High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 11

False sharing

- threads that share an array may use different parts of the array; similarly, threads may use their own private variables
- logically, these memory locations are not shared
- however, if these memory locations used by different threads are on the *same* cache line, then sharing does physically occur
- this is called *false sharing* and can hurt performance
- cache lines are 64 bytes on x86 processors (at all levels), and cache lines are read/written from/to main memory as a unit

False sharing example: `false_sharing.c`

Generating a sequence of random numbers for each thread:

```
int *data = (int *) malloc(LEN*sizeof(int));
__declspec(align(64)) int seeds[16];

#pragma omp parallel num_threads(16)
{
  int threadid = omp_get_thread_num();
  #pragma omp for
  for (i=0; i<LEN; i++)
    data[i] = rand_r(&seeds[threadid]);
}
```

- The array `seeds` is on a single cache line. When one thread writes to the array, the entire cache line is invalidated
- Note: this is a bad way to generate random numbers in parallel (sequences may overlap)

False sharing example: `false_sharing2.c`

Generating a sequence of random numbers for each thread:

```
int *data = (int *) malloc(LEN*sizeof(int));
__declspec(align(64)) int seeds[16*16];

#pragma omp parallel num_threads(16)
{
  int threadid = omp_get_thread_num();
  #pragma omp for
  for (i=0; i<LEN; i++)
    data[i] = rand_r(&seeds[16*threadid]);
}
```

Timings

```
joker:~$ icc -qopenmp false_sharing.c
joker:~$ ./a.out
time: 8.207102
```

Timings

```
joker:~$ icc -qopenmp false_sharing.c
joker:~$ ./a.out
time: 8.207102
```

```
joker:~$ icc -qopenmp false_sharing2.c
joker:~$ ./a.out
time: 0.503792
```

16 times faster! Why do we get a factor of 16?

Timings

```
joker:~$ icc -qopenmp false_sharing.c
joker:~$ ./a.out
time: 8.207102
```

```
joker:~$ icc -qopenmp false_sharing2.c
joker:~$ ./a.out
time: 0.503792
```

16 times faster! Why do we get a factor of 16?

10 times faster if we use 10 threads.

Avoiding false sharing

Assure that threads write to different cache lines (but don't need to worry if only reading data)

- use padding of memory locations to cache line boundaries
- replicate data, e.g., by using `private` (but this can deplete cache if many threads)

Brownian dynamics with hydrodynamic interactions

- Small particles in a fluid interact hydrodynamically
- Instead of Brownian forces on each particle that are independent, the Brownian forces are *correlated*
- The correlation matrix for hydrodynamic interactions is called the Rotne-Prager-Yamakawa (RPY) mobility matrix, M
- To generate a *correlated* Brownian displacement vector, compute the Cholesky factorization $M = LL^T$ and then compute $y = Lz$, where z is a vector with a standard normal distribution
- To simulate hydrodynamic interactions, use this correlated vector y instead of the uncorrelated vector z

RPY mobility matrix

- For n particles, this is a $3n \times 3n$ matrix
- Example for 2 particles (assuming particles do not overlap, and assuming non-periodic boundary conditions):

$$M_{ii} = 1/6\pi\eta a \cdot I$$

$$M_{ij} = \frac{1}{8\pi\eta\|r_{ij}\|}\left[\left(I + \frac{r_{ij}r_{ij}^T}{\|r_{ij}\|^2}\right) + \frac{2a^2}{\|r_{ij}\|^2}\left(\frac{1}{3}I - \frac{r_{ij}r_{ij}^T}{\|r_{ij}\|^2}\right)\right]$$

RPY mobility matrix with periodic boundary conditions

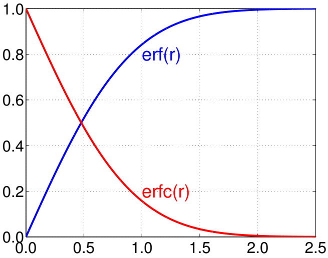
Infinite sum:

$$M_{ij} = \sum_{j'} \frac{1}{8\pi\eta\|r_{ij'}\|} \left[\left(I + \frac{r_{ij'} r_{ij'}^T}{\|r_{ij'}\|^2} \right) + \frac{2a^2}{\|r_{ij'}\|^2} \left(\frac{1}{3} I - \frac{r_{ij'} r_{ij'}^T}{\|r_{ij'}\|^2} \right) \right]$$

where j' is an image of j.

Ewald summation for the RPY matrix

$$
\begin{aligned}
M_{ij} &= M_{ij} \cdot \mathrm{erfc}(\xi r_{ij}) + M_{ij} \cdot \mathrm{erf}(\xi r_{ij}) \\
M_{ij} &= Mreal_{ij} + Mrecip_{ij} \\
Mreal_{ij} &= \sum_{m}^{\infty} M_1(r_{ij} + mL) \approx \sum_{r_{ij} < r_{cut}} M_1(r_{ij}) \\
Mrecip_{ij} &= \frac{1}{L^3} \sum_{k \neq 0}^{\infty} \exp(-ik \cdot r_{ij}) M_2(k) \approx \frac{1}{L^3} \sum_{k \neq 0}^{k_\infty} \exp(-ik \cdot r_{ij}) M_2(k)
\end{aligned}
$$



Code

The code `rpy_ewald_polyd.c` computes the (scaled) RPY mobility matrix for a given set of particle positions and a periodic box width L.

A matlab version of the code is also provided.

Mini-Project 2

- Parallelize, by using multithreading and vectorization, the computation of M, the Ewald-summed mobility matrix.
- You may want to consider
  - false sharing
  - SIMD-enabled functions

Mini-Project 2: Grading

- 0-5 points for correctness of computing M, the Ewald-summed mobility matrix, using multithreading and vectorization
- 0-4 points for overall speed on one Intel Xeon Phi coprocessor
  - provide a makefile for compiling vectorized and unvectorized (vectorization turned off, see below) versions of your code, and for running these versions on the coprocessors
- 0-3 points for vectorization
  - how fast is your code compared to your code when vectorization is turned off with `-qno-openmp-simd -no-vec -no-simd`
- 0-3 points for report ('proj2.pdf')
  - graph the time (on a log scale) for computing M vs. number of threads for the vectorized case and the case with vectorization turned off. Use the the input file `lac1_novl2.xyz` and parameters `xi`$= 1.5\pi/L$, `nr=2` and `nk=3`.
  - graph the *speedup* for the vectorized and non-vectorized cases
  - describe your implementation choices and explain why they are expected to yield higher performance than other choices

Mini-Project 2: things to consider

- Code computes one 3x3 block at a time. For better vector performance could try to compute all blocks at the same time, i.e., invert the loops and do inner loops first (maybe use elemental functions?)
- Possibly will observe better vectorization with larger matrices
- C code only computes a triangular part, need to compute the entire matrix
  - rewrite code to compute all entries
  - utilize symmetry to compute the other triangular part
- Matrix `lda` being a multiple of 64 bytes could improve efficiency
  - do not share cache lines between threads
  - rows are aligned on 64 byte boundaries
- In real applications, matrix is computed repeatedly for different particle positions
  - could separate out the preprocessing step (computing coefficients for reciprocal space calculation)
  - time 100 iterations (or whatever the test harness does) of matrix construction (rather than 1)

Mini-Project 2

Due Wed., Oct. 12, at 10 pm