Compiler Optimisation

11 – Parallelisation

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2019

Introduction

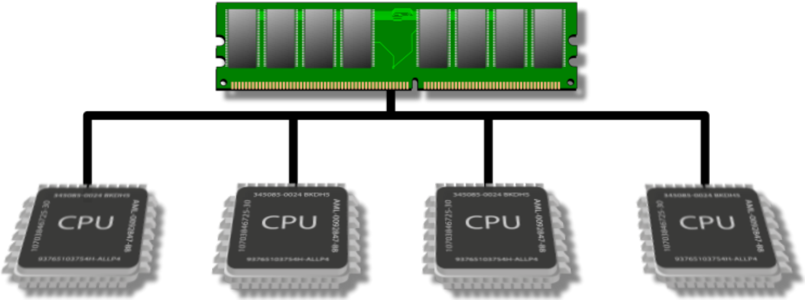
This lecture:

- 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.

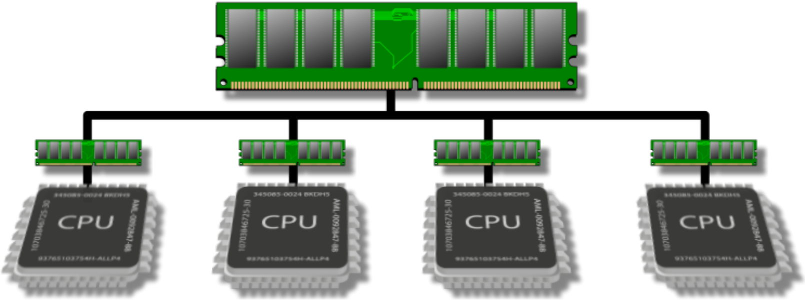
- Two approaches to parallelisation
  - Traditional shared memory
    Single address space
    Based on finding parallel loop iterations
  - Distributed memory compilation
    Physically distributed memory uses a mixture of both
    Focus on mapping data, computation
- Can show equivalence
  Implement shared memory on distributed
  Implement distributed memory on shared

Introduction
Approaches to parallelisation

Shared memory - single address space

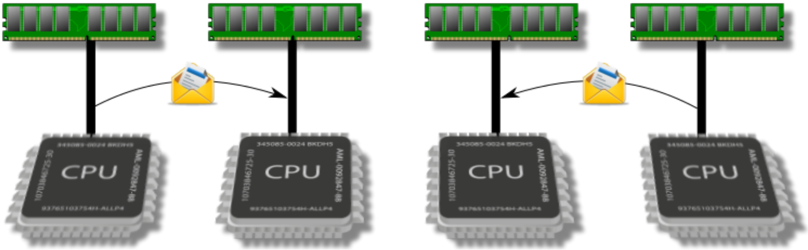


Shared memory - probably private caches, but looks like single address space



Introduction
Approaches to parallelisation

Distributed memory - each machine has own address space
Use message passing



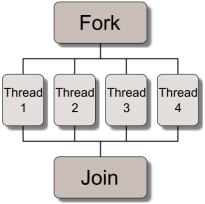
Loop Parallelisation

- 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 (create) threads at beginning of loop
- Thread executes one or more iterations.
  Depend on later scheduling policy
- Join (synchronisation/barrier) at end of loop
- Synchronisation expensive
  - Favour outer loop parallelism
  - Loop interchange



Original

```
Do i = 1, N
   A(i)=B(i)
   C(i)=A(i)
Enddo
```

Driver

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

Per thread

```
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

Loop Parallelisation

Using loop interchange

Original

```
Do i = 1, N
  Do j = 1, M
    a(i+1,j) = a(i,j)+c
  Enddo
Enddo
```

$O(n)$ synchronisation points

```
Do i = 1, N
  Parallel Do j = 1, M
    a(i+1,j) = a(i,j)+c
  Enddo
Enddo
```

Interchanged

```
Do j = 1, M
  Do i = 1, N
    a(i+1,j) = a(i,j)+c
  Enddo
Enddo
```

1 synchronisation point

```
Parallel Do j = 1, M
  Do i = 1, N
    a(i+1,j) = a(i,j)+c
  Enddo
Enddo
```

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 illegal if changes the dependence direction

Two loops - same bounds

```
Do i = 1, N
  a(i) = b(i) + c
Enddo
Do i = 1, N
  d(i) = a(i) + e
Enddo
```

Fused

```
Do i = 1, N
  a(i) = b(i) + c
  d(i) = a(i) + e
Enddo
```

Profitability: Parallel and sequential loops should not generally be merged

Loop Fusion

Fusion illegal if changes the dependence direction

Two loops - same bounds

```
Do i = 1, N
  a(i) = b(i) + c
Enddo
Do i = 1, N
  d(i) = a(i+1) + e
Enddo
```

Fused

```
Do i = 1, N
  a(i) = b(i) + c
  d(i) = a(i+1) + e
Enddo
```

Take care that fusing does not prevent parallelisation

Data Parallelism

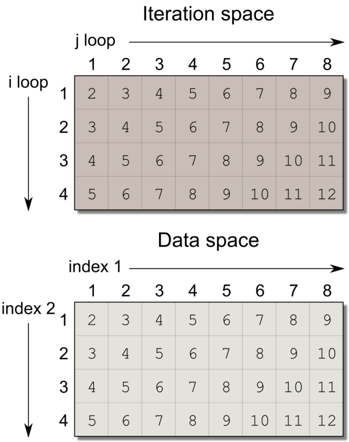
- Alternative approach where we focus on mapping data rather than control flow to the machine
- Data is partitioned/distributed across the processors of the machine
- The computation is then mapped to follow the data - typically such that work writes to local data. Local write/owner computes rule.
- 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.

- 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

Data Placement

Simple Fortran example

```
Dimension Integer a(4,8)
Do i = 1, 4
  Do j = 1, 8
    a(i,j) = i + j
  Enddo
Enddo
```



Note that here data and iteration spaces line up. Generally not the case

Data Placement

Simple Fortran example

Partitioning by columns of a and hence iterator j : Local writes

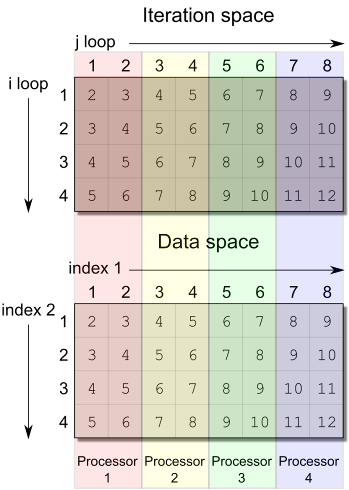
Processor 1

```
Dimension Integer
a(4,1..2)
Do i = 1, 4
  Do j = 1 ,2
    a(i,j) = i + j
  Enddo
Enddo
```

...

Processor 3

```
Dimension Integer
a(4,5..6)
Do i = 1, 4
  Do j = 5, 6
    a(i,j) = i + j
  Enddo
Enddo
```



Data Placement
Simple Fortran example

Partitioning by rows of a and hence iterator i: Local writes

Processor 1

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

...

Processor 3

```
Dimension Integer
a(3..3,1..8)
Do i = 3, 3
   Do j = 1, 8
      a(i,j) = i + j
   Enddo
Enddo
```

Iteration space

j loop →, i loop ↓

i loop \ j loop	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12

Data space

index 1 →, index 2 ↓

index 2 \ index 1	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12

Linear program representation

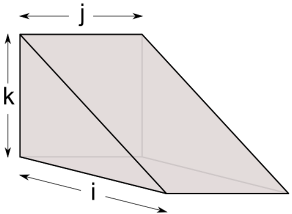
- Iteration space defined by loop bound constraints
- Constraints are affine ($\vec{a}\vec{i} \leq \vec{c}$)
- Matrix standard form ($A\vec{i} \leq \vec{c}$)
- Each constraint defines half space
- Iteration space is intersection of half spaces (polytope)
- Iterations at integer lattice points within iteration space
    - Typically unit lattices
- Array access patterns as affine functions over iteration vectors ($f(\vec{i}) = B\vec{i} + d$)

Linear program representation
Example

Iteration constraints

```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j)
               +a(i,k)*b(j,k)
```

$$1 \le i$$
$$1 \le j$$
$$i \le k$$
$$i \le 16$$
$$j \le 16$$
$$k \le 16$$

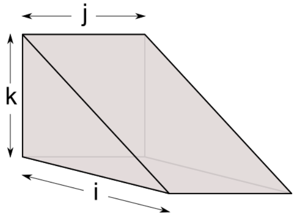


Linear program representation
Example

Make into standard form

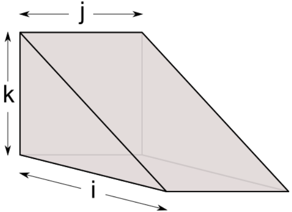
```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j)
               +a(i,k)*b(j,k)
```

$$
\begin{aligned}
1 - i &\leq 0 \\
1 - j &\leq 0 \\
i - k &\leq 0 \\
i &\leq 16 \\
j &\leq 16 \\
k &\leq 16
\end{aligned}
$$



Linear program representation
Example

```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j)
                +a(i,k)*b(j,k)
```



Make into standard form

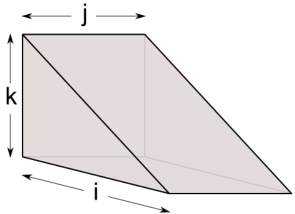
$$
\begin{aligned}
-i &\leq -1 \\
-j &\leq -1 \\
i - k &\leq 0 \\
i &\leq 16 \\
j &\leq 16 \\
k &\leq 16
\end{aligned}
$$

Linear program representation
Example

Make into standard form

```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j)
               +a(i,k)*b(j,k)
```

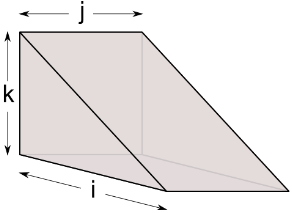
$$
\begin{aligned}
-1.i + 0.j + 0.k &\leq -1 \\
0.i + -1.j + 0.k &\leq -1 \\
1.i + 0.j + -1.k &\leq 0 \\
1.i + 0.j + 0.k &\leq 16 \\
0.i + 1.j + 0.k &\leq 16 \\
0.i + 0.j + 1.k &\leq 16
\end{aligned}
$$



Linear program representation

Example

```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j)
                +a(i,k)*b(j,k)
```



Make into standard form

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ -1 \\ 0 \\ \hline 16 \\ 16 \\ 16 \end{array}\right]$$

Linear program representation
Example

Make into standard form

```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j)
               +a(i,k)*b(j,k)
```

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ 0 \\ \hline 16 \\ 16 \\ 16 \end{array}\right]$$

Access matrices $\mathcal{U}_c\ \mathcal{U}_a\ \mathcal{U}_b$

$$\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]_c \left[\begin{array}{c} i \\ j \\ k \end{array}\right], \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right]_a \left[\begin{array}{c} i \\ j \\ k \end{array}\right], \left[\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]_b \left[\begin{array}{c} i \\ j \\ k \end{array}\right]$$

- Many transformations[1] are affine functions over linear program
- **Scanning** then regenerates code
- Partitioning loop for different processors by adding partition constraints

[1] Skew, reverse, interchange, etc

Linear program representation

Partitioning example

Split four processors equally along i
Processor 2

```
Do i = 5,8
  Do j = 1,16
    Do k = i,16
      c(i,j) = c(i,j)
              +a(i,k)*b(j,k)
```

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline -1 & 0 & 0 \\ 1 & 0 & 0 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ 0 \\ \hline 16 \\ 16 \\ 16 \\ \hline -5 \\ 8 \end{array}\right]$$

Determine local array bounds λ_z, υ_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 $\lambda_z \leq \mathcal{U}_c \leq \upsilon_z, 5 \leq i \leq 8$ and add to polytope

Works for arbitrary loop structures and accesses

Load balancing

- Load describes amount of work each processor must do
- For simple loop bodies is number of iterations assigned to each processor
- All processors wait for slowest at join point
- Want to minimise idle time at join

```
Do i = 1, 16
  Do j = 1, 16
    Do k = i, 16
      c(i,j) = c(i,j) + a(i,k) * b(j,k)
```

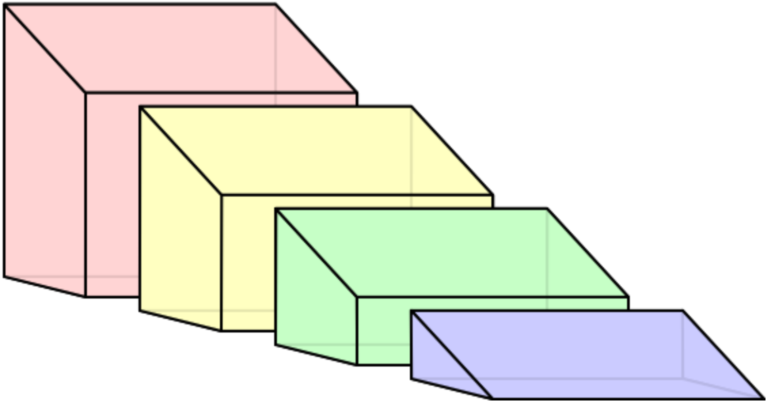
Assuming c, a, b are to be partitioned in a similar manner
How should we partition to minimise load imbalance?

- Row (along i): processor load 928, 672, 416, 160 iterations
- Column (along j): processor load 544, 544, 544, 544 iterations

Why this variation?

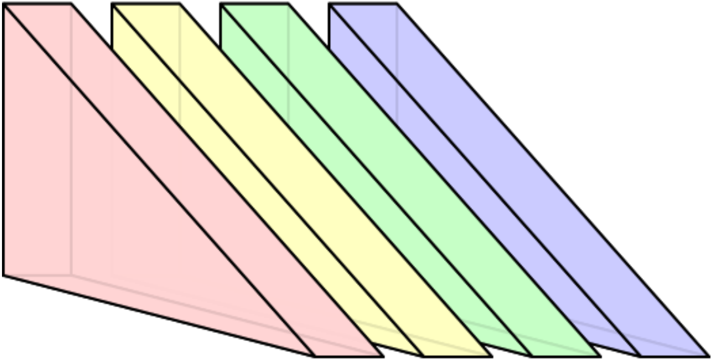
Load balance
Example

Partition by row (along i)



Load balance
Example

Partition by column (along j)



Partition by "invariant" iterator j.

Load balance
Polytope based

- Generally straightforward to 'read' from polytope
- Iteration variable with zeros elsewhere in rows and columns is 'invariant'
- Partitioning on 'invariant' yields balance

i 'conflicts' with k

$$\left[\begin{array}{ccc} \mathbf{-1} & 0 & 0 \\ 0 & -1 & 0 \\ \mathbf{1} & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

j 'invariant'

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

Reducing Communication

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

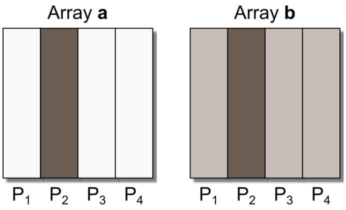
```
Dimension a(n,n) b(n,n)
Do i = 1, n
  Do j = 1, n
    Do k = 1, n
      a(i,j) = b(i,k)
    Enddo
  Enddo
Enddo
```

How should we partition to reduce communication?

Reducing communication

Each processor has rows of *a* and *b* allocated to it
Look at access pattern of second processor

```
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
```

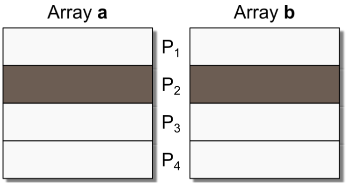


The columns of *a* scheduled to P2 access all of b $n^2 - \frac{n^2}{p}$ remote access

Reducing communication

Each processor has rows of *a* and *b* allocated to it
Look at access pattern of second processor

```
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
```



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

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}_a, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}_b$$

Can transform array layout to make arrays more aligned for partitioning.
Find $\mathcal{A}$ such that $\mathcal{A}\mathcal{U}_x$ is maximally aligned with $\mathcal{U}_y$
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. (local write 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

PPar CDT Advert