Parallel(la) Programming

Concurrency, Deadlock and Interconnect

James Pallister
Research Engineer, Embecosm
PhD Student, University of Bristol
About me

- PhD studying *software-based* energy efficiency.
  - Currently looking at compiler technology
- Last summer:
Outline

Parallelism

Deadlock & race conditions

Epiphany's interconnect
Why Parallel?

The world is parallel.

But when programming, we've been taught to think sequentially.
What is Parallelism?

- Multiple things happening at the same time to progress towards a common goal.
Amdahl's Law

1x
Serial

2x
3s
Parallelisable

3x
2.3s

4x
2s

Number of cores
Amdahl's Law

Amdahl's Speed-up

Number of processors

Speed-up

0.8
0.9
0.95
0.99
1
Real life Amdahl's Law

![Real life Amdahl's Law Graph]
Flynn's Taxonomy
The 7 dwarves of parallel computation

Structured Grid

Monte Carlo

Spectral

Dense Linear Algebra

Unstructured Grid

N-Body

Sparse Linear Algebra
Parallelism - Paradigms

- Pipelines
- Task farms
  - Server - client
- Geometric
  - Matrix multiply
  - Structured grid
Motion of cars through pipeline

from "Alan Chalmers, Practical Parallel Processing, 1996"
Pipelining

Sequential Loop:

Pipepline:

one operation at a time

n = 3 operations at a time
Task Farms

- The 'farmer' distributes work
- The workers do the work, and when finished ask for the next item.
- The farmer does a lot of communication – can become communication bound
Geometric Parallelism

- Often based on the layout of the task
- Matrix multiply
  - Grid-based
- Physical simulation
  - Often grid-based
**An Example**

**Task Decomposition:** blocks of independent computations

**Data Decomposition:** same computations over different, independent data items

**Pipelining Decomposition**
More Examples

- SIMD Parallelism
- Pipelining (Chain Process)
- Replication
- Hierarchical Fork
- Farming in Star Topology
Deadlock and Race Conditions

http://csunplugged.org/routing-and-deadlock
The Dining Philosophers
Conditions for deadlock

- Mutual exclusion
- Resource holding
- No pre-emption
- Circular waiting
The Dining Philosophers

- Plato
- Epicurus
- Paul of Tarsus
- Descartes
- Aristotle
## Race Conditions

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;hello &quot;);</code></td>
<td><code>printf(&quot;good &quot;);</code></td>
</tr>
<tr>
<td><code>printf(&quot;world &quot;);</code></td>
<td><code>printf(&quot;bye &quot;);</code></td>
</tr>
</tbody>
</table>

The possible outputs are:

- hello world good bye
- hello good bye world
- hello good world bye
- good bye hello world
- good hello bye world
- good hello world bye
Interconnect

The eMesh
The eMesh

• 3 meshes
  – Read, write, off-chip
  – Fast write mesh
• Read/write to any core
• Everything handled transparently
  – But it is useful to know some of the details for performance reasons
Routing

1. Core 32,32 sends a read request to core 39,39 and specifies its own core id (32,32) as a return (source) address.

2. Along the path, routing nodes compare the transaction address to the routers hardcoded ID and decide if the transaction should be sent north, east, south, or west.

3. Core 39,39 receives read transaction, fetches data from memory and returns the data to core 32,32 (source) using the cMesh or xMesh network.
Mesh Determinism

- Relaxed consistency
- Be careful when writing to other cores!
- Order of writing to 2 different cores is not deterministic – race conditions and deadlock
- Writing to a remote core, then attempting to read back the value is not deterministic
Safe Example

0 → 1 → 2

send to core 1
receive from core 0
print “hello”
send to core 2
receive from core 1
print “world”

hello world
Unsafe example

send to core 1
send to core 2

receive from core 0
print “hello”

receive from core 1
print “world”

hello world

Lucky
Unsafe example

send to core 1
send to core 2

receive from core 0
print “hello”

receive from core 1
print “world”

world hello

Unlucky
Make it safe

- How do we make it safe?
  - Synchronisation

```
receive from core 1
send to core 0

send to core 1
receive from core 1
send to core 2
receive from core 0
print “hello”
send to core 0
receive from core 1
print “world”

hello world
```
• Write mesh is non-blocking
• Try to structure programs and algorithms to only write to non-local memory
• Read and write from local memory is fast
A simple example - slow

Running on core 0

```c
int a[10], b[10]; // On core 0
int c;           // On core 1

for(i = 0, c = 0; i < 10; ++i)
   c += a[i] * b[i];
```

Note – without synchronisation this program is also non-deterministic
A simple example - fast

Running on core 0

```c
int a[10], b[10];  // On core 0
int c_temp;        // On core 0
int c;             // On core 1

for(i = 0, c_temp = 0; i < 10; ++i)
    c_temp += a[i] * b[i];

c = c_temp;
```

Fortunately this program is deterministic
A tricky example

Running on core 0

```c
int a[10], b[10]; // On core 0
int c[10]; // On core 1

for(i = 0; i < n; ++i)
    c[i] += a[i] * b[i];
```
Another Solution?

Core 0

```c
int a[10], b[10];
int *temp;

temp = &partial;

for(i = 0; i < 10; ++i)
{
    temp[i] = a[i] * b[i];
    sync();
}
```

Core 1

```c
int c[10];
int partial[10];

for(i = 0; i < 10; ++i)
{
    c[i] += partial[i];
    sync();
}
```

Is it faster?
Conclusion/Tips

- Many different paradigms for parallelism
- Careful of resource sharing
- Race conditions
  - Unique challenges with accessing remote cores
- Writing is faster
  - But it's challenging to pick the best way of doing this
More Info

Epiphany architecture reference manual

http://www.adapteva.com
http://www.parallella.org
http://www.embecosm.com

Questions?