Lecture 6:
A Monte Carlo case study;
OpenMP programming

David Bindel

10 Feb 2010
For HW 1, I will look for two things when grading:

- Did you find some optimization strategy that made the code faster? Getting 2 Gflop/s (say) should be reasonable.
- Did you write a correct and comprehensible description of your strategy, telling me what did or did not work?

If you copied over the files one at a time to crocus and are getting a “permission denied” error when you try `make run`, make sure that `make_sge.sh` is executable:

```
chmod +x make_sge.sh
```
Plan for today

- Last time: pthreads
- This time: OpenMP
- But first, some motivation: parallel Monte Carlo
Monte Carlo

Basic idea: Express answer $a$ as

$$a = E[f(X)]$$

for some random variable(s) $X$.

Typical toy example:

$$\frac{\pi}{4} = E[\chi_{[0,1]}(X^2 + Y^2)] \text{ where } X, Y \sim U(-1, 1).$$

We’ll be slightly more interesting...
A toy problem

Given ten points \((X_i, Y_i)\) drawn uniformly in \([0, 1]^2\), what is the expected minimum distance between any pair?
Toy problem: Version 1

Serial version:

```matlab
sum_fX = 0;
for i = 1:ntrials
    x = rand(10,2);
    fX = min distance between points in x;
    sum_fX = sum_fX + fx;
end
result = sum_fX/ntrials;
```

Parallel version: run twice and average results?! No communication — *embarrassingly parallel*

Need to worry a bit about `rand`...
Error estimators

Central limit theorem: if $R$ is computed result, then

$$R \sim N \left( E[f(X)], \frac{\sigma_f(X)}{\sqrt{n}} \right).$$

So:

- Compute sample standard deviation $\sigma_f(\hat{X})$
- Error bars are $\pm \sigma_f(\hat{X})/\sqrt{n}$
- Use error bars to monitor convergence
Serial version:

```
sum_fX = 0;
sum_fX2 = 0;
for i = 1:ntrials
    x = rand(10,2);
    fX = min distance between points in x;
    sum_fX = sum_fX + fX;
    sum_fX2 = sum_fX + fX*fX;
    result = sum_fX/i;
    errbar = sqrt(sum_fX2-sum_fX*sum_fX/i)/i;
    if (abs(errbar/result) < reltol), break; end
end
result = sum_fX/ntrials;
```

Parallel version: ?
Pondering parallelism

Two major points:

▶ How should we handle random number generation?
▶ How should we manage termination criteria?

Some additional points (briefly):

▶ How quickly can we compute $f_X$?
▶ Can we accelerate convergence (variance reduction)?
Pseudo-random number generation

- Pretend deterministic and process is random.
  \[ \Rightarrow \text{We lose if it doesn’t look random!} \]
- RNG functions have state
  \[ \Rightarrow \text{Basic } \text{random}() \text{ call is not thread-safe!} \]
- Parallel strategies:
  - Put RNG in critical section (slow)
  - Run independent RNGs per thread
    - Concern: correlation between streams
  - Split stream from one RNG
    - E.g. thread 0 uses even steps, thread 1 uses odd steps
    - Helpful if it’s cheap to skip steps!
- Good libraries help! Mersenne twister, SPRNG, ...?
One solution

- Use a version of Mersenne twister with no global state:

  ```c
  void sgenrand(long seed,
              struct mt19937p* mt);
  double genrand(struct mt19937p* mt);
  ```

- Choose pseudo-random seeds per thread at startup:

  ```c
  long seeds[NTHREADS];
  srandom(clock());
  for (i = 0; i < NTHREADS; ++i)
      seeds[i] = random();
  ...
  /* sgenrand(seeds[i], mt) for thread i */
  ```
sum_fX = 0; sum_fX2 = 0; n = 0;
for each thread in parallel
do
    fX = result of one random trial
    ++n;
    sum_fX += fX;
    sum_fX2 += fX*fX;
    errbar = ...
    if (abs(errbar/result) < reltol), break; end loop
end
result = sum_fX/n;
sum_fX = 0; sum_fX2 = 0; n = 0; done = false;
for each thread in parallel
  do
    fX = result of one random trial
    get lock
    ++n;
    sum_fX = sum_fX + fX;
    sum_fX2 = sum_fX2 + fX*fX;
    errbar = ...
    if (abs(errbar/result) < reltol)
      done = true;
    end
    release lock
  until done
end
result = sum_fX/n;
Toy problem: Version 2.3p

```plaintext
sum_fX = 0; sum_fX2 = 0; n = 0; done = false;
for each thread in parallel
    do
        batch_sum_fX, batch_sum_fX2 = B trials
        get lock
            n += B;
            sum_fX += batch_sum_fX;
            sum_fX2 += batch_sum_fX2;
            errbar = ...
        if (abs(errbar/result) < reltol)
            done = true;
        end
        release lock
    until done or n > n_max
end
result = sum_fX/n;
```
Toy problem: actual code (pthreads)
Some loose ends

- Alternative: “master-slave” organization
  - Master sends out batches of work to slaves
  - Example: SETI at Home, Folding at Home, ...
- What is the right batch size?
  - Large $B \implies$ amortize locking/communication overhead
    (and variance actually helps with contention!)
  - Small $B$ avoids too much extra work
- How to evaluate $f(X)$?
  - For $p$ points, obvious algorithm is $O(p^2)$
  - Binning points better? No gain for $p$ small...
- Is $f(X)$ the right thing to evaluate?
  - Maybe $E[g(X)] = E[f(X)]$ but $\text{Var}[g(X)] \ll \text{Var}[f(X)]$?
  - May make much more difference than parallelism!
The problem with pthreads revisited

 pthreads can be painful!
 ▶ Makes code verbose
 ▶ Synchronization is hard to think about

 Would like to make this more automatic!
 ▶ ... and have been trying for a couple decades.
 ▶ OpenMP gets us part of the way
OpenMP: Open spec for MultiProcessing

- Standard API for multi-threaded code
  - Only a spec — multiple implementations
  - Lightweight syntax
  - C or Fortran (with appropriate compiler support)
- High level:
  - Preprocessor/compiler directives (80%)
  - Library calls (19%)
  - Environment variables (1%)
#include <stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    printf("Hello world from %d\n", omp_get_thread_num());

    return 0;
}
Parallel sections

- Basic model: fork-join
- Each thread runs same code block
- Annotations distinguish shared ($s$) and private ($i$) data
- Relaxed consistency for shared data
double s[MAX_THREADS];
int i;
#pragma omp parallel shared(s) private(i)
{
    i = omp_get_thread_num();
    s[i] = i;
}
...
Critical sections

- Automatically lock/unlock at ends of *critical section*
- Automatically memory flushes for consistency
- Locks are still there if you really need them...
Critical sections

```c
#pragma omp parallel {
    ...
    #pragma omp critical my_data_cs
    {
        ... modify data structure here ...
    }
}
```
#pragma omp parallel
for (i = 0; i < nsteps; ++i) {
    do_stuff
    #pragma omp barrier
}

Toy problem: actual code (OpenMP)
A practical aside...

- GCC 4.3+ has OpenMP support by default
  - Earlier versions may support (e.g. latest Xcode gcc-4.2)
  - GCC 4.4 (prerelease) for my laptop has buggy support!
  - -O3 -fopenmp == death of an afternoon

- Need -fopenmp for both compile and link lines

```bash
gcc -c -fopenmp foo.c
gcc -o -fopenmp mycode.x foo.o
```
Parallel loops

Parallel for $i = \ldots$

- Independent loop body? At least order doesn’t matter\(^1\).
- Partition index space among threads
- Implicit barrier at end (except with `nowait`)

\(^1\)If order matters, there’s an ordered modifier.
/* Compute dot of x and y of length n */
int i, tid;
double my_dot, dot = 0;
#pragma omp parallel \\
    shared(dot,x,y,n) \\
    private(i,my_dot)
{
    tid = omp_get_thread_num();
    my_dot = 0;

#pragma omp for
for (i = 0; i < n; ++i)
    my_dot += x[i]*y[i];

#pragma omp critical
    dot += my_dot;
}
/* Compute dot of x and y of length n */
int i, tid;
double dot = 0;
#pragma omp parallel \
    shared(x,y,n) \
    private(i) \
    reduction(+:dot)
{
    #pragma omp for
    for (i = 0; i < n; ++i)
        dot += x[i]*y[i];
}
Partition index space different ways:

- **static[(chunk)]**: decide at start of loop; default chunk is \(\frac{n}{n\text{threads}}\). Lowest overhead, most potential load imbalance.
- **dynamic[(chunk)]**: each thread takes \(\text{chunk}\) iterations when it has time; default \(\text{chunk}\) is 1. Higher overhead, but automatically balances load.
- **guided**: take chunks of size unassigned iterations/threads; chunks get smaller toward end of loop. Somewhere between **static** and **dynamic**.
- **auto**: up to the system!

Default behavior is implementation-dependent.
Other parallel work divisions

- **single**: do only in one thread (e.g. I/O)
- **master**: do only in one thread; others skip
- **sections**: like cobegin/coend
Essential complexity?

Fred Brooks (*Mythical Man Month*) identified two types of software complexity: essential and accidental.

Does OpenMP address accidental complexity? Yes, somewhat!

Essential complexity is harder.
Things to still think about with OpenMP

- Proper serial performance tuning?
- Minimizing false sharing?
- Minimizing synchronization overhead?
- Minimizing loop scheduling overhead?
- Load balancing?
- Finding enough parallelism in the first place?
Some OpenMP resources

▶ http://www.openmp.org
▶ http://computing.llnl.gov/tutorials/openMP