CSE 160
Lecture 4

Workload Distribution
Cache Coherence and Consistency
Announcements

• Room

• In class quiz on Friday 1/18
  ‣ 15 minutes
  ‣ Closed book, no notes
Today’s lecture

• Workload decomposition
• Assignment #1
• False sharing
• Cache Coherence and Consistency
Programming Lab #1

- Mandelbrot set computation with pthreads
- Observe speedups on up to 8 cores
- Load balancing
- Assignment will be automatically graded
  - Tested for correctness
  - Performance measurements
- Code available in $PUB/HW/A1
- Due on Friday 1/25 at 9pm
A quick review of complex numbers

- Define \( i = \sqrt{-1} \quad i^2 = -1 \)
- A complex number \( z = x + iy \)
  - \( x \) is called the real part
  - \( y \) is called the imaginary part
- Associate each complex number with a point in the \( x-y \) plane
- The magnitude of a complex number is the same as a vector length: \( |z| = \sqrt{x^2 + y^2} \)
- \( z^2 = (x + iy)(x + iy) \)
  \[ = (x^2 - y^2) + 2xyi \]
The Mandelbrot set

- Named after B. Mandelbrot
- For which points \( c \) in the complex plane does the following iteration remain bounded?
  \[
  z_{k+1} = z_k^2 + c, \quad z_0 = 0
  \]
  \( c \) is a complex number
- When \( |z_{k=1}| \geq 2 \) the iteration is guaranteed to diverge to \( \infty \)
- Plot the rate at which points in a given region diverge
- Stop the iterations when \( |z_{k+1}| \geq 2 \)
  or \( k \) reaches some limit
- Plot \( k \) at each position
- The Mandelbrot set is “self similar:” there are recursive structures
Parallelizing the computation

- Split the computational box into regions, assigning each region to a thread
- Different ways of subdividing the work
- “Embarrassingly” parallel, so no communication between threads
Changing the input

- Exploring different regions of the bounding box will result in different workload distributions

![Diagram showing different workload distributions for i=100 and i=1000.](image)
Load imbalances

- Some points iterate longer than others
- If we use uniform decomposition, some threads finish later than others
- We have a load imbalance

\[
\begin{align*}
\text{do} \\
\quad z_{k+1} &= z_k^2 + c \\
\text{until } (|z_{k+1}| \geq 2)
\end{align*}
\]
Visualizing the load imbalance

for i = 0 to n-1
  for j = 0 to n-1
    z = Complex (x[i], y[i])
    while (|z| < 2 or k < MAXITER)
      z = z^2 + c
    Output[i, j] = k
Load balancing efficiency

- If we ignore serial sections and other overheads, then we may express load imbalance in terms of a **load balancing efficiency** metric.

- Let each processor \( i \) complete its assigned work in time \( T_i \).

- Thus, the running time \( T_P = \text{MAX} ( T_i ) \).

- Define \( \overline{T} = \sum_i T_i \).

- We define the **load balancing efficiency** \( \eta = \frac{\overline{T}}{PT_P} \).

- Ideally \( \eta = 1.0 \).
Load balancing strategy

• Divide rows into bundles of CHUNK consecutive rows
• Processor k gets chunks 0, $NT, 2*$NT, ...
• Also called *round robin* or *block cyclic*

```plaintext
for i = 0 to n-1 on the values in thread T’s equivalence class
  for j = 0 to n-1
    z_0 = Complex (x[i],y[i])
    while (|z_k| < 2 or k > MAXITER)
      z_{k++} += z_k^2 + c
    Output(i,j) = k
```
Multidimensional chunking

• Divide by rows and columns

for i = 0 to n-1 on the values in my thread’s equivalence class
  for j = 0 to n-1 on the values in my thread’s equivalence class
    \( z_0 = \text{Complex} \ (x[i], y[i]) \)
    while (\( |z_k| < 2 \) or \( k > \text{MAXITER} \))
      \( z_{k+1} = z_k^2 + c \)
    Output(i,j) = k
Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- A spawned thread executes asynchronously until it completes
Today’s lecture

• Workload decomposition
• Assignment #1
• False sharing
• Cache Coherence and Consistency
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
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

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

t 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

```
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>

© 2010 Scott B. Baden / CSE 160 / Winter 2010
A difficult case

• Sweeping a 2D array
• Large strides also create interference patterns

Parallel Computer Architecture, Culler, Singh, & Gupta
Today’s lecture

• Workload decomposition
• Assignment #1
• False sharing

• Cache Coherence and Consistency
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

\[ X := 1 \]  \quad Memory
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

• Processors listen to bus activity
Memory consistency and correctness

• Cache coherence tells us that memory will \textit{eventually} be consistent

• The memory consistency policy tells us \textit{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
  - Definition of a coherent view of memory
  - Serialization of writes
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 model

• The memory consistency model determines when a written value will be seen by a reader

• **Sequential Consistency**
  ‣ Maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams [Lamport]
  ‣ Expensive to implement

• **Relaxed consistency**
  ‣ Enforce consistency only at well defined times
  ‣ Useful when handling false sharing

• In practice, we use sequential consistency only when absolutely necessary

• Will return to this when we discuss C++threading models