Project 5: Parallelization

Michael Gordon
Outline

- Data Parallelism
- Shared Memory
- Threading Library
- Loop Dependence Analysis
- Parallelization Analysis Library
Data Parallelism

- Data parallelism refers to multiple cores executing the same (similar) code with different data.

For us:
  - Finding loops that are parallelizable
    - No loop carried dependences
    - Reductions on associative operations
  - Distribute iterations across cores
    - Use shared memory mechanisms to communicate
  - Wait for all iterations to finish
Shared Memory

- Physically separate memories addressed as one logically shared address space
- load and store operations communicate data implicitly
Shared Memory Consistency

- Memory accesses from a single processor occur in program order.
- Memory consistency across multiple cores see pgs. 159 – 183 of AMD Architecture Programmer’s Manual Vol. 2!
- Simple solution:
  - A core cannot write a location that another core will read or write (past or future)!
Examples

for I = 0, N {
}

for I = 0, N {
    int x;
    x = A[10] + 1;
    B[I] = x * x;
}

for I = 0, N {
    x += A[I];
}

for I = 0, N {
    x = A[10] + 1;
    B[I] = x * x;
}
AMD Opteron 270

- Memory is shared on a cache-line granularity.
  - Opteron 270 has 32 byte cache lines
- How can this affect performance?

```c
for I = 0, N {
}
```

```c
for I = 0, N step 2 {
}
```

Core 0

```c
for I = 1, N step 2 {
}
```

Core 1

BAD!
AMD Opteron 270

• Memory is shared on a cache-line granularity.
  – Opteron 270 has 32 byte cache lines
• How can this affect performance?

```
for I = 0, N {
}
```

Core 0
```
for I = 0, N/2 {
}
```

Core 1
```
for I = N/2 + 1, N {
}
```
Threading Library

• You have 4 cores available to run code
• If you can split a loop into multiple independent iterations, you can execute different iterations on different cores
  – Need to do analysis to make sure this transformation is LEGAL!
  – You are given a framework to do some analysis
  – Much more information in Lecture 16
• You are given a library to spawn threads
Parallel Execution: Threads

• We use multiple threads of a single process.
• Each iteration block will be assigned to a thread.
• Each thread has its own registers and stack, but they share global data.
• When a thread is spawned,
  – It is passed a function that defines the loop that should be run.
  – A fresh stack is created and control is passed to the function.
# Memory

![Memory Diagram](image)

**User Address Space**

<table>
<thead>
<tr>
<th>Thread 2 (stack)</th>
<th>Thread 1 (stack)</th>
<th>Stack Pointer</th>
<th>Stack Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>routine2() var1</td>
<td>routine1() var1</td>
<td>Prgrm. Counter</td>
<td>Prgrm. Counter</td>
</tr>
<tr>
<td>var2</td>
<td>var2</td>
<td>Registers</td>
<td>Registers</td>
</tr>
<tr>
<td>var3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>Process ID</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>arrayA</td>
<td>User ID</td>
<td>Locks</td>
</tr>
</tbody>
</table>
routine1()         |arrayB            |Group ID       |Sockets        |
routine2()         |                  |               |               |
|...               |                  |               |               |

---

*Courtesy: Blaise Barney, Lawrence Livermore National Laboratory*
Spawning Threads

- **void set_num_threads(int)**
  - tells the library how many threads should be spawned

- **void create_and_run_threads**
  - Spawns threads and calls function `f` from each thread
  - Waits for all function calls to return before exiting
  - **ASM:**
    - `movq $f, %rdi`
    - `call create_and_run_threads`

- **The library will pass the thread ID as the first parameter**
Threading Example (in C)

```c
void* f (void *x) {
    printf ("thread %d\n", x);
}

int main() {
    start_caliper ();
    set_num_threads (4);
    create_and_run_threads (f);
    end_caliper ();
    return 0;
}
```

silver % ./a.out
thread 0
thread 2
thread 3
thread 1
Timer: 289 usecs
silver %
Parallelization Analysis Library

- Now you can spawn threads, but how do you know your code will be correct?
- You only should worry about loops
- Dependencies between loops are carried through local and global variables
  - Local loop dependencies are easy to figure out
  - Global loop dependencies on simple variables are also easy to figure out
  - Arrays are difficult
Parallelization

int a[100];

for i=1,100
  a[i] = a[i]+1

for j=1,99
  a[j] = a[j+1]+1;

• Which loop can you parallelize?
• The first one has all loop iterations independent
• The second one depends on another value!
Parallelization Transformation

```
int a[100];
for i=1,100
  a[i] = a[i]+1
```

```
int a[100];
void* f (void *x) {
  for i=1,25
    a[i+25*(int)x] = a[i+25*(int)x]+1
}
set_num_threads (4);
create_and_run_threads (f);
```
int a[100];
for j=1,99
    a[j] = a[j+1]+1

int a[100],b[100];
void* f1 (void *x) {
    for i=1,25
        b[i+25*(int)x] = a[i+1+25*(int)x]+1
}

void* f2 (void *x) {
    for i=1,25
        a[i+25*(int)x] = b[i+25*(int)x]
}

set_num_threads (4);
create_and_run_threads (f1);
create_and_run_threads (f2);
```c
int a[100];
for (j=1; j<99; j++)
a[j] = a[j+1]+1

int a[100]

void* f (void *x) {
    for (i=1; i<=24; i++)
        a[i+25*(int)x] = a[i+1+25*(int)x]+1
}

int t1, t2, t3;
t1 = a[26]; t2 = a[51]; t3 = a[76];
set_num_threads (4);
create_and_run_threads (f);
a[25] = t1+1;
a[50] = t2+1;
a[75] = t3+1;
```
Distance Vectors

• A loop has a distance \( d \) if there exist a data dependence from iteration \( i \) to \( j \) and \( d = j-i \)

\[
\begin{align*}
\text{FOR } I &= 0 \text{ to } 5 \\
\end{align*}
\]

\[
\begin{align*}
\text{FOR } I &= 0 \text{ to } 5 \\
\end{align*}
\]

\[
\begin{align*}
\text{FOR } I &= 0 \text{ to } 5 \\
\end{align*}
\]

\[
\begin{align*}
\text{FOR } I &= 0 \text{ to } 5 \\
A[I] &= A[0] + 1
\end{align*}
\]
Multi-Dimensional Dependence

FOR I = 1 to n
    FOR J = 1 to n

\[ dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \]
Multi-Dimensional Dependence

FOR I = 1 to n
    FOR J = 1 to n

\[ dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \]

FOR I = 1 to n
    FOR J = 1 to n

\[ dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \]
Recognizing FORALL Loops

• Find data dependences in loop
  – For every pair of array accesses to the same array
    If the first access has at least one dynamic instance (an iteration) in which it refers to a location in the array that the second access also refers to in at least one of the later dynamic instances (iterations).
    Then there is a data dependence between the statements
  – (Note that same array can refer to itself – output dependences)

• Definition
  – Loop-carried dependence:
    dependence that crosses a loop boundary

• If there are no loop carried dependences $\rightarrow$ parallelizable
Distance Vector Method

• The $i^{th}$ loop is parallelizable for all dependence $d = [d_1, \ldots, d_i, \ldots, d_n]$
either
  one of $d_1, \ldots, d_{i-1} \neq 0$
or
  all $d_1, \ldots, d_i = 0$
Is the Loop Parallelizable?

\[ dv = [0] \quad \text{Yes} \]
FOR I = 0 to 5

\[ dv = [1] \quad \text{No} \]
FOR I = 0 to 5

\[ dv = [2] \quad \text{No} \]
FOR I = 0 to 5

\[ dv = [*] \quad \text{No} \]
FOR I = 0 to 5
A[I] = A[0] + 1
Are the Loops Parallelizable?

For $I = 1$ to $n$
For $J = 1$ to $n$

$dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
Yes
No

For $I = 1$ to $n$
For $J = 1$ to $n$

$dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
No
Yes
Are the Loops Parallelizable?

FOR $I = 1$ to $n$
    FOR $J = 1$ to $n$

$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

No
Yes

FOR $I = 1$ to $n$
    FOR $J = 1$ to $n$
        $B[I] = B[I-1] + 1$

$dv = \begin{bmatrix} 1 \\ * \end{bmatrix}$

No
Yes
Parallelization Analysis Library

- The library will help you find the distance vectors
  - For each pair of accesses
  - You will have to combine multiple dependence vectors into dependence ranges
  - May be used in perfectly nested loops
Parallelization Analysis Library

for $i_1 = S_1$, $T_1$ {
    for $i_2 = S_2$, $T_2$ {
        :  
        for $i_n = S_n$, $T_n$ {
            $A[x_1i_1 + x_2i_2 + ... + x_ni_n + c] = ...$
            $... = A[y_1i_1 + y_2i_2 + ... + y_ni_n + d]$
        }
    }
}

Create 2 Integer Arrays in Java:

firstAccess = \{x_1, x_2, \ldots, x_n, c\}
secondAccess = \{y_1, y_2, \ldots, y_n, d\}
Parallelization Analysis Library

Analyze.AccessPattern d =
Analyze.getAccessPattern(first, second);

Analyze.AccessPattern
• boolean distanceExists
• int[] distance
• int[] step
Parallelization Analysis Library

• The library is finding the distance vectors for:

\[
A[x_1i_1 + x_2i_2 + \ldots + x_ni_n + c] = \ldots \\
\ldots = A[y_1i_1 + y_2i_2 + \ldots + y_ni_n + d]
\]

• Solving:
  – over the normalized iteration space (0 start)
  – for the distance vectors between iterations where the 2 accesses access the same location

• Returning the base solution in `AccessPattern.distance`

• Step for additional solutions in `AccessPattern.step`

• distance + c × step, for any integer c is a solution
Example

FOR $I = 2$ to $n$
    FOR $J = 1$ to $n-1$

Distance vector: $[1,-1]$

• But our arrays are single dimensional!

FOR $I = 2$ to $n$
    FOR $J = 1$ to $n-1$

Distance vector: $[1,-1]$
Example


- Expand the index calculation to be just a linear equation with constants and loop vars:

```java
Integer ft[] = new Integer [3];
Integer st[] = new Integer [3];

ft [0] = 6; ft [1] = 1; ft [2] = 0;
```

Analyze.AccessPattern d = Analyze.getAccessPattern(ft, st);

The result will be:

```java
d.distanceExists = true
d.distance = [0,5]
d.step = [1,-6]
```
## Distance

FOR $I = 2$ to $n$
    FOR $J = 1$ to $n-1$

|-------|----------------|

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$1,4$</td>
<td>$A[23] = A[18]$</td>
</tr>
</tbody>
</table>

[0,5]
Distance + Step

FOR I = 2 to 6
FOR J = 1 to 5


[0,5] + c[1, -6] = [-1, 11] (for c = -1)
[1, -1] (for c = 1)
[2, -7] (for c = 2)
FOR I = 1 to 6
    FOR J = 1 to 6

FOR I = 1 to 6
    FOR J = 1 to 6

FOR I = 1 to 6
  FOR J = 1 to 6

\[
\begin{align*}
\end{align*}
\]

\[d.\text{distanceExists} = \text{true}\]
\[d.\text{distance} = [0,-7]\]
\[d.\text{step} = [1,-6]\]
Can We Parallelize Example?

• For our example [1, -1] is the distance vector.
• We know that we can parallelize the inner loop.