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

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

Hybrid MPI+OpenMP Programming

Each MPI process spawns multiple OpenMP threads

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

In a Slurm script:
mpirun -n 2
In the code:
omp_set_num_threads(3);
MPI+OpenMP Calculation of $\pi$

- **Spatial decomposition:** Each MPI process integrates over a range of width $1/nproc$, as a discrete sum of $nbin$ bins each of width $\text{step}$

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

\[
\pi = \int_{0}^{1} \frac{4}{1 + x^2} \, dx \equiv \Delta \sum_{i=0}^{N-1} \frac{4}{1 + x_i^2}
\]
MPI+OpenMP Calculation of $\pi$: hpi.c

```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**

**Local variables: Different values needed for different threads**

**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**
  ```bash
  #!/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**
  ```bash
  [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 → thread
```

See the outermost loop over `mc[3]` in function `compute_accel()` in program `pmd.c`
Linked-List Cell Block

Variables

- \(vthrd[0|1|2] = \# \text{ of OpenMP threads per MPI process in the x|y|z direction.}\)
- \(nthrd = \# \text{ of OpenMP threads} = vthrd[0] \times vthrd[1] \times vthrd[2].\)
- \(thbk[3]: thbk[0|1|2] \text{ is the } \# \text{ of linked-list cells in the x|y|z direction that each thread is assigned.}\)

In main():

```c
omp_set_num_threads(nthrd);
```

In init_params():

```c
/* 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 */
}
```

In hmd.h:

```c
int vthrd[3]={2,2,1}, nthrd=4;
int thbk[3];
```
Variables

- \( \text{std} = \) scalar thread index.
- \( \text{vtd}[3]: \text{vtd}[0|1|2] \) is the x|y|z element of vector thread index.
- \( \text{mofst}[3]: \text{mofst}[0|1|2] \) is the x|y|z offset cell index of cell-block.

\[
\begin{align*}
\text{std} &= \text{omp\_get\_thread\_num}(); \\
\text{vtd}[0] &= \text{std}/(\text{vthrd}[1]*\text{vthrd}[2]); \\
\text{vtd}[1] &= (\text{std}/\text{vthrd}[2])\%\text{vthrd}[1]; \\
\text{vtd}[2] &= \text{std}\%\text{vthrd}[2]; \\
\text{for} \ (a=0; \ a<3; \ a++) \\
& \quad \text{mofst}[a] = \text{vtd}[a]*\text{thbk}[a];
\end{align*}
\]

Call \texttt{omp\_get\_thread\_num()} within an OpenMP parallel block.
• 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

```c
#pragma omp parallel private(mc,...)
{
  ...
  for (mc[0]=mofst[0]+1; mc[0]<=mofst[0]+thbk[0]; (mc[0])++)
      }
  ...
}
```
Avoiding Critical Sections (1)

- Remove the critical section

```c
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

- Interthread reduction after join
  ```
  for (i=0; i<nthrd; i++) lpe += lpe_td[i];
  ```
define shared;

... if used here

```c
#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.

```bash
#!/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
```

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

```bash
mpicc -O -o hmd hmd.c -lm -fopenmp
```
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 254685
salloc: job 254685 queued and waiting for resources
salloc: job 254685 has been allocated resources
salloc: Granted job allocation 254685
salloc: Waiting for resource configuration
salloc: Nodes d06-[25-26] are ready for job
[anakano@d06-25 cs596]$

You are logged in to one of the allocated nodes

<table>
<thead>
<tr>
<th>Partition</th>
<th>Nodes</th>
<th>CPUs</th>
<th>Memory (GB)</th>
<th>CPU type</th>
<th>GPUs</th>
<th>Nodelist</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>84</td>
<td>24</td>
<td>94</td>
<td>xeon-4116</td>
<td>None</td>
<td>d05-[03-15,26-42],d06-[15-29],d11-[09-47]</td>
</tr>
</tbody>
</table>

https://carc.usc.edu/user-information/user-guides/high-performance-computing/discovery/discovery-resources
2. Submit a two-process MPI program (named hmd); each of the MPI process will spawn 4 OpenMP threads.

   [anakano@d06-25 cs596]$ mpirun -bind-to none -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 (type ‘q’ to stop).

   [anakano@discovery ~]$ ssh d06-26
   [anakano@d06-26 ~]$ 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
   ...
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
...  
%Cpu23 :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
```
5. Without `-bind-to none` option, hmd process (and all spawned threads by it) is bound to one core.

[anakano@d06-25 cs596]$ mpirun -n 2 ./hmd

[anakano@d06-26 ~]$ top

...
How Hybrid MPI+OpenMP MD Runs

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);
```

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

On `d05-29`:
```
mpirun -bind-to none -n 2 ./hmd
```

On `d05-29` & `d05-30`:
```
top (then type H and 1)
```
Validation of Hybrid MD

2 MPI process; 4 threads

In hmd.h:

\[ \text{vproc} = \{1,1,2\}, \text{nproc} = 2; \]
\[ \text{vthrd} = \{2,2,1\}, \text{nthrd} = 4; \]

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Temperature</th>
<th>Potential energy</th>
<th>Total energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.050000</td>
<td>0.877345</td>
<td>-5.137153</td>
<td>-3.821136</td>
</tr>
<tr>
<td>0.100000</td>
<td>0.462056</td>
<td>-4.513097</td>
<td>-3.820013</td>
</tr>
<tr>
<td>0.150000</td>
<td>0.510836</td>
<td>-4.587287</td>
<td>-3.821033</td>
</tr>
<tr>
<td>0.200000</td>
<td>0.527457</td>
<td>-4.611958</td>
<td>-3.820772</td>
</tr>
<tr>
<td>0.250000</td>
<td>0.518668</td>
<td>-4.598798</td>
<td>-3.820796</td>
</tr>
<tr>
<td>0.300000</td>
<td>0.529023</td>
<td>-4.614343</td>
<td>-3.820808</td>
</tr>
<tr>
<td>0.350000</td>
<td>0.532890</td>
<td>-4.620133</td>
<td>-3.820798</td>
</tr>
<tr>
<td>0.400000</td>
<td>0.536070</td>
<td>-4.624899</td>
<td>-3.820794</td>
</tr>
<tr>
<td>0.450000</td>
<td>0.539725</td>
<td>-4.630387</td>
<td>-3.820799</td>
</tr>
<tr>
<td>0.500000</td>
<td>0.538481</td>
<td>-4.628514</td>
<td>-3.820792</td>
</tr>
</tbody>
</table>

See the lecture on “order-invariant real-number summation”
Strong Scalability of Hybrid MD

1 MPI process; 1-8 threads

In hmd.h:

```c
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}

\[ N = 4 \times 24^3 = 55296 \text{ atoms} \]

pmd.in

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

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

\[ P: \text{Number of cores} \]

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

\[ E_P = \frac{S_P}{P} \]
Improved Strong Scalability of Hybrid MD

1 MPI process; 1-8 threads

In hmd.h:

```
initu = {24,24,24};
N = 4 \times 24^3 = 55296 \text{ atoms}
```

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

```
mpirun ... -n 1 ... ./hmd
```

```
2 4 8
```

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

```
SP = \frac{T(N,1)}{T(N,P)}
```

```
EP = \frac{SP}{P}
```

**Figure:**

- Graph showing Wall-clock time (s) vs. Number of threads.
- Graph showing Speedup vs. Number of threads.
- Graph showing Efficiency vs. Number of threads.

**Equations:**

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

\( E_P = \frac{S_P}{P} \)

**Legend:**

- Dual quadcore Intel Xeon 2.3 GHz
- Dual quadcore Intel Xeon 2.3 GHz
- Dual quadcore Intel Xeon 2.3 GHz
- Dual quadcore Intel Xeon 2.3 GHz
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`:

```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
        ...
    }
    ...
}
```

![Graph showing efficiency vs. number of threads for hmdm and Dual quadcore Intel Xeon 2.3 GHz]
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 $r_c$


• 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**

  ```c
  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**

  ```c
  #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

Graph showing the efficiency of parallel reduction in different scenarios:
- hmdred
- omp parallel reduction
- hmd
- lpe_td[nthrd]
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()`

```c
int main(int argc, char **argv) {
    ...
    vthrd[0] = atoi(argv[1]);
    vthrd[1] = atoi(argv[2]);
    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

```c
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

![Graph showing the relationship between wall clock time (s) and number of threads for atomic and original operations. The atomic operation has a higher wall clock time compared to the original, especially as the number of threads increases.](image-url)
Spatially Compact Thread Scheduling

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

- Reduced memory: \(\Theta(nq) \rightarrow \Theta(n+n^{2/3}q^{1/3})\)
- Strong scaling parallel efficiency 0.9 on quad quad-core AMD Opteron
- 2.6\(\times\) 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)

### CCM performance varies:

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

### Examples:

#### HTM/critical section

```c
#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

```c
#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

![Data privatization diagram](image-url)

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

1 million particles on 64 Blue Gene/Q nodes with 16 cores per node

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)

**Concurrence control mechanism**  **Parallel efficiency**

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

M. Kunaseth et al., PDSEC’13 Best Paper
IEEE PDSEC Best Paper Award

Performance Characteristics of Hardware Transactional Memory for Molecular Dynamics Application on BlueGene/Q
Manaschai Kunaseth, Rajiv Kalia, Ailchiro Nakano, Priya Vashishta, David Richards, James Gosli

The 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-13), held in Boston, Massachusetts, USA, May 24, 2013.
It All Started as a CSCI596 Final Project