Evaluating the Portability of UPC to the Cell Broadband Engine

Dipl. Inform. Ruben Niederhagen

June 4, 2008
JSC Cell Meeting
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 $\Rightarrow$ OpenMP

distributed memory architecture $\Rightarrow$ MPI
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>data[n+1]</td>
<td>data[n+2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i, j</td>
<td>i, j</td>
<td></td>
<td></td>
<td>i, j</td>
</tr>
</tbody>
</table>

Shared

Private
**Introduction – UPC**

### Shared Data: Blocking

```c
int i, j;
shared [2] 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>i, j</td>
<td>i, j</td>
<td></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.

---

**Porting UPC to Cell**

*Chair for Operating Systems*
**Introduction – UPC**

**Shared Data: Blocking**

```c
int i, j;
shared [2] 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>data[1]</td>
<td></td>
<td></td>
<td></td>
<td>data[2n]</td>
</tr>
<tr>
<td>i, j</td>
<td>i, j</td>
<td></td>
<td></td>
<td>i, j</td>
</tr>
</tbody>
</table>

- Shared data can be directly accessed by all threads.
- UPC takes care for remote accesses.
Introduction – UPC

Work Sharing

the loop:

\[
\text{upc}_{\text{forall}} (\text{int } i = 0; \ i < n; \ i++; \ i) \\
\text{data}[i] = \text{MYTHREAD};
\]

translates to:

\[
\text{for} \ (\text{int } i = 0; \ i < n; \ i++) \\
\text{if} \ ((i \ % \ \text{THREADS} == \ \text{MYTHREAD}) \\
\text{data}[i] = \text{MYTHREAD};
\]
Introduction – UPC

Work Sharing

equally:

\[
\text{upc}_\text{forall} \ (\text{int} \ i = 0; \ i < n; \ i++; \ \&\text{data}[i])
\]
\[
\text{data}[i] = \text{MYTHREAD};
\]

translates to:

\[
\text{for} \ (\text{int} \ i = 0; \ i < n; \ i++)
\]
\[
\text{if} \ (\&\text{data}[i] \text{ is an affine address})
\]
\[
\text{data}[i] = \text{MYTHREAD};
\]
Introduction – UPC

Work Sharing

equally:

upc_forall (int i = 0; i < n; i++; &data[i])
  data[i] = MYTHREAD;

translates to:

for (int i = 0; i < n; i++)
  if (&data[i] is an affine address)
    data[i] = MYTHREAD;

- Distribute work over several threads easily.
- Utilize data locality.
Example: Matrix Multiplication

```c
shared [N*P /THREADS] int a[N][P];
shared [N*M /THREADS] int c[N][M];
// a and c are row-wise blocked shared matrices

shared [M/THREADS] int b[P][M];
// column-wise blocking

void main (void) {
    int i, j , l; // private variables
    upc_forall (i = 0 ; i<N ; i++; &c[i][0]) {
        for (j=0 ; j<M ; j++) {
            c[i][j] = 0;
            for (l= 0 ; l<P ; l++)
                c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
Introduction – UPC

Example: Matrix Multiplication

```
upc_forall (i = 0 ; i<N ; i++; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l=0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}
```
Example: Matrix Multiplication

\[
\begin{array}{c}
\text{upc\_forall}\ (i = 0 \ ; \ i < N \ ; \ i++; \ &c[i][0])
\\
\{
\text{for (j=0 ; j < M ; j++)}
\\
\{
\text{c[i][j]} = 0;
\text{for (l= 0 ; l < P ; l++)}
\\
\text{c[i][j] += a[i][l]*b[l][j];}
\\
\}
\}
\end{array}
\]
Introduction – UPC

Example: Matrix Multiplication

```
upc_forall (i = 0 ; i<N ; i++ ; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l=0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}
Introduction – UPC

Example: Matrix Multiplication

```
upc_forall (i = 0 ; i<N ; i++ ; &c[i][0])
{
  for (j=0 ; j<M ; j++)
  {
    c[i][j] = 0;
    for (l=0 ; l<P ; l++)
    {
      c[i][j] += a[i][l]*b[l][j];
    }
  }
}
```
Introduction – UPC

Example: Matrix Multiplication

\[
\text{upc}\_\text{forall}(i = 0 ; i<N ; i++; &c[i][0])
\{
    \text{for (j=0 ; j<M ;j++)}
    \{
        c[i][j] = 0;
        \text{for (l= 0 ; l<P ; l++)}
        \quad c[i][j] += a[i][l]*b[l][j];
    \}
\}
\]
Example: Matrix Multiplication

```
upc forall (i = 0 ; i<N ; i++; &c[i][0])
{
  for (j=0 ; j<M ; j++)
  {
    c[i][j] = 0;
    for (l= 0 ; l<P ; l++)
      c[i][j] += a[i][l]*b[l][j];
  }
}
```
Example: Matrix Multiplication

```
upc forall (i = 0 ; i<N ; i++ ; &c[i][0])
{
    for (j=0 ; j<M ; j++)
    {
        c[i][j] = 0;
        for (l= 0 ; l<P ; l++)
            c[i][j] += a[i][l]*b[l][j];
    }
}
```
Example: Matrix Multiplication

\[
\text{upc}_\text{forall} (i = 0\ ;\ i < N\ ;\ i++\ ;\ &c[i][0])
\{
    \text{for} (j = 0\ ;\ j < M\ ;\ j++)
    \{
        c[i][j] = 0;
        \text{for} (l = 0\ ;\ l < P\ ;\ l++)
            c[i][j] += a[i][l]*b[l][j];
    \}
\}
Introduction – UPC

Example: Matrix Multiplication

\[
\text{upc\\_forall}(i = 0 \; ; \; i < N \; ; \; i++; \; &c[i][0])
\{
\text{for (j=0 \; ; \; j < M \; ; j++)}
\{
\text{c[i][j] = 0;}
\text{for (l = 0 \; ; \; l < P \; ; l++)}
\text{c[i][j] += a[i][l]*b[l][j];}
\}
\}
\]
Introduction – UPC

And Some More Stuff

• Dynamic Shared Memory Allocation
• Synchronization
  ▶ Barriers
  ▶ Split-Phase Barriers
  ▶ Locks
• UPC Libraries
  ▶ Collective Library
  ▶ I/O Library
• ...

Porting UPC to Cell

10 Chair for Operating Systems
Memory Consistency

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];
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.

![Diagram showing local and global views of memory accesses, with relaxed and strict accesses indicated.]
strict shared int flag = 0;
relaxed shared int data[2];

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

Thread 1:
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
Introduction – Cell

- 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.
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 resides in main memory.
- Data is transferred between main memory and LS on demand.
- Data partitioning has to be maintained to achieve data affinity:

```
```
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
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

![Diagram of memory and local storage with 128 byte blocks](image)
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
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?
Load **Cache Line** to a **Cache Frame** in LS on data access.
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
Optimization: Software Managed Cache

Cache Flush

- Cache must be flushed on synchronization points: strict access, UPC barrier, UPC locks
- In contrast to classic HW-caches, only dirty bytes are allowed to be transferred to main memory:

```
Memory
```

```
<table>
<thead>
<tr>
<th>Local Storage 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 bytes</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Local Storage 2</th>
</tr>
</thead>
</table>
```

Porting UPC to Cell
Optimization: Software Managed Cache

Cache Flush

![Graph showing Cache Flush times for different numbers of competing SPEs and transfer counts.]

- One transfer
- Two transfers
- Three transfers
- Four transfers
- Five transfers
- Atomic unit

µ-secs vs. number of competing SPEs
Cache Strategy for Reads

read access

resides data in cache?

yes

return data

no

load cache line to empty cache frame
Cache Strategy for Reads

- **read access**
  - yes: return data
  - no: load cache line to empty cache frame

- **resides data in cache?**
  - yes
  - no: blocking call!
Cache Strategies for Writes

Direct Write Through

write access

resides data in cache?

no

use empty cache frame as buffer

yes

copy data to cache

transfer data to main memory
Cache Strategies for Writes

Direct Write Through

write access

resides data in cache?

yes

no

use empty cache frame as buffer

transfer data to main memory

in background

copy data to cache
Optimization: Software Managed Cache

Cache Strategies for Writes

Bundled Writes With Cache Line Read

write access

resides data in cache?

no

read cache line to empty cache frame

yes

copy data to cache
Cache Strategies for Writes

Bundled Writes With Cache Line Read

write access

resides data in cache?

yes

no

blocking call!

read cache line to empty cache frame

copy data to cache
Cache Strategies for Writes

Bundled Writes With Cache Line Read On Demand

write access

resides data in cache?

yes

no

copy data to cache

copy data to empty cache frame
Cache Strategies for Writes

Bundled Writes With Cache Line Read On Demand

- **write access**
  - resides data in cache?
    - yes: copy data to cache
    - no: mark cache frame as write only!
      - copy data to empty cache frame
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?