Parallelisation

Michael O’Boyle

March 2014
Lecture Overview

- Parallelisation for fork/join
- Mapping parallelism to shared memory multi-processors
- Loop distribution and fusion
- Data Partitioning and SPMD parallelism
- Communication, synchronisation and load imbalance.
Approaches to parallelisation

• Two approaches to parallelisation
  – Traditional shared memory. Based on finding parallel loop iterations
  – Distributed memory compilation. Focus on mapping data, computation follows

• Now single address space, physically distributed memory uses a mixture of both.

• Actually, can show equivalence
Loop Parallelisation

• Assume a single address space machine. Each processor sees the same set of addresses. Do not need to know physical location of memory reference.

• Control-orientated approach. Concerned with finding independent iterations of a loop. Then map or schedule these to the processor.

• Aim: find maximum amount of parallelism and minimise synchronisation.

• Secondary aim: improve load imbalance. Inter-processor communication not considered.

• Main memory just part of hierarchy - so use uni-processor approaches.
Loop Parallelisation: Fork/join

• Fork/join assumes that there is a forking of parallel threads at the beginning of a parallel loop

• Each thread executes one or more iterations. Depend on later scheduling policy

• There is a corresponding join or synchronisation at the end

• For this reason loop parallel approaches favour outer loop parallelism

• Can use loop interchange to improve the fork/join overhead.
### Parallel Loop: DOALL Implementation

<table>
<thead>
<tr>
<th>Do $i = 1 , N$</th>
<th>$A(i) = B(i)$</th>
<th>$C(i) = A(i)$</th>
<th>SUBROUTINE $x_{\text{sub}}()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = \text{get_num_proc()}$</td>
<td>fork ($x_{\text{sub}}, p$)</td>
<td>join()</td>
<td>$p = \text{get_num_proc()}$</td>
</tr>
<tr>
<td></td>
<td>$z = \text{my_id()}$</td>
<td></td>
<td>$ilo = N/p \times (z-1) + 1$</td>
</tr>
<tr>
<td></td>
<td>$ihi = \min(N, ilo+N/p)$</td>
<td></td>
<td>$Do \ i = ilo , ihi$</td>
</tr>
<tr>
<td></td>
<td>$A(i) = B(i)$</td>
<td></td>
<td>$A(i) = B(i)$</td>
</tr>
<tr>
<td></td>
<td>$C(i) = A(i)$</td>
<td></td>
<td>$C(i) = A(i)$</td>
</tr>
<tr>
<td></td>
<td>Enddo</td>
<td></td>
<td>Enddo</td>
</tr>
<tr>
<td></td>
<td>END</td>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

- Generate $p$ independent threads of work
  - Each has private local variables, $z$, $ilo$, $ihi$
  - Access shared arrays $A$, $B$ and $C$
Loop Parallelisation: Using loop interchange

\[
\begin{align*}
\text{Do } i &= 1, N \\
\text{Do } j &= 1, M \\
&\quad a(i+1,j) = a(i,j) + c \\
&\quad \text{Enddo} \\
&\text{Enddo}
\end{align*}
\]

Interchange has reduced synchronisation overhead from $O(N)$ to 1.
Parallelisation approach

- Loop distribution eliminates carried dependences and creates opportunity for outer-loop parallelism.

- However increases number of synchronisations needed after each distributed loop.

- Maximal distribution often finds components too small for efficient parallelisation

- Solution: fuse together parallelisable loops.
Loop Fusion

- Fusion is illegal if fusing two loops causes the dependence direction to be changed

\[
\text{Do } i = 1, N \\
\quad a(i) = b(i) + c \\
\text{Enddo}
\]

\[
\text{Do } i = 1, N \\
\quad a(i) = b(i) + c \\
\quad d(i) = a(i+1) + e \\
\text{Enddo}
\]

- Profitability: Parallel loops should not generally be merged with sequential loops: Tapered fusion
Data Parallelism

- Alternative approach where we focus on mapping data rather than control flow to the machine

- Data is partitioned/distributed across the processors of the machine

- The computation is then mapped to follow the data - typically such that work writes to local data. Local write/owner computes rule.

- All of this is based on the SPMD computational model. Each processor runs one thread executing the same program, operating on the different data

- This means that loop bounds change from processor to processor.
Data Parallelism: Mapping

• Placement of work and data on processors. Assume parallelism found in a previous stage

• Typically program parallelism $O(n)$ is much greater than machine parallelism $O(p)$, $n >> p$

• We have many options as to how to map a parallel program

• Key issue: What is the best mapping that achieves $O(p)$ parallelism but minimises cost

• Costs include communication, load imbalance and synchronisation
Simple Fortran example

Dimension Integer a(4,8)
Do i = 1, 4
    Do j = 1,8
        a(i,j) = i + j
    Enddo
Enddo
Partitioning by columns of $a$ and hence iterator $j$ : Local writes

Dimension Integer $a(4,1..2)$

Do $i = 1, 4$  
    Processor 1
        Do $j = 1, 2$
            $a(i,j) = i + j$
        Enddo
    Enddo
Enddo

...  

Dimension Integer $a(4,5..6)$

Do $i = 1, 4$  
    Processor 3
        Do $j = 5, 6$
            $a(i,j) = i + j$
        Enddo
    Enddo
Enddo  
etc..
Partitioning by rows of a and hence iterator i: Local writes

Dimension Integer a(1..1,1..8)
Do i = 1, 1 Processor 1
  Do j = 1,8
    a(i,j) = i + j
  Enddo
Enddo

... Dimension Integer a(3..3,1..8)
Do i = 3, 3 Processor 3
  Do j = 1,8
    a(i,j) = i + j
  Enddo
Enddo etc..
Linear Program representation

\[
\begin{align*}
\text{Do } i &= 1, 16 \\
\text{Do } j &= 1, 16 \\
\text{Do } k &= i, 16 \\
\quad c(i,j) &= c(i,j) + a(i,k) \cdot b(j,k)
\end{align*}
\]

Polytope \(AJ \leq b\). Access matrices \(U_c U_a U_b\)

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}_c
\begin{bmatrix}
i \\
j \\
k
\end{bmatrix},
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}_a
\begin{bmatrix}
i \\
j \\
k
\end{bmatrix},
\begin{bmatrix}
1 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}_b
\begin{bmatrix}
i \\
j \\
k
\end{bmatrix}
\]

Can we automatically generate code for each processor given that writes must be local?
Partitioning: Ex. 1st index: 4 procs: c(16,16), a(16,16), b(16,16)

\[
\text{Do } i = 5,8 \\
\text{Do } j = 1,16 \\
\text{Do } k = i,16 \\
c(i,j) = c(i,j) + a(i,k) \times b(j,k)
\]

Partitioning: Determine local array bounds \( \lambda_z, \upsilon_z \) for each processor \( 1 \leq z \leq p \).

\( \lambda_1 = 1, \lambda_2 = 5, \lambda_3 = 9, \lambda_4 = 13 \)
\( \upsilon_1 = 4, \upsilon_2 = 8, \upsilon_3 = 12, \upsilon_4 = 16 \)

Determine local write constraint \( \lambda_z \leq U_c \leq \upsilon_z \), \( 5 \leq i \leq 8 \) and add to polytope

Works for arbitrary loop structures and accesses
Load Balance : 4 procs

Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j) + a(i,k)*b(j,k)
  
Assuming c, a, b are to be partitioned in a similar manner

How should we partition to minimise load imbalance?

- Row: 928,672,416,160 per processor, load imbalance: 384
- Column: 544 iterations per processor

Why this variation?
Partition by ""invariant"" iterator $j$.

Can be expressed as a polytope condition
Reducing Communication

We wish to partition work and data to reduce amount of communication or remote accesses.

Dimension \(a(n,n)\) \(b(n,n)\)

Do \(i = 1,n\)
  Do \(j = 1,n\)
    Do \(k = 1,n\)
      \(a(i,j) = b(i,k)\)
    Enddo
  Enddo
Enddo

How should we partition to reduce communication?
Reducing Communication: Column Partitioning

Each processor has columns of $a$ and $b$ allocated to it.

Look at access pattern of second processor.

The columns of $a$ scheduled to P2 access all of $b$ $n^2 - \frac{n^2}{p}$ remote access.
Reducing Communication: Row Partitioning

Each processor has rows of $a$ and $b$ allocated to it.

Look at access pattern of second processor.

The rows of $a$ scheduled to P2 access corresponding rows of $b$.

0 remote accesses.
Alignment

• The first index of a and b have the same subscript a(i,j), b(i,k)

• They are said to be aligned on this index

• Partitioning on an aligned index makes all accesses local to that array reference

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}_a, \quad \begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}_b
\]

Can transform array layout to make arrays more aligned for partitioning.

Find A such that \( AU_x \) is maximally aligned with \( U_y \)

Global alignment problem
Synchronisation

- Alignment information can also be used to eliminate synchronisation

- Early work in data parallelisation did not focus on synchronisation

- The placement of message passing synchronous communication between source and sink would (over!) satisfy the synchronisation requirement

- When using data parallel on new single address space machines, have to reconsider this.

- Basic idea, place a barrier synchronisation where there is a cross-processor data dependence.
Synchronisation

Do i = 1,16
 a(i) = b(i)
Enddo

Do i = 1,16
 c(i) = a(i)
Enddo

Do i = 1,16
 a(17-i) = b(i)
Enddo

Do i = 1,16
 c(i) = a(i)
Enddo

• Barrier placed between each loop. But are they necessary?

• Data that is written always local. (localwrite rule)

• Data that is aligned on partitioned index is local.

• No need for barriers here
Summary

- VERY brief overview of auto-parallelism
- Parallelisation for fork/join
- Mapping parallelism to shared memory multi-processors
- Data Partitioning and SPMD parallelism
- Multi-core processor are common place
- Sure to be an active area of research for years to come