Developing PIC Codes for the Next Generation Supercomputer using GPUs

Viktor K. Decyk
UCLA

Abstract

The current generation of supercomputer (petaflops scale) cannot be scaled up to exaflops (1000 petaflops), To achieve exascale computing, new classes of highly parallel, low power devices are being designed. The most successful so far has been the Graphical Processing Unit (GPU). I will describe how our research in developing algorithms for Particle-in-Cell codes for GPUs should help us exploit the next generation supercomputer when it arrives.

During the last 40 years, I have experienced 2 revolutions in Supercomputing

The first was vector computing:
We had to learn how to “vectorize” our codes

The second was distributed memory parallel computing:
We had to learn how to “parallelize” our codes with message-passing

Each time, it took the scientific community nearly a decade to adapt

I believe we are in the middle of a third revolution in hardware.

This revolution is driven by

- Increasing shared memory parallelism, because increasing frequency is not practical
- A requirement for low power computing
- Goal is Exaflops computing (10^{18} Floating Point operations per second)

Revolution in Hardware

Many new architectures

- Multi-core processors
- Low power SIMD accelerators, GPUs
- Low power embedded processors
- FPGAs
- Heterogeneous processors

In a revolution, it is difficult to know which path leads to prosperity

GPUs are graphical processing units originally developed for graphics and games

- Programmable in 2007

GPUs consist of:

- 12-30 SIMD multiprocessors, each with small (16-48KB), fast (4 clocks) shared memory
- Each multi-processor contains 8-32 processor cores
- Large (0.5-6.0 GB), slow (400-600 clocks) global shared memory, readable by all units
- No cache on some units
- **Very fast (1 clock) hardware thread switching**

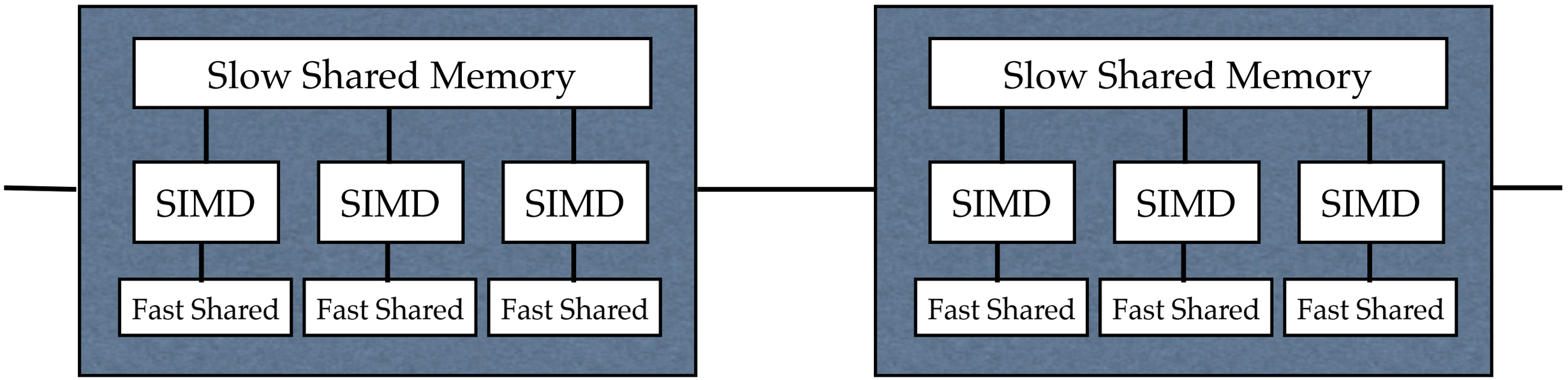
GPU Technology has two special features:

- High bandwidth access to global memory (>100 GBytes/sec)
- Ability to handle thousands of threads simultaneously, greatly reducing memory "stalls"

Challenges:

- High global memory bandwidth is achieved mainly for stride 1 access
  (Stride 1 = adjacent threads read adjacent locations in memory)
- Best to read/write global memory only once (streaming algorithms)
- shared memory is very small

Simple Hardware Abstraction for Next Generation Supercomputer



MPI node

MPI node

A distributed memory node consists of

- SIMD (vector) unit works in lockstep with fast shared memory and synchronization
- Multiple SIMD units coupled via “slow” shared memory and synchronization

Distributed Memory nodes coupled via MPI

Each MPI node is a powerful computer by itself
A supercomputer is a hierarchy of such powerful computers

OpenCL programming model uses such an abstraction for a single node

This hardware model matches GPUs very well

- But can be adapted to other hardware.

On NVIDIA GPU:

- Vector length = block size (typically 32-128)
- Fast shared memory = 16-64 KB.

On Intel multicore:

- Vector length for SSE = 4
- Fast shared memory = L1 Cache

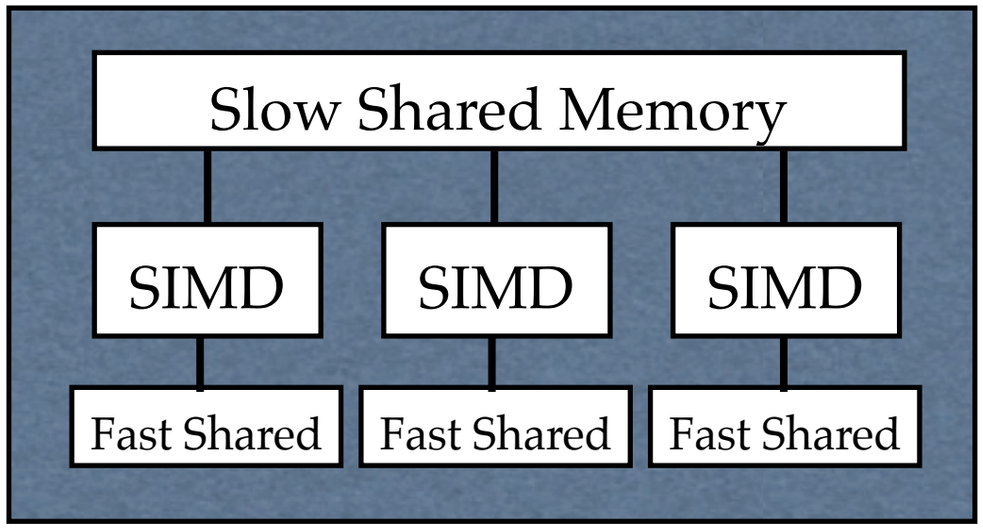
Intel MIC (Knights Corner)

- Vector length = 16

Designing new algorithms for next generation computers

This abstract machine contains a number of familiar hardware elements

- SIMD (vector) processors
- Small working memory (caches)
- Distributed memories



Scientific programmers have experience with each of these elements

Vector algorithms

- Calculation on a data set whose elements are independent (can be done in any order)
- Long history on Crays, Fijitsu, NEC supercomputers

Blocking (tiling) algorithms

- When data will be read repeatedly, load into faster memory and calculate in chunks
- Long history on RISC processors

Domain decomposition algorithms

- Partition memory so different threads work on different data
- Long history on distributed memory computers

Designing new algorithms for next generation computers

Programming these new machines uses many familiar elements,
but put together in a new way.

But some features are unfamiliar:

Languages (CUDA, OpenCL, OpenACC) have new features

GPUs require many more threads than physical processors

- Hides memory latency
- Hardware uses master-slave model for automatic load balancing among blocks

Optimizing data movement is very critical

Simplest example: **Vector** algorithms

In PIC code, often appears in field solvers

- Stride 1 memory access optimal (to avoid memory bank conflicts)

Here is a code fragment of a vectorized Poisson solver, $\mathbf{k} \bullet \mathbf{E} = 4\pi qn(k)$

- Usually inner loops were vectorized

```
complex qt(nyv,nxvh), fxyt(nyv,2,nxvh), ffct(nyhd,nxhd)

do j = 2, nxh
   dkx = dnx*real(j - 1)
   do k = 2, nyh
      k1 = ny2 - k
      at1 = real(ffct(k,j))*aimag(ffct(k,j))
      at2 = dkx*at1
      at3 = dny*real(k - 1)*at1
      zt1 = cmplx(aimag(qt(k,j)),-real(qt(k,j)))
      zt2 = cmplx(aimag(qt(k1,j)),-real(qt(k1,j)))
      fxyt(k,1,j) = at2*zt1
      fxyt(k,2,j) = at3*zt1
      fxyt(k1,1,j) = at2*zt2
      fxyt(k1,2,j) = -at3*zt2
   enddo
enddo
```

Vector algorithms

Here is a code fragment of a Poisson solver written for CUDA Fortran

- The changes were minimal: loop index replaced by thread ids
- If test to check loop bounds
- Both loops are running in parallel

```
k = threadIdx%x + blockDim%x*(blockIdx%x - 1)
j = blockIdx%y

dkx = dnx*real(j - 1)
if ((k > 1) .and. (k < nyh1) .and. (j > 1)) then
   k1 = ny2 - k
   at1 = real(ffct(k,j))*aimag(ffct(k,j))
   at2 = dkx*at1
   at3 = dny*real(k - 1)*at1
   zt1 = cmplx(aimag(qt(k,j)),-real(qt(k,j)))
   zt2 = cmplx(aimag(qt(k1,j)),-real(qt(k1,j)))
   fxyt(k,1,j) = at2*zt1
   fxyt(k,2,j) = at3*zt1
   fxyt(k1,1,j) = at2*zt2
   fxyt(k1,2,j) = -at3*zt2
endif
```

See my presentation, "Development of numerical algorithms for the next generation of supercomputers," at US-Japan Workshop for more examples: http://www-fps.nifs.ac.jp/ohtani/JIFT2011/program.shtml

Dawson2 at UCLA:
Experimental Cluster of GPUs is a prototype for next generation supercomputer

96 nodes: ranked 235 in top 500

- Each node has: 12 Intel G7 X5650 CPUs and 3 NVIDIA M2090 GPUs.
- Each GPU has 512 cores: total GPU cores=147,456 cores, total CPU cores=1152

Dawson2
Performance: 70 TFlops
Cost: about US$1,000,000
8千万円 = 1千6百石
Power: 96 KW.
Not a production facility

NIFS Plasma Simulator
Performance: 77 TFlops
Production facility

Cost, power will drive supercomputers in this direction

Main Application for Dawson2 cluster: Particle-in-Cell Codes

PIC codes integrate the trajectories of many particles interacting self-consistently via electromagnetic fields. They model plasmas at the most fundamental, microscopic level of classical physics.

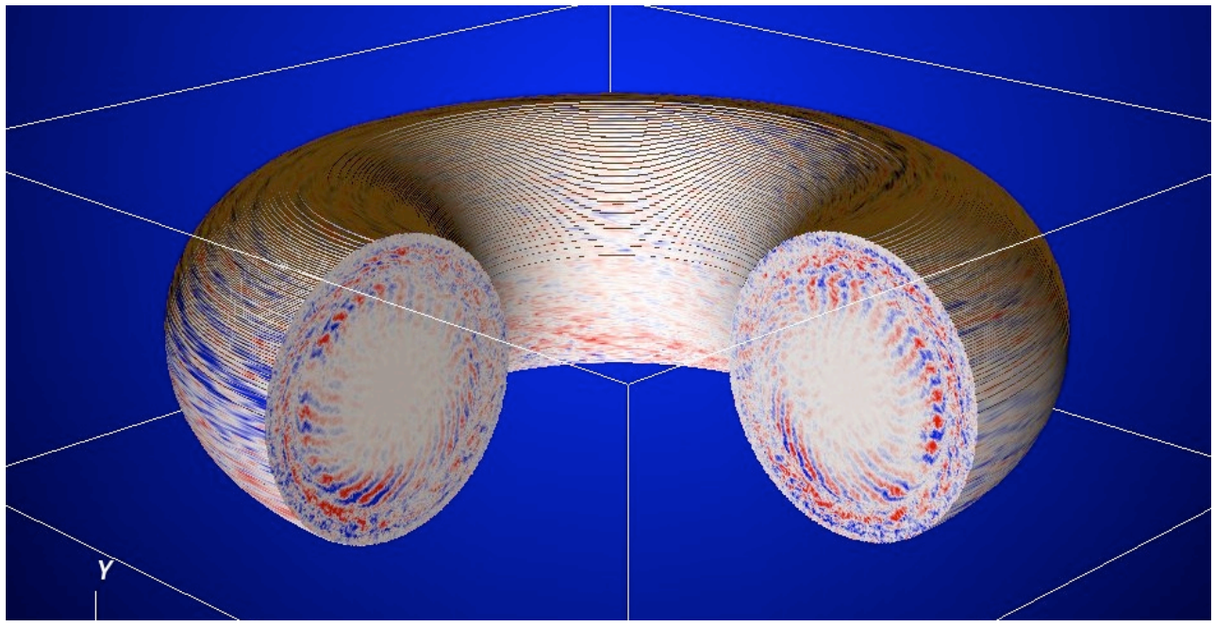
PIC codes are used in almost all areas of plasma physics, such as fusion energy research, plasma accelerators, space physics, ion propulsion, plasma processing, and many other areas.

Most complete, but most expensive models. Used when more simple models fail, or to verify the realm of validity of more simple models.

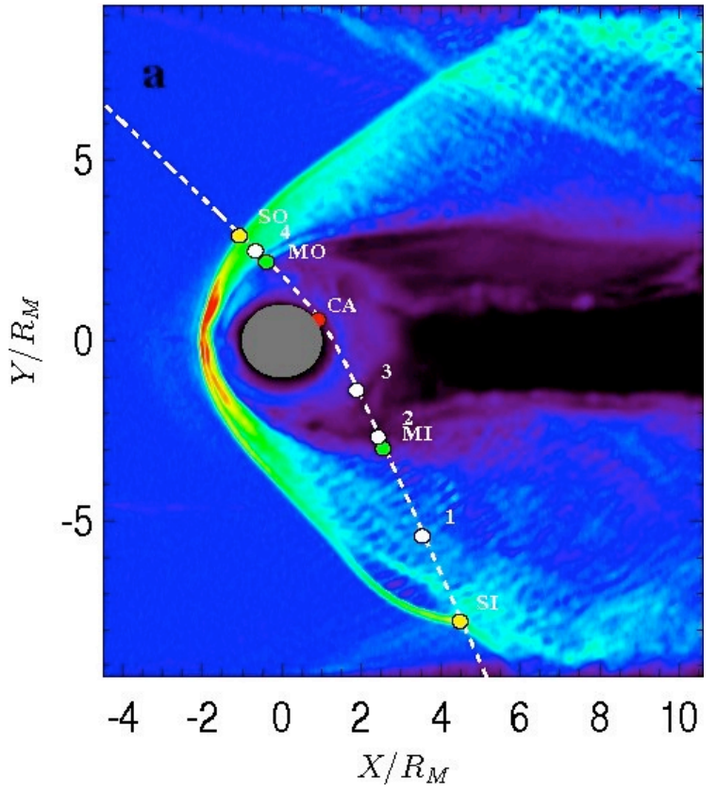
Largest calculations:

- ~1 trillion interacting particles (on Roadrunner)
- ~300,000 processors (on Jugene)

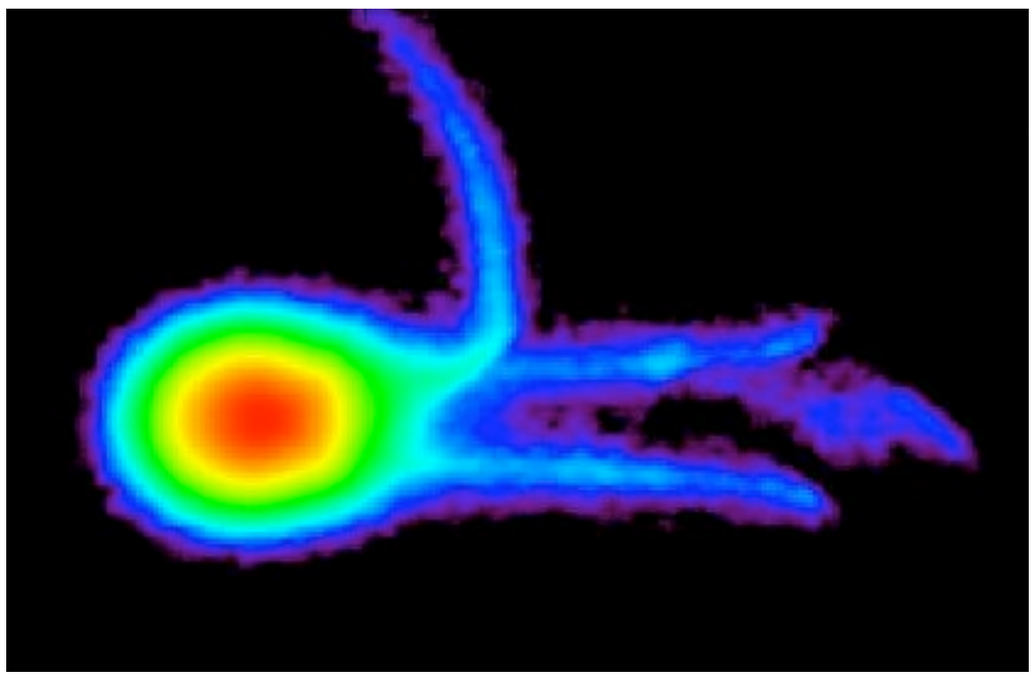
Particle-in-Cell Codes



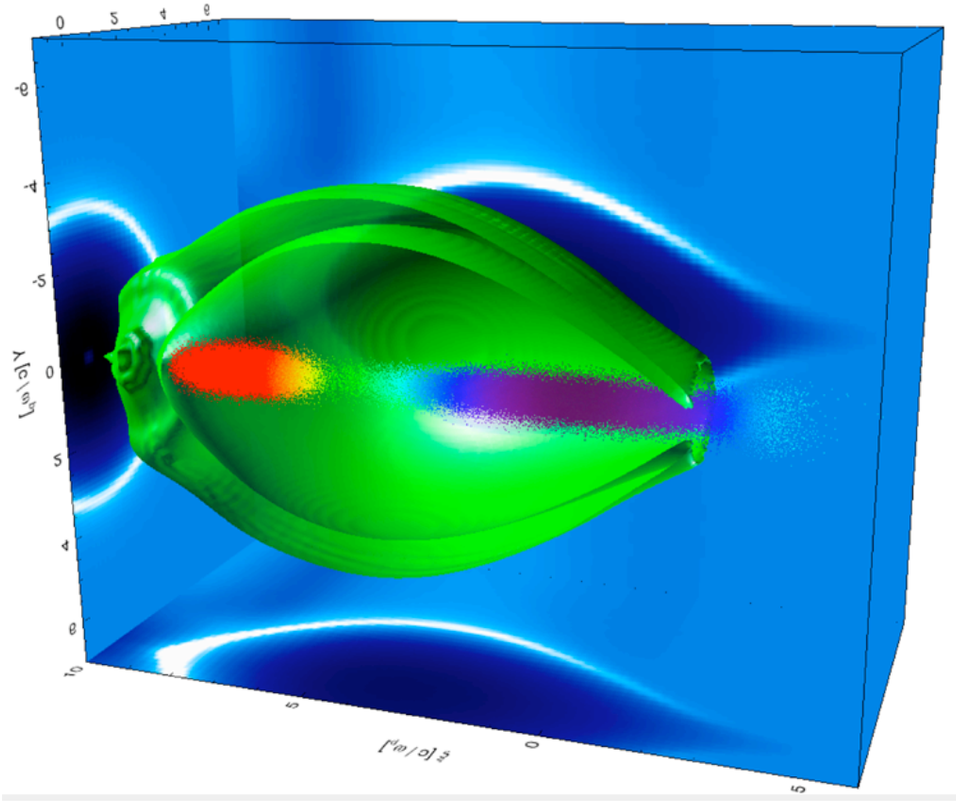
Simulation of ITG turbulence in Tokamaks
R. Sanchez et. al.



Simulation of magnetosphere of Mercury
P. Travnicek et.al,

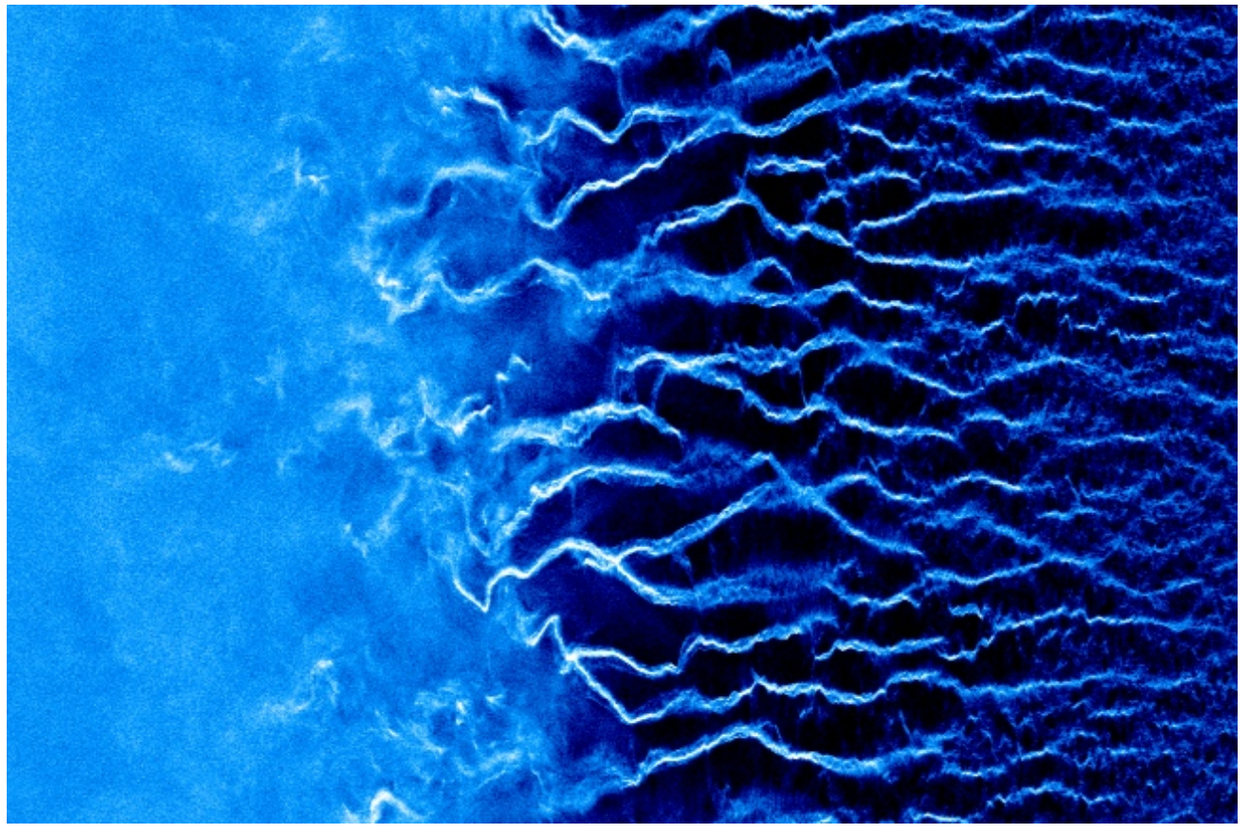


Energetic electrons in laser-driven fusion simulation
J. May and J. Tonge

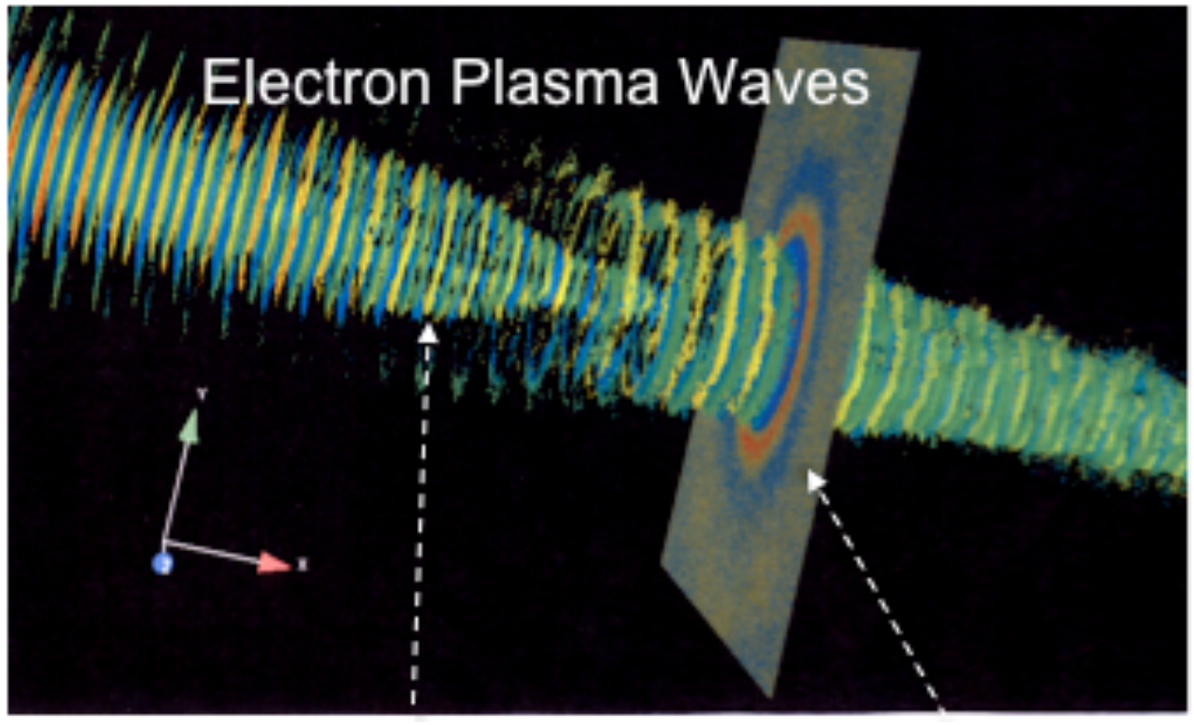


Simulation of compact plasma wakefield accelerator
C. Huang, et. al.

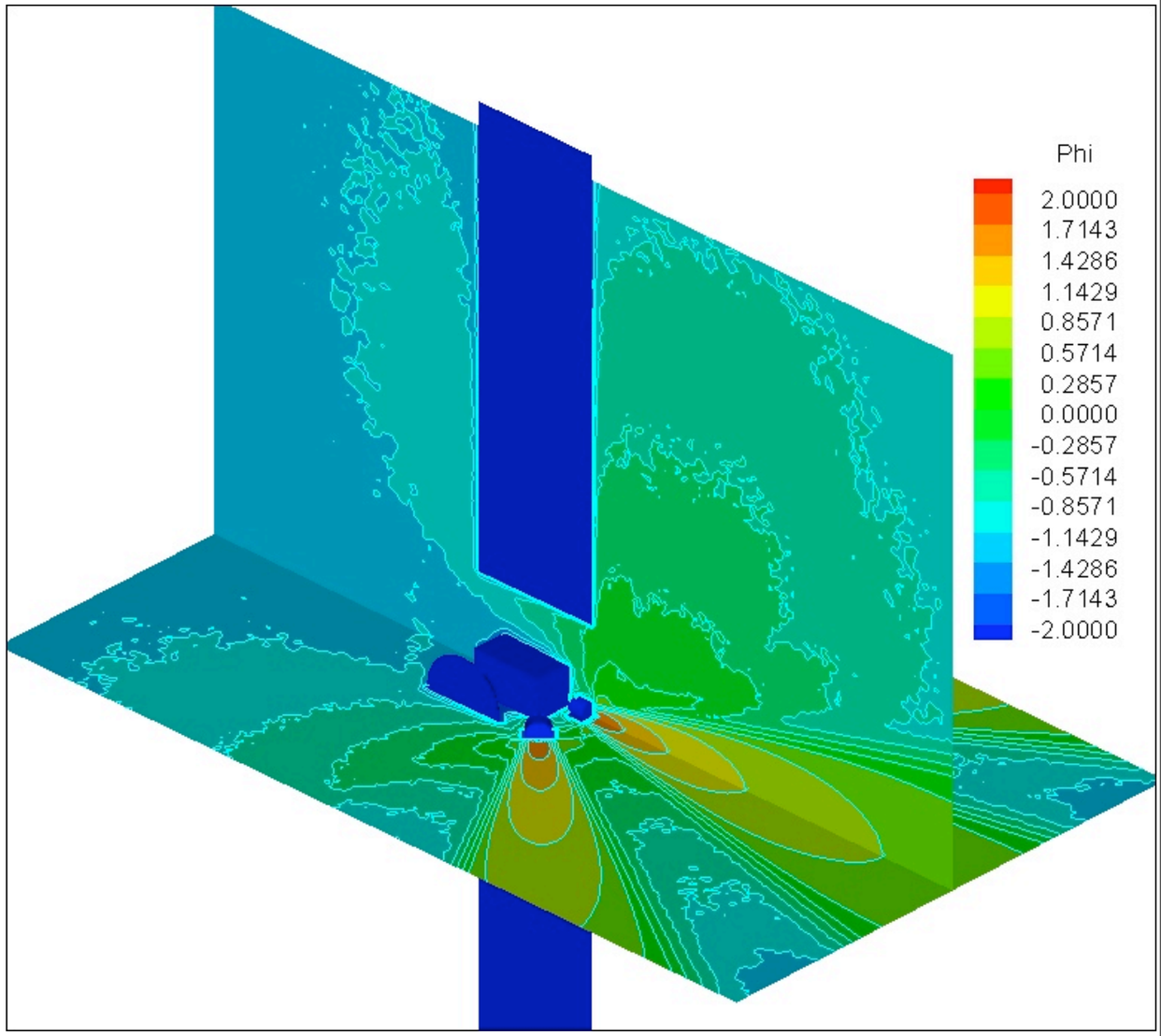
Particle-in-Cell Codes



Relativistic shock simulation, S. Martins, et. al.



Simulation of intense laser-plasma interactions, L. Yin, et. al.



Ion propulsion modeling, J.Wang, et. al.

Particle-in-Cell Codes

Simplest plasma model is electrostatic:

1. Calculate charge density on a mesh from particles:

$$\rho(\boldsymbol{x}) = \sum_i q_i S(\boldsymbol{x} - \boldsymbol{x}_i)$$

2. Solve Poisson's equation:

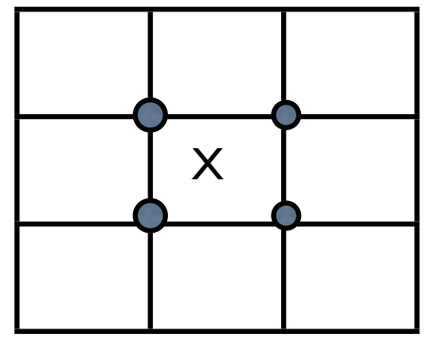
$$\nabla \cdot \boldsymbol{E} = 4\pi\rho$$

3. Advance particle's co-ordinates using Newton's Law:

$$m_i \frac{d\boldsymbol{v}_i}{dt} = q_i \int \boldsymbol{E}(\boldsymbol{x}) S(\boldsymbol{x}_i - \boldsymbol{x})\, d\boldsymbol{x} \qquad \frac{d\boldsymbol{x}_i}{dt} = \boldsymbol{v}_i$$

Inverse interpolation (scatter operation) is used in step 1 to distribute a particle's charge onto nearby locations on a grid.

Interpolation (gather operation) is used in step 3 to approximate the electric field from grids near a particle's location.



Distributed Memory Programming for PIC

This is dominant technique used on today's supercomputers

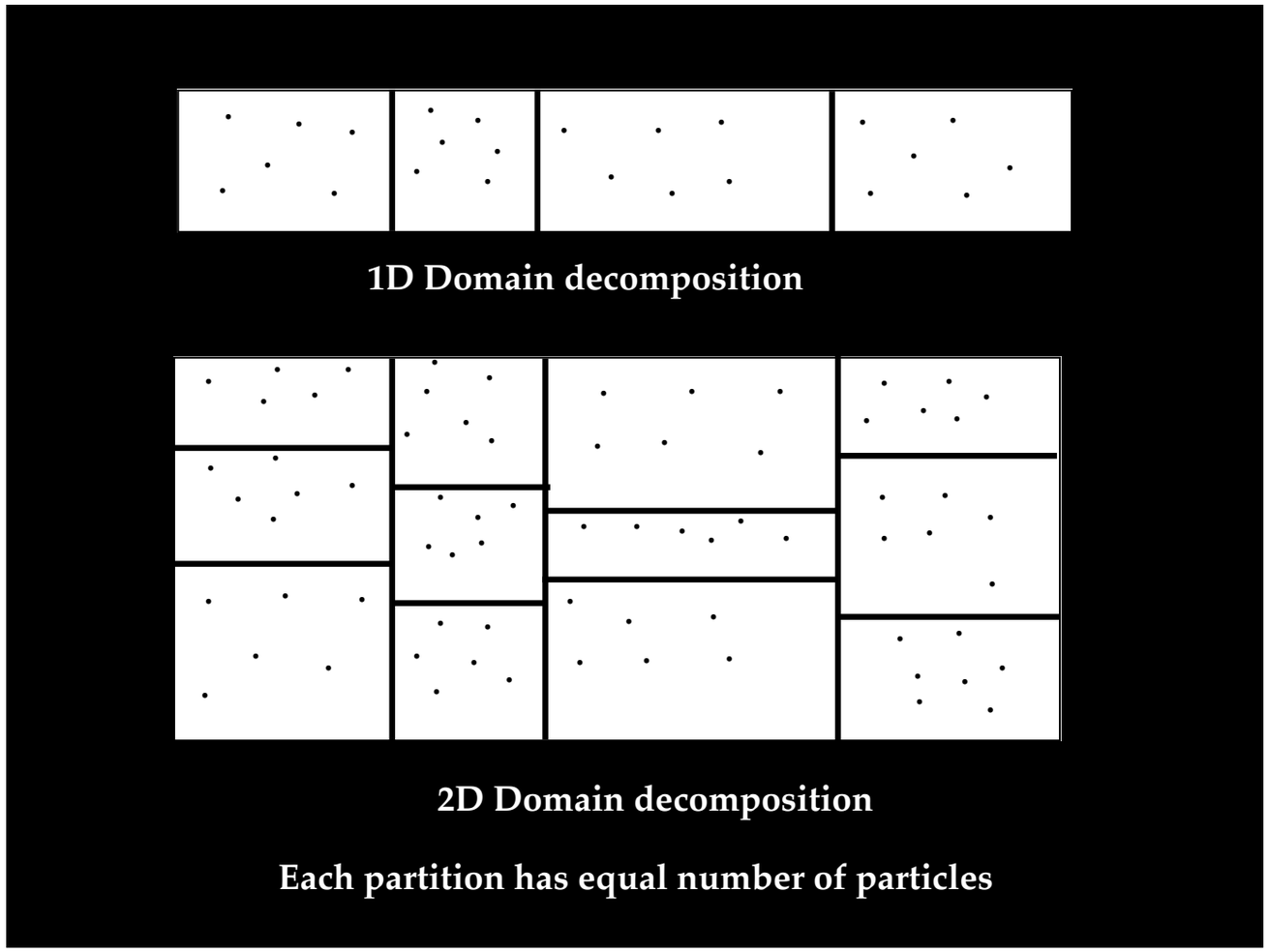
Domain decomposition is used to assign which data reside on which processor
- No shared memory, data between nodes sent with message-passing (MPI)
- Most parallel PIC codes are written this way

Details can be found at:

P. C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes," J. Computational Phys. 85, 302 (1989)

Distributed Memory Programming for PIC

Domain decomposition with MPI



Primary Decomposition has non-uniform partitions to load balance particles

- Sort particles according to spatial location
- Same number of particles in each non-uniform domain
- Scales to thousands of processors

Particle Manager responsible for moving particles

- Particles can move across multiple nodes

Distributed Memory Programming for PIC

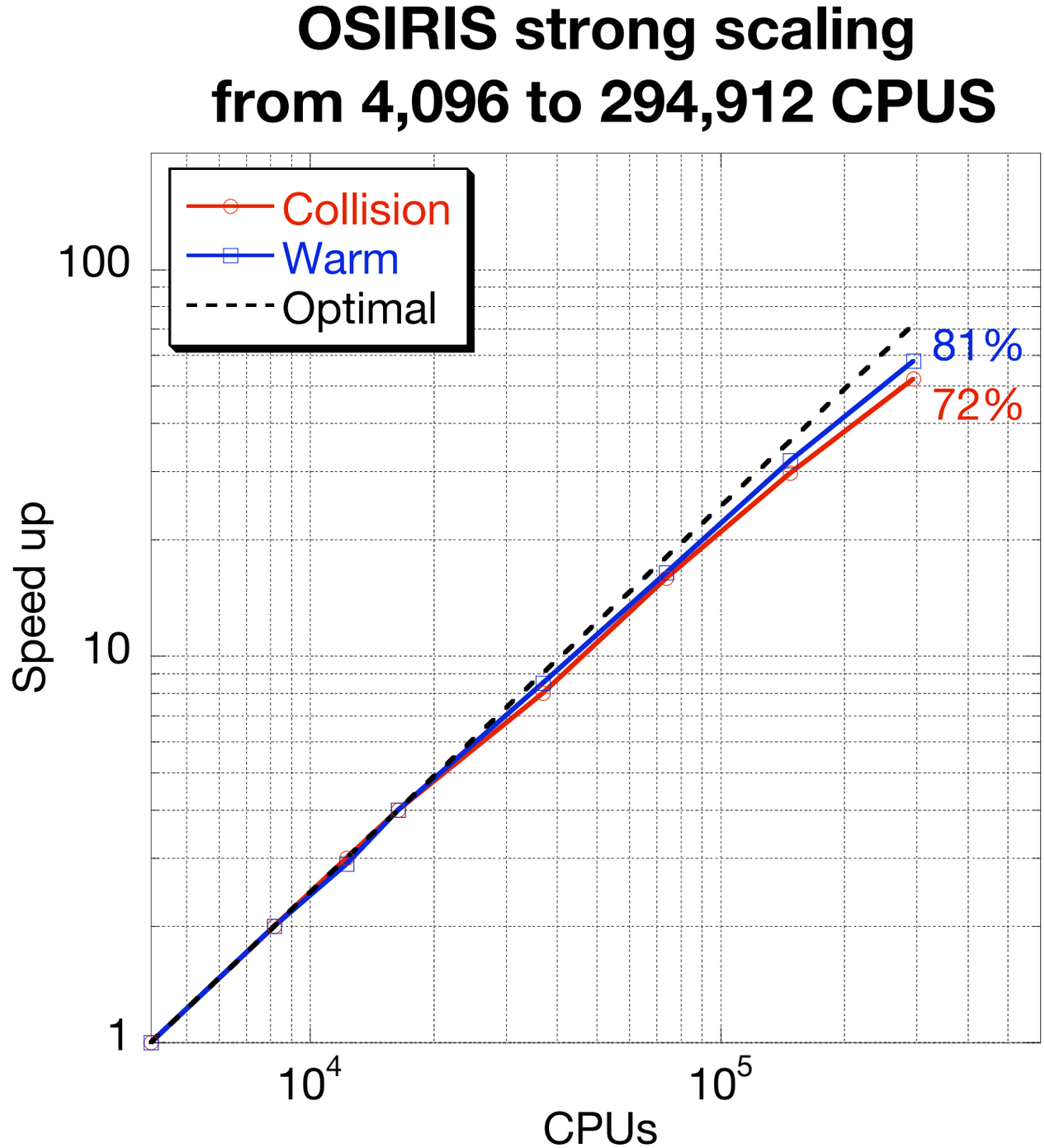
In our codes, we also keep the communication and computation procedures separate
- Bulk Synchronous Programming

There are 4 main communication procedures in the PIC code:
- Particle manager (move particles to correct processor)
- Field Manager (add/copy guard cells)
- Partition manager (move field elements between uniform/non-uniform partitions)
- Transpose (used by FFT)

Electromagnetic Code
Blue Gene Benchmark:

1024x1024x512 cells
8,589,934,592 particles

Not trivial to parallelize:
Two data structures
and inhomogeneous density.



GPU Programming for PIC: One GPU

Most important bottleneck is memory access
- PIC codes have low computational intensity (few flops/memory access)
- Memory access is irregular (gather/scatter)

Memory access can be optimized with a streaming algorithm

PIC codes can implement a streaming algorithm by keeping particles ordered by cell.
- Minimizes global memory access since field elements need to be read only once.
- Cache is not needed, gather/scatter can be avoided.
- Deposit and particles update can have optimal memory access.
- Single precision can be used for particles

Adaptable PIC Algorithm
- Sorting cell can contain multiple grids
- Adaptable with 4 adjustable parameters to match architecture
- Domain decomposition used to avoid data dependencies

Challenge: optimizing particle reordering

GPU Programming for PIC: Maintaining Particle Order is Main Challenge

Stream compaction: reorder elements into substreams of like elements

- Less than a full sort, low overhead if already ordered

Two cases

- 1: Particle moves to another thread in same vector or block
  Use fast memory
- 2: Particle moves to another thread in a different vector or block
  Essentially message-passing, except buffer contains multiple destinations

The reordering algorithm does not match the architecture well

- Does not have stride 1 access (poor data coalescing)
- Does not run in lockstep (has warp divergence)

Reference:

V. K. Decyk and T. V. Singh, "Adaptable Particle-in-Cell Algorithms for Graphical Processing Units," Computer Physics Communications, 182, 641, 2011.

See also: http://www.idre.ucla.edu/hpc/research/

Evaluating New Particle-in-Cell (PIC) Algorithms on single GPU: Electromagnetic Case

We also implemented a 2-1/2D electromagnetic code
- Relativistic Boris mover
- Deposit both charge and current density
- Reorder twice per time step
- 10 FFTs per time step

Optimal parameters were lth=64, and
On Fermi C2050, ngpx = 2, ngpy = 2
On Tesla C1060 and GTX280, ngpx = 1, ngpy = 2

Observations:
- About 2.9% of the particles left the cell each time step on Fermi C2050
- About 4.6% of the particles left the cell each time step on Tesla C1060, GTX 2010
- Field solver took about 9% of the code

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: Electromagnetic Case

Warm Plasma results with c/vth = 10, dt = 0.04

	Intel Nehalem	Fermi C2050	Tesla C1060	GTX 280
Push	81.7 ns.	0.89 ns.	1.13 ns.	1.08 ns.
Deposit	40.7 ns.	0.78 ns.	1.06 ns.	1.04 ns.
Reorder	0.5 ns.	0.57 ns.	1.13 ns.	0.97 ns.
Total Particle	122.9 ns.	2.24 ns.	3.32 ns.	3.09 ns.

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 55x,
on the Telsa C1060 was 37x, and on the GTX 280 was 40x.

Cold Plasma (asymptotic) results with vth = 0, dt = 0.025

	Intel Nehalem	Fermi C2050	Tesla C1060	GTX 280
Push	78.5 ns.	0.51 ns.	0.79 ns.	0.74 ns.
Deposit	37.3 ns.	0.58 ns.	0.82 ns.	0.81 ns.
Reorder	0.4 ns.	0.10 ns.	0.16 ns.	0.15 ns.
Total Particle	116.2 ns.	1.20 ns.	1.77 ns.	1.70 ns.

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 97x,
on the Telsa C1060 was 66x, and on the GTX 280 was 69x.

Current Research

How do we design code for a cluster of GPUs?

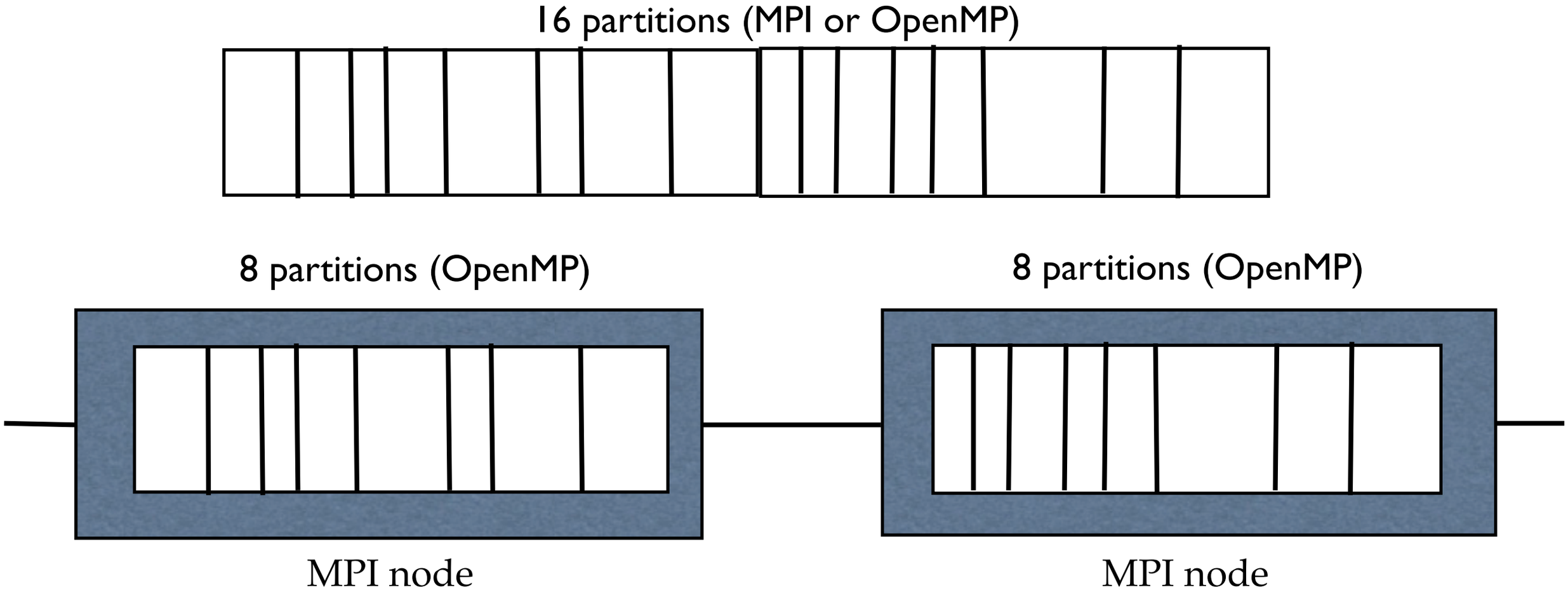
We need a hybrid MPI/GPU design
- Scientific software typically lasts longer than hardware
- Can we keep the code general enough to support other architectures?
- Can we minimize the amount of specialized code?

Current Solution: implement a generic MPI/OpenMP code, specialize later to CUDA
- Simpler to program and understand

Hybrid MPI/OpenMP Programming for PIC

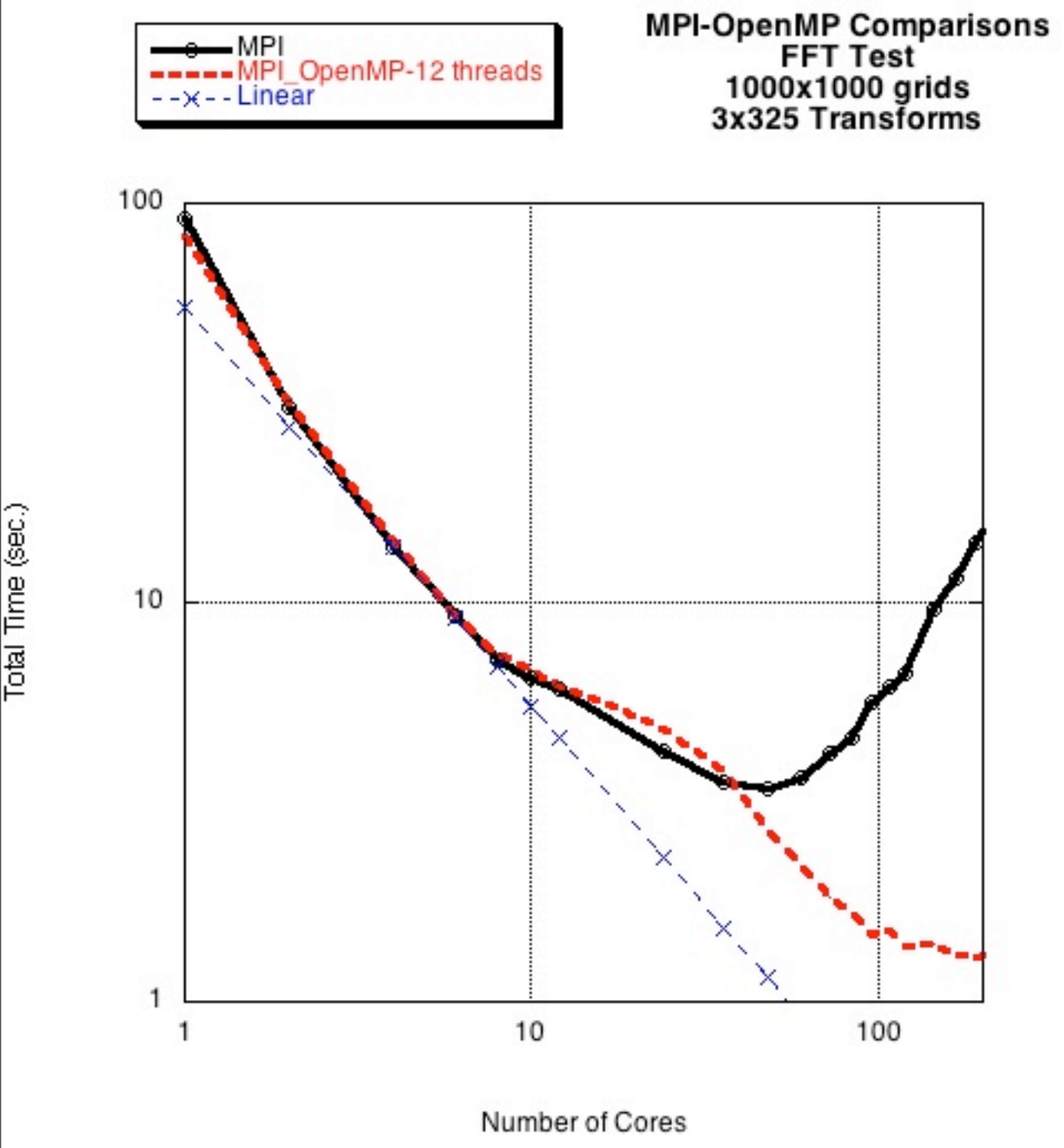
Nested (hierarchical) domain decompositions

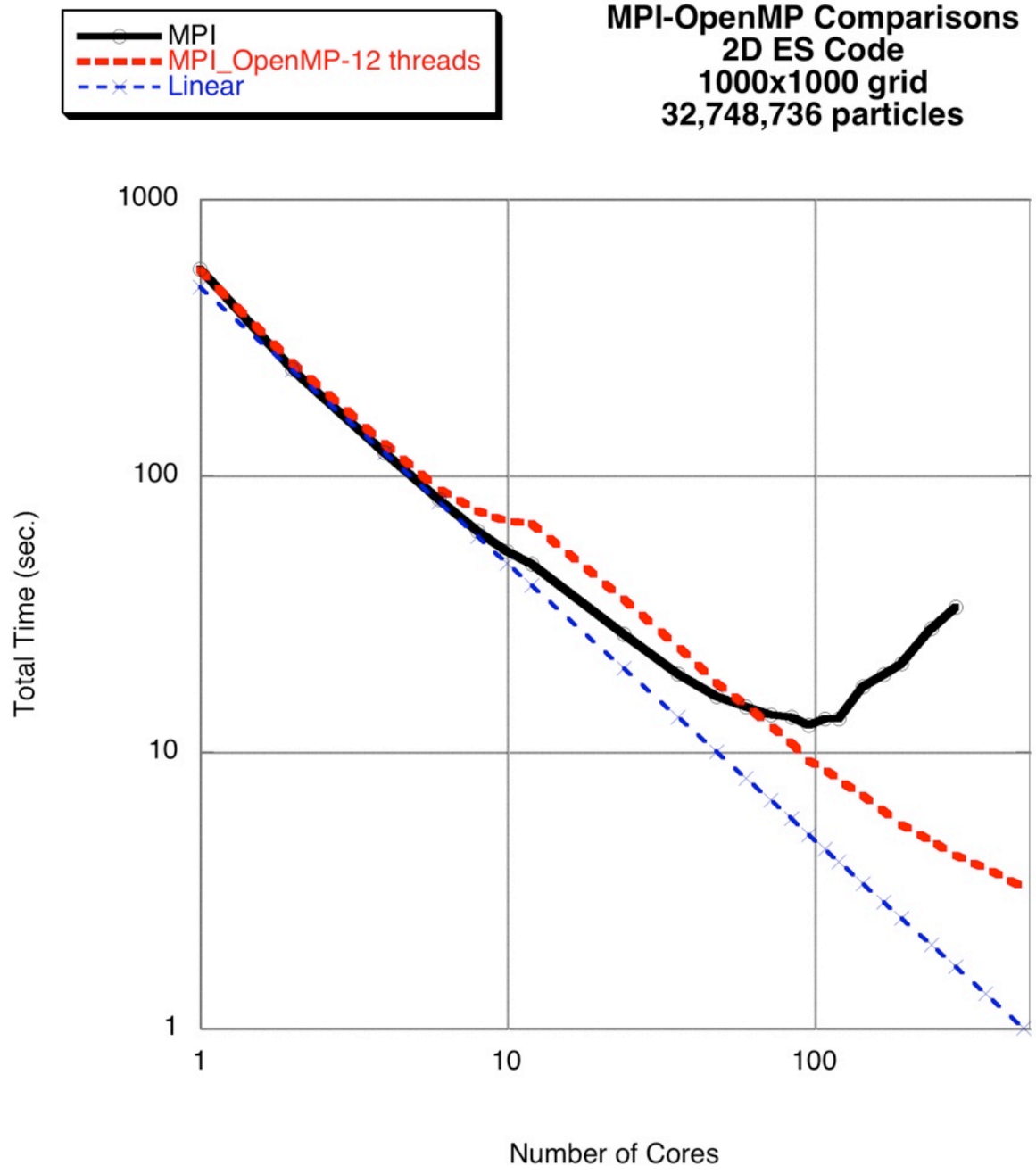
- Each shared memory node has its own domain
- Edges of shared memory domain define MPI domain



Allows fine grained partitions with reduced communications requirements

- Prototype for GPU-MPI decomposition





FFT gives nearly 3 times better performance with MPI-OpenMP than MPI alone
Overall, spectral code gets more than 4 times better performance

Conclusions

PIC Algorithms on GPUs are largely a hybrid combination of previous techniques

- Vector techniques from Cray
- Blocking techniques from cache-based architectures
- Message-passing techniques from distributed memory architectures

- Programming to Hardware Abstraction leads to common algorithms
- Streaming algorithms optimal for memory-intensive applications

Scheme should be portable to other architectures with similar hardware abstractions

- OpenCL version allows the code to run on AMD GPU and Intel Multicore

Hierarchical (Nested) Domain Decompositions look promising for GPU cluster

- Also improves performance on shared memory/distributed memory architectures

Although GPUs and NVIDIA may not survive in the long term, I believe exascale computers will have similar features.

We do not plan to wait 10 years to start developing algorithms for the next generation supercomputer