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 PBS script:
```bash
mpirun -np 2
```

In the code:
```c
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 $\pi$

- **Spatial decomposition:** Each MPI process integrates over a range of width $\frac{1}{n\text{proc}}$, as a discrete sum of $n\text{bin}$ bins each of width $\text{step}$

- **Interleaving:** Within each MPI process, $n\text{threads}$ 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**

Used in both serial & parallel sections.
MPI+OpenMP Example: hpi.c

- **Compilation on** hpc-login3.usc.edu
  
  ```
  source /usr/usc/openmpi/default/setup.csh
  mpicc -o hpi hpi.c -fopenmp
  ```

- **PBS script**

  ```bash
  #!/bin/bash
  #PBS -l nodes=2:ppn=2
  #PBS -l walltime=00:00:59
  #PBS -o hpi.out
  #PBS -j oe
  #PBS -N hpi
  WORK_HOME=/auto/rcf-proj/an2/anakano
  cd $WORK_HOME
  cat $PBS_NODEFILE | uniq > nodefile
  np=$(cat nodefile | wc -l)
  mpirun -np $np -machinefile 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
  ```

  ![The uniq command eliminates duplicate lines](image)

  $PBS_NODEFILE  nodefile

  hpc1006  hpc1006  hpc1007  hpc1007

  uniq
Hybrid MPI+OpenMP Parallel MD

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

- **MPI process = spatial decomposition subsystem**
- **Cached layer of linked-list cells**
- **Linked-list cell**
- **Thread processing a block of cells in a raster-scan order**
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 = \(vthrd[0] \times vthrd[1] \times vthrd[2]\).
- \(thbk[3]\): \(thbk[0|1|2]\) is the \# of linked-list cells in the x|y|z direction that each thread is assigned.

In main():

```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];
```
OpenMP Threads for Cell Blocks

Variables

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

\begin{verbatim}
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];
\end{verbatim}

Call \texttt{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

```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:

```c
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
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
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,walltime=00:29:00
qsub: waiting for job 24435338.hpc-pbs1.hpcc.usc.edu to start
qsub: job 24435338.hpc-pbs1.hpcc.usc.edu ready

----------------------------------------
Begin PBS Prologue Thu Oct  5 08:19:20 PDT 2017
Job ID: 24435338.hpc-pbs1.hpcc.usc.edu
Username: anakano
Accountname: default
Group: m-csci
Project: default
Name: STDIN
Queue: quick
Shared Access: no
All Cores: no
Has MIC: no
Is hsda: false
Nodes: hpc0971 hpc0972
Scratch is: /scratch
TMPDIR: /tmp/24435338.hpc-pbs1.hpcc.usc.edu
End PBS Prologue Thu Oct  5 08:19:49 PDT 2017

----------------------------------------
[anakano@hpc0971 ~]$ You are logged in to one of the allocated nodes
2. Type the following sequence of commands. (hpc/cs596 is a symbolic link to my working directory, where the executable hmd is located.)

[anakano@hpc0971 ~]$ bash
bash-4.2$ cd hpc/cs596
bash-4.2$ cp $PBS_NODEFILE nodefile

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

(Original nodefile)   (Edited nodefile)

```
  hpc0971  hpc0971
  hpc0971  hpc0972
  hpc0971
  hpc0971
  hpc0972
  hpc0972
  hpc0972
  hpc0972
```

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

```
bash-4.2$ mpirun --bind-to none -np 2 -machinefile nodefile ./hmd
```

Note: Without the “--bind-to none” option, OpenMPI would bind each process to a single core, hence all OpenMP threads would run on the same core
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-login3 ~]$ ssh hpc0971
[anakano@hpc2361 ~]$ top (then type H)

Top - 08:22:54 up 14 days, 2:16, 1 user, load average: 1.64, 1.05, 0.93
Threads: 471 total, 5 running, 466 sleeping, 0 stopped, 0 zombie
%Cpu(s): 25.4 us, 1.0 sy, 0.0 ni, 73.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 24513544 total, 21659772 free, 853952 used, 1999820 buff/cache
KiB Swap: 8388604 total, 8388604 free, 0 used. 23495648 avail Mem

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>20051</td>
<td>anakano</td>
<td>20</td>
<td>0</td>
<td>261780</td>
<td>23088</td>
<td>3064</td>
<td>R</td>
<td>99.9</td>
<td>0.1</td>
<td>0:38.46</td>
<td>hmd</td>
</tr>
<tr>
<td>20053</td>
<td>anakano</td>
<td>20</td>
<td>0</td>
<td>261780</td>
<td>23088</td>
<td>3064</td>
<td>R</td>
<td>72.2</td>
<td>0.1</td>
<td>0:27.61</td>
<td>hmd</td>
</tr>
<tr>
<td>20054</td>
<td>anakano</td>
<td>20</td>
<td>0</td>
<td>261780</td>
<td>23088</td>
<td>3064</td>
<td>R</td>
<td>72.2</td>
<td>0.1</td>
<td>0:27.61</td>
<td>hmd</td>
</tr>
<tr>
<td>20055</td>
<td>anakano</td>
<td>20</td>
<td>0</td>
<td>261780</td>
<td>23088</td>
<td>3064</td>
<td>R</td>
<td>71.9</td>
<td>0.1</td>
<td>0:27.61</td>
<td>hmd</td>
</tr>
<tr>
<td>19633</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>1604944</td>
<td>52988</td>
<td>1516</td>
<td>S</td>
<td>0.3</td>
<td>0.2</td>
<td>0:00.03</td>
<td>TimerQ</td>
</tr>
<tr>
<td>20046</td>
<td>anakano</td>
<td>20</td>
<td>0</td>
<td>156192</td>
<td>2388</td>
<td>1524</td>
<td>R</td>
<td>0.3</td>
<td>0.0</td>
<td>0:00.24</td>
<td>top</td>
</tr>
</tbody>
</table>
...
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).

```bash
#!/bin/bash
#PBS -l nodes=2:ppn=4
#PBS -l walltime=00:00:59
#PBS -o hmd.out
#PBS -j oe
#PBS -N hmd
WORK_HOME=/home/rcf-proj/an2/anakano
cd $WORK_HOME
cat $PBS_NODEFILE | uniq > nodefile
np=$(cat nodefile | wc -l)
mpirun --bind-to none -np $np -machinefile nodefile ./hmd
```

This way is recommended!
Effect of Core Binding

1. `mpirun --bind-to none -np 2 -machinefile nodefile ./hmd`

   [anakano@hpc0971 ~]$ top (then type H)
   top - 08:22:54 up 14 days,  2:16,  1 user,  load average: 1.64, 1.05, 0.93
   Threads: 471 total,   5 running, 466 sleeping,  0 stopped,  0 zombie
   %Cpu(s): 25.4 us,  1.0 sy,  0.0 ni, 73.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
   KiB Mem : 24513544 total, 21659772 free,  853952 used, 1999820 buff/cache
   KiB Swap:  8388604 total,  8388604 free,      0 used. 23495648 avail Mem

   PID USER      PR  NI     VIRT    RES    SHR  S  %CPU %MEM     TIME+ COMMAND
   20051 anakano 20   0  261780  23088   3064 R 99.9  0.1   0:38.46 hmd
   20053 anakano 20   0  261780  23088   3064 R 72.2  0.1   0:27.61 hmd
   20054 anakano 20   0  261780  23088   3064 R 72.2  0.1   0:27.61 hmd
   20055 anakano 20   0  261780  23088   3064 R 71.9  0.1   0:27.61 hmd
   19633 root      20   0 1604944  52988   1516 S  0.3  0.2   0:00.03 TimerQ
   ...

2. `mpirun -np 2 -machinefile nodefile ./hmd`

   [anakano@hpc0971 ~]$ top (then type H)
   top - 08:32:40 up 13 days,  8:04,  1 user,  load average: 0.89, 0.58, 0.81
   Threads: 357 total,   5 running, 352 sleeping,  0 stopped,  0 zombie
   %Cpu(s):  8.2 us,  0.1 sy,  0.0 ni, 91.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
   KiB Mem : 24513548 total, 21898024 free,  714840 used, 1900684 buff/cache
   KiB Swap:  8388604 total,  8388308 free,  296 used. 23650348 avail Mem

   PID USER      PR  NI     VIRT    RES    SHR  S  %CPU %MEM     TIME+ COMMAND
   28950 anakano 20   0  261912  25100  3052 R 28.9  0.1   0:04.28 hmd
   28953 anakano 20   0  261912  25100  3052 R 23.9  0.1   0:03.51 hmd
   28954 anakano 20   0  261912  25100  3052 R 23.6  0.1   0:03.48 hmd
   28952 anakano 20   0  261912  25100  3052 R 23.3  0.1   0:03.49 hmd
   2403 root      20   0  553152 18504  5784 S  0.3  0.1   1:09.22 tuned
   ...

Q. What will this do?

   mpirun -np 2 -machinefile $PBS_NODEFILE ./hmd
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

```
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
```

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

1 MPI process; 1-8 threads

In hmd.h:

\[
\begin{align*}
\text{vproc} & = \{1,1,1\}, \ nproc = 1; \\
\text{vthrd} & = \{1,1,1\}, \ nthrd = 1; \\
\end{align*}
\]

2 1 1 2
2 2 1 4
2 2 2 8

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

\[N = 4 \times 24^3\]

\[= 55296 \text{ atoms}\]

\[\text{pmd.in}\]

\[
\begin{align*}
24 & 24 & 24 \ \text{InitUcell[3]} \\
0.8 & & \text{Density} \\
1.0 & & \text{InitTemp} \\
0.005 & & \text{DeltaT} \\
100 & & \text{StepLimit} \\
10 & & \text{StepAvg}
\end{align*}
\]

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

\[E_P = \frac{S_P}{P}\]

\[P: \text{Number of cores}\]
Improved Strong Scalability of Hybrid MD

1 MPI process; 1-8 threads

In hmd.h:

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

\[
\begin{array}{c@{\quad}c@{\quad}c@{\quad}c}
2 & 1 & 1 & 2 \\
2 & 2 & 1 & 4 \\
2 & 2 & 2 & 8 \\
\end{array}
\]

#PBS -l nodes=1:ppn=8

mpirun ... --np 1 ... ./hmd1

\[
2 \quad 4 \quad 8
\]

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

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

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

\[
E_P = \frac{S_P}{P}
\]

\[P: \text{Number of cores}\]
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
Spatially Compact Thread Scheduling

Concurrency-control mechanism: Data privatization

- 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× speedup over MPI by hybrid MPI+OpenMP on 32,768 IBM Blue Gene/P cores

M. Kunaseth et al., PDPTA’11; J. Supercomput. (’13)
Concurrence-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

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

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 $\rightarrow$ data privatization
- Conflict region is small $\rightarrow$ OMP-critical
- Small amount of updates $\rightarrow$ OMP-atomic
- Conflict rate is low $\rightarrow$ HTM
- Other $\rightarrow$ OMP-critical* (poor performance)

\[
\begin{array}{l}
\text{Concurrency control mechanism} \\
\text{Parallel efficiency}
\end{array}
\]

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

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 Kunaseht, Rajiv Kalia, Aichiro Nakano, Priya Vashishtha, David Richards, James Glosli

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