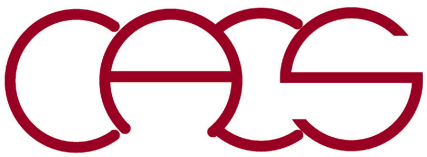
Hybrid MPI+OpenMP Parallel MD

Aiichiro Nakano

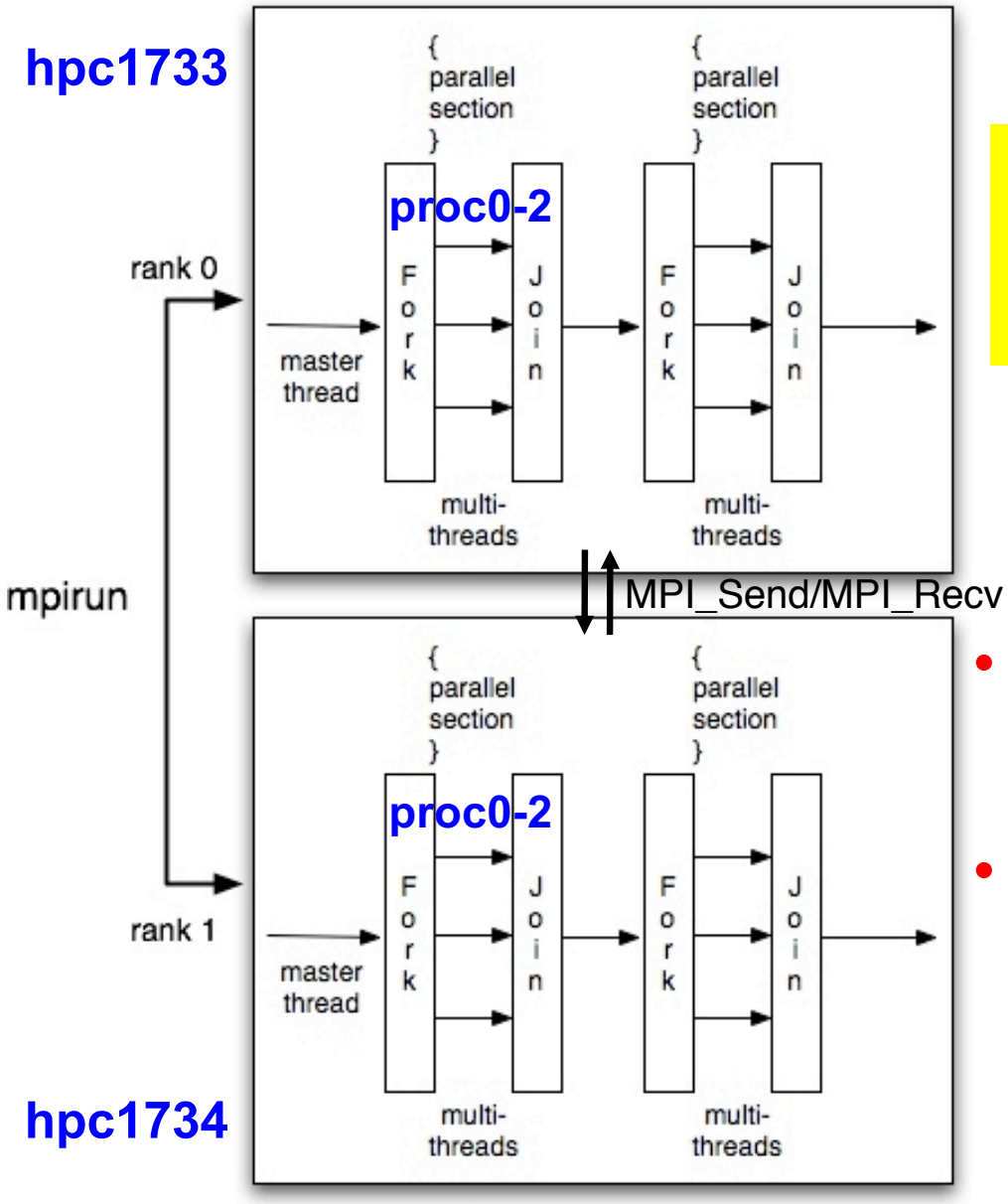
Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Chemical Engineering & Materials Science
Department of Biological Sciences
University of Southern California

Email: anakano@usc.edu



Hybrid MPI+OpenMP Programming

Each MPI process spawns multiple OpenMP threads



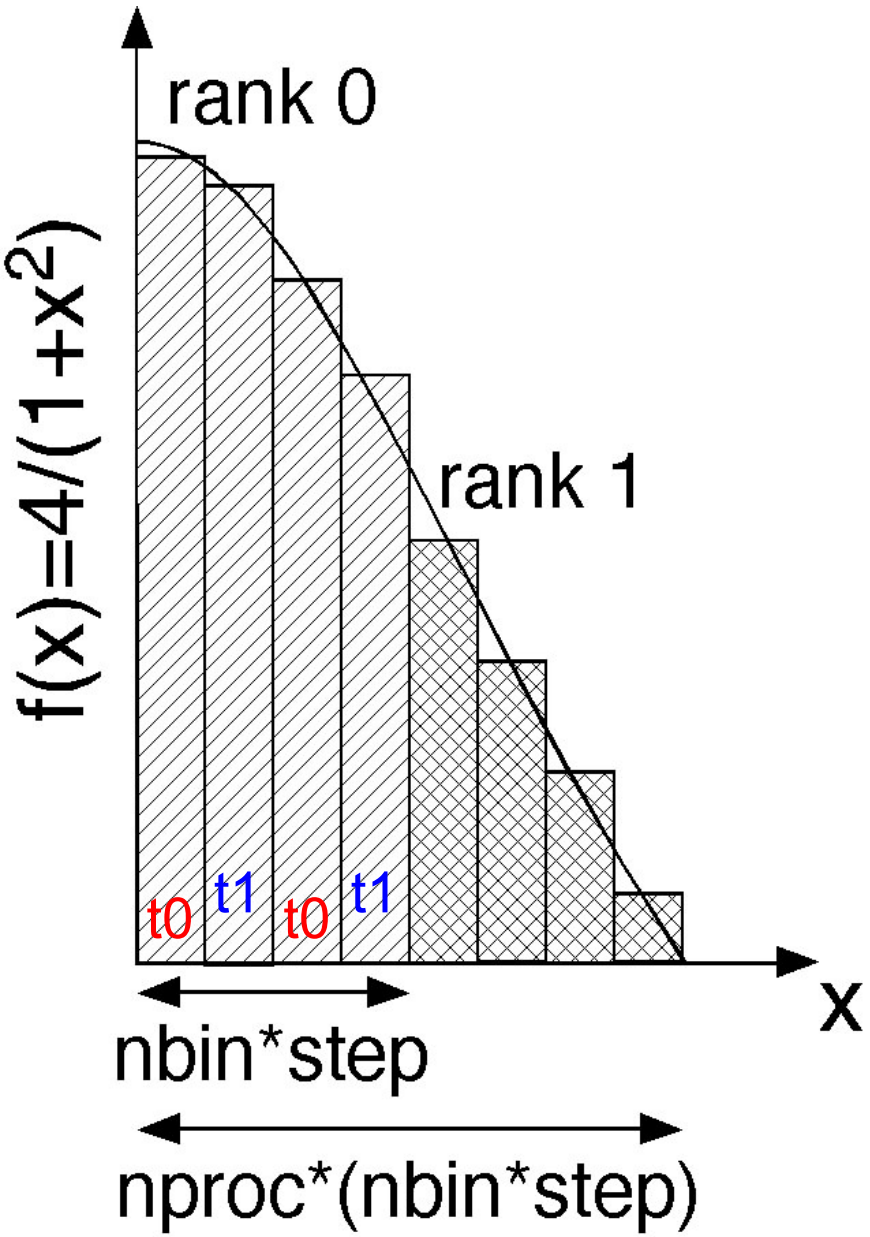

```
In a Slurm script:
  mpirun –np 2
In the code:
  omp_set_num_threads(3);
```

- MPI processes communicate by sending/receiving messages
- OpenMP threads communicate by writing to/reading from shared variables

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`
- **Task interleaving:** Within each MPI process, `nthreads` OpenMP threads perform part of the sum as in `omp_pi.c`

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



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) {
      x = (i+0.5)*step; sum[tid] += 4.0/(1.0+x*x); }
    printf("rank tid sum = %d %d %e\n",myid,tid,sum[tid]);
  }
  for (tid=0; tid<nthreads; tid++) pi += sum[tid]*step;
  MPI_Allreduce(&pi,&pig,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  if (myid==0) printf("PI = %f\n",pig);
  MPI_Finalize();}
```

Shared variables among all threads

nbin is # of bins per MPI rank

Local variables: Different values needed for different threads

Who does what!

Data privatization to avoid race conditions

Thread & rank reductions

MPI+OpenMP Example: `hpi.c`

- **Compilation on `hpc-login3.usc.edu`**

```
source /usr/usc/openmpi/default/setup.sh
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 lc_an2
WORK_HOME=/home/rcf-proj/an2/anakano
cd $WORK_HOME
srun -n $SLURM_NNODES ./hpi
```

1 MPI rank/node

2 OpenMP threads/rank

Number of computing nodes

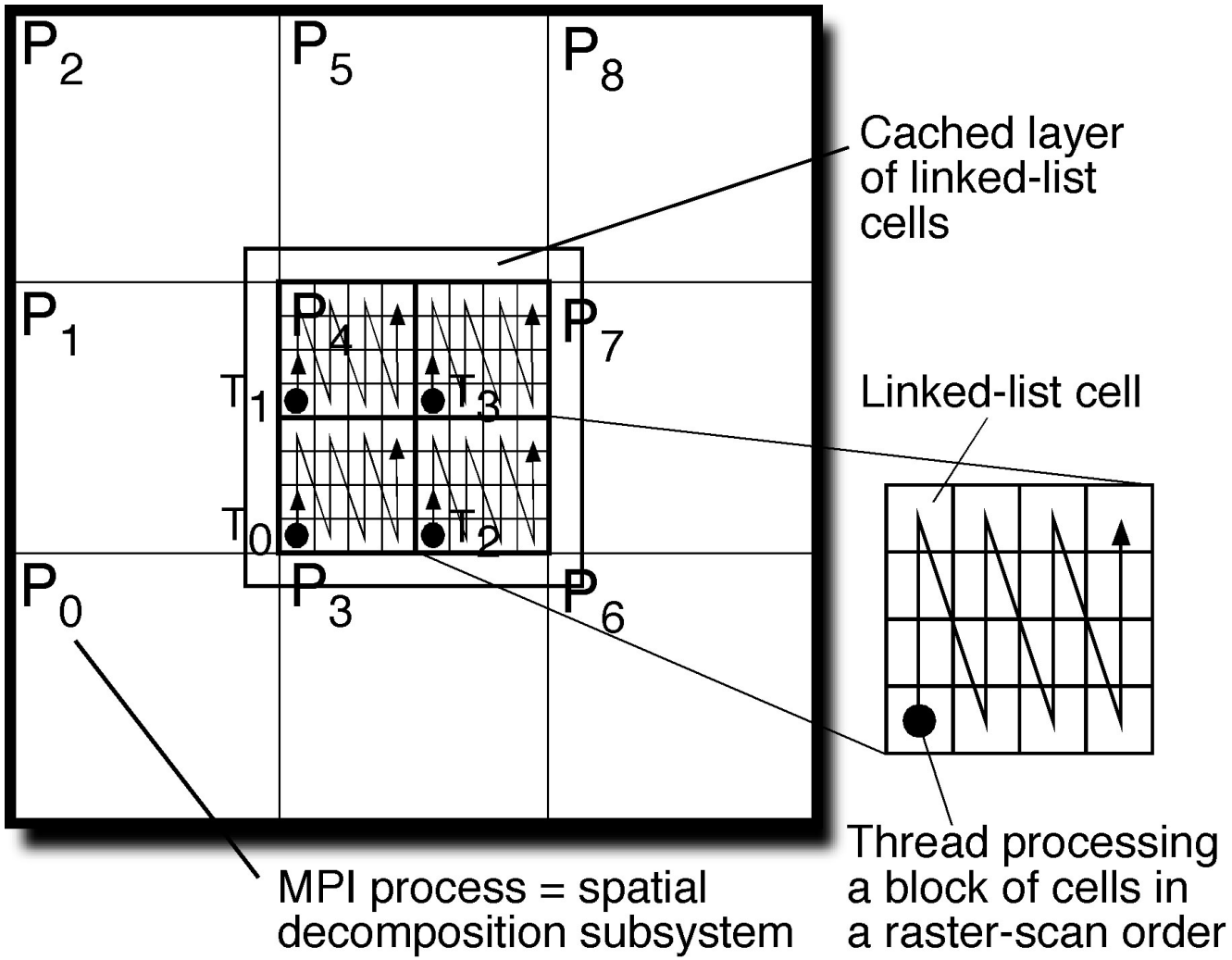
- **Output**

```
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
```

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 $\xrightarrow[\text{map}]{}$ thread



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];
```

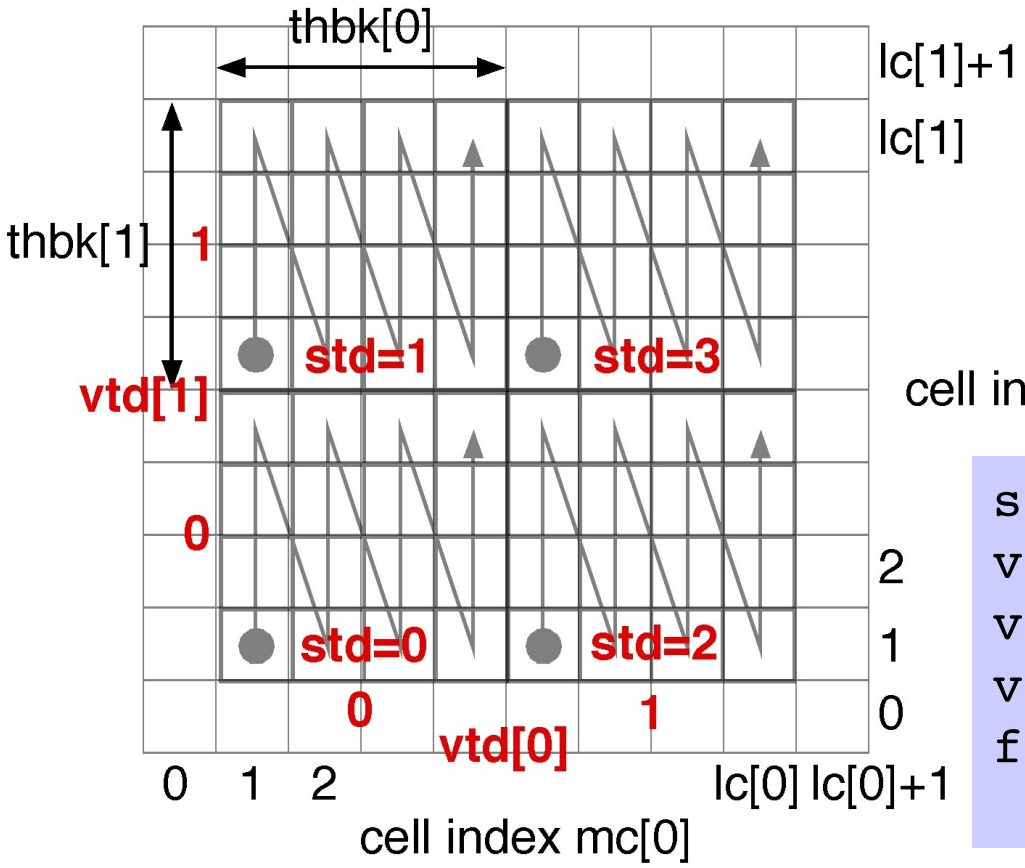
In init_params():

```
/* 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 x|y|z element of vector thread index.
- `mofst[3]`: `mofst[0|1|2]` is the x|y|z offset cell index of cell-block.




```
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**

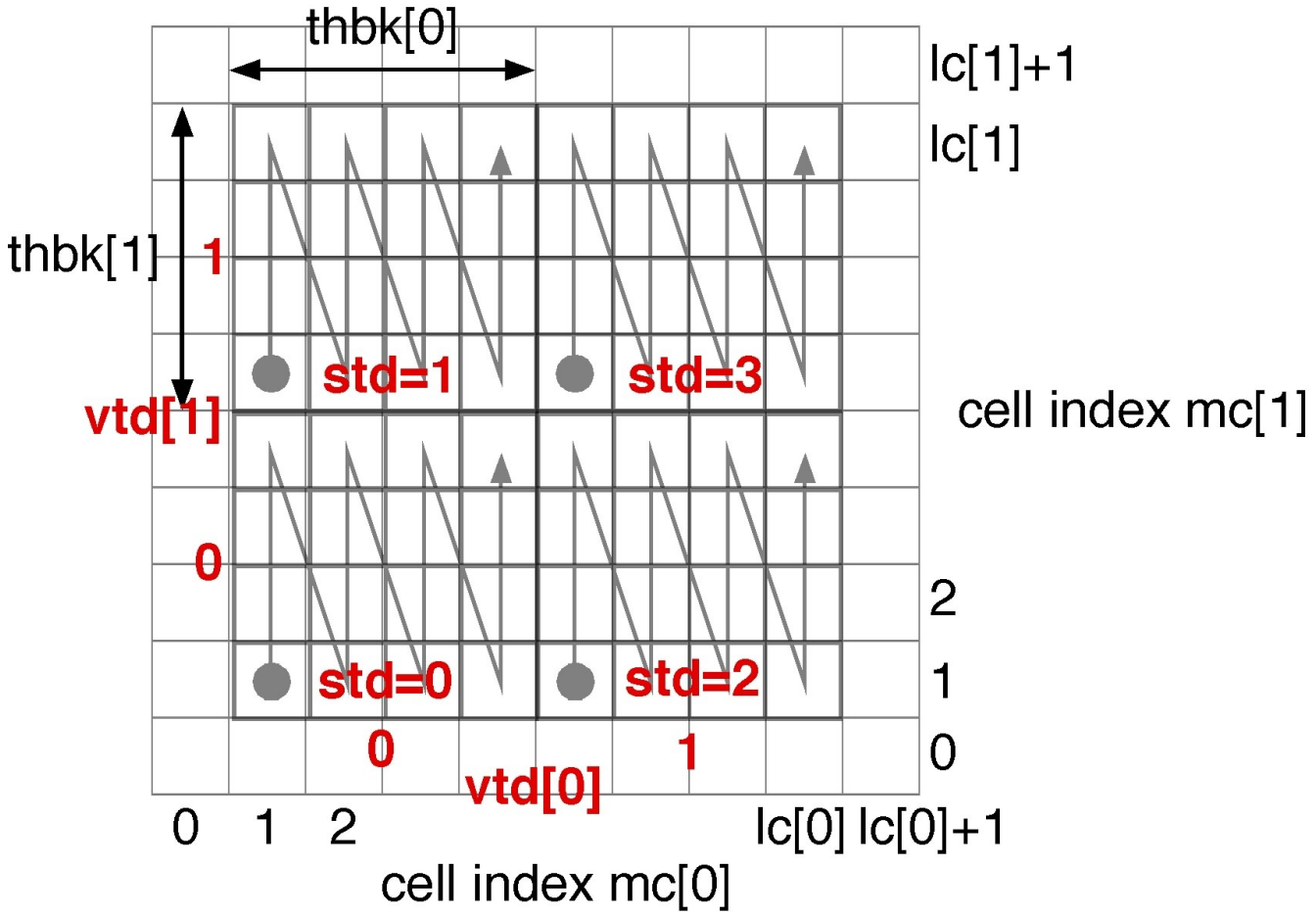
```
#pragma omp parallel private(mc,...)
{
  ...
  for (mc[0]=mofst[0]+1; mc[0]<=mofst[0]+thbk[0]; (mc[0])++)
  for (mc[1]=mofst[1]+1; mc[1]<=mofst[1]+thbk[1]; (mc[1])++)
  for (mc[2]=mofst[2]+1; mc[2]<=mofst[2]+thbk[2]; (mc[2])++)
  {
    Each thread handles thbk[0]×thbk[1]×thbk[2] cells independently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

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



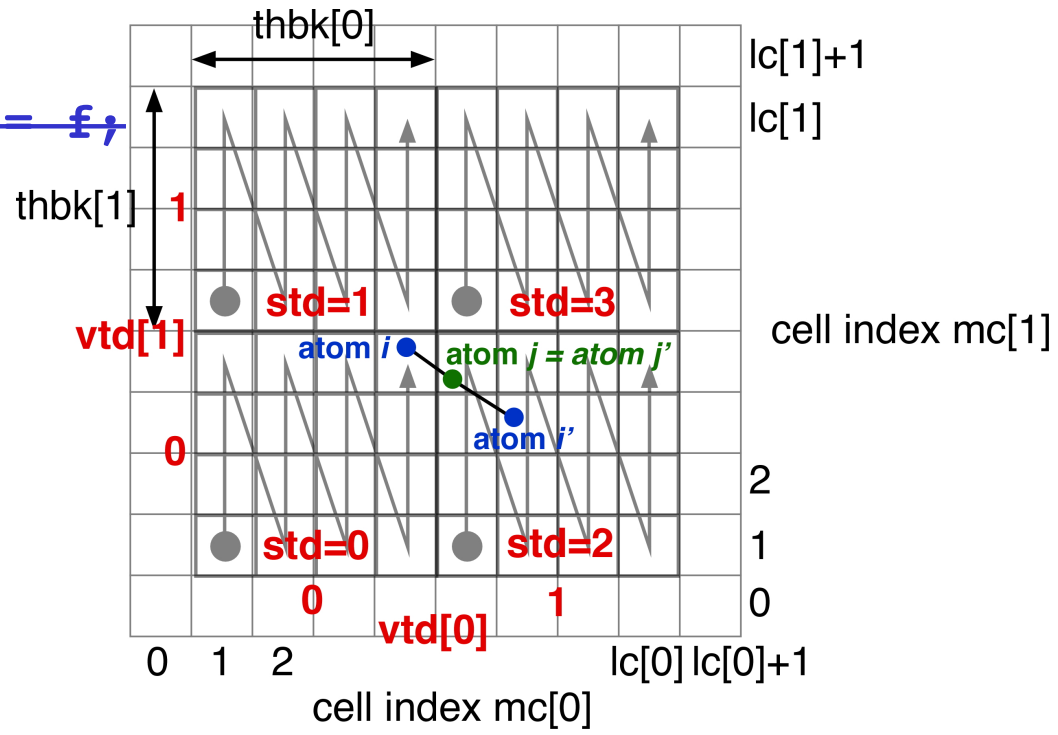
Data privatization: *cf.* `omp_pi.c` & `hpi.c`

Avoiding Critical Sections (2)

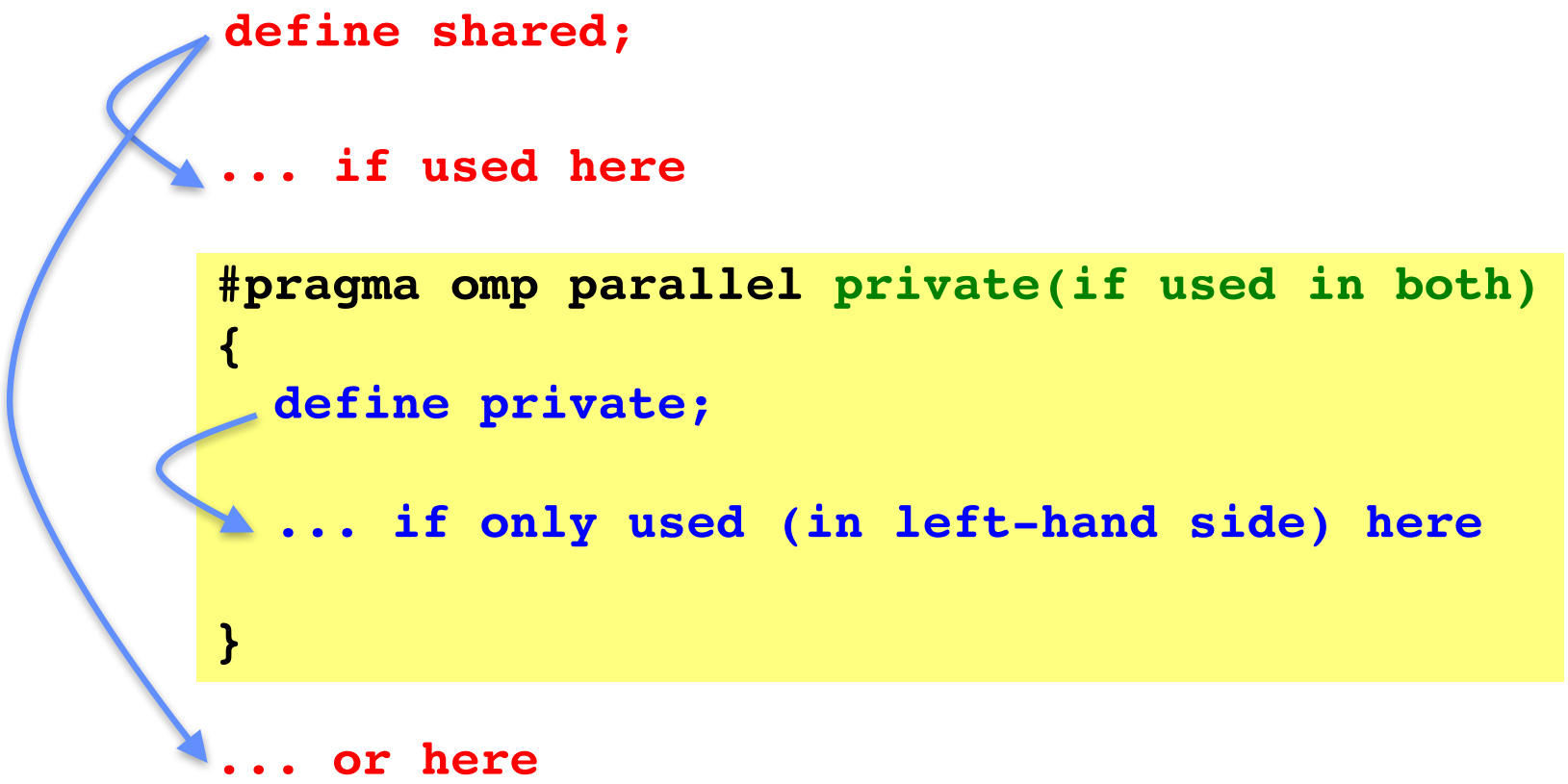
- To avoid multiple threads to access an identical force array element, stop using the Newton's third law:

```
int bintra;
...
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



OpenMP Essential



Running HMD at HPC

- **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 lc_an2

WORK_HOME=/home/rcf-proj/an2/anakano
cd $WORK_HOME

srun -n 2 ./hmd
```

Interactively Running HMD at HPC (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: Granted job allocation 1784580
salloc: Waiting for resource configuration
salloc: Nodes hpc[0966,0969] are ready for job
----------------------------------------
Begin SLURM Prolog Fri Sep 28 11:03:56 2018
Job ID:          1784580
Username:        anakano
Accountname:     lc_an2
Name:            sh
Partition:       quick
Nodes:           hpc[0966,0969]
TasksPerNode:    1(x2)
CPUSPerTask:     4
TMPDIR:          /tmp/1784580.quick
SCRATCHDIR:      /staging/scratch/1784580
Cluster:         uschpc
HSDA Account:    false
End SLURM Prolog
----------------------------------------
[anakano@hpc0966 anakano]$
```

hpc0965	hpc0972	8	Dual Hexcore Intel Xeon 3.0 GHz 24GB Memory	160GB	main large quick	sl160	Xeon	x86_64	hexcore

You are logged in to one of the allocated nodes

Interactively Running HMD at HPC (2)

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

```
$ srun -n 2 ./hmd
```

3. **While the job is running, you can open another window & log in to both the nodes to check that all processors on each node are busy. Type 'H' to show individual threads.**

```
[anakano@hpc-login3 ~]$ ssh hpc0966
[anakano@hpc0969 ~]$ top (then type H)
KiB Mem : 24512700 total, 16214580 free,   712256 used,  7585864 buff/cache
KiB Swap:  8388604 total,  8290972 free,    97632 used. 22798280 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 9265 anakano   20   0  209016  20168   3208 R 99.9  0.1   1:18.03 hmd
 9281 anakano   20   0  209016  20168   3208 R 87.8  0.1   1:08.47 hmd
 9282 anakano   20   0  209016  20168   3208 R 87.8  0.1   1:08.47 hmd
 9280 anakano   20   0  209016  20168   3208 R 87.5  0.1   1:08.47 hmd
 9243 anakano   20   0  160336   2328   1552 R  1.0  0.0   0:00.86 top
  622 root      20   0       0      0      0 S  0.3  0.0  25:16.07 xfsaild/sda3
    1 root      20   0   43604   3408   2136 S  0.0  0.0   0:51.15 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:03.91 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:17.43 ksoftirqd/0
...
```

Type 1 to show core-usage summary

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
```

```
0.050000  0.877345 -5.137153 -3.821136
0.100000  0.462056 -4.513097 -3.820013
0.150000  0.510836 -4.587287 -3.821033
0.200000  0.527457 -4.611958 -3.820772
0.250000  0.518668 -4.598798 -3.820796
0.300000  0.529023 -4.614343 -3.820808
0.350000  0.532890 -4.620133 -3.820798
0.400000  0.536070 -4.624899 -3.820794
0.450000  0.539725 -4.630387 -3.820799
0.500000  0.538481 -4.628514 -3.820792
```

Time — Temperature — Potential energy — Total energy

See the lecture on "order-invariant real-number summation"

Strong Scalability of Hybrid MD

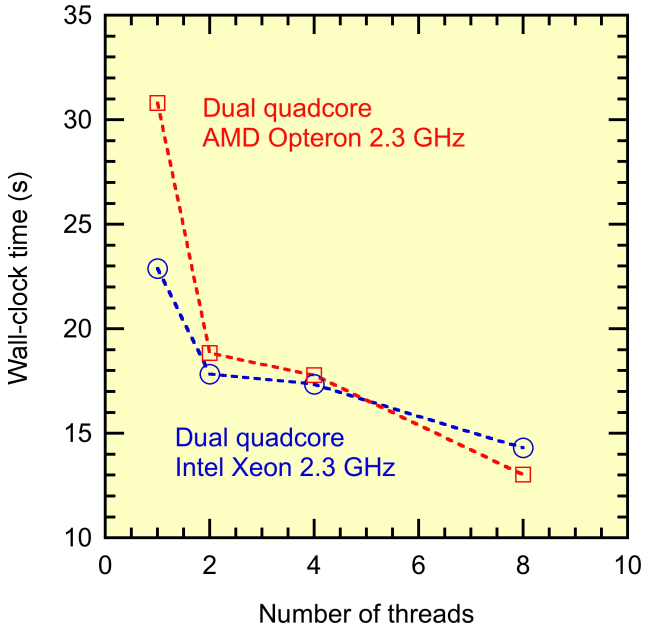
1 MPI process; 1-8 threads

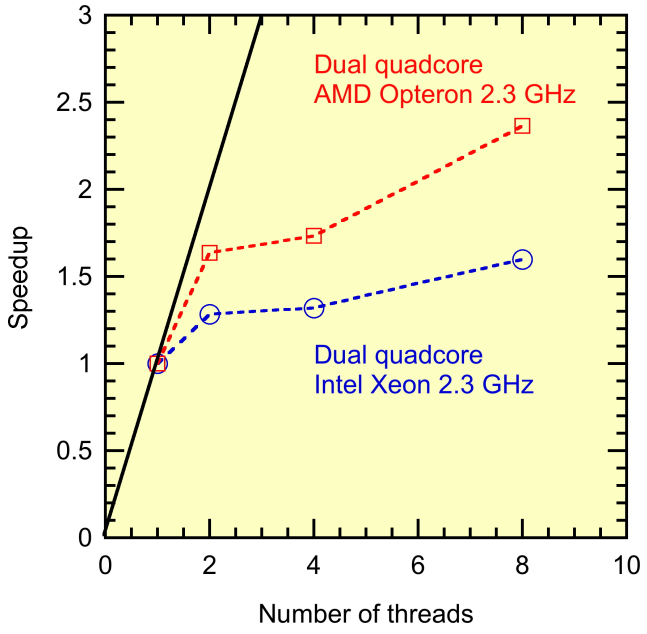
In hmd.h:

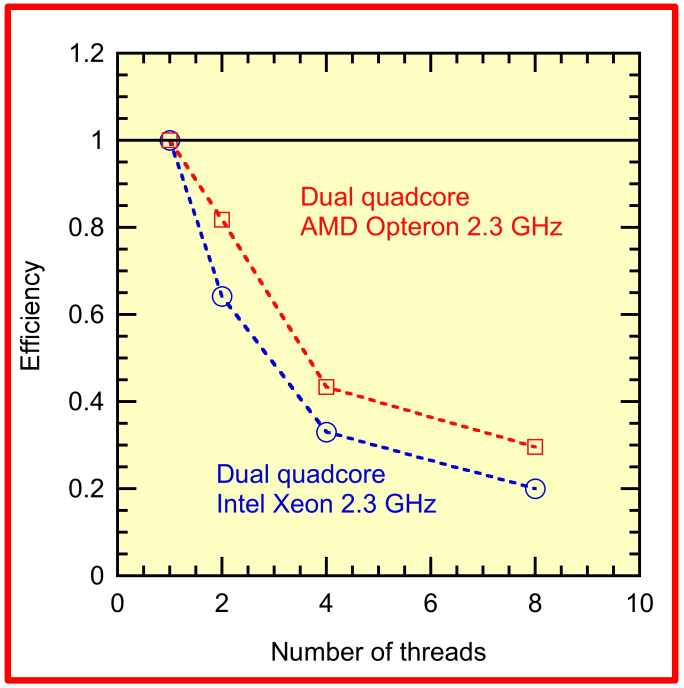
```
vproc = {1,1,1}, nproc = 1;
vthrd = {1,1,1}, nthrd = 1;
         2 1 1           2
         2 2 1           4
         2 2 2           8
```

pmd.in

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







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

$N = 4 \times 24^3$

$= 55296$ 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

`#pragma omp parallel only once` in main() to minimize overhead

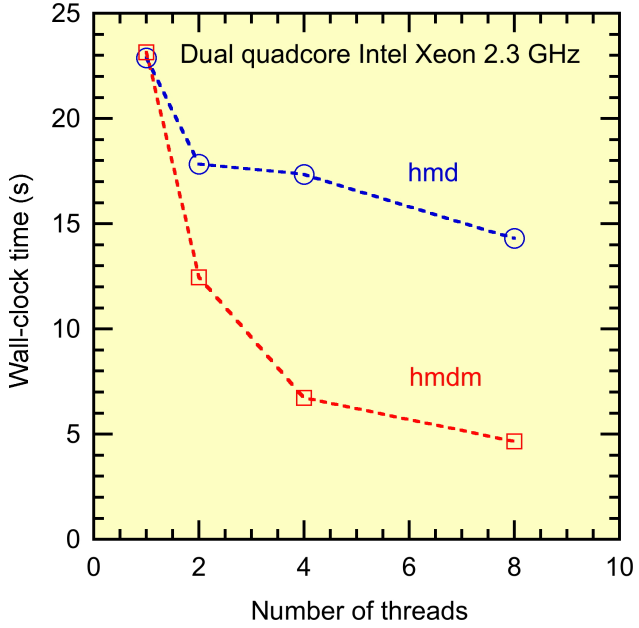
1 MPI process; 1-8 threads

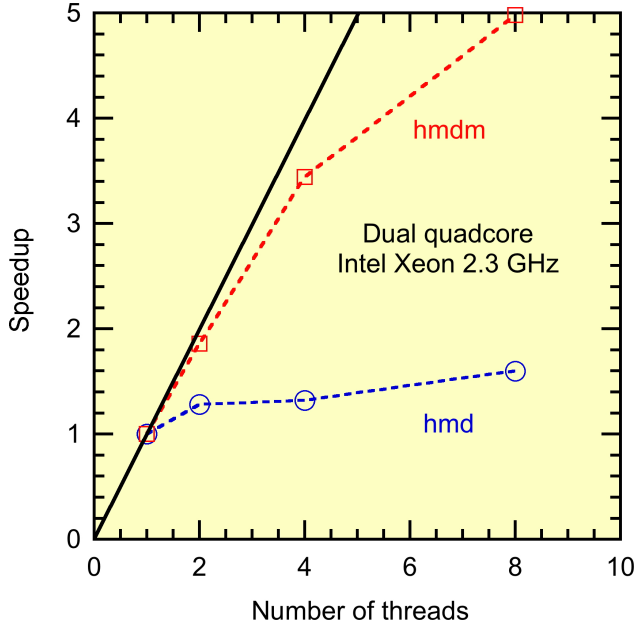
In hmd.h:

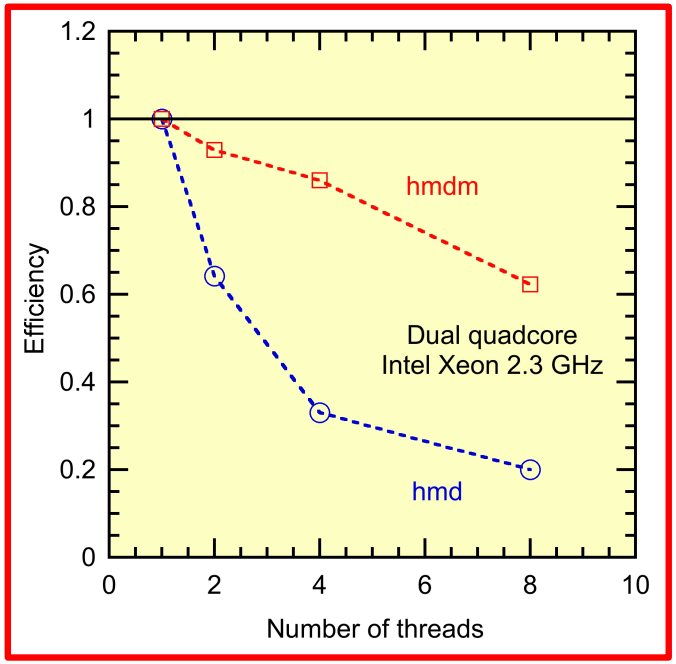
```
vproc = {1,1,1}, nproc = 1;
vthrd = {1,1,1}, nthrd = 1;
         2 1 1          2
         2 2 1          4
         2 2 2          8
```

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

mpirun ... –np 1 ... ./hmd1
                          2
                          4
                          8
```








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

$N = 4 \times 24^3$

$= 55296$ atoms

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

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

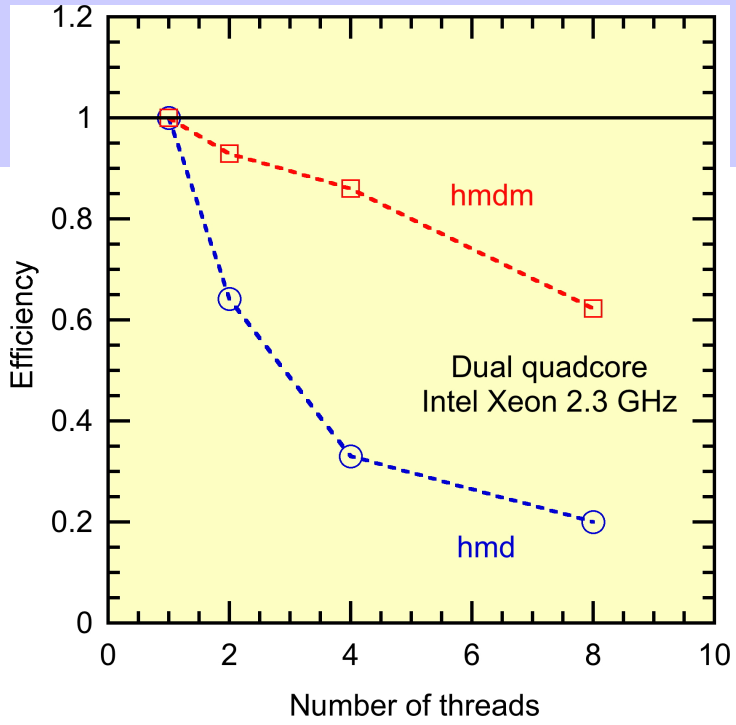
P: Number of cores

More on Multithreading MD

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

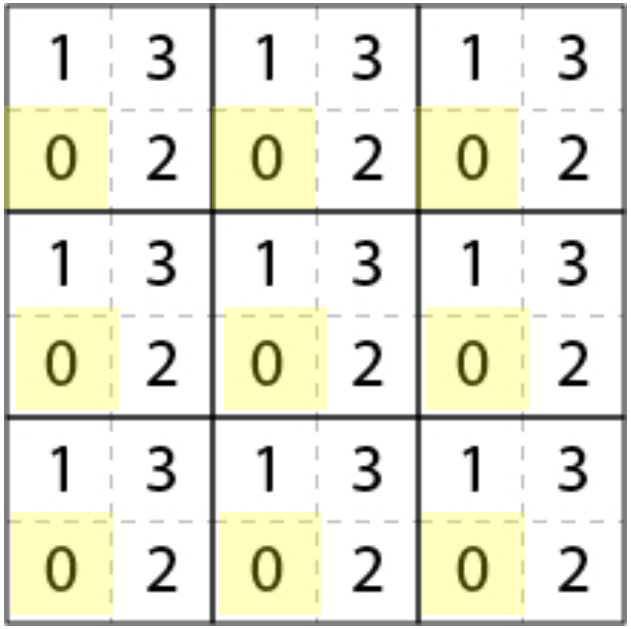
In hmdm.c:

```
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



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**

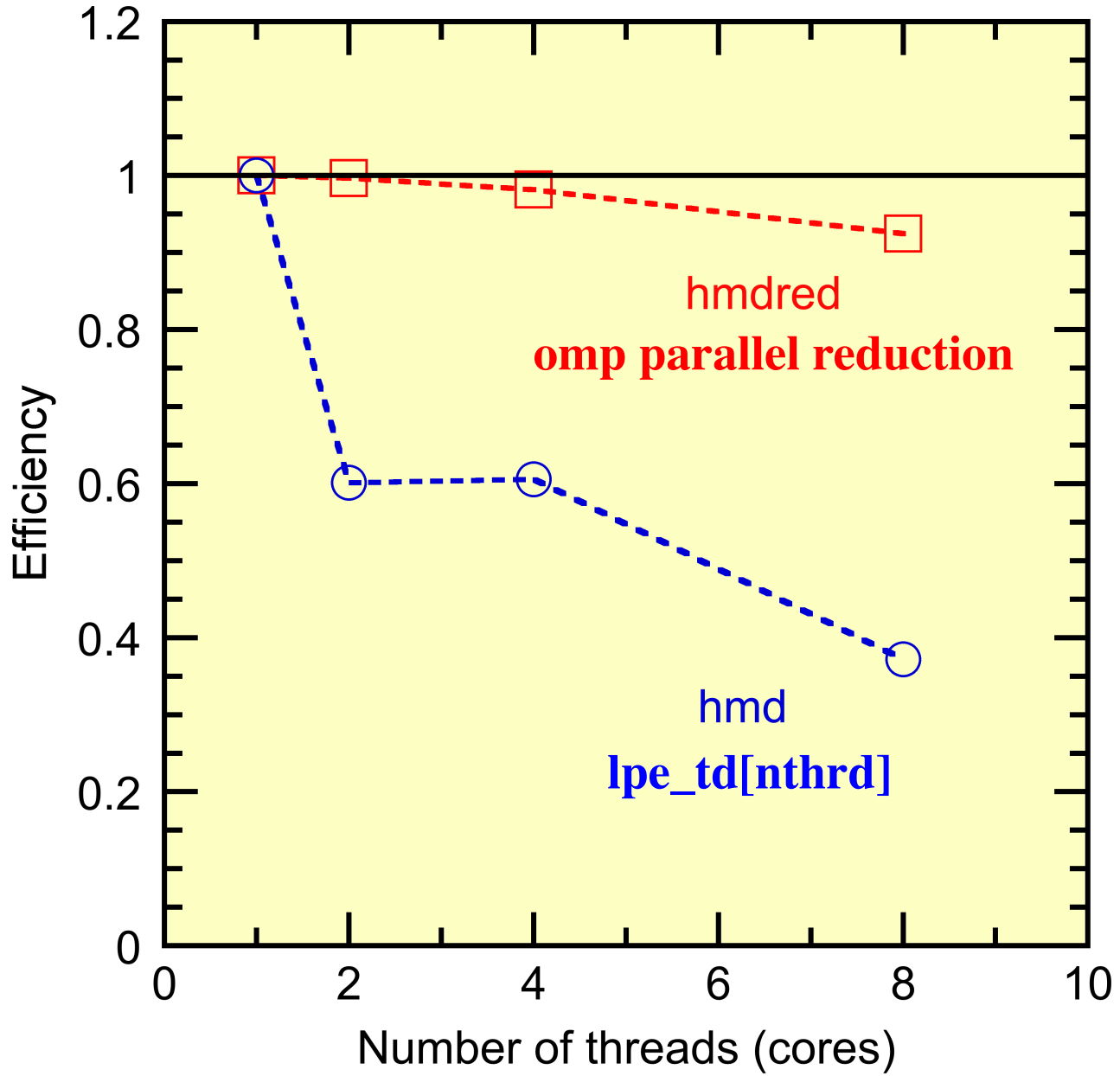
```
struct lpe_t {
  double lpe;
  double pads[7]; // assume intel CPU with 64 byte cache line
};
struct lpe_t lpe_td[nthrd];
```

- **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



Atomic Operation

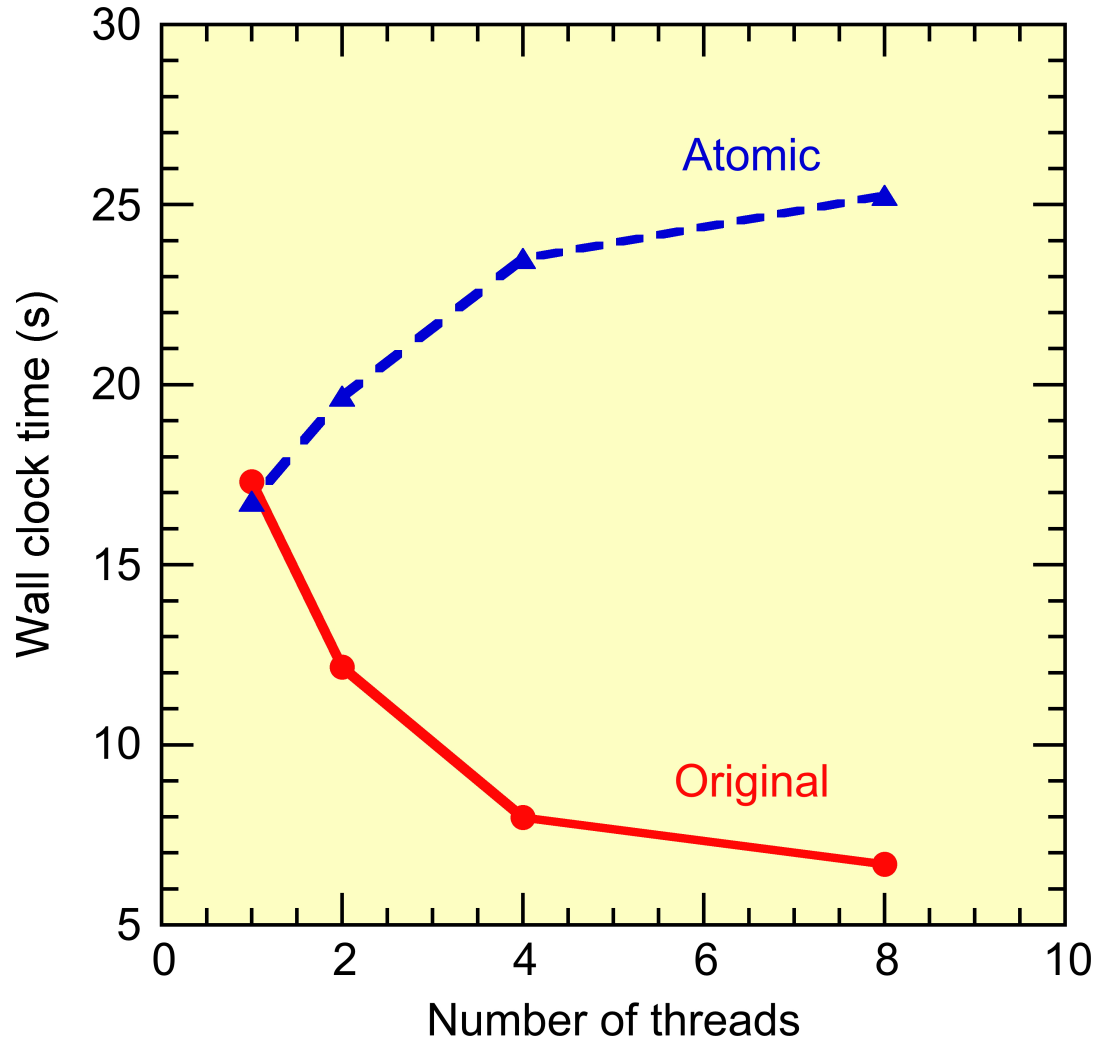
- **Restore Newton's third law & handle race conditions with the omp atomic directive**

```
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
    }
  }
}
```

Simpler solution than graph coloring?

Atomic Operation Is Expensive

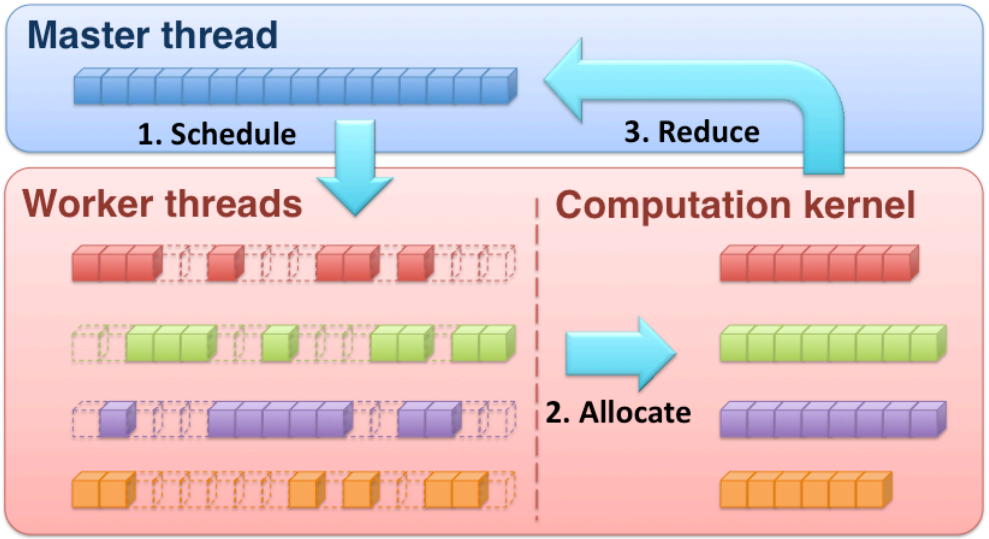
No, getting slower!

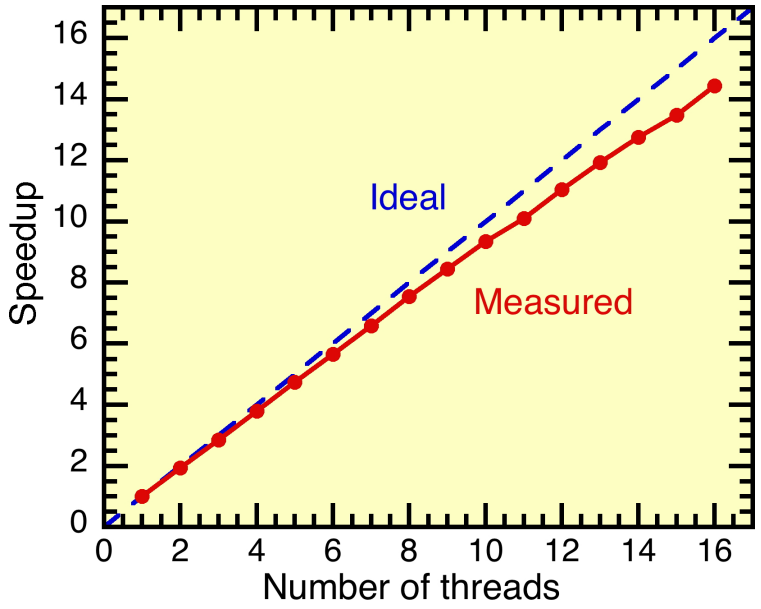


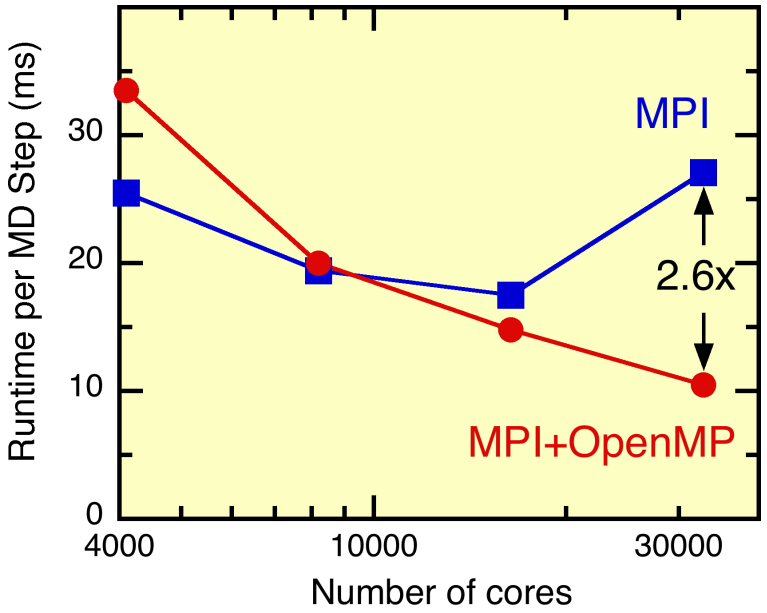
Spatially Compact Thread Scheduling

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

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







M. Kunaseth *et al.*, *PDPTA'11*; *J. Supercomput.* ('13)

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)**

# of atoms per node

# of threads

CCM performance varies:

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

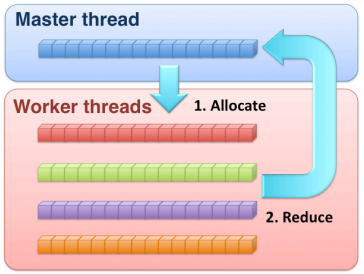
HTM/critical section

```
#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

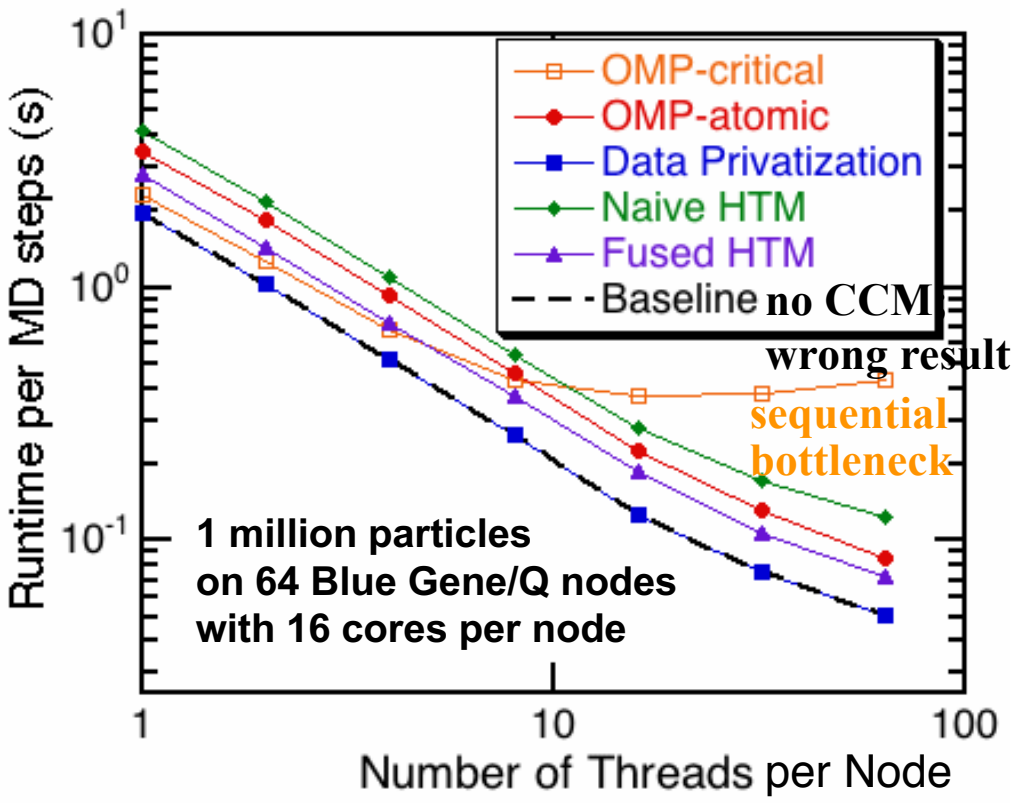
Transactional memory (TM): An opportunistic CCM

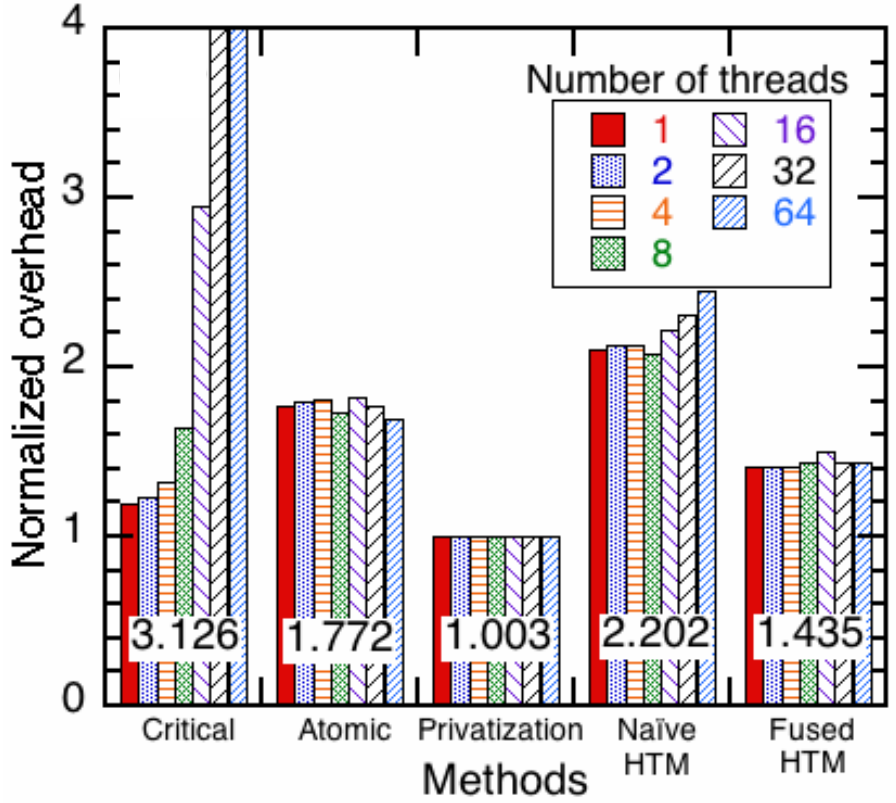
- **Avoids memory conflicts by monitoring a set of speculative operations (*i.e.* transaction)**
- **If two or more transactions write to the same memory address, transaction(s) will be restarted—a process called rollback**
- **If no conflict detected in the end of a transaction, operations within the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





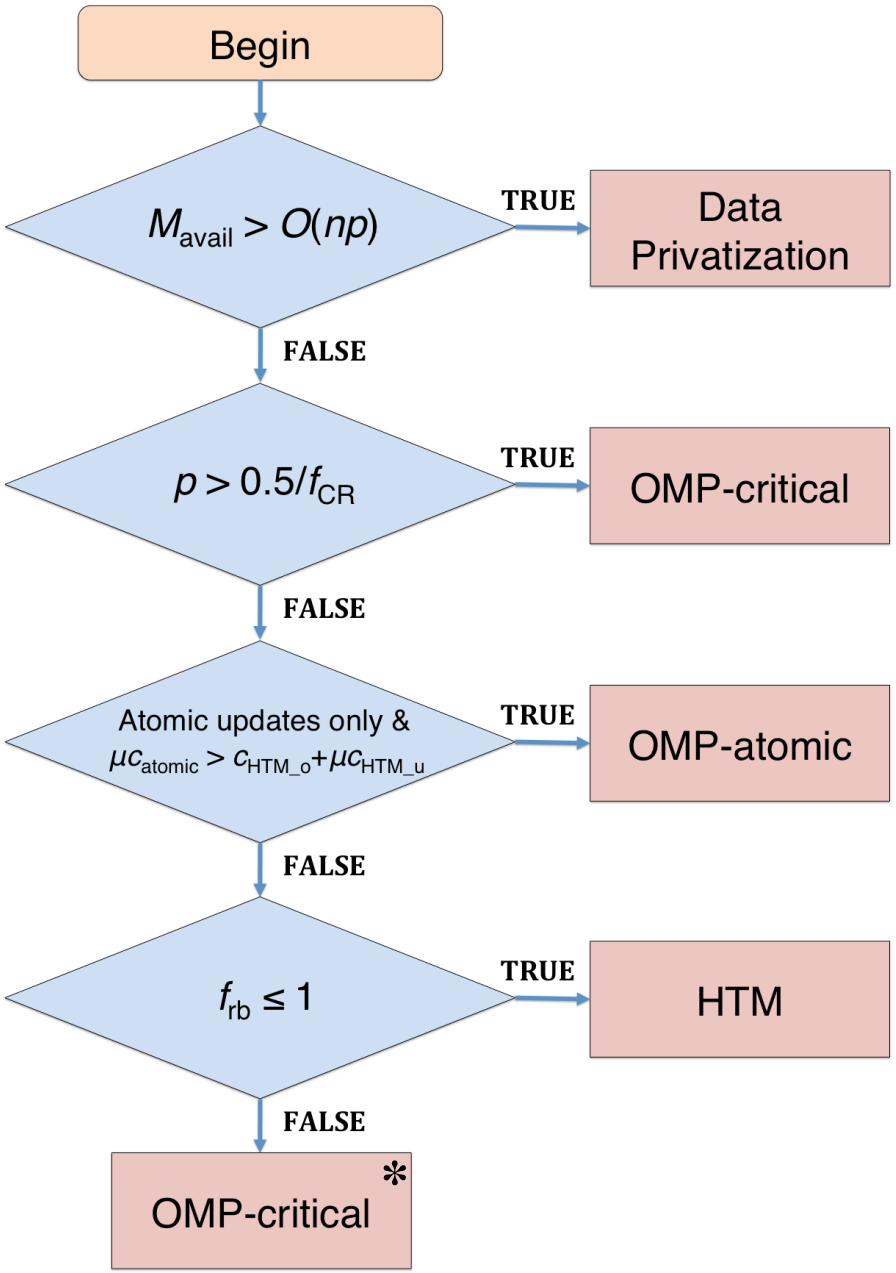
*Baseline: No CCM; the result is wrong

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**

M. Kunaseth *et al.*, *PDSEC'13* Best Paper

Threading Guideline for Scientific Programs



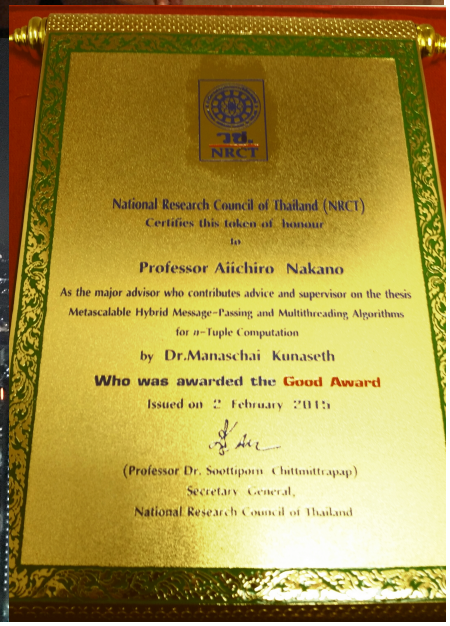
Focus on minimizing runtime (best performance):

- **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	$e = \min(\frac{1}{pf_{CR}}, 1)$
OMP-atomic	$e = \frac{t_{total}}{t_{total} + m\mu c_{atomic}}$
Data privatization	$e = \frac{t_{total}}{t_{total} + c_{reduction} n \log p}$
HTM	$e = \frac{t_{total}}{t_{total} + m(c_{HTM\_overhead} + \mu c_{HTM\_update})}$

M. Kunaseth *et al.*, *PDSEC'13* Best Paper

IEEE PDSEC Best Paper & Beyond



It All Started as a CSCI596 Final Project