Speculative Parallelisation

Michael O'Boyle

March, 2014

School of informatics

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

<pre>Do i = 1 , N
  A(i) = B(i)
  C(i) = A(i)
Enddo</pre>
	<pre>p = get_num_proc()
	fork (x_sub, p)
	join()</pre>
		<pre>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</pre>

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

Privatisation

<pre>Do i = 1 , N
  temp = A(i)
  A(i) = B(i)
  B(i) = temp
Enddo</pre>
	<pre>DO i = ilo , ihi
	  private temp
	  temp = A(i)
	  A(i) = B(i)
	  B(i) = temp
	Enddo</pre>

- Temp is used as temporary storage on each iteration
  - Its value is never used on subsequent iterations
  - However there is a cross iteration anti-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

<pre>Do i = 1 to N
  a= a +B(i)
Enddo</pre>
	<pre>pa(z) = 0
	Do i = ilo, ihi
	  pa(z) = pa(z) + B(i)
	Enddo
	call barrier_sync()
	if (z .EQ. 1)
	Do x = 1,p
	  a = a + pa(x)
	Enddo
	endif</pre>

- 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 $X(i1) = X(i2)$ $i1 \neq i2$
- Cross iteration Flow/anti dependent if any $X(i1) = Y(i2)$ $i1 \neq i2$
- 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)`
	` Do i = 1 , N`
	`   A(i+k) = A(i) + B(i)`
	` Enddo`
	`else`
	` Doall i = 1 , N`
	`  A(i+k) = A(i) + B(i)`
	` Enddo`

- Select dynamically between pre-optimised versions of the code
  - Analysis at runtime
  - Here check simple but can be more complex

Runtime Parallelisation: Irregular Applications

<pre>Do i = 1 , N
  A(w(i)) = A(r(i)) + B(i)
Enddo
 	
Assume parallel then
 	
fallback if fail</pre>
	<pre>DOALL i = 1 , N
	  trace (w(i), r(i))
	  A(w(i)) = A(r(i)) + B(i)
	Enddo
	Analyse
	if (fail) // Sequential
	 DO i = 1 , N
	   A(w(i)) = A(r(i)) + B(i)
	 Enddo
	endif</pre>

Loop not parallel if any $r(i1) = w(i2)$, $i1 \neq i2$

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

<pre>A(4), B(5),K(5), L(5)
Do i = 1,5
 z = A(K(i))
 if B(i) then
   A(L(i)) = z + C(i)
 endif
Enddo</pre>
	B(1:5) = (1,0,1,0,1)
	K(1:5) = (1,2,3,4,1)
	L(1:5) = (2,2,4,4,2)

Unsafe if A(K(i1)) = A(L(i2)) , i1 $\neq$ i2

LRPD test Marking phase

- Allocate shadow arrays Aw, Ar, Anp 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 Aw(i), clear Ar(i)
  - (Only concerned with cross-it deps)

- Mark read A(i):
  - If A(i) not already written in iter, mark Ar(i) and mark Anp(i)
  - Note Anp(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

```
  A(4), B(5),K(5), L(5)
  Do i = 1,5
   z = A(K(i))
   if B(i) then
     A(L(i)) = z + C(i)
   endif
  Enddo

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

	1	2	3	4
Aw(1:4)	0	1	0	1
Ar(1:4)	1	0	1	0
Anp(1:4)	1	0	1	0
Aw&&Ar	0	0	0	0
Aw&&Anp	0	0	0	0

tw =3, tm = 2

where tm(A) = sum over Aw

Total number of distinct elements written

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 : Fail
tw $\neq$ tm : Fail
Aw && Anp =0 : Fail
Overall privatise - remove output dependence

LRPD test Marking phase: Handling reductions

- Extended to handle 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
- Read paper for details and example

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