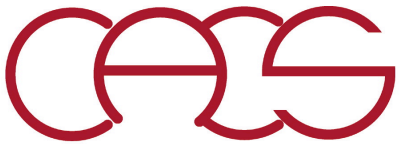
Hybrid MPI+OpenMP Parallel MD

Aiichiro Nakano

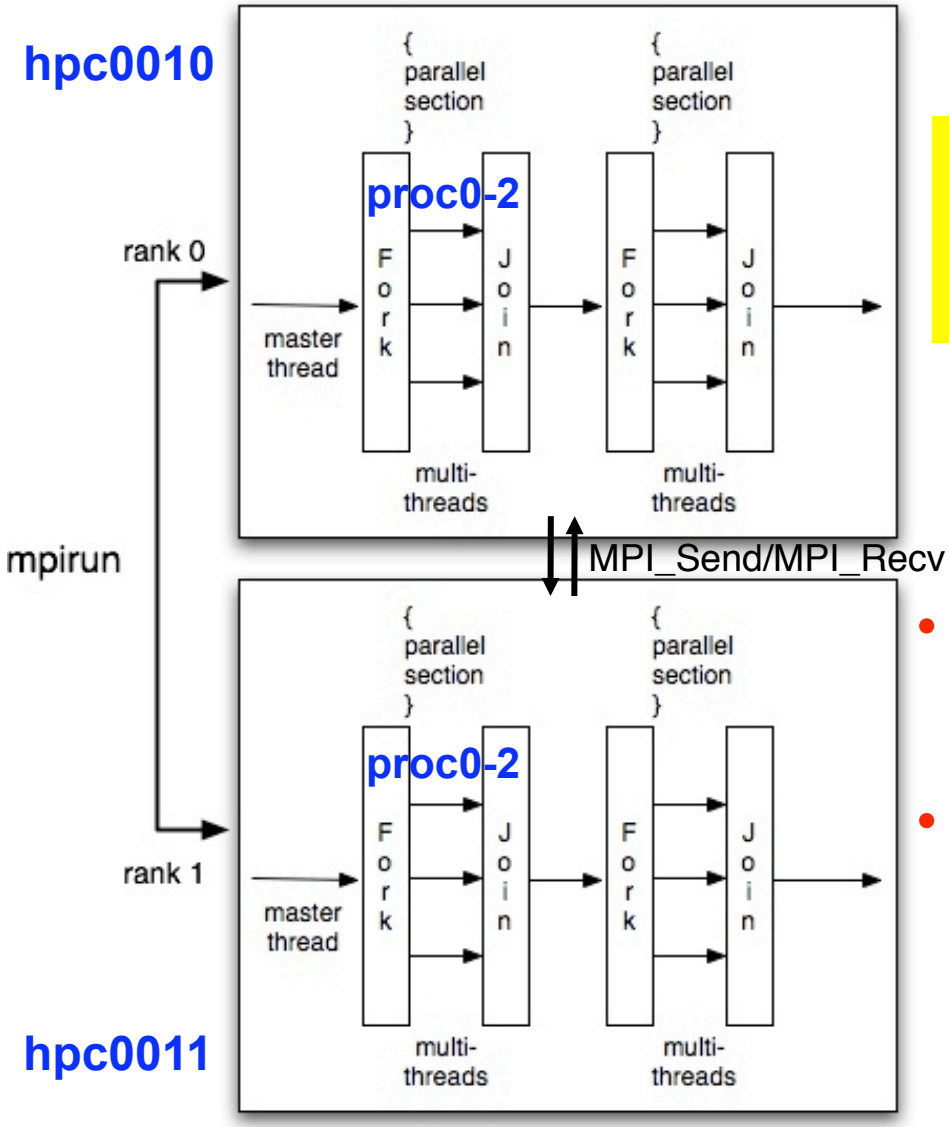
Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Chemical Engineering & Materials Science
University of Southern California

Email: anakano@usc.edu



Hybrid MPI+OpenMP Programming

Each MPI process spawns multiple OpenMP threads



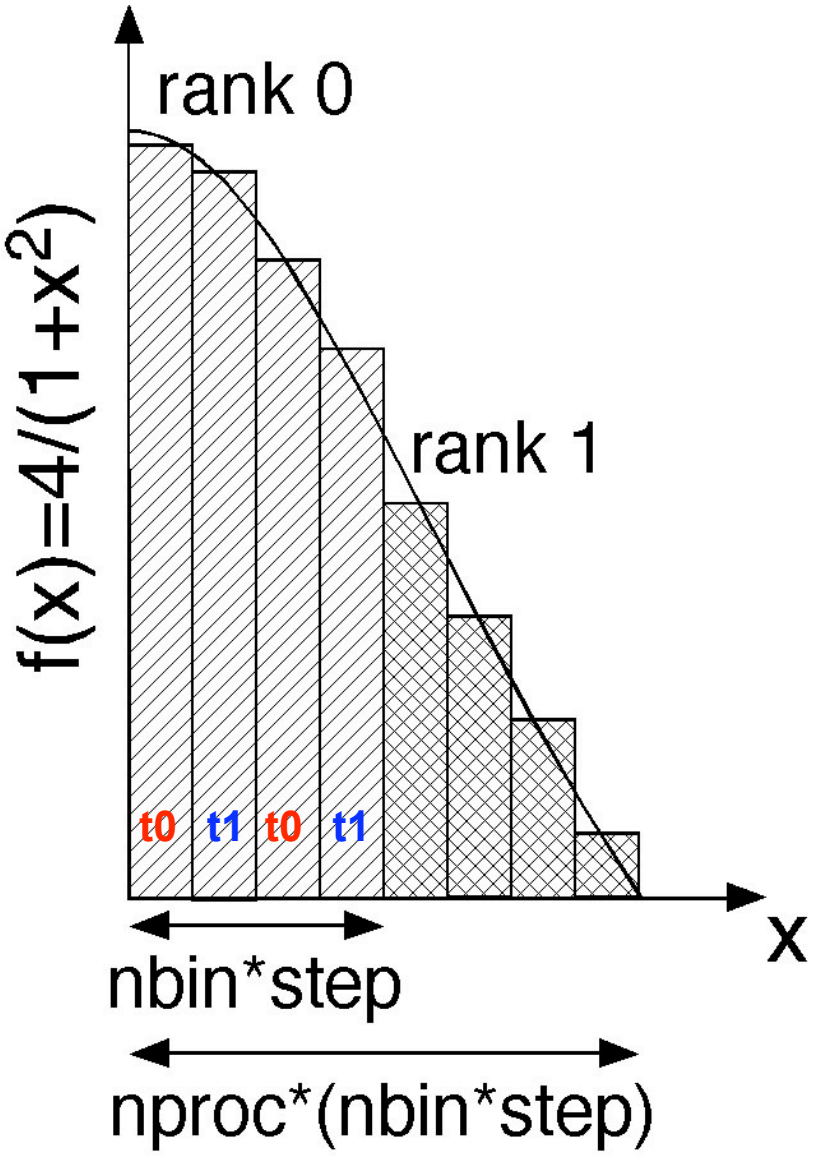

```
In a PBS script:
  mpirun –np 2
In the code:
  omp_set_num_threads(3);
```

- 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`**
- **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();}
```

MPI+OpenMP Example: `hpi.c`

- **Compilation on `hpc-login2.usc.edu`**

```
source /usr/usc/mpich/default/mx-intel/setup.csh
mpicc -o hpi hpi.c -openmp
```

- **PBS script**

```
#!/bin/bash
#PBS -l nodes=2:ppn=1,arch=x86_64
#PBS -l walltime=00:00:59
#PBS -o hpi.out
#PBS -j oe
#PBS -N hpi
WORK_HOME=/home/rcf-proj/csci653/your_ID
cd $WORK_HOME
np=$(cat $PBS_NODEFILE | wc -l)
mpirun -np $np -machinefile $PBS_NODEFILE ./hpi
```

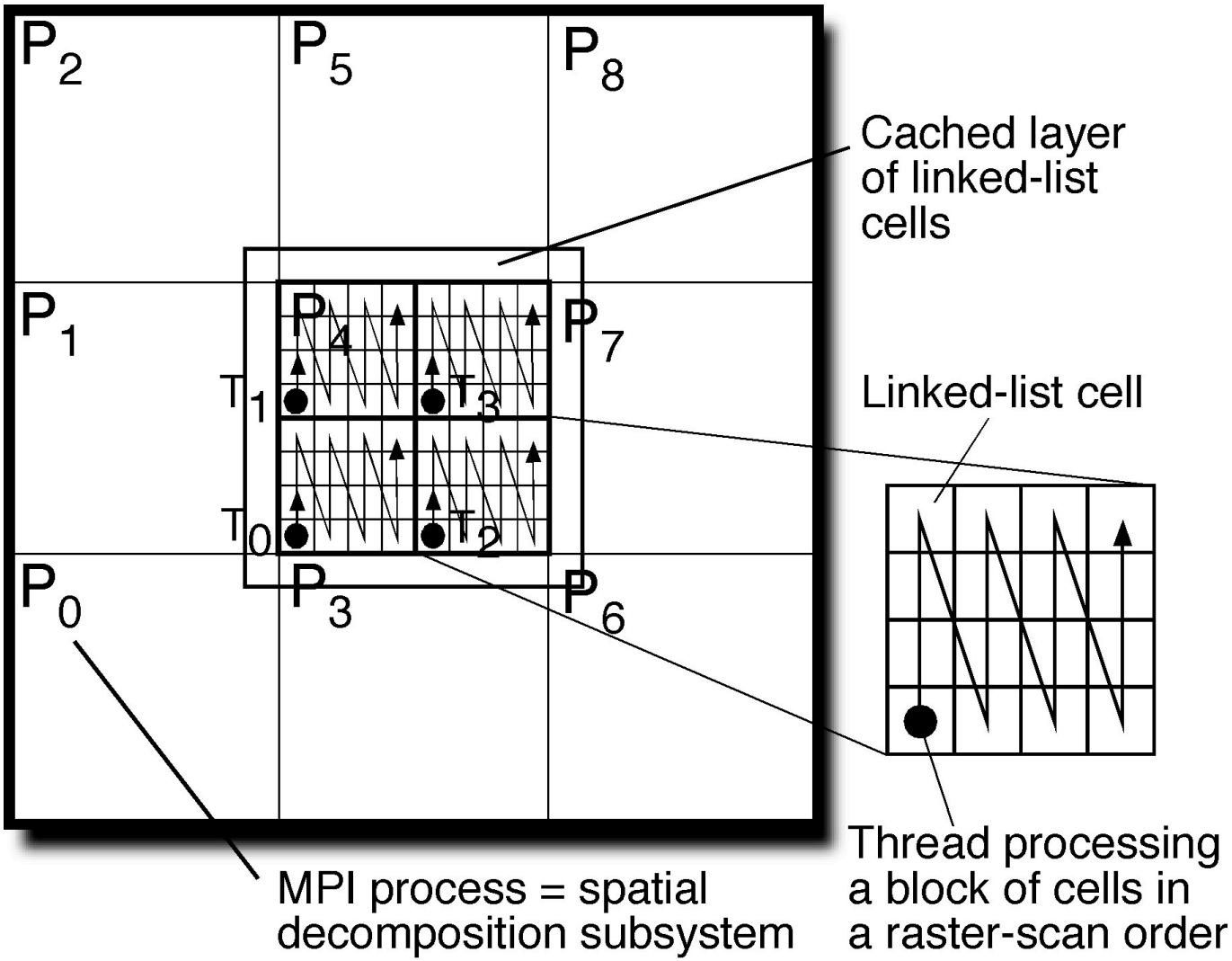
- **Output**

```
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
```

More proper way to submit a hybrid MPI+OpenMP job later

Hybrid MPI+OpenMP Parallel MD

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



Linked-List Cell Block

Variables

- **`vthrd[0|1|2]` = # of OpenMP threads per MPI process in the xlylz 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 xlylz direction that each thread is assigned.**

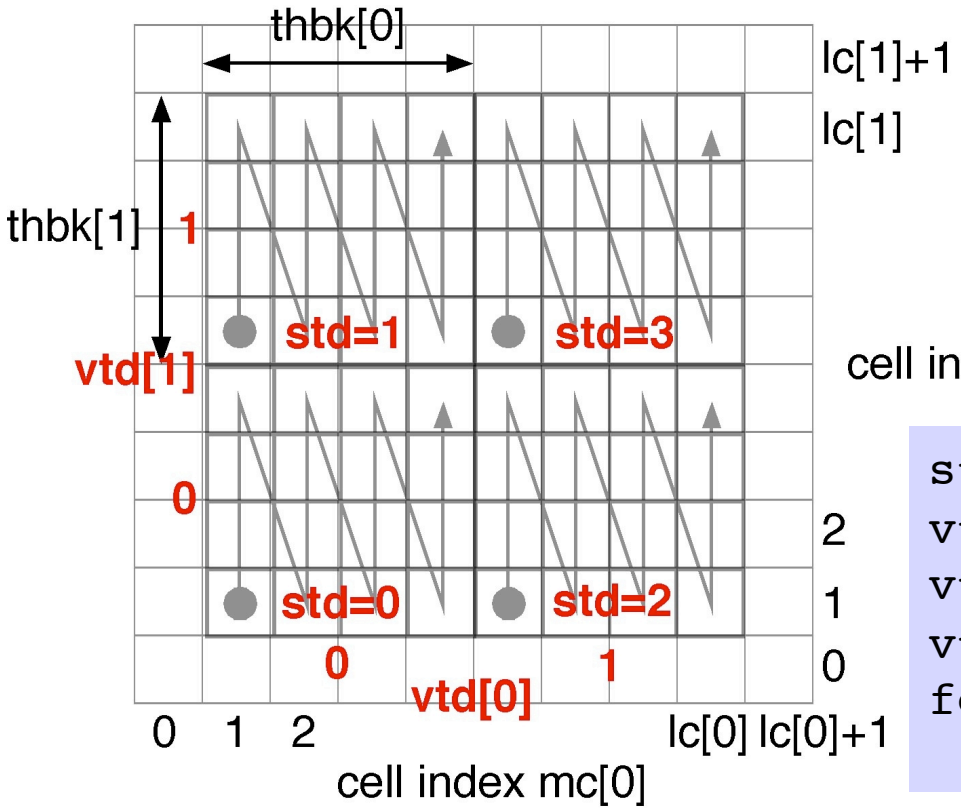
```
In main():
omp_set_num_threads(nthrd);

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 */
for (a=0; a<3; a++)
  thbk[a] = lc[a]/vthrd[a];
/* # of cells = integer multiple of the # of threads */
for (a=0; a<3; a++) {
  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**

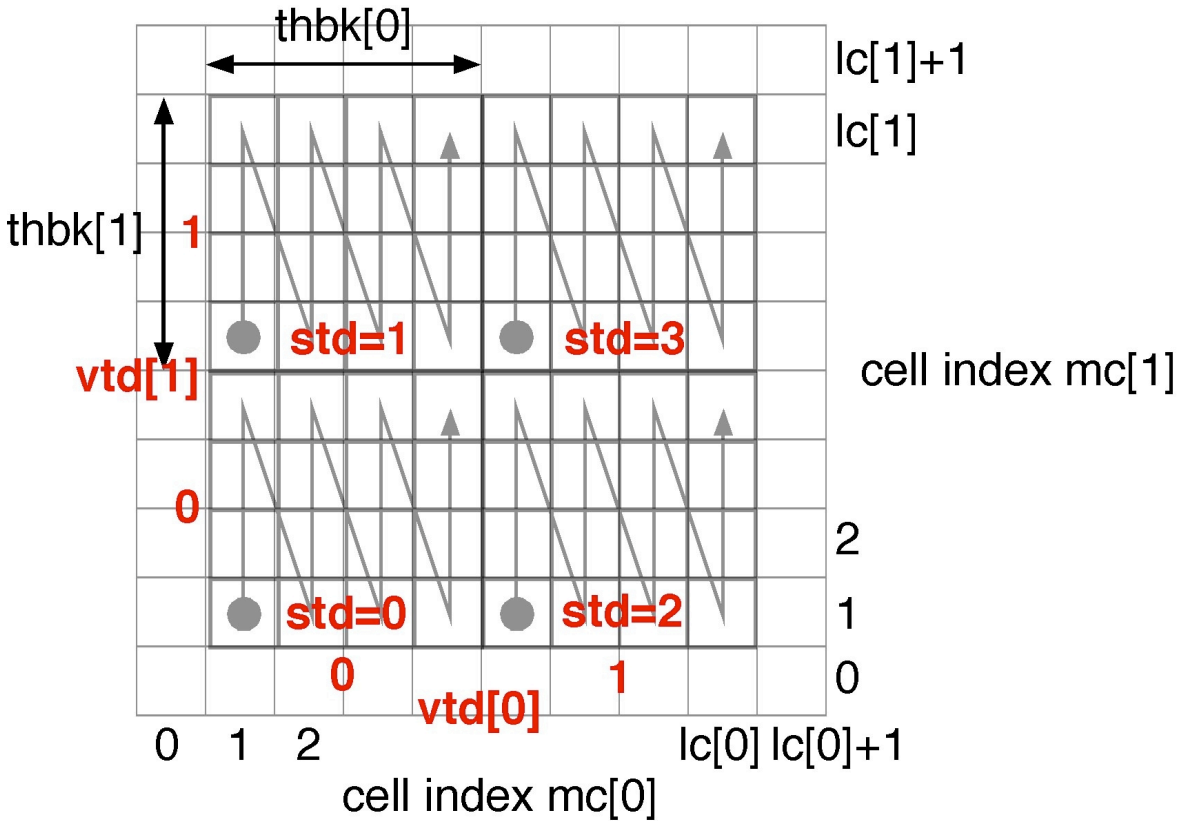
```
#pragma omp parallel private(mc,...)
{
  ...
  for (mc[0]=mofst[0]+1; mc[0]<=mofst[0]+thbk[0]; (mc[0])++)
  for (mc[1]=mofst[1]+1; mc[1]<=mofst[1]+thbk[1]; (mc[1])++)
  for (mc[2]=mofst[2]+1; mc[2]<=mofst[2]+thbk[2]; (mc[2])++)
  {
    Each thread handles thbk[0]×thbk[1]×thbk[2] cells independently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

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, *i.e.*, data privatization

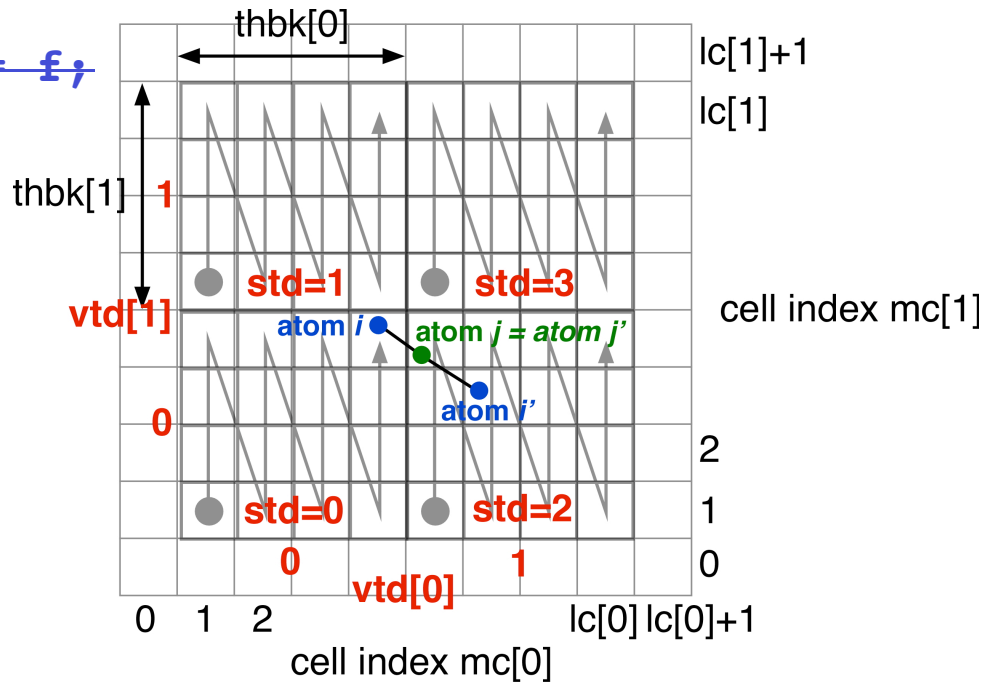


Avoiding Critical Sections (2)

- 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[][]!



Running HMD at HPC (1)

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

```
$ qsub -I -l nodes=2:ppn=4,arch=x86_64 -l walltime=00:29:00
qsub: waiting for job 5280707.hpc-pbs.hpcc.usc.edu to start
qsub: job 5280707.hpc-pbs.usc.edu ready
----------------------------------------
Begin PBS Prologue Mon Sep 30 09:53:29 PDT 2013
Job ID:            5280707.hpc-pbs.hpcc.usc.edu
Username:          anakano
Group:             m-csci
Name:              STDIN
Queue:             quick
Shared Access:     no
All Cores:         no
Nodes:             hpc1939 hpc1940
PVFS:              /scratch (124G), /staging (328T)
TMPDIR:            /tmp/5280707.hpc-pbs.hpcc.usc.edu
End PBS Prologue Mon Sep 30 09:53:37 PDT 2013
----------------------------------------
[anakano@hpc1939 ~]$
```

Running HMD at HPC (2)

2. Type the following sequence of commands. (In this example, hpc/cs653 is my working directory, where the executable hmd is located.)

```
[anakano@hpc1939 ~]$ bash
bash-4.1$ cd hpc/cs653
bash-4.1$ cp $PBS_NODEFILE nodefile
```

3. Edit nodefile, which originally consisted of 8 lines, to delete 6 lines.

(Original nodefile)

```
hpc1939
hpc1939
hpc1939
hpc1939
hpc1940
hpc1940
hpc1940
hpc1940
```

→

(Edited nodefile)

```
hpc1939
hpc1940
```

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

```
bash-4.1$ mpirun -np 2 -machinefile nodefile ./hmd
```

Running HMD at HPC (3)

5. **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-login2 ~]$ ssh hpc1939
[anakano@hpc1939 ~]$ top (then type H)
top - 09:57:03 up 300 days, 20:21,  1 user,  load average: 1.54, 0.55, 1.48
Tasks: 272 total,   5 running, 267 sleeping,   0 stopped,   0 zombie
Cpu(s): 49.7%us,  0.1%sy,  0.0%ni, 50.2%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  12191580k total,  1136412k used, 11055168k free,   186248k buffers
Swap:  1048568k total,    62040k used,   986528k free,   529684k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27801 anakano   20   0  129m  33m  14m R 99.9  0.3   0:32.01 hmd
27807 anakano   20   0  129m  33m  14m R 99.9  0.3   0:31.94 hmd
27805 anakano   20   0  129m  33m  14m R 99.3  0.3   0:31.92 hmd
27806 anakano   20   0  129m  33m  14m R 99.3  0.3   0:31.47 hmd
...
```

Running HMD at HPC (4)

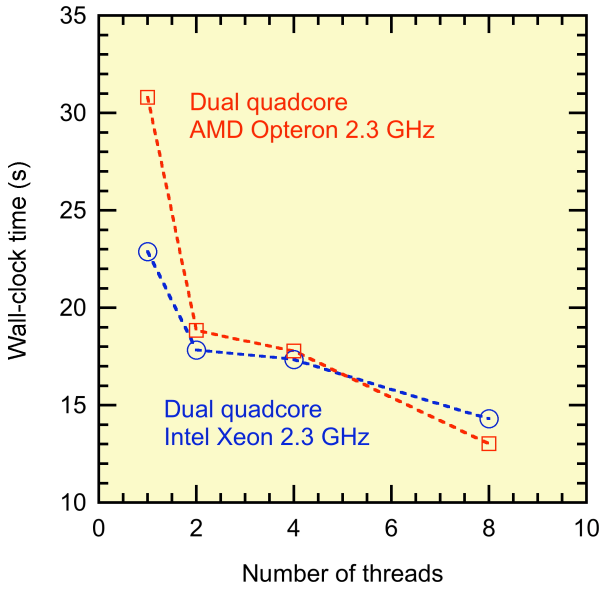
- **Instead of the interactive PBS job, you can submit a batch job using the following script (the uniq command will eliminate duplicated lines in $PBS_NODEFILE and retain only one line per node).**

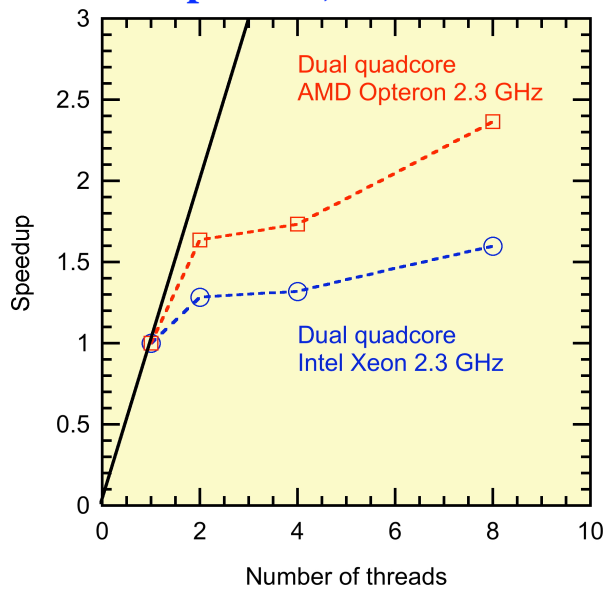
```
#!/bin/bash
#PBS -l nodes=2:ppn=4,arch=x86_64
#PBS -l walltime=00:00:59
#PBS -o hmd.out
#PBS -j oe
#PBS -N hmd
WORK_HOME=/home/rcf-proj/csci653/your_ID
cd $WORK_HOME
cat $PBS_NODEFILE | uniq > nodefile
np=$(cat nodefile | wc -l)
mpirun -np $np -machinefile nodefile ./hmd
```

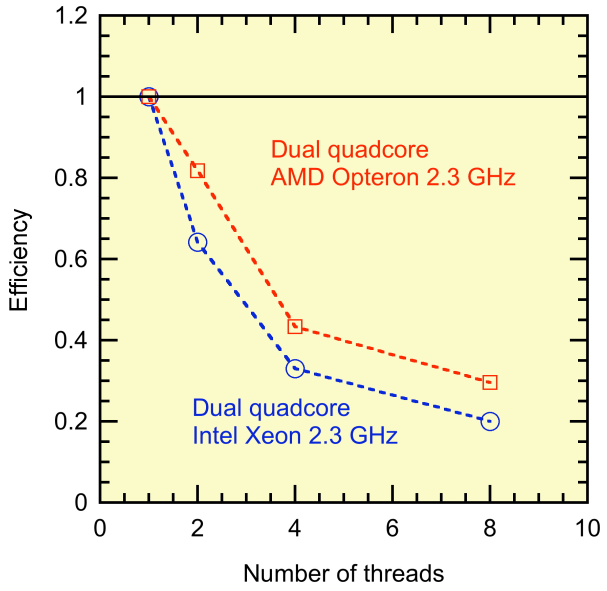
This way is recommended!

Strong Scalability of Hybrid MD

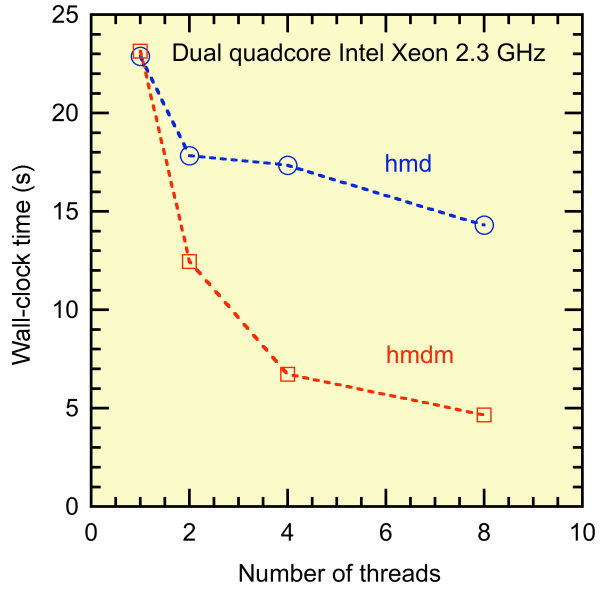
1 MPI process; 1-8 threads

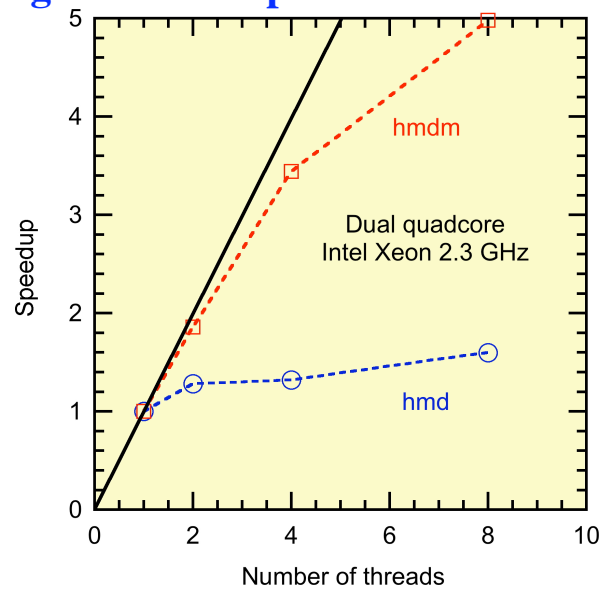


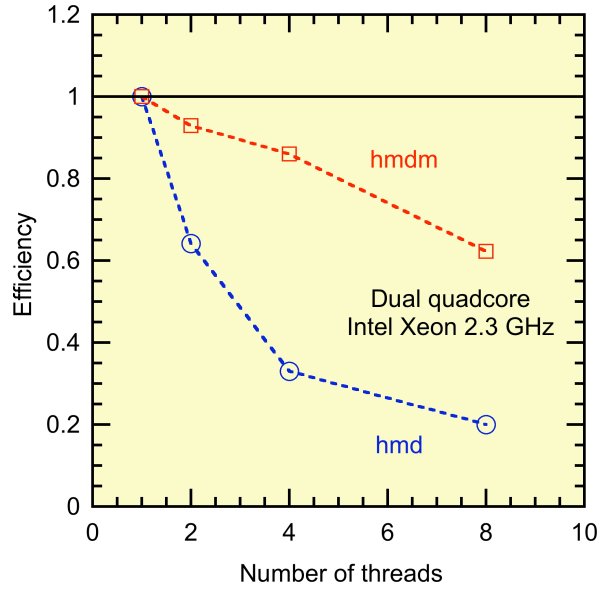




Algorithmic improvement on Xeon



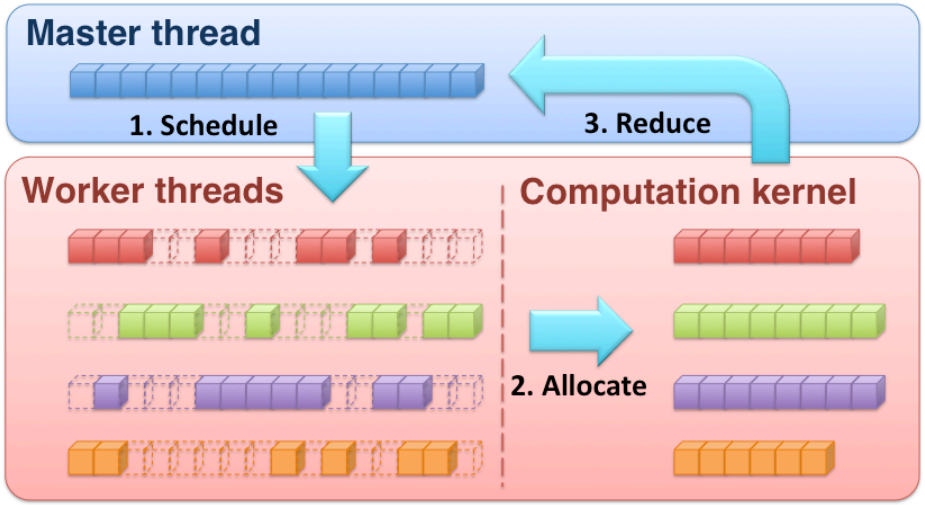


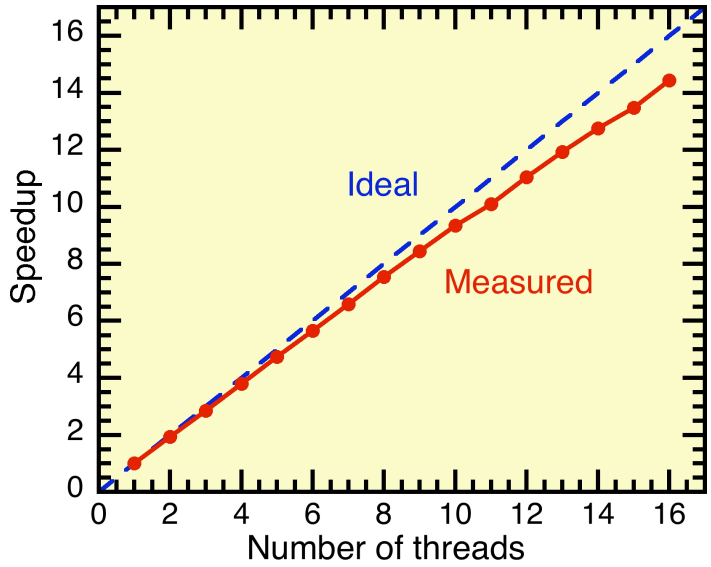


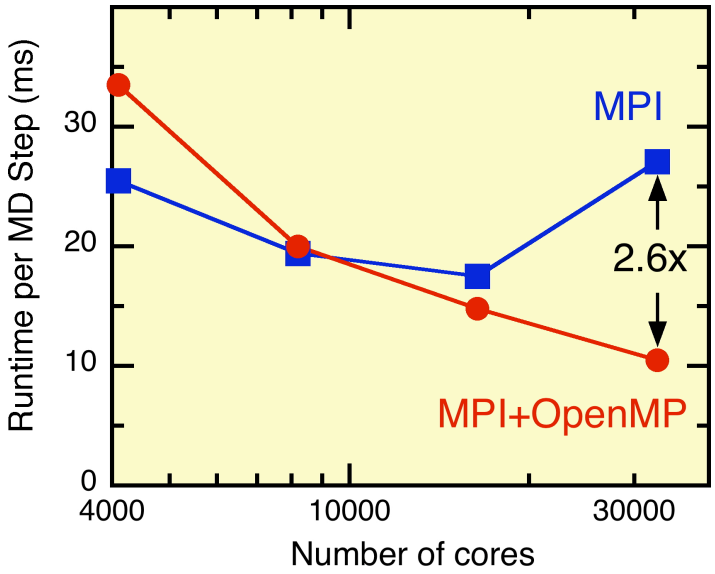
Spatially Compact Thread Scheduling

Concurrency-control mechanism: Data privatization

- **Reduced memory:** # of atoms / # 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**
- **2.6× speedup over MPI by hybrid MPI+OpenMP on 32,768 IBM BlueGene/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**

CCM performance varies:

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

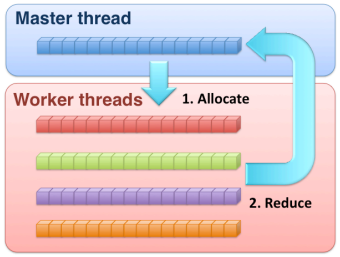
HTM/critical section

```
#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

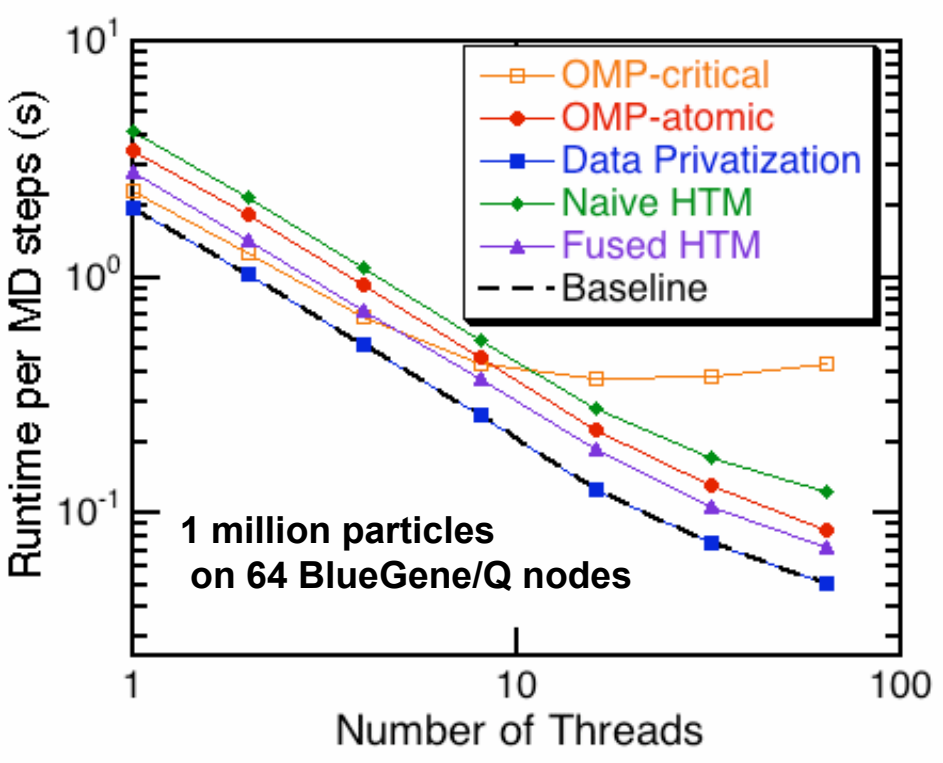
Transactional memory (TM): An opportunistic CCM

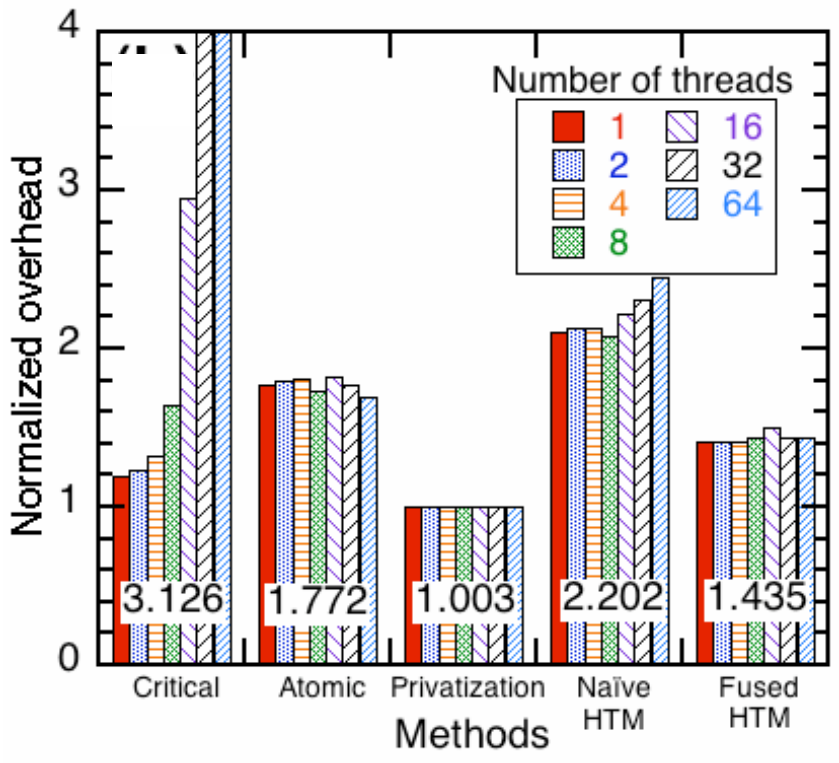
- Avoids memory conflicts by monitoring a set of speculative operations (*i.e.* transaction)
- If two or more transactions write to the same memory address, transaction(s) will be restarted—a process called rollback
- If no conflict detected in the end of a transaction, operations within the transaction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

Hardware TM on IBM BlueGene/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



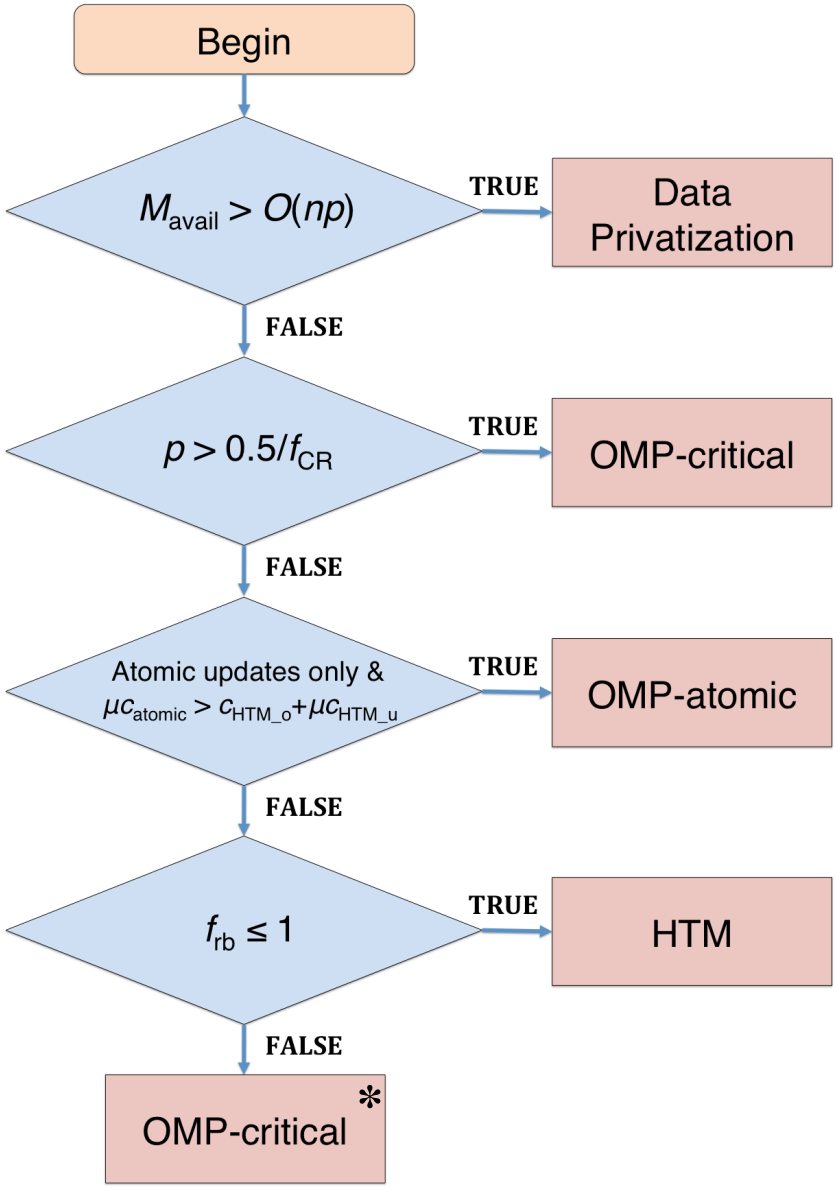


Developed a fundamental understanding of CCMs:

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

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

Threading Guideline for Scientific Programs



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

Focus on minimizing runtime (best performance):

- **Have enough memory → data privatization**
- **Conflict region is small → OMP-critical**
- **Small amount of updates → OMP-atomic**
- **Conflict rates 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	$e = \frac{t_{total}}{t_{total} + m(c_{HTM\_overhead} + \mu c_{HTM\_update})}$