



National Energy Research  
Scientific Computing Center



U.S. DEPARTMENT OF  
**ENERGY**

Office of  
Science

# Miniapps and Proxy Apps for Fusion/Plasma Physics and Practical Programming Studies using NPBs



Alice Koniges, Jean-Luc Vay, Rebecca Yuan  
(LBNL)

Viktor Decyk, (UCLA)

Stephane Ethier and Weixing Wang (PPPL)

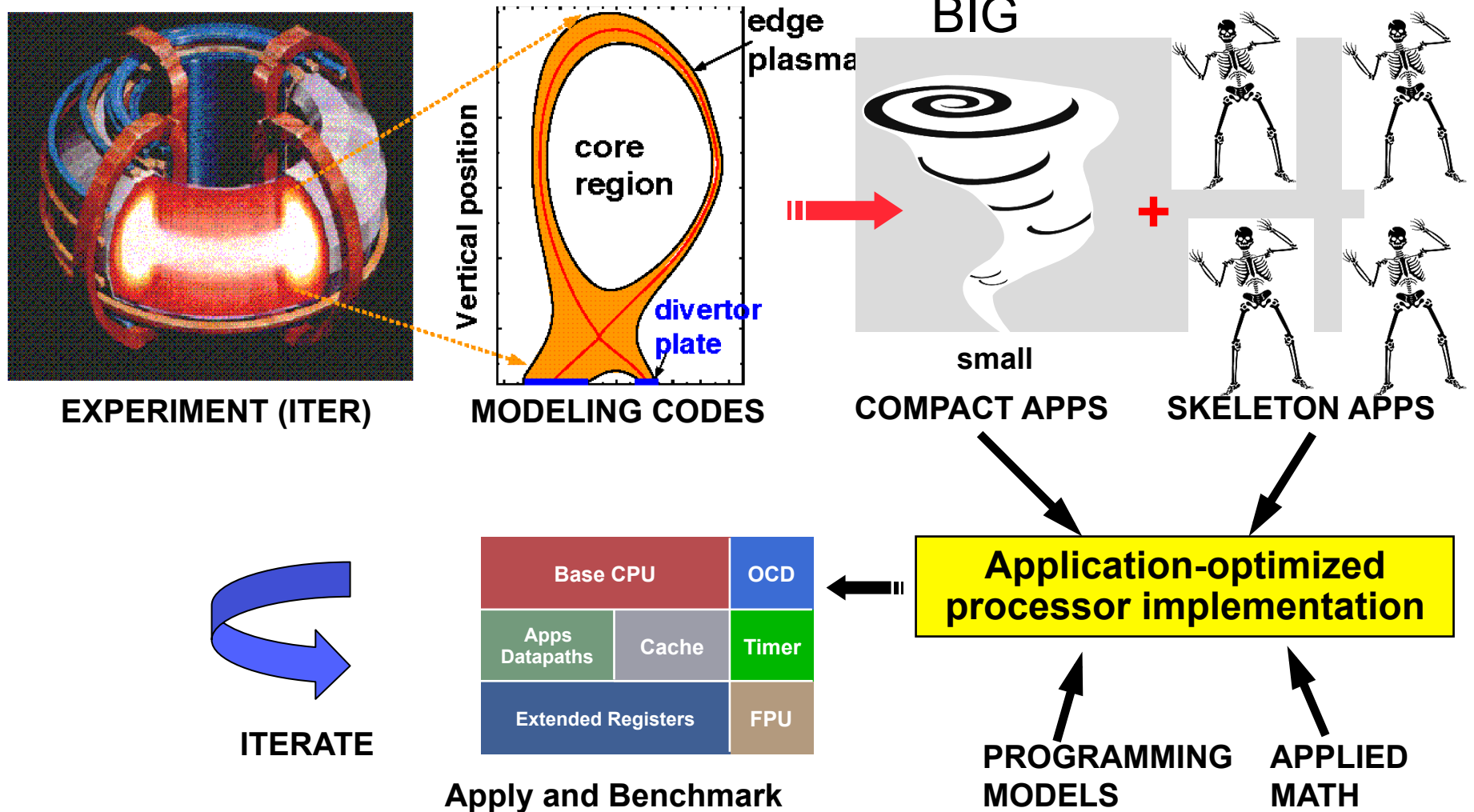
Berkeley UPC Group

**FROM: Using Application Proxies for Co-design of Future HPC  
Computer Systems and Applications**

**SC12 Full Day Tutorial**

**Michael A. Heroux, Alice E. Koniges, David F. Richards,  
Richard F. Barrett, Thomas Brunner**

# Value of Proxy Apps



# We will study examples of Compact and Skeleton Apps Relevant to Fusion

- **Example Compact App: PIC Electrostatic, GTC\_simple**
  - Reduced I/O, No diagnostics, Possible reduced physics and dimensionality
- **Example Skeleton App: Shifter**



# Fusion Energy\* is the answer to our energy future

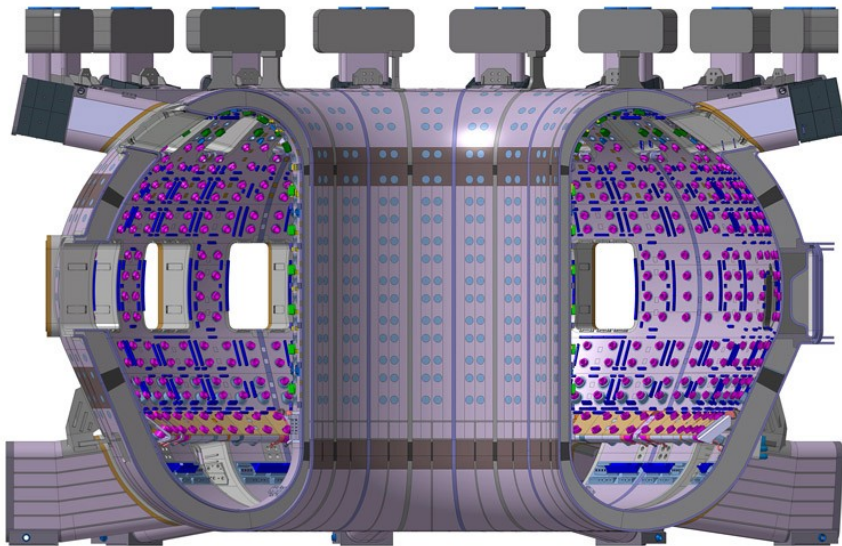
- One gallon of sea water same energy as 300 gallons gasoline
- Fifty cups of water equals 2 tons of coal
- Fusion power plant produces no climate changing gases
- Dramatically lower radioactive bi-products than fission plants
- No danger of runaway reaction or core melt down
- Two major approaches: Magnetic and Inertial, each with own issues
- Simulation is key to saving big \$\$ per discharge/shot
- Extreme range of time/space scale for magnetic fusion is challenging
  - e.g., there are 14 orders of magnitude difference between the electron cyclotron and discharge time scales



**\*still have not figured out exactly how**

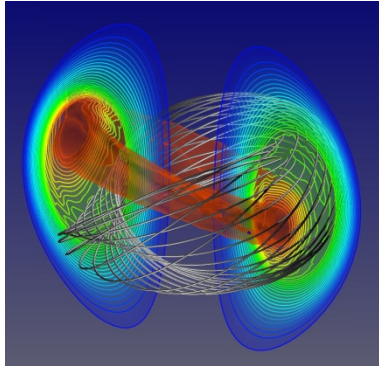


**ITER, currently under construction in the South of France, aims to demonstrate that fusion is an energy source of the future**

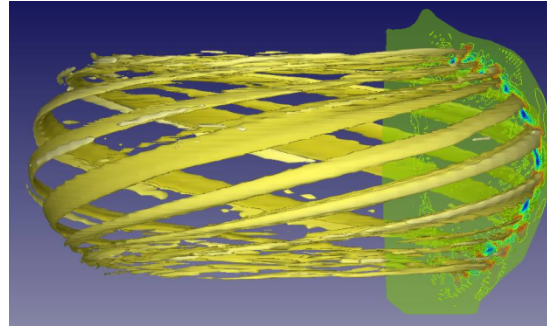


- **Top-to-bottom exascale computer design is essential for efficient design/operation of large-scale experiments**
  - Typical ITER discharge can be estimated at 1M\$

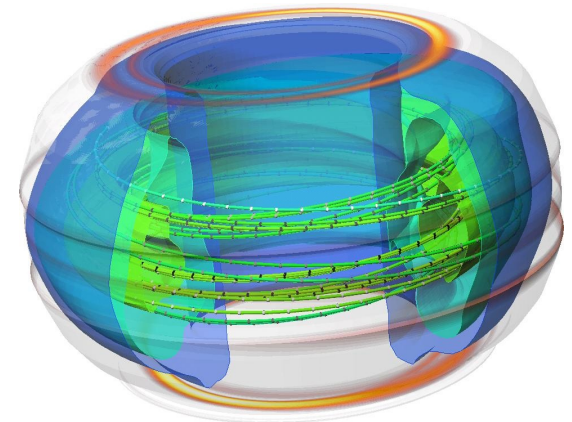
# Magnetic fusion codes control instabilities and other plasma phenomena critical to ITER



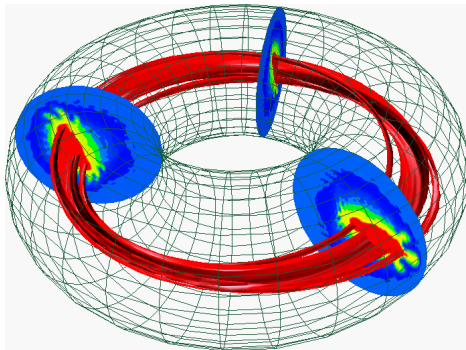
“sawtooth oscillations”



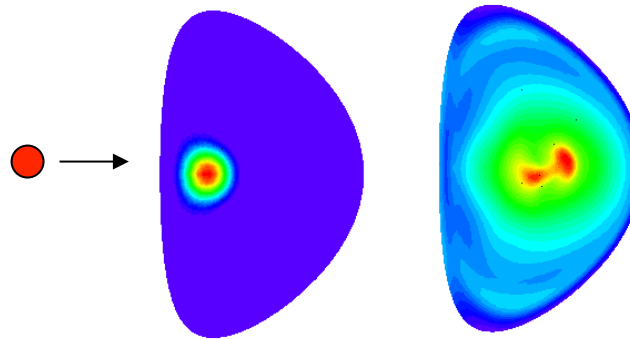
Edge Localized Modes



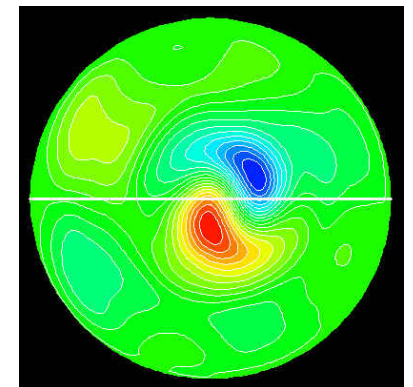
Disruption forces,  
RE, and heat loads  
during disruption



Disruptions caused by short  
wave-length modes interacting  
with helical structures.



Mass redistribution  
after pellet injection



Interaction of high-  
energy particles with  
global modes

Slide: Steve Jardin, PPPL

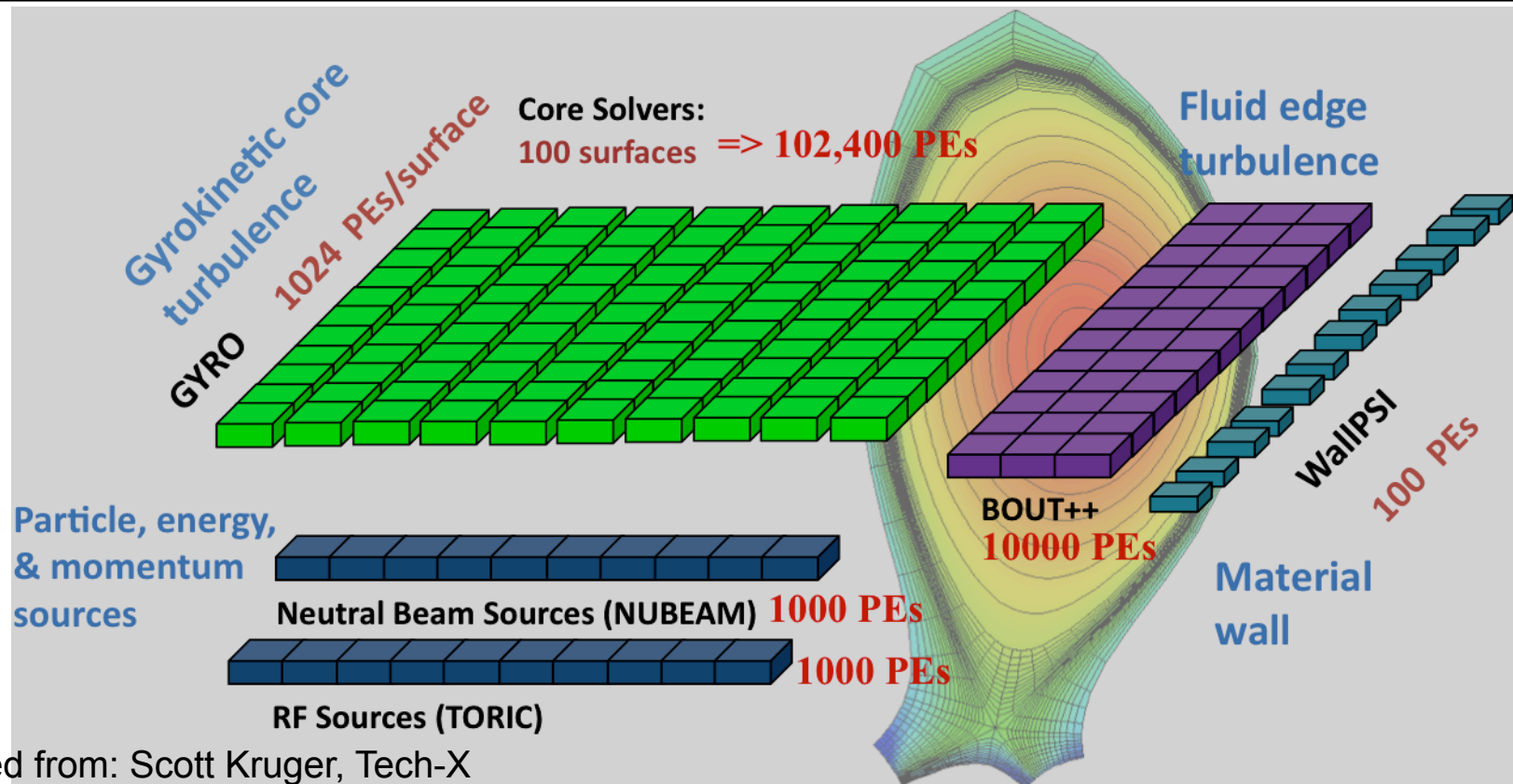
# **The exascale machine design for fusion must enable a multiphysics approach**

- **Achieving exascale, may not mean just scale current codes**
- **Exascale machines must be sufficiently well-balanced to handle phenomena on a variety of scales through ingenious use of heterogeneous architectures and interconnects, and storage facilities to support real design studies**
  - GPU's accelerating FE quadrature in MHD codes
  - Novel programming languages such as CAF using fast on-node communication patterns in PIC routines
  - Fault-tolerant-aware numerical algorithms for 100,000's of cores
  - Integrated simulations at many scales connected and tested via simulator glue



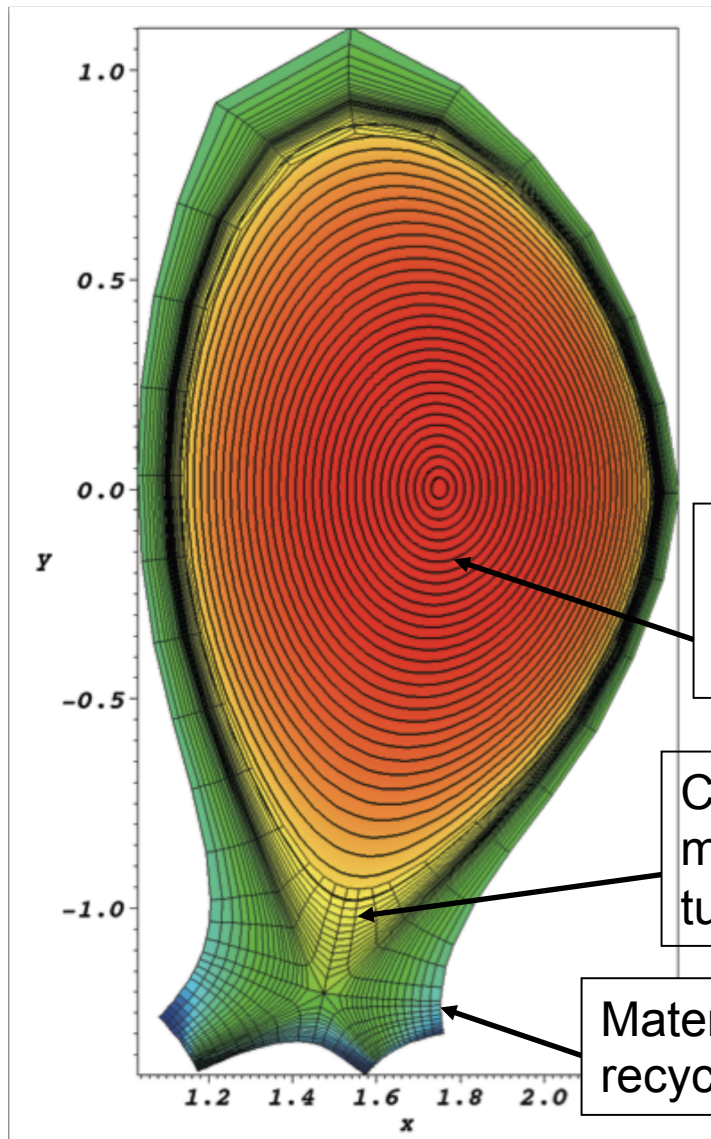
# A representative suite of tokamak models includes a variety of temporal and spatial discretization schemes

- Core Transport: GYRO/NEO
- Collisional Edge Plasma: BOUT++
- MHD: M3D-C1, NIMROD
- Explicit PIC Modeling: GTS, VORPAL
- Wave heating, Wall interaction



Adapted from: Scott Kruger, Tech-X

# What is the most common proxy app for fusion? Answer: PIC Codes



- Coupling on short time scales
- Inter-processor and in-memory communication
- Implicit coupling

Hot central plasma: nearly completely ionized, magnetic lines lie on flux surfaces, 3D turbulence embedded in **1D** transport

Cooler edge plasma: atomic physics important, magnetic lines terminate on material surfaces, 3D turbulence embedded in **2D** transport

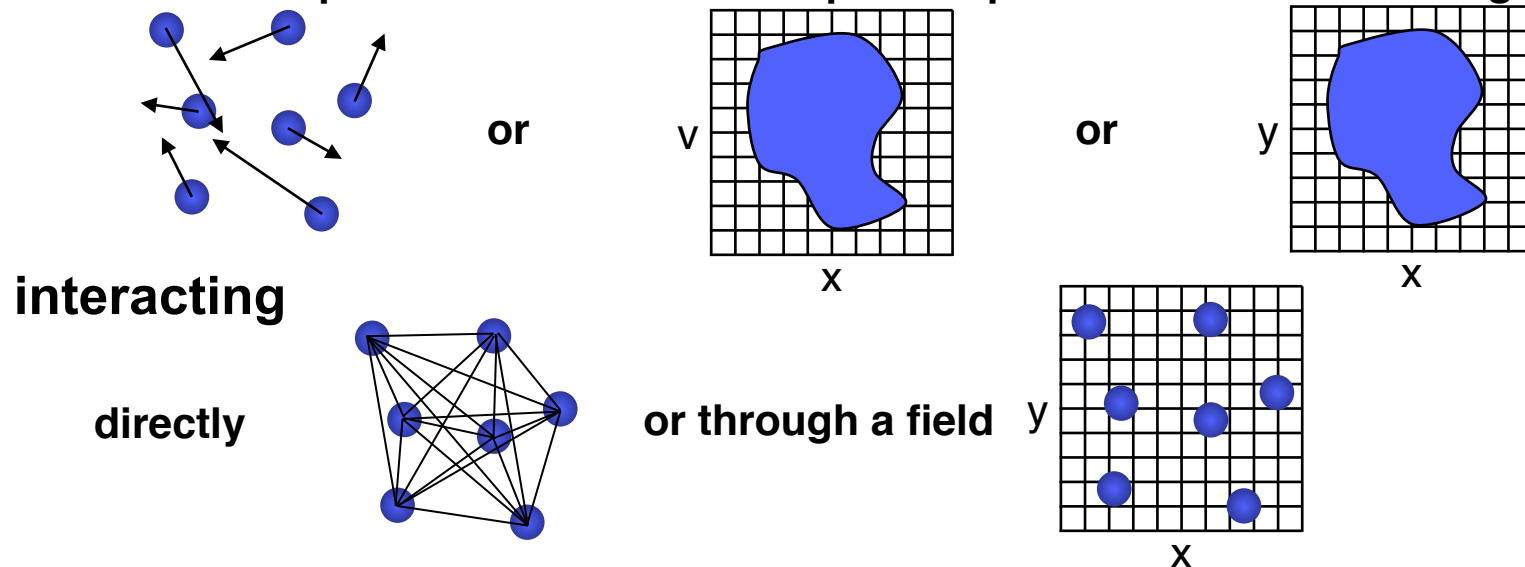
Material walls, embedded hydrogenic species, recycling

# Modeling of a plasmas – possibilities

- **PLASMA: Collection of a large number of interacting charged particles**
  - **Particles** mathematically described by
    - Lagrangian approach: evolution of singularities
      - ✿ Klimontovitch eq.
    - Eulerian approach: evolution of an incompressible fluid
      - ✿ in phase-space: Boltzmann/Fokker-Planck eq. (collisions), Vlasov eq. (no collisions)
      - ✿ in real space: fluid/MHD eq.
  - **Interactions** mathematically described by
    - Lagrangian approach: sum from all singularities, instantaneous or with retardation
    - Eulerian approach: fields
      - ✿ instantaneous: Poisson
      - ✿ with retardation: Maxwell

# Modeling of a plasmas - classification

- In summary, the modeling of a plasma implies the modeling of  
a collection of particles      fluid cells in phase-space      fluid cells in configuration space



- The numerical integration leads to further splitting
  - Partial differential equations: finite-differences/volumes/elements, Monte-Carlo, semi-Lagrangian,
  - Time integration: explicit/implicit,
  - Direct interaction: direct summation, multipole expansion (tree-codes),
  - ...

# Modeling of a plasmas

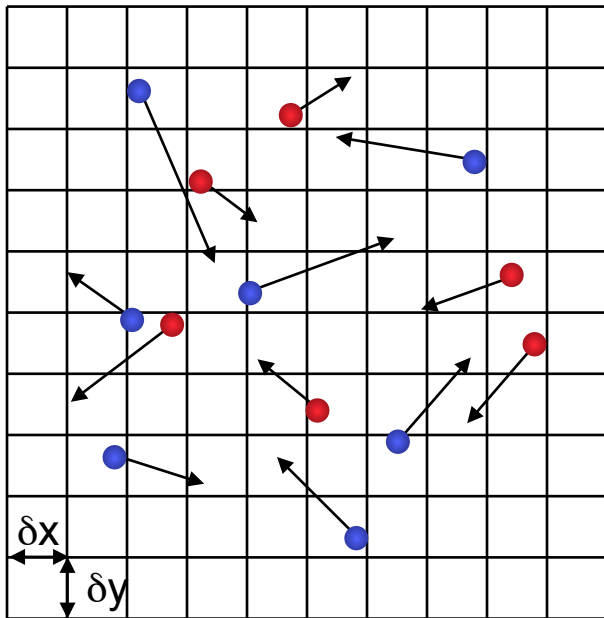
- All these methods have in common that they must update the status of  $N$  quantities (particle/fluid/field quantities) from time  $t$  to time  $t+\Delta t$
- One of the most scalable methods of plasma modeling is PIC Codes, and thus is the basis for Exascale-aiming Proxy Apps
- PIC codes differ in the basic equation they are solving, the grid, solver methods, etc.
- We will consider both a simple geometry electrostatic PIC code, and a complex toroidal geometry PIC code



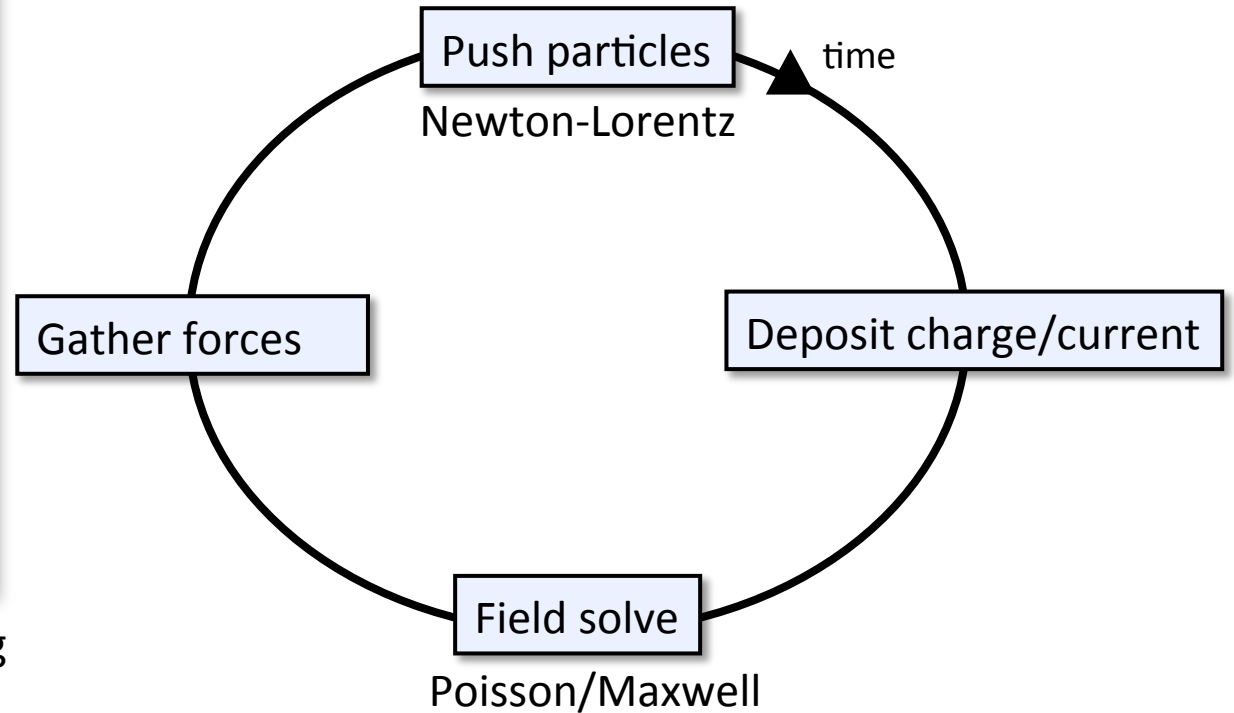
# The Particle-in-Cell (PIC) method for Plasma Simulation

- "particle-in-cell" because plasma macro-quantities(  
number density, current density, etc.) are assigned to simulation particles (i.e., the *particle weighting*)
- Particles can live anywhere on the domain, but field and macro-quantities are calculated only on the mesh points
- Inter-particle forces of less than a grid cell are smoothed
  - Particles are like clouds, because they can pass through each other
- **Basic steps:**
  - Integration of the equations of motion.
  - Interpolation of charge and current source terms to the field mesh.
  - Computation of the fields on mesh points (field solve)
  - Interpolation of the fields from the mesh to the particle locations.
- **PIC codes differ from Molecular Dynamics in use of fields on a grid rather than direct binary interactions**
  - This also adds the requirement of a field solve

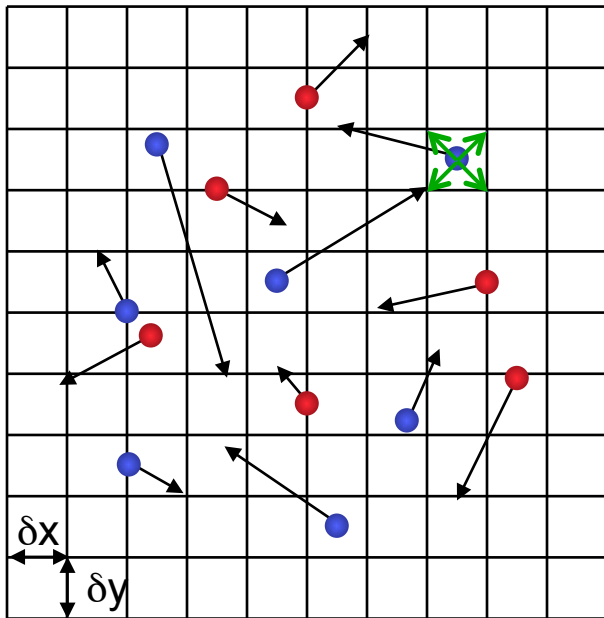
# Particle-In-Cell workflow



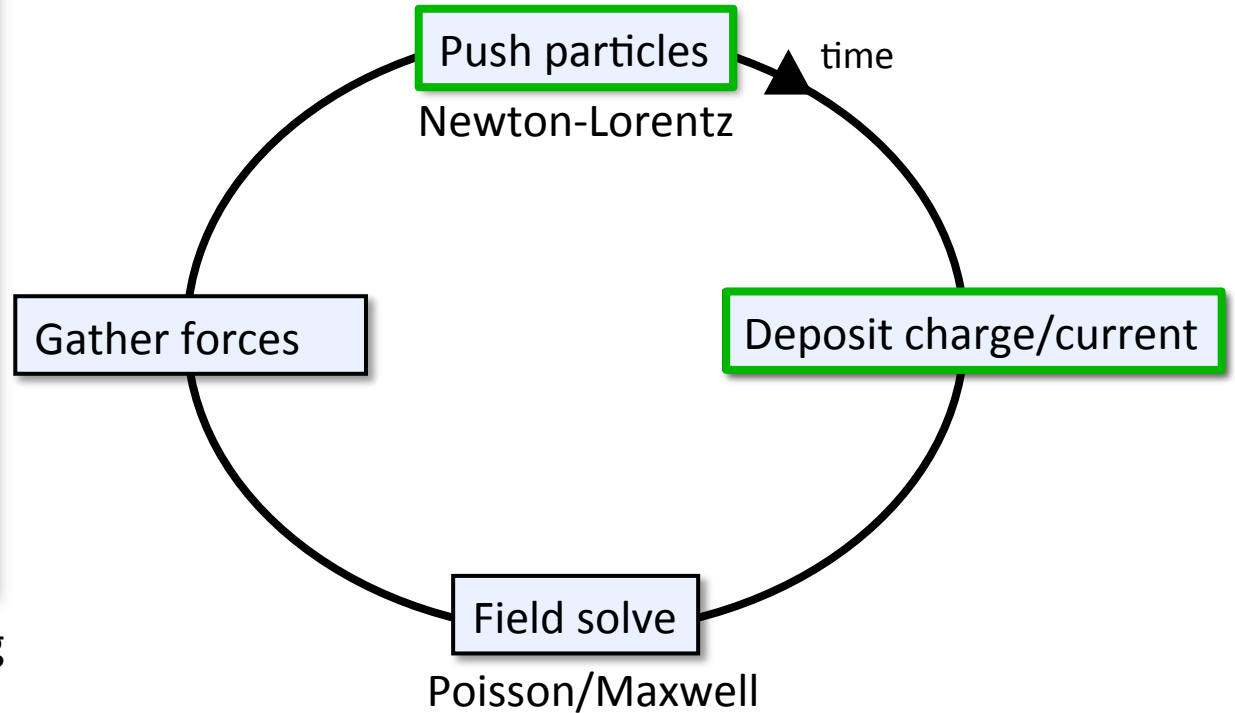
Plasma=collection of interacting charged particles



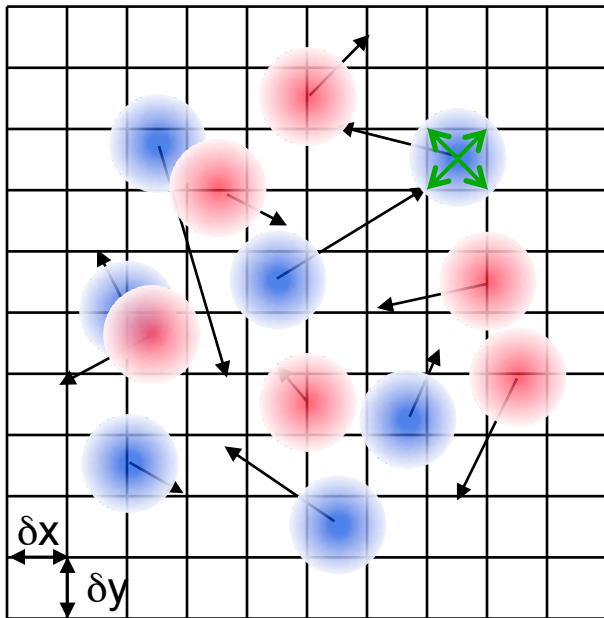
# Particle-In-Cell workflow



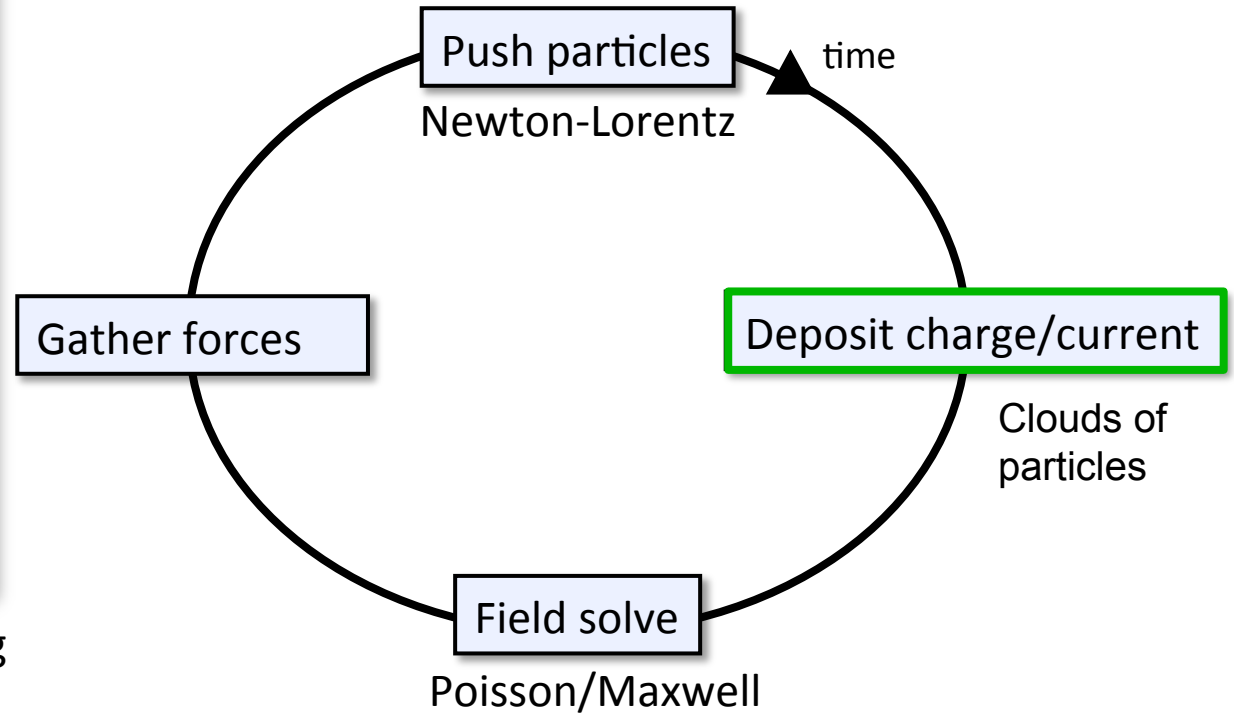
Plasma=collection of interacting charged particles



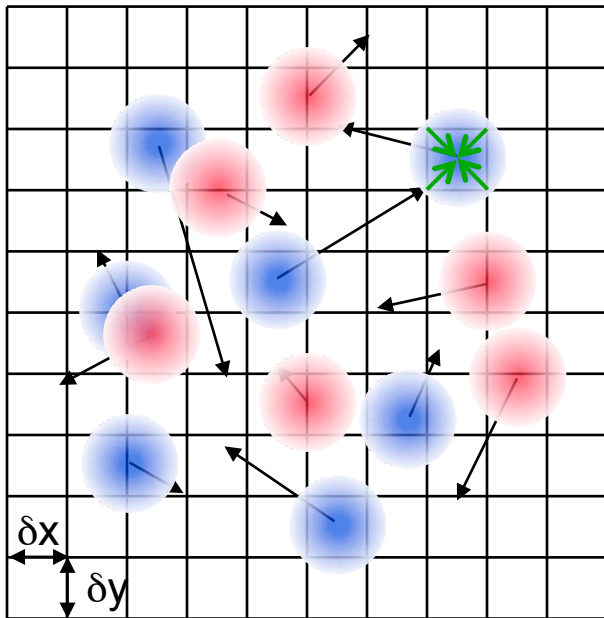
# Particle-In-Cell workflow



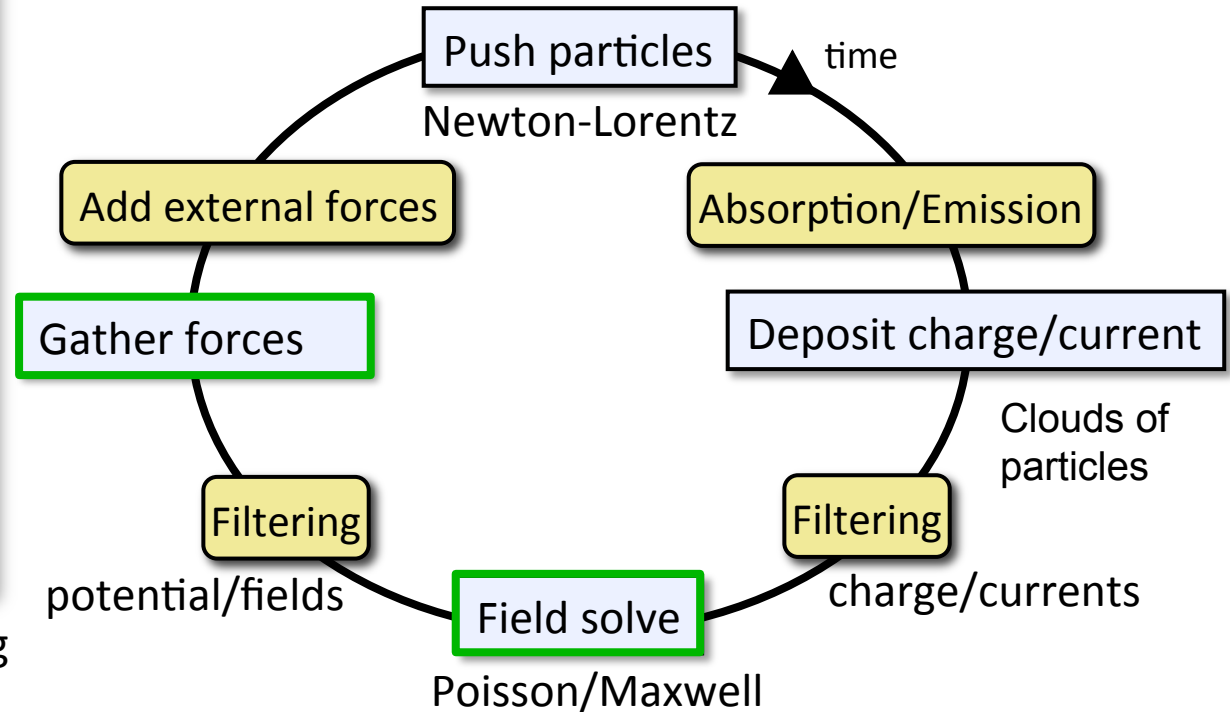
Plasma=collection of interacting charged particles



# Particle-In-Cell workflow



Plasma=collection of interacting charged particles



+ **filtering** (charge,currents and/or potential,fields).

+ **absorption/emission** (injection, loss at walls, secondary emission, ionization, etc),

+ **external forces** (accelerator lattice elements),

- **Classic Textbooks:**

- Plasma Physics via Computer Simulation by C. K. Birdsall and A. B. Langdon
- Computer Simulation Using Particles by R. W. Hockney and J. W. Eastwood

# A Simple Electrostatic Plasma Code

## Viktor K. Decyk, UCLA

- **Calculate charge density on a mesh from particles**
  - Scatter Operation is use to distribute a particles charge onto nearby grid locations

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

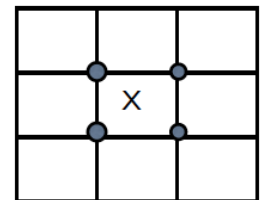
- **Solve Poisson's Equation (this is what makes it “electrostatic”)**

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

- **Advance the particles' co-ordinates using Newton's Law (force to move comes from the Electric Field, Newton's Law calculates velocity)**
  - Gather Operation (from interpolation) is used to get the approximate field value at the particle's location

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

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i$$



# From Viktor Decyk UCLA PIC Bootcamp



# Particle-in-Cell codes

Particle Push step uses equations of motion. Here, we see a typical Time-difference of eqns of motion: second order leap-frog scheme

$$\frac{d\mathbf{v}_i}{dt} = \frac{q_i}{m_i} \mathbf{E}_s \approx \frac{\mathbf{v}_i(t + \Delta t / 2) - \mathbf{v}_i(t - \Delta t / 2)}{\Delta t} = \frac{q_i}{m_i} \mathbf{E}_s(t)$$
$$\frac{d\mathbf{x}_i}{dt} \approx \frac{\mathbf{x}_i(t + \Delta t) - \mathbf{x}_i(t)}{\Delta t} = \mathbf{v}_i(t + \Delta t / 2)$$

Solution is explicit time advance:

$$\mathbf{v}_i(t + \Delta t / 2) = \mathbf{v}_i(t - \Delta t / 2) + \frac{q_i}{m_i} \mathbf{E}_s(\mathbf{x}_i(t)) \Delta t$$

$$\mathbf{x}_i(t + \Delta t / 2) = \mathbf{x}_i(t) + \mathbf{v}_i(t + \Delta t / 2) \Delta t$$

Note: time step should resolve plasma frequency

## 2D Electrostatic Skeleton PIC code: pic2

- Fortran and C versions available

Main data structures in Fortran

```
real, dimension(4,np) :: part      ! particle array
real, dimension(nx+2,ny+1) :: qe    ! charge density array
real, dimension(2,nx+2,ny+1) :: fxye ! electric field array
real :: wke, we                    ! kinetic, potential energies
```

Main iteration loop (C function names):

Charge Deposit

1. (cgpost2l) Deposit charge: update qe
2. (caguard2l) Add guard cells: update qe

Field Solver

3. (cwfft2rx) Transform charge to fourier space: update qe
4. (cpois22) Calculate force/charge in fourier space: update fxye, we
5. (cwfft2r2) Transform force to real space: update fxye

Particle Push

6. (ccguard2l) Copy guard cells with standard procedure: updates fxye
7. (cgpush2l) Push particles: update part, wke

Particle Sort

8. (cdsortp2yl) Occasionally sort particles by cell: update part

## Particle-in-Cell Codes

The two most important procedures are:

Particle Push: `cgpsh2l`

Charge Deposit: `cgpost2l`

Challenges in optimizing PIC codes

- Low computational intensity (2-3 FLOPs / memory access)
- 2D Electrostatic code has 55 FLOPs / particle update (11 for deposit, 34 for push)
- Memory access is largely irregular (gather / scatter pattern)

## Particle-in-Cell Codes

Particle coordinates are stored in grid units, so that if there are  $N_x$  grid points, the  $x$  coordinate lies within  $0 < x < N_x$ .

The particle data is stored in the array `part`, where

`part[n][0]` = position  $x$  of particle  $n$

`part[n][1]` = position  $y$  of particle  $n$

`part[n][2]` = velocity  $v_x$  of particle  $n$

`part[n][3]` = velocity  $v_y$  of particle  $n$

This code uses bi-linear interpolation, involving 4 nearest cells to the particle coordinate. The integer part of the coordinate indicates the leftmost cell the particle is in. The difference between the cell and the actual coordinate is the interpolation weight.

```

        for (j = 0; j < nop; j++) {
/* find nearest cells and interpolation weights */
            nn = part[idimp*j];
            mm = part[1+idimp*j];
            dxp = part[idimp*j] - (float) nn;
            dyp = part[1+idimp*j] - (float) mm;
            nn = 2*nn;
            mm = nxv2*mm;
            amx = 1.0 - dxp;
            mp = mm + nxv2;
            amy = 1.0 - dyp;
            np = nn + 2;
/* interpolate to find acceleration */
            dx = dyp*(dxp*fxy[np+mp] + amx*fxy[nn+mp])
                + amy*(dxp*fxy[np+mm] + amx*fxy[nn+mm]);
            dy = dyp*(dxp*fxy[1+np+mp] + amx*fxy[1+nn+mp])
                + amy*(dxp*fxy[1+np+mm] + amx*fxy[1+nn+mm]);
/* calculate new velocity */
            vx = part[2+idimp*j];
            vy = part[3+idimp*j];
            dx = vx + qtm*dx;
            dy = vy + qtm*dy;
/* calculate average kinetic energy */
            vx += dx;
            vy += dy;
            sum1 += vx*vx + vy*vy;
            part[2+idimp*j] = dx;
            part[3+idimp*j] = dy;
/* calculate new position */
            dx = part[idimp*j] + dx*dt;
            dy = part[1+idimp*j] + dy*dt;
        }

```

procedure cgpsh21

```

    for (j = 0; j < nop; j++) {
/* find nearest cells and interpolation weights */
        nn = part[idimp*j];
        mm = part[1+idimp*j];
        dxp = qm*(part[idimp*j] - (float) nn);
        dyp = part[1+idimp*j] - (float) mm;
        mm = nxv*mm;
        amx = qm - dxp;
        mp = mm + nxv;
        amy = 1.0 - dyp;
        np = nn + 1;
/* deposit charge */
        q[np+mp] += dxp*dyp;
        q[nn+mp] += amx*dyp;
        q[np+mm] += dxp*amy;
        q[nn+mm] += amx*amy;
    }

```

procedure cgpost2l

## Particle-in-Cell Codes

### Challenges in parallelizing PIC codes

- With domain decomposition, keeping field data and particle data together

Particle push is easier, all particles are independent

For charge deposit, we can have data collisions in accumulating density

- Two different particles can attempt to update the same density location simultaneously

## Particle-in-Cell Codes

2D Electrostatic PIC codes are contained in the file `pic2.tar.gz`

- Fortran and C versions both exist.

Particles are initialized with a uniform distribution in space, and a gaussian distribution in velocity space. The number of grids must be a power of 2.

The only diagnostic is field, kinetic and total energy. The initial and final values are printed out. Total energy should be approximately conserved.

The important inputs to the code are the following:

`indx` = exponent which determines length in x direction, where  $n_x = 2^{\text{indx}}$

`indy` = exponent which determines length in y direction, where  $n_y = 2^{\text{indy}}$

`npx` = initial number of particles distributed in x direction

`npv` = initial number of particles distributed in y direction

`tend` = time at end of simulation, in units of plasma frequency

`dt` = time interval between successive calculations, total number of steps =  $\text{tend} / \text{dt}$

`vtx/vty` = thermal velocity of electrons in x/y direction

`sortime` = number of time steps between electron sorting



## Particle-in-Cell Codes

The initial values are set in the code to be:

`indx = 9; indy = 9; npx = 3072; npy = 3072; sortime = 50`

`tend = 10.0, dt = 0.1; vtx = vty = vtz = 1.0`

The main programs are either `pic2.f` or `pic2.c`

The procedures are in the files, `push2.f` or `push2.c`

The Makefile is set up to use gcc and gfortran with Linux.

Executing `make` will compile both programs, `fpic2` and `cpic2`

You can also compile just one or the other, for example by executing  
`make cpic2`

Timings for the important procedures are calculated using the unix `gettimeofday` function.

More information about the mathematics behind these PIC codes are contained in the file `ESModels.pdf`

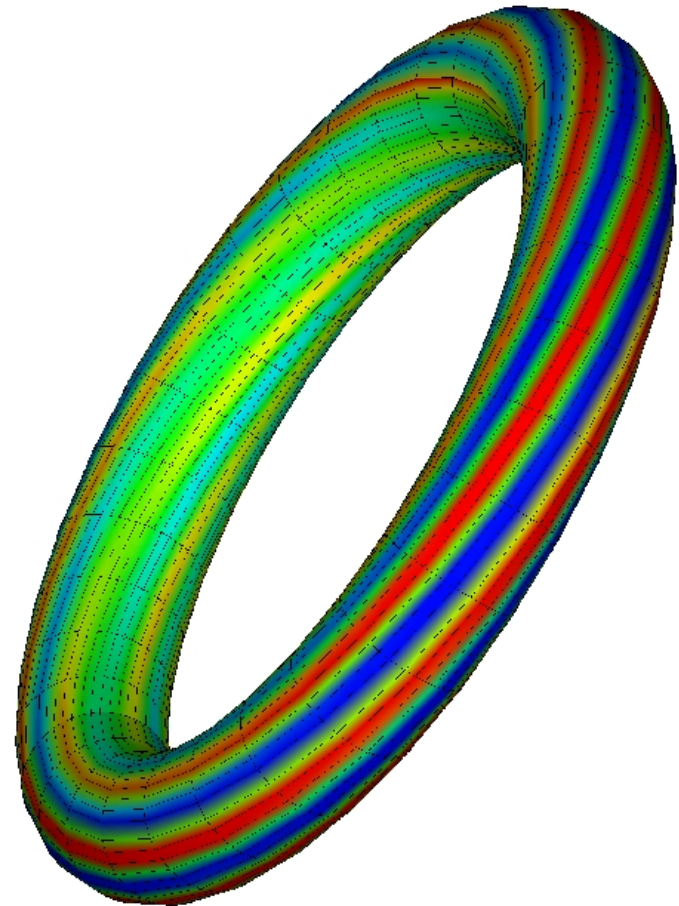
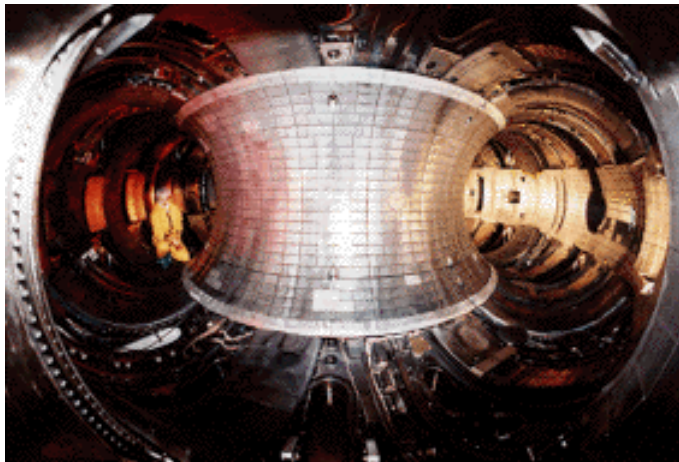
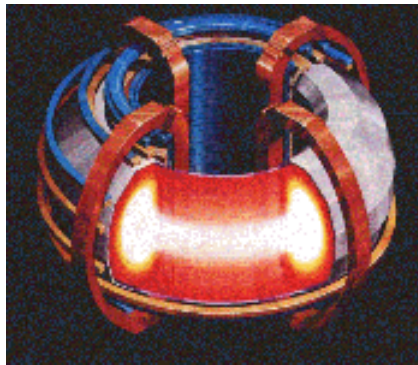
# Differences in Particle-In-Cell Codes

- **Particle-in-Cell codes are used for a wide variety of applications**
- **The family of codes known as GTC/GTS implements a Particle method to solve the Gyrokinetic Equations in Tokamaks and other toroidal fusion devices**
- **The first version of GTC was created by Zhihong Lin, currently of UCI. Versions of this code are released to the public**
  - <http://phoenix.ps.uci.edu/GTC/index.php>
- **Latter versions and important extensions include:**
  - GTS Stephane Ethier and Weixing Wang, PPPL
  - XGC CS Chang, PPPL
  - Versions based outside of US
- **The principal steps of GTC most often optimized by computer scientists are very similar**
- **Computer scientists generally work on a version of GTC obtainable from PPPL with permission or the version available by Lin**

# GTS (Gyrokinetic Tokamak Simulation)

- **GTS (Gyrokinetic Tokamak Simulation), uses PIC and the simulation particles are moved along the characteristics in phase space. This reduces the complex gyro-averaged Vlasov equation, a 5-dimensional partial differential equation, to a simple system of ordinary differential equations.**
- **Straight-field-line magnetic coordinates in toroidal geometry are employed since they are the natural coordinates for describing the tokamak magnetic equilibrium field**
  - accurate time-stepping -- even when a relatively low order method, such as second-order Runge-Kutta, is employed.
- **In PIC, a grid replaces the direct binary interaction between particles by accumulating the charge of those particles on the grid at every time step and solving for the electromagnetic field, which is then gathered back to the particles' positions.**

**Some of the complexity of GTC/GTS is due to the coordinate system for toroidal magnetic fusion devices**



# A Primary use of GTC: Turbulence in Fusion Plasmas

- **Turbulence is believed to be the mechanism for cross-field transport in magnetically confined plasmas:**
  - Size and cost of a fusion reactor determined by particle and energy confinement time and fusion self-heating.
- **Plasma turbulence is a complex nonlinear phenomenon:**
  - Large time and spatial scale separations similar to fluid turbulence.
  - Self-consistent electromagnetic fields: many-body problem
  - Strong nonlinear wave-particle interactions: kinetic effects.
  - Importance of plasma spatial inhomogeneities, coupled with complex confining magnetic fields, as drivers for microinstabilities and the ensuing plasma turbulence.

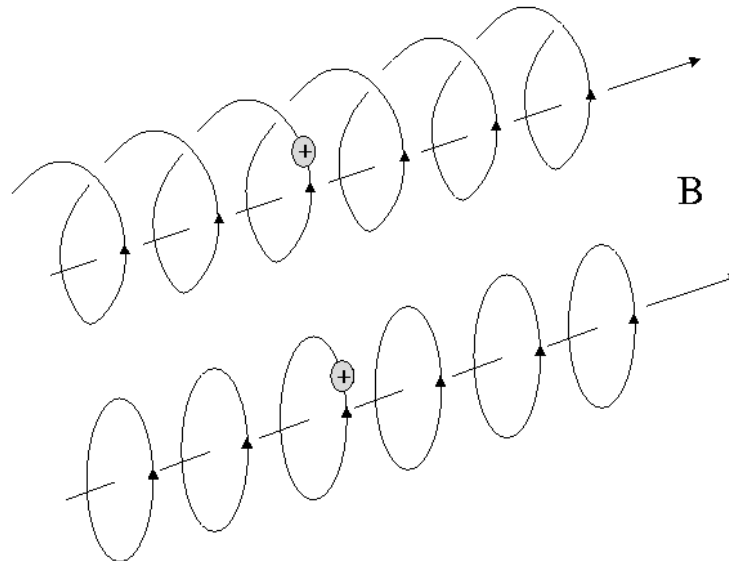
# The Gyrokinetic Toroidal Code GTC

- **Description:**

- Particle-in-cell code (PIC)
- Developed by Zhihong Lin (now at UC Irvine)
- Non-linear gyrokinetic simulation of microturbulence [Lee, 1983]
- Fully self-consistent
- Uses magnetic field line following coordinates  $(\psi, \theta, \zeta)$  [Boozer, 1981]
- Guiding center Hamiltonian [White and Chance, 1984]
- Non-spectral Poisson solver [Lin and Lee, 1995]
- Low numerical noise algorithm ( $\delta \mathbf{f}$  method)
- Full torus (global) simulation

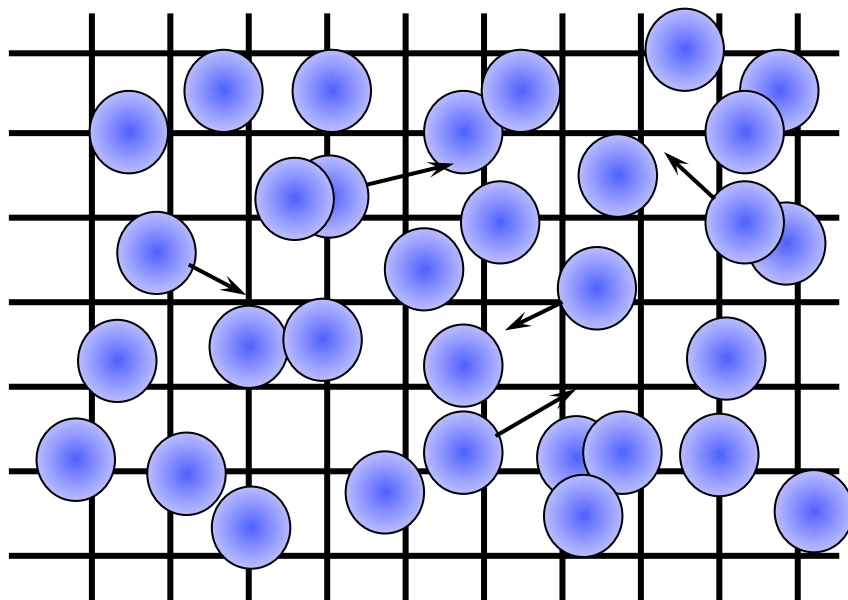
# Gyrokinetic approximation for low frequency modes

- Gyrokinetic ordering  $\frac{\omega}{\Omega} \sim \frac{\rho}{L} \sim \frac{e\phi}{T} \sim k_{\parallel}\rho \ll 1$   
 $k_{\perp}\rho \sim 1$
- Gyro-motion: guiding center drifts + charged ring
- Gyrophase-averaged 5D kinetic (Vlasov) equation



# Recall our basic particle-in-cell (PIC) method: now adapt for gyrokinetic

- Particles sample distribution function (markers).
- The particles interact via a grid, on which the potential is calculated from deposited charges.



## The PIC Steps

- “**SCATTER**”, or deposit, charges on the grid (nearest neighbors)
- Solve Poisson equation
- “**GATHER**” forces on each particle from potential
- Move particles (**PUSH**)
- Repeat...



# The particle moving equations are more complicated for Gyrokinetic PIC

- **Equations of motion for the particles along the characteristics:**
  - We solve ODEs instead of PDEs

$$\frac{dR}{dt} = v_{\parallel} \hat{B} - \left( \frac{q}{m\Omega} \right) \left( \frac{\partial \Psi}{\partial R} \times \hat{B} \right)$$

$$\frac{dv_{\parallel}}{dt} = - \left( \frac{q}{m} \right) \frac{\partial \Psi}{\partial R} \cdot \hat{B}$$

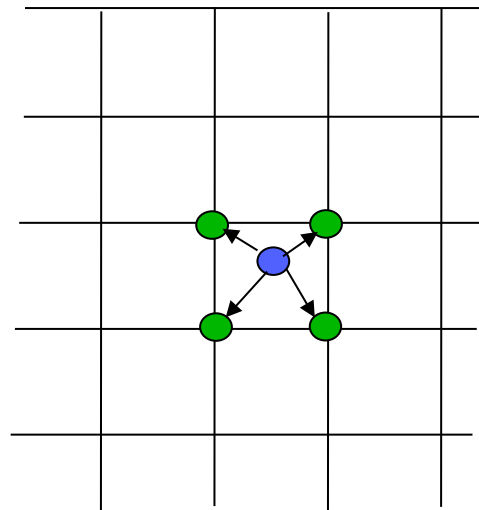
$$\frac{dw_j}{dt} = - \left[ \left( \frac{q}{m\Omega} \right) \frac{\partial \Psi}{\partial R} \times \hat{B} \cdot \hat{x}_k - \left( \frac{q}{m} \right) \frac{\partial \Psi}{\partial R} \cdot \hat{B} \frac{1}{f_0} \frac{\partial f_0}{\partial v_{\parallel}} \right]_{R_j, \mu_j}$$

$$\text{with } w_j = \delta f / f$$

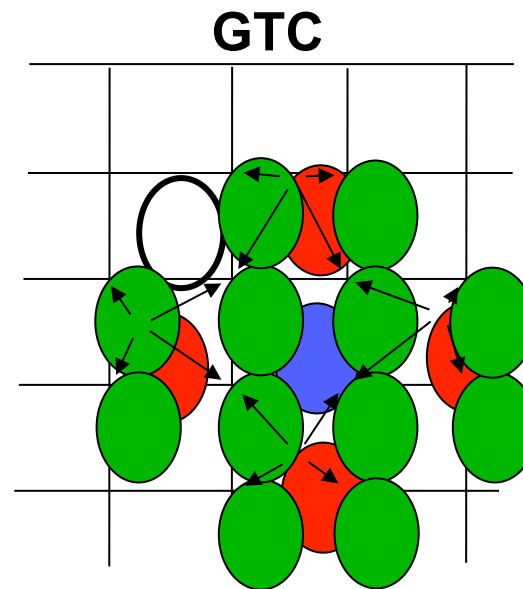
Adapted from Ethier

# Charge Deposition for charged rings: 4-point average method

## Charge Deposition Step (SCATTER operation)



Classic PIC



4-Point Average GK  
(W.W. Lee)

# A field solve is still required, and various methods may be used. Example: Poisson Equation Solver

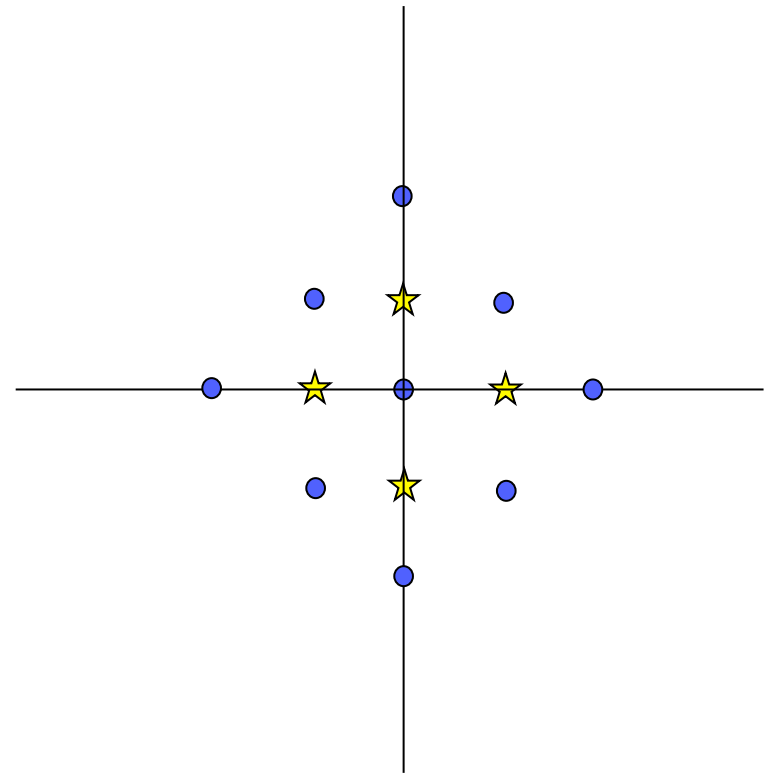
- Done in real space (iterative solver)
- Four or eight-point average method

$$\frac{\tau}{\lambda_D^2} (\Phi - \tilde{\Phi}) = 4\pi e (\bar{n}_i - n_e)$$

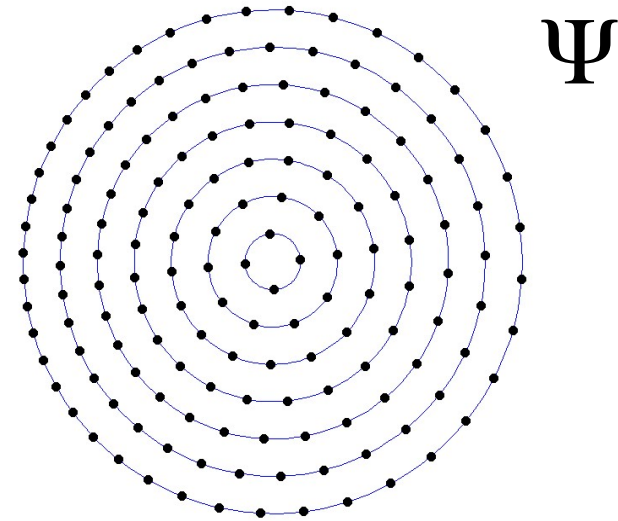
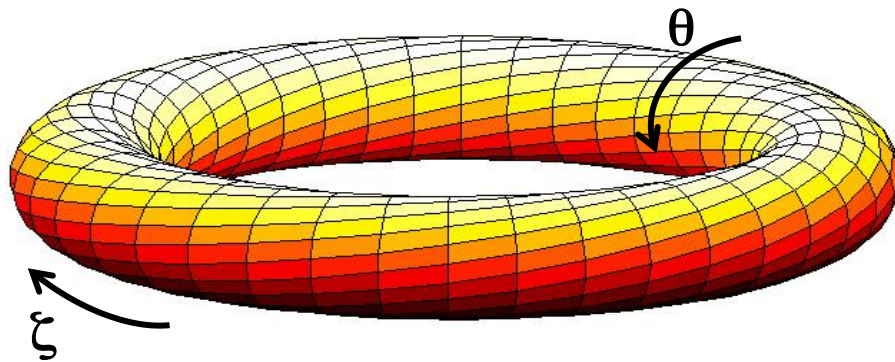
where  $\tilde{\Phi}$  is the second  
gyrophase - averaged potential

Ethier

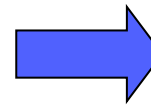
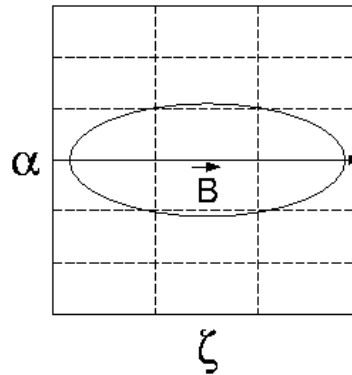
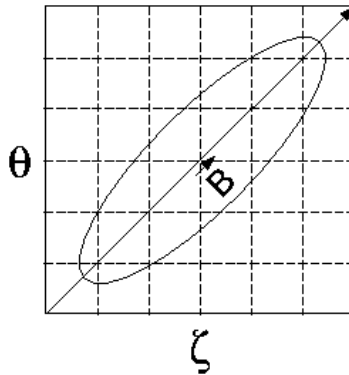
[ Z. Lin and W. W. Lee, *Phys.Rev. E* **52**, 5646--5652 (November 1995).]



# GTC mesh and geometry: Field-line following coordinates



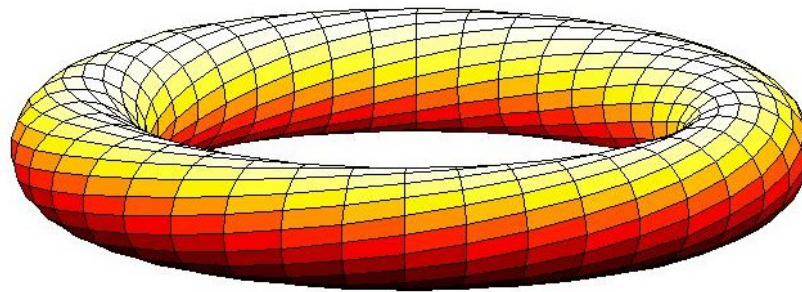
$$(\Psi, \alpha, \zeta) \Rightarrow \alpha = \theta - \zeta/q$$



**Saves a factor of  
about 100 in CPU  
time**

# Domain Decomposition

- **Domain decomposition:**
  - each MPI process holds a toroidal section
  - each particle is assigned to a processor according to its position
- **Initial memory allocation is done locally on each processor to maximize efficiency**
- **Communication between domains is done with MPI calls (runs on most parallel computers)**

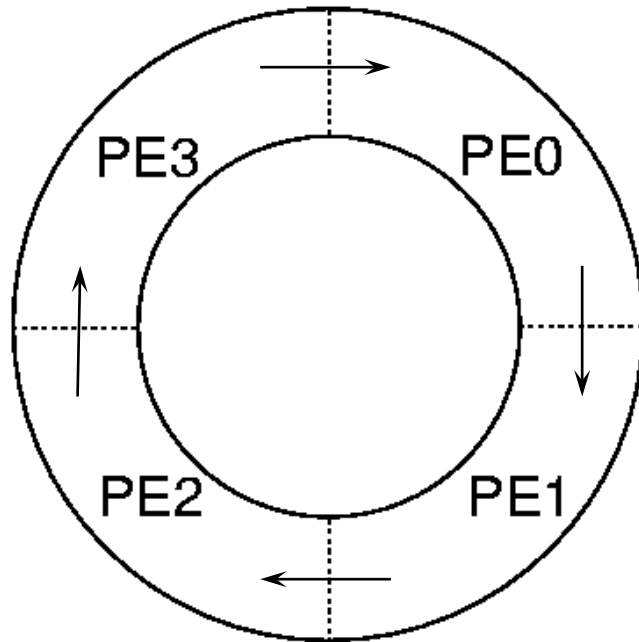


# **Summary: there are 3 Levels of Parallelism in GTC and GTS**

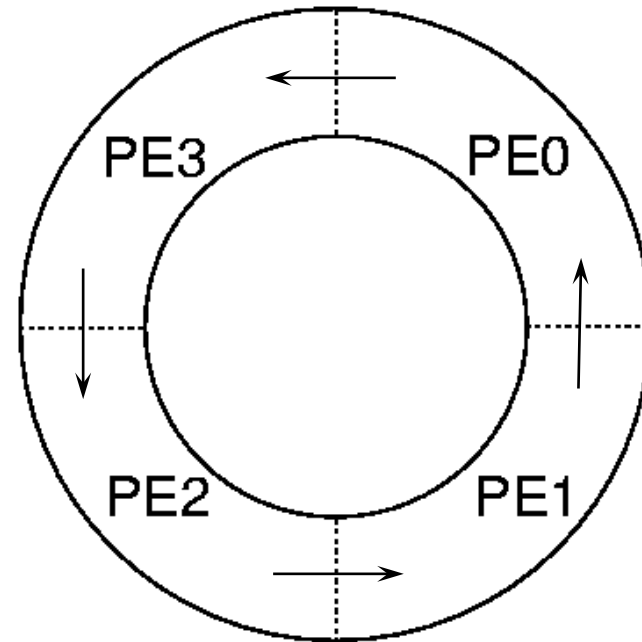
- **1-D domain decomposition in the symmetric, toroidal (MPI). Each MPI process in charge of a toroidal domain with both particles and fields. Particles moved from one domain to another while they traveled around the torus. All communications was one-way traffic to avoid congestion.**
- **Second level of parallelism: Within each toroidal domain, divide the particles between several MPI processes, but each process keeps a copy of all the fields on a single toroidal plane. A “particle-domain” communicator links the MPI processes within a toroidal domain of the original 1D domain decomposition, while a “toroidal-domain” communicator links in a ring-like fashion all the MPI processes with the same intra-domain rank.**
- **Third level of parallelism at the loop level using OpenMP compiler directives**

# Efficient Communications

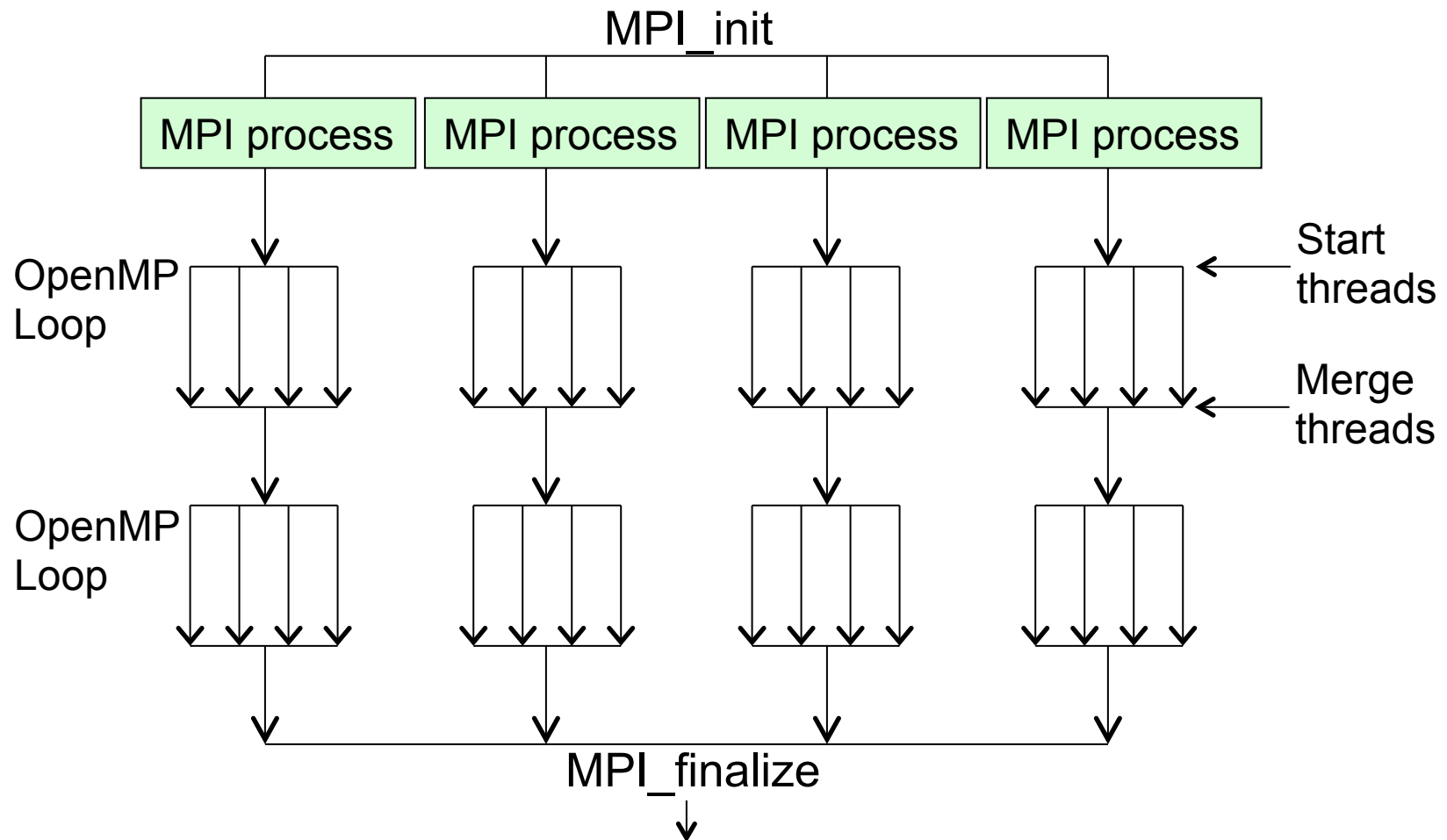
STEP 1



STEP 2



## 2<sup>nd</sup> Level of Parallelism: Loop-level





---

## **Part II How to use proxy apps – Case study: NAS Parallel Benchmarks**

# Hardware Trends are forcing changes in programming models

- Future performance increases rely on increasing concurrency (number of cores/processors/GPUs, etc)
- Energy restrictions negate increases in processor speed
- Data movement is the most significant component of energy use
- Memory per floating point unit suffers a reduction

Proxy apps allow us to consider different programming models

- Control over layout and locality to minimize data movement
- What is the parallel execution model?
- How do we communicate?
- How do we share data on increasingly non-uniform memory architectures
- How can programming models make this easier?

# Programming Models are Changing to Accommodate the Architectural Changes

What do we need to define?

- **Shared Memory (includes globally addressable memory models)**
  - Processes (or threads) communicate through memory addresses accessible to each
- **Distributed memory**
  - Processes move data from one address space to another via sending and receiving messages
- **Parallel programming models are expressed:**
  - In libraries callable from conventional languages (MPI)
  - In languages compiled by their own special compilers (UPC)
  - In structured comments that modify the behavior of a conventional compiler (OpenMP)
  - New ideas or “natural ways” to parallel program (CnC)
- **Hybrid Models combine various models**

**The eight NAS parallel benchmarks (NPBs) have been written in various languages including hybrid (MZ or Multizone) for three**

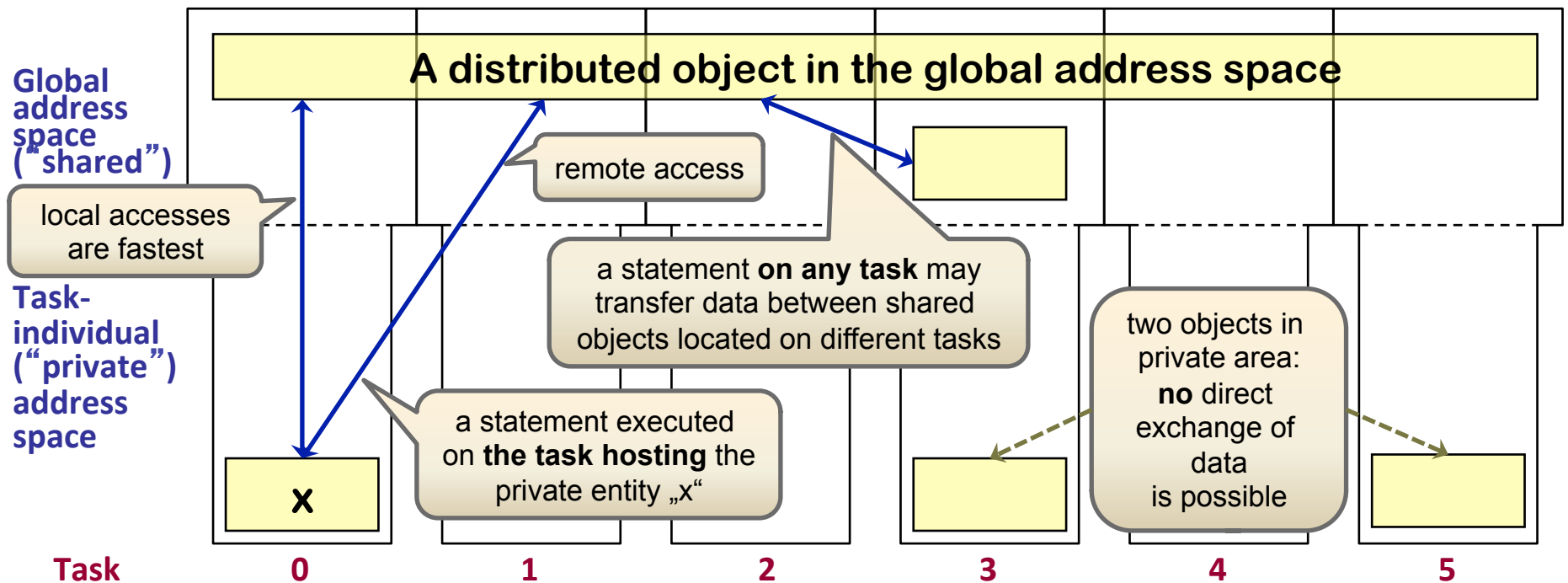
|                |  |  |    |
|----------------|--|--|----|
| MG             | Multigrid  | Approximate the solution to a three-dimensional discrete Poisson equation using the V-cycle <u>multigrid method</u>                          |    |
| CG             | Conjugate Gradient                                   | Estimate smallest <u>eigenvalue</u> of <u>sparse SPD matrix</u> using the <u>inverse iteration</u> with the <u>conjugate gradient method</u> |    |
| FT             | Fast Fourier Transform                               | Solve a three-dimensional PDE using the <u>fast Fourier transform</u> (FFT)  |    |
| IS             | Integer Sort   | Sort small integers using the bucket sort algorithm  |    |
| EP             | Embarrassingly Parallel                              | Generate independent <u>Gaussian random variates</u> using the <u>Marsaglia polar method</u>   |    |
| BT<br>SP<br>LU | Block Tridiagonal<br>Scalar Pentadiag<br>Lower/Upper | Solve a system of <u>PDEs</u> using 3 different algorithms   | MZ |

# **We will consider three different programming model implements of NPBs**

- **UPC (Unified Parallel C) is a PGAS (Partitioned Global Address Space) Language**
  - A number of threads working independently in a SPMD fashion
  - Number of threads specified at compile-time or run-time depending how program is written. NPBs use static threads (compile time).
- **MPI (Message Passing Interface)**
- **Hybrid (Multizone for NPBs) includes implementations with MPI +OpenMP**
  - MPI+OpenMP is most common hybrid programming mode
  - Many variations of number of threads vs. MPI processes

# Partitioned Global Address Space (PGAS) Languages

- **Defining PGAS principle: extended memory model**
  - 1) The *Global Address Space*: a special memory area that allows any task to read or write memory anywhere in the system
  - 2) It is *Partitioned* to allow an efficient implementation of distributed objects (“symmetric heap”)



Author: Rolf Rabenseifner, HLRS

Fusion Mini/Proxy Apps and NPBs

# The PGAS Languages

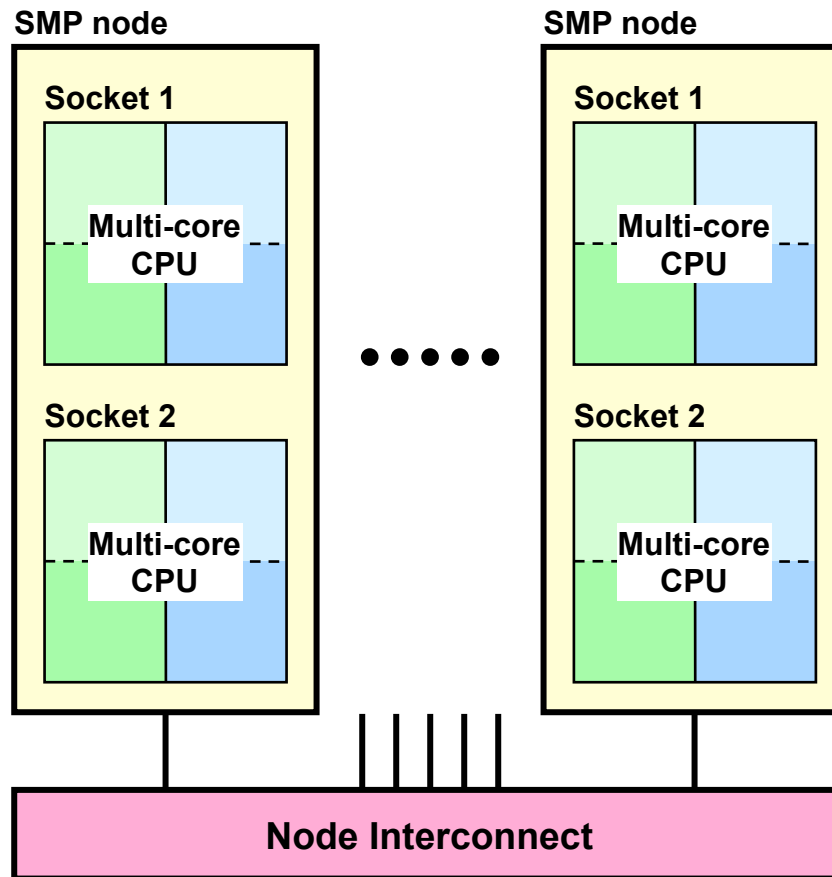
- **PGAS (Partitioned Global Address Space) languages attempt to combine the convenience of the global view of data with awareness of data locality, for performance**
  - Co-Array Fortran, an extension to Fortran-90)
    - SPMD – Single program, multiple data
    - Replicated to a number of images
    - Variables declared as co-arrays are accessible by another image through a set of array subscripts, delimited by [ ] and mapped to image indices by the usual rule
  - UPC (Unified Parallel C), an extension to C
    - UPC is an extension of C (not C++) with shared and local addresses
    - Introduces *Shared* keyword in type declarations
    - processes are called *threads* in UPC
    - *Global address space*: thread may directly read/write remote data
  - Various newer PGAS Languages including Chapel, X10, etc.

# MPI and Threads

- **MPI describes parallelism between *processes* (with separate address spaces)**
- ***Thread* parallelism provides a shared-memory model within a process**
- **OpenMP and Pthreads are common but different models**
  - OpenMP provides convenient features for loop-level parallelism
  - Pthreads provide more complex and dynamic approaches
  - OpenMP 3.0 (which adds task parallelism) adds some of these capabilities to OpenMP
- **MPI combined with OpenMP is the most common current means of adapting for heterogenous architectures**
  - Doesn't always work
  - Is not able to deal with NUMA on the nodes

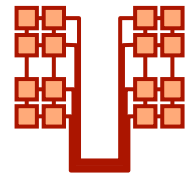


# Within the MPI-OpenMP hybrid model, there are variants depending on system and application

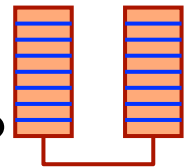


Which programming model is fastest?

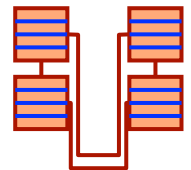
MPI everywhere?



Fully hybrid MPI & OpenMP?



In - between?  
(Mixed model)



Historically hybrid programming can be **slower** than pure MPI

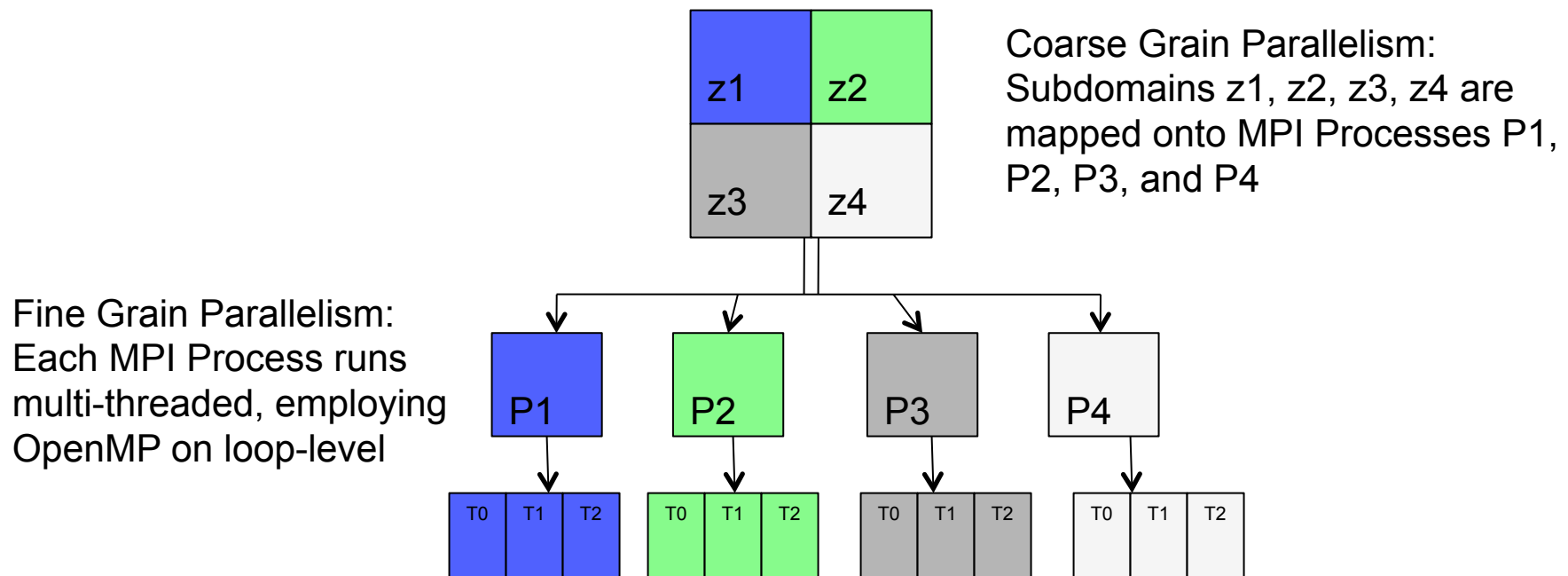


# New Models MPI + x or ?

- **We are considering new programming models that combine MPI with another language such as UPC or CAF in addition to the standard hybrid method of MPI+OpenMP**
- **There are also a large number of new languages for example:**
  - Intel's CnC or Concurrent Collections
    - Invites users to rethink their problem into 2 pieces:
      - Data dependence and control dependence
  - Microsoft's parallel language suites including:
    - Axum, Parallel Patterns Library
  - OpenCL
    - A framework for writing parallel programs on heterogeneous
  - OpenACC
    - Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.
- **Also, most current languages (OpenMP, MPI, etc) are looking at what changes should be made for architecture evolution**

# The NAS MZ benchmarks allow us to explore Multi-Level Parallelism in Applications

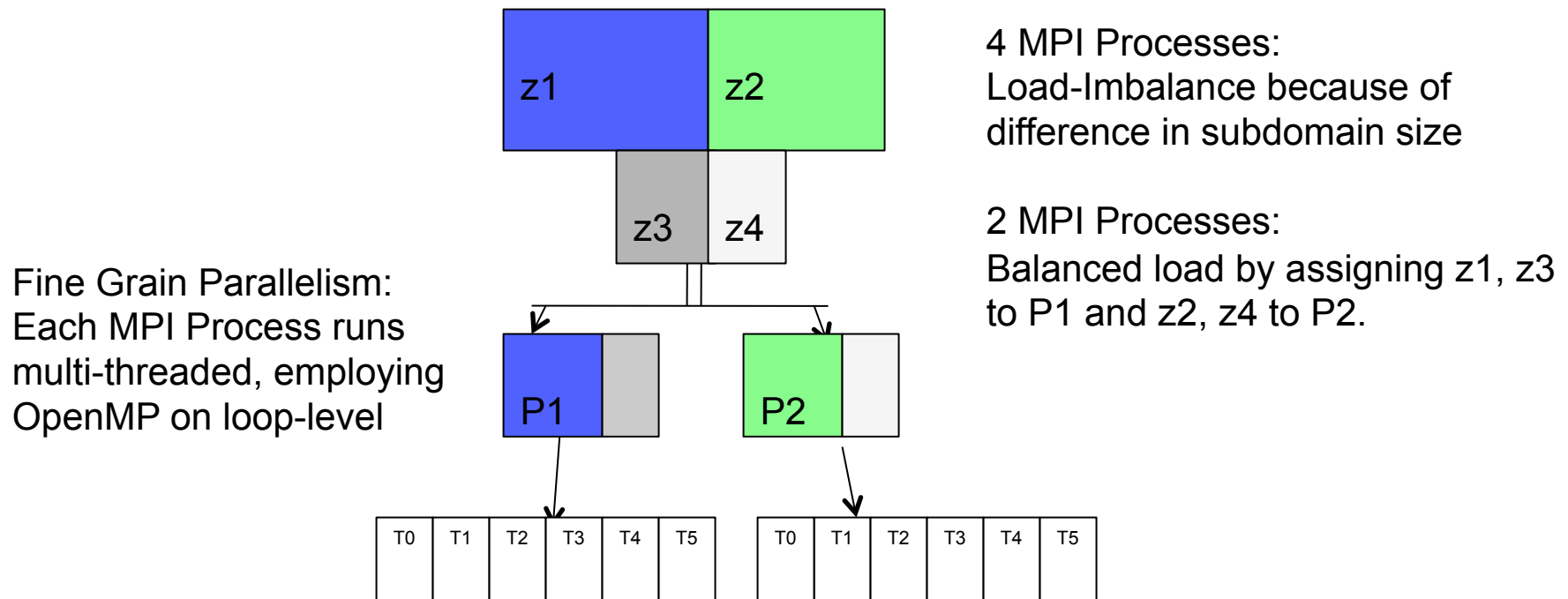
- Extract additional Parallelism in case of Limited coarse grain Parallelism



Adapted from Gabriele Jost, Supersmith, [gjost@supersmith.com](mailto:gjost@supersmith.com)

# Coarse Grain Load-Balancing

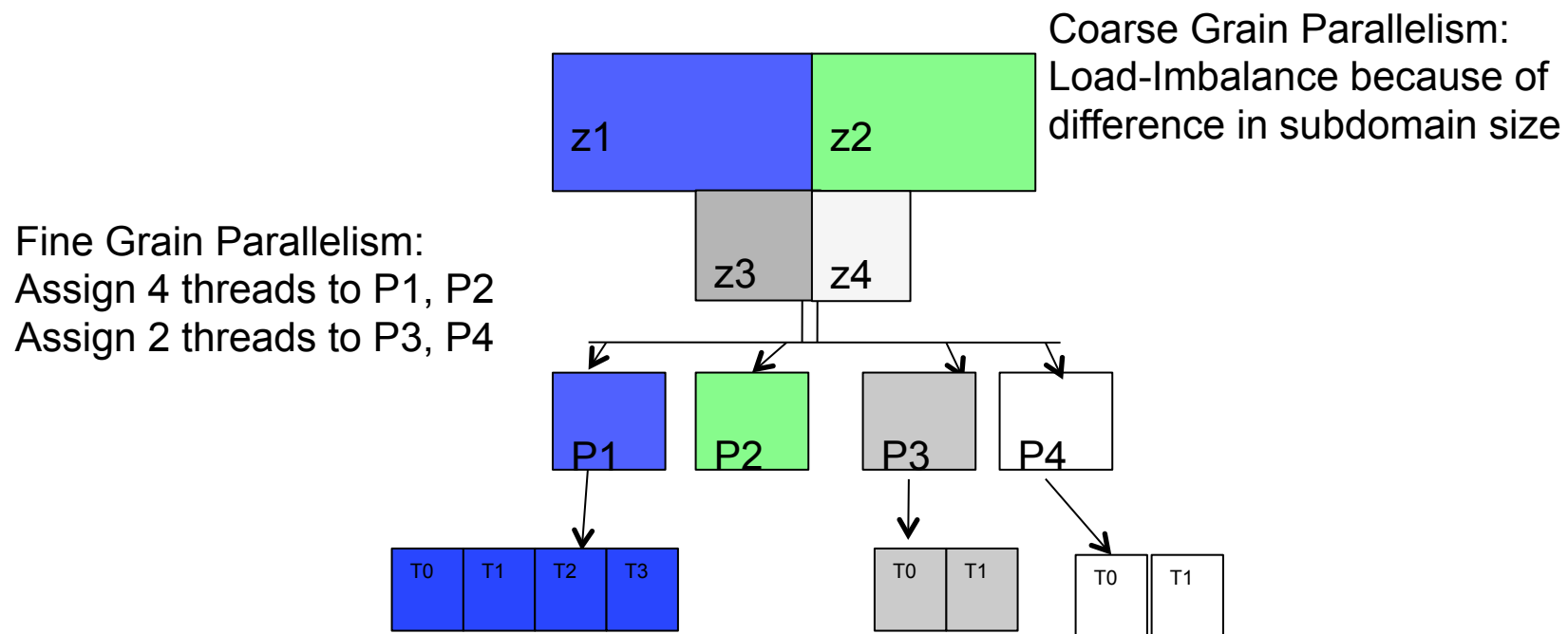
- **Improve Load-Balance**
  - Restrict #MPI Processes
  - Exploit loop level parallelism instead



From: Gabriele Jost, Supersmith, [gjost@supersmith.com](mailto:gjost@supersmith.com)

# Fine Grain Load-Balancing

- **Improve Load-Balance on Fine Grain**
  - Assign more threads to MPI Process with high workload



# The Multi-Zone NAS Parallel Benchmarks combine MPI and OpenMP

Initialization

Exchange of Boundary Values

Kernel Solvers: LU/SP/BG

- **Multi-zone versions of the NAS Parallel Benchmarks LU, BT, and SP were developed by dividing the discretization mesh into a two-dimensional tiling of three-dimensional zones**
- **Same kernel solvers in the multi-zone code**

NAS Parallel Benchmarks, Multi-Zone Versions,  
NAS-03-010 (PDF-128KB) for BT-MZ, SP-MZ, LU-MZ. **Rob F. Van der Wijngaart, Haoqiang Jin**

# Multi-zone Benchmark Characteristics

- **Aggregate sizes:**

- Class C: 480 x 320 x 28 grid points
- Class D: 1632 x 1216 x 34 grid points
- Class E: 4224 x 3456 x 92 grid points

Expectations:

- **BT-MZ: (Block-tridiagonal Solver)**

- #Zones: 256 (C), 1024 (D), 4096 (E)
- Size of the zones varies widely:
  - large/small about 20
  - requires multi-level parallelism to achieve a good load-balance

Pure MPI: Load-balancing problems!

Good candidate for MPI+OpenMP

- **LU-MZ: (Lower-Upper Symmetric Gauss Seidel Solver)**

- #Zones: 16 (C, D, and E)
- Size of the zones identical:
  - no load-balancing required
  - limited parallelism on outer level

Limited MPI Parallelism:  
→ MPI+OpenMP increases Parallelism

- **SP-MZ: (Scalar-Pentadiagonal Solver)**


- #Zones: 256 (C), 1024 (D), 4096 (E)
- Size of zones identical
  - no load-balancing required

Load-balanced on MPI level: Pure MPI should perform best

Adapted from Gabriele Jost, Supersmith, [gjost@supersmith.com](mailto:gjost@supersmith.com)

# BT-MZ based on MPI/OpenMP

## Coarse-grain MPI Parallelism

```
call omp_set_numthreads (weight)
do step = 1, itmax
  call exch_qbc(u, qbc, nx,...)
  
  call mpi_send/recv
  do zone = 1, num_zones
    if (iam .eq.pzone_id(zone))
    then
      call comp_rhs(u,rsd,...)
      call x_solve (u, rhs,...)
      call y_solve (u, rhs,...)
      call z_solve (u, rhs,...)
      call add (u, rhs,...)
    end if
  end do
end do
...
```

## Fine-grain OpenMP Parallelism

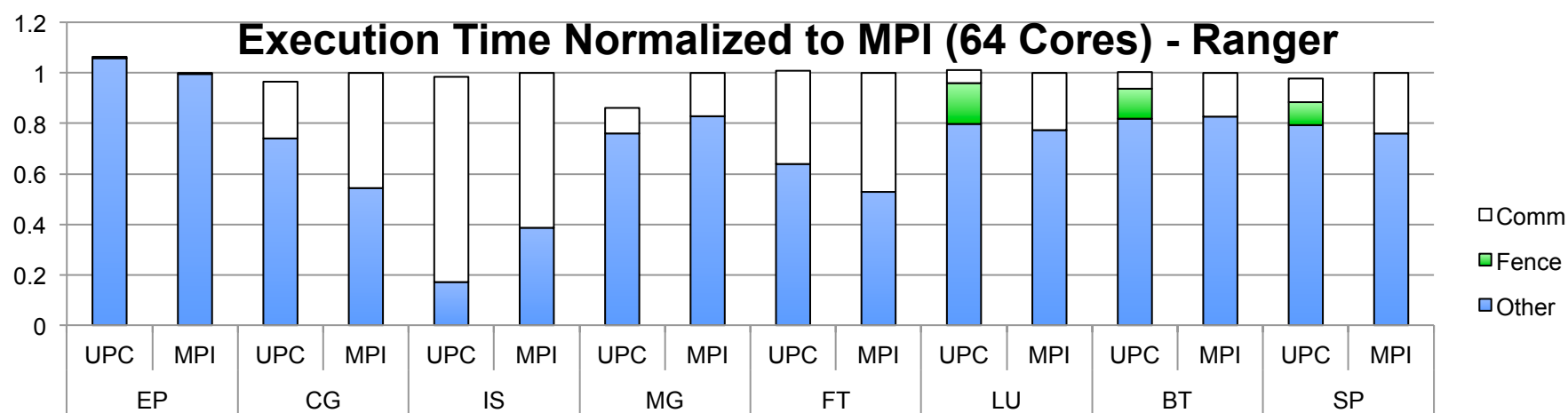
```
subroutine x_solve (u, rhs,
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP& PRIVATE(i,j,k, isize...)
isize = nx-1
!$OMP DO
  do k = 2, nz-1
    do j = 2, ny-1
      ....
      call lhsinit (lhs, isize)
      do i = 2, nx-1
        lhs(m,i,j,k)= ..
      end do
      call matvec ()
      call matmul ()....
    end do
  end do
end do
!$OMP END DO nowait
!$OMP END PARALLEL
```

Adapted from Gabriele Jost, Supersmith, [gjost@supersmith.com](mailto:gjost@supersmith.com)



# The NPBs in UPC are also useful for studying various PGAS issues

- **Using customized communication to avoid hot-spots**
  - UPC Collectives do not support certain useful communication patterns
- **Blocking vs. Non-Blocking (Asynchronous) communication**
  - In FT and IS using non-blocking gave significantly worse performance
  - In MG using non-blocking gave small improvement
- **Benefits of message aggregation depends on the arch./interconnect**
- **UPC – Shared Memory Programming studied in FT and IS**
  - Less communication but reduced memory utilization



## Class D NPBs have been run recently on two PF/s class machines at LRZ and LBL

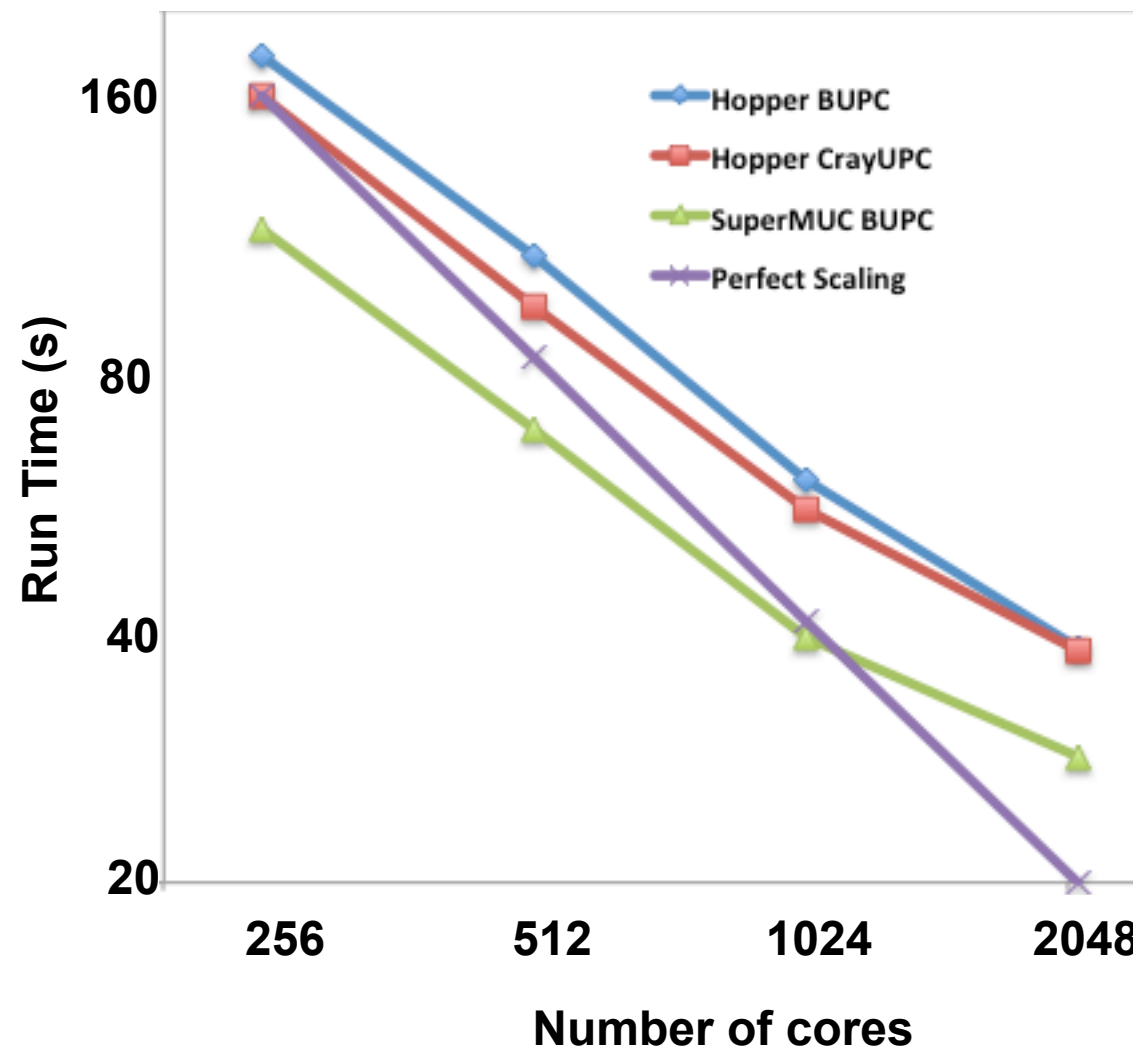
| Property         | SuperMuc            | Hopper             |
|------------------|---------------------|--------------------|
| Peak Performance | 3.19 PF/s (#4)      | 1.28 PF/s (#16)    |
| Number of Cores  | 147,456             | 153,216            |
| Clock Speed      | 2.7 (3.5 Turbo) GHz | 2.1 GHz            |
| Interconnect     | Infiniband FDR10    | Gemini in 3D torus |
| Total Memory     | 288 TBytes          | 217 TBytes         |

| MG.D 1024 cores<br>Machine name and<br>Compiler | Speed for 5<br>runs | No<br>Flags | Message<br>Aggregation | Message<br>Aggregation +<br>Strided Comm |
|---|---------------------|-------------|------------------------|--|
| Hopper with Cray UPC                            | Avg Gops/s          | 433.68      | 440.97 (+ 2%)          | 456.10 (+ 5%)                            |
| SuperMUC with Berkeley UCP                      | Avg Gops/s          | 891.70      | 1034.5 (+16%)          | 1041.4 (+ 17%)                           |
| Hopper with Cray UPC                            | SD Gops/s           | 3.55        | 12.93                  | 6.61                                     |
| SuperMUC with Berkeley UCP                      | SD Gops/s           | 32.6        | 54.2                   | 72.4                                     |

SuperMUC data from Reinhold Bader, LRZ

# NPB can be used to study scalability as well machine and compiler effects

## LU.D NPB



For MG.D the 2X faster cores on SuperMuc compared to Hopper gave 2X reduction in run time but for LU.D the reduction is only 1.5X

SuperMUC data from Reinhold Bader, LRZ

Fusion Mini/Proxy Apps and NPBs

## **Some comments for hands-on**

# README – UPC

## on Cray XE6-Hopper: UPC / PGI

Initialization: module load `bupc`

Interactive PBS shell grab two nodes:

In the SC tutorial

```
qsub -q special qsub -I -V -q interactive -l mppwidth=48
```

Again to the working directory:

```
cd $PBS_O_WORKDIR
```

Compilation:

Parallel Execution:

# README – UPC

## on Cray XE6-Hopper: Cray UPC

Initialization: module switch PrgEnv-pgi **PrgEnv-cray**

Interactive PBS shell:

In the SC tutorial

```
qsub -q special qsub -I -V -q interactive -l mppwidth=48
```

Again to the working directory:

```
cd $PBS_O_WORKDIR
```

Compilation:

Parallel Execution: