Performance Study of the 3D Particle-in-Cell Code GTC on the Cray X1

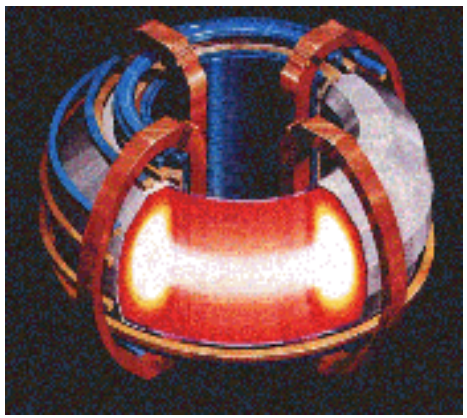
Stéphane Ethier
Princeton Plasma Physics Laboratory

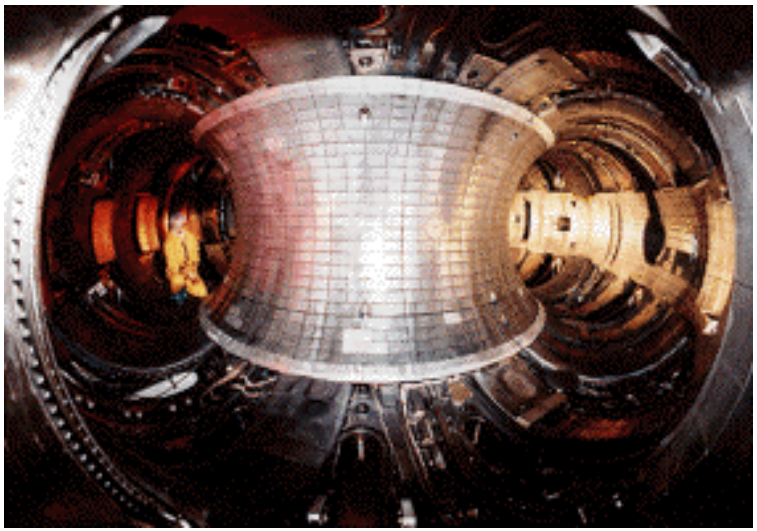
CUG 2004
Knoxville, TN
May 20, 2004

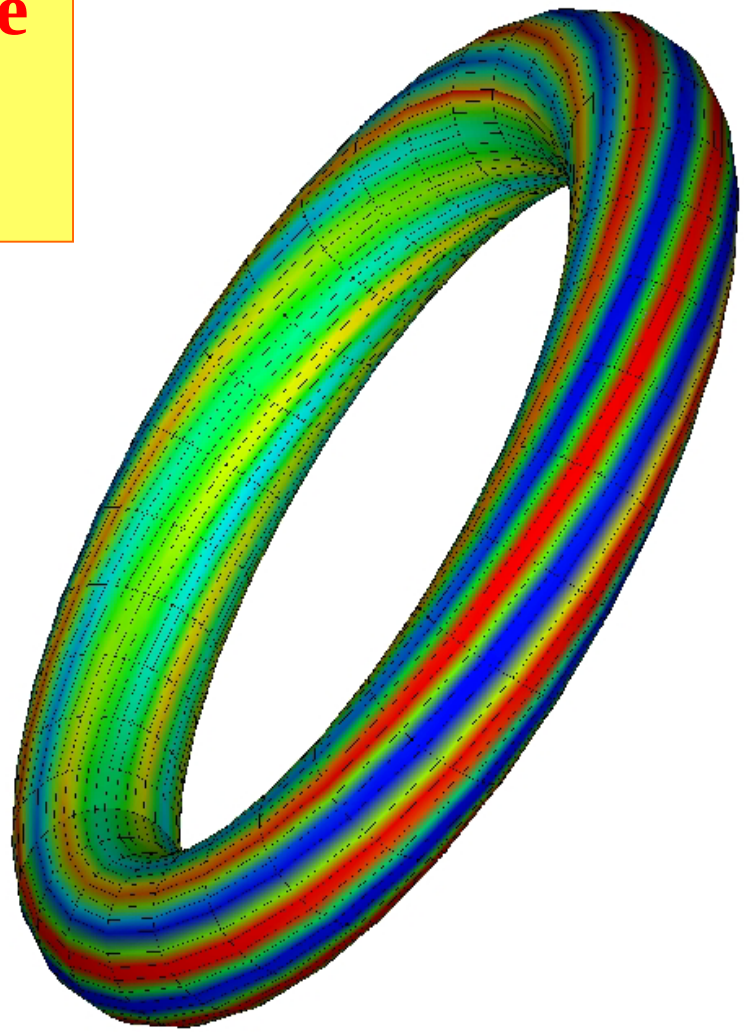
Work Supported by DOE Contract No.DE-AC02-76CH03073 and by the DOE SciDAC Plasma Microturbulence Project.
Work done in collaboration with LBNL Leonid Oliker, lead P.I. for the benchmarking project on Advanced Vector Architectures.

Magnetic Confinement Fusion

The Ultimate Source of Energy!







Importance of 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 (ψ,θ,ζ) [Boozer, 1981]
  - Guiding center Hamiltonian [White and Chance, 1984]
  - Non-spectral Poisson solver [Lin and Lee, 1995]
  - Low numerical noise algorithm ($\delta \boldsymbol{f}$ method)
  - Full torus (global) simulation

Gyrokinetic Toroidal Code (GTC)

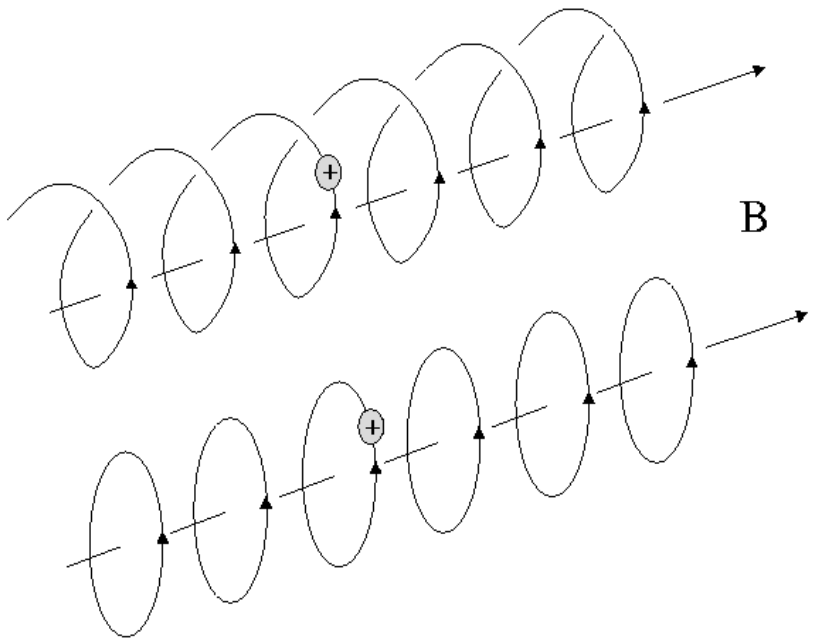
- 3D particle-in-cell code solving gyrokinetic Vlasov equation in toroidal geometry.
- Real space iterative solver for gyrokinetic Poisson's equation.
- Low noise df method: ideal for load balance considerations.
- Global code (general geometry torus with shaped plasmas as opposed to a flux tube).
- Electrostatic approximation with adiabatic electrons (currently being upgraded to non-adiabatic electrons).
- Include spatial(E x B)- and velocity-space nonlinearities.
- Written in Fortran 90/95

Gyrokinetic approximation for low frequency modes

- Gyrokinetic ordering $\frac{\omega}{\Omega} \sim \frac{\rho}{L} \sim \frac{e\phi}{T} \sim k_{//}\rho << 1$

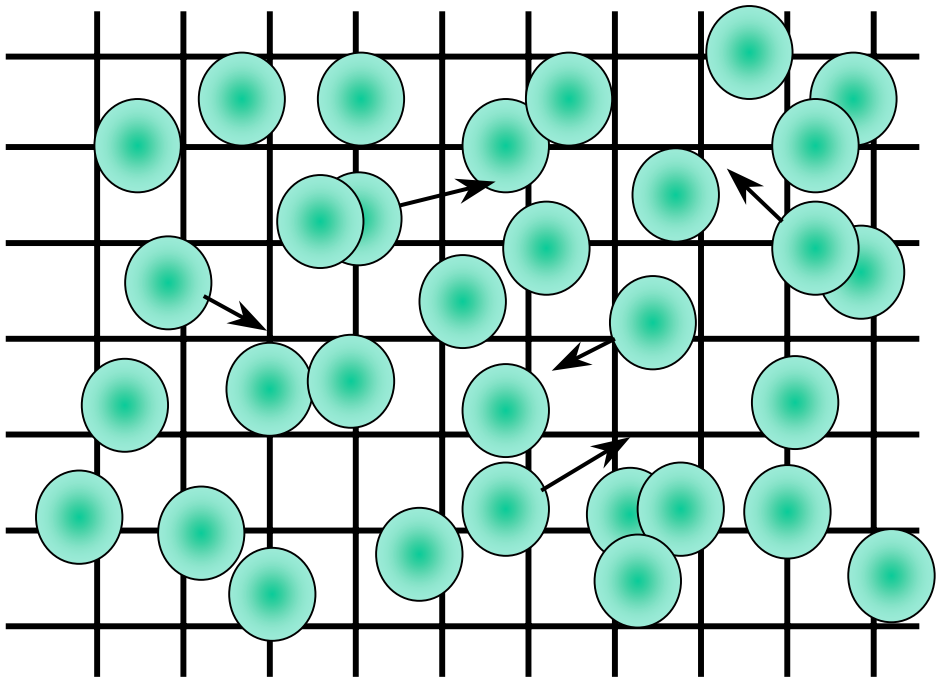
  $k_{\perp}\rho \sim 1$

- Gyro-motion: guiding center drifts + charged ring
- Gyrophase-averaged 5D kinetic (Vlasov) equation



Particle-in-cell (PIC) method

- 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…**

Advantages of PIC

- Naturally includes all nonlinearities
- Scales as N instead of N^2.
- Equations of motion for the particles along the characteristics:
  - We solve ODEs instead of PDEs

$$\frac{d\mathrm{R}}{dt} = \mathrm{v}_{\|}\hat{\mathrm{B}} - \left(\frac{q}{m\Omega}\right)\left(\frac{\partial\Psi}{\partial\mathrm{R}}\times\hat{\mathrm{B}}\right)$$

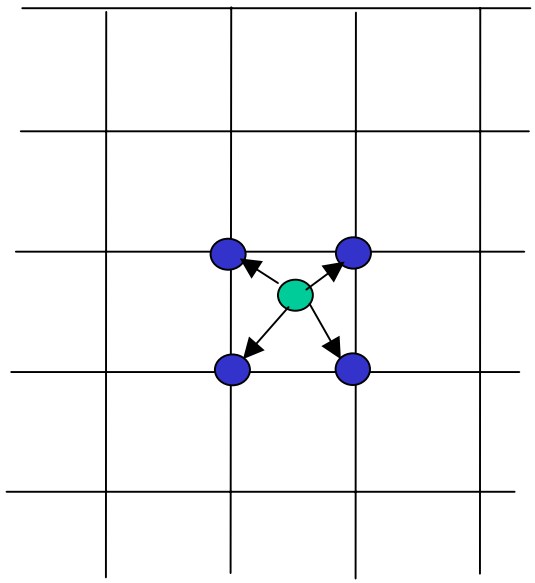
$$\frac{d\mathrm{v}_{\|}}{dt} = -\left(\frac{q}{m}\right)\frac{\partial\Psi}{\partial\mathrm{R}}\cdot\hat{\mathrm{B}}$$

$$\frac{d\mathrm{w}_j}{dt} = -\left[\left(\frac{q}{m\Omega}\right)\frac{\partial\Psi}{\partial\mathrm{R}}\times\hat{\mathrm{B}}\cdot\hat{\mathrm{x}}_k - \left(\frac{q}{m}\right)\frac{\partial\Psi}{\partial\mathrm{R}}\cdot\hat{\mathrm{B}}\frac{1}{f_0}\frac{\partial f_0}{\partial\mathrm{v}_{\|}}\right]_{\mathrm{R}_j,\mu_j}$$

with $\mathrm{w}_j = \delta f / f$

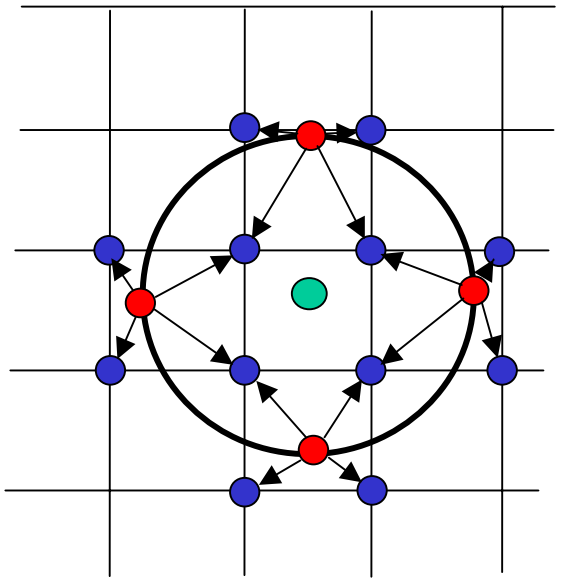
Charge Deposition for charged rings: 4-point average method

Charge Deposition Step (SCATTER operation)



Classic PIC

GTC



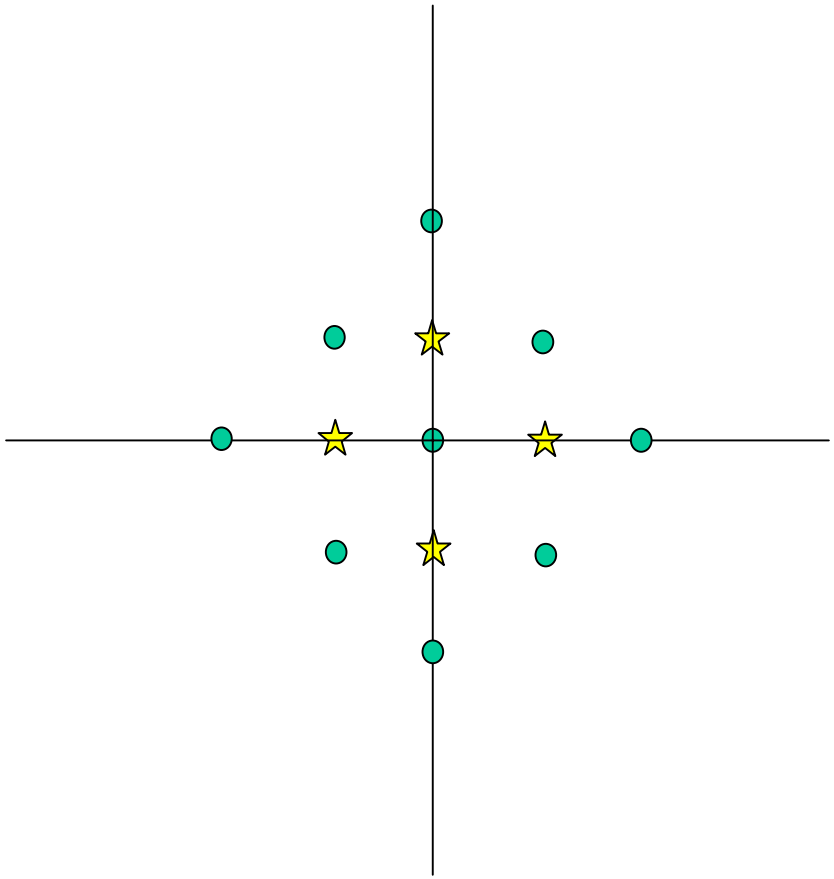
4-Point Average GK (W.W. Lee)

Poisson Equation Solver

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

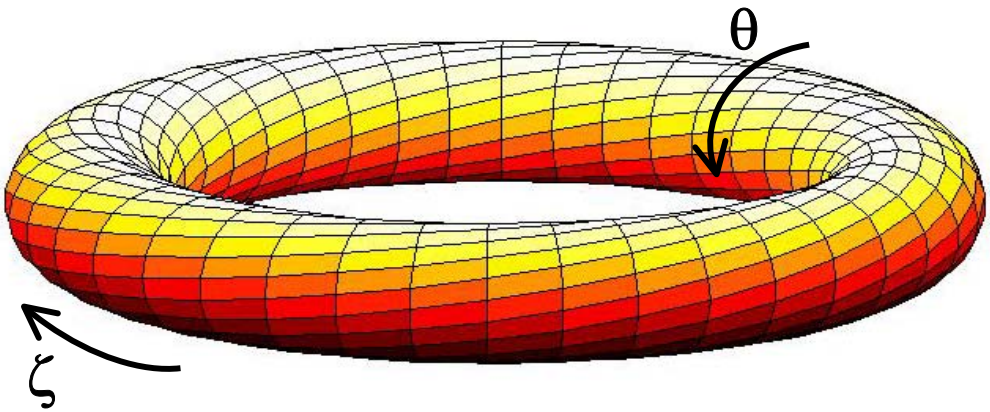
$$\frac{\tau}{\lambda_D^2}\left(\Phi - \tilde{\Phi}\right) = 4\pi e\left(\bar{n}_i - n_e\right)$$

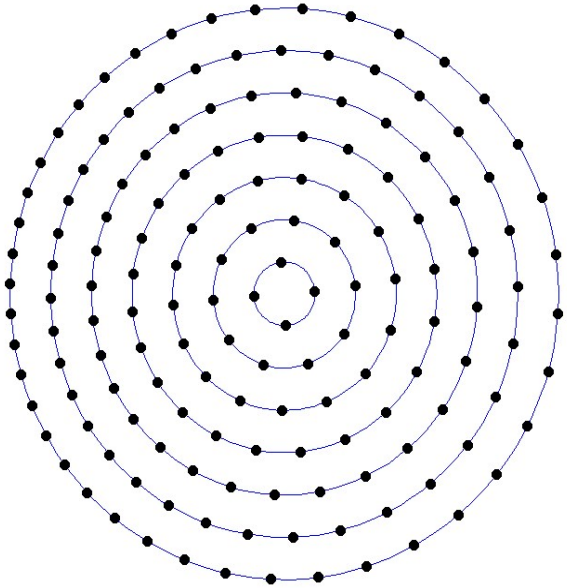
where $\tilde{\Phi}$ is the second gyrophase-averaged potential



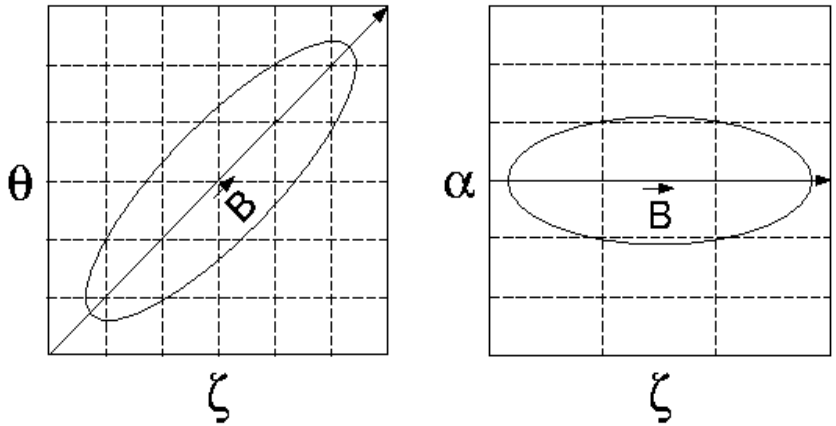
[Z. Lin and W. W. Lee, *Phys.Rev. E* **52**, 5646--5652 (November 1995).]

GTC mesh and geometry: Field-line following coordinates





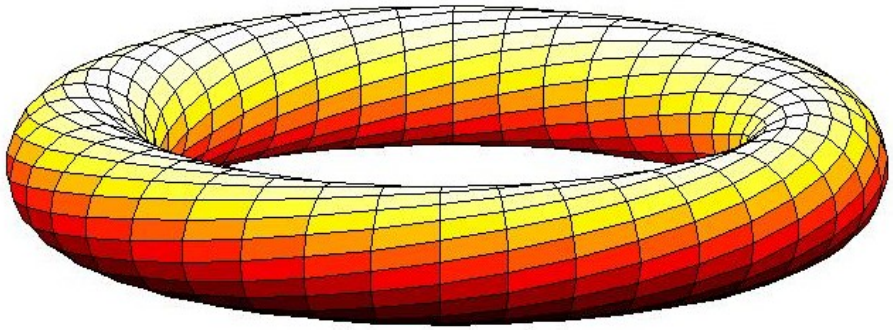
$(\Psi,\alpha,\zeta) \Rightarrow \alpha = \theta - \zeta/q$



Saves a factor of about 100 in CPU time

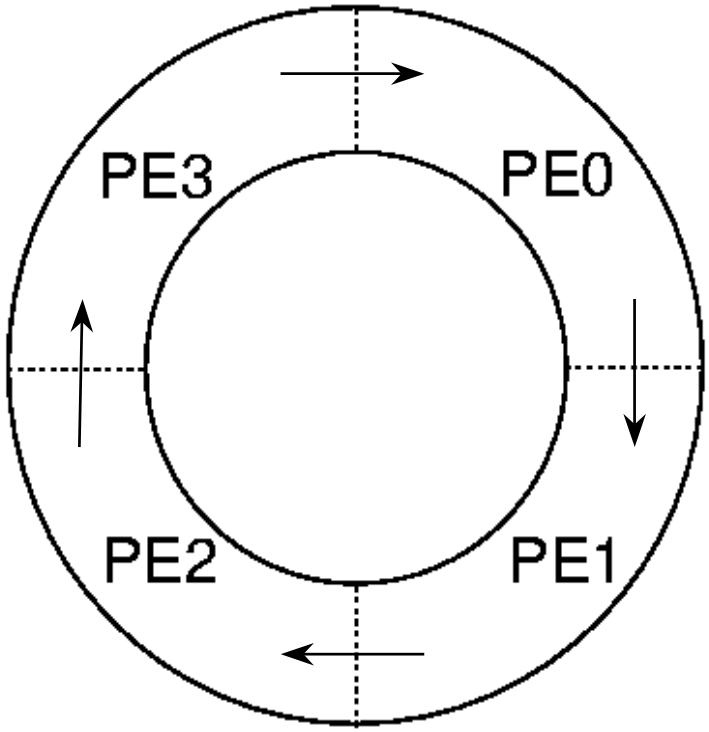
Domain Decomposition

- 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)

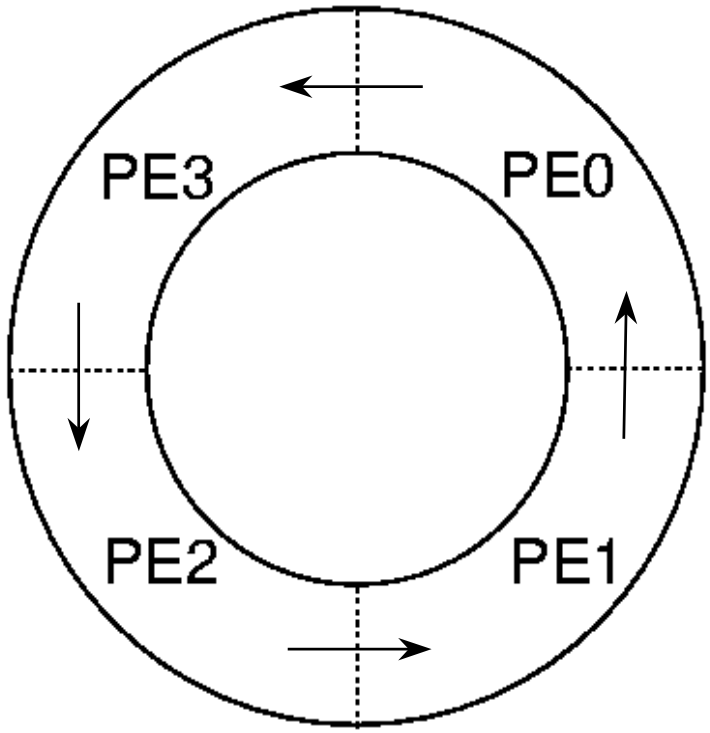


Efficient Communications

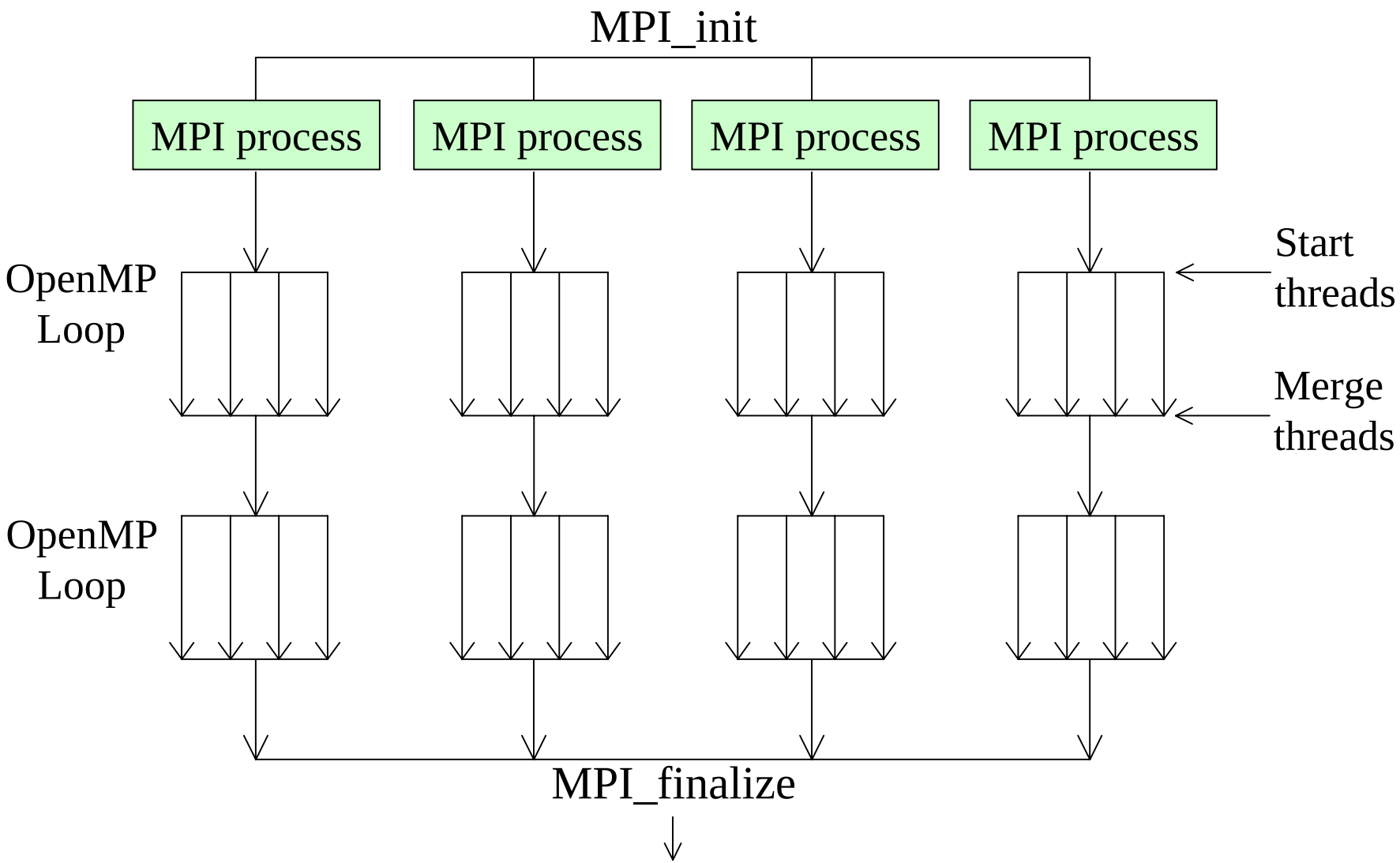
STEP 1



STEP 2



2nd Level of Parallelism: Loop-level

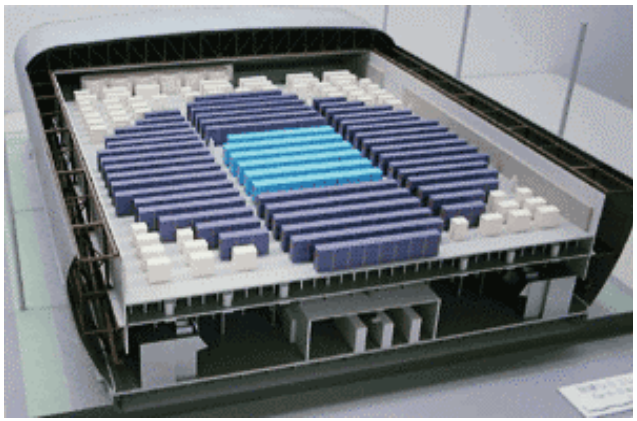


Computational Facts about GTC

- Only 5000 lines.
- Written in standard Fortran 90/95.
- Highly portable. GTC runs on most parallel computers as long as the MPI library is available.
- Part of the NERSC benchmark suite of codes to evaluate new computers.
- Runs in single precision (4-byte REALs)
- Only 5 to 10% of wall-clock time spent in communications for non-linear runs.
- Uses FFTs only for diagnostics and mode decoupling in linear runs.
- Typical runs done on 1024 processors on IBM SP Power3 (Seaborg) at NERSC.

GTC vectorization work

- Started on the single-node CRAY/NEC SX-6 at ARSC
- Porting GTC was very easy although the first tests on a single processor gave a very low performance
- Real work starts: profiling, vectorizing, optimizing, test, and… repeat
- Multi-processor optimization done on to the Earth Simulator and CRAY X1



GTC's most time-consuming routines on vector computers

- Charge deposition on the grid ~40% ("scatter" operation)
  - Not very cache-friendly
  - The particles can be anywhere in the volume
  - Most challenging routine to vectorize/optimize
  - ~42% of cpu time on Power3/4
- Particle push ~35% ("gather" operation + ODE solving)
  - The gather operation is more efficient than the scatter operation but needs many non-sequential memory access (reads).
  - Was easily vectorized after removing a conditional test that included an I/O statement.
  - ~40% of cpu time on Power3/4
- Poisson solver ~15%
  - Some modifications to improve memory access…
  - A bigger percentage of compute time in next version of the code
  - ~ 7% of cpu time on Power3/4

Vectorization challenge for PIC: Scatter operation

- The charge deposition step (scatter operation) writes to the charge accumulation array in a random fashion (particle positions are random), producing dependencies and memory conflicts whenever 2 or more particles have a common neighboring grid point → this prevents vectorization
- In 1D, the charge deposition step with linear interpolation looks like this:

```
do i=1,nparticles
    x = particle_position(i)
    ix_grid = int(x)
    dx = x – real(ix_grid)
    charge(ix_grid) = charge(ix_grid)+q*(1-dx)
    charge(ix_grid+1) = charge(ix_grid+1)+q*dx
end do
```

Indirect addressing!

Potential Conflicts

Avoiding memory dependencies in the scatter operation

- Remove memory conflicts by having a copy of the charge accumulating array for each element in the vector register
  - Achieves 100% vectorization!…
  - …But uses a lot of memory
  - Still have to do a lot of random reads and writes to memory.
- We could also sort the particles according to their positions… but the overhead of the sorting routine would have to be taken into account.
  - More Flops… but also longer time to solution.

Avoiding memory dependencies: The work-vector method (Nishiguchi '85)

Example of loop with indirect addressing similar to charge deposition:

```
DO i=1,np
   charge(ix(i))=charge(ix(i)) + q(i)
END DO
```

Fully vectorizable loop using multiple copies (vector length of 256):

```
ALLOCATE(charge_tmp(256,ngrid))
DO i=1,np,256
   DO j=1,min(256,np-i+1)
      charge_tmp(j,ix(i+j-1))=charge_tmp(j,ix(i+j-1)) + q(i+j-1)
   END DO
END DO
DO i=1,256
   DO ig=1,ngrid
      charge(ig)=charge(ig) + charge_tmp(i,igrid)
   END DO
END DO
```

Uses 256*ngrid*sizeof(charge_tmp) of extra memory! (can be 1GB)

Single Processor Performance: SX-6 *vs.* IBM Power 3/4

Processor	Max speed (Gflops)	GTC test (Mflops)	Efficiency (real/max)	Relative speed (user time)
Power3 (Seaborg)	1.5	173.6	12 %	1
Power4 (Cheetah)	5.2	304.5	6 %	1.9
SX-6 (Rime)	8.0	715.7	9 %	5.2

Mflops/sec given by hpmcount on Power3 and Power4, and by FTRACE on SX-6

Cache-less memory access issues on the SX-6 and ES

- Better memory access is the secret to higher performance
- True for STORING to memory as well as FETCHING from it!

```
do m=1,mi
     psitmp=zion(1,m)
     thetatmp=zion(2,m)
     zetatmp=zion(3,m)
     rhoi=zion(6,m)*smu_inv
     r=sqrt(2.0*psitmp)
     ip=max(0,min(mpsi,int((r-a0)*delr+0.5)))
     jt=max(0,min(mtheta(ip),int(thetatmp*pi2_inv*delt(ip)+0.5)))
     ipjt=igrid(ip)+jt
     wz1=(zetatmp-zetamin)*delz
     …
```

Repeatedly accessing the same memory bank before the bank busy time is over from the last access leads to poor memory performance!

Duplicate small arrays like “igrid” and “mtheta”: !$duplicate

37% improvement on chargei, but uses even more memory…

***** Small vectors can be cached on the CRAY X1!!**

Vector performance of main routines on the Earth Simulator

ORIGINAL CODE BEFORE MODIFICATIONS:

PROG.UNIT	EXCLUSIVE TIME[sec](%)	MFLOPS	V.OP RATIO	AVER. V.LEN	BANK CONF
chargei	282.677(54.4)	62.0	0.65	98.1	0.0000
pushi	125.211(24.1)	320.1	67.51	196.8	4.3336
poisson	57.878(11.1)	418.9	94.26	107.2	0.3158

Note: the 2 tests do not have the same number of time steps so the times are different

CODE AFTER MODIFICATIONS TO CHARGEI, PUSHI, POISSON:

PROG.UNIT	EXCLUSIVE TIME[sec](%)	MFLOPS	V.OP RATIO	AVER. V.LEN	BANK CONF
chargei	89.924(33.3)	1314.3	99.65	248.1	6.5002
pushi	93.877(34.7)	2426.6	99.38	255.9	8.8139
poisson	26.239(9.7)	918.1	99.71	252.7	3.2485

Total = 1.412 Gflops per proc

What about the CRAY X1?

- Same vectorizations apply except for the !duplicate directive trick
- Easier to prevent vectorization of small inner loops
- Also needs the work-vector method with the same dimensions of 256:
  - 4 streams x 64 (vector length)
  - Uses as much extra memory as the Earth Simulator
- The Fortran “modulo” function prevented vectorization
  - No big deal… was changed for equivalent “mod” statement
- New dominant routine: **shifti**
  - 54% of the time spent in that routine according to “pat”
  - Was only 11% on the ES

The culprit in **shifti**

- "Unstreamed" and "unvectorized" loop due to nested if blocks:

```
do m=m0,mi
   zetaright=min(2.0*pi,zion(3,m))-zetamax
   zetaleft=zion(3,m)-zetamin
   if( zetaright*zetaleft > 0 )then
      zetaright=zetaright*0.5*pi_inv
      zetaright=zetaright-real(floor(zetaright))
      msend=msend+1
      kzi(msend)=m
      if( zetaright < 0.5 )then
         msendright(1)=msendright(1)+1
         iright(msendright(1))=m
      else
         msendleft(1)=msendleft(1)+1
         ileft(msendleft(1))=m
      endif
   endif
enddo
```

Why such a large impact?

- Same scalar to vector peak performance ratio
- Earth Simulator processor:
  - 8 Gflops/s vector processor (500MHz X 8 pipes X 2 flops/cycle)
  - 1 Gflops/s scalar processor (500MHz X 2 flops/cycle)
  - Scalar to vector peak performance ratio = 1/8
- CRAY X1 SSP:
  - 3.2 Gflops/s vector (800MHz X 2 pipes X 2 flops/cycle)
  - 0.4 Gflops/s scalar (400 Mhz X 1 flop/cycle)
  - Scalar to vector peak performance ratio = 1/8
- However, in MSP mode, a loop that does NOT stream and does NOT vectorize uses only 1 of 4 SSPs, giving an effective scalar to vector ratio of only 1/32!

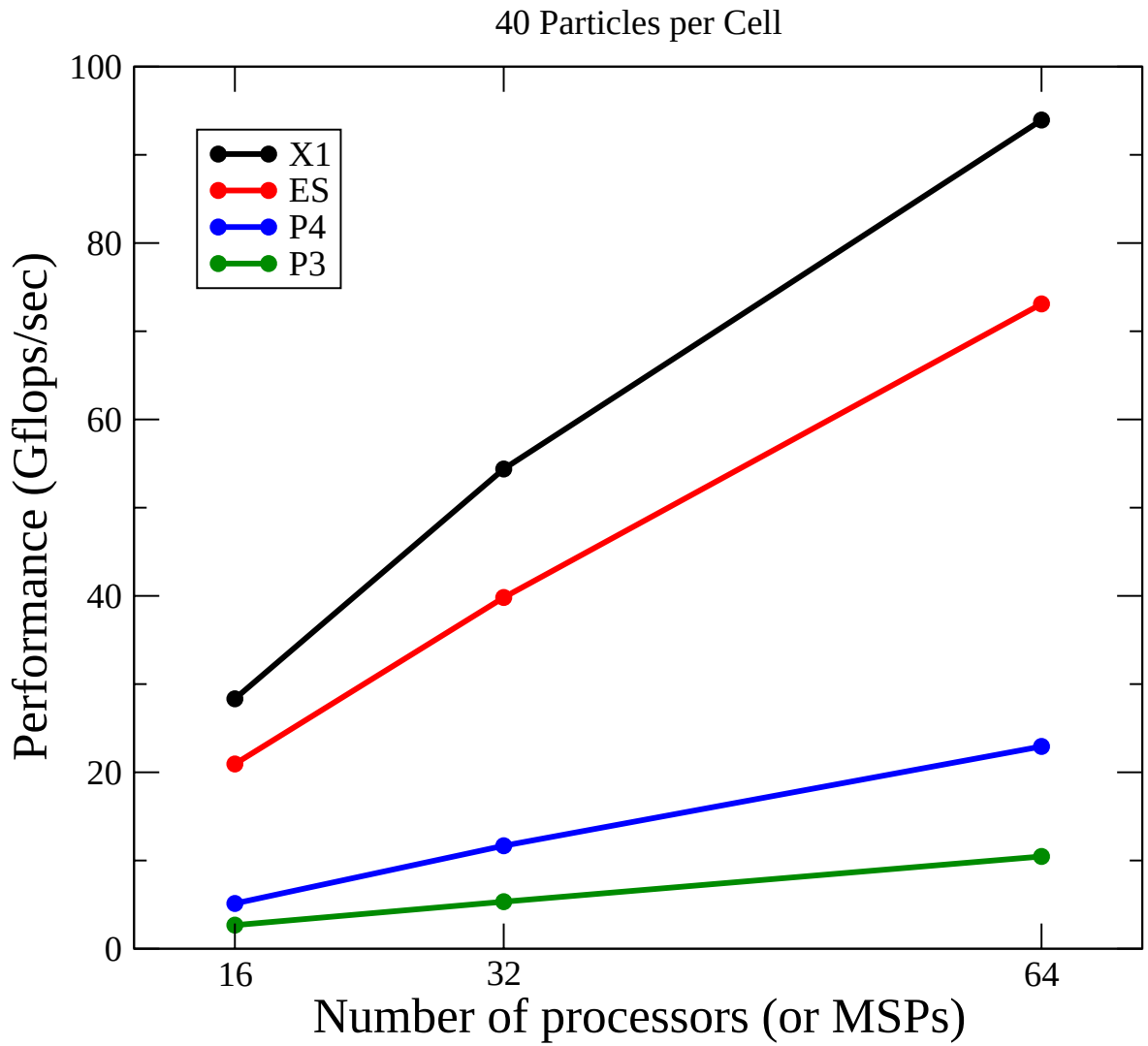
New loop in shift

```
!dir$ preferstream
      do imm=1,4
!dir$ prefervector
         do m=(imm-1)*mi/4+1,imm*mi/4
            zetaright=min(2.0*pi,zion(3,m))-zetamax
            zetaleft=zetamin-zion(3,m)
            alpha=pi2*aint(1.0-pi4_inv*zetaleft)
            beta=pi2*aint(1.0-pi4_inv*zetaright)
            kappa=pi2*aint(1.0+zetaleft*zetaright*pi2sq_inv)
            aright=(alpha+zetaleft) - (beta+zetaright) - kappa
            aleft=(alpha+zetaleft) - (beta+zetaright) + kappa
            if( aright > 0.0 )then
               msend_r(imm)=msend_r(imm)+1
               kzi_r(msend_r(imm),imm)=m
            endif
            if( aleft < 0.0 )then
               msend_l(imm)=msend_l(imm)+1
               kzi_l(msend_l(imm),imm)=m
            endif
         enddo
      enddo
```

Did it work?

- Yes, the overall time spent in shifti went from 54% to only 4%!!
- This new algorithm has not been tested on the Earth Simulator but will certainly have a positive effect as well, although not as dramatic.

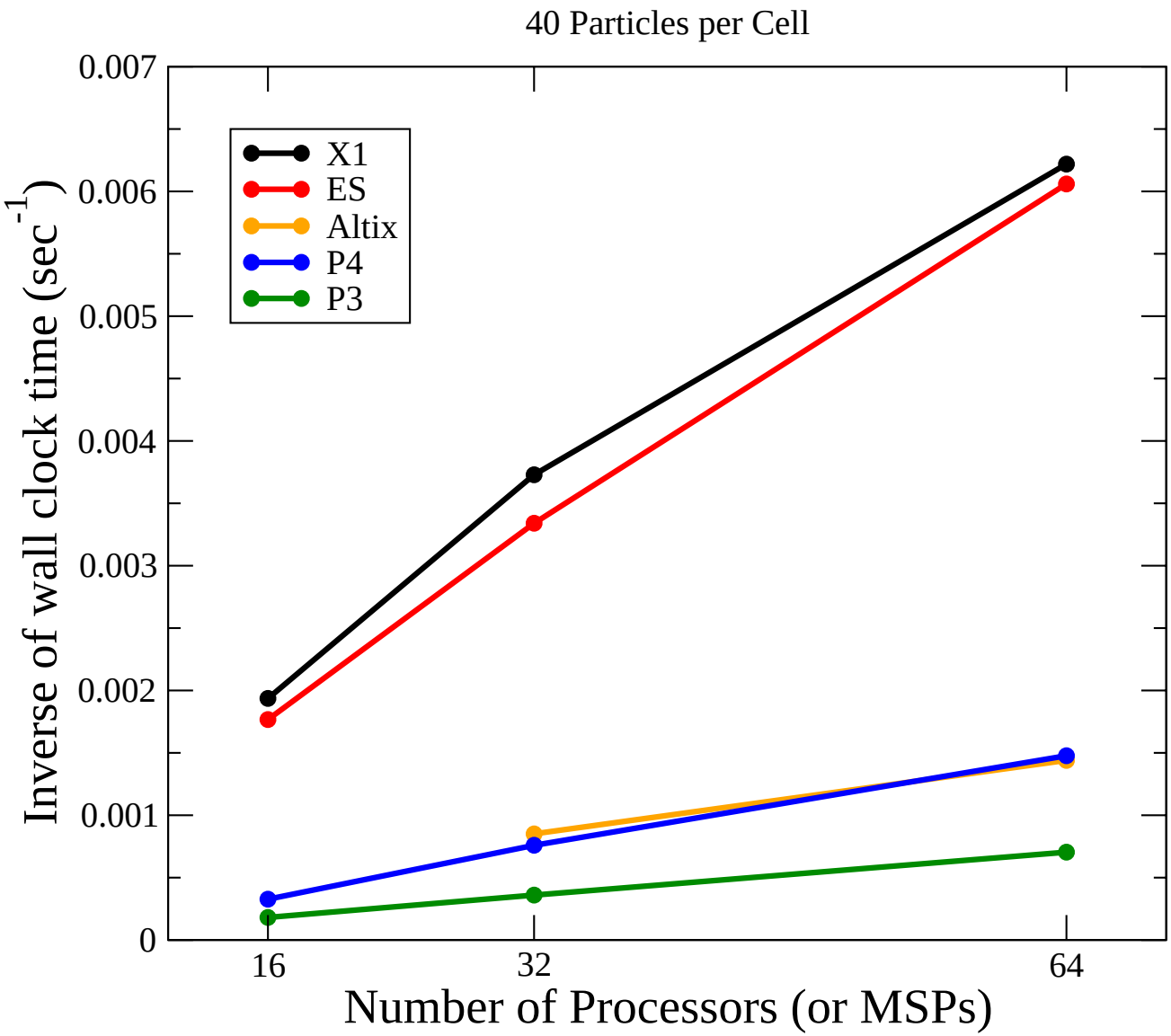
Results: Flops/sec count based on CPU time



Cray X1 is the fastest!
ES scales somewhat better

Test case
- 2,076,736 grid pts
- 100 time steps

Results: Inverse of wall clock time



Scaling

- As the number of processors increases, the domain decomposition reduces the size of the vector loops. The observed scaling is mainly due to a decrease in vector efficiency caused by the smaller loops rather than poor MPI communications.
- Fastest per-processor performance of any tested architecture so far.

The 64-processor test with 100 particles per cell on the Cray X1 and the Earth Simulator runs >1.20 times faster than the same test run on 1024 cpus on the Power 3 Seaborg!!

The numbers…

2,076,736 grid pts

Part cell	P	Power 3		Power4		ES		Cray X1	
		Mflop/s/P	tP3/tES	Mflop/s/P	tP4/tES	Mflop/s/P	%peak	Mflop/s/P	tES/tX1
10	32	134	7.2	280	3.4	961	12.0	1223	1.04
10	64	130	6.3	279	3.0	823	10.3	973	0.96
100	32	135	9.9	281	4.8	1344	16.8	1871	1.13
100	64	133	9.4	274	4.5	1245	15.6	1712	1.09

- **Single processor efficiency now at 18% of peak**
- **ES and X1 10 times faster than Seaborg**

tP3: wall clock time on the Power 3
tP4: wall clock time on the Power 4

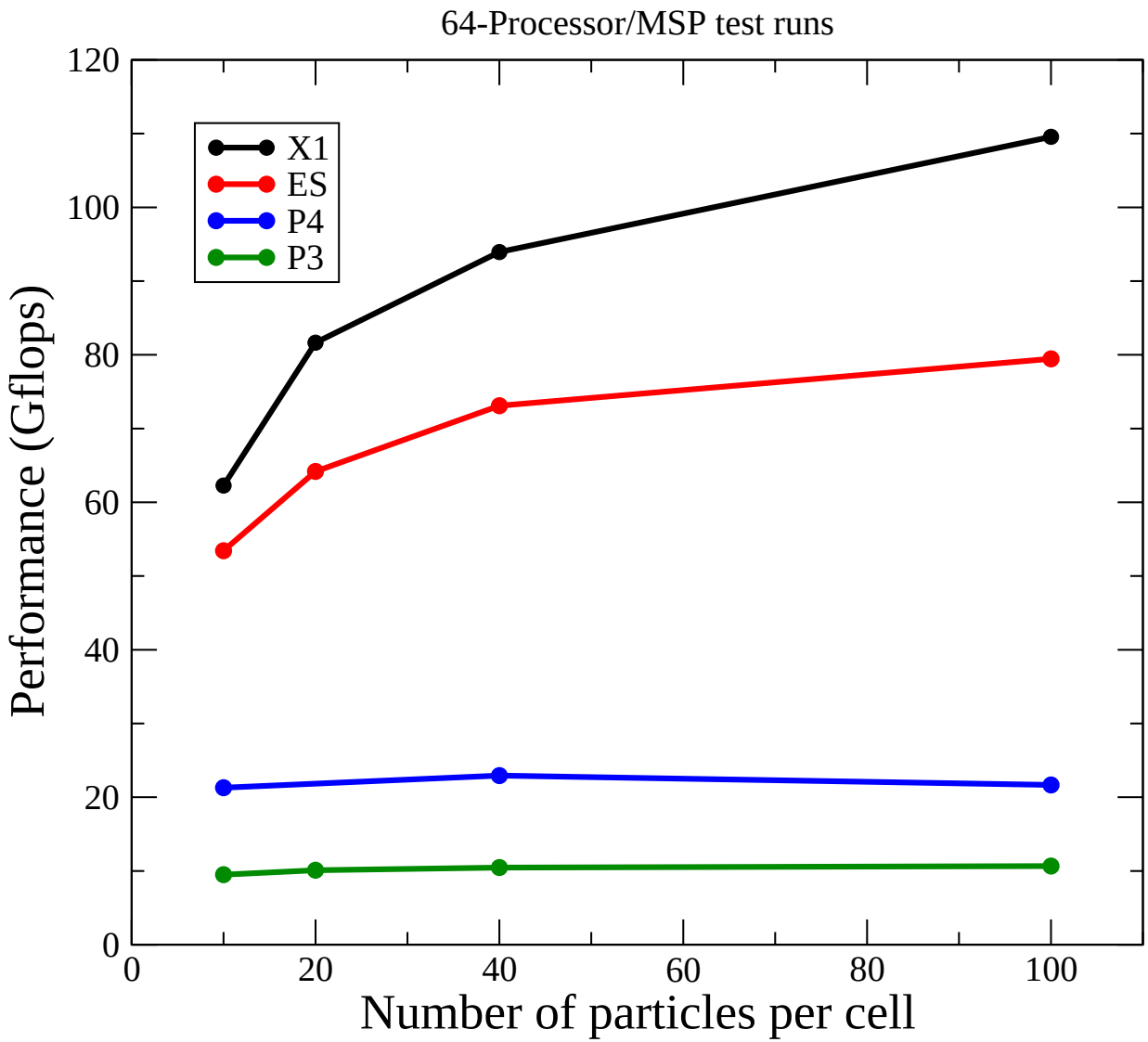
tES: wall clock time on the ES
tX1: wall clock time on the Cray X1

Results: Average vector length and vector operation ratio

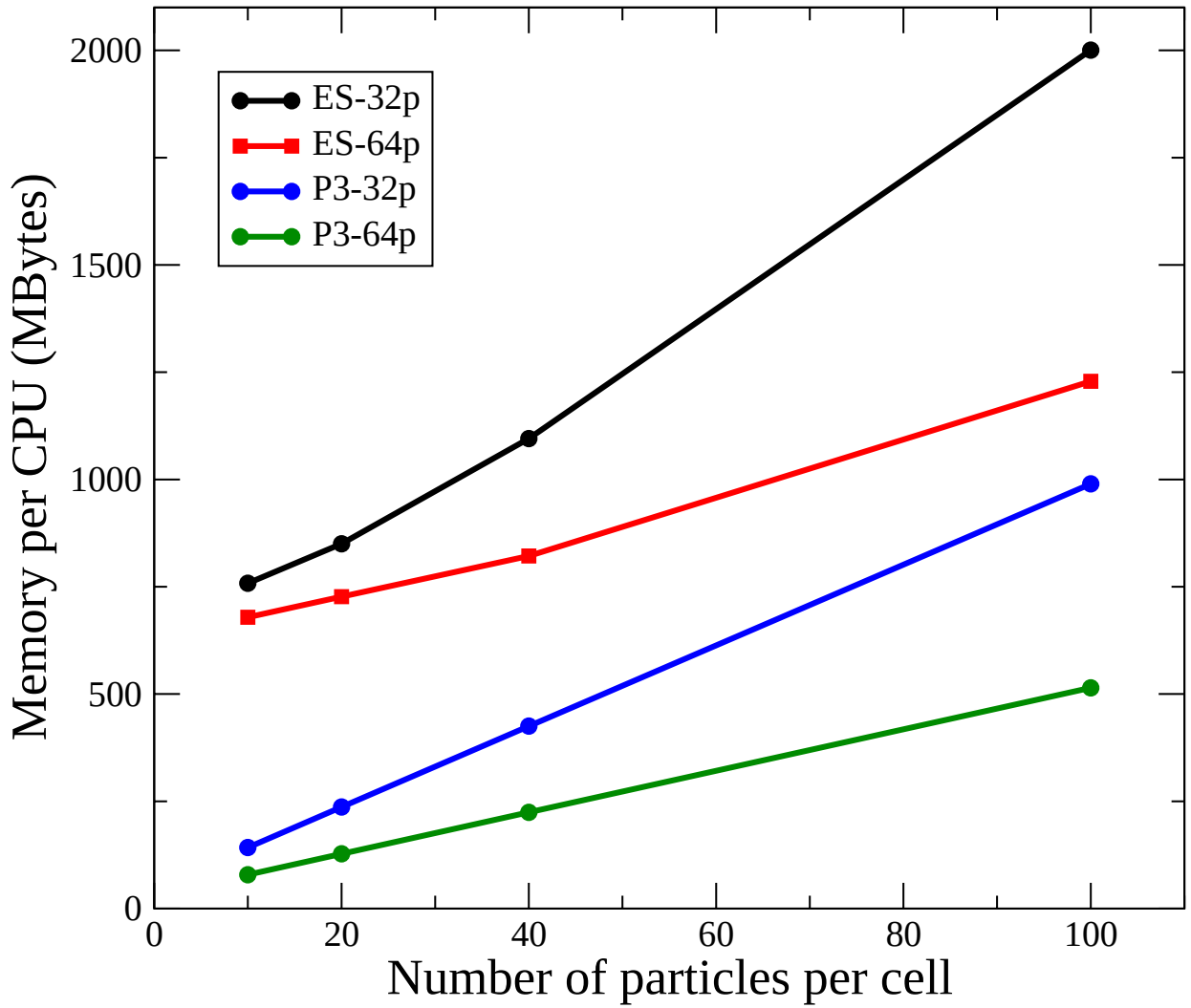
Part cell	P	ES			Cray X1		
		Mflop/s/P	AVL	VOR	Mflop/s/P	AVR	VOR
10	32	961	209.9	98.14	1223	56.6	95.29
10	64	823	184.2	97.48	973	56.5	94.00
100	32	1344	240.7	98.85	1871	62.4	96.83
100	64	1245	228.5	98.59	1712	61.7	96.52

- Better performance with a higher number of particles per cell

Results: Flops/sec for higher particle resolution



Memory used by the vectorized version of GTC (per processor)



- For micell=10 memory on the ES is up to 8 times more than one the Power 3!
- It gets better as the number of particles per cell increases

Conclusions

- Particle-in-cell is a very powerful method to study plasma micro-turbulence but it is a challenge to all types of processors because of its gather/scatter operations.
- However, let's not forget the most important: **time to solution for a given resolution/accuracy!**
- The modifications made to the code have been very successful but require a lot of extra memory, from 2 to 8 times what is used by the version running on the Power 3. Sorting may be the way to go…
- Need to get the OpenMP going as the next step.
- Try out Co-Array Fortran…
- The X1 has the fastest per-processor performance of any architecture tested so far!!