Accelerating classical MD for multi-core CPUs and GPUs

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

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

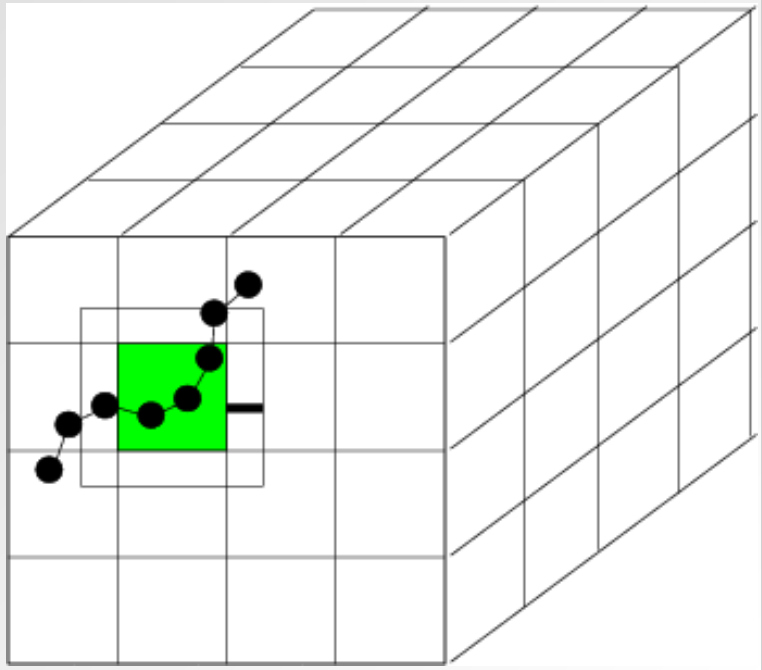
a.kohlmeyer@temple.edu

The Abdus Salam International Centre for Theoretical Physics

LAMMPS Users and Developers Workshop and Symposium, March 24th-28th 2014

Standard LAMMPS Parallelization

- MPI based (MPI emulator for serial execution)
- Uses domain decomposition with 1 domain per MPI task (= processor). Each MPI task looks after the atoms in its domain
- Atoms move from MPI task to MPI task as they move through the system
- Assumes same amount of work (force computations) in each domain.



Why Bother Adding OpenMP?

1. Why **not** do it?
   a) LAMMPS is already very parallel
   b) Even more run-time settings to optimize
   c) OpenMP is often less effective than MPI (for MD)
2. Why do it **anyway**?
   a) On multi-core machines (Cray XT5) LAMMPS can run faster with MPI when some CPU cores are idle
   b) Parallelization over **particles**, not domains
   c) PPPM has scaling limitations. At high node counts it would be better to run it only on a subset of tasks

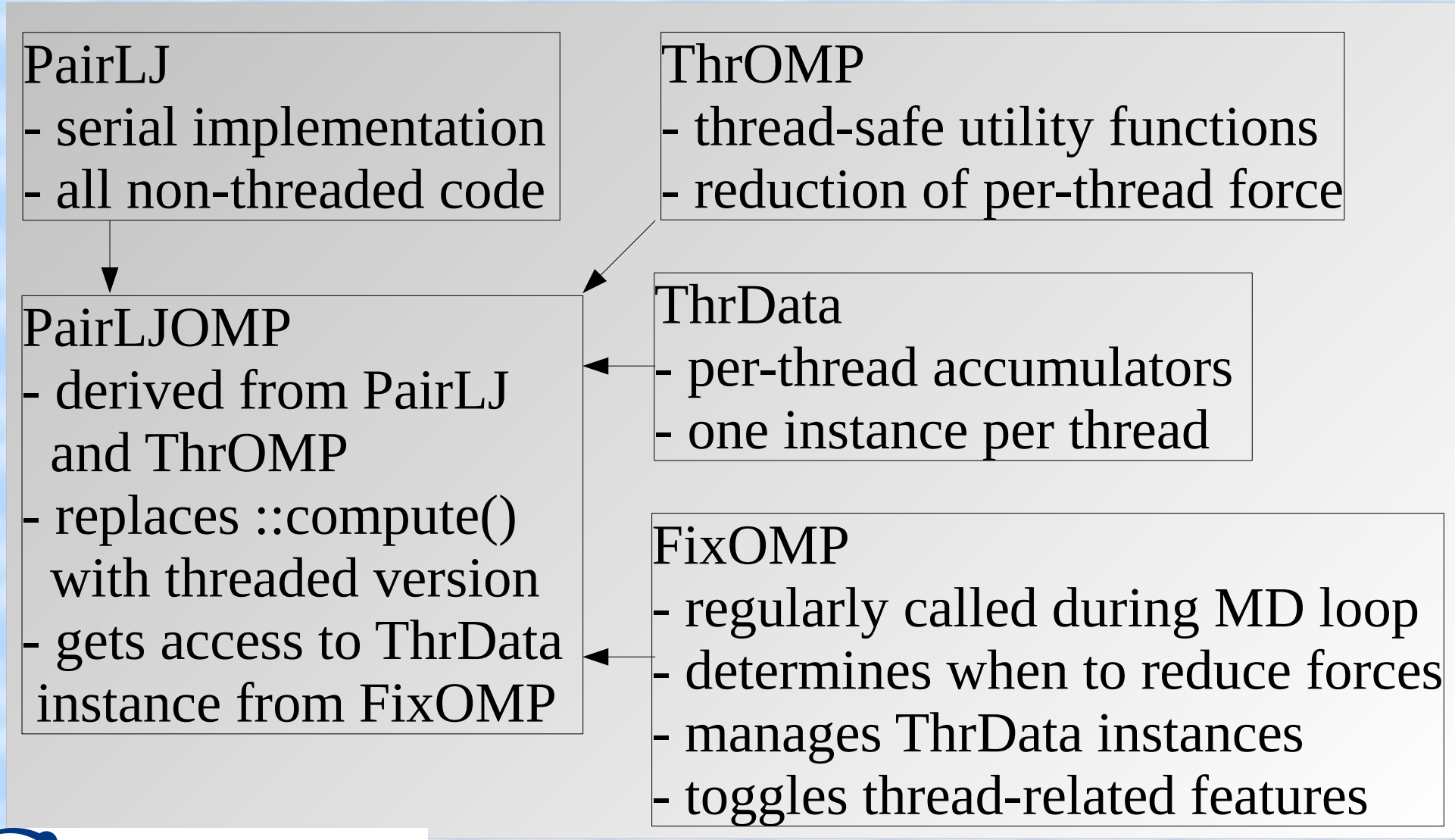
OpenMP Parallelization

- OpenMP is directive based
  => well written code works with or without
- OpenMP can be added incrementally
- OpenMP only works in shared memory
  => multi-core processors are now ubiquitous
- OpenMP hides the calls to a threads library
  => less flexible, more overhead, but less effort
- Caution: need to worry about race conditions, memory corruption, false sharing, Amdahl's law

How to add OpenMP to LAMMPS

- LAMMPS is very modular, just add new classes derived from non-threaded implementation
- Pairwise interactions (consume most time)
  - i,j nested loop over neighbors can be parallelized
  - each thread processes different “i” atoms
- Neighbor list build (binning still serial)
  - i,j nested loop over atoms and neighboring bins
- Dihedrals and other bonded interactions
- Replace selected function(s) in derived class

Threading Class Relations



Naive OpenMP LJ Kernel

```
#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];
        [...]

#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;
    }
}
```

Each thread will work on different values of “i”

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”
Or some “j” may be an “i” of another thread
=> multiple threads update the same location

Timings (108 atoms):

serial:	4.0s
1 thread:	4.2s
2 threads:	7.1s
4 threads:	7.7s
8 threads:	8.6s

Alternatives to “omp critical”

- Use **omp atomic** to protect each force addition
  => requires hardware support (modern x86)
  1Thr: 6.3s, 2Thr: 5.0s, 4Thr: 4.4s, 8Thr: 4.2s
  => faster than **omp critical** for multiple threads
  but it is slower than the serial code (4.0s)
- Don't use Newton's 3rd Law
  => no race condition
  1Thr: 6.5s, 2Thr: 3.7s, 4Thr: 2.3s, 8Thr: 2.1s
  => better scaling, but 2 threads ~= serial speed
  => this is what is done on GPU (many threads)

“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 number is like MPI rank

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

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

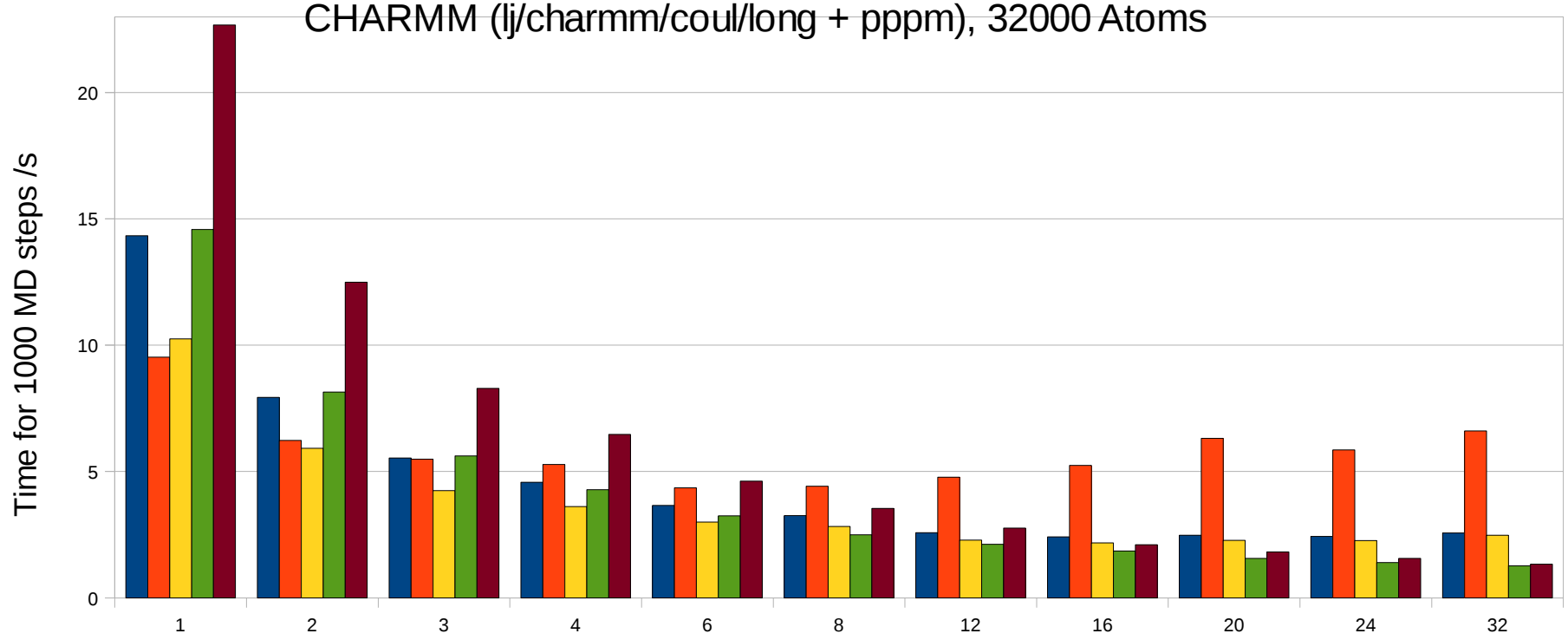
Use threads to parallelize the reductions

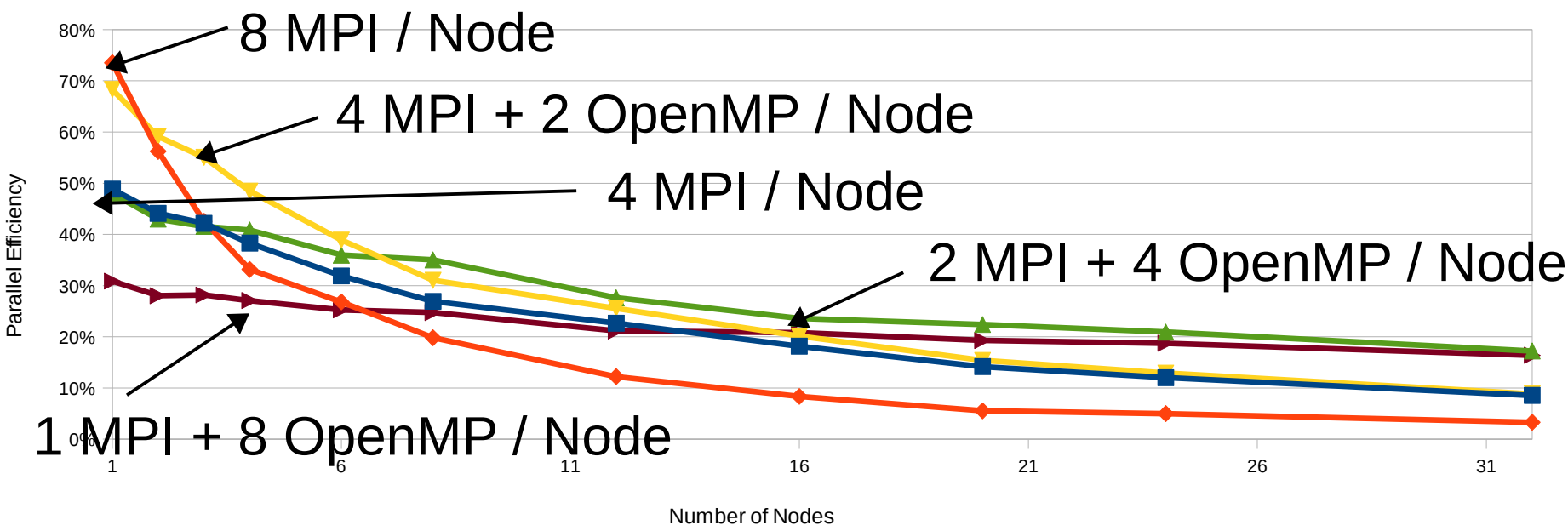
OpenMP Timings Comparison

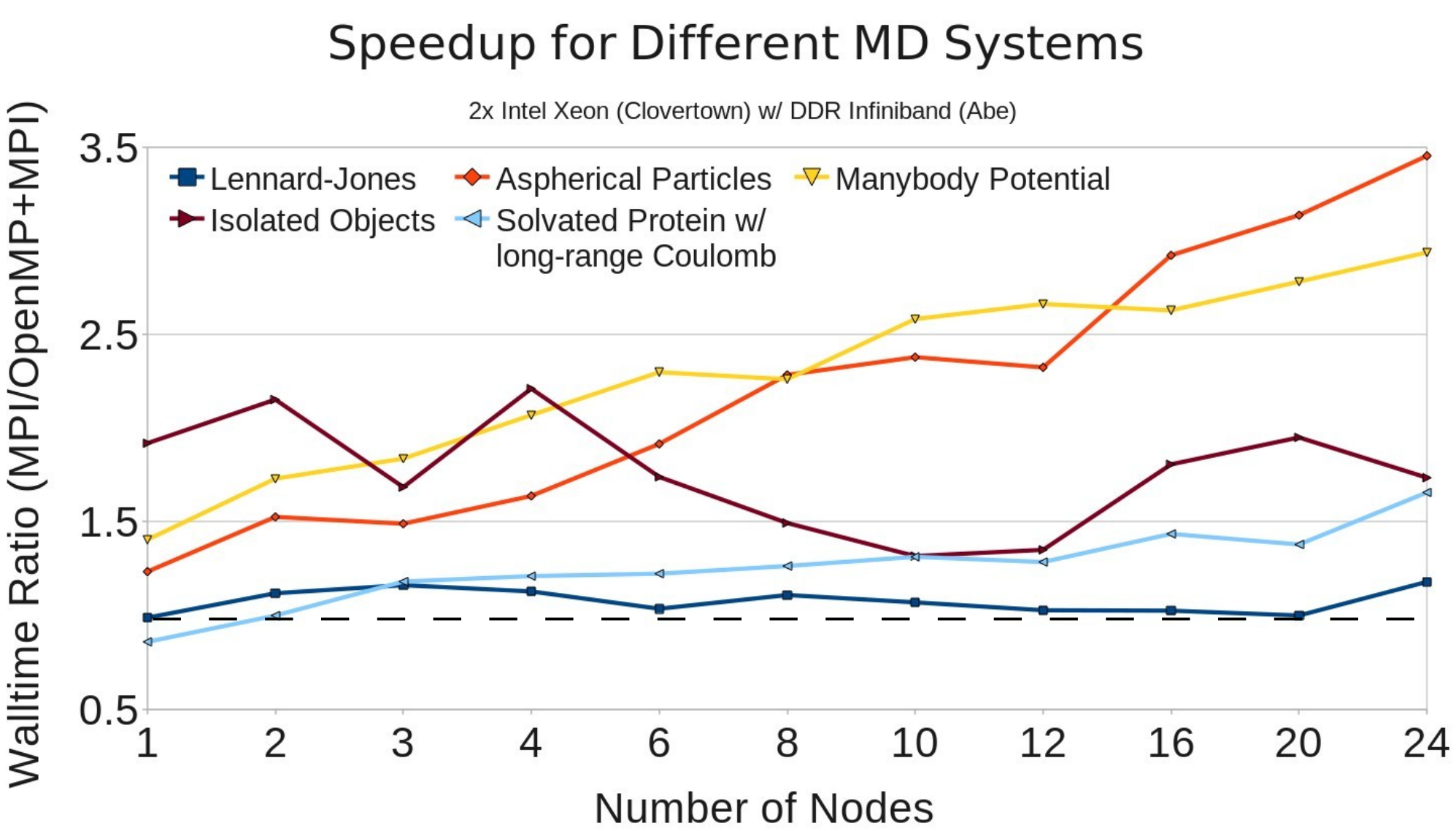
- **omp critical** timings
  1Thr: 4.2s, 2Thr: 7.1s, 4Thr: 7.7s, 8Thr: 8.6s
- **omp atomic** timings
  1Thr: 6.3s, 2Thr: 5.0s, 4Thr: 4.4s, 8Thr: 4.2s
- **omp parallel** region (MPI-like) timings
  1Thr: 4.0s, 2Thr: 2.5s, 4Thr: 2.2s, 8Thr: 2.5s
- No Newton's 3rd law timings
  1Thr: 6.5s, 2Thr: 3.7s, 4Thr: 2.3s, 8Thr: 2.1s

  => the **omp parallel** variant is best for few threads, no Newton's 3rd variant better for more threads
  => cost for force reduction larger for more threads

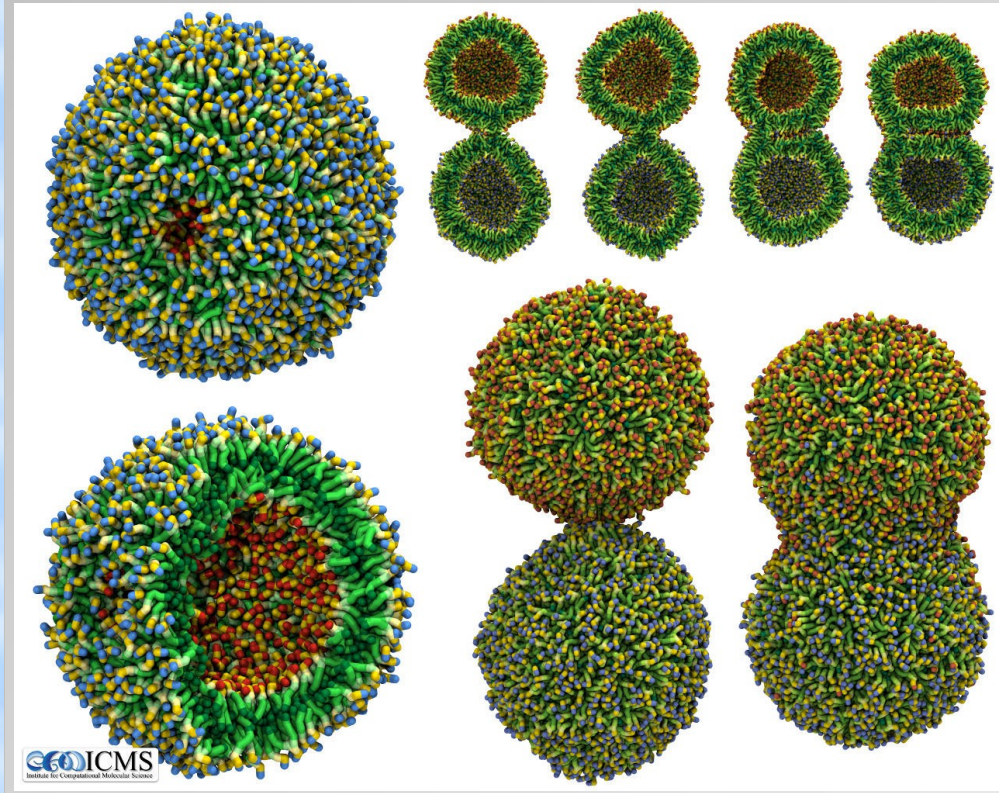
2x Intel Xeon 2.66Ghz (Harpertown) w/ DDR Infiniband





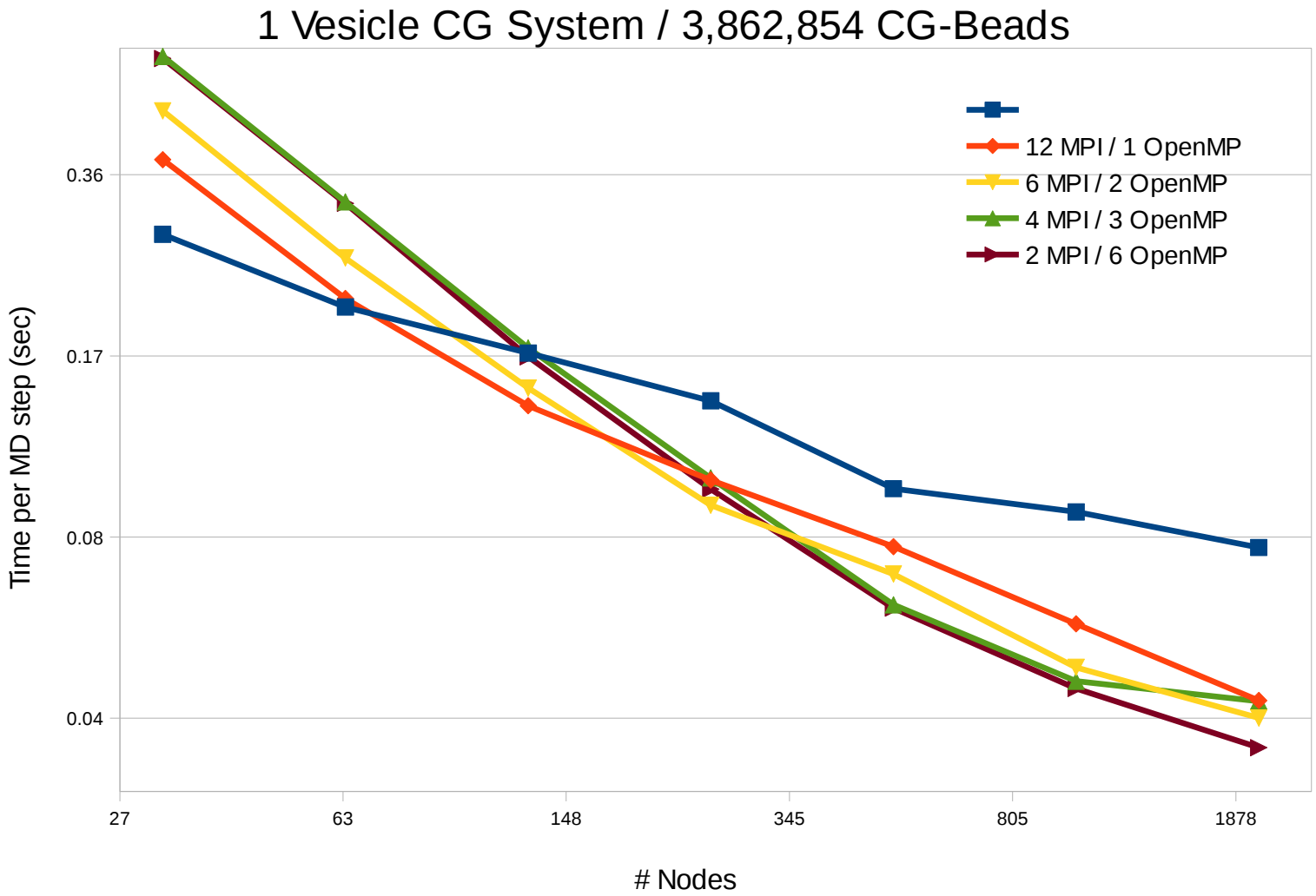

Speedup for Different MD Systems
2x Intel Xeon (Clovertown) w/ DDR Infiniband (Abe)
Walltime Ratio (MPI/OpenMP+MPI)
3.5
2.5
1.5
0.5
Lennard-Jones
Aspherical Particles
Manybody Potential
Isolated Objects
Solvated Protein w/
long-range Coulomb
1
2
3
4
6
8
10
12
16
20
24
Number of Nodes

Running Big

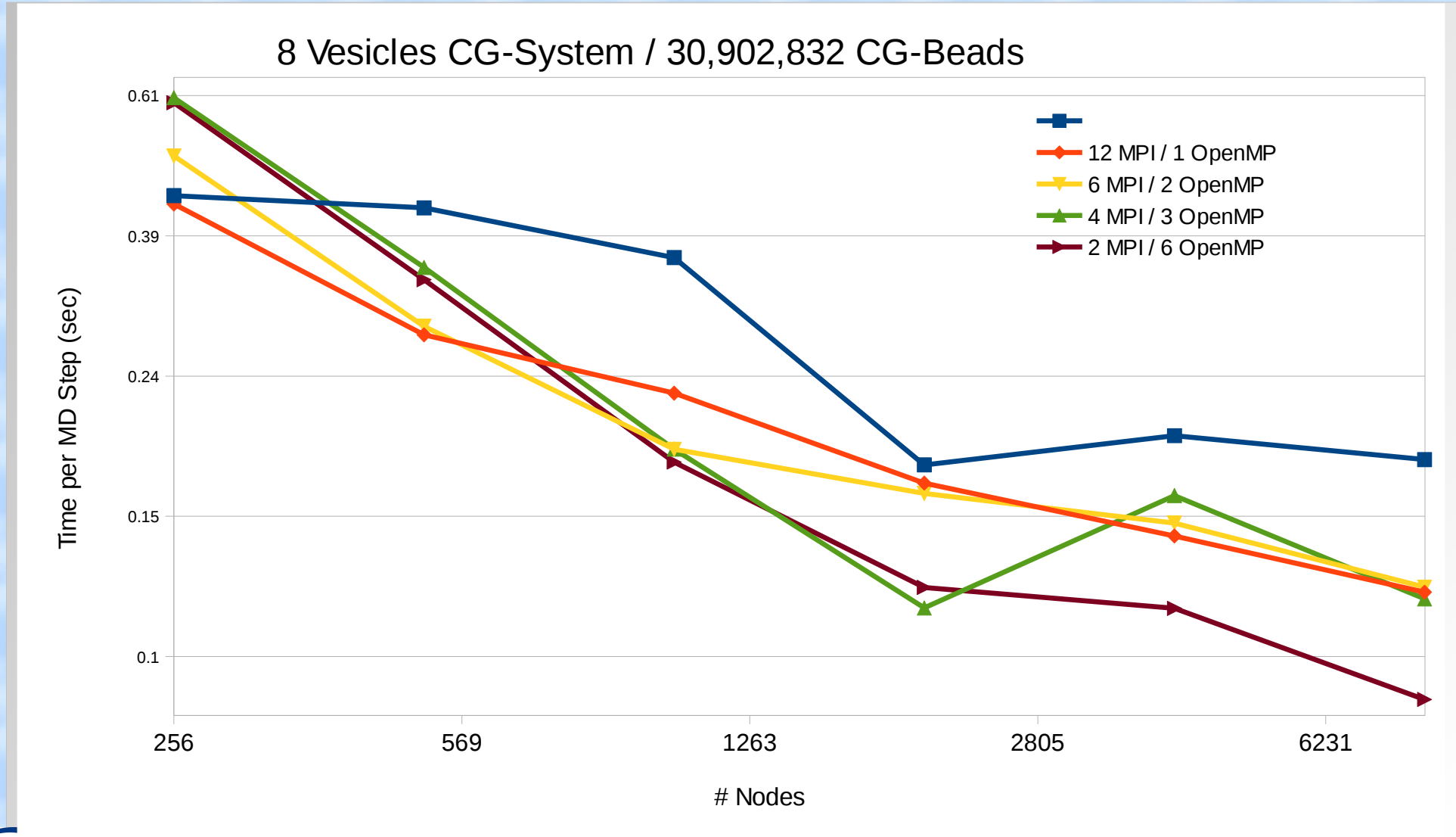


- Vesicle fusion study: impact of lipid ratio in binary mixture
- cg/cmm/coul/long
- Experimental size => 4M CG-beads for 1 vesicle and solvent
- 30,000,000 CPU hour INCITE project

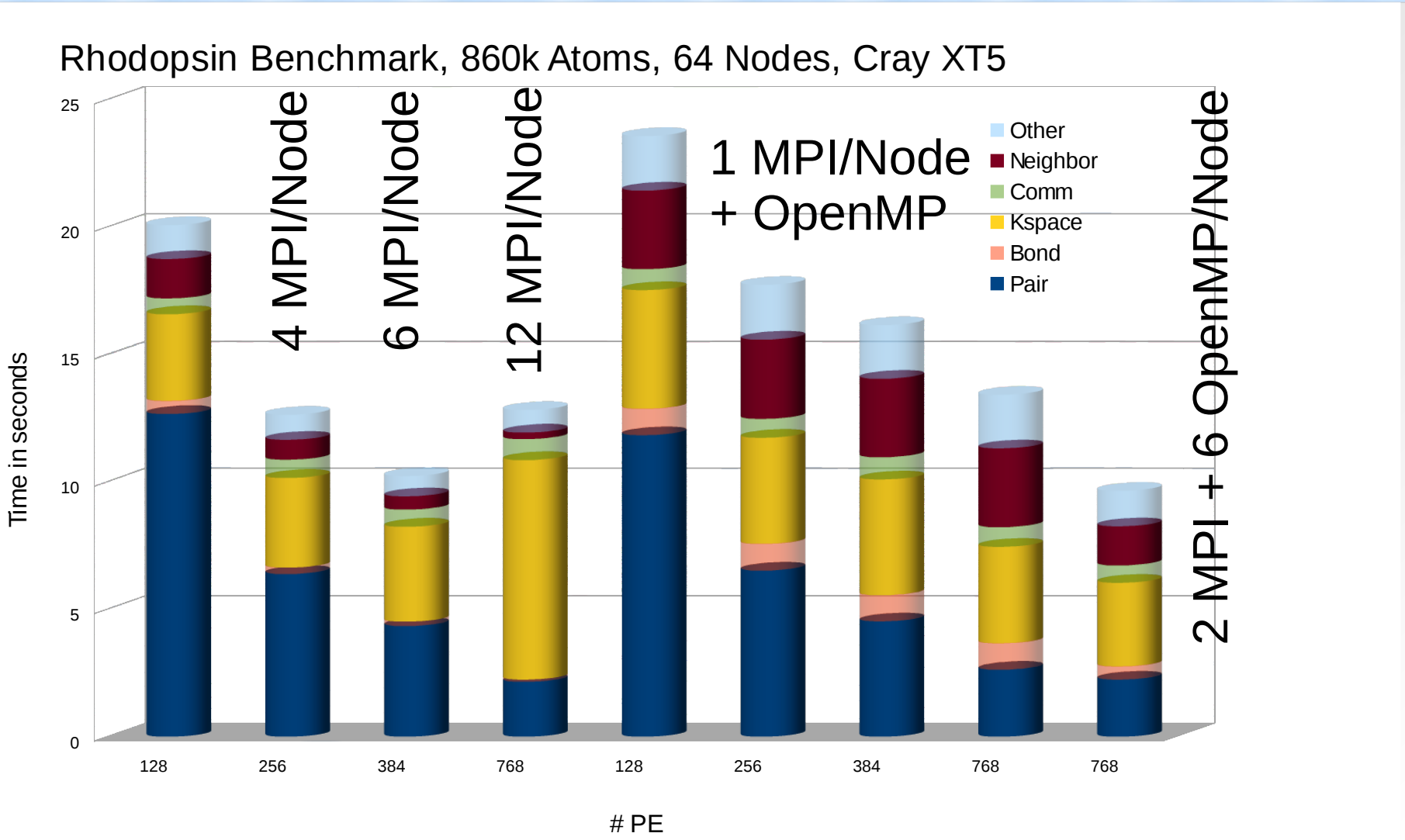
Strong Scaling (Cray XT5)



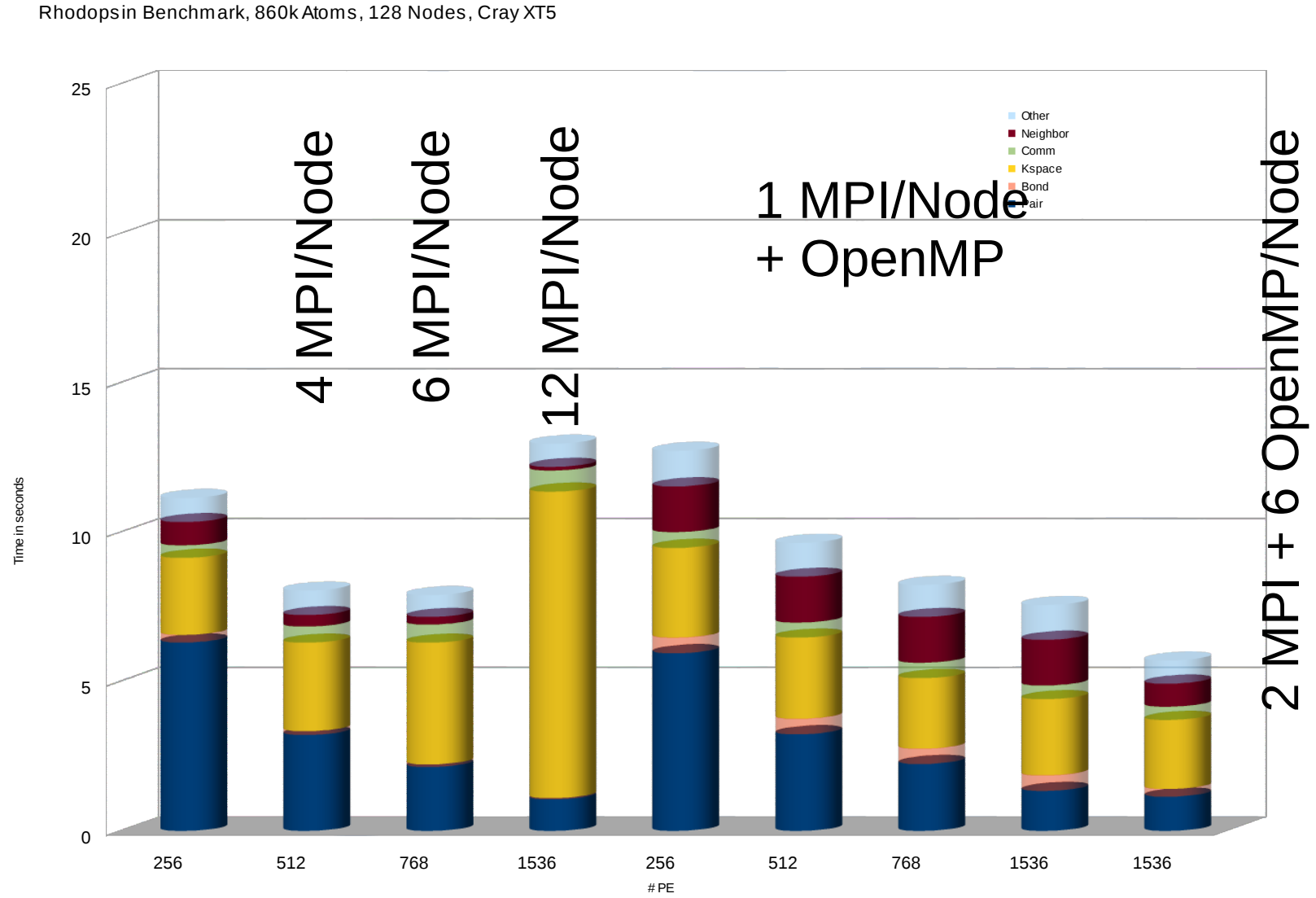
Strong Scaling (2) (Cray XT5)



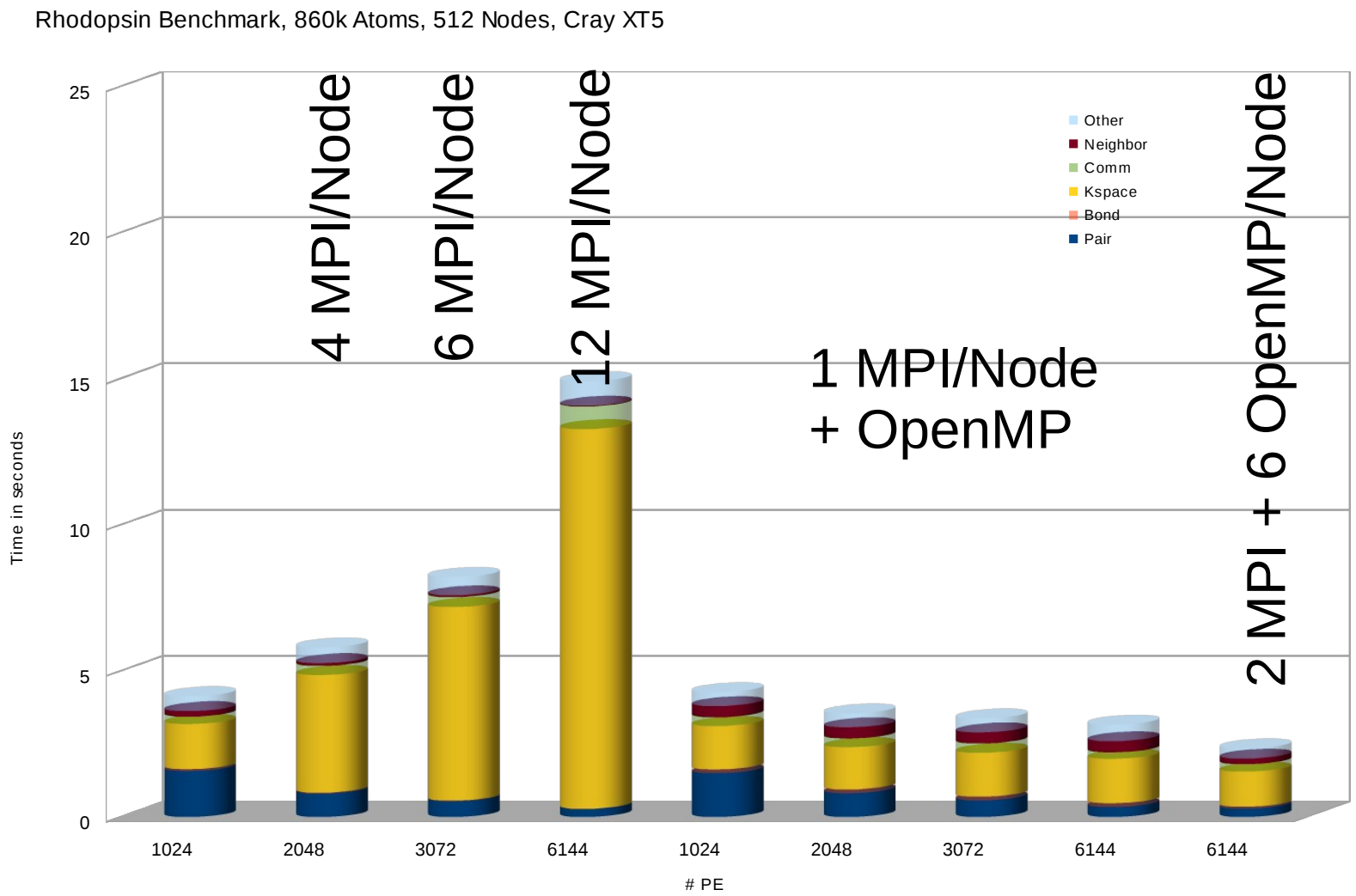
The Curse of the k-Space (1)



The Curse of the k-Space (2)



The Curse of the k-Space (3)



Additional Improvements

- OpenMP threading added to charge density accumulation and force application in PPPM
- Force reduction only done on last /omp style
- Integration style verlet/split contributed by Voth group which run k-space on separate partition (compatible with OpenMP version of PPPM)
- Added threading to selected fixes like charge equilibration for COMB many-body potential
- Added threading to fix nve/sphere integrator

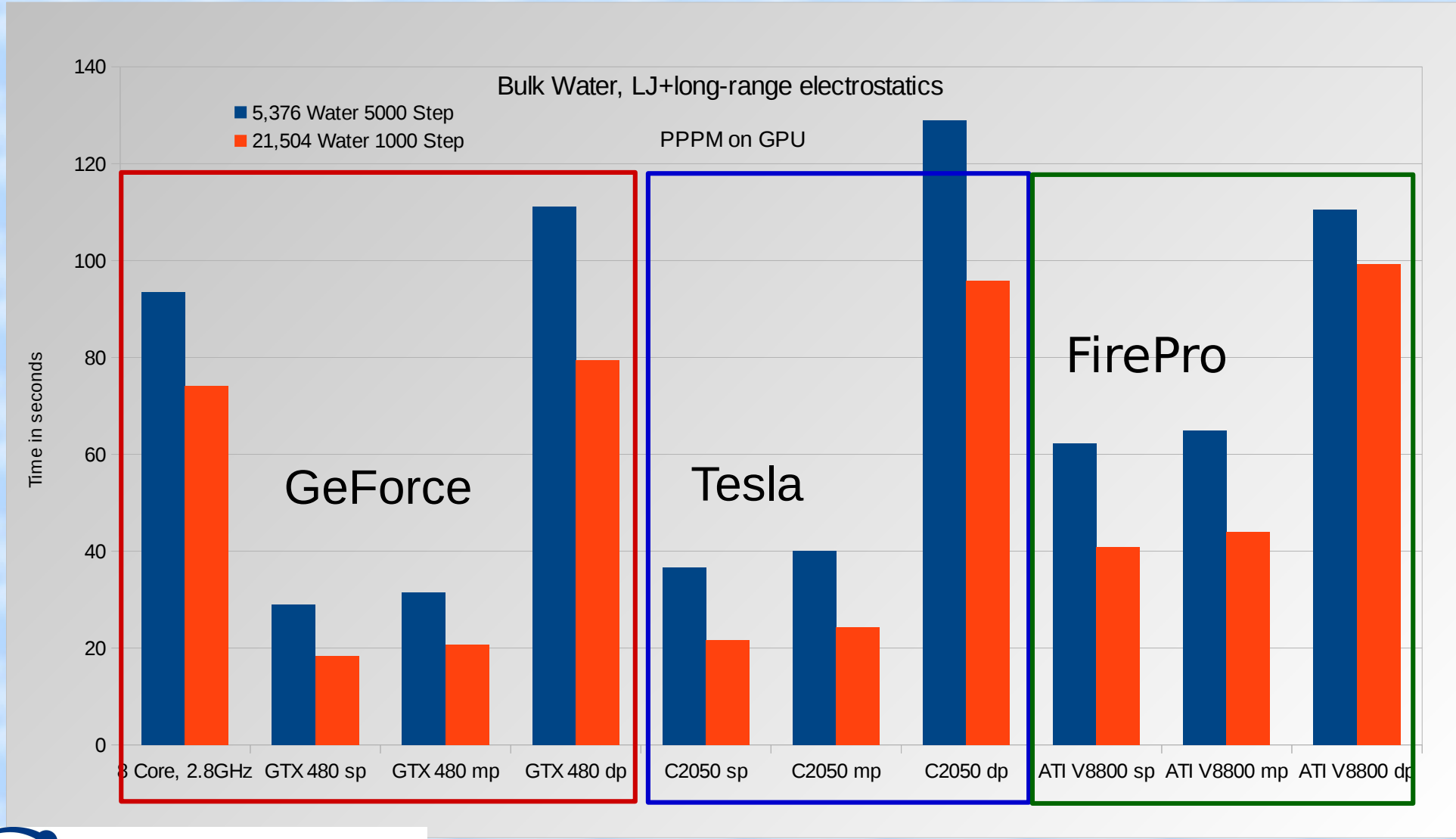
Current GPU Support in LAMMPS

- Multiple developments from different groups
- Converged to two efforts with two philosophies
- GPU package (minimalistic)
  - pair styles, neighbor lists and k-space (optional):
  - Download coordinates, retrieve forces
  - Run asynchronously to bonded (and k-space)
- USER-CUDA package (see next talk)
  - Replace all classes that touch atom data
  - Data transfer between host and GPU as needed

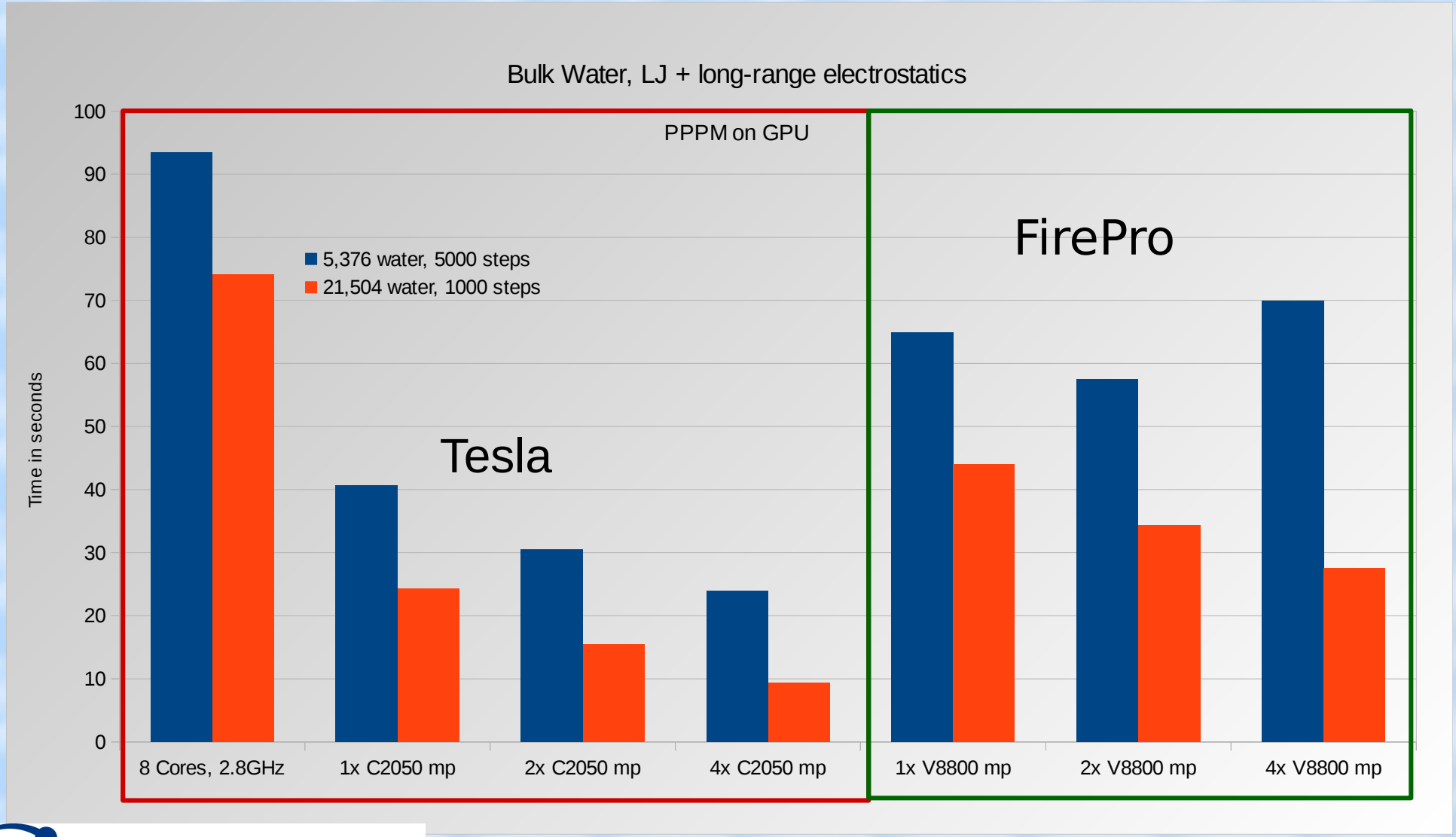
Special Features of “GPU” Package

- Can be compiled for CUDA or OpenCL due to using “Geryon” preprocessor macros
- Can attach multiple MPI tasks to one GPU for improved GPU utilization (up to 4x over-subscription on “Fermi”, up to 15x on “Kepler”)
- Uses a “fix” to manage GPUs and compute kernel dispatch, “styles” dispatch kernels asynchronously, “fix” then retrieves the forces after all other force computations are completed
- Tuned for good scaling with fewer atoms/GPU

1x GPU Performance in LAMMPS



Multiple GPUs per Node



Comments on GPU Acceleration

- Mixed precision (force computation in single, force accumulation in double precision) good compromise: little overhead, good accuracy on forces, stress/pressure less so
- GPU acceleration larger for models that require more computation in force kernel
- Acceleration drops with lower number of atoms per GPU => limited strong scaling on “big iron”
- Acceleration amount dependent on host & GPU

Installation of USER-OMP and GPU

- USER-OMP package:
  - `make yes-user-omp` to install sources
  - Add -fopenmp (GNU) or -openmp (Intel) to CC and LINK definitions in your makefile to enable OpenMP
  - Compilation without OpenMP => similar to OPT
- GPU package:
  - Compile library in `lib/gpu` for CUDA or OpenCL
  - `make yes-gpu` to install style sources which are wrappers for GPU library
  - Tweak `lib/gpu/Makefile.lammps.???` as needed

Using Accelerated Code

- All accelerated styles are optional and need to be activated in the input or from command line
- Naming convention lj/cut -> lj/cut/omp lj/cut/gpu
- From command line **`-sf omp`** or **`-sf gpu`**
- Inside script: **`suffix omp`** or **`suffix gpu`** and **`suffix on`** or **`suffix off`**
- Use package omp/gpu command to adjust settings for acceleration and selection of GPUs
- -sf command line flag implies default settings

Conclusions and Outlook: OpenMP

- OpenMP+MPI is almost always a win, especially with large node counts (=> capability computing)
- USER-OMP also contains serial optimizations and thus useful without OpenMP compiled in
- Minimal changes to LAMMPS core code
- USER-OMP only a transitional implementation since efficient only on a small number of threads
- Longer-term solution also needs to consider vectorization and thus be more GPU-like and benefits from different data layout (see next talk)