University of Leeds

Lecture 11: Reducti

Previous lectures

In the last lecture we looked at **data reorganisation** and **colle**

-
- **Collective communication** invol
  one-to-many, many-to-one or many-to
-
  Reduce the communication time *t*
  point-to-point communications.
- In MPI: `MPI_Bcast()`, `MPI_Scatter()`, `MPI_Gather()`.

Today's lecture

Today we will look at a common combination of data
reor

-
-
- Support for many parallel APIs, including
- Often optimised using a **binary tre**
- Binary trees also useful for collective comm

Reminder: Serial reduction

- Start with a large data set.
- Apply **binary operations** to *reduce* to a smaller set.

Exa


```
sum = 0;
for( i=0; i<N; i++ )
  sum += a[i];
```


Example 2: Finding the maximum element

```
max = a[0];
for( i=1; i<N; i++ )
  if( a[i]>max ) max = a[i];
```

Note each operation is performed **sequentially**.

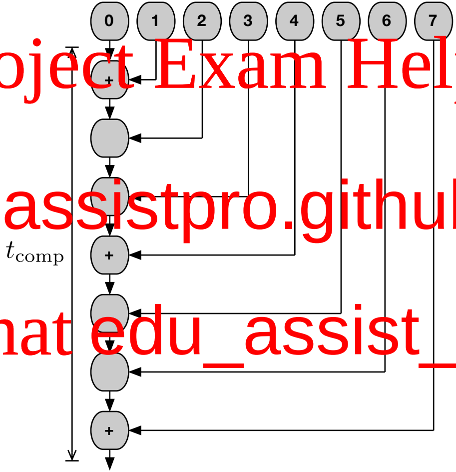
Total computation time t_{comp}

array s

- i.

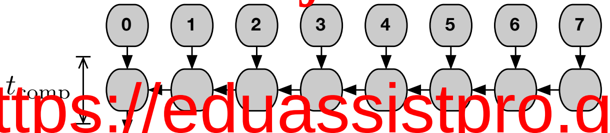
  complexity is $\mathcal{O}(n)$.

If these were **processing units**, most would be **idle** throughout most of the calculation.



Parallel reduction

Ideally we would want to perform all calculations **simultaneously**:



This *would* have a time complexity of t $=$ (1), but is not possible to achieve in practice.

For now, note that:

Any parallel reduction **must** change the sequence of calculations

Some concrete examples will be given later in this lecture.

Recap: Commutativity and associativity

Let $\otimes$ denote a general binary operator: $c = a \otimes b$.

As parallel reduction alters the sequence in which calculations are perfo

$\otimes c$

If $\otimes$ is only **approximately associative**

reduction will be (slightly) **different fro**

Some parallel reduction algorithms also requir

commutative:

An operator $\otimes$ is **commutative** if $a \otimes b = b \otimes a$

Commutativity and associativity (examples)

Combination	Examples
Ass	ul-
Ass	
Commutative; **not** associative	**Fin** t add 1, sig
Neither commutative **nor** associative	Subtraction, division

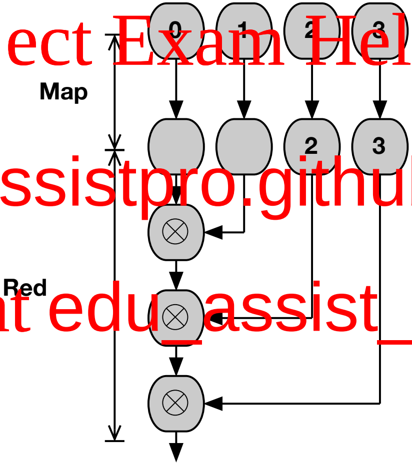
[1]Only *approximately* associative. See Worksheet 2 Question 6.
[2]*e.g.* `fn(a,b)=(a+b<1?a+b:1)` with `a=0.8`, `b=0.5` and `c=-0.3`.

MapReduce

An important application of
reduct

Map

- Fusion of a **map** followed by a **reduction**.
- Can avoid the need for **synchronisation** after the map.



[1]McCool *et al.*, *Structured parallel programming* (Morgan-Kaufmann, 2012).

Distributed systems example

Suppose a database is **distributed** over nodes in a cluster.

- Each node has access to part of the full database.

Supp[illegible]ce:

- Each node searches its local database *('map')*.
- Local results are combined to give the requir[illegible] *('reduce')*.

This **MapReduce** was developed by Google and was one of the reasons for their early success.

Example: Vector dot product

Consider the **vector dot** product (*aka* inner or scalar product):

$$n$$

In seri


```c
float dot=0.0;
for( i=0; i<n; i++ )
  dot += a[i] * b[i];
```


Note this is a **map** *(the multiplication)* followed by a **reduction** *(the summation)*.

[1] Recall maths indexing starts from 1 but computer indexing starts from 0.

Reduction in OpenMP

Code on Minerva: dotProduct_OpenMP.c

In OpenMP (*i.e.* shared memory systems), reduction is supported by the **reduction clause**:

```
float do
#prag
for( i=0
  do   +
```


- Specify the **binary operation** ('+' ('dot').
- Compiler and runtime will implement an **efficient** reduction **for the given architecture**.
- Details of the implementation **opaque** to the user.

Reduction in MPI

Code on Minerva: dotProduct_MPI.c

For MPI, distribute the full arrays on rank 0 to local arrays on each process using MPI_Scatter()[1]:

```
MPI_S

MPI_S
```


Each process then calculates its own local dot prod


```
float local_dot=0.0;
for( i=0; i<numPerProc; i++ )
  local_dot += local_a[i]*local_b[i];
```

[1]This step is the same as for vector addition; cf. Lecture 9.

MPI_Reduce()

The MPI standard supports reduction through MPI_Reduce():


```
float dot;
MPI_R
```

-
- Applied to local_dot on all process
- Reduced to dot on rank 0 (the 6t
- Other operations are supported, e MPI_MIN, logical and binary boolean operators.
- Implementation opaque to the user, but *should* be optimised for the system on which it is installed.

Efficient parallel reduction

How OpenMP and MPI implement reduction is not specified by their respective standards.

-

Usua

to consider possible implementation details to help understand performance and identify potential issues.

Parallel reduction starts after each of ads, processes) have completed their **local reduction**.

- That is, calculated the **partial sums** of all the data each processing unit is 'responsible' for.

Binary trees

The most common method to implement parallel reduction is with a **binary tree**:

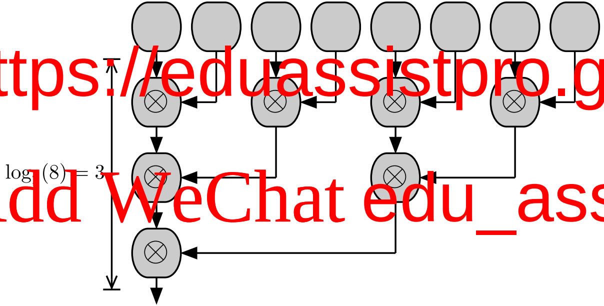
-
-
- Perform calculations **in parallel** at each level.
- Reduction time is then $\mathcal{O}(\log_2(p)$ … an $\mathcal{O}(p)$ for large p.

[1] If p is not a power of 2, round up.

Binary tree: Example 1

Not all binary trees are valid for all binary operators $\otimes$.

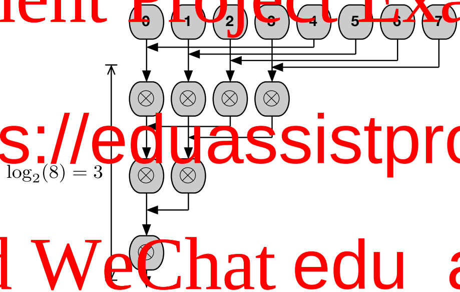
For instance, this version requires that $\otimes$ be **associative**:



The **indexing**, *i.e.* which processing units are performing the operations at each level, can be performed using bitwise arithmetic.

Binary tree: Example 2

For this example, $\otimes$ must also be **commutative**:



Indexing is easier than the previous example:

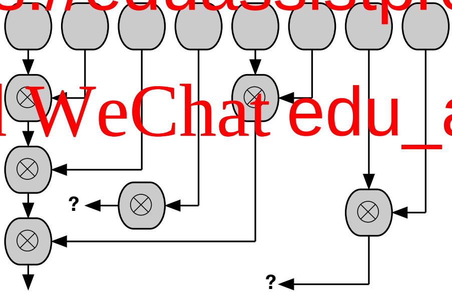
- In the first level, units 0 to $p/2$ perform the operations.
- In the next level, units 0 to $p/4$ perform the operations.
- ...

Synchronisation between levels

Note we must ensure each level's calculations have been **completed** before continuing to the **next** level.

This e

at bes



Barriers

Most parallel APIs provide a means to synchronise all processing units

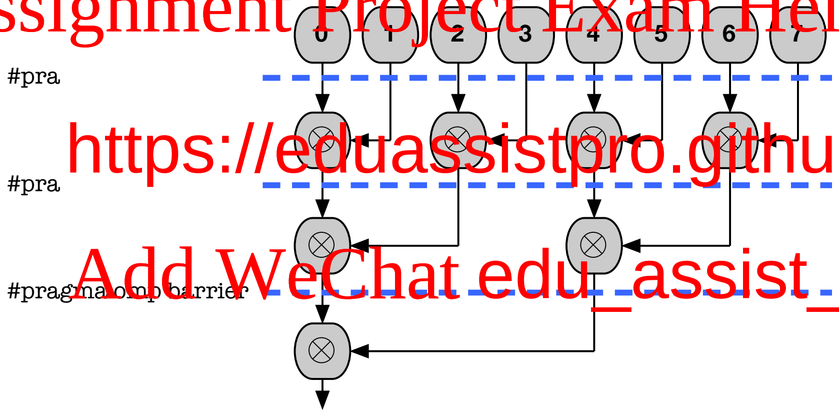
-

For instance, in OpenMP (in a parallel region):

```
#pragma omp barrier
```


- No processing unit (*i.e.* thread) will proceed past the barrier command until **all** units have reached it.

Barrier synchronisation in a binary tree



Synchronisation in MPI

MPI also provides a barrier operation:

```
MPI_B
```

How

can be

- MPI_Send(), MPI_Recv() will not re
  been sent or received

- Provides the necessary synchronisation
  processes.

Binary trees in collective communication

Note that MPI_Reduce() is a **collective communication**:

-

The binary tree pattern is typically used for all collective communication.

- Communication time $t_{\text{comm}} = \mathcal{O}($
- Faster than the $\mathcal{O}(p)$ for a loop of send-a
- 'Inverted' in the case of MPI_Bcast() and MPI_Scatter().

Summary and next lecture

Today we have looked at **parallel reduction**:

-
-
-
- In MPI, the necessary synchronisation pro **blocking communication**.

Next time we will look at **non-blocking**, or **asynchronous**, communication.