Programming Hybrid HPC Systems What is Old is New Again

Viktor K. Decyk
UCLA

Abstract

The next generation of supercomputers will likely consist of a hierarchy of parallel computers. If we can define each node as a parameterized abstract machine, then it is possible to design algorithms even if the actual hardware varies. Such an abstract machine can be defined to consist of a collection of vector (SIMD) processors, each with a small shared memory, communicating via a larger global memory. This abstraction fits a variety of hardware, such as Graphical Processing Units (GPUs), and multi-core processors with vector extensions. To program such an abstract machine, one can use ideas familiar from the past: vector algorithms from vector supercomputers, blocking or tiling algorithms from cache-based machines, and domain decomposition from distributed memory computers. Examples from our GPU Particle-in-Cell code will be shown to illustrate this approach.

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:

- ~3 trillion interacting particles

I have been using "Supercomputers" for 40 years, starting with the IBM 360/91 at UCLA in 1973.

In 1975, the performance of my
2D Particle-in-Cell (PIC) code was:

99 microsec/particle/time-step

Cost: 30 cents/time step

A typical overnight run cost the same as my monthly salary as a TA!

The first machine I used at NERSC (then MFECC) was the A machine, a CDC 7600
The A machine had a memory hierarchy: small core and large core memories

On the A machine, my 2D PIC code gave a performance of:
118 microsec/particle/time-step

During this time, I have experienced 2 revolutions in Supercomputing

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

In 1984, the performance of my
2D Particle-in-Cell code was:

Cray 1s
8.0 microsec/particle/time-step

Fujitsu VP-100:
5.3 microsec/particle/time-step

An improvement of more than 10!

Vector algorithms

In PIC codes, often appears in field solvers

- vectorizable means elements can be processed in any order
- Stride 1 memory access optimal (to avoid memory bank conflicts)

Here is a code fragment of a vectorized Poisson solver, **k•E** = 4πqn(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)
cdir$ ivdep
      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
```

The second revolution was the introduction of distributed memory computing:
We had to learn how to "parallelize" our codes with message-passing

In 1997, the performance of my
2D Particle-in-Cell code was:

Cray T3E with 128 processors:
13 nanosec/particle/time-step

Mac G3/266: 2.4 microsec/particle/time-step

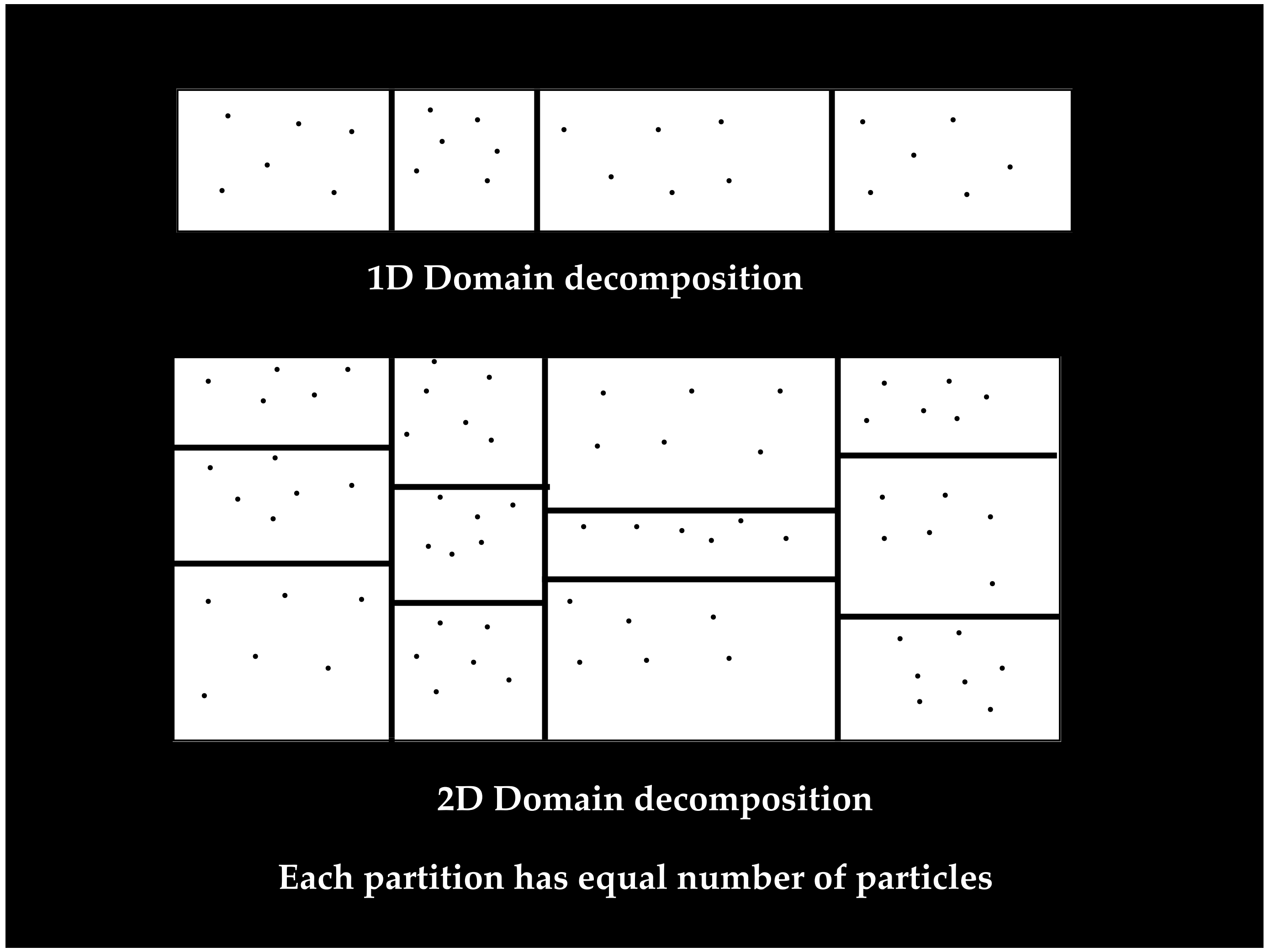
An improvement of over 500 in 13 years for supercomputers
A desktop machine becomes equal to a supercomputer in ~10 years.

3D PIC now became practical:
my 3D Particle-in-Cell code performance was:

Cray T3E with 128 processors:
74 nanosec/particle/time-step

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 many 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 only 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)

Domain decomposition algorithms
In PIC codes, often appears when processing edges of domains

Here is a code fragment from the particle manager, using MPI

Particles which leave a domain are buffered and sent to and received from neighboring processors

```
c post receive
      call MPI_IRECV(rbufl,nbsize,mreal,kl-1,iter-1,lgrp,msid(1),ierr)
      call MPI_IRECV(rbufr,nbsize,mreal,kr-1,iter,lgrp,msid(2),ierr)
c send particles
      call MPI_ISEND(sbufr,idimp*jsr(1),mreal,kr-1,iter-1,lgrp,msid(3),
     1ierr)
      call MPI_ISEND(sbufl,idimp*jsl(1),mreal,kl-1,iter,lgrp,msid(4),
     1ierr)
c wait for particles to arrive
      call MPI_WAIT(msid(1),istatus,ierr)
      call MPI_GET_COUNT(istatus,mreal,nps,ierr)
      jsl(2) = nps/idimp
      call MPI_WAIT(msid(2),istatus,ierr)
      call MPI_GET_COUNT(istatus,mreal,nps,ierr)
      jsr(2) = nps/idimp
      ...
c make sure sbufr and sbufl have been sent
      call MPI_WAIT(msid(3),istatus,ierr)
      call MPI_WAIT(msid(4),istatus,ierr)
```

Blocking (tiling) algorithms useful due to presence of memory hierarchies

- Process data is small chunks, which fit within fastest cache memory

Here is a code fragment of a complex transpose, used by FFT

```
complex f(nxv,ny), g(nyv,nx)
do k = 1, ny
   do j = 1, nx
      g(k,j) = f(j,k)
   enddo
enddo
```

f is read with optimal stride 1,
but g is written with large jumps in memory

Break up loops into small blocks, where block is typically size of smallest fast memory

```
   complex f(nxv,ny), g(nyv,nx), buff(nblok+1,nblok)
! loop over tiles
   do kb = 1, ny/nblok
      koff = nblok*(kb - 1)
      do jb = 1, nx/nblok
         joff = nblok*(jb - 1)
! copy data into tile, with optimal stride
         do k = 1, nblok
            do j = 1, nblok
               buff(j,k) = f(j+joff,k+koff)
            enddo
         enddo
! copy data out of tile with optimal stride
         do j = 1, nblok
            do k = 1, nblok
               g(k+koff,j+joff) = buff(j,k)
            enddo
         enddo
      enddo
   enddo
```

f , g are accessed with optimal stride 1,
buff is accessed in fast memory with
smaller jumps in memory

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

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)

NVIDIA GPU

Intel Phi

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**

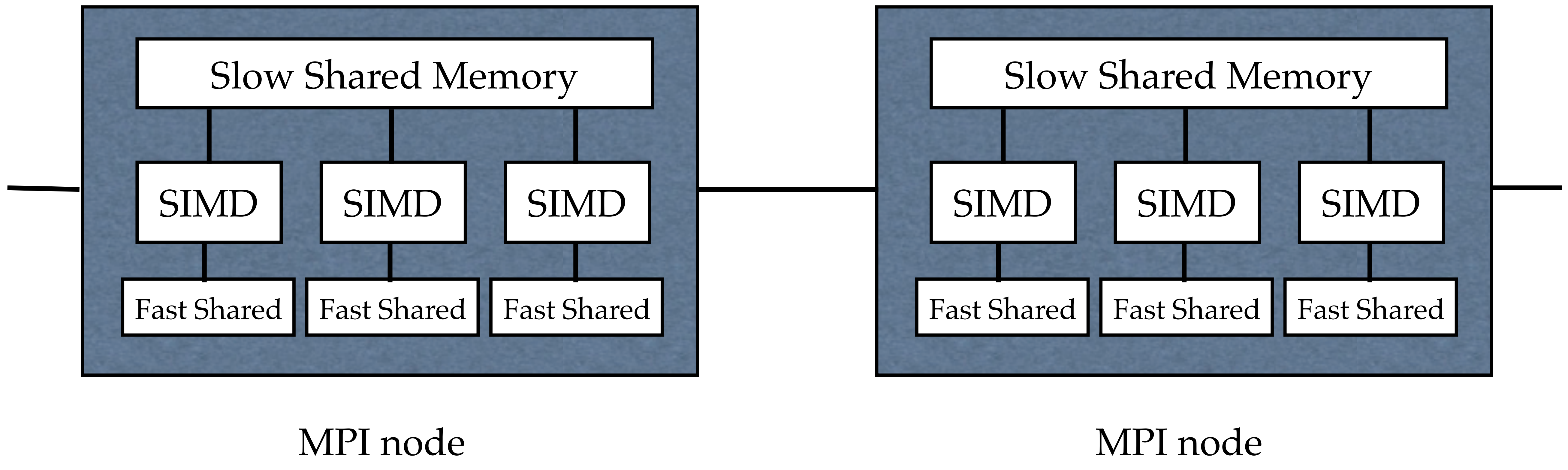
GPU Technology has two special features:
- High bandwidth access to global memory (>100 GBytes/sec), but for ordered access
- Ability to handle thousands of threads simultaneously, greatly reducing memory “stalls”

Intel MIC Architecture has Xeon Phi coprocessor
- Introduced in 2012

Phi consists of:
- 50-60 512-bit SIMD processors, each with L2 cache (512 KB)
- Large (up to 8.0 GB) global shared memory

Intel Phi Technology has two special features:
- High bandwidth access to global memory (up to 320 GBytes/sec)
- Four hardware threads per core, reducing memory latency

Simple Hardware Abstraction for Next Generation Supercomputer



A distributed memory node consists of

- SIMD (vector) unit works in lockstep with fast shared memory and synchronization
- Multiple SIMD units coupled via global 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 a variety of processors

On NVIDIA GPU:

- Vector length = block size (typically 32-128)
- Number of vector processors =12-30
- Fast shared memory = 16-64 KB, plus L2 cache

On Intel MIC (PHI):

- Vector length for single precision = 16
- Number of vector processors = 50-60
- Fast shared memory = L1 Cache (32 KB), L2 Cache (512 KB)

On Dual Intel Xeon multicore:

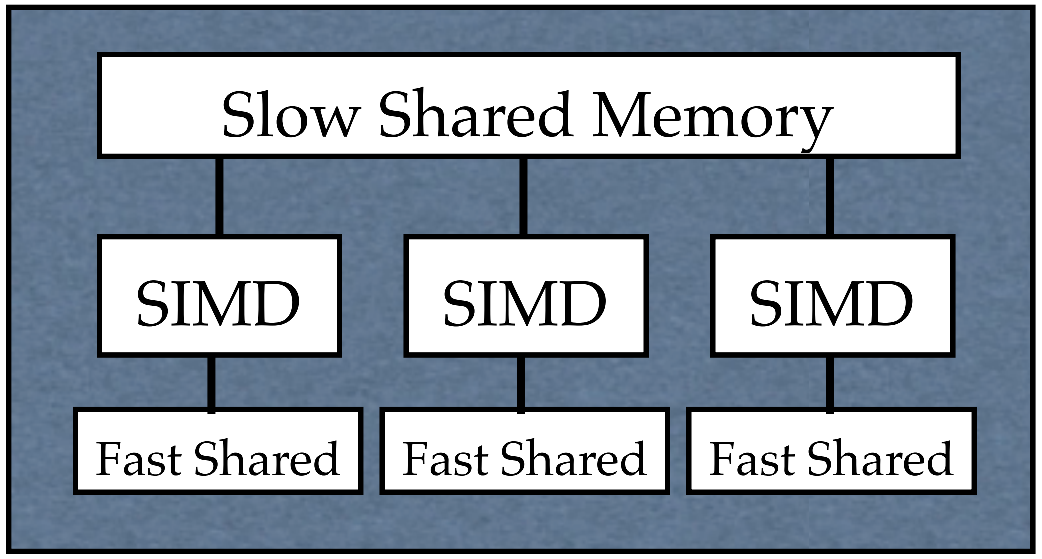
- Vector length for single precision with SSE2 = 4
- Number of vector processors = 4-16
- Fast shared memory = L1, L2 Cache

On Cray X-MP:

- Vector length for double precision = 64
- Number of vector processors = 2-4
- Fast shared memory = none

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.

Two levels of parallelism, vector/shared memory parallelism

Data parallel programming also useful, but little history at NERSC

But some features are unfamiliar:

Languages (CUDA, OpenCL, OpenACC, OpenMP) have new features

GPUs require many more threads than physical processors
- Hides memory latency
- Hardware can use master-slave model for automatic load balancing among blocks

Optimizing data movement is very critical

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
```

Blocking (tiling) algorithms

Here is a code fragment of a complex transpose written for CUDA Fortran

- Changes were minimal: loop index replaced by thread ids
- If test to check loop bounds
- Synchronization needed

Original Fortran

```
  complex f(nxv,ny), g(nyv,nx)
  complex buff(nblok+1,nblok)

! loop over tiles
   do kb = 1, ny/nblok
      koff = nblok*(kb - 1)
      do jb = 1, nx/nblok
         joff = nblok*(jb - 1)
! copy data into tile, with optimal stride
         do k = 1, nblok
            do j = 1, nblok
               buff(j,k) = f(j+joff,k+koff)
            enddo
         enddo
! copy data out of tile with optimal stride
         do j = 1, nblok
            do k = 1, nblok
               g(k+koff,j+joff) = buff(j,k)
            enddo
         enddo
      enddo
   enddo
```

CUDA Fortran

```
   complex, shared, &
  &dimension(nblok+1,nblok) :: buff

   koff = nblok*(blockIdx%y - 1)
   joff = nblok*(blockIdx%x - 1)
! copy data into tile, with optimal stride
   j = threadIdx%x
   k = threadIdx%y
   jx = j + joff
   ky = k + koff
   if ((jx <= nx) .and. (ky <= ny)) then
      buff(j,k) = f(jx,ky)
   endif
! make sure all threads are done
   call syncthreads()
! copy data out of tile with optimal stride
   j = threadIdx%y
   k = threadIdx%x
   jx = j + joff
   ky = k + koff
   if ((jx <= nx) .and. (ky <= ny)) then
      g(ky,jx) = buff(j,k)
   endif
```

Particle-in-Cell Codes

Simplest plasma model is electrostatic:

1. Calculate charge density on a mesh from particles:

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

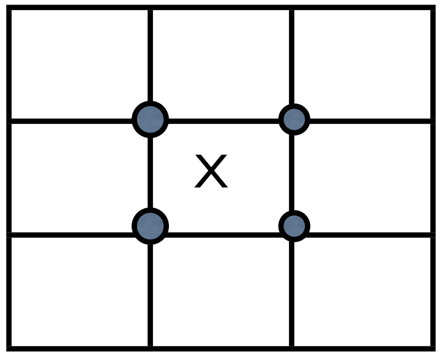
2. Solve Poisson's equation:

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

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.

When running in parallel, data collisions might occur

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 (global data read only once)

PIC codes can implement a streaming algorithm by keeping particles ordered by tiles.

- Minimizes global memory access since field elements need to be read only once.
- global gather/scatter can be avoided.
- Deposit and particles update can have optimal memory access.

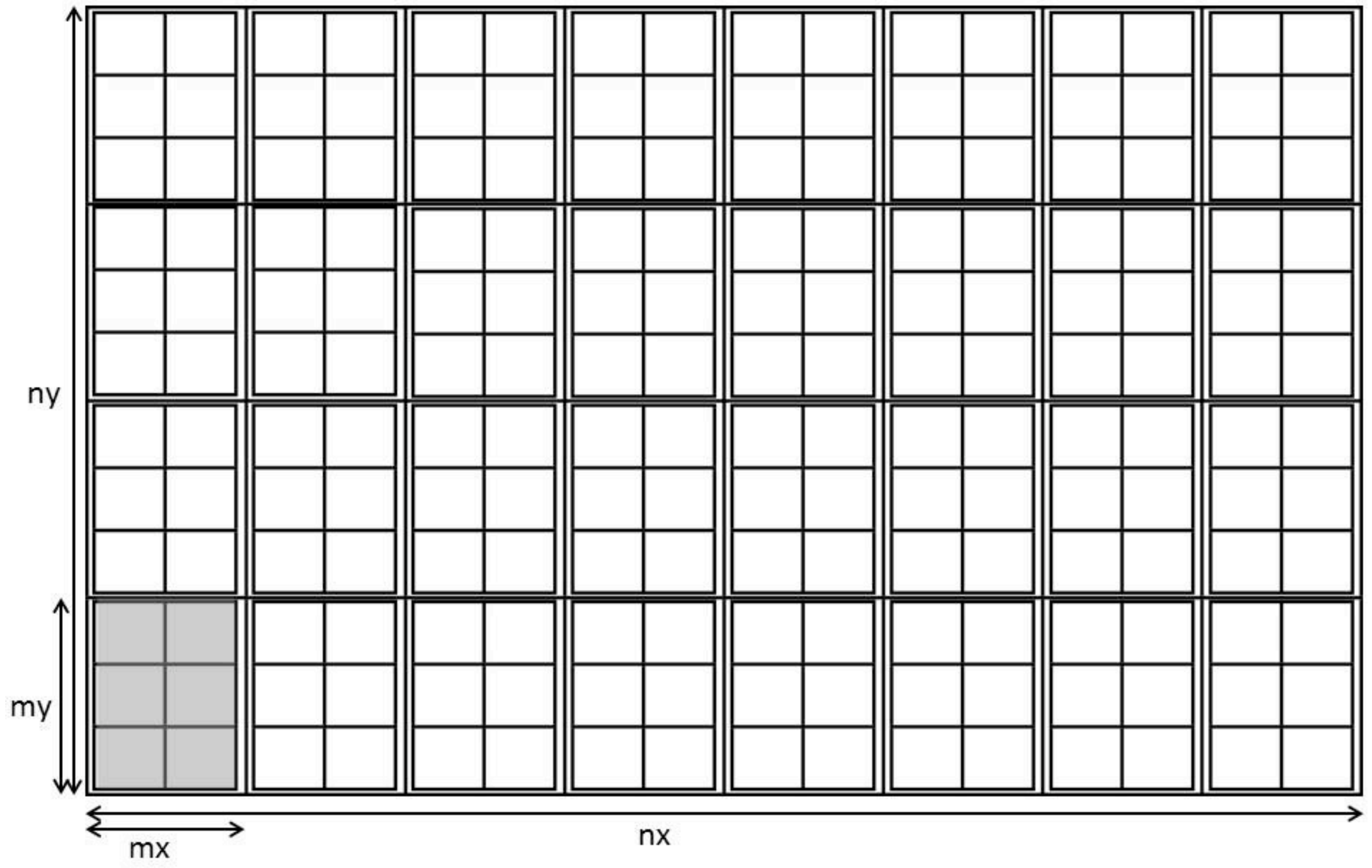
Challenge: optimizing particle reordering

Designing New Particle-in-Cell (PIC) Algorithms:
Particles ordered by tiles, varying from 2 x 2 to 16 x 16 grid points

We created a new data structure for particles, partitioned among threads blocks:

```
   dimension ppart(idimp,npmax,num_tiles)
```

This is domain decomposition, but on a microscopic scale!



Designing New Particle-in-Cell (PIC) Algorithms: **Push/Deposit Procedures**:

Within a tile, all particles read or write the same block of fields.

- Before pushing particles, copy fields to fast memory
- After depositing charge to fast memory, write to global memory
- Different tiles can be done in parallel.

Each tile contains data for the grids in the tile, plus guard cells: an extra column of grids on the right, and an extra row of grids on the bottom for linear interpolation.

For push, parallelization is easy, each particle is independent of others, no data hazards

- Similar to MPI code, but with tiny partitions

For deposit, parallelization is also easy if each tile is controlled by one thread

- This avoids data collisions where two threads try to update the same memory

However, if each tile is controlled by a vector of threads, data collisions are possible.

- Atomic updates (which treat an update as an uninterruptible operation) are one approach

Designing New Particle-in-Cell (PIC) Algorithms: **Maintaining Particle Order**

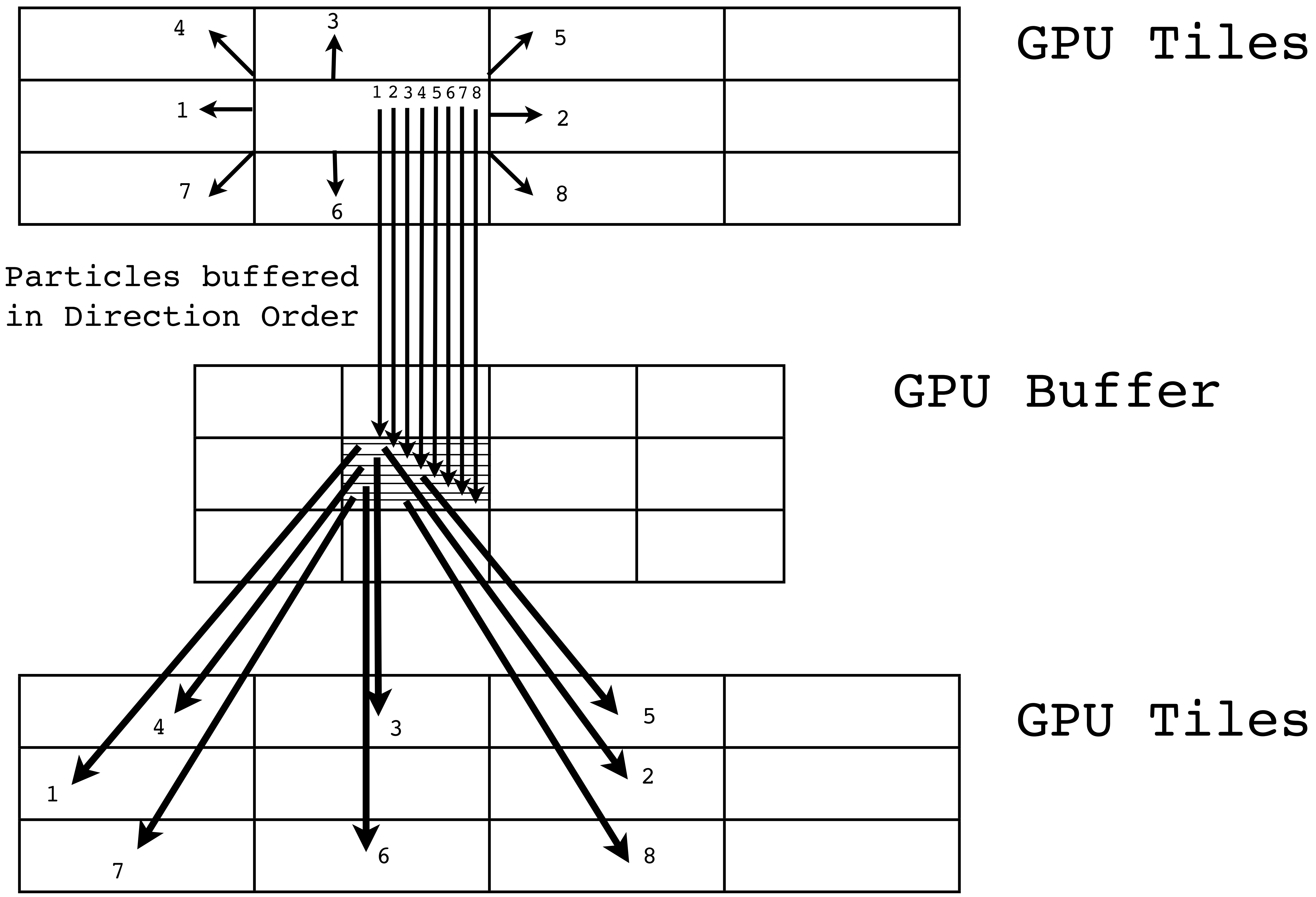
Three steps:
1. Create a list of particles which are leaving a tile, and where they are going
2. Using list, each thread places outgoing particles into an ordered buffer it controls
3. Using lists, each tile copies incoming particles from buffers into particle array

- Less than a full sort, low overhead if particles already in correct tile
- Can be done in parallel
- **Essentially message-passing, except particle buffer contains multiple destinations**

In the end, the particle array belonging to a tile has no gaps
- Incoming particles are moved to any existing holes created by departing particles
- If holes still remain, they are filled with particles from the end of the array

Straightforward to implement with one thread per tile
- Much more complex with multiple threads per tile

GPU Particle Reordering



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

2D ES Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision

GPU algorithm also implemented in OpenMP

```
Hot Plasma results with dt = 0.1
             CPU:Intel i7   GPU:Fermi M2090  OpenMP(12 CPUs)
Push            22.1 ns.       532 ps.          1.678 ns.
Deposit          8.5 ns.       227 ps.          0.818 ns.
Reorder          0.4 ns.       115 ps.          0.113 ns.
Total Particle  31.0 ns.       874 ps.          2.608 ns.

The time reported is per particle/time step.
The total particle speedup on the Fermi M2090 was 35x compared to 1 CPU
Field solver takes an additional 6% on GPU, 11% on CPU.
```

In 16 years, one GPU card is about 15 times the speed of a 128 processor T3E
Cost: ~7 x 10^{-8} cents per step

A GPU is a card which can fit in your desktop

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

2-1/2D EM Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision

GPU algorithm also implemented in OpenMP

```
Hot Plasma results with dt = 0.04, c/vth = 10, relativistic
              CPU:Intel i7    GPU:Fermi M2090  OpenMP(12 cores)
Push             66.5 ns.        0.426 ns.          5.645 ns.
Deposit          36.7 ns.        0.918 ns.          3.362 ns.
Reorder           0.4 ns.        0.698 ns.          0.056 ns.
Total Particle  103.6 ns.        2.042 ns.          9.062 ns.

The time reported is per particle/time step.
The total particle speedup on the Fermi M2090 was 51x compared to 1 CPU.
Field solver takes an additional 10% on GPU, 11% on CPU.
```

Designing New Particle-in-Cell (PIC) Algorithms: **Multiple GPUs**

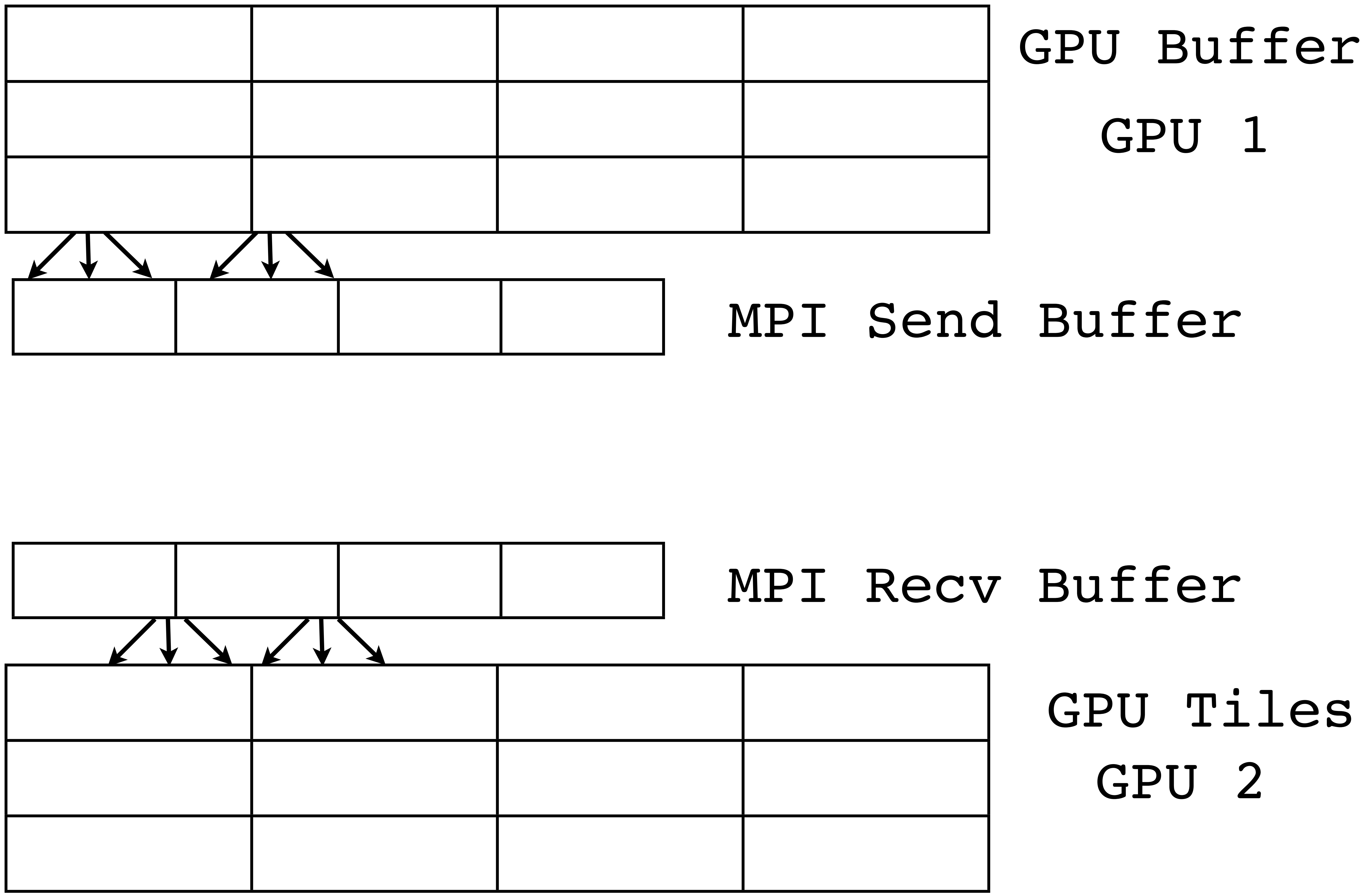
Multiple GPUs can be controlled with MPI
- Merge MPI and GPU algorithms
- We now have 3 levels of parallelism in the code, 3 levels of memory to manage

We started with existing 2D MPI codes from UPIC Framework
- Replacing MPI push/deposit with GPU version was no major challenge

With multiple GPUs, we need to integrate two different particle partitions
- MPI and GPU each have their own particle managers to maintain particle order

Only the first/last row or column of tiles on GPU interacts with neighboring MPI node
- Particles in row/column of tiles collected in MPI send buffer
- Table of outgoing particles are also sent
- Table is used to determine where incoming particles must be placed

GPU-MPI Particle Reordering



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

2-1/2D EM Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision. Fermi M2090 GPU

Hot Plasma results with dt = 0.04, c/vth = 10, relativistic

	CPU:Intel i7	1 GPU	2 GPUs	3 GPUs
Push	66.5 ns.	0.422 ns.	0.211 ns.	0.141 ns.
Deposit	36.7 ns.	1.284 ns.	0.645 ns.	0.432 ns.
Reorder	0.4 ns.	0.690 ns.	0.346 ns.	0.232 ns.
Total Particle	103.6 ns.	2.398 ns.	1.205 ns.	0.806 ns.

The time reported is per particle/time step.
The total speedup on the 3 Fermi M2090s compared to 1 core was 129x,
Speedup on 3 M2090s compared to 1 M2090 was 2.98x

FFT has not yet been optimized on multiple GPUs

I : OSIRIS CUDA

Sample Osiris CUDA Performance Measurements

- **Thermal Plasma with given temperature**

	Cold Plasma	T=1Kev	T=5Kev	T=10Kev
Fermi M2090	1.706 ns	1.727 ns	1.847 ns	1.934 ns
Kepler K20c	1.381 ns	1.344 ns	1.457 ns	1.529 ns

- Simulation performed on a single GPU of indicated type. Each simulation has 1536x1536 spatial cells (cell size ~ 1 Debye Length) with 36 particles per cell (~89 million particles total). Times are quoted in nanoseconds per particle per time step. Calculation uses single precision floating point numbers with linearly interpolated grid quantities.

- **Multi-Node MPI Performance (Weak Scaling)**

  Below we show the nanoseconds per particle per time step per node as we vary the number of MPI nodes from 1 to 225. Each MPI has one M2090 GPU. We are simulating a 1Kev thermal plasma..

	1	2	3	12	30	45	72	96	150	192	225
	1.72	1.72	1.76	1.82	1.83	1.82	1.85	1.84	1.84	1.82	1.84

- Each node has 1536x1536 spatial cells (cell size ~ 1 Debye Length) containing 42 particles per cell (so there are ~99 million particles on each node). Times are quoted in nanoseconds per particle per time step per node. Calculation uses single precision floating point numbers with linearly interpolated grid quantities.

UCLA Particle-in-Cell and Kinetic Simulation Software Center (PICKSC), NSF funded
Goal is to provide and document parallel Particle-in-Cell (PIC) and kinetic codes.

Planned activities

- Provide parallel skeleton codes for various PIC codes on traditional and new parallel hardware and software systems.
- Provide MPI-based production PIC codes that will run on desktop computers, mid-size clusters, and the largest parallel computers in the world.
- Provide key components for constructing new parallel production PIC codes for electrostatic, electromagnetic, and other codes.
- Provide interactive codes for teaching of important and difficult plasma physics concepts
- Facilitate benchmarking of kinetic codes by the physics community, not only for performance, but also to compare the physics approximations used
- Documentation of best and worst practices, which are often unpublished and get repeatedly rediscovered.
- Provide some services for customizing software for specific purposes

Key components and codes will be made available through standard open source licenses and as an open-source community resource, contributions from others are welcome.

Parallel Plasma PIC Codes Available https://idre.ucla.edu/hpc/parallel-plasma-pic-codes

Skeleton Particle-in-Cell (PIC) Codes: These skeleton codes contain the critical algorithms used in various PIC codes, but do not contain the wide range of inputs, diagnostics, and outputs normally found in such codes. Their main purpose is to teach students how such codes are structured. They can also be used for benchmarking purposes and as a starting point in evaluating new computer architetures and languages. Most of the examples are in Fortran and C.

Basic Serial Codes:
1. 2D Serial Electrostatic Spectral code (pic2): Download the source gzipped tarball- pic2.tar.gz
2 2-1/2D Serial Electromagnetic Spectral code (bpic2): Download the source gzipped tarball- bpic2.tar.gz

Basic Parallel Codes -Level 1: These families of codes illustrate how to implement PIC codes with one level of parallelism.

The first family uses a tiling technique designed for shared memory processors with OpenMP:

1. 2D Parallel Electrostatic Spectral code (mpic2): Download the source gzipped tarball- mpic2.tar.gz
2. 2-1/2D Parallel Electromagnetic Spectral code (mbpic2): Download the source gzipped tarball- mbpic2.tar.gz

The second family uses domain decomposition designed for distributed memory processors with MPI:

1. 2D Parallel Electrostatic Spectral code(ppic2): Download the source gzipped tarball- ppic2.tar.gz
2. 2-1/2D Parallel Electromagnetic Spectral code(pbpic2): Download the source gzipped tarball- pbpic2.tar.gz

Basic Parallel Codes -Level 2: These families of codes illustrate how to implement PIC codes with two levels of parallelism.

The first family uses CUDA C on the NVIDIA GPU, with a tiling technique for each thread block, and with SIMD vectorization within a block.

1. 2D Parallel Electrostatic Spectral code(gpupic2): Download the source gzipped tarball- gpupic2.tar.gz
2. 2-1/2D Parallel Electromagnetic Spectral code(gpubpic2): Download the source gzipped tarball- gpubpic2.tar.gz

The second family uses a hybrid MPI/OpenMP scheme, with a tiling technique on each shared memory multi-core node using OpenMP, and domain decomposition connecting such nodes.

1. 2D Parallel Electrostatic Spectral code(mppic2): Download the source gzipped tarball- mppic2.tar.gz
2. 2-1/2D Parallel Electromagnetic Spectral code(mpbpic2): Download the gzipped tarball- mpbpic2.tar.gz

Conclusions

PIC Algorithms on emerging architectures are largely a combination of previous techniques

- Vector techniques from Cray
- Blocking techniques from cache-based architectures
- Message-passing techniques from distributed memory architectures
- Data parallel techniques (vectorization with array syntax)

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

Scheme should be portable to architectures with similar hardware abstractions

Further information available at:

V. K. Decyk, and T. V. Singh, "Particle-in-Cell Algorithms for Emerging Computer Architectures," Computer Physics Communications, 185, 708 (2014).

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