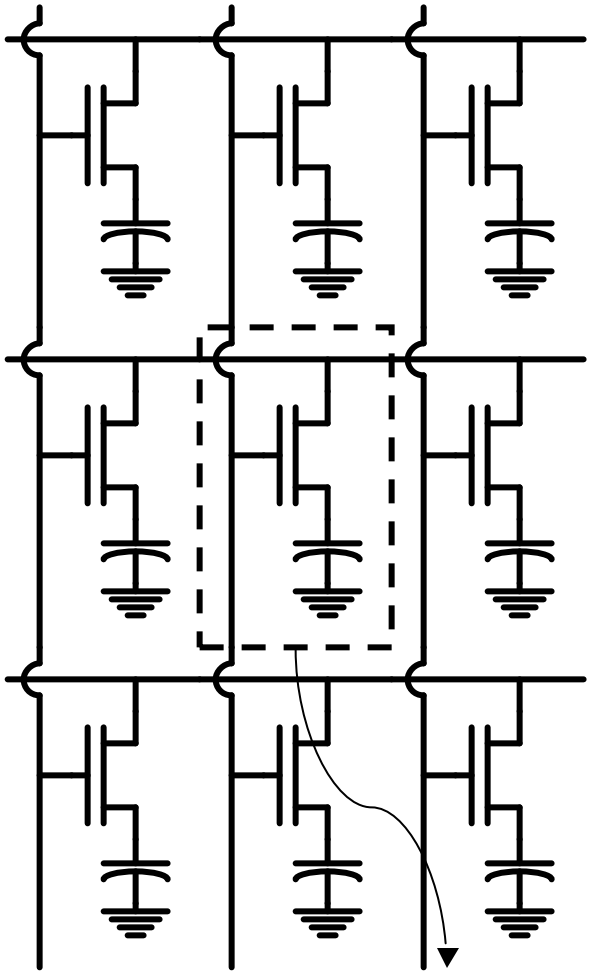
- Node organization
- Memory subsystem
- Cache considerations
- Floating point considerations

Node Organization



- The organization tells how a code should be implemented

Memory Is “Light Years” Away



~0.2 µm² (90nm process)

- Consider a 3 GHz compute node with 4 GB of ECC DRAM:
  - The characteristic time for a signal to propagate around the memory chips, considering ***only*** speed of light:

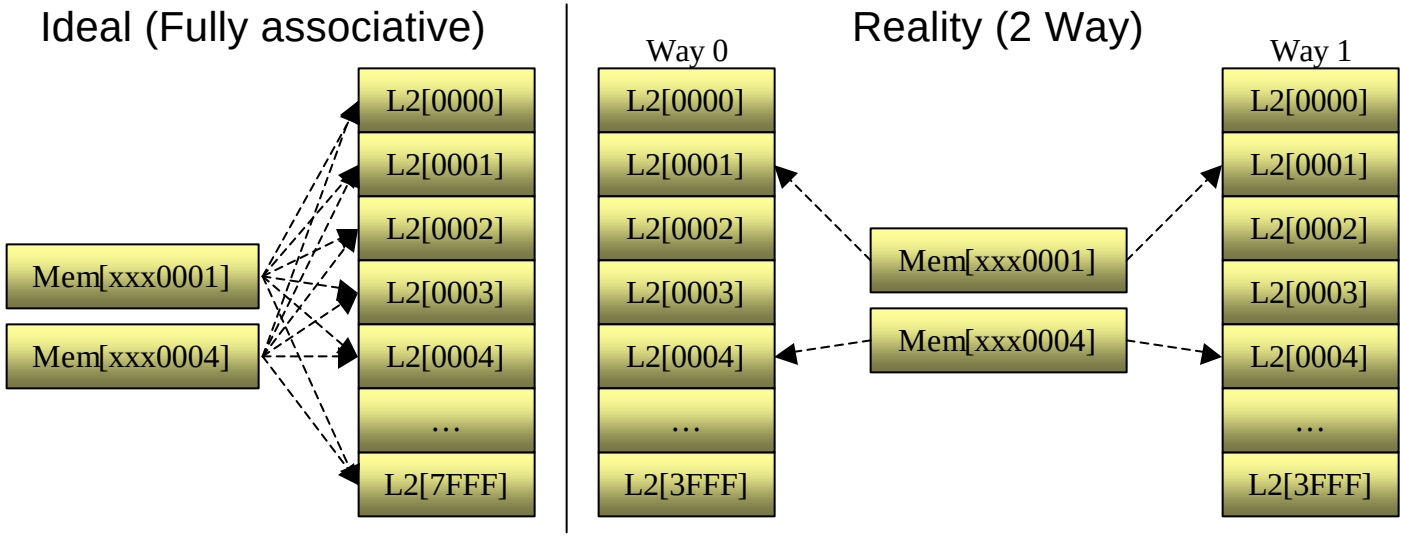
$$t_{round} \sim \frac{4n_{PCB}}{c}\sqrt{\left(\frac{0.2\,\mu m^2}{1\,cell}\right)\left(\frac{9\,cells}{1\,ECC\ byte}\right)\left(\frac{2^{30}\,bytes}{1\,GB}\right)4\,GB} \sim 3.5\,ns$$

  - This is already ~11 CPU clocks!
  - This is incredibly optimistic … other delays due to RAS/CAS, sense amplifiers, chipset translation, bus contention … can increase this to hundreds to thousands of CPU clocks to access main memory
- The speed of light is too slow!

NNSA

Los Alamos

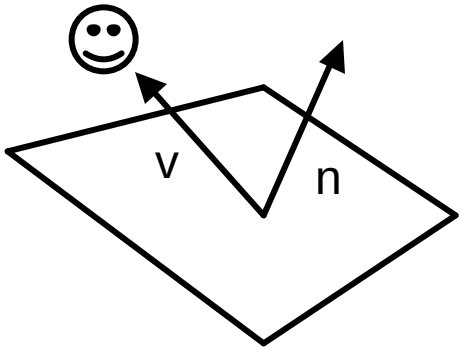
Caches are not very versatile



- Consider a 512K L2 cache w/ 16 bytes per line (32768 lines)
  - A cache line contains cached data, a tag and status bits.
  - The cache needs to tell quickly (~ CPU clock) if a request is a “hit”.
  - If a data can reside anywhere (“fully associative”) in cache, this requires comparing with all the tags … 32768 28-bit compares.
  - This is difficult … real caches are divided into “ways” (~1-16 ways).
  - A line in a “way” is restricted to certain memory addresses.

HPC is a niche market

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



$$v \cdot n = |v||n| \cos q_{vn}$$

- Modern commodity CPUs are designed for games.
- FPU intensive games use:
  - 4x4 matrix vector multiply
  - Short vector dot product
- x86 and Apple have SIMD extensions for floating point 4-vectors (SSE and AltiVec respectively).
- AXPY's (ax + y) on large vectors are ***very*** slow as the processor designers do not optimize for them.

Hardware Summary

- High performance scientific codes should:
  - Minimize memory traffic.
  - Maximize data locality.
  - Minimize floating point operations.
    - But not at the expense of increased memory traffic!
  - Reorganize loops to use more dot products and to avoid large axpy's
- Algorithm choice is important.
  - Dense-matrix-like algorithms naturally have high efficiency (this includes boundary element methods, high order spectral methods, …).
  - Other algorithms need more consideration to achieve high efficiency (PIC among them).

Implementation Considerations

- Relativistic 3d Particle Update
- Structure of arrays versus array of structures
- Implicit versus explicit cell identification
- Global versus local coordinates

Relativistic 3d Particle Update

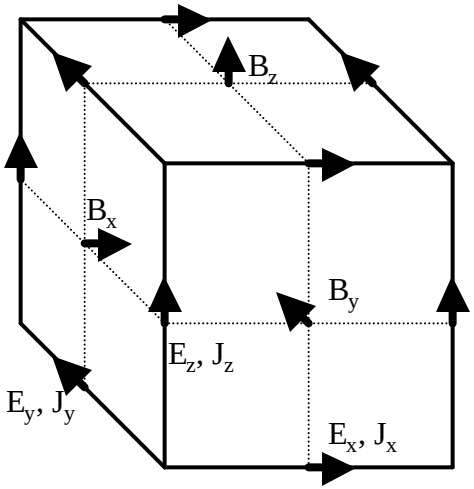
Field Interp

$$d_x = \frac{r_x|_n^t}{d_x},\ i_x = \lfloor d_x \rfloor, d_x = d_x - i_x$$

...

$$E_x|_n^t = E_x|_{i_x+0.5,i_y,i_z}^t (1-d_y)(1-d_z)$$
$$+ E_x|_{i_x+0.5,i_y+1,i_z}^t d_y (1-d_z)$$
$$+ E_x|_{i_x+0.5,i_y,i_z+1}^t (1-d_y) d_z$$
$$+ E_x|_{i_x+0.5,i_y+1,i_z+1}^t d_y d_z$$

...



Current Accumulate (Inbounds Only)

$$d_x = \frac{r_x|_n^t + r_x|_n^{t+1}}{2d_x},\ i_x = \lfloor d_x \rfloor, d_x = d_x - i_x$$

...

$$J_x|_{i_x+0.5,i_y,i_z}^{t+0.5} = J_x|_{i_x+0.5,i_y,i_z}^{t+0.5}$$
$$+ \frac{q v_x|_n^{t+0.5}}{d_x d_y d_z}[(1-d_y)(1-d_z) + \frac{d_t^2 v_y|_n^{t+0.5} v_z|_n^{t+0.5}}{12 d_y d_z}]$$

...

Half Advance E

$$u|_n^{t'} = u|_n^{t-0.5} + \frac{q d_t E|_n^t}{2mc}$$

Boris rotation

$$g = \sqrt{1 + \left(u|_n^{t'}\right)^2}$$

$a \sim$ Cyclotron freq correction

$$u|_n^{t''} = u|_n^{t'} + \left(u|_n^{t'} + u|_n^{t''}\right) \times \frac{q d_t B|_n^t}{2mg} a$$

Half Advance E (2)

$$u|_n^{t+0.5} = u|_n^{t''} + \frac{q d_t E|_n^t}{2mc}$$

Position U pdate

$$g = \sqrt{1 + \left(u|_n^{t+0.5}\right)^2}$$
$$v|_n^{t+0.5} = c u|_n^{t+0.5} / g$$
$$r|_n^{t+1} = r|_n^t + d_t v|_n^{t+0.5}$$

Choices You Didn't Know You Made

- Scientific codes often use data structures that are easy to implement quickly but limit flexibility and scalability in long run.
- Conventional PIC implementation
  - Structure of arrays layout (separate arrays for each particle property, separate arrays for each field component … i.e. FORTRAN style / vectorized style)
  - Particle position stored in the global coordinate system

Data locality is important

- Conventional particle update accesses at least 37 separate sections of memory
  - 7 particle coordinates (r_x, r_y, r_z, u_x, u_y, u_z, q)
  - 6 field arrays (e_x, e_y, e_z, b_x, b_y, b_z) and 4 accumulation arrays (ρ, j_x, j_y, j_z) in 3 non-contiguous locations (current cell, y- stride and z- stride)
- Sometimes have multiple passes through particle lists per time steps
- Randomly ordered particle lists further “thrash” the cache
- Literally, no “way” for cache to keep up

Structure of Arrays Versus Array of Structures

SoA

RX_0	RX_1	RX_2	RX_3	RX_4	…
RY_0	RY_1	RY_2	RY_3	RY_4	…
RZ_0	RZ_1	RZ_2	RZ_3	RZ_4	…
UX_0	UX_1	UX_2	UX_3	UX_4	…
UY_0	UY_1	UY_2	UY_3	UY_4	…
UZ_0	UZ_1	UZ_2	UZ_3	UZ_4	…

AoS

$RX_0\ RY_0\ RZ_0\ UX_0\ UY_0\ UZ_0$
$RX_1\ RY_1\ RZ_1\ UX_1\ UY_1\ UZ_1$
$RX_2\ RY_2\ RZ_2\ UX_2\ UY_2\ UZ_2$
$RX_3\ RY_3\ RZ_3\ UX_3\ UY_3\ UZ_3$
$RX_4\ RY_4\ RZ_4\ UX_4\ UY_4\ UZ_4$
…

- AoS has better data locality.
  - Momentum update all position and momentum coordinates simultaneously.
  - Can be adapted to 4-vector operations.

Structure of Arrays Versus Array of Structures Comparison

- Memory hierarchies require a sorted AoS particle data layout for high performance.
- Below calculations are for a minimal 2d2v electrostatic PIC simulation.

Pentium III 800/133 ATC
Dual channel RDRAM 800

FP Subsystem	M flop/s
3-cycle pipelined MAC	798

Memory Subsystem		M mop/s
Load	L1 cache	651
	L2 cache	427
	Memory	97.3
Store	L1 cache	664
	L2 cache	265
	Memory	**29.6**

Structure-of-Arrays (vectorized)

$$1.7\text{M pa/s} \approx \left(\frac{20\,\text{ldmm}}{97.3\text{M mop/s}} + \frac{10\,\text{stmm}}{29.6\text{M mop/s}} + \frac{49\,\text{flop}}{798\text{M flop/s}} \right)^{-1}$$

Array-of-Structures (thrashed)

$$2.0\text{M pa/s} \approx \left(\frac{16\,\text{ldmm}}{97.3\text{M mop/s}} + \frac{8\,\text{stmm}}{29.6\text{M mop/s}} + \frac{49\,\text{flop}}{798\text{M flop/s}} \right)^{-1}$$

Array-of-Structures (sorted)

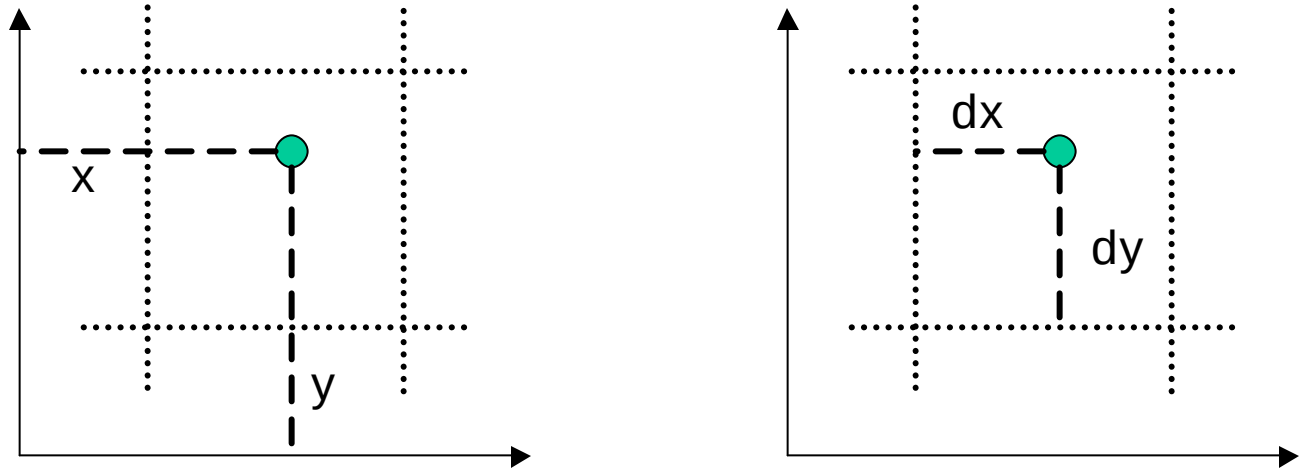
$$3.6\text{M pa/s} \approx \left(\frac{4\,\text{ldmm}}{97.3\text{M mop/s}} + \frac{12\,\text{ldl2}}{427\text{M mop/s}} + \frac{4\,\text{stmm}}{29.6\text{M mop/s}} + \frac{4\,\text{stl2}}{265\text{M mop/s}} + \frac{49\,\text{flop}}{798\text{M flop/s}} \right)^{-1}$$

NNSA Los Alamos

Implicit Versus Explicit Cell Identification

- Conventional implicit particle-centric ("i_x=floor[x/δ_x]") is problematic.
  - Makes using anything but an axis-aligned uniform mesh hard.
  - Makes using single precision unsafe on large meshes as many bits of precision are used to resolve the mesh coordinates.
  - Many compilers implement float to integer operations very poorly (can reduce overall performance over ~50%).
- Implicit cell-centric (each cell tracks particles contained therein) can be cumbersome.
  - Memory management issues, esp. for non-uniform plasmas.
- Instead, particles explicitly store the index of the cell containing them.
  - It is the only viable strategy for non-uniform, curvilinear and unstructured arbitrary mesh partitions anyway.

Global Versus Local Coordinates



- Local coordinates have many advantages.
  - No bits of precision are wasted (safe to use single precision).
  - Halves required memory bandwidth over double precision.
  - Allows optimized 4-vector operations to be used.
  - Reduces the field interpolation flop count greatly.
  - Generalizes to curvilinear meshes.
- Need to convert local coordinate system when changing cells.
  - Trivial to do on structured meshes (flip sign of appropriate coordinate)
  - No worse than other options on unstructured meshes.

Implementation Details

- The optimized inbounds handler
- The particle
- The interpolator
- The accumulator
- The guard list
- The message passing module
- The V4 class
- The V-PIC class
- Input deck processing (or lack thereof)
- Machine independent Makefile

The Optimized Inbounds Handler

- For a 3d thermal plasma simulation ($\omega_p\delta_t$~0.2, δ_x~λ_d), 89% of the time particles of the fast species will not leave their initial cell in a time step.
  - 63% of the time, all in a group of 4 do not cross cells.
  - An optimized handler is written for this common case using the V4 class to push 4 particles at a time on a processor.
  - Cell crossing particles are detected and handled separately.
  - Virtually all of the slow species (ions) stay put
- From an implementation standpoint, PIC algorithms mostly differ in how cell-crossing particles are handled.
  - This approach allows commonalities between different algorithms to be exploited (maximizes code reuse).

The Particle

```
typedef struct {
  float dx, dy, dz; // Position in cell on [-1,1]
  int i;            // Cell of particle
  float ux, uy, uz; // Normalized momentum
  float q;          // Charge
} particle_t;
```

- The particle structure is exactly 32 bytes.
  - Fits nicely into caches.
  - Can be treated a two 4-vectors.
- Variably weighted particles allowed.
  - Many algorithms require weighted particles.
  - Another (less flexible) option is to store “inv_gamma” instead but most modern processors have fast reciprocal square roots (for games, of course) which negate the savings.

The Interpolator

```
typedef struct {
  float ex, dexdy, dexdz, d2exdydz;
  float ey, deydz, deydx, d2eydzdx;
  float ez, dezdx, dezdy, d2ezdxdy;
  float cbx, dcbxdx, cby, dcbydy;
  float cbz, dcbzdz, pad0, pad1;
} interpolator_t;
```

- Redundant interpolation coefficients are stored for each cell.
  - It takes maximum advantage of the fact particle positions are stored in the cell's local coordinates; number of flop needed to interpolate the fields is greatly reduced.
  - Fields can be interpolated by accessing the cell's interpolator instead of jumping around 6 different arrays and up and down y- and z- strides within those arrays.
  - Fields can be efficiently interpolated with 4-vector operations.
  - Generalizes to other interpolation schemes, curvilinear meshes

The Accumulator

```
typedef struct {
  float jx[4], jy[4], jz[4];
} accumulator_t;
```

- Each cell has an accumulator.
  - All the contributions from a current streak through a cell can be accumulated without hopping around memory.
  - Contributions to J at a given mesh point are collected from cell accumulators sharing that mesh point.
  - Generalizes to other accumulations, curvilinear meshes

Single Pass Processing and The Guard List

- The vast majority of particles are processed in a single pass
  - Main performance bottleneck is the time it takes to load and store the particle list … want to minimize the number of passes through the particle list
  - Particle velocity advance, particle position update and charge conserving accumulate are done in same loop
- The indices and remaining displacements of particles which hit walls, cross to other processors or do anything requiring additional processing are stored on the guard list for subsequent handling
  - Eliminates the necessity of guard cells for particles.
  - Removes slow (and problem-specific) operations from the critical path and avoids thrashing instruction cache.

The Message Passing Module

- Encapsulates a subset of message passing primitives.
- Transparently maintains buffers necessary to handle overlapped synchronous communications.
- Allows different message passing libraries to be used without changing code body (MPI-1, MPI-2, PVM, vendor specific message passing extensions, …).
- No more typing MPI_COMM_WORLD!
- Status: MPI-1 version implemented.

The V4 Class

```
// Interpolate ex for the next 4 particles
swizzle( interp[ i(0) ].ex, interp[ i(1) ].ex,
         interp[ i(2) ].ex, interp[ i(3) ].ex,
         ex, dexdy, dexdz, d2exdydz );
ex = (ex + dy*dexdy) + dz*(dexdz + dy*d2exdydz);
```

- Enables the use of optimized 4-vector operations portably.
- Observed to double performance over non-v4 version on x86-SSE.
- Based on C++ operator overloading.
- A “v4float” and a “v4int” act exactly like a user expects under all valid integer and floating point operators.
- Inline class functions give optimized memory access patterns, conditional operations and selection operations (for example, a “swizzle” loads and transposes a 4x4 matrix).
- Status: Portable and SSE implementations complete with test suite.

The V-PIC Simulation Class

```
vpic_simulation sim;
if( dump_file ) sim.restart(dump_file);
else            sim.init();
while(sim.advance());
```

- The guts of “main()” are shown above.
- V-PIC uses no global variables.
- V-PIC has its own internal aligned memory management routines, random number generator (period 2^{19937}-1), input deck helpers, diagnostics routines, …
- Class allows V-PIC to be embedded in other codes.
- Class allows for multiple V-PIC simulations in a single code.

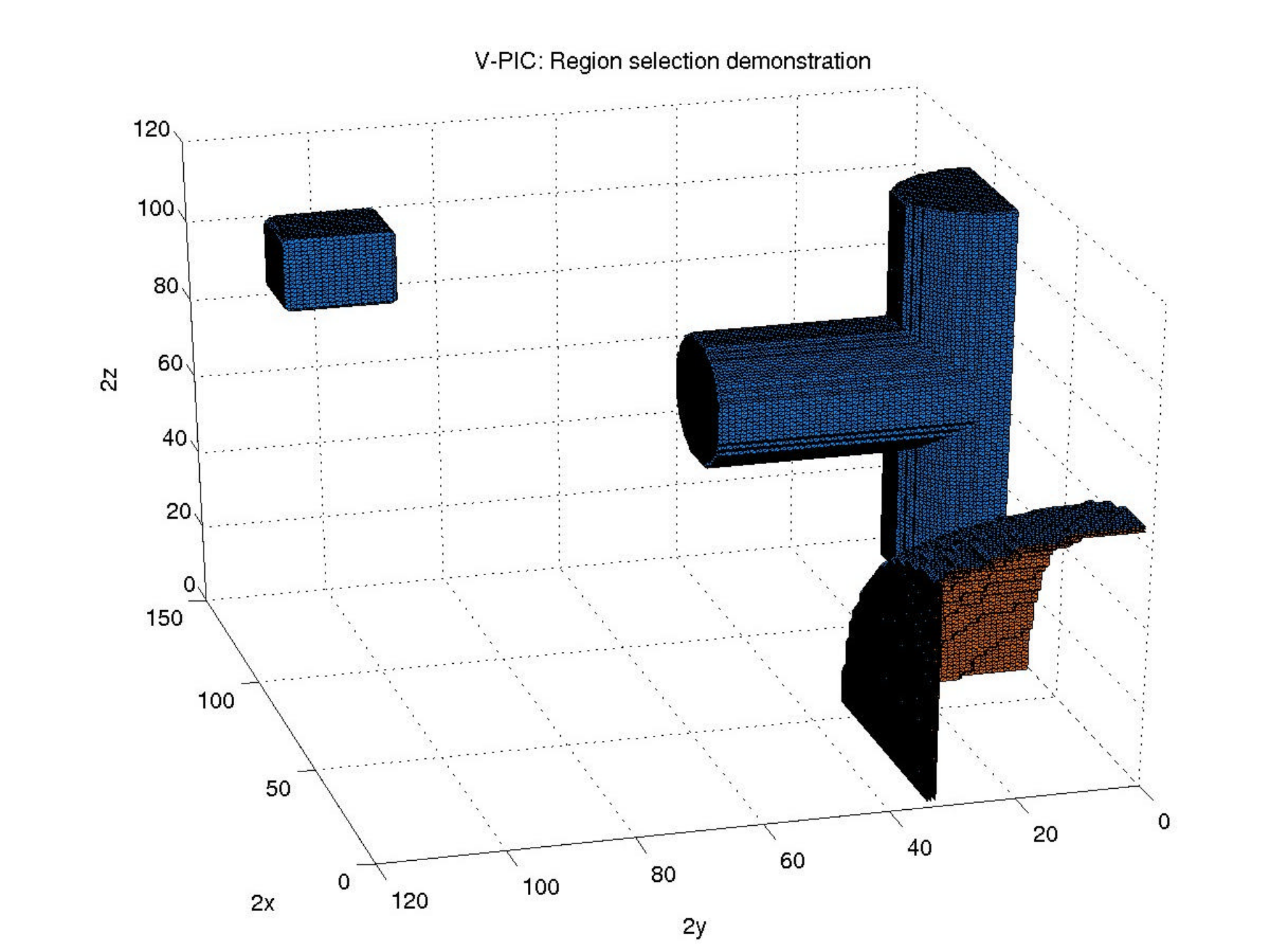
Input Deck Processing

(or Lack Thereof)

- Physicists are not computer science language experts.
  - Often end up writing obscure syntaxes with inconsistencies.
- The many woes of input deck processing and validation:
  - It is tedious to write and debug.
  - It will straight-jacket a code if not sufficiently flexible.
  - It is equivalent to a special purpose programming language.
- Use the compiler as the input deck processor.
  - Zero code development time is spent writing non-physics related code (a huge savings).
  - Syntax is already familiar to end-users and helper functions can be provided for common user initialization tasks.
  - Compiler's lexical parser is better debugged and consistent.
  - Allows very flexible input decks and diagnostics outputs.
  - Similar to "simulation driving" approaches of some of predecessor codes ("LEGO" for example).

An “Input Deck”

```
begin_globals {};
begin_initialization {
  num_step = 1;
  status_interval = 1;
  double lx = 64, ly = 64, lz = 64;
  double nx = 64, ny = 64, nz = 64;
  grid->cvac = 299792458;
  grid->eps0 = 8.854187817e-12;
  grid->damp = 0.01;
  grid->dt = 0.95*
    courant_length(lx,ly,lz,nx,ny,nz)/grid->cvac;

  // Simulation box has absorbing walls
  define_absorbing_grid( -lx/2,-ly/2, -lz/2,
                          lx/2, ly/2,  lz/2,
                          nx,   ny,    nz,
                          1,    1,     1     );

  // Space initially filled with the 1st material
  define_material( "vacuum",  1);
  define_material( "copper",  1, 1, 5.8e7 );
  define_material( "calcite", 2.2, 2.2, 2.7,
                              1,   1,   1,
                              0,   0,   0   );
  finalize_materials();
  finalize_species(); // No particles here

# define clip_sphere y<16 && sqrt(x*x+y*y+z*z)<32
# define cube x>48 && x<56 && \
              y>48 && y<56 && \
              z>48 && z<56
# define pipes (z>8 && z<56 &&                \
                sqrt((x-48)*(x-48)+y*y)<8) || \
               (y>-24 && y<24 &&              \
                sqrt((x-48)*(x-48)+(z-32)*(z-32))<8)

  set_region_material(clip_sphere,
                      "calcite",  // Interior
                      "copper" ); // Surface
  set_region_bc( clip_sphere,
                 absorb_particles,    // Interior
                 absorb_particles,    // Int surface
                 reflect_particles ); // Ext surface
  set_region_field( clip_sphere,
                    2*sin(x/lx), 0, 0,  // E
                    0, 0, 0  );         // B
  set_region_material( cube, leave_unchanged,
                             "copper" );
  set_region_bc( cube, leave_unchanged,
                       absorb_particles,
                       reflect_particles );
  set_region_material( pipes, "calcite" );
};
begin_diagnostics {
  if( step==0 ) dump_materials("materials");
  if( step==0 ) dump_grid( "grid" );
  if( step==0 ) dump_fields( "field" );
};
begin_particle_injection {};
begin_current_injection {};
begin_field_injection {};
```


V-PIC: Region selection demonstration
120
100
80
60
40
20
0
2z
150
100
50
0
2x
0
120
100
80
60
40
20
0
2y

Machine Independent Makefile

```
AddModule(algorithms,src/algorithms,cpp,
          advance_f advance_p sort_p div_clean)
AddModule(control,src/control,cpp,
          advance restart diagnostics)
MakeExecutable(bin/bench_p,src/bench,cpp,bench_p)
```

- X11 “imake” is used to convert this into a regular GNU “Makefile”.
- A custom “imake” template was developed to facilitate a modular programming style, the use of multiple programming languages and the use of source code directory hierarchies.
- Code modules are compiled to a static library to create V-PIC.
- Executables are linked against the V-PIC modules library.
- A simulation executable is created by running a script which links a user input deck against the V-PIC modules library.
- Porting V-PIC largely consists of writing a machine description file (roughly 15 required lines).

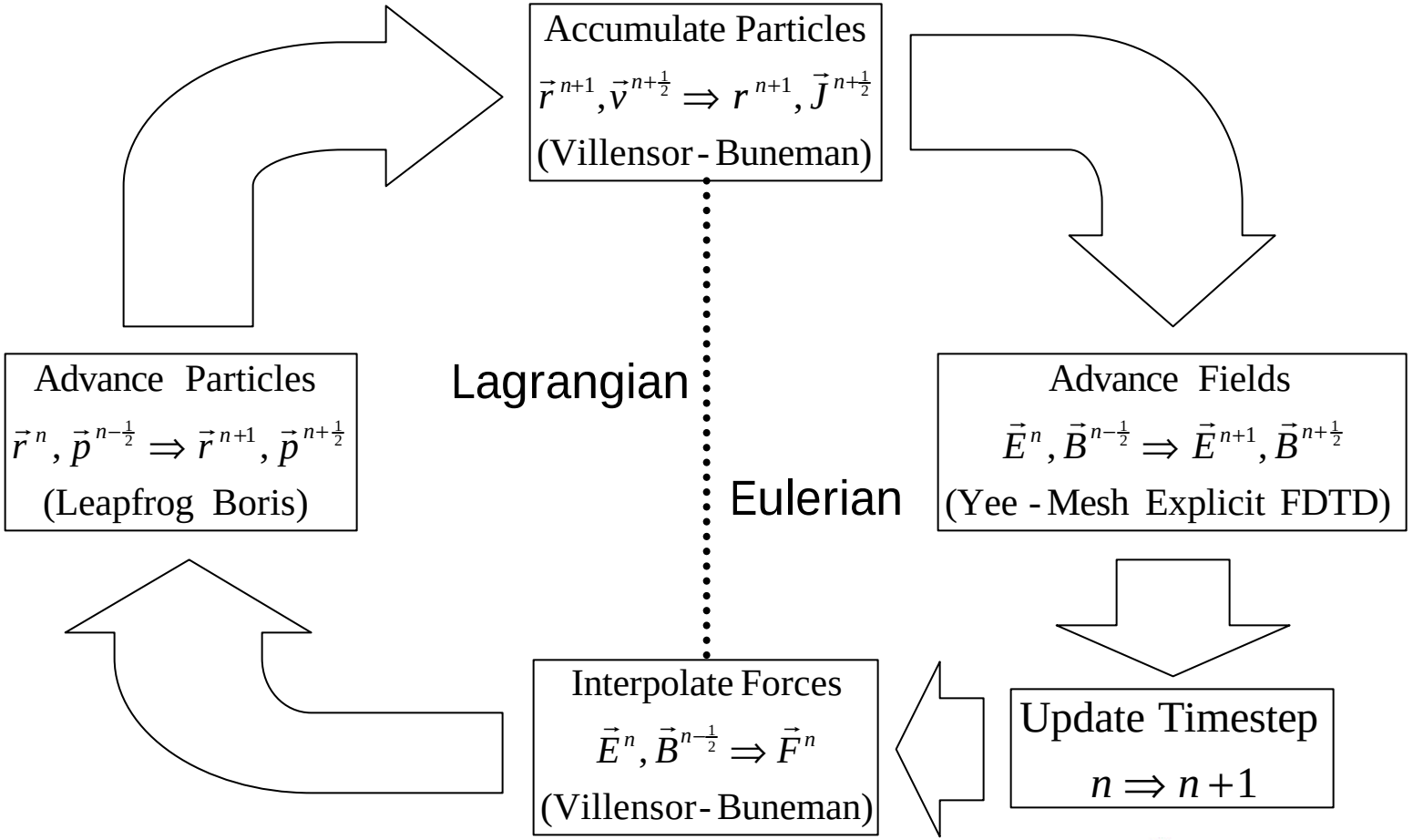
A Machine Description File

```
/* LANL ASCI Q machine description. This has been tested against modules
   "gcc_default" and "MPI_default" */

/* File extension to use for libraries and test suite executables */
#define EXTENSION qsc

/* Machine specific libraries and include directories */
LIBRARIES    +=
INCLUDE_DIRS +=
LIBRARY_DIRS +=

/* What programs to use on this machine? */
CC     = gcc
CPP    = g++
LD     = g++
AR     = ar
RANLIB = ranlib
RM     = rm -f

/* How should the programs be invoked? */
/* Note: There is no platform specific version of the V4 class for qsc.
   Leave V4VERSION undefined ... see “machine/README” */
CCFLAGS  = -Wall -pedantic -ansi -O6 -ffast-math -mcpu=ev67 -DINT32_TYPE=int
           -DRESTRICT=__restrict
CPPFLAGS = -Wall -pedantic -ansi -O6 -ffast-math -mcpu=ev67 -DINT32_TYPE=int
           -DRESTRICT=__restrict
LDFLAGS  = -Wall -pedantic -ansi -O6 -ffast-math -mcpu=ev67 -DINT32_TYPE=int
           -DRESTRICT=__restrict
ARFLAGS  = clq
```

Implemented Algorithms

- Standard PIC loop
- Energy conserving interpolation
- High-order Boris rotation
- Charge conserving accumulation
- Radiation damping
- Divergence cleaning
- Sorting

Standard PIC Loop



Energy Conserving Interpolation

- Electric field is bilinear in transverse direction and nearest grid point in parallel direction.
- Magnetic field is nearest grid point in transverse direction and linear in parallel direction.
- These are consistent with a roof-top finite element formulation of the Yee-mesh FDTD algorithm and it can be generalized to curvilinear hexahedral meshes.

High Order Boris Rotation

- Boris rotations can be characterized by how they approximate:

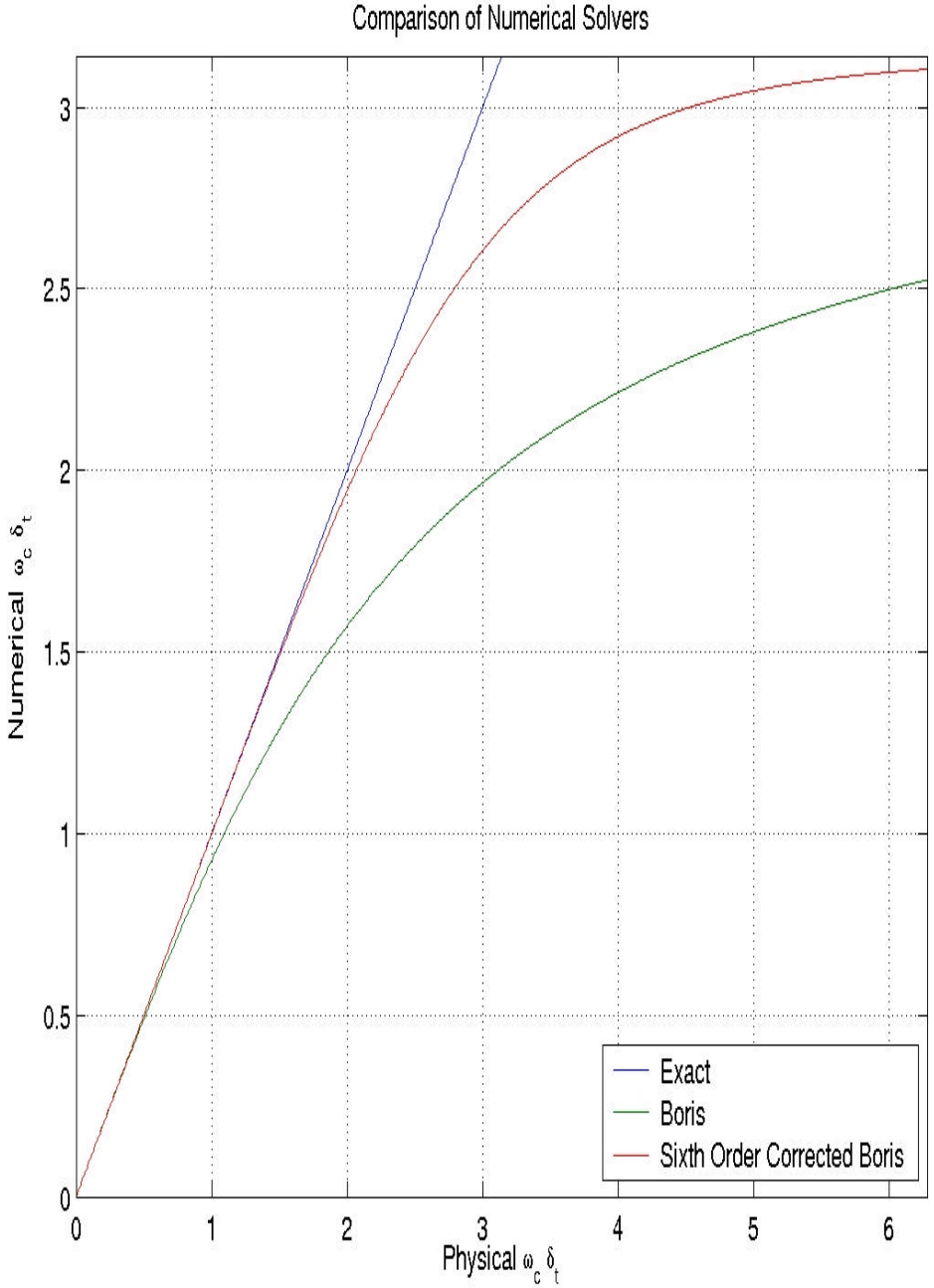
$$a = \frac{2}{w_c \Delta_t} \tan \frac{w_c \Delta_t}{2}$$

- Standard Boris:

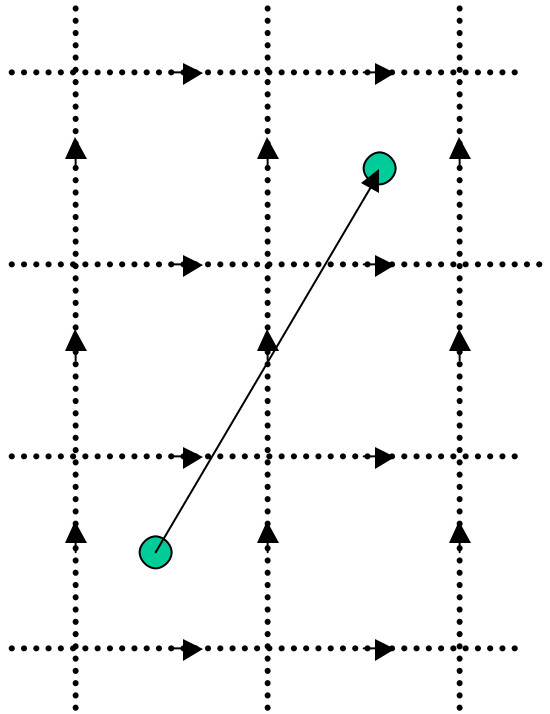
$$a \approx 1$$

- High order (6th):

$$a \approx 1 + \frac{1}{3}\left(\frac{w_c \Delta_t}{2}\right)^2 + \frac{2}{15}\left(\frac{w_c \Delta_t}{2}\right)^4$$



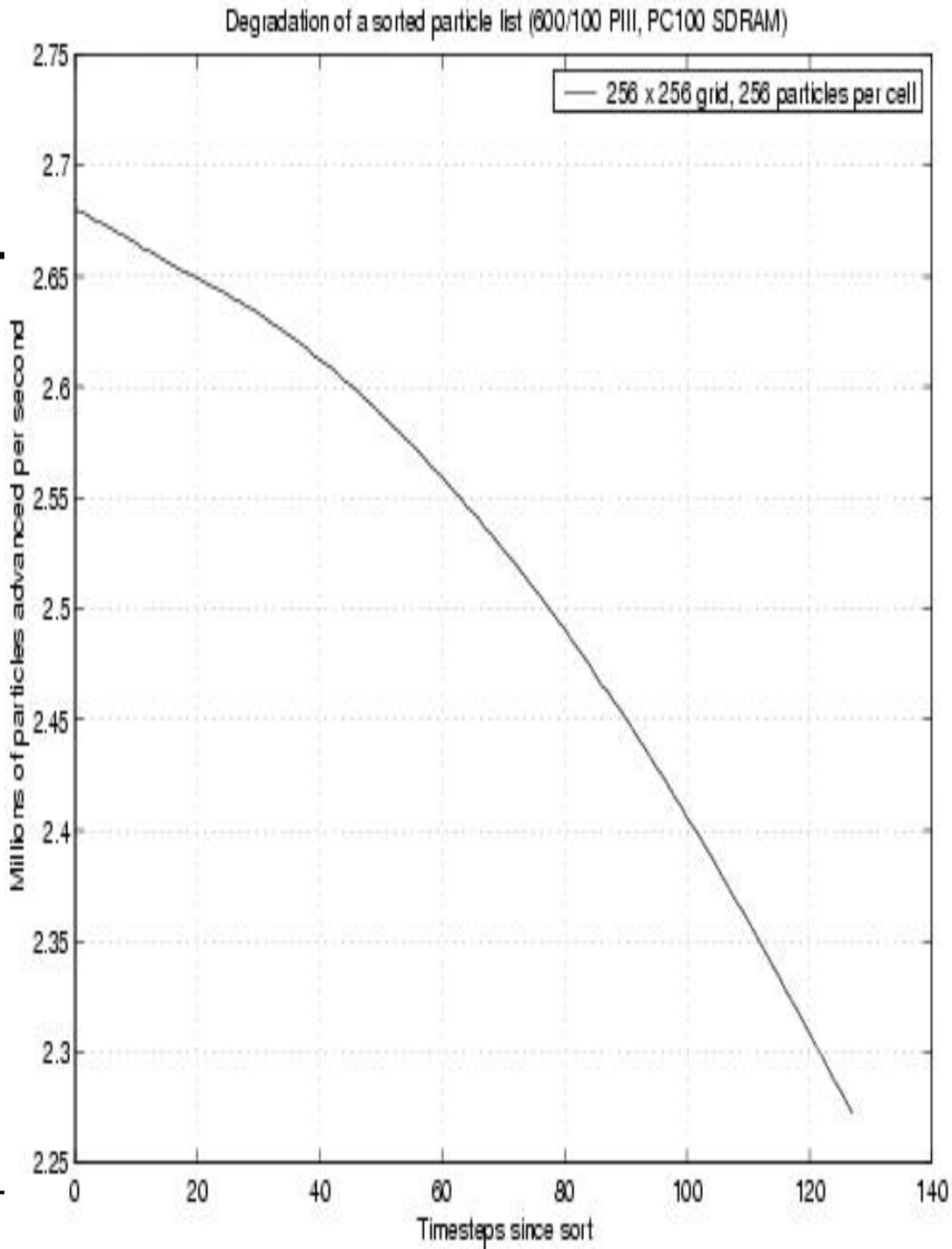
Charge Conserving Current Accumulation



- Particle trajectory is diced into current streaks, one for every cell passed through.
- Each current streak is weighted to the Yee-mesh currents of the cell's accumulator.
- Discretized continuity of charge (at grid points) is satisfied under nodal tri-linear charge accumulation (bi-linear accumulation in 2d).
- During field solve, total J is obtained from looking at all accumulators for cells which share edges.

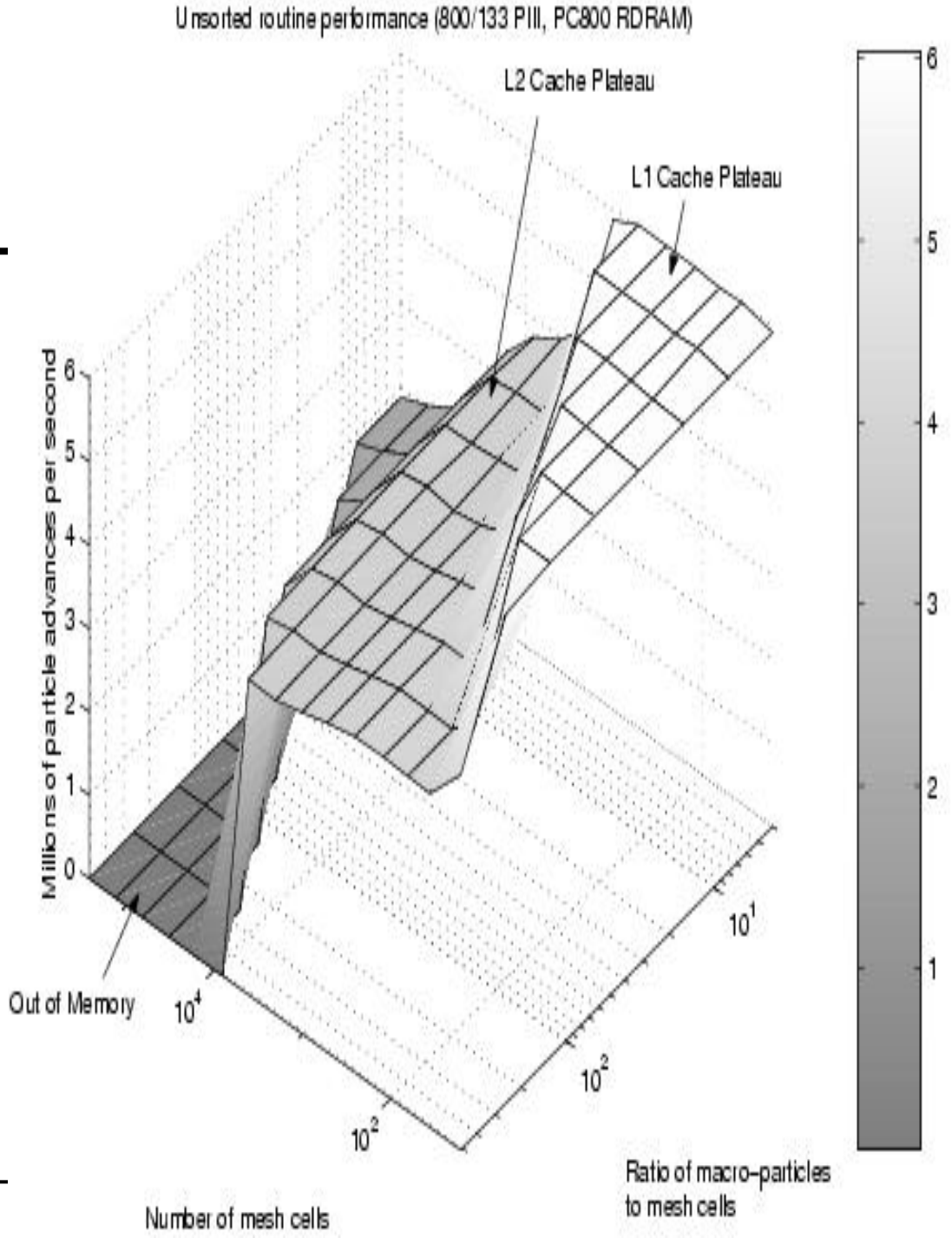
O(n) Sorting

- PIC simulation performance is hindered by cache thrashing.
- An O(n) stabilized in-place counting sort is implemented for periodic sorting.



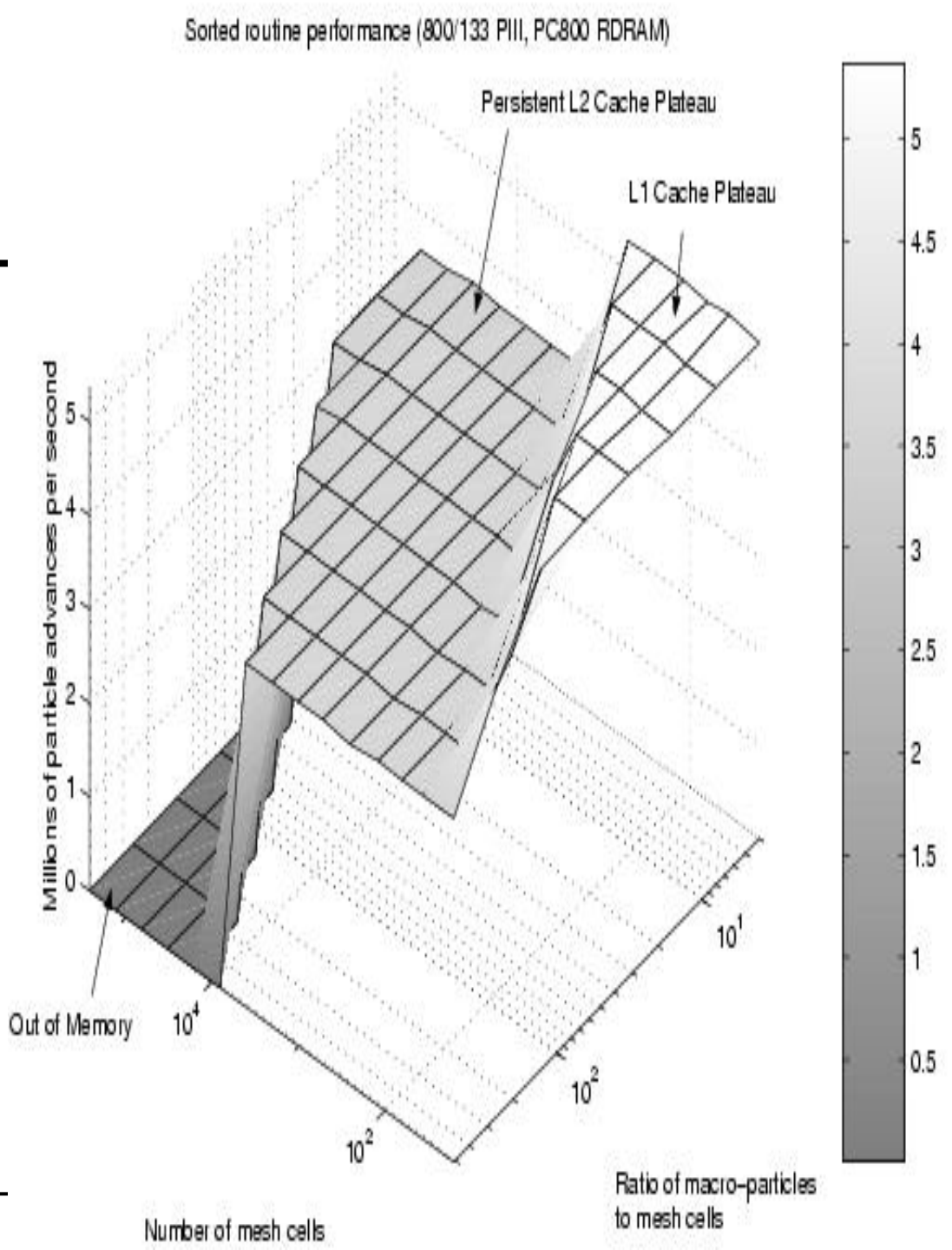
Sorting Results

- Unsorted AoS routine shows impact of memory hierarchies on performance.



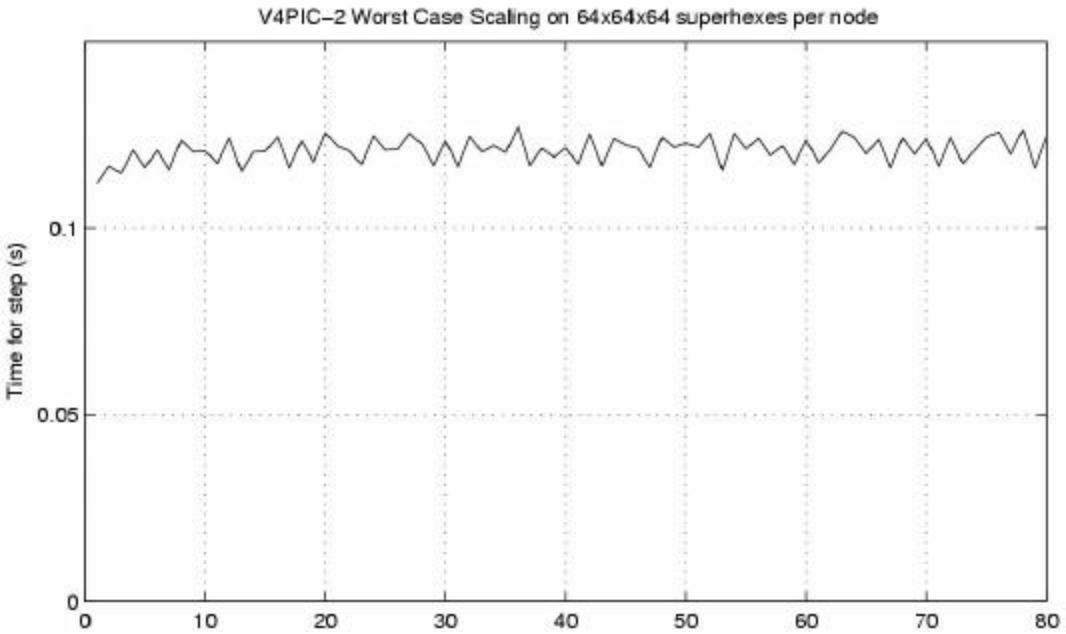
Sorting Results (cont)

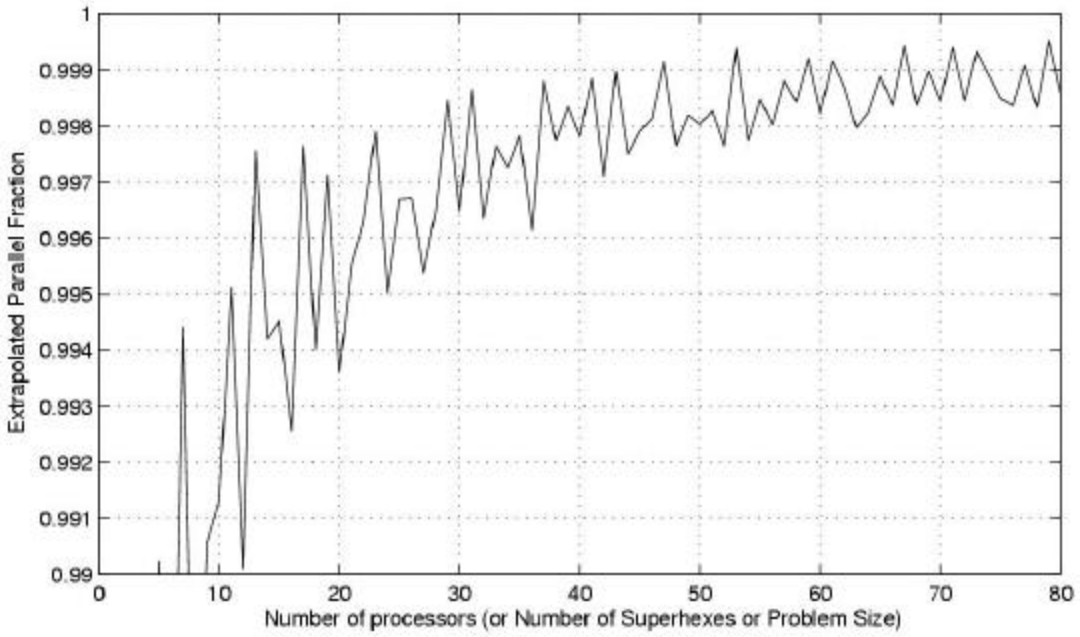
- Sorted AoS routine performance does not degrade for large problems.
- Performance is consistent with expectations.
- Over 70% performance boost seen for large problems.



Field Solver

- Yee-mesh explicit FDTD with exponentially differenced Ampere
- Superhexahedral domain decomposition (arbitrary interdomain connectivity) with fully overlapped commun.
- Support for diagonal tensor ε, μ and σ (even non-ideal thin foils)
- ~99.85% scalability measured on Gigabit Ethernet cluster

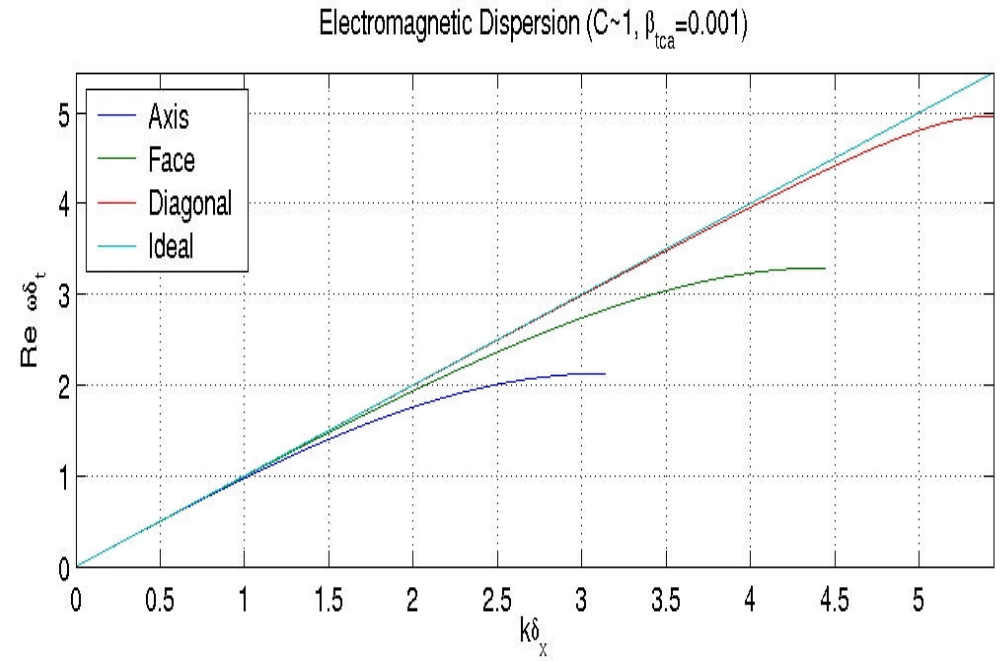


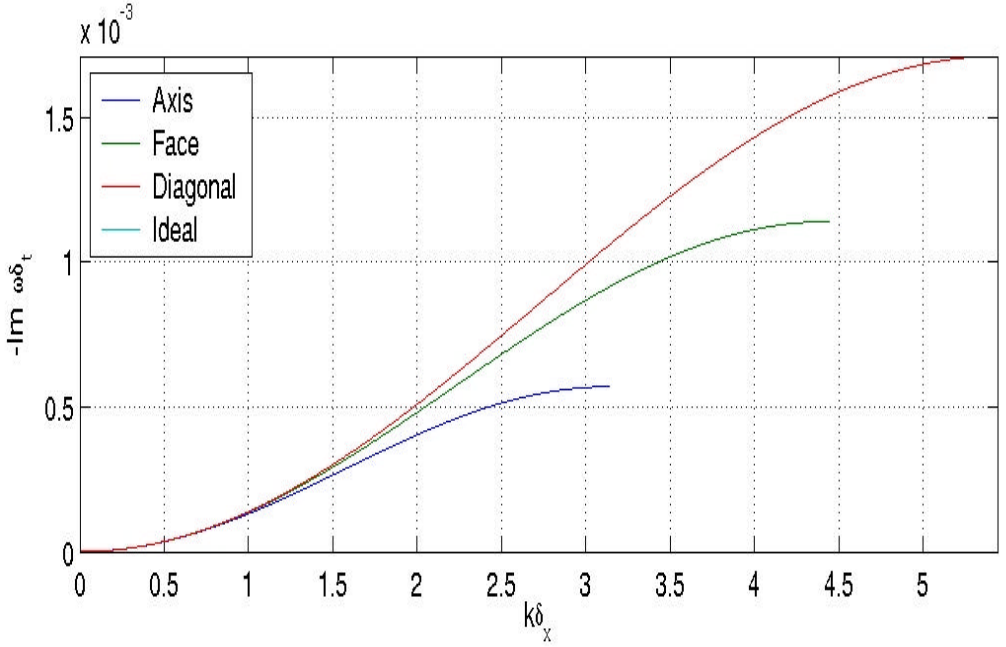


Radiation Damping

- To reduce particle noise and numerical Cherenkov radiation, a divergence free current is added to damp spurious modes.

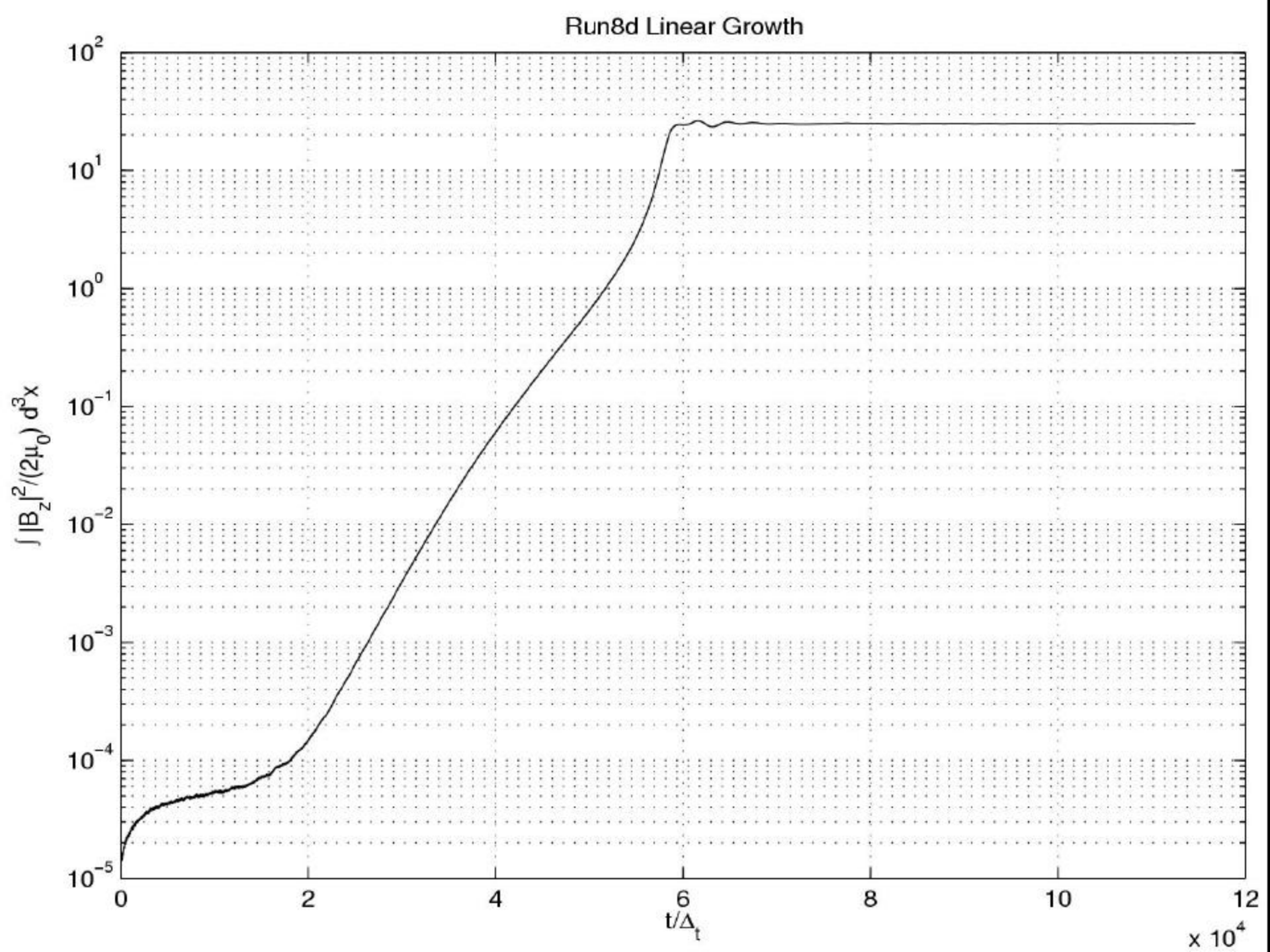
$$J_{TCA} = -b \frac{\partial}{\partial t} \nabla \times B$$

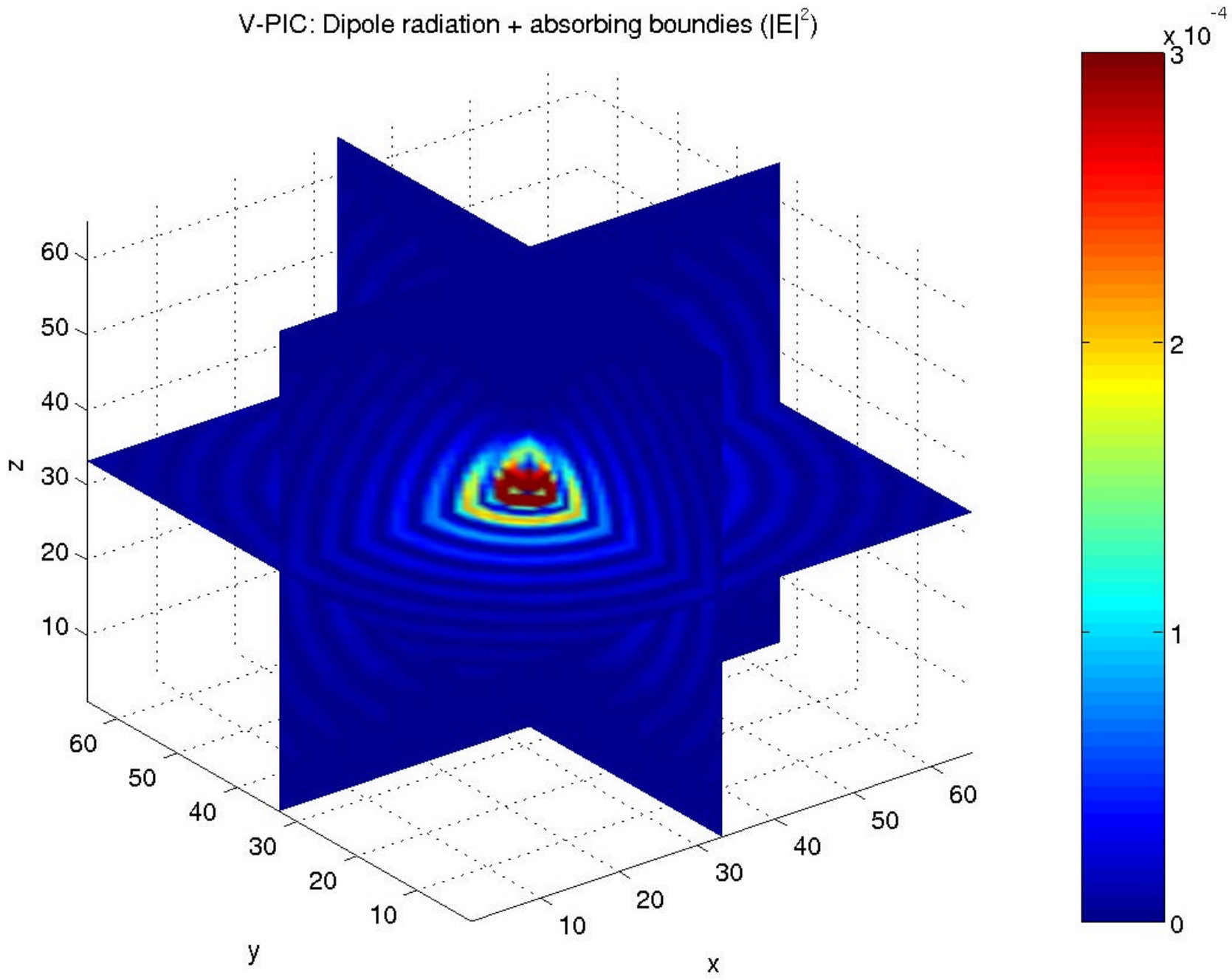


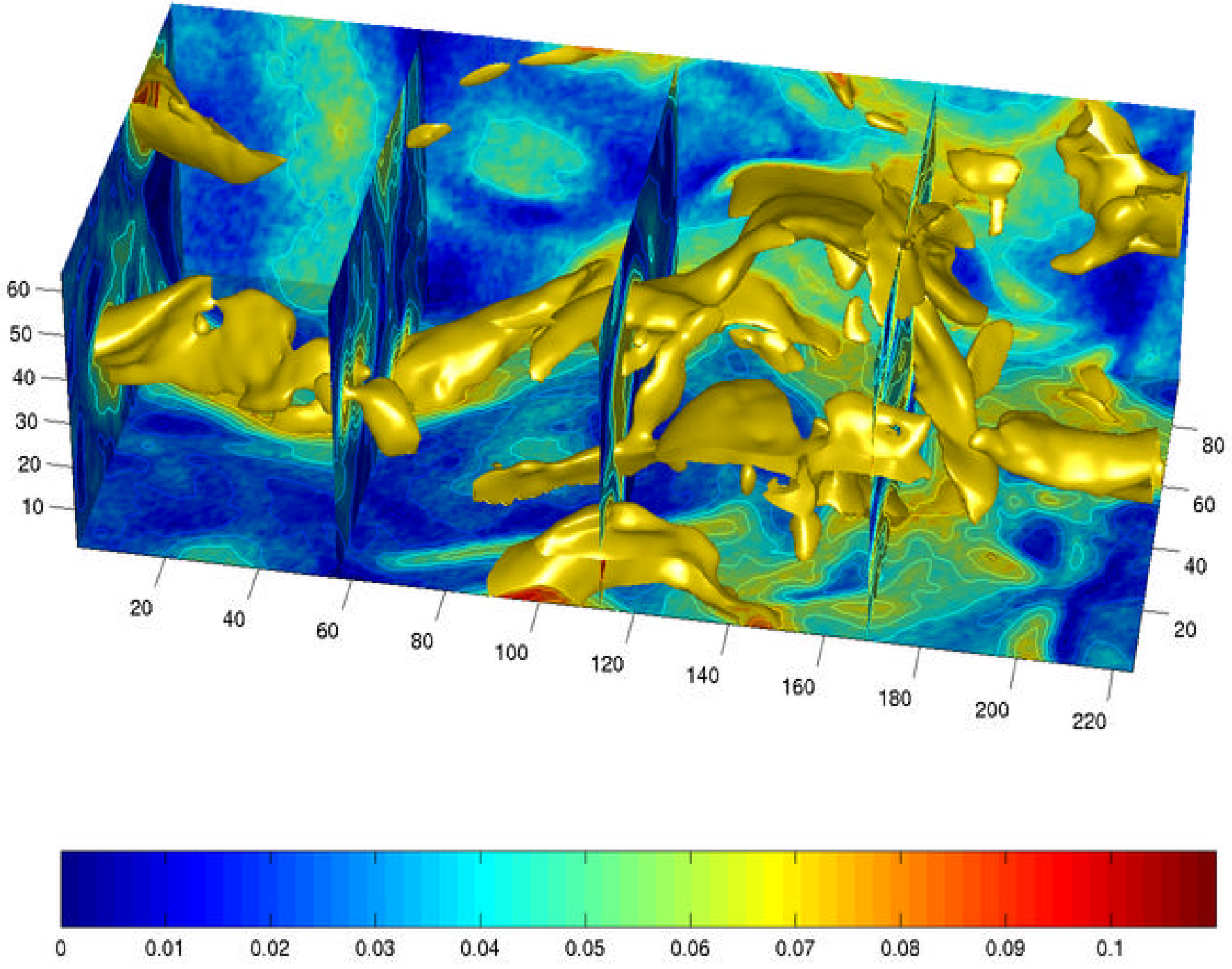


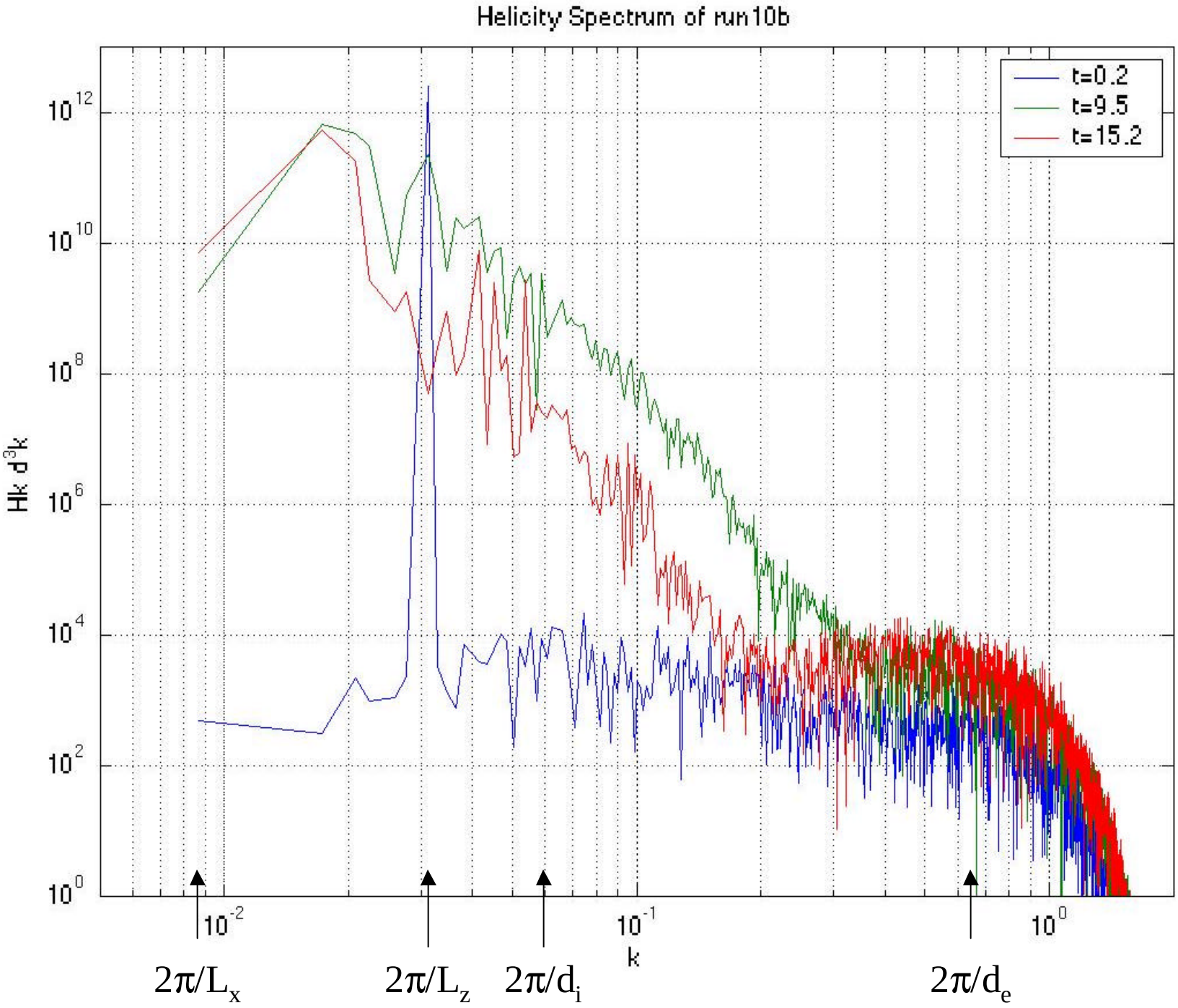
Divergence Error Cleaning

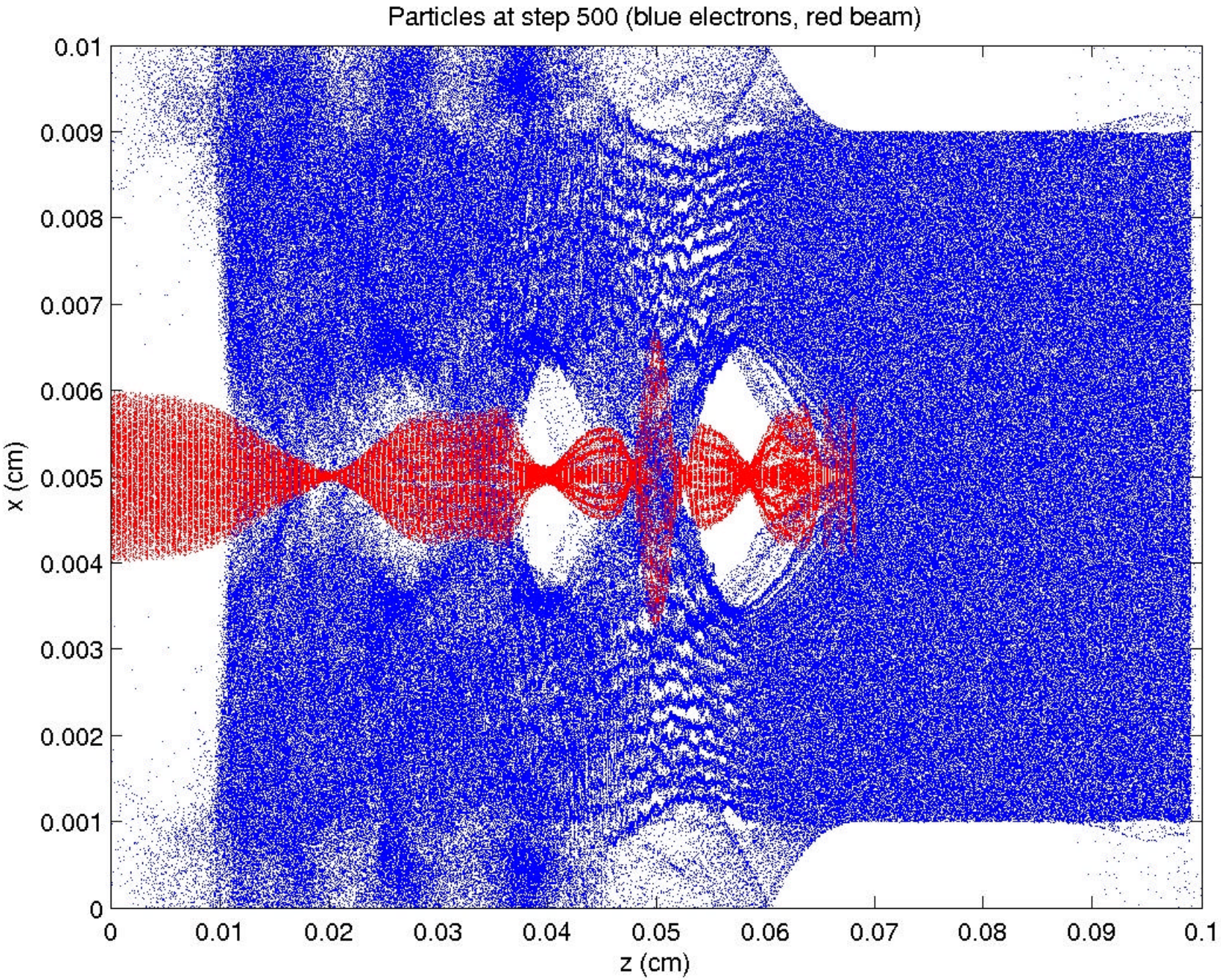
- To facilitate long simulation runs, Marder pass divergence error correction is provided.
  - Despite charge conserving accumulation, divergence errors can accumulate due to single precision arithmetic over millions of time steps.
  - The power spectral density of accumulated round-off divergence error can be computed analytically. It can be shown that a Marder pass every 100 to 200 time steps will keep total RMS divergence error in the electric and magnetic field near numerical precision.
  - Since a Marder pass over the electric and magnetic fields is seldom done and the pass consists of purely local operations (easily parallelized; no Poisson solver required), keeping divergence error negligible has a negligible cost.
  - Divergence error cleaning can be done in simulations with materials, absorbing boundaries, etc.


Run8d Linear Growth
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
$\int |B_z|^2/(2\mu_0)\, d^3x$
0
2
4
6
8
10
12
t/Δ_t
x 10^4


V-PIC: Dipole radiation + absorbing boundies (|E|^2)
x 10^-4
3
2
1
0
z
60
50
40
30
20
10
y
60
50
40
30
20
10
x
10
20
30
40
50
60


60
50
40
30
20
10
20
40
60
80
100
120
140
160
180
200
220
80
60
40
20
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1


Helicity Spectrum of run10b
t=0.2
t=9.5
t=15.2
10^{12}
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
Hk d^3k
10^{-2}
10^{-1}
10^{0}
k
$2\pi/L_x$
$2\pi/L_z$
$2\pi/d_i$
$2\pi/d_e$


Particles at step 500 (blue electrons, red beam)
x (cm)
z (cm)
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1

V-PIC Current Status

- A proof-of-concept 3d fully relativistic PIC code was written.
- Sustained 7.8M particles advanced per second per processor (~0.13 μs) in the common case demonstrated
  - On RedHat Linux 7.3 + gcc-3.3.3 (patched) + LAM-MPI 7.0.4 (untweaked) on 2.533Ghz Pentium 4 + Gigabit ethernet
  - Memory subsystem is at theoretical limits.
  - Floating point subsystem is near theoretical limits (~60-80%).
  - Substantially faster than many other PIC codes.
- Superhexahedral domain decomposition field solver
  - Standalone parallel fraction at ~99.85% (same platform)
- Routinely running ~100^3 meshes with ~0.6B particles for ~50K time steps on 16 to 32 processors (overnight-ish).
- Ported to x86 clusters and LANL's Q machine and validated against test problems and magnetic reconnection problems.

V-PIC Development Path

- Phase 1 => Proof of concept
- Phase 2 => Super-hexahedral domain decomposition
  - Field solver implemented
  - Particle advance implemented
- Phase 3 => Cartesian mesh arbitrary domain decomposition
  - Particle advance implemented
- Phase 4 => Unstructured mesh arbitrary domain decomposition
- Other possible additions
  - Advanced field solvers (Courant-free and exact dispersion explicit FDTD)
  - Hybrid models (PIC-Fluid hybrid, gyrokinetic PIC, …)
  - Monte-Carlo collisions, advanced emission algorithms

References

- Sorting techniques and memory bandwidth minimization: Bowers. “Accelerating a Particle-In-Cell Simulation Using a Hybrid Counting Sort.” Journal of Computational Physics. 173. 393-411. 2001.
- Energy conserving interpolations in 3d: Eastwood et al. “Body-fitted Electromagnetic PIC Software for use on Parallel Computers.” Computer Physics Communications. 87. 155-178. 1995.
- Another code using a 6th order magnetic field advance: Blahovec et al. “3-D ICEPIC Simulations of the Relativistic Klystron Oscillator.” IEEE Transactions on Plasma Science. 28. 821. 2000.
- Production Use of V-PIC: Bowers and Li. “Helicity Dissipation in 3D PIC Simulations of Magnetic Reconnection in a Force Free Configuration.” APS-DPP. Albuquerque, NM. 2003.

Acknowledgments

- This work was supported by the University of California, operator of the Los Alamos National Laboratory under Contract W-7405-ENG-36 with the United States Department of Energy.
- Partial support for this work came from Los Alamos Laboratory Directed Research and Development (LDRD) funding sources.