CSE 160
Lecture 6

Memory hierarchies
N-body simulations

Scott B. Baden
Announcements

• SDSC Tour on Friday 11/1
  ‣ SDSC visit survey
    Response required to get your A1 grade feedback: http://tinyurl.com/kuqcn4h

• New partnerships? Be sure to re-register to get access to SVN for PA#2
  http://tinyurl.com/n2skwcf

• Scheduling
  ‣ No class on 10/22
  ‣ My office hours cancelled on 10/18 and 10/21
Today’s lecture

• Tree Summation
• N-body (particle) methods: Assignment #2
• Cache Coherence and Consistency
• False Sharing
Tree algorithms

- Input: an array \( x[], \) length \( N \gg P \)
- Output: Sum of the elements of \( x[] \)
- Goal: Compute the sum in \( \lg P \) time

\[
\text{sum} = 0;
\text{for } i = 0 \text{ to } N-1
\quad \text{sum += } x[i]
\]

- Assume \( P \) is a power of 2, \( K = \lg P \)
- Starter code

\[
\text{for } m = 0 \text{ to } K-1 \{ \\
\}
\]
Visualizing the Summation

Thread 0  Thread 2  Thread 4  Thread 6

0  1  2  3  4  5  6  7
0+1  2+3  4+5  6+7
0...3  4..7
0..7
Time constrained scaling

- Sum \( N \) numbers on \( P \) processors
- Let \( N \gg P \)
- Determine the largest problem that can be solved in time \( T=10^4 \) time units on 512 processors
- Let time to perform one addition = 1 time unit
- Let \( \beta \) = time to add a value inside a critical section
Performance model

• Local additions: \( \frac{N}{P} - 1 \)
• Reduction: \( \beta (\log P - 1) \)
• Since \( N \gg P \)
  \[ T(N,P) \sim \left( \frac{N}{P} \right) + \beta (\log P - 1) \]
• Determine the largest problem that can be solved in time \( T = 10^4 \) time units on \( P = 512 \) processors, \( \beta = 1000 \) time units
• Constraint: \( T(512,N) \leq 10^4 \)
  \[ \Rightarrow \left( \frac{N}{512} \right) + 1000 (\log 512 - 1) = (\frac{N}{512}) + 1000*(8) \leq 10^4 \]
  \[ \Rightarrow N \leq 1 \times 10^6 \text{ (approximately)} \]
Today’s lecture

• Tree Summation
• N-body (particle) methods: Assignment #2
• Cache Coherence and Consistency
• False Sharing
The N-body problem

- Compute trajectories of a system of N bodies often called *particles*, moving under mutual influence
  - The Greek word for particle: *somatidion* = “little body”
  - No general analytic (exact) solution when N > 2
  - Numerical simulations required
  - N can ranges from thousands to millions

- A **force law** governs the way the particles interact
  - We may not need to perform all \( O(N^2) \) force computations
  - Introduces non-uniformity due to uneven distributions
Discretization

- Particles move continuously through space and time according to a force, a continuous function of position and time: \( F(x,t) \)
- We approximate continuous values using a discrete representation
- Evaluate force field at discrete points in time, called *timesteps* \( \Delta t, 2\Delta t, 3\Delta t \)
  \( \Delta t = \text{discrete time step} \) (a parameter)
- “Push” the bodies according to Newton’s third law: \( F = ma = m \frac{du}{dt} \)
- There is no self induced force

```plaintext
while (current time < end time)
  forall bodies \( i \in 1:N \)
    compute force \( F_i \) induced by all bodies \( j \in 1:N \)
    // \( F = \text{mass} \ast \text{Acceleration} \)
    forall bodies \( i \in 1:N \)
      update velocity \( v_i \) by \( a_i \Delta t \)
      update position \( x_i \) by \( v_i \Delta t \)
  current time += \( \Delta t \)
end
```
Computing the force

• The running time of the computation is dominated by the force computation, which runs in time $O(N^2)$

Force on particle $i = \sum_{j=0:N-1} F(x_i, x_j)$

$F(\ )$ is the **force law**

• In our assignment we’ll use a simple repulsive force

  if $\text{dist}(x_i, x_j) > 0.01 \Rightarrow F(x,y) = 0$

  else $F(x,y) = C*(dx,dy)$

Where $C = (0.01/r^2 - 1/r^3)$

$r^2 = \max(dx^2 + dy^2, 10^{-6})$

$(dx,dy) = ( (x_j - x_i), (y_j - y_i))$
Overall simulation

• Synchronization?

```c
void SimulateParticles()
{
    for( int step = 0; step < nsteps; step++ ) {
        apply_forces(particles,n);

        move_particles(particles,n);

        VelNorms(particles,n,uMax,vMax,uL2,vL2);
    }
}
```
Accelerating the force computation

```c
void apply_forces( particle_t* particles, int n) {
    for( int i = 0; i < n; i++ ) {
        particles[i].ax = particles[i].ay = 0;
        for (int j = 0; j < n; j++ ){
            dx    = particles[j].x- particles[i].x;
            dy    = particles[j].y- particles[i].y;
            r2    = dx^2 + dy^2;
            if (r2 <= cutoff)
                particles[i].{ax,ay} += coef * {dx,dy};
        }
    }
}
```

©2013  Scott B. Baden / CSE 160 / Fall 2013
Reducing the cost

• We have to compute the distance check between each pair of particles, even if we don’t compute the force.
• But many particles lay beyond the cutoff

\[ \text{if } \text{dist}(x_i, x_j) > \delta \Rightarrow F(x,y) = 0 \]
\[ \text{else } \quad F(x,y) = C*(dx,dy) \]

Where
\[ C = \left( \frac{0.01}{r^2} - \frac{1}{r^3} \right) \]
\[ r^2 = \max(dx^2 + dy^2, 10^{-6}) \]
\[ (dx,dy) = ( (x_j - x_i), (y_j - y_i)) \]
\[ \delta = 0.01 \]
Reducing the cost of the force computation

- We can avoid a costly square root
- But we still have an $O(N^2)$ running time

```c
void apply_forces( particle_t* particles, int n){
    for( int i = 0; i < n; i++ ) {
        particles[i].ax = particles[i].ay = 0;
        for (int j = 0; j < n; j++ ){
            dx = particles[j].x - particles[i].x;
            dy = particles[j].y - particles[i].y;
            r2 = dx^2 + dy^2;
            if (r2 <= cutoff)
                particles[i].{ax,ay} += coef * {dx,dy};
        }
    }
}
```
Implementation

• We don’t need to compute all $O(N^2)$ distance tests
• To speed up the search for nearby particles, sort into a *chaining mesh* (Hockney & Eastwood, 1981)
• Compute forces one box at a time
• Consider particles in the 8 surrounding cells only

Jim Demmel, U. C. Berkeley
Reducing the cost of the force computation

- We still have an $O(N^2)$ running time, but have reduced the constant

```c
for( int i = 0; i < nx; i++ )
  for( int j = 0; j < ny; j++ )
    for (int i0 = -1; i0 < 2; i0++)
      for (int j0 = -1; j0 < 2; j0++)
        Update forces on Box[i,j] from particles in Box [i,j]
```

©2013 Scott B. Baden / CSE 160 / Fall 2013
Your assignment

• Split the computational box into strips-- called BLOCK partitioning
• You’ll need barrier synchronization
  • Start with the mutex implementation as discussed in class
  • Build a more efficient logarithmic time barrier using a tree structure
Tree barrier

- We can use the same idea to build a logarithmic time barrier
- Assume $P$ is a power of 2, $K = \lg P$
- Starter code
  
  ```
  for m = 0 to K-1 {
  }
  ```
Today’s lecture

- Tree Summation
- N-body (particle) methods: Assignment #2
- Cache Coherence and Consistency
- False Sharing
Cache – a recap

• **cache hit**: an access that finds the data in cache
• **cache miss**: an access that’s not
• **hit time**: time to access the higher cache
• **miss penalty**: time to move data from lower level to upper, then to cpu
• **hit rate**: percentage of time the data is found in the higher cache [**miss rate**: (1 - hit rate)]
• **cache block size** or **cache line size**: the amount of data that gets transferred on a cache miss
• **Instruction (data)** cache: a cache that holds only instructions (data); unified cache holds both I+D
Direct mapped cache

- Simplest cache

![Diagram of a direct mapped cache with three lines: Line 0, Line 1, and Line L-1. Each line contains a valid, tag, and cache block segment. The diagram shows the relationship between the tag bits, line index, and block offset.]

Randal E. Bryant and David R. O
Accessing a direct mapped cache

• Look up the line indexed by the line index
• Match the stored tag against the higher order address bits

(1) The valid bit must be set

(2) The tag bits in the cache line must match the tag bits in the address

(3) If (1) and (2) are true: we have a cache hit
Different types of caches

- Separate Instruction (I) and Data (D)
- Unified (I+D)
- Direct mapped / Set associative
- Write Through / Write Back
- Allocate on Write / No Allocate on Write
- Last Level Cache (LLC)
- Translation Lookaside Buffer (TLB)
Set associative cache

• Why use the middle bits for the index?

Randal E. Bryant and
David R. O
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, *all* others must *eventually* see the write

\[
\text{Memory} \quad \text{X:=1}
\]
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn’t have a coherent value for X
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write
SMP architectures

• Employ a *snooping protocol* to ensure coherence

• Cache controllers listen to bus activity updating or invalidating cache as needed
Memory consistency and correctness

- Cache coherence tells us that memory will eventually be consistent.
- The memory consistency policy tells us when this will happen.
- Even if memory is consistent, changes don’t propagate instantaneously.
- These give rise to correctness issues involving program behavior.
Memory consistency

• A memory system is consistent if the following 3 conditions hold
  ‣ Program order (you read what you wrote)
  ‣ Definition of a coherent view of memory ("eventually")
  ‣ Serialization of writes (a single frame of reference)
Program order

• If a processor writes and then reads the same location X, and there are no other intervening writes by other processors to X, then the read will always return the value previously written.
Definition of a coherent view of memory

- If a processor P reads from location X that was previously written by a processor Q, then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.
Serialization of writes

• If two processors write to the same location X, then other processors reading X will observe the same the sequence of values in the order written.

• If 10 and then 20 is written into X, then no processor can read 20 and then 10.
Memory consistency models

• Should it be impossible for both if statements to evaluate to true?

• With sequential consistency the results should always be the same provide that
  ‣ Each processor keeps its access in the order made
  ‣ We can’t say anything about the ordering across different processors: access are interleaved arbitrarily

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0</td>
<td>B=0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A=1</td>
<td>B=1</td>
</tr>
<tr>
<td>if (B==0) ...</td>
<td>if (A==0) ...</td>
</tr>
</tbody>
</table>
Undefined behavior in C++11

Global

```cpp
int x, y;
```

Thread 1                      Thread 2
```
x = 17
y = 37;
```
```
cout << y << " ";
cout << x << endl;
```

- Compiler may rearrange statements to improve performance
- Processor may rearrange order of instructions
- Memory system may rearrange order that writes are committed
- Memory might not get updated; “eventually can be a long time” (though in practice it’s often not)
Today’s lecture

• Tree Summation
• N-body (particle) methods: Assignment #2
• Cache Coherence and Consistency
• False Sharing
False sharing

• Consider two processors that write to different locations mapping to different parts of the same cache line
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory

©2013 Scott B. Baden / CSE 160 / Fall 2013
False sharing

- P1’s write updates main memory
- Snooping protocol invalidates the corresponding block in P0’s cache
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

- Cleanly separate locations updated by different processors
  - Manually assign scalars to a pre-allocated region of memory using pointers
  - Spread out the values to coincide with a cache line boundaries
How to avoid false sharing

- Reduce number of accesses to shared state
- False sharing occurs a small fixed number of times

```c
static int counts[];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[TID]++;

int _count = 0;
for (int k = 0; k < reps; k++){
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            _count++;
    counts[TID] = _count;
}
```

4.7s, 6.3s, 7.9s, 10.4 [NT=1,2,4,8] 3.4s, 1.7s, 0.83, 0.43 [NT=1,2,4,8]
Spreading

- Put each counter in its own cache line

```c
static int counts[];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[TID]++;

static int counts[][LINE_SIZE];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[TID][0]++;
```

<table>
<thead>
<tr>
<th></th>
<th>NT=1</th>
<th>NT=2</th>
<th>NT=4</th>
<th>NT=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>4.7 sec</td>
<td>6.3</td>
<td>7.9</td>
<td>10.4</td>
</tr>
<tr>
<td>Optimized</td>
<td>4.7</td>
<td>5.3</td>
<td>1.2</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Cache performance bottlenecks in nearest neighbor computations

• Recall the image smoothing algorithm

\[
\text{for } (i,j) \text{ in } 0:N-1 \times 0:N-1 \\
I_{\text{new}}[i,j] = \left( I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i, j+1] \right)/4
\]
Memory access pattern

- Some nearest neighbors in space are far apart in memory
- Stride = N along the vertical dimension

\[
\text{for } (i,j) \text{ in } 0:N-1 \times 0:N-1 \\
I_{\text{new}}[i,j] = \frac{(I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])}{4}
\]
False sharing and conflict misses

- False sharing involves internal boundaries, poor spatial locality, cache line internally fragmented
- Large memory access strides: conflict misses, poor cache locality
- Even worse in 3D: large strides of $N^2$
- Contiguous access on a single processor

On a single processor

On multiple processors

Cache block straddles boundary

Parallel Computer Architecture, Culler, Singh, & Gupta

©2013 Scott B. Baden / CSE 160 / Fall 2013