Lecture 6

Cache Interference
Load balancing
Announcements

• Quiz
Revisiting the memory model
A critical section with Dekker’s algorithm

A=1;
while (B==1){
  if (turn == 2){
    A=0;
    while (turn == 2)
      ;
    A=1;
  }
}
// Critical section
turn = 2;
A=0;

B=1;
while (A==1){
  if (turn == 1){
    B=0;
    while (turn == 1)
      ;
    B=1;
  }
}
// Critical section
turn = 1;
B=0;
Cache Interference
How to avoid false sharing

• Reduce number of accesses to shared state

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

```java
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

• Need to know the cache line size
• Your mileage will vary

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

```
4.7s, 6.3s, 7.9s, 10.4 [NT=1,2,4]
```

```
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]++;
```

```
4.7s, 5.3s, 1.2, 1.3 [NT=1,2,4,8]
```
Spreading + Padding

• Need to know the cache line size
• Your mileage will vary

```java
counts = new int [32+ NT*32];
offs = 32 + TID*32;
for (int k = 0; k<reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[offs]++;
```

4.4s, 2.2s, 1.2s, 0.55  [NT=1,2,4]

```java
counts = new int [NT*32];
offs = TID*32;
for (int k = 0; k<reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[offs]++;
```

4.4s, 5.2s, 4.7, 2.3 [NT=1,2,4,8]
A difficult case

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

Cache block

Parallel Computer Architecture, Culler, Singh, & Gupta
Load Balancing
Load imbalance in the Mandelbrot set computation

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

\[
\begin{align*}
do \\
z_{k+1} &= z_k^2 + c \\
\text{until } (|z_{k+1}| \geq 2)
\end{align*}
\]
Loop Schedules in PJ

- By default, ForLoops split the work evenly using a **Fixed Schedule**
- There is also a **dynamic schedule**, that allows threads to share the work more evenly
- AKA **Processor self-scheduling**
Dynamic Schedules

- **Chunk size** specifies granularity
- **Guided schedule** adapts chunk size to remaining work
Granularity tradeoff

• We need to choose an appropriate task granularity
• The finest granularity: each point is a task
• Coarsest granularity: one block per processor
• What is the tradeoff?

Running time

Increasing granularity →
Simulations

• Let’s simulate the workload distribution
• Assumptions:
  ▶ Work distribution is instantaneous (zero cost)
  ▶ Running time time is proportional to $k$
• Vary the chunk size $b$
• Optimal running time on 16 processors: 3044
Limits to performance

- What happened when the chunk size was too large?
- Consider P=8
- The running times on the processors:
  4771  5855  6018  6101
  7129  6390  6470  5964
- The optimal running time is 6088
Performance under PJ

- Minimum times of 3 runs done on ieng6-203
- Fixed, single run: 94 580 3951 7208 7193 3942 580 93
- Dynamic(32): 1909 1907 1906 1914 1906 1906 1911 1912
- $\texttt{pub/examples/MandelbrotSetSmp4.java}$

<table>
<thead>
<tr>
<th>NT</th>
<th>F</th>
<th>D</th>
<th>D(8)</th>
<th>D(16)</th>
<th>D(32)</th>
<th>D(64)</th>
<th>G(8)</th>
<th>G(16)</th>
<th>G(32)</th>
<th>G(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>12329114</td>
<td>96614899</td>
<td>1489614892</td>
<td>1497723108</td>
<td>2311119022</td>
<td>19021</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>211707</td>
<td>7502</td>
<td>7534</td>
<td>7494</td>
<td>7532</td>
<td>7518</td>
<td>10570</td>
<td>9814</td>
<td>9809</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>411144</td>
<td>3818</td>
<td>3827</td>
<td>3803</td>
<td>3805</td>
<td>3822</td>
<td>4466</td>
<td>4535</td>
<td>4430</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>87289</td>
<td>1997</td>
<td>2044</td>
<td>1999</td>
<td>1993</td>
<td>2864</td>
<td>2662</td>
<td>2504</td>
<td>2608</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>162906</td>
<td>1993</td>
<td>2082</td>
<td>2068</td>
<td>2104</td>
<td>2313</td>
<td>2255</td>
<td>2394</td>
<td>2243</td>
</tr>
</tbody>
</table>
Under the hood

• Sample a shared counter (or queue) to obtain work
• Chunk size trades off the overhead of sampling the counter against increased load imbalance
• Each thread obtains a unique set of indices, will change from run to run

High overheads

Load imbalance

Running time

Increasing granularity →
How does self scheduling work?

private AtomicInteger N1;
private Range myLoopRange;
public Range next(int theThreadIndex) {
    for (; ;) {
        int oldN1 = N1.get();
        Range result = myLoopRange.chunk (oldN1, chunkSz);
        int N = result.length();
        if (N == 0) return null;
        int newN1 = oldN1 + N;
        // Atomically set N1 ← newN1 if (N1 == oldN1)
        if (N1.compareAndSet (oldN1, newN1))
            return result;
    }
}
How does self scheduling work?

```java
new ParallelTeam().execute (new ParallelRegion() {
    public void run() throws Exception {
        execute (0, height-1, new IntegerForLoop() {
            public void run (int first, int last) throws Exception {
                for (int r = first; r <= last; ++ r) {...}
            };
        });
    }
});
```
How does PJ assign the work?

```java
public abstract class ParallelRegion extends ParallelConstruct{
    public final void execute(
            long first,long last, LongForLoop theLoop, BarrierAction action)
    {
        ...
        theLoop.start();
        LongRange chunk;
        while ((chunk = next(TID)) != null)
        {
            run(chunk.lb(), chunk.ub());
        }
        theLoop.finish();
    }
```