Liquid Argon Molecular Dynamics

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

http://sites.google.com/site/akohlmey/

akohlmey@ictp.it

The Abdus Salam International Centre for Theoretical Physics

Workshop on Computer Programming and Advanced Tools for Scientific Research Work

Contents of this Show

0) Overture: The physics of the model

1) First Act: Writing and optimizing a serial code

2) Intermezzo: Improve scaling with system size

3) Second Act: MPI parallelization

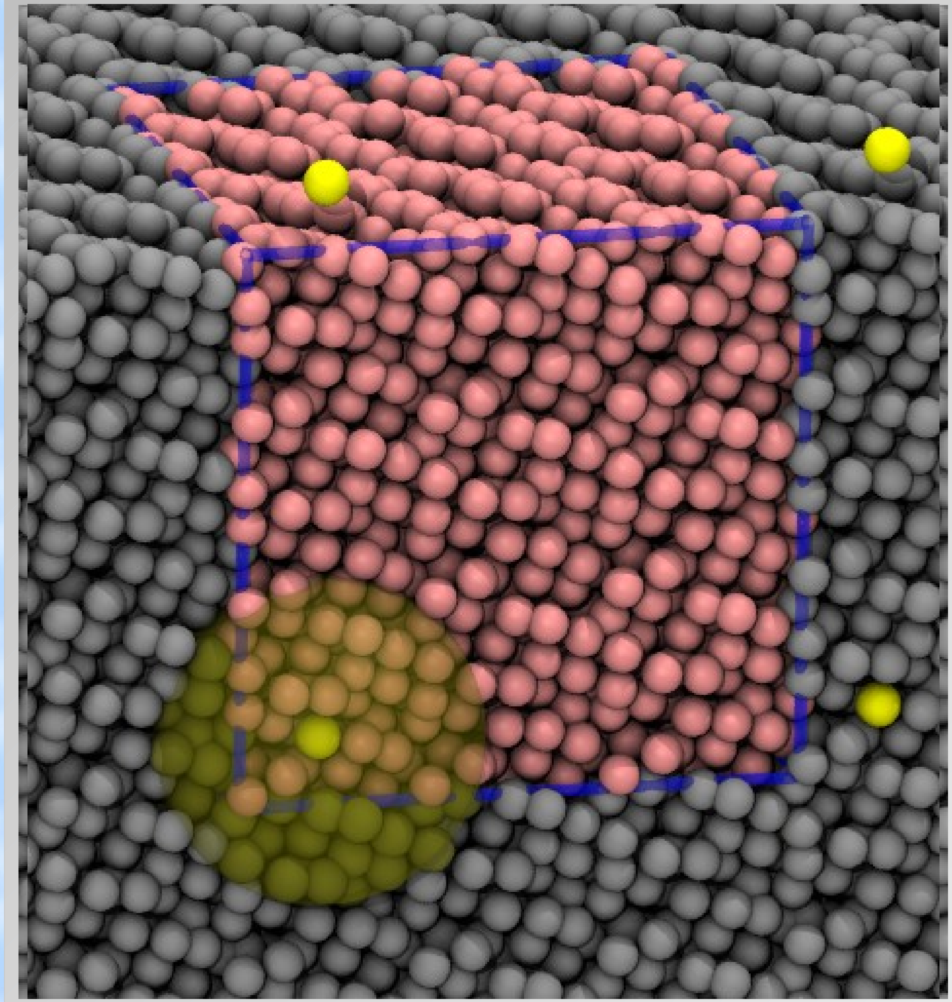
4) Third Act: OpenMP parallelization

5) Finale: Hybrid MPI/OpenMP parallelization

6) Encore: Lessons learned

7) ...and now for something completely different

0) The Model for Liquid Argon



- Cubic box of particles with a Lennard-Jones type pairwise additive interaction potential

$$U(r)=\sum_{i,j}\begin{cases}4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12}-\left(\frac{\sigma}{r_{ij}}\right)^{6}\right], & r_{ij}<r_c\\ 0, & r_{ij}\geq r_c\end{cases}$$

- Periodic boundary conditions to avoid surface effects

Newton's Laws of Motion

- We consider our particles to be *classical objects* so Newton's laws of motion apply:

  1. In absence of a force a body rests or moves in a straight line with constant velocity

  2. A body experiencing a force **F** experiences an acceleration **a** related to **F** by **F** = *m***a**, where *m* is the mass of the body.

  3. Whenever a first body exerts a force **F** on a second body, the second body exerts a force **−F** on the first body (*Bonus Law*)

Velocity-Verlet Algorithm

- The Velocity-Verlet algorithm is used to propagate positions and velocities of the atoms

$$\vec{x}_i(t+\Delta t) = \vec{x}_i(t) + \vec{v}_i(t)\Delta t + \frac{1}{2}\vec{a}_i(t)(\Delta t)^2$$

$$\vec{v}_i\left(t+\frac{\Delta t}{2}\right) = \vec{v}_i(t) + \frac{1}{2}\vec{a}_i(t)\Delta t$$

$$\vec{x}_i(t+\Delta t) = \vec{x}_i(t) + \vec{v}_i\left(t+\frac{\Delta t}{2}\right)\Delta t$$

$$\vec{a}_i(t+\Delta t) = -\frac{1}{m}\nabla V(\vec{x}_i(t+\Delta t))$$

Force calculation

$$\begin{cases} 4\epsilon\left[-12\left(\frac{\sigma}{r_{ij}}\right)^{13} + 6\left(\frac{\sigma}{r_{ij}}\right)^{7}\right], & r_{ij} < r_c \\ 0, & r_{ij} \geq r_c \end{cases}$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\vec{a}_i(t+\Delta t)\Delta t$$

L. Verlet, Phys. Rev. 159, 98 (1967); Phys. Rev. 165, 201 (1967).

What Do We Need to Program?

1. Read in parameters and initial status and compute what is missing (e.g. accelerations)
2. Integrate Equations of motion with Velocity Verlet for a given number of steps
   a) Propagate all velocities for half a step
   b) Propagate all positions for a full step
   c) Compute forces on all atoms to get accelerations
   d) Propagate all velocities for half a step
   e) Output intermediate results, if needed

1) Initial Serial Code: Velocity Verlet

```
void velverlet(mdsys_t *sys) {
    for (int i=0; i<sys->natoms; ++i) {
        sys->vx[i] += 0.5*sys->dt / mvsq2e * sys->fx[i] / sys->mass;
        sys->vy[i] += 0.5*sys->dt / mvsq2e * sys->fy[i] / sys->mass;
        sys->vz[i] += 0.5*sys->dt / mvsq2e * sys->fz[i] / sys->mass;
        sys->rx[i] += sys->dt*sys->vx[i];
        sys->ry[i] += sys->dt*sys->vy[i];
        sys->rz[i] += sys->dt*sys->vz[i];
    }

    force(sys);

    for (int i=0; i<sys->natoms; ++i) {
        sys->vx[i] += 0.5*sys->dt / mvsq2e * sys->fx[i] / sys->mass;
        sys->vy[i] += 0.5*sys->dt / mvsq2e * sys->fy[i] / sys->mass;
        sys->vz[i] += 0.5*sys->dt / mvsq2e * sys->fz[i] / sys->mass;
    }
}
```

Initial Code: Force Calculation

```
double pbc(double x, const double boxby2) {
   while (x >  boxby2) x -= boxby2 + boxby2;
   while (x < -boxby2) x += boxby2 + boxby2;
   return x;
}
```

```
for(i=0; i < (sys->natoms); ++i) {
   for(j=0; j < (sys->natoms); ++j) {
      if (i==j) continue;

      rx=pbc(sys->rx[i] - sys->rx[j], 0.5*sys->box);
      ry=pbc(sys->ry[i] - sys->ry[j], 0.5*sys->box);
      rz=pbc(sys->rz[i] - sys->rz[j], 0.5*sys->box);
      r = sqrt(rx*rx + ry*ry + rz*rz);

      if (r < sys->rcut) {
         ffac = -4.0*sys->epsilon*(-12.0*pow(sys->sigma/r,12.0)/r
                       +6*pow(sys->sigma/r,6.0)/r);
         sys->epot += 0.5*4.0*sys->epsilon*(pow(sys->sigma/r,12.0)
                       -pow(sys->sigma/r,6.0));
         sys->fx[i] += rx/r*ffac;
         sys->fy[i] += ry/r*ffac;
         sys->fz[i] += rz/r*ffac;
}}
```

Compute distance between atoms i & j

Compute energy and force

Add force contribution of atom j on atom i

How Well Does it Work?

- Compiled with:
  `gcc -o ljmd.x -pg ljmd.c -lm`
  Test input: 108 atoms, 10000 steps: 49s
  Let us get a profile (using gprof):

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
73.70	13.87	13.87	10001	1.39	1.86	force
24.97	18.57	4.70	346714668	0.00	0.00	pbc
0.96	18.75	0.18				main
0.37	18.82	0.07	10001	0.01	0.01	ekin
0.00	18.82	0.00	30006	0.00	0.00	azzero
0.00	18.82	0.00	101	0.00	0.00	output
0.00	18.82	0.00	12	0.00	0.00	getline

Step One: Compiler Optimization

- Use of pbc() is convenient, but costs 25% time
  => compiling with -O3 should inline it
- Loops should be unrolled for superscalar CPUs
  => compiling with -O2 or -O3 should do it for us

  Time now: 39s (1.3x faster) Only a bit faster than 49s
- Now try more aggressive optimization options:
  -ffast-math -fexpensive-optimizations -msse3

  Time now: 10s (4.9x faster) Much better!
- Compare to LAMMPS: 3.6s => need to do more

Now Modify the Code

- Use physics! Newton's 3rd law: $F_{ij} = -F_{ji}$

```
for(i=0; i < (sys->natoms)-1; ++i) {
  for(j=i+1; j < (sys->natoms); ++j) {
    rx=pbc(sys->rx[i] - sys->rx[j], 0.5*sys->box);
    ry=pbc(sys->ry[i] - sys->ry[j], 0.5*sys->box);
    rz=pbc(sys->rz[i] - sys->rz[j], 0.5*sys->box);
    r = sqrt(rx*rx + ry*ry + rz*rz);
    if (r < sys->rcut) {
      ffac = -4.0*sys->epsilon*(-12.0*pow(sys->sigma/r,12.0)/r
                               +6*pow(sys->sigma/r,6.0)/r);
      sys->epot +=      4.0*sys->epsilon*(pow(sys->sigma/r,12.0)
                               -pow(sys->sigma/r,6.0));
      sys->fx[i] += rx/r*ffac;      sys->fx[j] -= rx/r*ffac;
      sys->fy[i] += ry/r*ffac;      sys->fy[j] -= ry/r*ffac;
      sys->fz[i] += rz/r*ffac;      sys->fz[j] -= rz/r*ffac;
}}}
```

Time now: 5.4s (9.0x faster) Another big improvement

More Modifications

- Avoid expensive math: pow(), sqrt(), division

```
c12=4.0*sys->epsilon*pow(sys->sigma,12.0);
c6 =4.0*sys->epsilon*pow(sys->sigma, 6.0);
rcsq = sys->rcut * sys->rcut;
for(i=0; i < (sys->natoms)-1; ++i) {
  for(j=i+1; j < (sys->natoms); ++j) {
    rx=pbc(sys->rx[i] - sys->rx[j], 0.5*sys->box);
    ry=pbc(sys->ry[i] - sys->ry[j], 0.5*sys->box);
    rz=pbc(sys->rz[i] - sys->rz[j], 0.5*sys->box);
    rsq = rx*rx + ry*ry + rz*rz;
    if (rsq < rcsq) {
     double r6,rinv; rinv=1.0/rsq;  r6=rinv*rinv*rinv;
     ffac = (12.0*c12*r6 - 6.0*c6)*r6*rinv;
     sys->epot += r6*(c12*r6 - c6);
     sys->fx[i] += rx*ffac;  sys->fx[j] -= rx*ffac;
     sys->fy[i] += ry*ffac;  sys->fy[j] -= ry*ffac;
     sys->fz[i] += rz*ffac;  sys->fz[j] -= rz*ffac;
}}}
```

=> 108 atoms: 4.0s (12.2x faster) still worth it

Improvements So Far

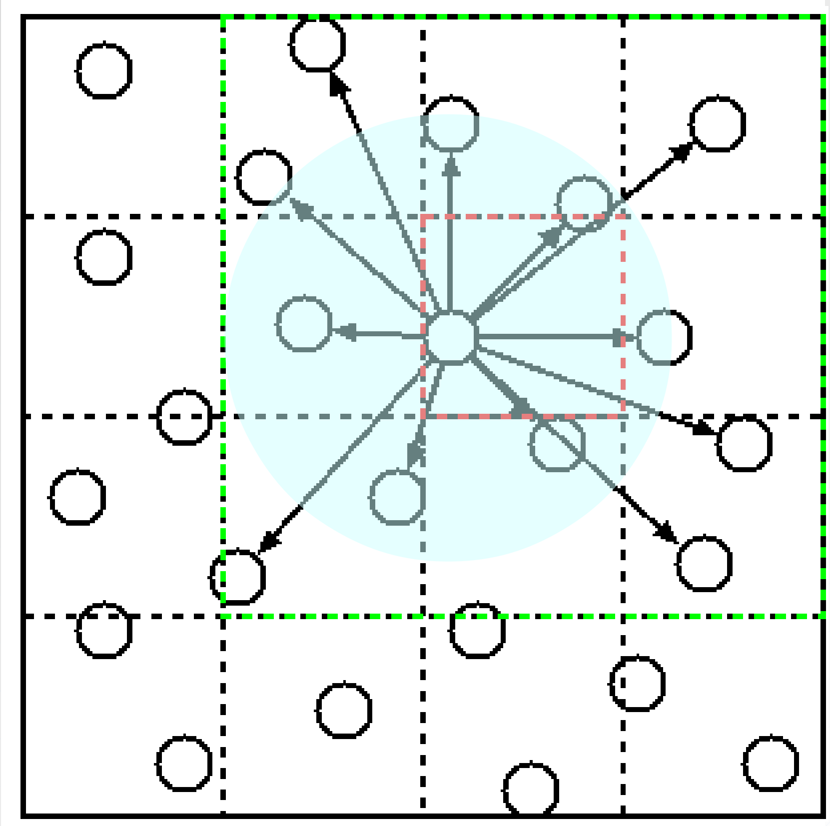
- Use the optimal compiler flags => ~5x faster but some of it: inlining, unrolling could be coded
- Use our knowledge of physics => ~2x faster since we need to compute only half the data.
- Use our knowledge of computer hardware => 1.35x faster. (could be more: SSE/AVX)

  We are within 10% (4s vs. 3.6s) of LAMMPS.
- Try a bigger system: 2916 atoms, 100 steps Our code: 13.3s LAMMPS: 2.7s => Bad scaling with system size

2) Making it Scale with System Size

- Lets look at the algorithm again:
  We compute all distances between pairs
- But for larger systems not all pairs contribute and our effort is $O(N^2)$
- So we need a way to avoid looking at pairs that are too far away

  => Sort atoms into cell lists, which is O(N)



The Cell-List Variant

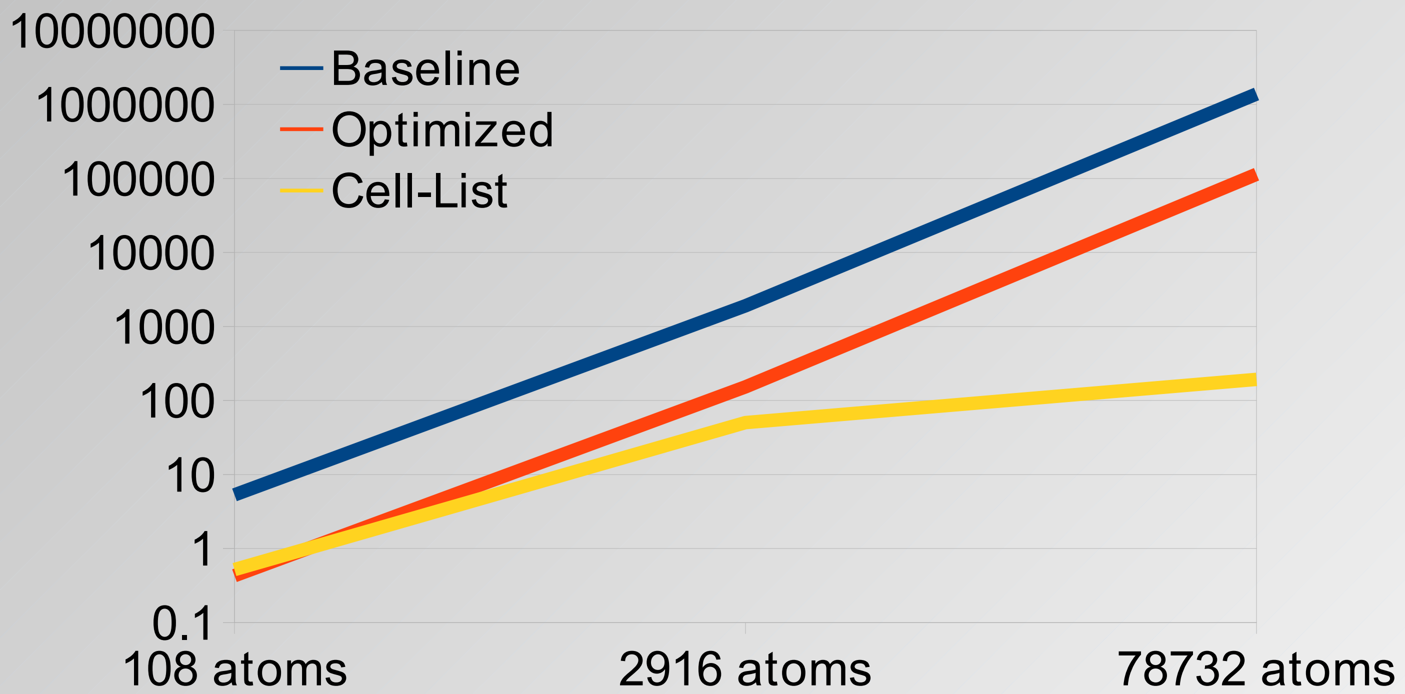
- At startup build a list of lists to store atom indices for atoms that “belong” to a cell
- Compute a list of pairs between cells which contain atoms within cutoff. Doesn't change!
- During MD sort atoms into cells
- Then loop over list of “close” pairs of cells *i* and *j*
- For pair of cells loop over pairs of atoms in them
- Now we have linear scaling with system size at the cost of using more memory and an O(N) sort

Cell List Loop

```
for(i=0; i < sys->npair; ++i) {
    cell_t *c1, *c2;
    c1=sys->clist + sys->plist[2*i];
    c2=sys->clist + sys->plist[2*i+1];

        for (int j=0; j < c1->natoms; ++j) {
            int ii=c1->idxlist[j];
            double rx1=sys->rx[ii];
            double ry1=sys->ry[ii];
            double rz1=sys->rz[ii];

            for(int k=0; k < c2->natoms; ++k) {
                double rx,ry,rz,rsq;
                int jj=c2->idxlist[k];
                rx=pbc(rx1 - sys->rx[jj], boxby2, sys->box);
                ry=pbc(ry1 - sys->ry[jj], boxby2, sys->box);
                ...
```

- 2916 atom time: 3.4s (4x faster), LAMMPS 2.7s

Scaling with System Size



- Cell list does not help (or hurt) much for small inputs, but is a huge win for larger problems => Lesson: always pay attention to scaling

3) What if optimization is not enough?

- Having linear scaling is nice, but twice the system size is still twice the work and takes twice the time. => Parallelization
- Simple MPI parallelization first
  - MPI is “share nothing” (replicated or distributed data)
  - Run the same code path with the same data but insert a few MPI calls
    - Broadcast positions from rank 0 to all before force()
    - Compute forces on different atoms for each rank
    - Collect (reduce) forces from all to rank 0 after force()

Replicated Data MPI Version

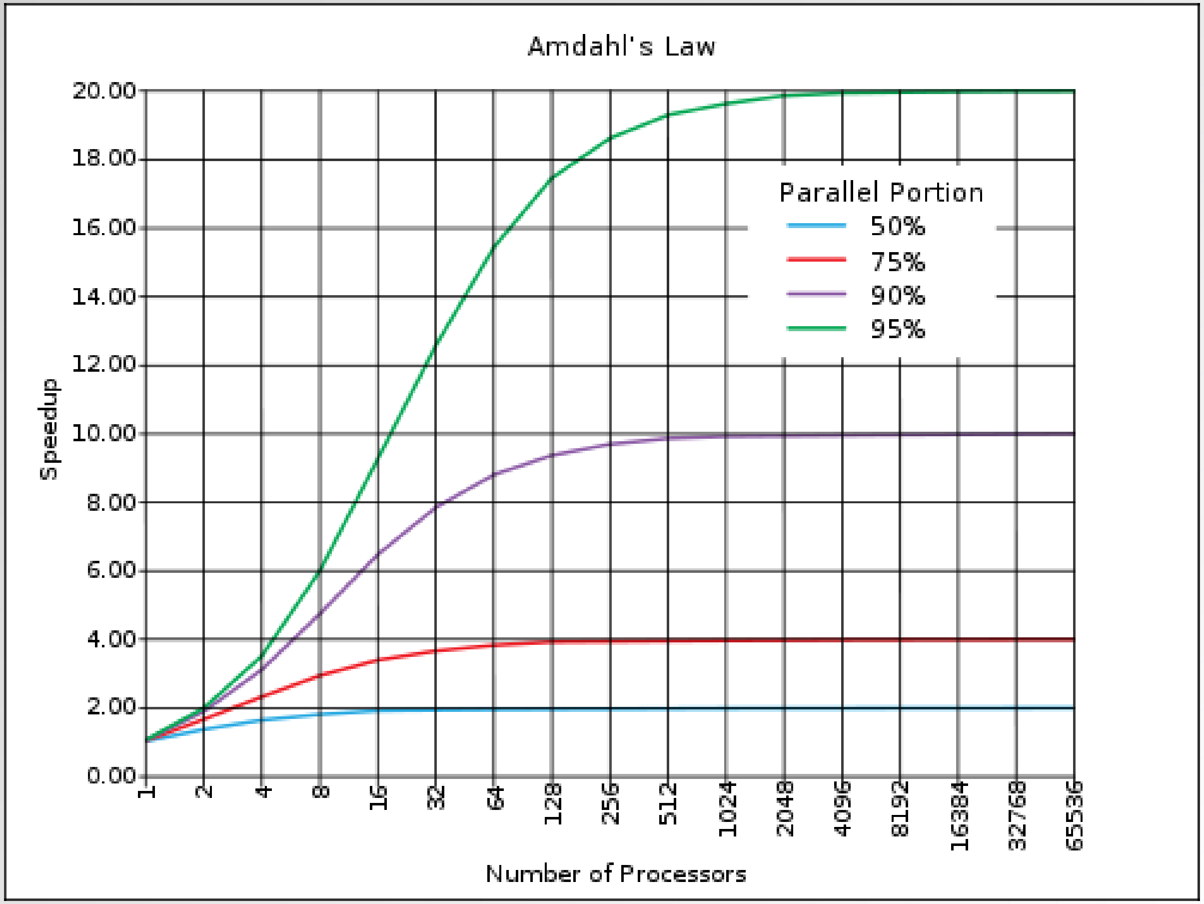
cx/cy/cz on all nodes; fx/fy/fz on master only

```
static void force(mdsys_t *sys) {
  double epot=0.0;
  azzero(sys->cx,sys->natoms); azzero(sys->cy,sys->natoms); azzero(sys->cz,sys->natoms);
  MPI_Bcast(sys->rx, sys->natoms, MPI_DOUBLE, 0, sys->mpicomm);
  MPI_Bcast(sys->ry, sys->natoms, MPI_DOUBLE, 0, sys->mpicomm);
  MPI_Bcast(sys->rz, sys->natoms, MPI_DOUBLE, 0, sys->mpicomm);
  for (i=0; i < sys->natoms-1; i += sys->nsize) {
    ii = i + sys->mpirank;
    if (ii >= (sys->natoms - 1)) break;
    for (j=i+1; i < sys->natoms; ++j) {
   [...]
          sys->cy[j] -= ry*ffac;
          sys->cz[j] -= rz*ffac;
  } }
  MPI_Reduce(sys->cx, sys->fx, sys->natoms, MPI_DOUBLE, MPI_SUM, 0, sys->mpicomm);
  MPI_Reduce(sys->cy, sys->fy, sys->natoms, MPI_DOUBLE, MPI_SUM, 0, sys->mpicomm);
  MPI_Reduce(sys->cz, sys->fz, sys->natoms, MPI_DOUBLE, MPI_SUM, 0, sys->mpicomm);
  MPI_Reduce(&epot, &sys->epot, 1, MPI_DOUBLE, MPI_SUM, 0, sys->mpicomm);
}
```

- Easy to implement, but lots of communication

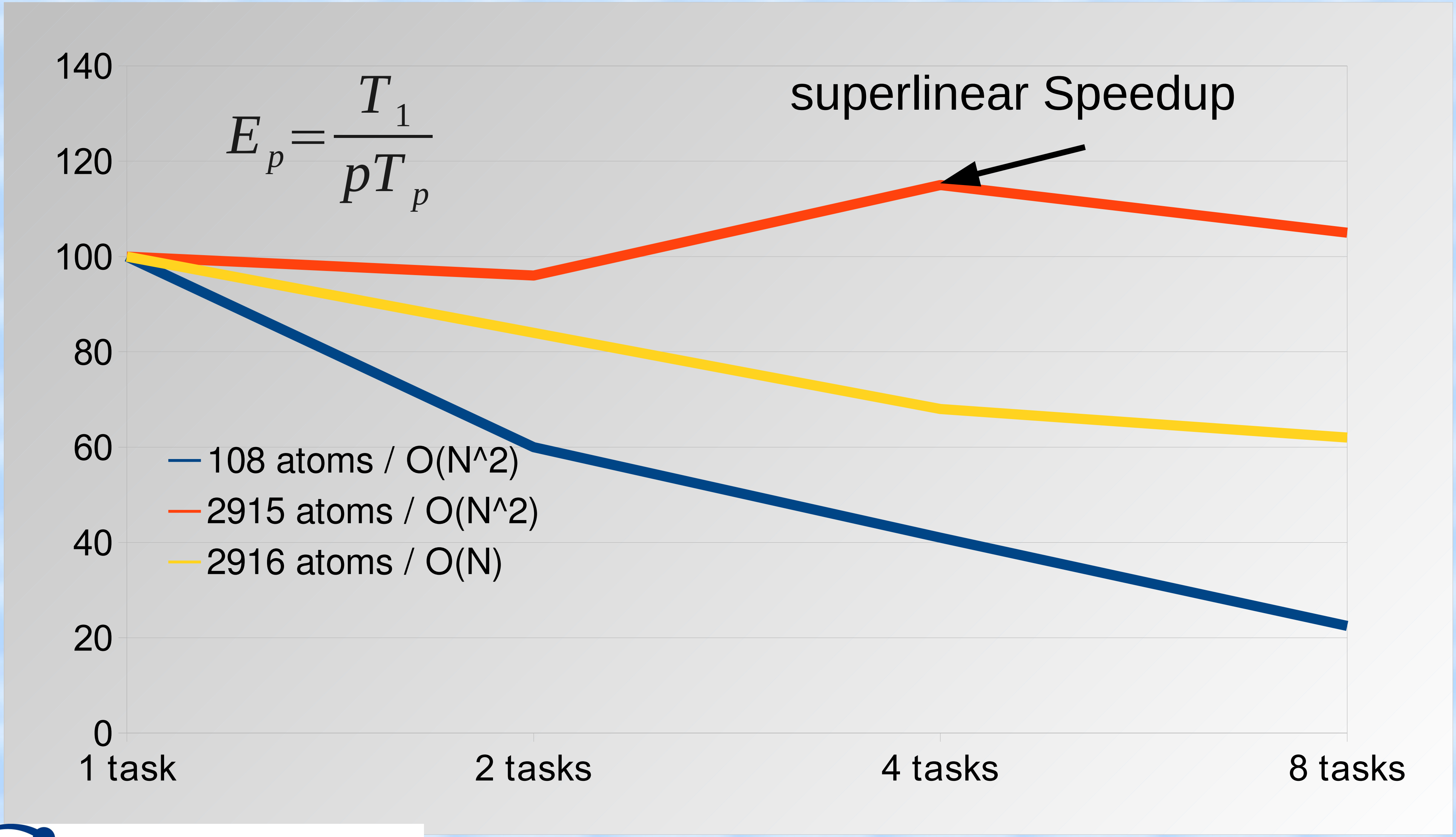
Replicated Data Limitations

- Amdahl's Law (we only parallelized the force computation)

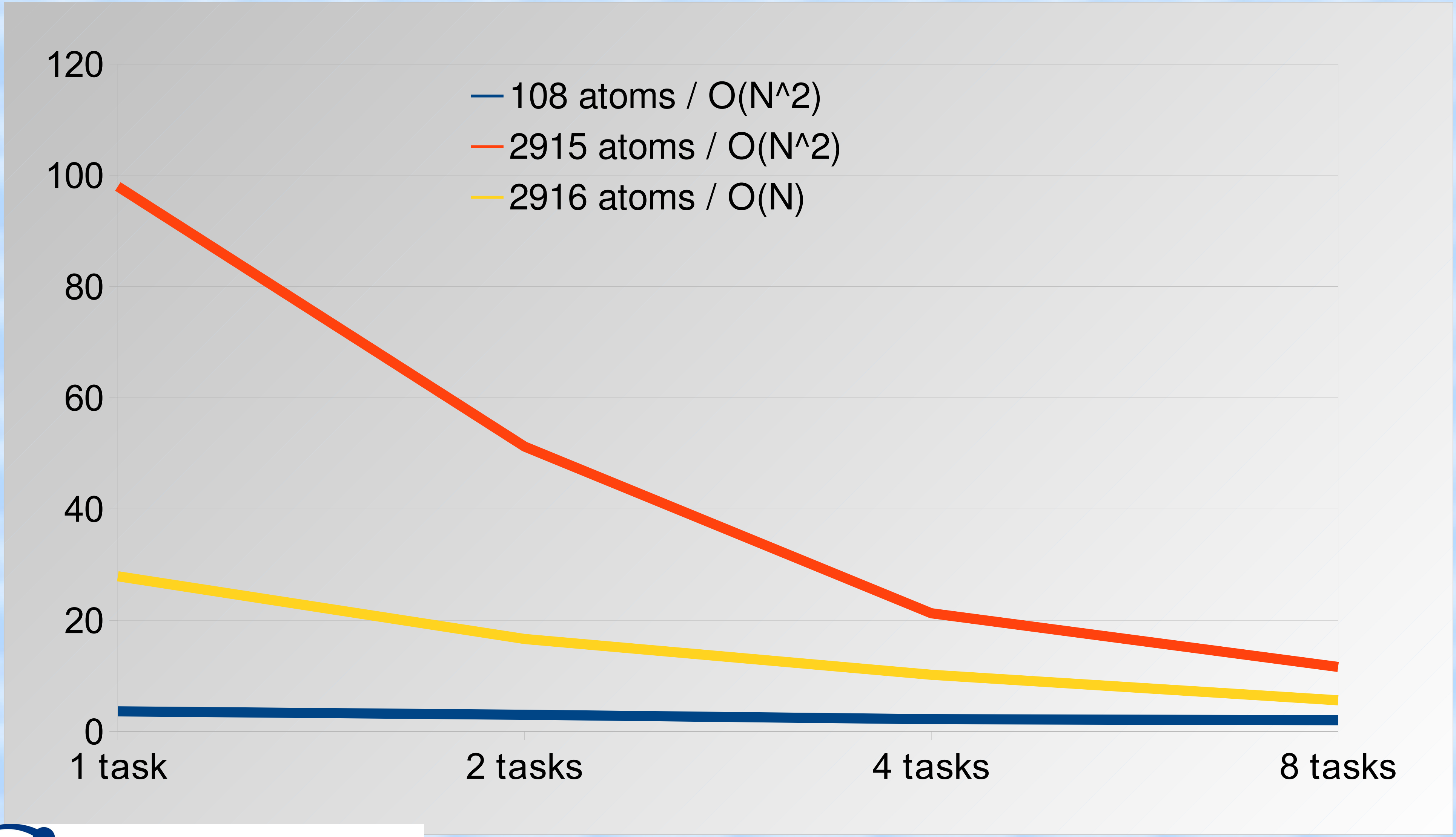


- Parallel overhead (grows with system size):
  - Broadcast
  - Reduction
- Limited scaling

MPI Parallel Efficiency



MPI Parallel Execution Times



4) OpenMP Parallelization

- OpenMP is directive based
  => code (can) work without them
- OpenMP can be added incrementally
- OpenMP only works in shared memory
  => multi-socket nodes, multi-core processors
- OpenMP hides the calls to a threads library
  => less flexible, but much less programming
- Caution: write access to shared data can easily lead to race conditions

Naive OpenMP Version

```
#if defined(_OPENMP)
#pragma omp parallel for default(shared) \
    private(i) reduction(+:epot)
#endif
    for(i=0; i < (sys->natoms)-1; ++i) {
        double rx1=sys->rx[i];
        double ry1=sys->ry[i];
        double rz1=sys->rz[i];
        [...]
```

Each thread will work on different values of "i"

```
#if defined(_OPENMP)
#pragma omp critical
#endif
            {
                sys->fx[i] += rx*ffac;
                sys->fy[i] += ry*ffac;
                sys->fz[i] += rz*ffac;
                sys->fx[j] -= rx*ffac;
                sys->fy[j] -= ry*ffac;
                sys->fz[j] -= rz*ffac;
            }
```

```
                sys->fx[j] -= rx*ffac;
                sys->fy[j] -= ry*ffac;
                sys->fz[j] -= rz*ffac;
            }
```

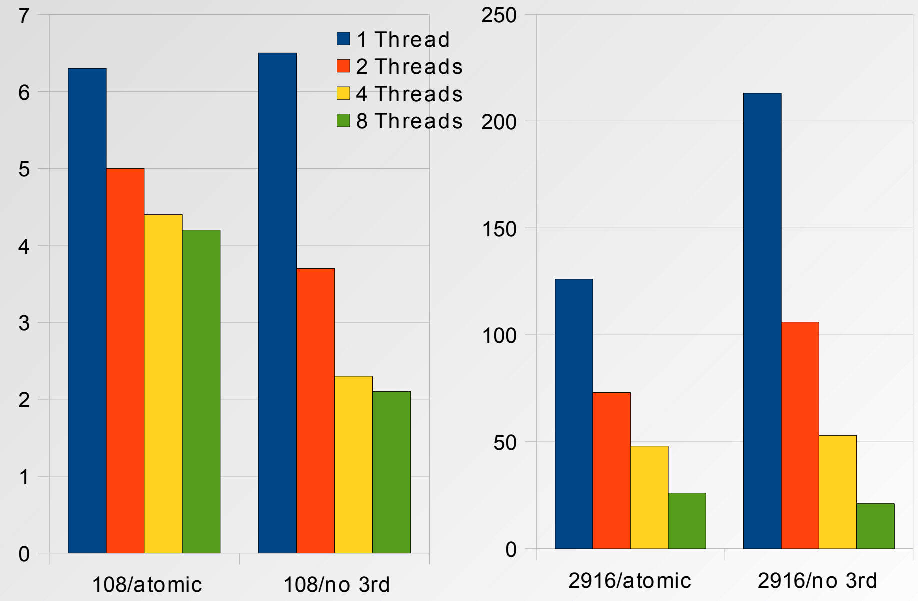
The "critical" directive will let only one thread execute this block at a time

Race condition: "i" will be unique for each thread, but not "j" => multiple threads may write to the same location concurrently

Timings (108 atoms):
1 thread: 4.2s
2 threads: 7.1s
4 threads: 7.7s
8 threads: 8.6s

OpenMP Improvements

- Use **omp atomic** to protect one instruction
  => faster, but requires hardware support
  => some speedup, but serial is faster for 108,
  at 2916 atoms we are often beyond cutoff
- No Newton's 3rd Law:
  => no race condition
  => better scaling, but
  we lose 2x serial speed
  => need 8 threads to
  be faster than **atomic**



The Abdus Salam International Centre for Theoretical Physics

MPI-like Approach with OpenMP

```
#if defined(_OPENMP)
#pragma omp parallel reduction(+:epot)
#endif
    {  double *fx, *fy, *fz;
#if defined(_OPENMP)
        int tid=omp_get_thread_num();
#else
        int tid=0;
#endif
        fx=sys->fx + (tid*sys->natoms); azzero(fx,sys->natoms);
        fy=sys->fy + (tid*sys->natoms); azzero(fy,sys->natoms);
        fz=sys->fz + (tid*sys->natoms); azzero(fz,sys->natoms);
        for(int i=0; i < (sys->natoms -1); i += sys->nthreads) {
            int ii = i + tid;
            if (ii >= (sys->natoms -1)) break;
            rx1=sys->rx[ii];
            ry1=sys->ry[ii];
            rz1=sys->rz[ii];
```

Thread Id is like MPI rank

sys->fx holds storage for one full fx array for each thread => race condition is eliminated.

MPI-like Approach with OpenMP (2)

- We need to write our own reduction:

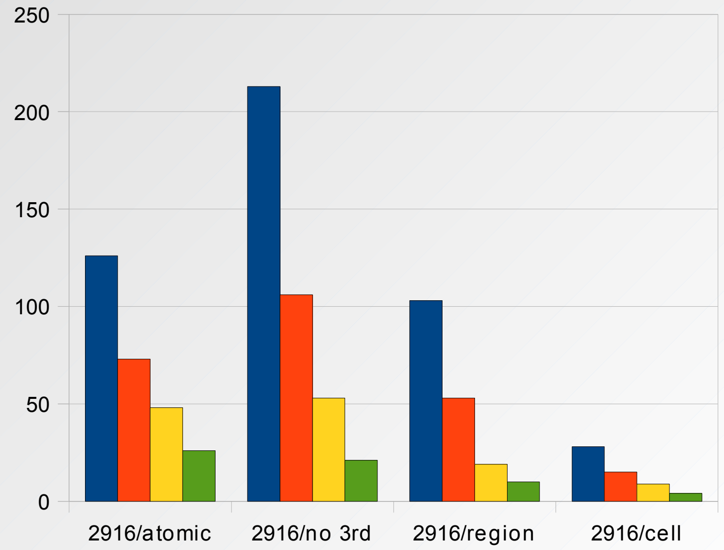
```
#if defined (_OPENMP)
#pragma omp barrier
#endif
    i = 1 + (sys->natoms / sys->nthreads);
    fromidx = tid * i;
    toidx = fromidx + i;
    if (toidx > sys->natoms) toidx = sys->natoms;

    for (i=1; i < sys->nthreads; ++i) {
        int offs = i*sys->natoms;
        for (int j=fromidx; j < toidx; ++j) {
            sys->fx[j] += sys->fx[offs+j];
            sys->fy[j] += sys->fy[offs+j];
            sys->fz[j] += sys->fz[offs+j];
        }
    }
```

Need to make certain, all threads are done with computing forces

Use threads to parallelize the reductions

More OpenMP Timings

- The **omp parallel** region timings
  2916: 1T: 103s, 2T: 53s, 4T: 19s, 8T: 10s
  => better speedup, but serial is faster for 108,
  at 2916 atoms we are often beyond cutoff
- This approach also works with cell lists
  => with 8 threads:
  4.1s = 6.8x speedup vs.
  serial cell list version (28s).
  That is **62x** faster than
  the first naive serial version



6) Hybrid OpenMP/MPI Version

- With multi-core nodes, communication between MPI tasks becomes a problem
  => all communication has to us2 one link
  => reduced bandwidth, increased latency
- OpenMP and MPI parallelization are orthogonal and can be used at the same time
  Caution: don't call MPI from threaded region!
- Parallel region OpenMP version is very similar to MPI version, so that would be easy to merge

Hybrid OpenMP/MPI Kernel

- MPI tasks are like GPU thread blocks
- Need to reduce forces/energies first across threads and then across all MPI tasks

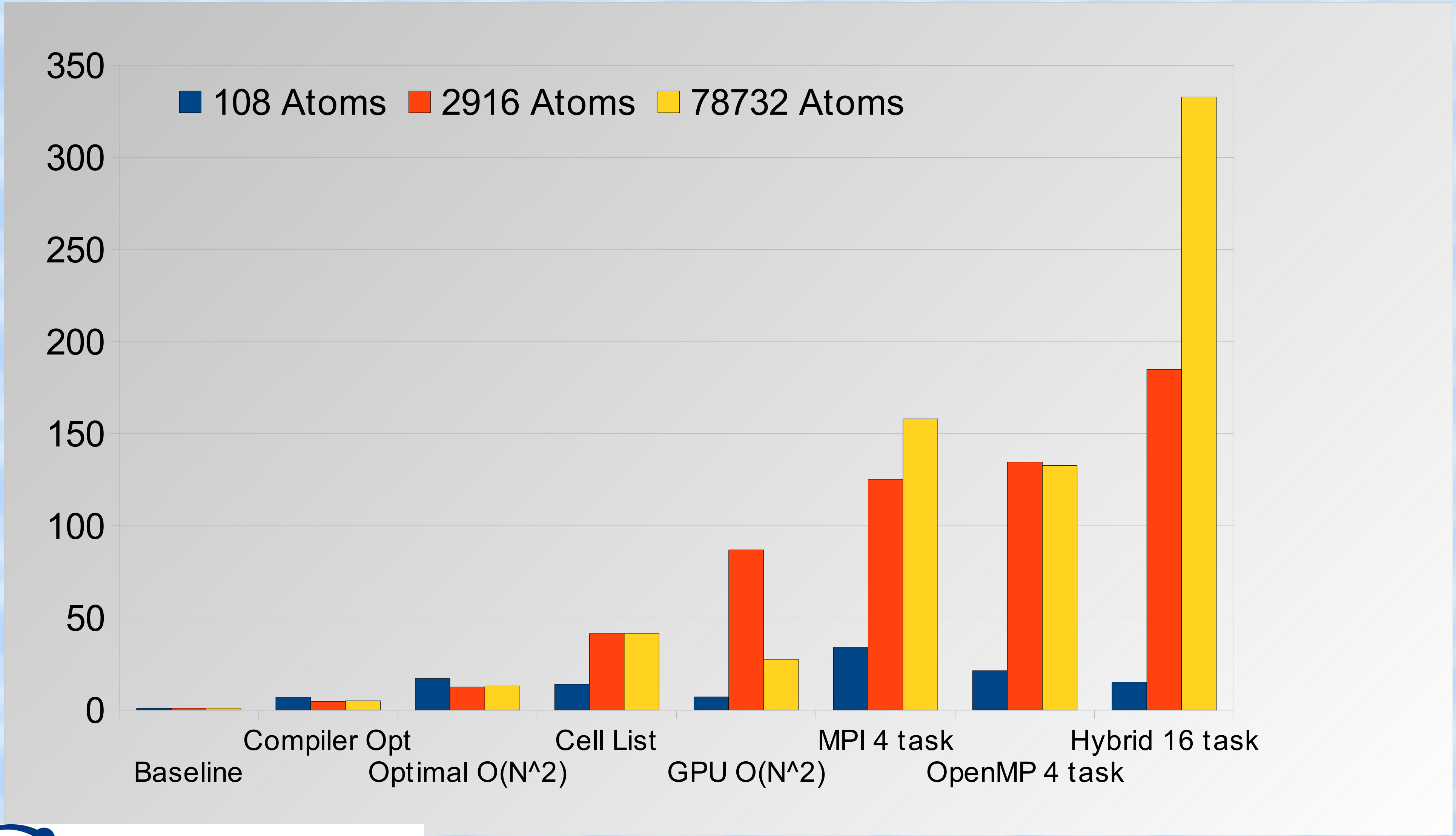
```
[...]
        incr = sys->mpisize * sys->nthreads;
        /* self interaction of atoms in cell */
        for(n=0; n < sys->ncell; n += incr) {
            int i,j;
            const cell_t *c1;

            i = n + sys->mpirank*sys->nthreads + tid;
            if (i >= sys->ncell) break;
            c1=sys->clist + i;

            for (j=0; j < c1->natoms-1; ++j) {
 [...]
```

Hybrid OpenMP/MPI Timings

	2916 atoms system:	78732 atoms system:
Cell list serial code:	18s	50.1s
16 MPI x 1 Threads:	14s	19.8s
8 MPI x 2 Threads:	5.5s	8.9s
4 MPI x 4 Threads:	4.3s	8.2s
2 MPI x 8 Threads:	4.0s	7.3s
=> Best speedup:	4.5x	6.9x
=>Total speedup:	**185x**	**333x**

Two nodes with 2x quad-core

Total Speedup Comparison



What about GPUs?

- GPUs are threading taken to the extreme
- Programming models: CUDA (like C), OpenCL (more explicit but portable across hardware), OpenACC (like OpenMP)
- Need to generate >1000 work units:
  => One (or more) thread(s) per “i atom”
  => good weak scaling, limited strong scaling
- Offload only some kernels (GPU=accelerator) vs. moving entire calculation (CPU=decelerator)
  => depends on problem size, choice of hardware

Conclusions

- Make sure that you exploit the physics of your problem well => Newton's 3rd law gives a 2x speedup for free (but interferes with threading!)
- Let the compiler help you (more readable code), but also make it easy to the compiler
  => unrolling, inlining can be offloaded
- Understand the properties of your hardware and adjust your code to match it
- Best strong scaling on current hardware with hybrid parallelization, e.g. MPI+OpenMP

What Else Can Be Done?

- Vectorization (“the” thing in the 1970s & 1980s)
  - MMX/SSE/AVX instructions allow processing of multiple data elements with one instruction (SIMD) => 64/128/256-bit registers for “packed” data
  - Since Pentium IV: 128-bit SSE2 unit can be used for double precision floating-point math.
  - Recent CPUs support 256-bit AVX and “fused multiply add” (FMA) instructions
  - Xeon Phi (and future CPUs) support 512-bit AVX2
  - Portability issues: different CPUs support different subsets of the vector instructions.

How to Add Vectorization

- Let the compiler do it:
  - On 32-bit need to specify architecture (Pentium IV+) 8 SSE registers supported, SSE2-unit independent of floating-point unit (unlike for MMX/SSE1)
  - On 64-bit SSE2 is supported by all hardware includes 16 SSE2 registers instead of 8 in 32-bit
  - Vectorization requires 16-byte aligned data; if not possible to tell, compiler will generate slower code (default on x86 is 8-byte alignment!)
  - Only addition, subtraction, multiplication, division and (inverse) square root are vectorized

How to Add Vectorization (2)

- Write explicit assembly code
  - Tedious, difficult, non-portable and requires detailed knowledge of the instruction set and the hardware
- Use compiler "intrinsics"
  - Available for C/C++, similar to macros
  - Portable between Microsoft, Intel, GNU compilers
  - d = a + b * c; for 2 double precision values becomes:

```
__m128d v1 = _mm_load_pd(&a); __m128d v2 = _mm_load_pd(&b);
__m128d v3 = _mm_load_pd(&c);
__m128d v4 = _mm_add_pd(v1, _mm_mul_pd(v2,v3));
_mm_store_pd(&d, v4);
```

Why Worry About Vectorization?

- Vector instructions already in the CPU
  => unused acceleration potential
- Programming model somewhat similar to GPU
  => optimization strategies that work well on GPUs should be transferable to vectorization
- OpenCL explicitly supports 3 types of hardware GPU, FPGA, and CPU (with vector unit)