Performance Issues in Parallelization
Saman Amarasinghe
Fall 2009
Today’s Lecture

Performance Issues of Parallelism

- Cilk provides a robust environment for parallelization
  - It hides many issues and tries to eliminate many problems
- Last lectures we looked at algorithmic issues → Work and Span
- Today, synchronization and memory impact on parallel performance
  - We will use OpenMP instead of Cilk
  - Most of these issues also affects Cilk programs
  - But easier to invoke and analyze without the complexities of the Cilk runtime system

Issues Addressed

- Granularity of Parallelism
- True Sharing
- False Sharing
- Load Balancing
Matrix Multiply in Cilk

cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
        for(int k=0; k < n; ++k) {
        }
    }
}

Scheduler
- Maps cilk_for into a divide and conquer pattern
- Distribute work according to a work stealing scheduler
- Hides computation distribution and load balance issues
OpenMP

A “simplified” programming model for parallelism

- Architecture independent (all shared-memory architectures)
- Fork-join model

- Parallel loops (data parallel) and parallel sections (task parallel)
- Can select from several static and dynamic scheduling policies

```c
#pragma omp for schedule (static, chunk)
for (i=0; i<N; i++)
    for(j=0; j<N; j++)
        for (k=0; k<N; k++)
            A[i][j] += B[i][k] * C[k][j];
```

© Saman Amarasinghe 2008
Static Schedules

Assume 4 processors

#pragma omp for schedule (static, 4)
for (i=0; i<16; i++)

......
Static Schedules

Assume 4 processors

#pragma omp for schedule (static, 4) for (i=0; i<16; i++)

......

#pragma omp for schedule (static, 2) for (i=0; i<16; i++)

......
Pthreads

“Assembly” level parallelism
- Directly expose the processors/cores to the programmer

You need to manage your own threads.

A good strategy
- A thread per core
  - Perhaps threads < cores so a few cores are free to run other apps and OS services
- Bind the threads to cores
- SPMD (Single Program Multiple Data) Programming

Pros:
- Full control.
- Any parallel programming pattern.

Cons:
- Small Bugs, Big Bugs and Heisenbugs

© Saman Amarasinghe 2008
Compare Performance

```c
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            A[i][j]+= B[i][k] * C[k][j];

#pragma omp parallel for
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            A[i][j]+= B[i][k] * C[k][j];

for(i =0; i < n; i++)
    #pragma omp parallel for
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            A[i][j]+= B[i][k] * C[k][j];
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1944.291</td>
<td>1.00</td>
</tr>
<tr>
<td>Outer</td>
<td>265.088</td>
<td>7.33</td>
</tr>
<tr>
<td>Inner</td>
<td>300.03</td>
<td>6.48</td>
</tr>
</tbody>
</table>
Execution of a data parallel region

Synchronization overhead
Fine Grain Parallelism

Why?
- Too little work within a parallel region
- Synchronization (start & stop parallel execution) dominates execution time

How to Detect Fine Grain Parallelism?
- Parallel execution is slower than the sequential execution or increasing the # of processors don’t increase the speedup as expected
- Measure the execution time within the parallel region

How to get Coarse Grain Parallelism?
- Reduce the number of Parallel Invocations
  - Outer loop parallelism
  - Large independent parallel regions

© Saman Amarasinghe 2008
Compare Performance

```c
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            A[i][j]+= B[i][k] * C[k][j];

#pragma omp parallel for
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            A[i][j]+= B[i][k] * C[k][j];

for(i =0; i < n; i++)
    #pragma omp parallel for
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
            A[i][j]+= B[i][k] * C[k][j];
```

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Speedup</th>
<th># of syncs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequentail</td>
<td>1944.291</td>
<td>1.00</td>
</tr>
<tr>
<td>Outer</td>
<td>265.088</td>
<td>7.33</td>
</tr>
<tr>
<td>Inner</td>
<td>300.03</td>
<td>6.48</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Parallel Performance

```c
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)

#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>30 1.00</td>
</tr>
<tr>
<td>Parallel</td>
<td>35 0.86</td>
</tr>
</tbody>
</table>
Memory System on the cagnodes

<table>
<thead>
<tr>
<th>L1 Data Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1 Instruction Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2 Cache</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 MB</td>
<td>64 bytes</td>
<td>14 cycles</td>
<td>24-way</td>
</tr>
</tbody>
</table>
MSI Protocol

Each cache line is labeled with a state:
- **M**: cache block has been modified. No other caches contain this block in M or S states.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid (same as not there).

Before a cache modifies a location, the hardware first invalidates all other copies.
True Sharing

© Saman Amarasinghe 2008
True Sharing

```c
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Instructions</th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Invalidations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>30</td>
<td>2.02E+08</td>
<td>1.00</td>
<td>0.01</td>
<td>0</td>
<td>25,840</td>
</tr>
<tr>
<td>Parallel</td>
<td>35</td>
<td>6.95E+08</td>
<td>3.44</td>
<td>0.01</td>
<td>0</td>
<td>4,875,962</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
True Sharing

No True Sharing within a data parallel region
- There cannot be read/write or write/write conflicts

Sharing across different data parallel regions/invocations

Identifying Excessive True Sharing
- Look for cache invalidations

Eliminating Excessive True Sharing
- Try to make sharing minimal
- Data in one core’s cache, let’s keep it there!
- Try to “align” computation across regions
- Enforce a scheduling technique that’ll keep the data aligned

© Saman Amarasinghe 2008
Eliminate True Sharing

```c
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Instructions</th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Invalidations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequential</strong></td>
<td>30</td>
<td>1.00</td>
<td>2.02E+08</td>
<td>1.00</td>
<td>0.01</td>
<td>25,840</td>
</tr>
<tr>
<td><strong>Parallel</strong></td>
<td>35</td>
<td>0.86</td>
<td>6.95E+08</td>
<td>3.44</td>
<td>0.01</td>
<td>4,875,962</td>
</tr>
<tr>
<td><strong>Parallel</strong></td>
<td>7.31</td>
<td>4.11</td>
<td>5.24E+08</td>
<td>2.59</td>
<td>0</td>
<td>197,679</td>
</tr>
<tr>
<td><strong>(Transformed)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Iteration Space

N deep loops → n-dimensional discrete cartesian space

- Normalized loops: assume step size = 1

```java
for(int i = 0; i < 8; i++)
    for(int j = i; j < 8; j++)
```

Iterations are represented as coordinates in iteration space

- \( \vec{i} = [i_1, i_2, i_3, \ldots, i_n] \)
Data Space

M dimensional arrays $\rightarrow$ m-dimensional discrete cartesian space

- a hypercube

```c
int A[10]
```

```latex
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
```

```c
double B[5][6]
```

```latex
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & & & & & \\
1 & & & & & \\
2 & & & & & \\
3 & & & & & \\
4 & & & & & \\
\end{array}
```
Triangular Matrix Add

for(i=0; i < n; i++)
    for(j=0; j<i; j++)
Parallelism via. Block Distribution

#pragma omp parallel for

for(i=0; i <n; i++)
    for(j=0; j<i; j++)
Parallelism via. Block Distribution

#pragma omp parallel for

for(i=0; i < n; i++)
  for(j=0; j < i; j++)

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>97.38 1.00</td>
</tr>
<tr>
<td>Block distribution</td>
<td>31.60 3.08</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Load Imbalance

Why?
- Each parallel sub-region has different amount of work
- Static: The amount of work for each sub-region is known at compile time
- Dynamic: The amount of work varies at runtime (cannot predict)

How to Detect Load Imbalance?
- Work done by each thread is not identical
  - However running many parallel sections can average this out
- Measure the difference between min and max time taken by each of the sub-regions of a parallel section. (keep the max of that and average parallel execution time over many invocation of the parallel region).

How to Eliminate Load Imbalance?
- Static: Use cyclic distribution
- Dynamic & Static: Use a runtime load balancing scheduler like a work queue or work stealing scheduler

© Saman Amarasinghe 2008
Parallelism via. Cyclic Distribution

#pragma omp parallel for
schedule(static 1)
for(i=0; i < n; i++)
    for(j=0; j<i; j++)
Parallelism via. Cyclic Distribution

#pragma omp parallel for
schedule(static 1)
for(i=0; i < n; i++)
    for(j=0; j<i; j++)
### Load Balance but no Speedup?

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Instructions</th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Invalidations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>97.38</td>
<td>666,337,984</td>
<td>1.00</td>
<td>1.00</td>
<td>0.01</td>
<td>2,331</td>
</tr>
<tr>
<td>Block distribution</td>
<td>31.60</td>
<td>144,004,800</td>
<td>0.22</td>
<td>0.75</td>
<td>1.56</td>
<td>0</td>
</tr>
<tr>
<td>Cyclic distribution</td>
<td>37.23</td>
<td>1,462,153,984</td>
<td>2.19</td>
<td>0.96</td>
<td>2.00</td>
<td>0</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
False Sharing in Cyclic Distribution

#pragma omp parallel for
  schedule(static 1)
  for(i=0; i <n; i++)
    for(j=0; j<i; j++)

© Saman Amarasinghe 2008
False Sharing

Why?
- Cache Line Bigger Than Data Size
- Cache line is shared while data is not
- Can be a problem in data parallel loops as well as across regions

How to Detect False Sharing?
- Too many conflicts (especially in a data parallel loop)

How to Eliminate False Sharing?
- Make data used within a core contiguous in memory
- Pad the ends so that no false sharing occurs at the boundaries

© Saman Amarasinghe 2008
Data Transformation

```c
int A[NP][N/NP][N];
for(p=0; p<NP; p++)
    for(i=0; i < N/NP; i++)
        for(j=0; j<i*NP+P; j++)
```
Data Transformation

```c
int A[NP][N/NP][N];
for(p=0; p<NP; p++)
    for(i=0; i <N/NP; i++)
    for(j=0; j<i*NP+P; j++)
```

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Instructions</th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Invalidations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>97.38</td>
<td>666,337,984</td>
<td>1.00</td>
<td>0.48</td>
<td>0.01</td>
<td>2,331</td>
</tr>
<tr>
<td>Block distribution</td>
<td>31.60</td>
<td>144,004,800</td>
<td>0.22</td>
<td>0.75</td>
<td>0.01</td>
<td>67,816</td>
</tr>
<tr>
<td>Cyclic distribution</td>
<td>37.23</td>
<td>1,462,153,984</td>
<td>2.19</td>
<td>0.96</td>
<td>0.01</td>
<td>1,196,448</td>
</tr>
<tr>
<td>Data Transformation</td>
<td>18.00</td>
<td>1,121,090,048</td>
<td>1.68</td>
<td>0.75</td>
<td>0.01</td>
<td>108,262</td>
</tr>
</tbody>
</table>
Cache Issues

**Cold Miss**
- The first time the data is available
- Prefetching may be able to reduce the cost

**Capacity Miss**
- The previous access has been evicted because too much data touched in between
- “Working Set” too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

**Conflict Miss**
- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays

**True Sharing Miss**
- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

**False Sharing Miss**
- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don’t get into the same cache line
Dependences

True dependence
\[ a = a \]

Anti dependence
\[ a = a \]

Output dependence
\[ a = a \]

Definition:
Data dependence exists for a dynamic instance i and j iff
- either i or j is a write operation
- i and j refer to the same variable
- i executes before j

How about array accesses within loops?
Array Accesses in a loop

for(int i = 0; i < 6; i++)
Array Accesses in a loop

```java
for(int i = 0; i < 6; i++)
```
Array Accesses in a loop

```java
for(int i = 0; i < 6; i++)
```
Array Accesses in a loop

for(int i = 0; i < 6; i++)
Array Accesses in a loop

```java
for(int i = 0; i < 6; i++)
```
How to Parallelize SOR

SOR – Successive Over Relaxation

- Ex: Simulate the flow of heat through a plane

```java
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
```
Data Dependences in SOR

\[
\text{for(int } t=1; t < \text{ steps; } t++ \text{)} \\
\text{ for(int } i=1; i < N-1; i++ \text{)} \\
\text{ for(int } j=1; j < N-1; j++ \text{)} \\
\]
Data Dependences in SOR

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)

© Saman Amarasinghe 2008
Data Dependences in SOR

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
Data Dependences in SOR

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
Data Dependences in SOR

for(int t=1; t < steps; t++)
  for(int i=1; i < N-1; i++)
    for(int j=1; j < N-1; j++)
Data Dependences in SOR

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
Data Dependences in SOR

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)

© Saman Amarasinghe 2008
Data Dependences in SOR

for(int t=1; t < steps; t++)
  for(int i=1; i < N-1; i++)
    for(int j=1; j < N-1; j++)
Data Dependences in SOR

\[
\text{for(int } t=1; \ t < \text{ steps; } t++ \text{)} \\
\quad \text{for(int } i=1; \ i < N-1; \ i++ \text{)} \\
\quad \quad \text{for(int } j=1; \ j < N-1; \ j++ \text{)} \\
\]

© Saman Amarasinghe 2008
Data Dependences in SOR

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)

© Saman Amarasinghe 2008
Creating a FORALL Loop

```c
forall??(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
```

© Saman Amarasinghe 2008
Creating a FORALL Loop

forall??(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)

© Saman Amarasinghe 2008
Creating a FORALL Loop

for (int t = 1; t < steps; t++)
    forall?? (int i = 1; i < N - 1; i++)
        for (int j = 1; j < N - 1; j++)
Creating a FORALL Loop

for(int t=1; t < steps; t++)
  forall??(int i=1; i < N-1; i++)
    forall??(int j=1; j < N-1; j++)
Creating a FORALL Loop

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        forall??(int j=1; j < N-1; j++)

© Saman Amarasinghe 2008
Creating a FORALL Loop

for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        forall??(int j=1; j < N-1; j++)
FORALL
- No “loop carried dependences”
- Fully parallel

FORACROSS
- Some “loop carried dependences”
for(int t=1; t < steps; t++) {
    #pragma omp parallel for schedule(static, 1)
    for(int i=1; i < N-1; i++) {
        for(int j=1; j < N-1; j++) {
            if (i > 1)
                pthread_cond_wait(&cond_vars[i-1][j], &cond_var_mutexes[i-1][j]);
            if (i < N-2)
                pthread_cond_signal(&cond_vars[i][j]);
        }
    }
}
Wavefront Execution

for(int t=1; t < steps; t++)
  for(int i=1; i < N-1; i++)
    for(int j=1; j < N-1; j++)
Wavefront Execution

for(int t=1; t < steps; t++)
  for(int i=1; i < 2*N-3; i++)
    for(int j=max(1, i-N+2); j < min(i, N-1); j++)
Parallelism via Wavefront

```c
for(int t=1; t < steps; t++)
    for(int i=1; i < 2*N-3; i++)
        forall(int j=max(1,i-N+2); j < min(i, N-1); j++)
```

© Saman Amarasinghe 2008
6.172 Performance Engineering of Software Systems
Fall 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.