A Portable Approach for PIC on Emerging Computer Architectures

Viktor K. Decyk

UCLA

Acknowledgments:
T. V. Singh, UCLA (GPUs)
Ricardo Fonseca, ISCTE (Vectorization)
Henry Gardner, ANU (Design Patterns, F2003)

Overview

Emerging Exascale architectures support 3 levels of parallelism:

- Distributed Memory Parallelism (eg, MPI)
- Shared Memory Parallelism (eg, OpenMP, pthreads,CUDA)
- Vectorization (eg, Intel vector intrinsics, compiler vectorization,CUDA)

Efficient use of such computers requires effective use of all three

Designing algorithms for each level requires different approaches

It is possible to express the algorithm in a generic way, but

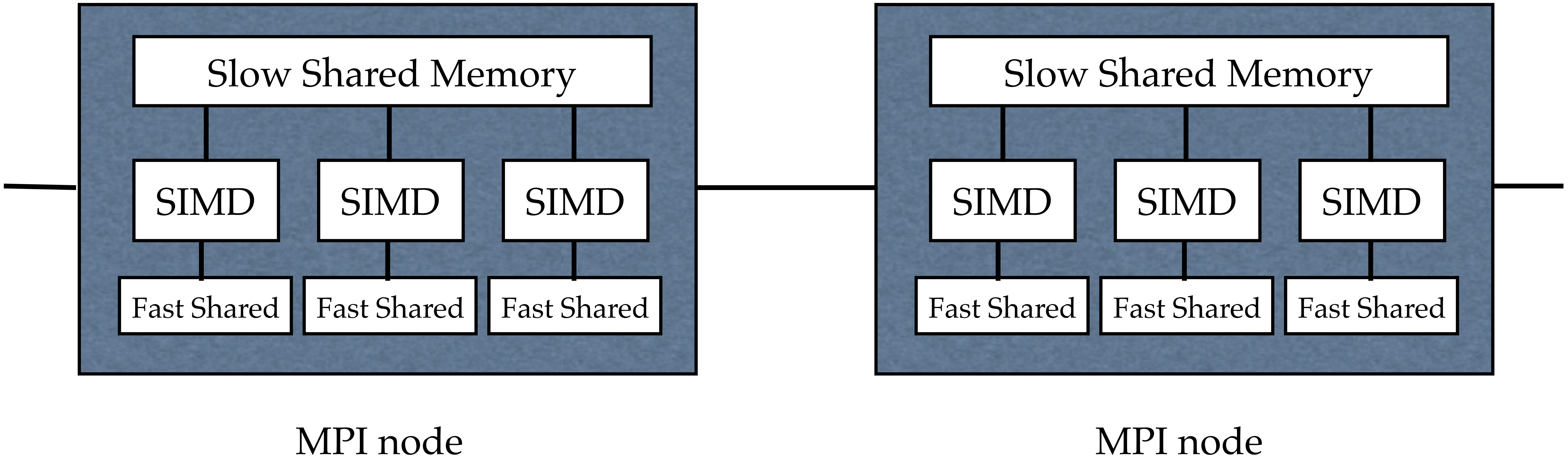
- Different implementations may to required for different architectures
- How to express parallel programming concepts is still evolving

Fortran2003 has new capabilities to make this easier

- Interoperability with C
- Abstract data types
- Make use of design patterns such as Strategy, Factory, Adapter, etc.

Overview

Simple Hardware Abstraction for Next Generation Supercomputer



NVIDIA GPU

Intel Phi

PIC Codes: General Approach

Start with the outermost, coarse-grained level: Distributed Memory Parallelism

- Typically using MPI
- Most common approach is domain decomposition
- Most PIC codes already do this

Next move to the middle level: Shared Memory Parallelism

- Typically using OpenMP
- Decomposition into small tiles
- Not very common

Finally, move to the innermost, fine-grained level: Vectorization

- Typically using compiler directives
- Old technique resurrected

Data structures will change as one proceeds

PIC Codes: Distributed Memory Parallelism

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

Has been used for a long time:

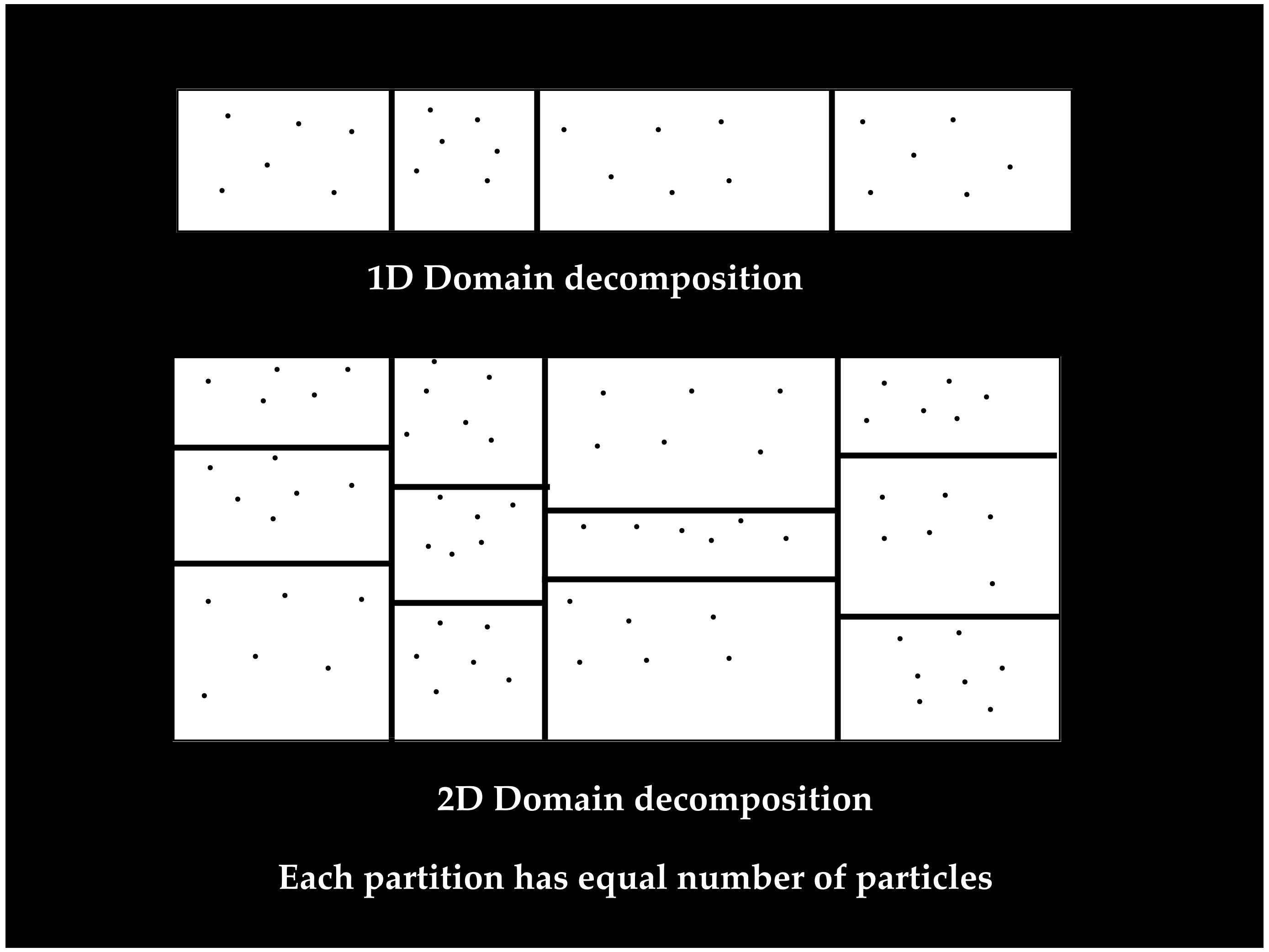
P. C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes," J. Computational Phys. 85, 302 (1989)

Sample MPI skeleton codes (mini-apps) can be found at UCLA's PICKSC web site:
http://picksc.idre.ucla.edu/software/skeleton-code/software-skeleton-code-mpi/

Particle data structure:
```
real part(idimp,npmax)
```
- `idimp` = number of particle coordinates, `npmax` = maximum number of particles

Distributed Memory Programming for PIC

Domain decomposition with MPI

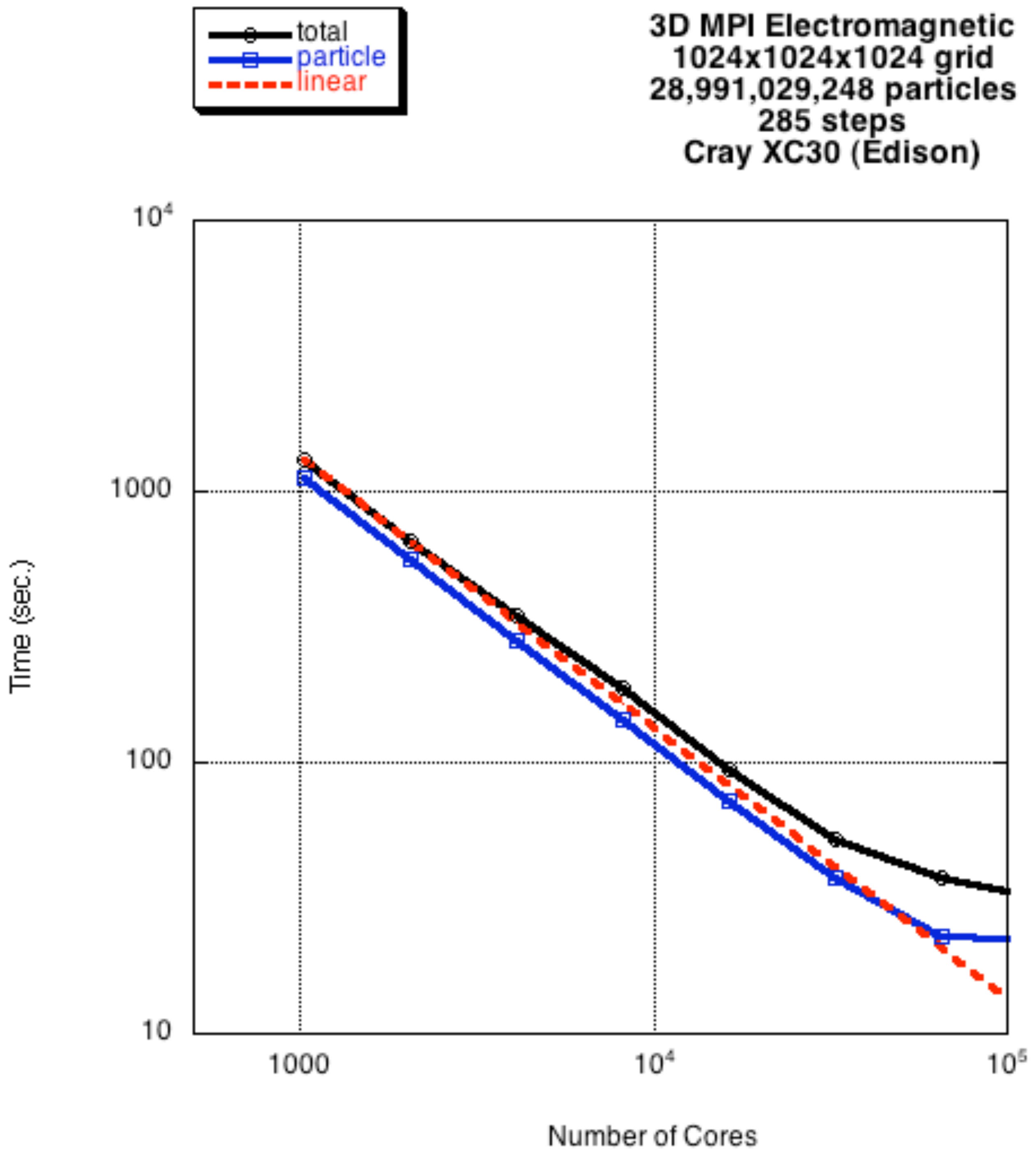


Primary Decomposition has non-uniform partitions to load balance particles

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

Particle Manager responsible for moving particles

- Particles can move across multiple nodes to implement dynamic repartitioning



Strong scaling of 3D MPI Spectral Electromagnetic code

Codes available at: http://picksc.idre.ucla.edu/software/skeleton-code/

PIC Codes: Shared Memory Parallelism

Initially use OpenMP on physical node (later can use CUDA or other language)

Biggest challenge is charge/current deposit:

- Different threads may attempt to deposit to the same grid location

Scalable solution is to partition data into small tiles

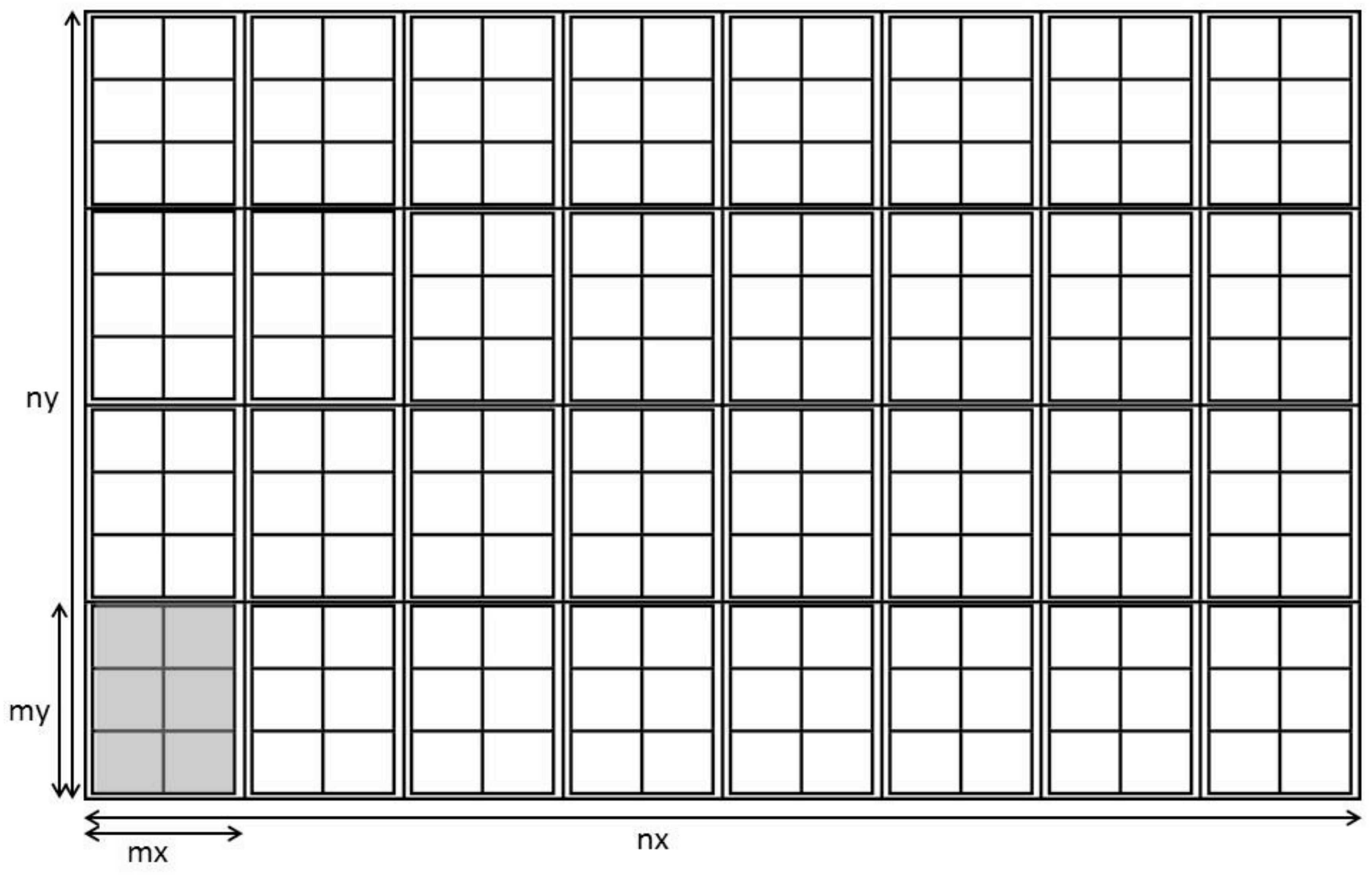
- Small enough to fit into L1 cache or GPU shared memory
- Micro-domain decomposition, separate from coarse-grained decomposition
- Re-order every time step

A fast reordering scheme is crucial

New particle data structure:

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

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



PIC Codes: Shared Memory Parallelism

Using micro-domains works much better than expected

- Use more tiles than cores
- Detailed load balance not always necessary (work load will often average out)
- Manual load balance simple (adjust how many tiles each thread controls)
- On GPUs, memory accesses can be pipelined with many threads outstanding

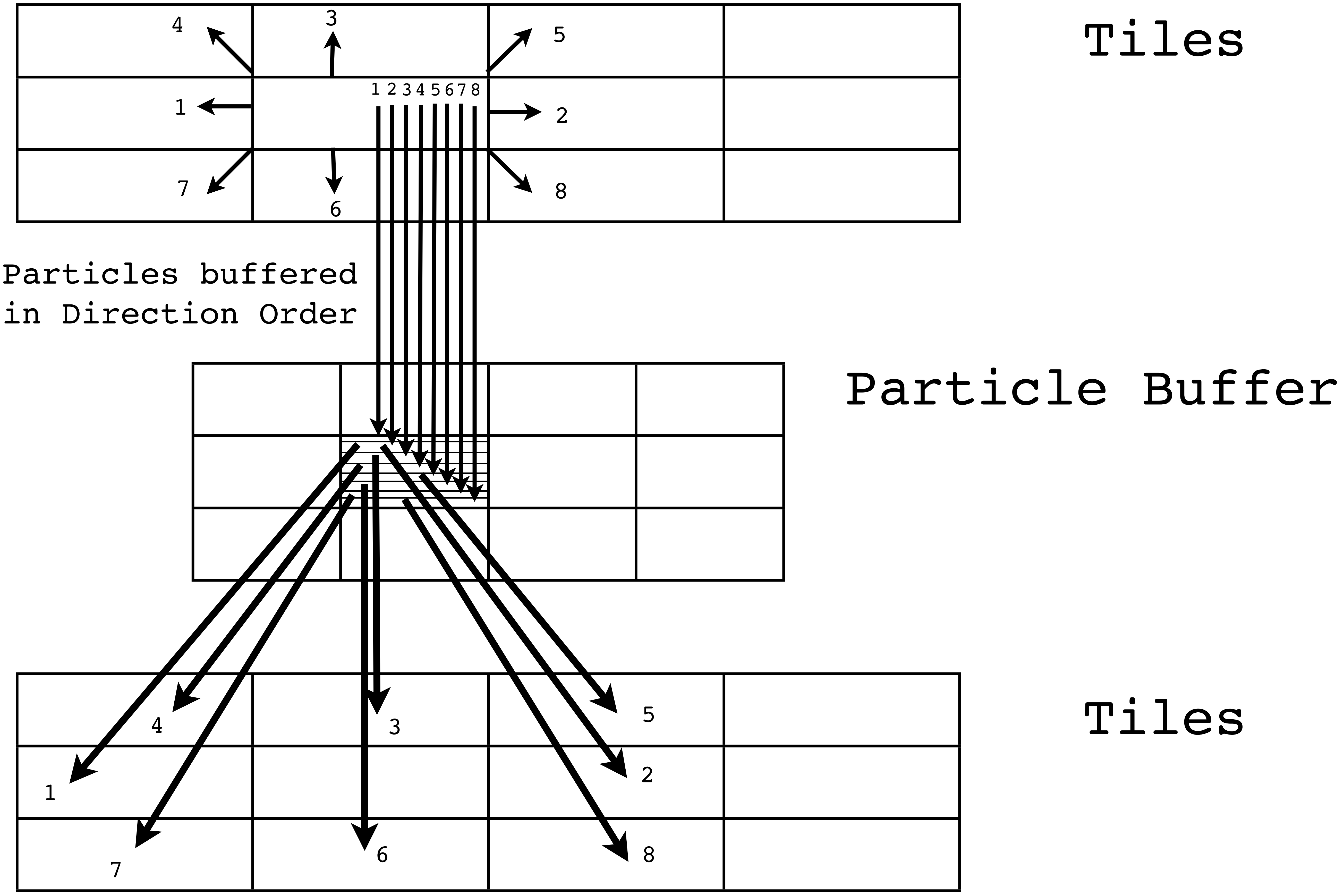
PIC Codes: Shared Memory Parallelism: Maintaining Particle Order

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 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

Particle Reordering in 2D



Evaluating New Particle-in-Cell (PIC) Shared Memory Algorithms: **Electromagnetic Case**

3D EM Benchmark with 128x128x128 grid, 56,623,104 particles, 36 particles/cell
Tile size = 8x8x8. Single precision, Dawson2 Intel i7 cluster

```
Hot Plasma results with dt = 0.035, c/vth = 10, relativistic
              CPU:Intel i7    OpenMP(12 cores)
Push             90.0 ns.        8.61 ns.
Deposit          61.6 ns.        6.09 ns.
Reorder           1.3 ns.        0.26 ns.
Total Particle  152.9 ns.       14.96 ns.

The time reported is per particle/time step.
```

Particle reordering time is not large

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), available at http://dx.doi.org/10.1016/j.cpc.2013.10.013.

Codes available at: http://picksc.idre.ucla.edu/software/skeleton-code/

PIC Codes: Shared and Distributed Memory Parallelism

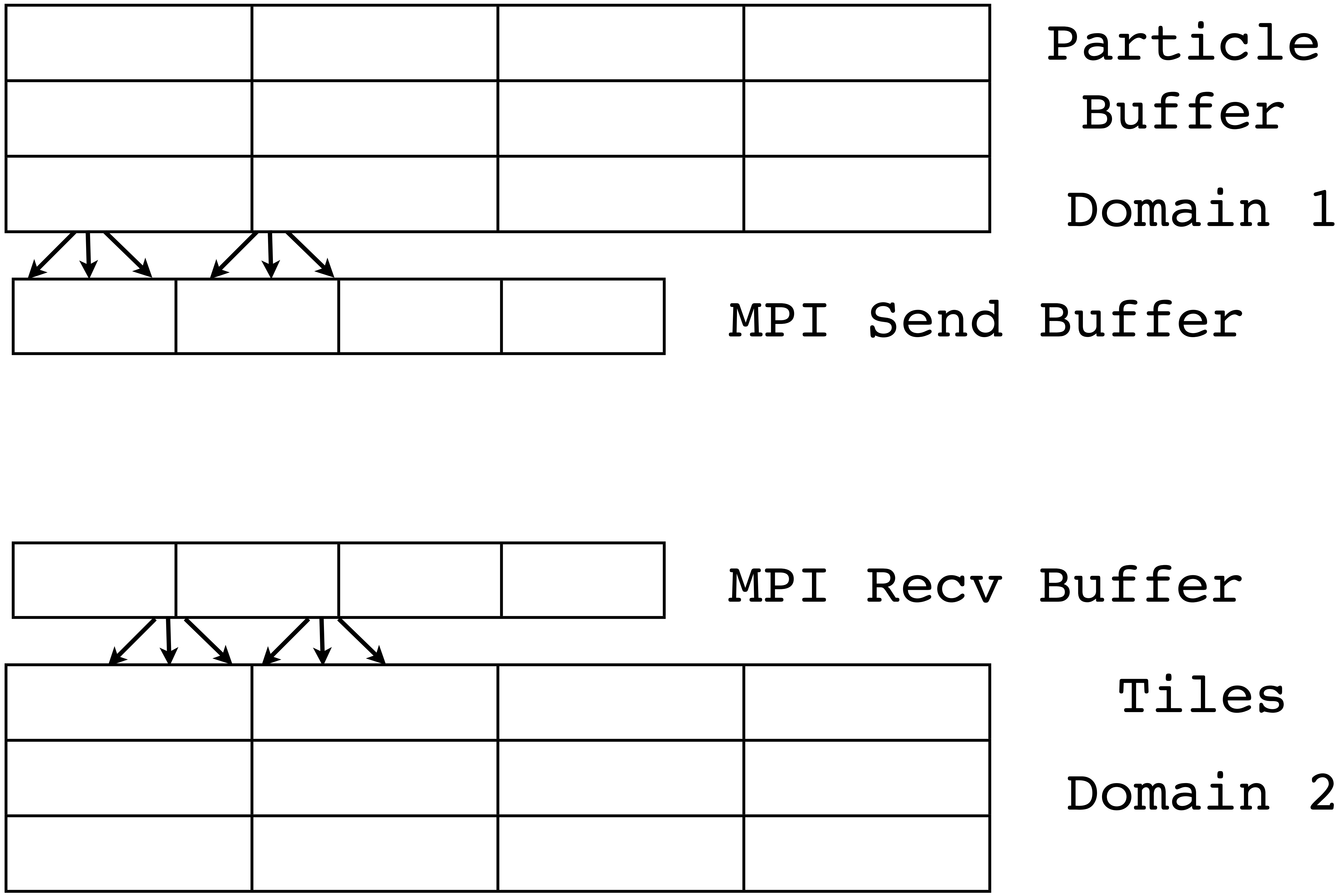
Typically use one or two MPI nodes per physical node
- OpenMP on physical node or on NUMA node

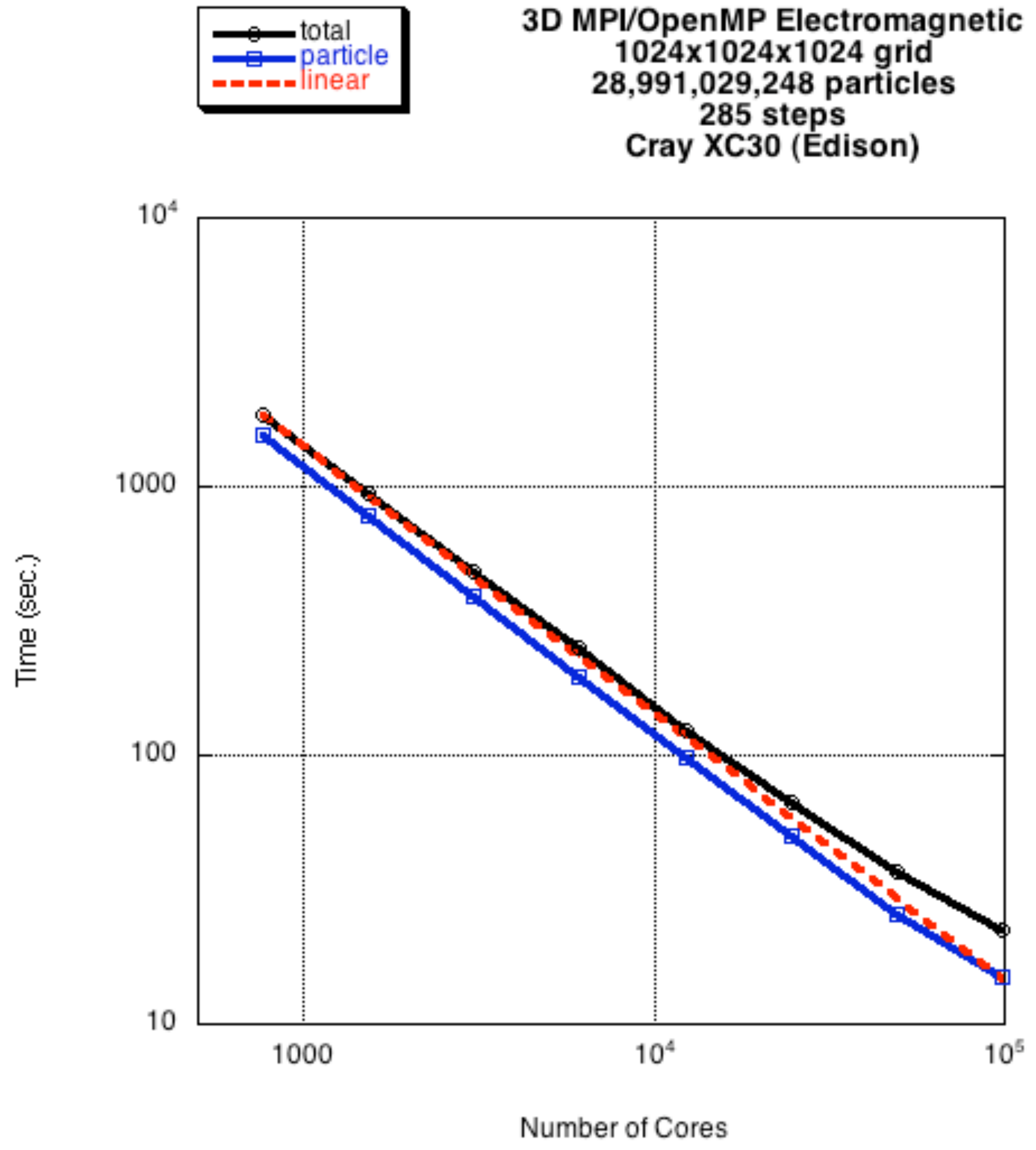
Biggest challenge is merging tile sorting and MPI particle manager:
- Particles first reordered by tile
- Then skim off particles at the edges going to another MPI node
- Need to send a table of which directions particles are going along with particles
- Particles can then be directly inserted into proper tile

New particle data structure:
```
real ppart(idimp,npmax,num_tiles_per node)
```

Straightforward with 1D MPI partition, complex with 2D or 3D MPI partitions

OpenMP-MPI Particle Reordering





Strong scaling of 3D MPI/OpenMP Spectral Electromagnetic code

Codes available at: http://picksc.idre.ucla.edu/software/skeleton-code/

PIC Codes: Vectorization

Vectorization today refers to making use of low level data parallel instructions
- Reading, processing, writing blocks of data simultaneously
- Can be done either with special vector intrinsics or compiler generated

Old idea resurrected (used on Cray machines in the 1980s)
- Data parallel vector intrinsics work well, but requires special skills and not portable
- Intel compiler directives currently get modest speedups, but improving
- Some old Cray code will not vectorize

Vectorize each OpenMP thread independently
- Outer loop OpenMP, inner loop vectorized

Biggest challenge is again charge/current deposit:
- Different vector elements may attempt to deposit to the same grid location

New particle data structure is transposed to enable efficient reading of blocks of data:

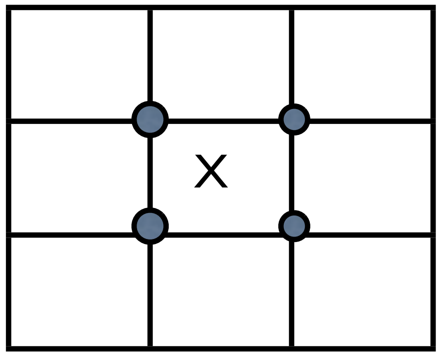
```
real ppart(npmax,idimp,num_tiles)
```

Vector Programming for PIC

- vectorizable means elements can be processed in any order
- **Deposit algorithm** uses long and short vectors to avoid data collisions

```
   dimension part(idimp,nop), q(nxv*ny)  ! nop=total particles
   dimension nn(4,npp), amxy(4,npp)      ! npp=particles in block

   do j = 1, npp         ! loop over blocks of particles
      n = part(1,j+joff)
      m = part(2,j+joff)
      dxp = qm*(part(1,j+joff) - real(n))
      dyp = part(2,j+joff) - real(m)
      amx = qm - dxp
      amy = 1.0 - dyp
      n = n + nxv*m + 1
! store locations nn and weights amxy for deposit
      nn(1,j) = n
      nn(2,j) = n + 1
      nn(3,j) = n + nxv
      nn(4,j) = n + nxv + 1
      amxy(1,j) = amx*amy
      amxy(2,j) = dxp*amy
      amxy(3,j) = amx*dyp
      amxy(4,j) = dxp*dyp
   enddo
! deposit charge
   do j = 1, npp         ! loop over blocks of particles
cdir$ ivdep
      do i = 1, 4        ! loop over distinct points
      q(nn(i,j)) = q(nn(i,j)) + amxy(i,j)
      endo
   enddo
```



PIC Codes: Vectorization

Sample code for Intel SSE2 vector intrinsics:

```
/* loop over particles in tile in groups of 4 */
      for (j = 0; j < nps; j+=4) {
/* find interpolation weights */
/*       x = ppart[j+npoff];       */
/*       y = ppart[j+nppmx+npoff]; */
         v_x = _mm_load_ps(&ppart[j+npoff]);
         v_y = _mm_load_ps(&ppart[j+nppmx+npoff]);
/*       nn = x; */
/*       mm = y; */
         v_nn = _mm_cvttps_epi32(v_x);
         v_mm = _mm_cvttps_epi32(v_y);
/*       dxp = qm*(x - (float) nn); */
         v_dxp = _mm_sub_ps(v_x,_mm_cvtepi32_ps(v_nn));
         v_dxp = _mm_mul_ps(v_dxp,v_qm);
/*       dyp = y - (float) mm; */
         v_dyp = _mm_sub_ps(v_y,_mm_cvtepi32_ps(v_mm));
/*       nn = nn - noff + mxv*(mm - moff); */
         v_nn = _mm_sub_epi32(v_nn,v_noff);
         v_mm = _mm_sub_epi32(v_mm,v_moff);
         v_it = _mm_mul_epu32(v_mxv,_mm_srli_si128(v_mm,4));
         v_mm = _mm_mul_epu32(v_mm,v_mxv);
         v_mm = _mm_add_epi32(v_mm,_mm_slli_si128(v_it,4));
         v_nn = _mm_add_epi32(v_nn,v_mm);
/*       amx = qm - dxp;   */
/*       amy = 1.0f - dyp; */
         v_amx = _mm_sub_ps(v_qm,v_dxp);
         v_amy = _mm_sub_ps(v_one,v_dyp);
/* calculate weights, for lower left/right, upper left/right */
         a = _mm_mul_ps(v_amx,v_amy);
         b = _mm_mul_ps(v_dxp,v_amy);
         c = _mm_mul_ps(v_amx,v_dyp);
         d = _mm_mul_ps(v_dxp,v_dyp);
         _mm_store_si128((__m128i *)ll,v_nn);
```

PIC Codes: Vectorization

Key idea is that grid points written by a single particle are guaranteed to be different

- These grid points are simultaneously written as vectors, one particle at a time

Source codes available at:
Sample OpenMP/Vector skeleton codes (mini-apps) can be found at:
http://picksc.idre.ucla.edu/software/skeleton-code/openmpvectorization/

PIC Codes: Controlling Variations

Fortran 2003 now has abstract data types

- Allows one to write more complex code in a manageable way

The important idea is write what is invariant in a generic (abstract) way,
and encapsulate what varies in specific (concrete) classes

PIC Codes: Controlling Variations

Here is an abstract type useful for either for an electrostatic or electromagnetic PIC code:

```
   type, abstract :: emf
   contains
      procedure(ajdeposit), deferred :: jdeposit
      procedure(aqdeposit), deferred :: qdeposit
      procedure(apoisson), deferred :: poisson
      procedure(amaxwell), deferred :: maxwell
      procedure(apush), deferred :: push
   end type
!
   abstract interface
      subroutine ajdeposit(this,particles,j)
! deposit current
      import :: emf
      implicit none
      class (emf), intent(in) :: this
      real, dimension(:,:), intent(inout) :: particles
      real, dimension(:,:,:), intent(inout) :: j
      end subroutine
      ...
   end interface
```

PIC Codes: Controlling Variations

Here is a concrete class useful for an electrostatic code:

```
      module es_class
! class for electrostatic type of plasma simulation
      use emf_class
      implicit none
!
      type, extends(emf) :: es
      contains
         procedure :: jdeposit => es_jdeposit
         procedure :: qdeposit
         procedure :: poisson
         procedure :: maxwell => es_maxwell
         procedure :: push => es_push
      end type
!
      contains
!
      subroutine qdeposit(this,particles,q)
! deposit charge
      class (es), intent(in) :: this
      real, dimension(:,:), intent(in) :: particles
      real, dimension(:,:), intent(inout) :: q
      ...
      end subroutine

      .....
      end module
```

PIC Codes: Controlling Variations

Similarly, one can implement a concrete class for an electromagnetic code

Here is an abstract procedure that can be used with any kind of emf class
• it knows nothing about the various concrete classes

```
      subroutine step_simul(this,particles,q,j,e,et,b)
! advance plasma one time step
      class (emf), intent(in) :: this
      real, dimension(:,:), intent(inout) :: particles
      real, dimension(:,:), intent(inout) :: q
      real, dimension(:,:,:), intent(inout) :: j, e, et, b
      call this%jdeposit(particles,j)
      call this%qdeposit(particles,q)
      call this%poisson(q,e)
      call this%maxwell(j,e,et,b)
      call this%push(particles,e,b)
      end subroutine
```

The main code looks like this:

```
      type (es), target :: es_sim
      type (em), target :: em_sim
      class (emf), pointer :: sim
   !
   ! set PIC simulation type to electromagnetic
      sim => em_sim
   ! run simulation one time step
      call step_simul(sim,particles,q,j,e,et,b)
```

PIC Codes: Status of current research

OpenMP/MPI layers works well

Still experimenting with Intel compiler based vectorization strategies

All three layers implemented on GPUs with CUDA

- OpenACC does not work well on GPUs with PIC at this time

GPUs and Intel PHI each have strengths and weaknesses

- On GPU everything must be vectorized (sorting very painful)
- On Intel both scalar and vector processors can be used together
- GPUs can handle many more threads simultaneously
- GPUs have fast hardware atomic operations