

Hybrid MPI+OpenMP Parallel MD

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations

Department of Computer Science

Department of Physics & Astronomy

Department of Quantitative & Computational Biology

University of Southern California

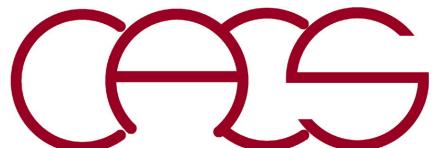
Email: anakano@usc.edu

Objective: Hands-on experience in default programming language (MPI+OpenMP) for hybrid parallel computing on a cluster of multicore computing nodes

Alternative to MPI-only: million ssh's & management of million processes by MPI daemon

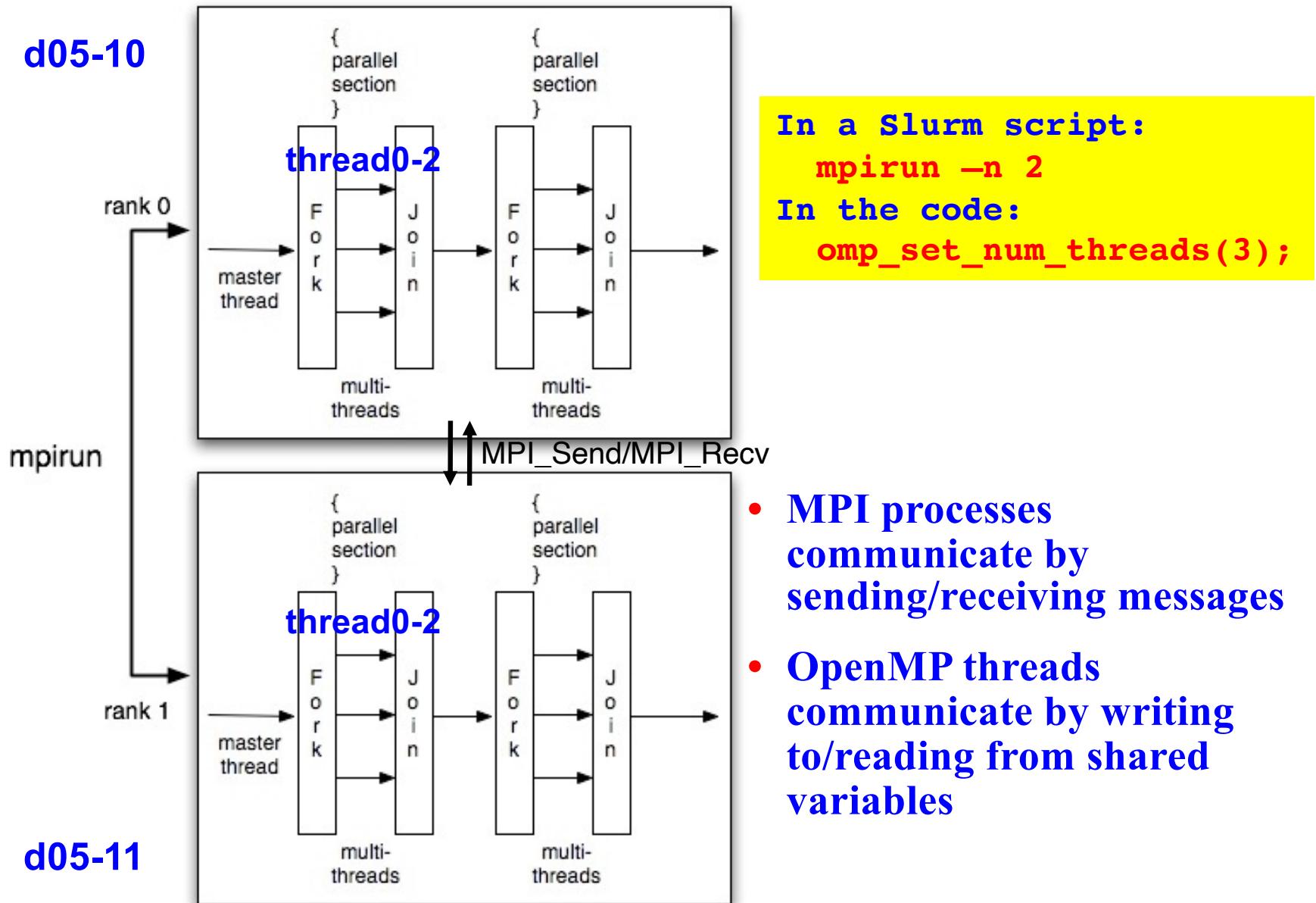
<https://aiichironakano.github.io/cs653/Kunaseh-HTM-PDSEC13.pdf>

MPI+X: <https://www.hpcwire.com/2014/07/16/compilers-mpix>



Hybrid MPI+OpenMP Programming

Each MPI process spawns multiple OpenMP threads



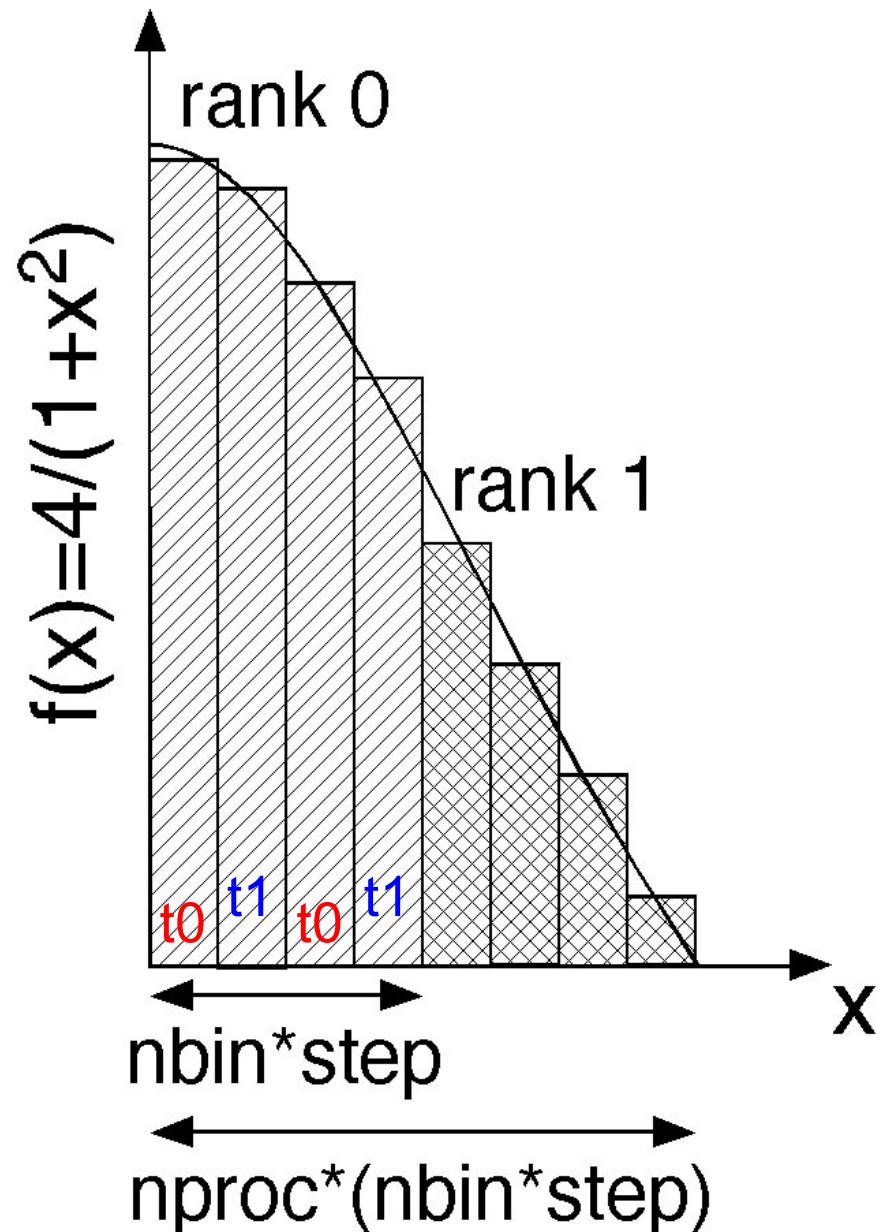
MPI+OpenMP Calculation of π

- **Spatial decomposition:** Each MPI process integrates over a range of width $1/nproc$, as a discrete sum of **nbin** bins each of width **step**
- **Interleaving:** Within each MPI process, **nthreads** OpenMP threads perform part of the sum as in **omp_pi.c**

https://aiichironakano.github.io/cs653/src/parallel/omp_pi.c

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$

$\overset{\text{step}}{\swarrow}$ $\overset{nproc * nbin}{\searrow}$



MPI+OpenMP Calculation of π : hpi.c

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main(int argc,char **argv) {
    int nbin,myid,nproc,nthreads,tid;
    double step,sum[MAX_THREADS]={0.0},pi=0.0,pig;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    nbin = NBIN/nproc; step = 1.0/(nbin*nproc);
    omp_set_num_threads(2);
    #pragma omp parallel private(tid)
    {
        int i;
        double x;
        nthreads = omp_get_num_threads();
        tid = omp_get_thread_num();
        for (i=nbin*myid+tid; i<nbin*(myid+1); i+=nthreads) { Who does what!
            x = (i+0.5)*step; sum[tid] += 4.0/(1.0+x*x); }
        printf("rank %d tid %d sum = %f\n",myid,tid,sum[tid]);
    }
    for (tid=0; tid<nthreads; tid++) pi += sum[tid]*step; Inter-thread reduction
    MPI_Allreduce(&pi,&pig,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
    if (myid==0) printf("PI = %f\n",pig); Inter-rank reduction
    MPI_Finalize();}
```

<https://aiichironakano.github.io/cs653/src/parallel/hpi.c>

Shared variables among all threads

$NBIN \rightarrow \lfloor NBIN/nproc \rfloor \times nproc$
of bins per rank
 $= \overbrace{nbin}^{\text{# of bins per rank}} \times nproc$

Local variables: Different values needed for different threads

Data privatization to avoid race condition

Who does what!

Inter-thread reduction

Inter-rank reduction

MPI+OpenMP Example: hpi.c

- Compilation on `discovery.usc.edu`

```
mpicc -o hpi hpi.c -fopenmp
```

- Slurm script

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=2
#SBATCH --time=00:00:59
#SBATCH --output=hpi.out
#SBATCH -A anakano_429
mpirun -n $SLURM_NNODES ./hpi
```

- Output

```
rank tid sum = 1 1 6.434981e+04
rank tid sum = 1 0 6.435041e+04
rank tid sum = 0 0 9.272972e+04
rank tid sum = 0 1 9.272932e+04
PI = 3.141593
```

- Find information about Slurm nodes & partitions

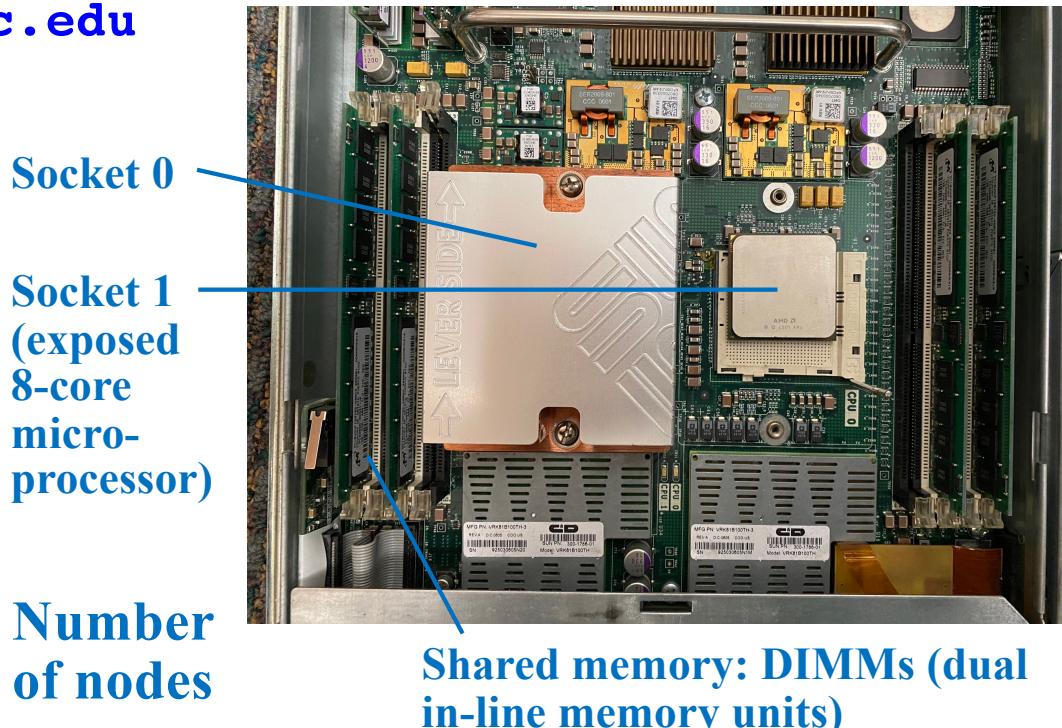
```
[anakano@discovery ~]$ sinfo View information
```

PARTITION	AVAIL	TIMELIMIT	NODES	STATE	NODELIST
main*	up	2-00:00:00	281	mix	d05-[08-15,26-29,31-37,39,42],...
epyc-64	up	2-00:00:00	26	alloc	b22-[01-09,11-25,28-29]

...

```
[anakano@discovery ~]$ sinfo2 View detailed information
```

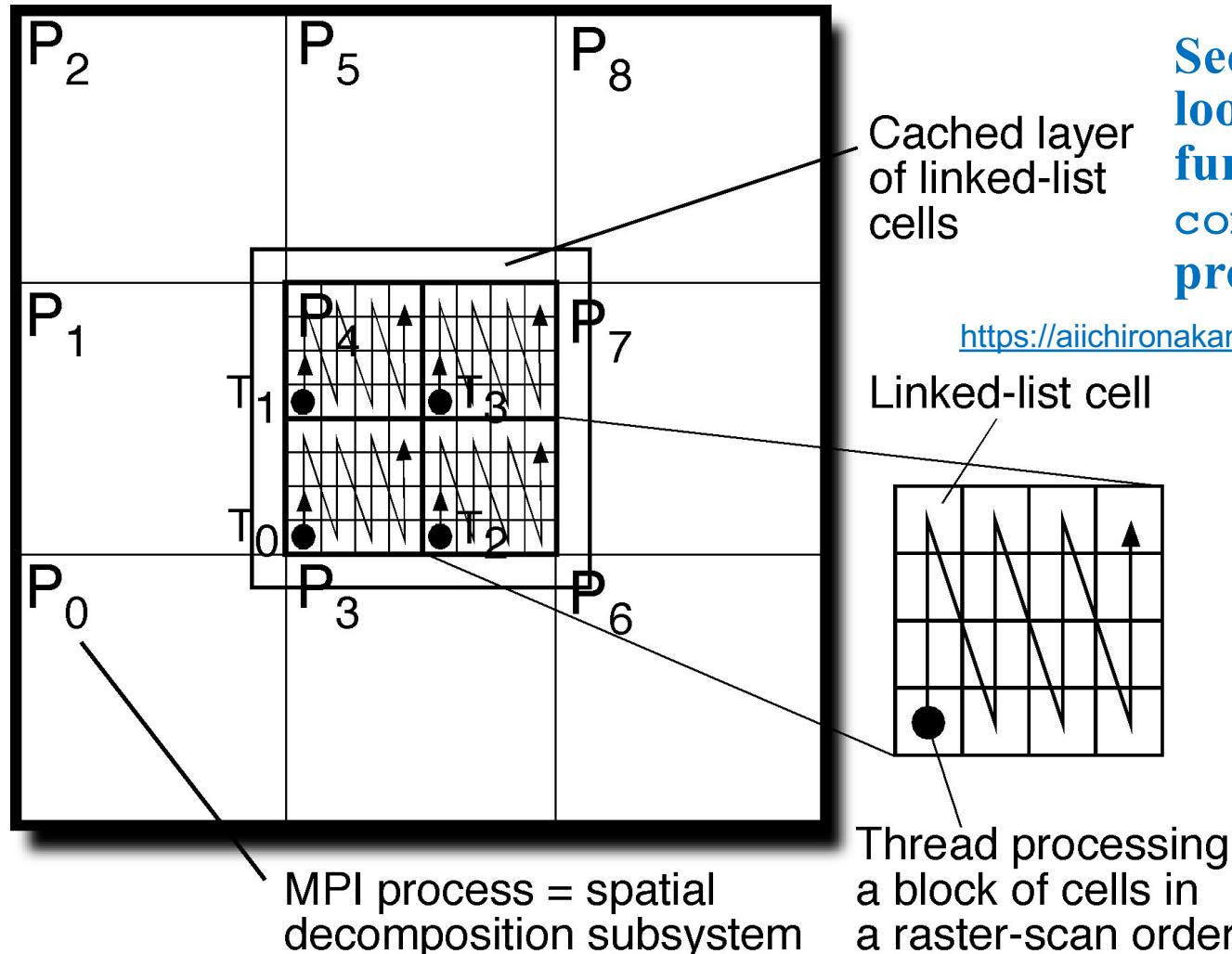
NODELIST	PARTITION	STATE	NODES	SOCKETS	CORES	MEMORY	GRES	ACTIVE_FEATURES
e13-35	main*	down*	1	2	8	63400	(null)	xeon-2640v3



Hybrid MPI+OpenMP Parallel MD

- OpenMP threads handle blocks of linked-list cells in each MPI process (= spatial-decomposition subsystem)

Big picture = who does what: loop index \rightarrow thread map



Linked-List Cell Block

Variables

- **vthrd[0|1|2]** = # of OpenMP threads per MPI process in the x|y|z direction.
- **nthrd** = # of OpenMP threads = **vthrd[0]×vthrd[1]×vthrd[2]**.
- **thbk[3]**: **thbk[0|1|2]** is the # of linked-list cells in the x|y|z direction that each thread is assigned.

In main():
omp_set_num_threads(nthrd);

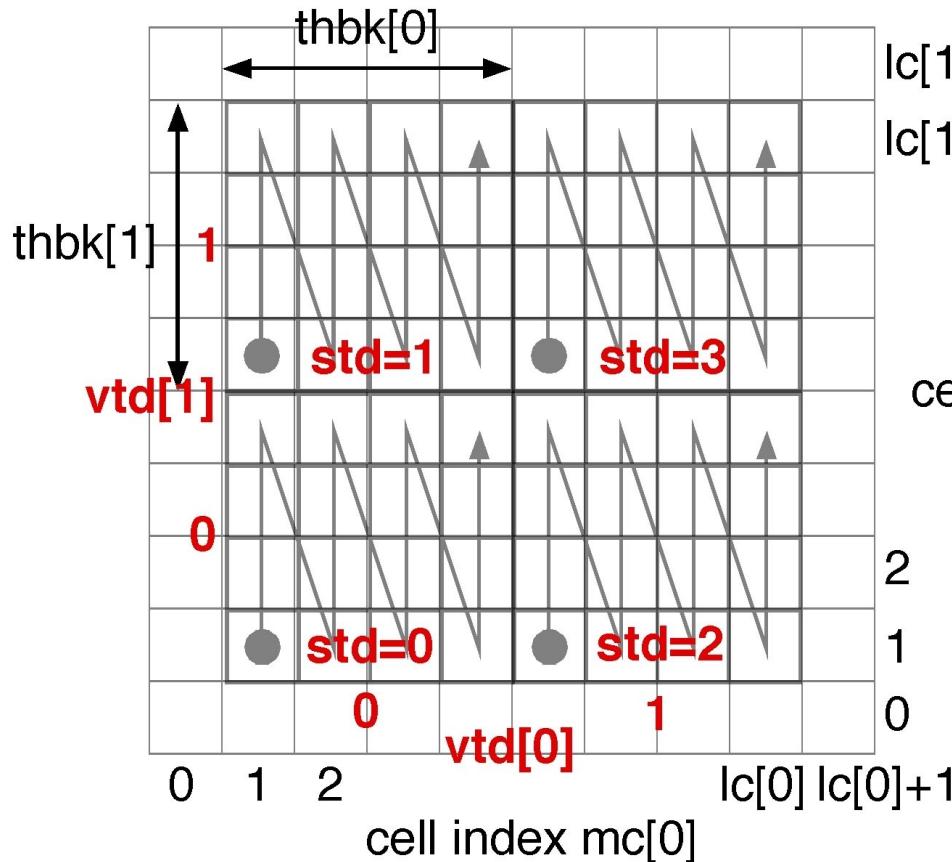
In hmd.h:
int vthrd[3]={2,2,1},nthrd=4;
int thbk[3];

In init_params():
/* Compute the # of cells for linked-list cells */
for (a=0; a<3; a++) {
 lc[a] = al[a]/RCUT; /* Cell size ≥ potential cutoff */
 /* Size of cell block that each thread is assigned */
 thbk[a] = lc[a]/vthrd[a];
 /* # of cells = integer multiple of the # of threads */
 lc[a] = thbk[a]*vthrd[a]; /* Adjust # of cells/MPI process */
 rc[a] = al[a]/lc[a]; /* Linked-list cell length */
}

OpenMP Threads for Cell Blocks

Variables

- **std** = scalar thread index.
- **vtd[3]**: **vtd[0 | 1 | 2]** is the xlylz element of vector thread index.
- **mofst[3]**: **mofst[0 | 1 | 2]** is the xlylz offset cell index of cell-block.



```
int std,vtd[ 3 ],mofst[ 3 ];  
std = omp_get_thread_num();  
vtd[ 0 ] = std/(vthrd[ 1 ]*vthrd[ 2 ]);  
vtd[ 1 ] = (std/vthrd[ 2 ])%vthrd[ 1 ];  
vtd[ 2 ] = std%vthrd[ 2 ];  
for (a=0; a<3; a++)  
    mofst[ a ] = vtd[ a ]*thbk[ a ];
```

Call `omp_get_thread_num()` within an OpenMP parallel block.

Threads Processing of Cell Blocks

- Start with the MPI parallel MD program, `pmd.c`
- Within each MPI process, parallelize the outer loops over central linked-list cells, `mc[]`, in the force computation function, `compute_accel()`, using OpenMP threads
- If each thread needs separate copy of a variable (e.g., loop index `mc[]`), declare it as `private` in the OpenMP parallel block

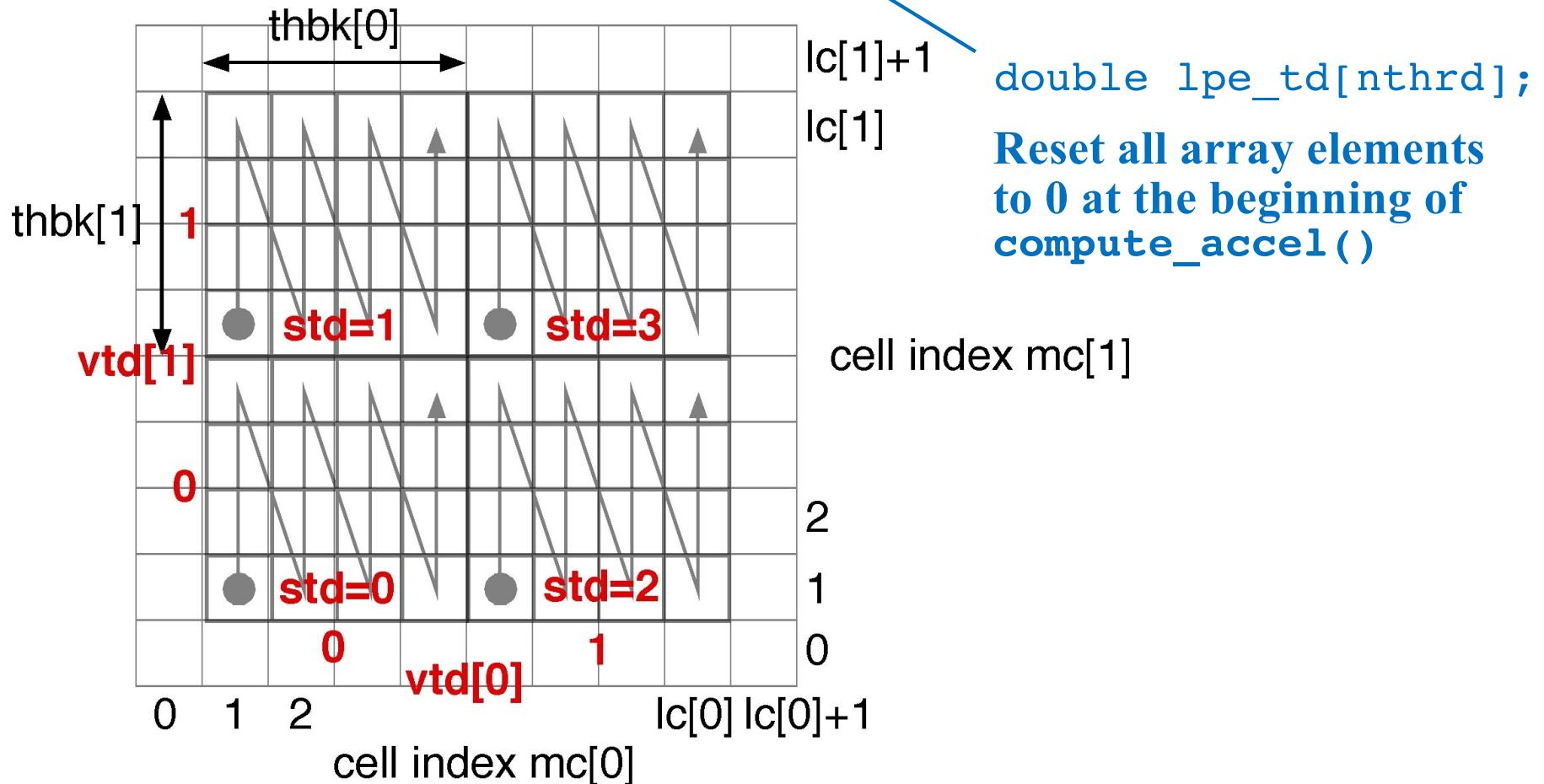
```
#pragma omp parallel private(mc,...)
{
    ...
    for (mc[0]=mofst[0]+1; mc[0]<=mofst[0]+thbk[0]; (mc[0])++) {
        for (mc[1]=mofst[1]+1; mc[1]<=mofst[1]+thbk[1]; (mc[1])++) {
            for (mc[2]=mofst[2]+1; mc[2]<=mofst[2]+thbk[2]; (mc[2])++) {
                Each thread handles thbk[0]xthbk[1]xthbk[2] cells independently
            }
        ...
    }
}
```

Avoiding Critical Sections (1)

- Remove the critical section

```
if (bintra) lpe += vVal; else lpe += 0.5*vVal;
```

by defining an array, `lpe_td[nthrd]`, where each array element stores the partial sum of the potential energy by a thread



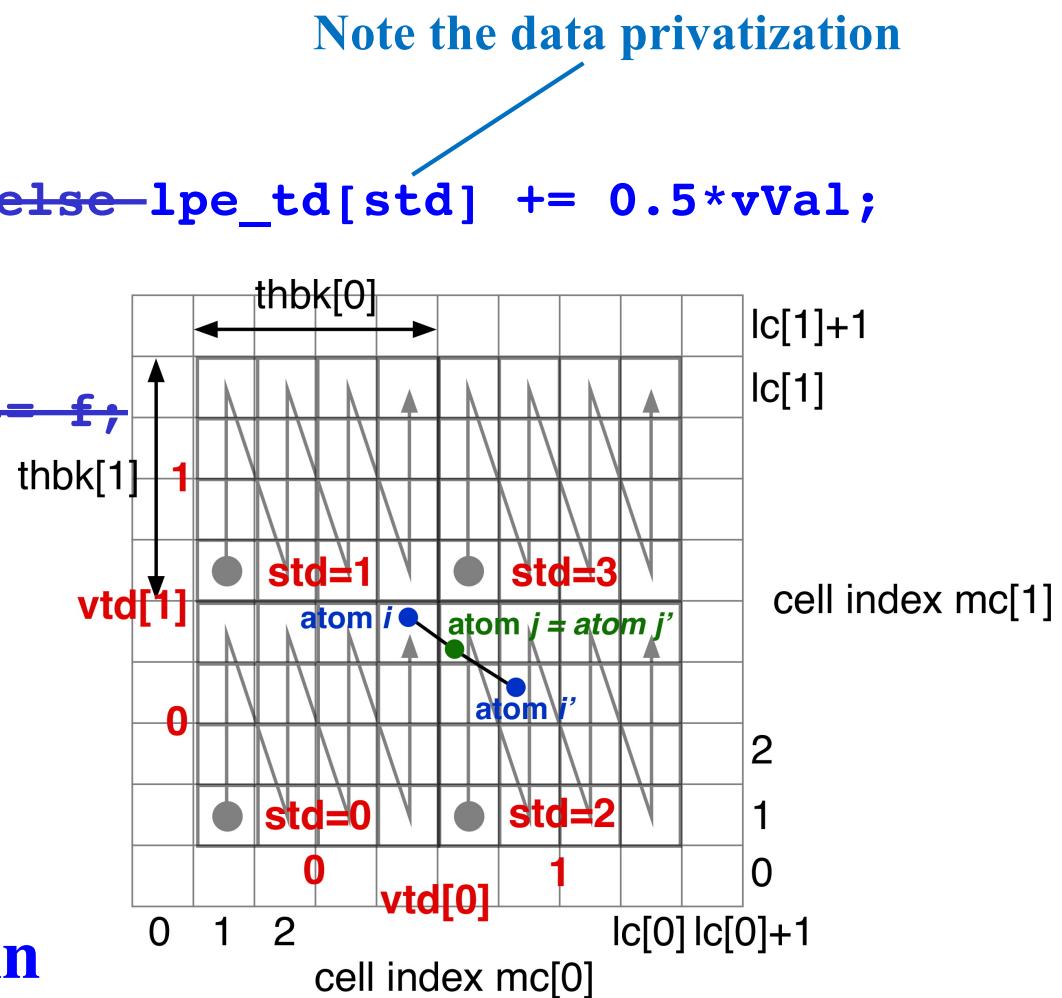
Data privatization: cf. `omp_pi.c` & `hpi.c`

Avoiding Critical Sections (2)

- To avoid multiple threads to access an identical force array element, stop using the Newton's third law:

```
int bintra;
...
bintra = (j < n);
...
if (i < j && rr < rrCut) {
    ...
    if (bintra) lpe += vVal; else lpe_td[std] += 0.5*vVal;
    for (a=0; a<3; a++) {
        f = fcVal*dr[a];
        ra[i][a] += f;
        if (bintra) ra[j][a] -= f;
    }
}
```

Mutually exclusive access
to `ra[][]` for preventing
race conditions



- Interthread reduction after join

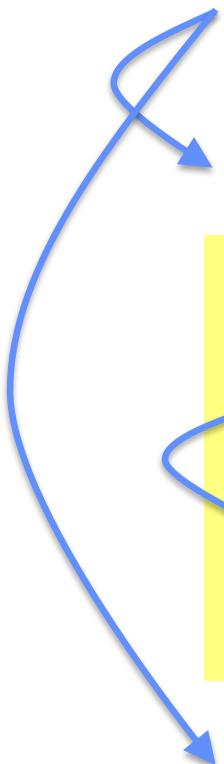
```
for (i=0; i<nthrd; i++) lpe += lpe_td[i];
```

OpenMP Essential

define shared;
... if used here

```
#pragma omp parallel private(if used in both)
{
    define private;
    ... if only used (in left-hand side) here
}
```

... or here



Running HMD at CARC

- Submit a batch job using the following Slurm script.

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=4
#SBATCH --time=00:01:59
#SBATCH --output=hmd.out
#SBATCH -A anakano_429

mpirun -bind-to none -n 2 ./hmd
```

To be explained later

- Note that hmd.c must have been compiled in the same directory as you submit this Slurm script:

```
mpicc -O -o hmd hmd.c -lm -fopenmp
```

Interactively Running HMD at CARC (1)

1. Interactively submit a Slurm job & wait until you are allocated nodes.
(Note that you will be automatically logged in to one of the allocated nodes.)

```
$ salloc --nodes=2 --ntasks-per-node=1 --cpus-per-task=4 -t 29
salloc: Pending job allocation 6064886
salloc: job 6064886 queued and waiting for resources
salloc: job 6064886 has been allocated resources
salloc: Granted job allocation 6064886
[anakano@d05-35 ~]$
```

 You are logged in to one of the allocated nodes

For CPU information, type more /proc/cpuinfo

Interactively Running HMD at CARC (2)

2. Submit a two-process MPI program (named hmd); each of the MPI process will spawn 4 OpenMP threads.

```
[anakano@d05-35 cs596]$ mpirun -bind-to none -n 2 ./hmd
```

3. While the job is running, you can open another window & log in to the node (or the other allocated node) to check that all processors are busy using top command. Type 'H' to show individual threads (type 'q' to stop).

```
[anakano@discovery ~]$ ssh d05-35
[anakano@d05-35 ~]$ top (then type H)

...

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
29861	anakano	20	0	443776	102836	7976	R	99.9	0.1	0:09.12	hmd
29871	anakano	20	0	443776	102836	7976	R	99.9	0.1	0:09.06	hmd
29869	anakano	20	0	443776	102836	7976	R	99.7	0.1	0:09.02	hmd
29870	anakano	20	0	443776	102836	7976	R	99.7	0.1	0:09.04	hmd
29661	anakano	20	0	164504	2624	1628	R	0.3	0.0	0:02.34	top
1	root	20	0	43572	3944	2528	S	0.0	0.0	2:06.33	systemd

```
...
```

Interactively Running HMD at CARC (3)

4. Type ‘1’ to show core-usage summary.

```
top - 12:36:48 up 48 days, 23:35, 1 user, load average: 3.62, 3.75, 2.86
Threads: 378 total, 5 running, 373 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.3 us, 0.0 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 : 99.7 us, 0.3 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu4 : 100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu5 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu6 : 0.0 us, 0.3 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu7 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
...
%Cpu19 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
```

Interactively Running HMD at CARC (4)

5. Without -bind-to none option, hmd process (and all spawned threads by it) is bound to one core.

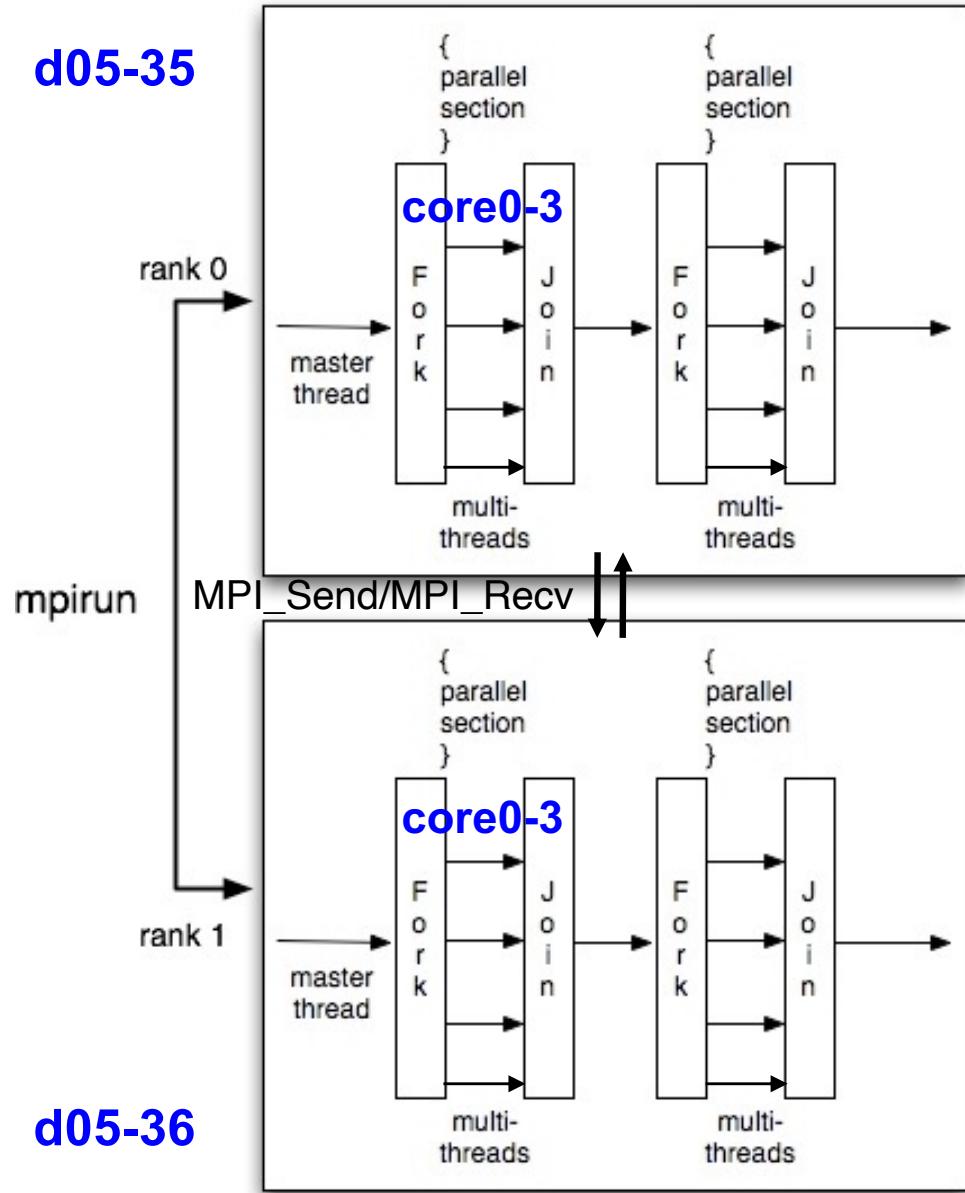
```
[anakano@d05-35 cs596]$ mpirun -n 2 ./hmd
```

```
[anakano@d05-36 ~]$ top
```

```
...
      PID USER      PR  NI      VIRT      RES      SHR S %CPU %MEM TIME+ COMMAND
29363 anakano    20   0  443556 108340    7580 R 27.9  0.1  0:18.43 hmd
29373 anakano    20   0  443556 108340    7580 S 24.3  0.1  0:15.96 hmd
29371 anakano    20   0  443556 108340    7580 S 23.9  0.1  0:16.06 hmd
29372 anakano    20   0  443556 108340    7580 S 23.9  0.1  0:15.96 hmd
29341 anakano    20   0 164504   2476   1608 R  0.7  0.0  0:00.37 top
1 root        20   0  43572   3944   2528 S  0.0  0.0  2:06.30 systemd
...
...
```

How Hybrid MPI+OpenMP MD Runs

d05-35



In hmd.h:

```
int vproc[3] = {1,1,2}, nproc = 2;  
int vthrd[3] = {2,2,1}, nthrd = 4;
```

In hmd.c:

```
omp_set_num_threads(nthrd);
```

mpirun

On discovery:

```
salloc --nodes=2 --ntasks-per-node=1  
--cpus-per-task=4 -t 30
```

On d05-35:

```
mpirun -bind-to none -n 2 ./hmd
```

On d05-35 & d05-36:

```
top (then type H and 1)
```

d05-36

Try it yourself!

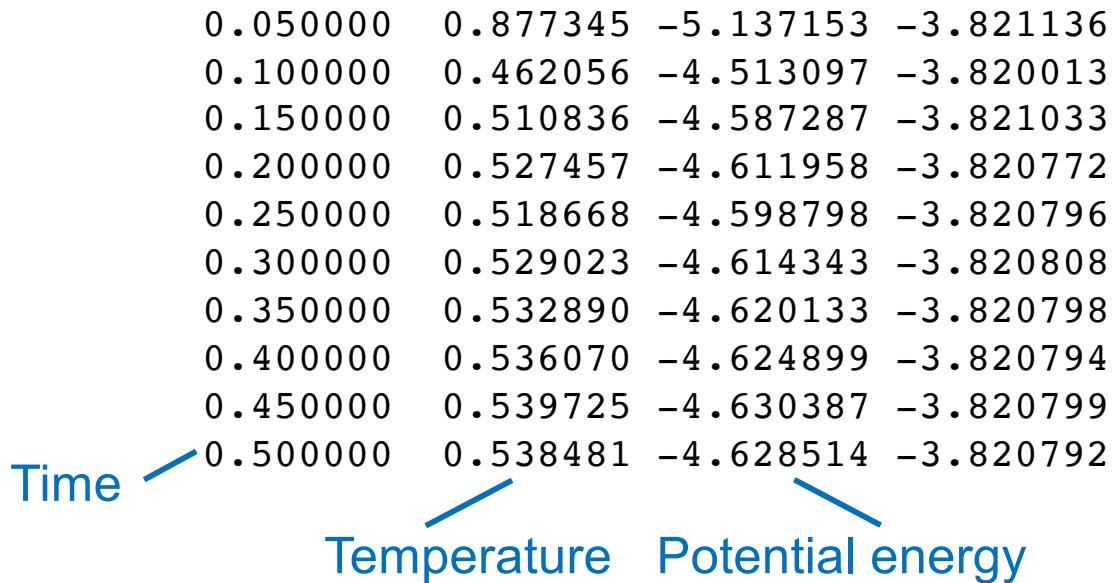
Validation of Hybrid MD

2 MPI process; 4 threads

In hmd.h:

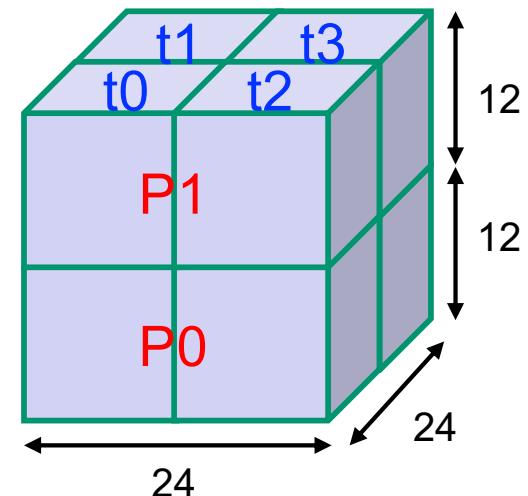
```
vproc = {1,1,2}, nproc = 2;  
vthrd = {2,2,1}, nthrd = 4;
```

Make sure that the total energy is the same as that calculated by pmd.c using the same input parameters, at least for ~5-6 digits



pmd.in

```
24 24 12 InitUcell[3]  
0.8 Density  
1.0 InitTemp  
0.005 DeltaT  
100 StepLimit  
10 StepAvg
```



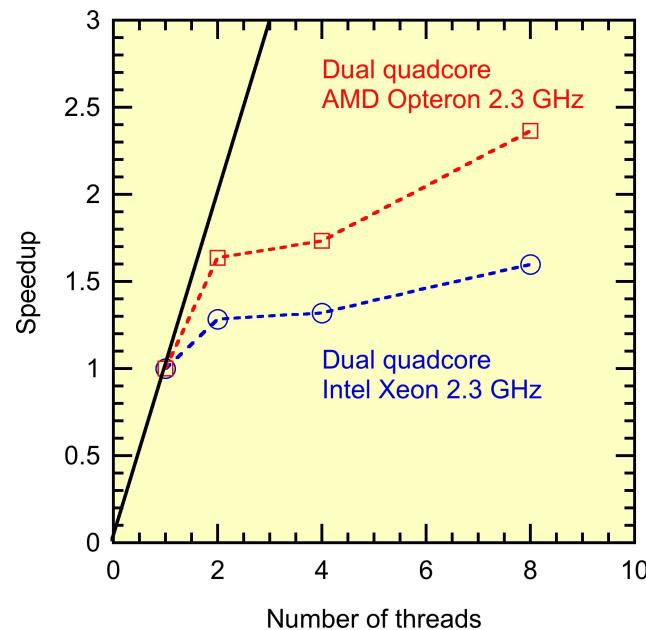
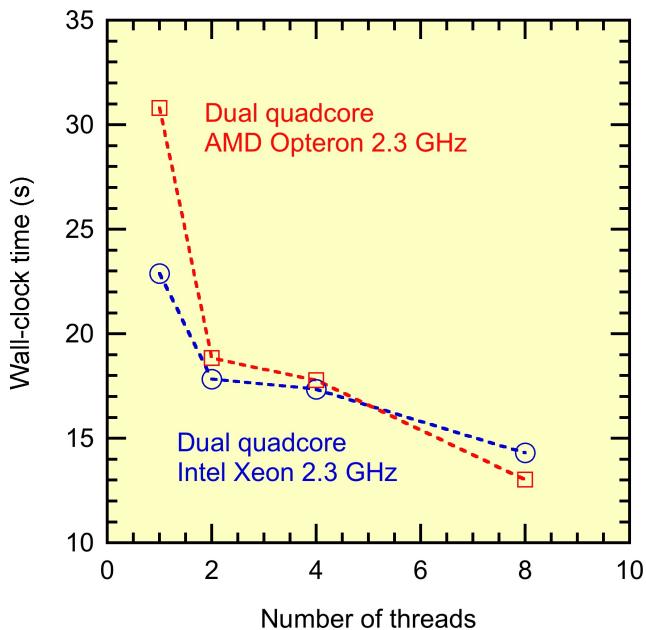
See the lecture on “order-invariant real-number summation”

Strong Scalability of Hybrid MD

1 MPI process; 1-8 threads

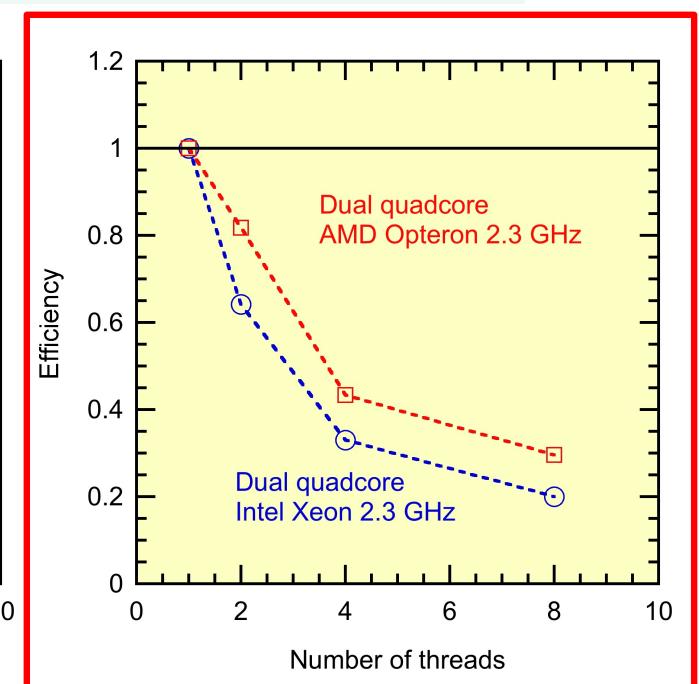
In `hmd.h`:

```
vproc = {1,1,1}, nproc = 1;
vthrd = {1,1,1}, nthrd = 1;
      2 1 1           2
      2 2 1           4
      2 2 2           8
```



`pmd.in`

```
24 24 24 InitUcell[3]
0.8          Density
1.0          InitTemp
0.005        DeltaT
100         StepLimit
101         StepAvg
```



`InitUcell[] = {24, 24, 24}`

$$N = 4 \times 24^3$$

$$= 55296 \text{ atoms}$$

$$S_P = \frac{T(N,1)}{T(N,P)}$$

$$E_P = \frac{S_P}{P}$$

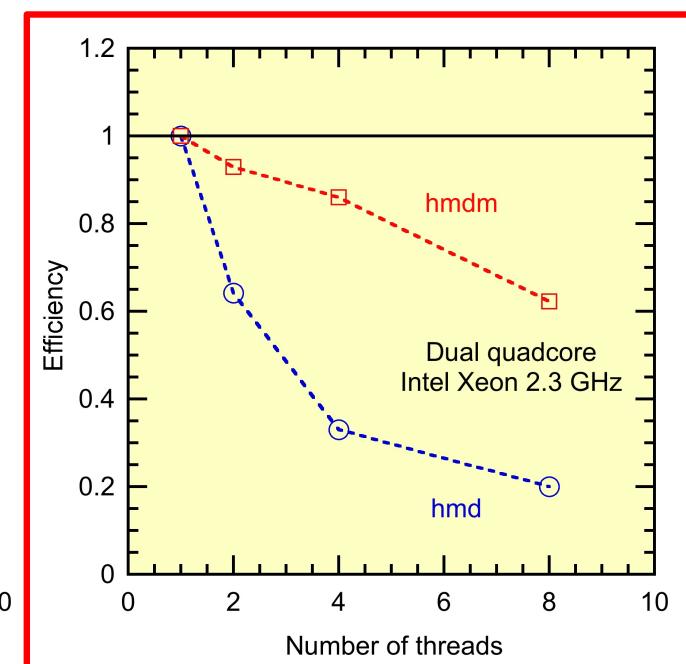
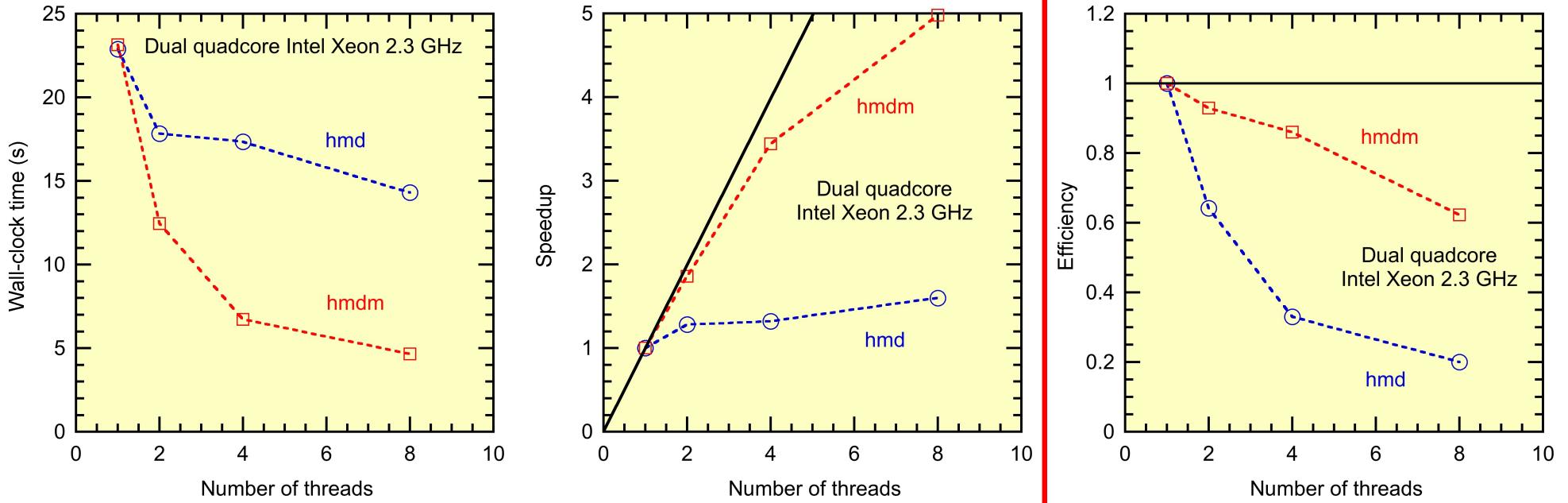
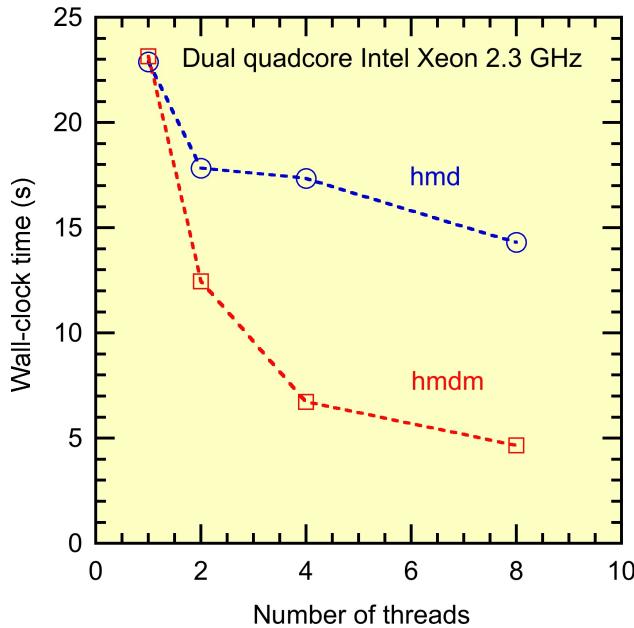
P: Number of cores

Improved Strong Scalability of Hybrid MD

1 MPI process; 1-8 threads

In hmd.h:

```
vproc = {1,1,1}, nproc = 1;
vthrd = {1,1,1}, nthrd = 1;
      2 1 1           2
      2 2 1           4
      2 2 2           8
```



InitUcell[] = {24, 24, 24}

$$N = 4 \times 24^3$$

= 55296 atoms

$$S_P = \frac{T(N,1)}{T(N,P)}$$

P: Number of cores

$$E_P = \frac{S_P}{P}$$

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8

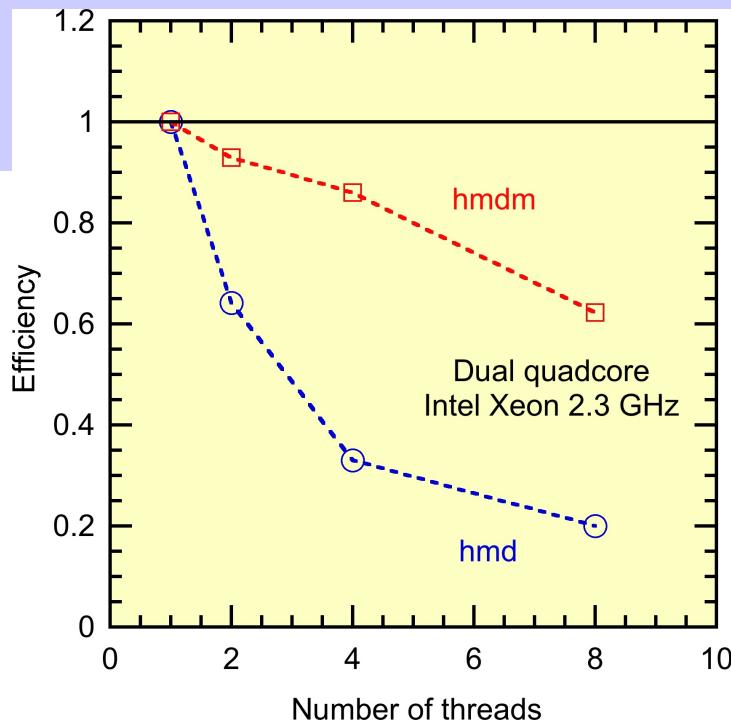
mpirun ... -n 1 ... ./hmd1
2
4
8
```

More on Multithreading MD

- Large overhead is involved in opening an OpenMP parallel section
→ Open it only once in the main function

In hmdm.c:

```
int main() {
    ...
    omp_set_num_threads(nthrd);
    #pragma omp parallel
    {
        #pragma omp master
        { // Do serial computations here}
        ...
        #pragma omp barrier // When threads need be synchronized
        ...
    }
    ...
}
```



More on Avoiding Race Conditions

- Program `hmd.c`: (1) used data privatization; (2) disabled the use of Newton's third law → this doubled computation
- Cell-coloring
 - > Race condition-free multithreading without duplicating pair computations
 - > Color cells such that no cells of the same color are adjacent to each other
 - > Threads process cells of the same color at a time in a color loop

1	3	1	3	1	3
0	2	0	2	0	2
1	3	1	3	1	3
0	2	0	2	0	2
1	3	1	3	1	3
0	2	0	2	0	2

Four-color (eight colors in 3D) solution requires the cell size to be twice the cutoff radius r_c

H. S. Byun et al.,
Comput. Phys. Commun.
219, 246 ('17)

- Use graph coloring in more general computations

False Sharing

- While eliminating race conditions by data privatization, the use of consecutive per-thread accumulators, `lpe_td[nthrd]`, degrades performance by causing excessive cache misses

See [false sharing](#) Wiki page

- **Solution 1: Padding**

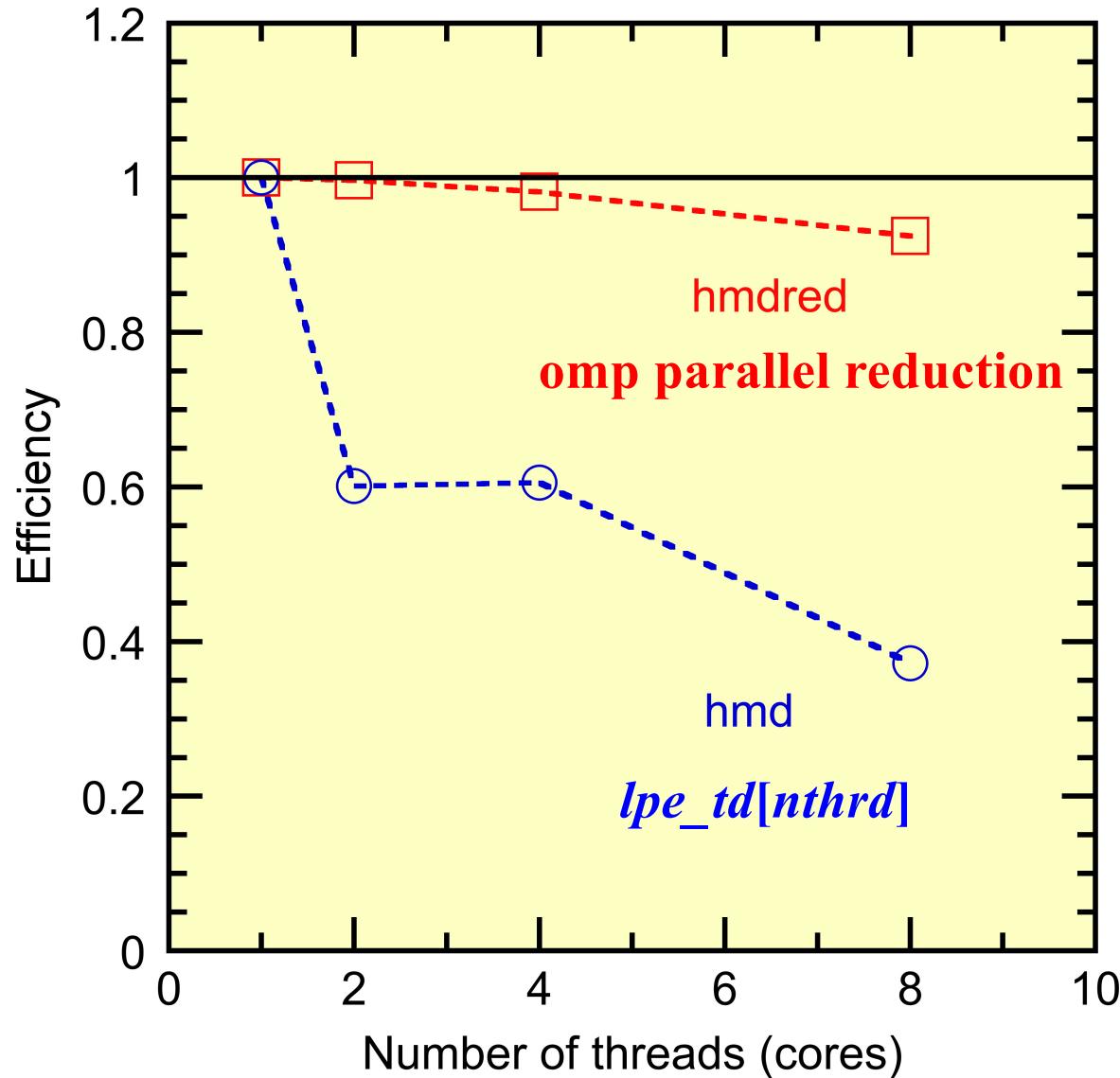
```
struct lpe_t {  
    double lpe;  
    double pads[7]; // assume intel CPU with 64 byte cache line  
};  
struct lpe_t lpe_td[nthrd];
```

- **Solution 2: System-supported data privatization**

```
#pragma omp parallel private (...) reduction(+:lpe)  
{  
    ...  
    lpe += 0.5*vVal;  
    ...  
}  
// No reduction over the threads is required here
```

1. Create private copies of the variable (`lpe`) in the reduction clause for all the threads
2. Perform the specified reduction operation (+) on the variable at the end of the parallel section

Scalability Test: False Sharing Matters



- Having now understood how multithreading works, start using reduction clause in real-life code development

Some Like It as Arguments

- Use command line arguments for scaling tests without re-compiling multiple times
- **hmd.c → hmdarg.c** by adding the following lines in **main()**

```
int main(int argc, char **argv) {  
    ...  
    vthrd[0] = atoi(argv[1]); string-to-integer conversion  
    vthrd[1] = atoi(argv[2]); command-line argument  
    vthrd[2] = atoi(argv[3]);  
    nthrd = vthrd[0]*vthrd[1]*vthrd[2];  
    printf("Number of threads = %d\n", nthrd);  
    ...  
}
```

- **Compiling**

```
mpicc -o hmdarg hmdarg.c -fopenmp -lm
```

Strong-Scaling Test with hmdarg.c

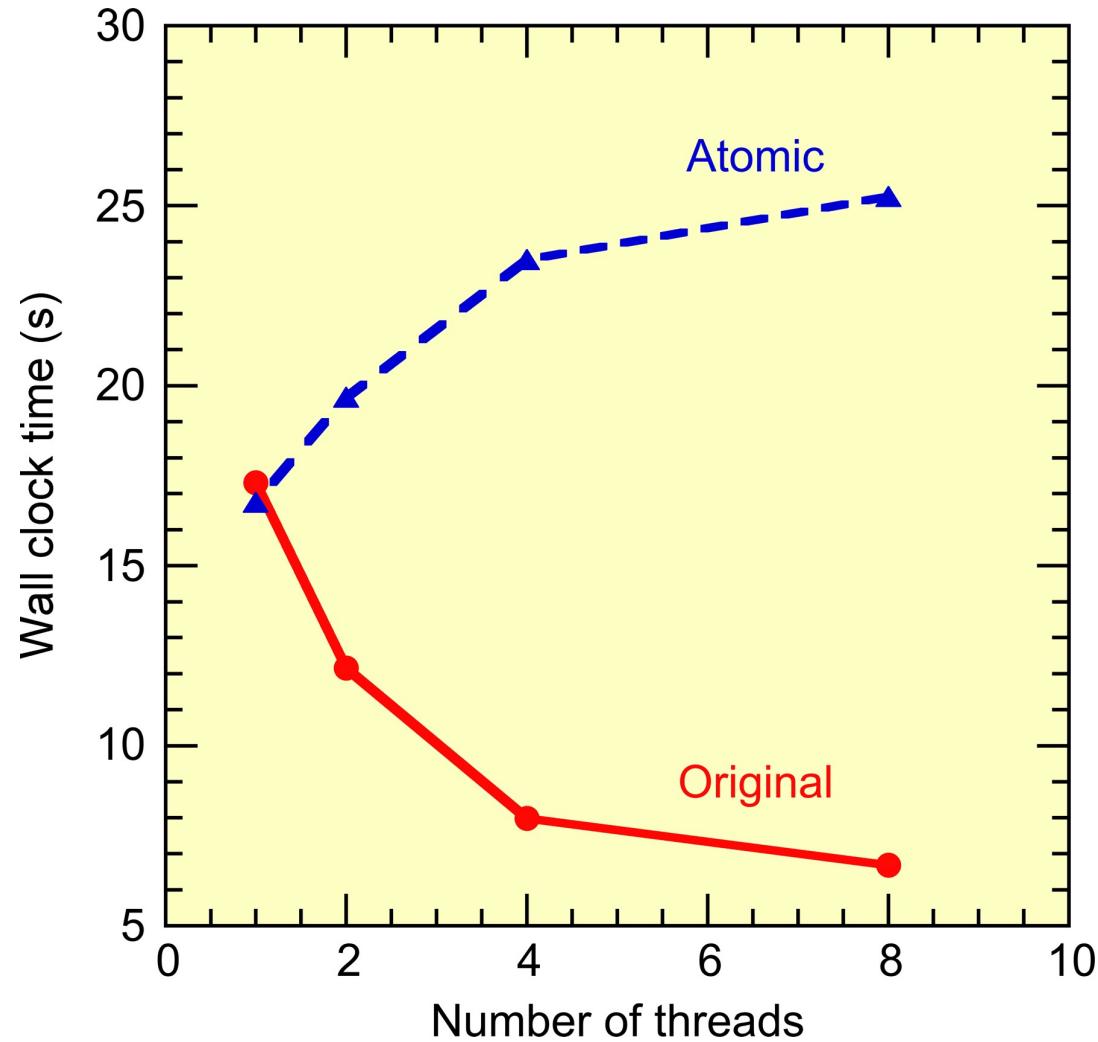
```
[anakano@discovery cs596]$ salloc --nodes=1 --ntasks-per-node=1 --cpus-per-task=8 -t 59
...
[anakano@d05-29 cs596]$ mpirun -bind-to none -n 1 ./hmdarg 1 1 1
Number of threads = 1
al = 4.103942e+01 4.103942e+01 4.103942e+01
lc = 16 16 16
rc = 2.564964e+00 2.564964e+00 2.564964e+00
thbk = 16 16 16
nglob = 55296
CPU & COMT = 1.073547e+01 2.005649e-02
[anakano@d05-29 cs596]$ mpirun -bind-to none -n 1 ./hmdarg 2 1 1
Number of threads = 2
...
thbk = 8 16 16
nglob = 55296
CPU & COMT = 6.804797e+00 1.980424e-02
[anakano@d05-29 cs596]$ mpirun -bind-to none -n 1 ./hmdarg 2 2 1
Number of threads = 4
...
thbk = 8 8 16
nglob = 55296
CPU & COMT = 4.956142e+00 1.981378e-02
[anakano@d05-29 cs596]$ mpirun -bind-to none -n 1 ./hmdarg 2 2 2
Number of threads = 8
...
thbk = 8 8 8
nglob = 55296
CPU & COMT = 4.078273e+00 2.253795e-02
```

Atomic Operation

- Restore Newton's third law & handle race conditions with the `omp atomic` directive

```
int bintra;  
...  
if (i<j && rr<rrCut) {  
    ...  
    if (bintra)  
        lpe_td[std] += vVal;  
    else  
        lpe_td[std] += 0.5*vVal;  
    for (a=0; a<3; a++) {  
        f = fcVal*dr[a];  
        ra[i][a] += f;  
        if (bintra) {  
            #pragma omp atomic  
            ra[j][a] -= f; // Different threads can access the same atom  
        }  
    }  
}
```

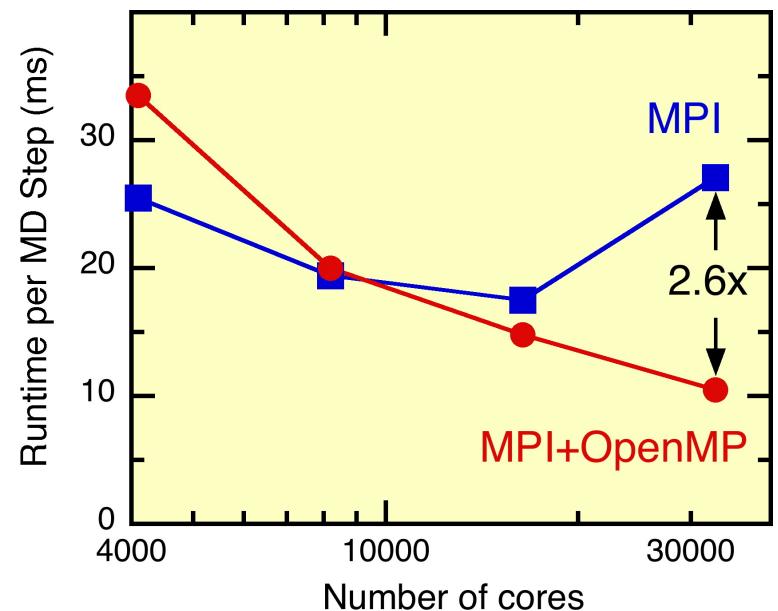
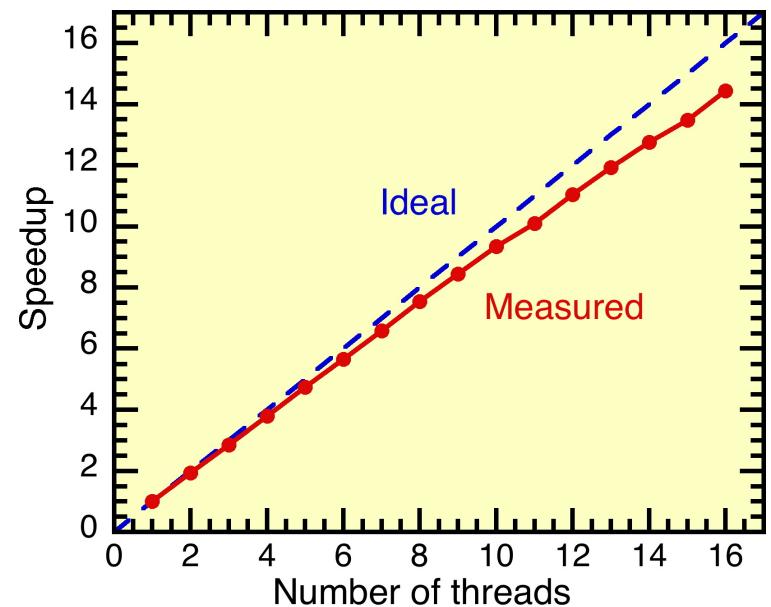
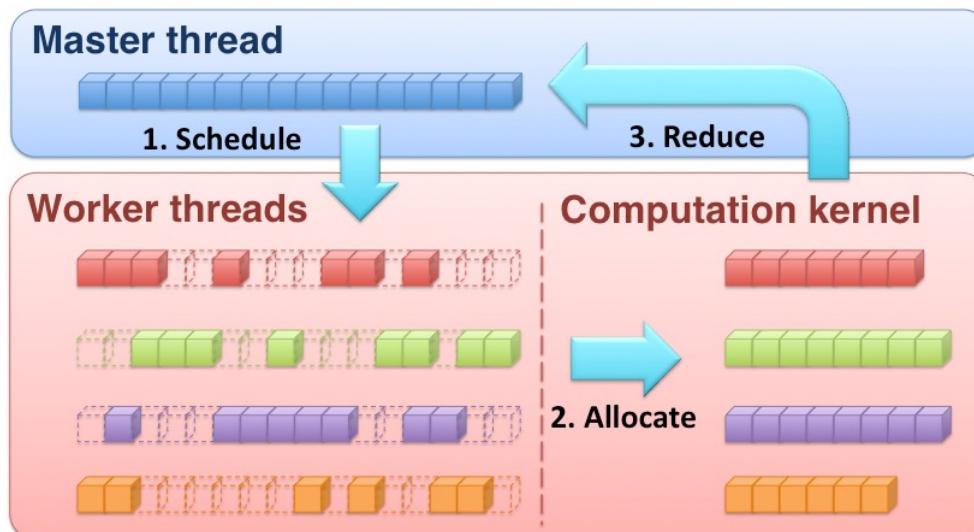
Atomic Operation Is Expensive



Spatially Compact Thread Scheduling

Concurrency-control mechanism:
Data privatization (duplicate the force array)

- Reduced memory: $\# \text{ of atoms} \rightarrow \# \text{ of threads}$
 $\Theta(nq) \rightarrow \Theta(n+n^{2/3}q^{1/3})$
- Strong scaling parallel efficiency 0.9 on quad quad-core AMD Opteron
- 2.6x speedup over MPI by hybrid MPI+OpenMP on 32,768 IBM Blue Gene/P cores



Concurrency-Control Mechanisms

A number of concurrency-control mechanisms (CCMs) are provided by OpenMP to coordinate multiple threads:

- Critical section: Serialization
- Atomic update: Expensive hardware instruction
- Data privatization: Requires large memory $\Theta(nq)$
- Hardware transactional memory: Rollbacks (on IBM Blue Gene/Q)

CCM performance varies:

- Depending on computational characteristics of each program
- In many cases, CCM degrades performance significantly

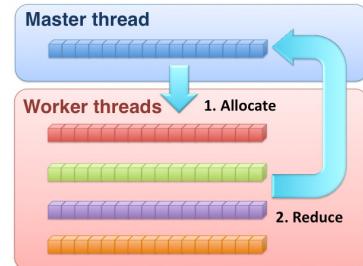
HTM/critical section

```
#pragma omp <critical|tm_atomic>
{
    ra[i][0] += fa*dr[0];
    ra[i][1] += fa*dr[1];
    ra[i][2] += fa*dr[2];
}
```

Atomic update

```
#pragma omp atomic
    ra[i][0] += fa*dr[0];
#pragma omp atomic
    ra[i][1] += fa*dr[1];
#pragma omp atomic
    ra[i][2] += fa*dr[2];
```

Data privatization



Goal: Provide a guideline to choose the “right” CCM

Hardware Transactional Memory

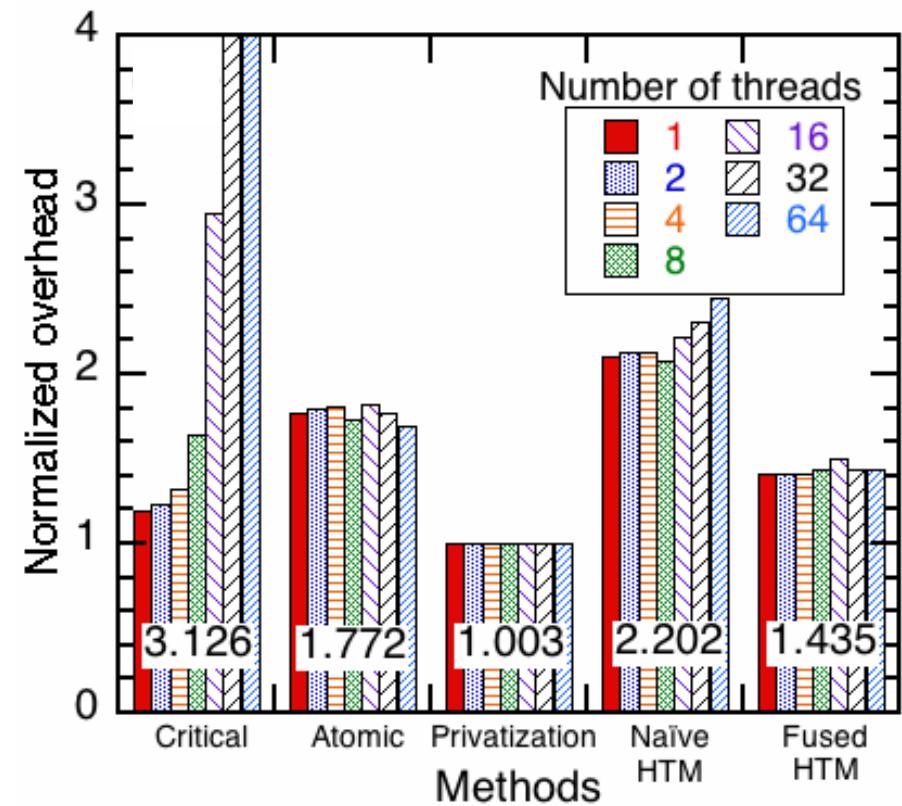
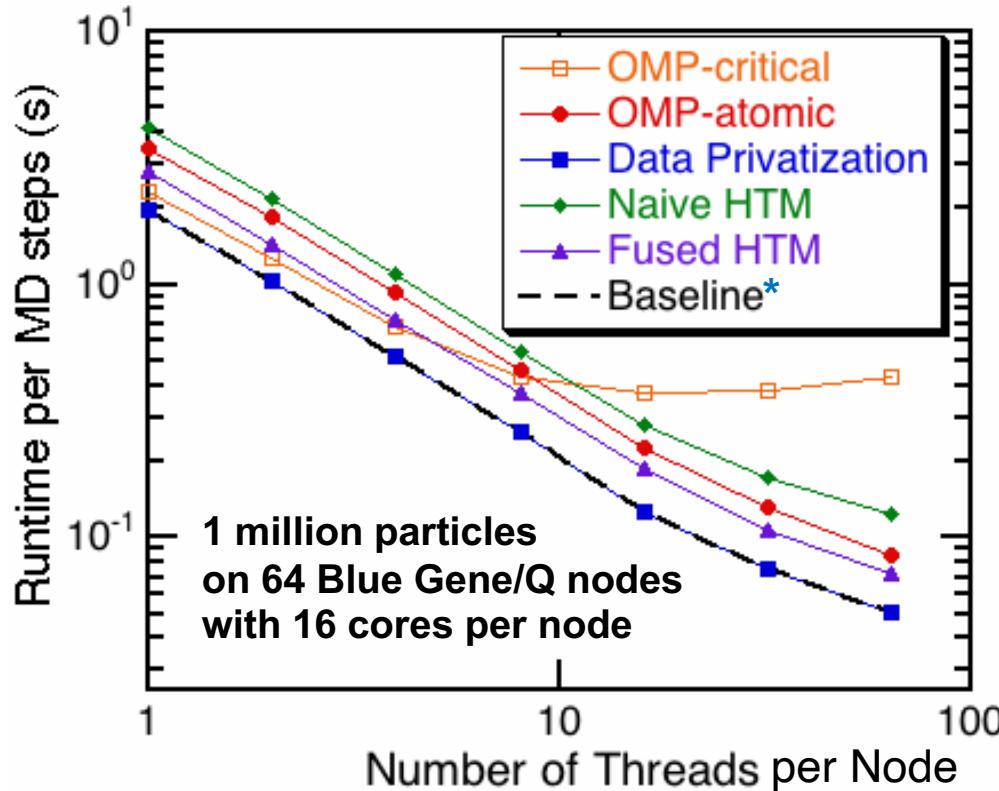
Transactional memory (TM): An opportunistic CCM

- Avoids memory conflicts by monitoring a set of speculative operations (*i.e.*, transaction)
- If two or more transactions write to the same memory address, transaction(s) will be restarted—a process called rollback
- If no conflict detected in the end of a transaction, operations within the transaction becomes permanent (*i.e.*, committed)
- Software TM usually suffers from large overhead

Hardware TM on IBM Blue Gene/Q:

- The first commercial platform implementing TM support at hardware level *via* multiversioned L2-cache
- Hardware support is expected to reduce TM overhead
- Performance of HTM on molecular dynamics has not been quantified

Strong-Scaling Benchmark for MD

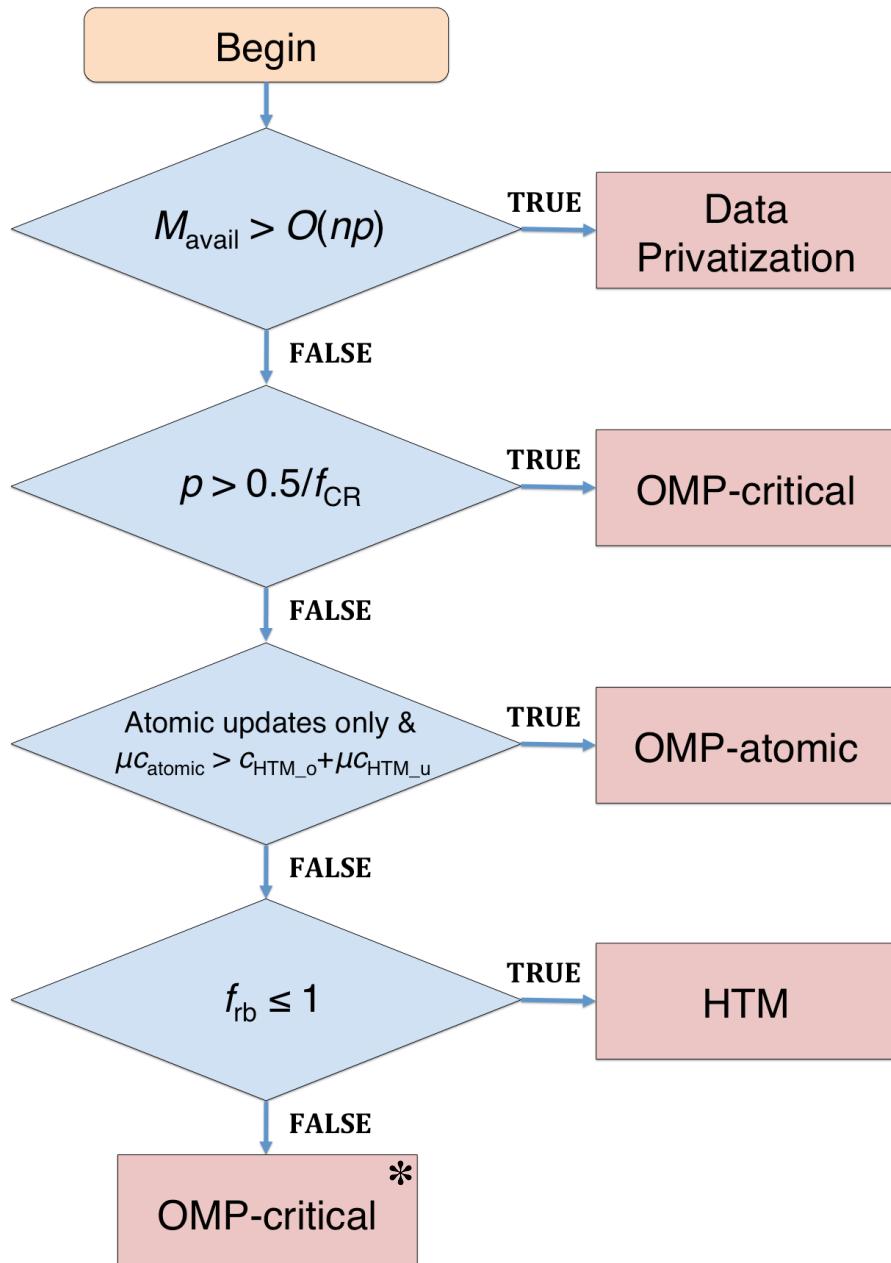


*Baseline: No CCM; the result is wrong

Developed a fundamental understanding of CCMs:

- OMP-critical has limited scalability on larger number of threads ($q > 8$)
- Data privatization is the fastest, but it requires $\Theta(nq)$ memory
- Fused HTM performs the best among constant-memory CCMs

Threading Guideline for Scientific Programs

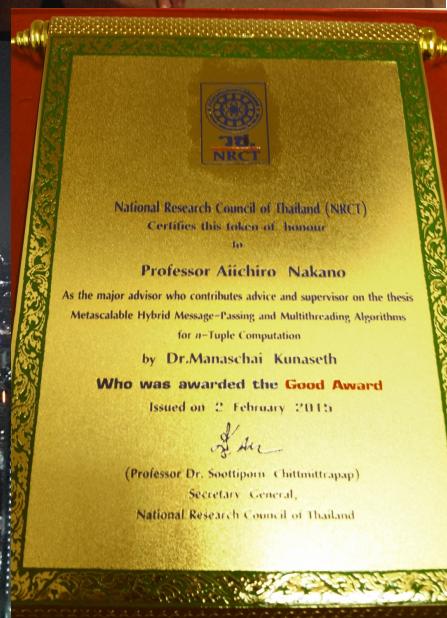
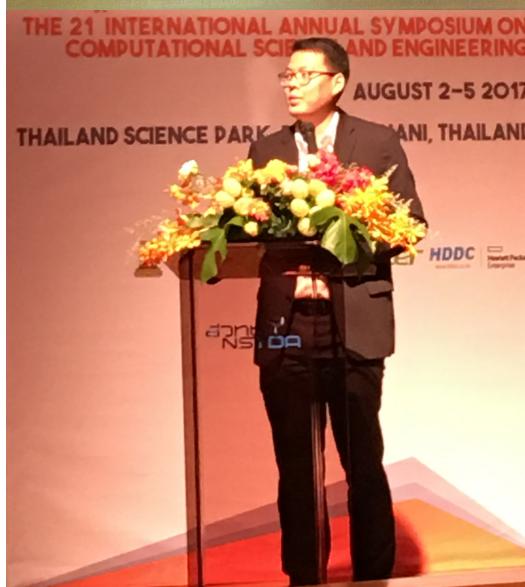


**Focus on minimizing runtime
(best performance):**

- Have enough memory → data privatization
- Conflict region is small → OMP-critical
- Small amount of updates → OMP-atomic
- Conflict rate is low → HTM
- Other → OMP-critical* (poor performance)

Concurrency control mechanism	Parallel efficiency
OMP-critical	$e = \min\left(\frac{1}{pf_{\text{CR}}}, 1\right)$
OMP-atomic	$e = \frac{t_{\text{total}}}{t_{\text{total}} + m\mu c_{\text{atomic}}}$
Data privatization	$e = \frac{t_{\text{total}}}{t_{\text{total}} + c_{\text{reduction}} n \log p}$
HTM	$e = \frac{t_{\text{total}}}{t_{\text{total}} + m(c_{\text{HTM_overhead}} + \mu c_{\text{HTM_update}})}$

IEEE PDSEC Best Paper & Beyond



It All Started as a CSCI596 Final Project

