Evaluating the Portability of UPC to the Cell Broadband Engine

Dipl. Inform. Ruben Niederhagen

9th International Workshop on State-of-the-Art in Scientific and Parallel Computing
Outline

- Introduction
  - UPC
  - Cell

- UPC on Cell
  - Mapping
  - Compiler and Runtime System

- Optimization: Software Managed Cache

- Conclusion
Unified Parallel C (UPC) is a C-language derivate for parallel computing.

**Shared Memory Architecture**
⇒ OpenMP

**Distributed Memory Architecture**
⇒ MPI
### Shared Data

```c
int i, j;
shared int data[2*n]; // n number of threads
```

<table>
<thead>
<tr>
<th>Global Address Space</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>...</th>
<th>Thread n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>data[n+1]</td>
<td>data[n+2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i, j</td>
<td>i, j</td>
<td>i, j</td>
<td>i, j</td>
<td></td>
</tr>
</tbody>
</table>

- Shared data can be directly accessed by all threads.
- UPC takes care for remote accesses.
shared int flag = 0;
shared int data[2];

Thread 0:
data[0] = 42;
data[1] = 23;
flag = 1;

Thread 1:
while (flag == 0);
data[0] += data[1];
Introduction – UPC

Memory Consistency

The memory consistency model of UPC defines two consistency types:

- **strict**: enforces completion of any memory access preceding a strict access, delays any subsequent access until the strict access has been finished
- **relaxed**: a sequence of memory accesses which are relaxed is allowed to be reordered as long as interdependencies are respected
**Memory Consistency**

```c
strict shared int flag = 0;
relaxed shared int data[2];
```

**Thread 0:**
```c
data[0] = 42;
data[1] = 23;
flag = 1;
```

**Thread 1:**
```c
while(flag == 0);
data[0] += data[1];
```
The Cell processor is a hybrid multicore processor:

- Power Processing Element
  - Cache
  - Power Processing Unit
- Element Interconnect Bus
- Memory Interface Controller

- Synergistic Processing Element
  - Memory Flow Controller
  - Local Store
  - Synergistic Processing Unit
The PPU runs the operating system and administrative tasks.
The SPUs handle the parallel workload.
The PPU has full transparent access to main memory.
The SPUs access their LS.
Data have to be transferred from main memory to LS explicitly.
UPC on Cell – Mapping

• no direct access to main memory → no classical Shared Memory
• LS too small for distributed data → no classical Distributed Memory
UPC on Cell – Mapping

Solution

- all shared data in main memory
- data is transferred between main memory and LS on demand
- data partitioning has to be maintained to achieve data locality:

```
```

```
for (int i = 0; i < n; i++; i)  // affinity!
data[i] = MYTHREAD;

affinity i: if (i % THREADS == MYTHREAD) ...
```
UPC on Cell – Mapping

Solution

- relaxed:
  - on read access: transfer data from main memory to LS → blocking transfer
  - on write access: transfer data from LS to main memory → non-blocking transfer

- strict:
  - as before but: usage of DMA barrier and synchronization mechanisms to maintain consistency
  → more expensive
Berkeley UPC

- use the Berkeley UPC-to-C compiler
- implement the UPC runtime layer for the Cell processor
Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks → inefficient bus utilization, retransmission of sequential data
- data must be aligned equally in a 16-byte grid → buffering and copying in LS necessary
- data size restricted to 1, 2, 4, 8, or n * 16 bytes → transfers of invalid size must be split
Optimization: Software Managed Cache

Disadvantages of the Simple Implementation

• every DMA transfer is parted into aligned 128 byte blocks → inefficient bus utilization, retransmission of sequential data
• data must be aligned equally in a 16byte grid → buffering and copying in LS necessary
Optimization: Software Managed Cache

Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks → inefficient bus utilization, retransmission of sequential data
- data must be aligned equally in a 16byte grid → buffering and copying in LS necessary
- data size restricted to 1,2,4,8, or n*16 bytes → transfers of invalid size must be split
Disadvantages of the Simple Implementation

- every DMA transfer is parted into aligned 128 byte blocks → inefficient bus utilization, retransmission of sequential data
- data must be aligned equally in a 16byte grid → buffering and copying in LS necessary
- data size restricted to 1, 2, 4, 8, or n*16 bytes → transfers of invalid size must be split

So - why not use a cache with 128 byte cache lines?
Cache and Consistency

- relaxed access: can be reordered and thus be cached without synchronization
- strict access: no cashing to achieve consistency

→ only relaxed accesses to shared data are allowed to be cached
⇒ this avoids further inter-node communication
Cache Flush

- Cache must be flushed on synchronization points: strict access, UPC barrier, UPC locks
- Only dirty bytes are allowed to be transferred to main memory:

![Graph showing performance with varying number of competing SPEs and transfers](image-url)

- One transfer
- Two transfers
- Three transfers
- Four transfers
- Five transfers
- Atomic unit
Cache Strategies for Writes

- Direct Write Through:
  - advantage: small administrative overhead
  - disadvantage: bus congestion
- Bundled Writes With Cache Line Read:
  - advantage: bus unloading
  - disadvantage: latency on first write
- Bundled Writes With Cache Line Read On Demand:
  - advantage: fast write on shared data
  - disadvantage: bus congestion on synchronized cache flush
Conclusion

What did we already do?
→ We thought about all that very carefully.

What will we do next?
→ We are going to
  • implement the proposed architecture.
  • implement several caching strategies.
  • compare and evaluate these strategies.
  • extend our approach to clusters of Cell blades.
  • integrate code partitioning into the system.
Questions?