Speculative Parallelisation

Michael O’Boyle

March, 2012
Course Structure

- 5 lectures on high level restructuring for parallelism
- Dependence Analysis
- Program Transformations
- Automatic vectorisation
- Automatic parallelisation
- Speculative Parallelisation
Lecture Overview

- Based on LPRD test: Speculative Run-time Parallelisation of loops with privatization and reduction parallelism
  - Lawrence Rachwerger PLDI 1995
  - Expect you to read and understand this paper. Many follow up papers

- Types of parallel loops

- Irregular parallelism

- LPRD test and examples
### Parallel Loop : DOALL Implementation

<table>
<thead>
<tr>
<th>Do i = 1 , N</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i) = B(i)</td>
</tr>
<tr>
<td>C(i) = A(i)</td>
</tr>
<tr>
<td>Enddo</td>
</tr>
</tbody>
</table>

| p = get_num_proc() |
| fork (x_sub, p) |
| join() |

| SUBROUTINE x_sub() |
| p = get_num_proc() |
| z = my_id() |
| ilo = N/p * (z-1) +1 |
| ihi = min(N, ilo+N/p) |
| Do i = ilo , ihi |
| A(i) = B(i) |
| C(i) = A(i) |
| Enddo |
| END |

- Generate p independent threads of work
  - Each has private local variables, z, ilo, ihi
  - Access shared arrays A, B and C
Privatisation

<table>
<thead>
<tr>
<th>Do i = 1 , N</th>
<th>DO i = ilo , ihi</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp = A(i)</td>
<td>private temp</td>
</tr>
<tr>
<td>A(i) = B(i)</td>
<td>temp = A(i)</td>
</tr>
<tr>
<td>B(i) = temp</td>
<td>A(i) = B(i)</td>
</tr>
<tr>
<td>Enddo</td>
<td>B(i) = temp</td>
</tr>
<tr>
<td></td>
<td>Enddo</td>
</tr>
</tbody>
</table>

- Temp is used as temporary storage on each iteration
  - Its value is never used on subsequent iterations
  - However there is a cross iteration ant-dependence and output dependence.
  - Each local iteration of $i$ happens in order
  - Could scalar expand - but increase storage: $O(1)$ to $O(N)$
  - Alternatively each processor has a private copy: $O(p)$ cost. $p << N$
Reduction Parallelism

\[
\text{Do } i = 1 \text{ to } N \\
\quad a = a + B(i) \\
\text{Enddo}
\]

\[
\text{pa}(z) = 0 \\
\text{Do } i = \text{ilo}, \text{ihi} \\
\quad \text{pa}(z) = \text{pa}(z) + B(i) \\
\text{Enddo} \\
\text{call barrier\_sync()} \\
\text{if (z .EQ. 1)} \\
\text{Do } x = 1, p \\
\quad a = a + \text{pa}(x) \\
\text{Enddo} \\
\text{endif}
\]

- Output flow and anti dependence
  - But can perform partial sums in parallel and merge
  - Works for associative and commutative operators
Irregular Parallelism

Do $i = 1$ to $N$
  $A(X(i)) = A(Y(i)) + B(i)$
Enddo

- Cross iteration Output dependent if any $X(i_1) = X(i_2)$ $i_1 \neq i_2$
- Cross iterating Flow/anti dependent if any $X(i_1) = Y(i_2)$ $i_1 \neq i_2$
- Dependence depends on values of $X$ and $Y$ - not compile-time knowable
- More than half scientific programs are irregular - sparse arrays
Runtime Parallelisation: The idea

Do \( i = 1, \ N \)
\[
A(i+k) = A(i) + B(i)
\]
Enddo

if \((-N < K < N)\)
\[
\begin{align*}
&\text{Do } i = 1, \ N \\
&\quad A(i+k) = A(i) + B(i) \\
&\text{Enddo}
\end{align*}
\]
else
\[
\begin{align*}
&\text{Doall } i = 1, \ N \\
&\quad A(i+k) = A(i) + B(i) \\
&\text{Enddo}
\end{align*}
\]

• Select dynamically between pre-optimised versions of the code
  – Analysis at runtime
  – Here check simple but can be more complex
Runtime Parallelisation: Irregular Applications

Do $i = 1 \, , \, N$
\begin{align*}
A(w(i)) &= A(r(i)) + B(i) \\
\end{align*}
Enddo

Assume parallel then

fallback if fail

Loop not parallel if any $r(i_1) = w(i_2), i_1 \neq i_2$

Collect data access pattern and verify if dependence could occur
Speculative Doall Marking and Analysis

• Record all accesses to shadows - one per processor. Check afterwards

• Parallel speculative execution
  – Mark read and write operations into different private shadow arrays, marking write implies clear read mark
  – Increment private write counter (# write operations)

• Post speculation analysis
  – Merge private shadow arrays to global shadow arrays
  – Count elements marked write
  – (write shadow && read shadow \(\neq 0\)) implies anti/flow dependence
  – (# mod elems < #write ops) implies output deps
LRPD test Example

\begin{center}
\begin{tabular}{|l|}
  \hline
  A(4), B(5), K(5), L(5) \\
  Do i = 1, 5 \\
  \hspace{0.5cm} z = A(K(i)) \\
  \hspace{0.5cm} if B(i) then \\
  \hspace{1cm} A(L(i)) = z + C(i) \\
  \hspace{0.5cm} endif \\
  Enddo \\
  \hline
\end{tabular}
\end{center}

\begin{center}
B(1:5) = (1,0,1,0,1) \\
K(1:5) = (1,2,3,4,1) \\
L(1:5) = (2,2,4,4,2)
\end{center}

Unsafe if \( A(K(i1)) = A(L(i2)) \), \( i1 \neq i2 \)
LRPD test Marking phase

- Allocate shadow arrays $A_w$, $A_r$, $A_{np}$ one per processor. $O(np)$ overhead. Speculatively privatise $A$ and execute in parallel. Record accesses to data under test in shadows.

- Mark write()
  - increment $tw_A$ (write counter)
  - If first time $A(i)$ written in iter, mark $A_w(i)$, clear $A_r(i)$
  - (Only concerned with cross-it deps)

- Mark read $A(i)$:
  - If $A(i)$ not already written in iter, mark $A_r(i)$ and mark $A_{np}(i)$
  - Note $A_{np}(i)$ not cleared by MarkWrite. np=not privatisable
LRPD test Marking phase

A(4), B(5), K(5), L(5)

Doall i = 1,5

z = A(K(i))

if B(i) then

markread(K(i))
markwrite(L(i))

A(L(i)) = z + C(i)

endif

Enddo

• Note markread occurs inside conditional
  – Read to A only considered if z accessed.
  – Otherwise ignore
### LRPD test Results after marking

<table>
<thead>
<tr>
<th></th>
<th>A(4), B(5), K(5), L(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do i = 1, 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>z = A(K(i))</td>
</tr>
<tr>
<td>if B(i) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A(L(i)) = z + C(i)</td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>Enddo</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aw(1:4)</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ar(1:4)</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Anp(1:4)</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Aw&amp;&amp;Ar</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Aw&amp;&amp;Anp</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B(1:5) = (1, 0, 1, 0, 1)
K(1:5) = (1, 2, 3, 4, 1)
L(1:5) = (2, 2, 4, 4, 2)

where \( \text{tm}(A) = \text{sum over } Aw \)

Total number of distinct elements written

tw = 3, tm = 2
LRPD test Analysis phase

• if $Aw && Ar$ then NOT doall – read and write in diff iterations to same elem

• else if $tw = tm$ then was a DOALL – unique iterator writes

• else if $Aw && Anp$ then NOT doall

• otherwise loop privatisation valid, DOALL

\[
Aw && Ar = 0 : \text{Fail} \\
\text{tw} \neq \text{tm} : \text{Fail} \\
Aw && Anp = 0 : \text{Fail}
\]

Overall privatise - remove output dependence
LRPD test Marking phase: Handling reductions

- Allocate shadow arrays Anx one per processor. $O(np)$ overhead.
- Record accesses to data under test in shadows
- Mark Redux ()
  - mark $A(i)$ if element is NOT valid reference in reduction statement - not a reduction variable
LRPD test Marking phase

<table>
<thead>
<tr>
<th>A(4), K(4), L(4), R(4)</th>
<th>Do all i = 1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do i = 1,4</td>
<td>markwrite(k(i))</td>
</tr>
<tr>
<td>A(k(i)) =</td>
<td>markredux(k(i))</td>
</tr>
<tr>
<td>= A(L(i))</td>
<td>A(k(i)) =</td>
</tr>
<tr>
<td>A(r(i)) = A(r(i)+exp</td>
<td>markread(L(i))</td>
</tr>
<tr>
<td>Enddo</td>
<td>markredux(L(i))</td>
</tr>
<tr>
<td></td>
<td>= A(L(i))</td>
</tr>
<tr>
<td></td>
<td>markwrite(r(i))</td>
</tr>
<tr>
<td></td>
<td>A(r(i)) = A(r(i)+exp</td>
</tr>
<tr>
<td></td>
<td>Enddoall</td>
</tr>
</tbody>
</table>

Note markredux on those accesses not forming reduction
LRPD test Reduction shadow arrays

Doall i = 1,4
  markwrite(k(i))
  markredux(k(i))
  A(k(i)) =
  markread(L(i))
  markredux(L(i))
  = A(L(i))
  markread(r(i))
  markwrite(r(i))
  A(r(i)) = A(r(i)+exp
Enddoall

k(1:4) = (3,4,3,4)
L(1:4) = (3,4,3,4)
R(1:5) = (1,2,1,2)

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\hline
Aw(1:4) & 1 & 1 & 1 & 1 \\
Ar(1:4) & 0 & 0 & 0 & 0 \\
Anp(1:4) & 1 & 1 & 0 & 0 \\
Anx(1:4) & 0 & 0 & 1 & 1 \\
\end{array}
\]

where \( tm(A) = \text{sum over } Aw \).
markread((r(i)) missing from original paper. Needed to mark Anp(i)

tw = 8 , tm = 4
LRPD test Reduction Analysis phase

Follows on from previous tests - two more added

- else $Aw && Anp && Anx$ then NOT doall

- otherwise loop privatisation valid, reduction parallelism

  $Aw && Ar = 0 : \text{Fail}$
  $tw \neq tm : \text{Fail}$
  **BUT**

  $Aw && Anp && Anx = 0 : \text{Fail}$

  So reduction parallelism
LRPD test Improvements

• One dependence can invalidate speculative parallelisation
  – Partial parallelism not exploited
  – Transform so that up till first dependence parallel
  – Reapply on the remaining iterators.

• Large overheads
  – Adaptive data structures to reduce shadow array overhead

• Large amount of work in speculative parallelisation
  – Hardware support for TLS, transactional memory
  – Compiler : Combined with static analysis
Summary

- Summary of parallelisation idioms
- Irregular accesses
- Shadow arrays
- Marking and analysis for doall and reductions
- Last lecture on parallelism. Next on adaptive compilation