10 April 2012

IN-MEMORY COMMUNICATION
EXPERIENCES WITH THE SCC

Randolf Rotta, Thomas Prescher, Jana Traue, Jörg Nolte
Brandenburg University of Technology Cottbus
OUTLINE

1. Why Messaging?
2. How we use it
3. Protocols on the Intel SCC
4. Benchmarks and Experiences
MESSAGES ARE FOR COORDINATION

Traditional usage

- transfer data
- synchronize running tasks
- initiate new tasks remotely

But: Many-core processors are hybrid systems

- distributed cores → message passing
- shared memory → data sharing

⇒ combine benefits of both
  no need for data transfers, messages only for coordination
PAST: DISTRIBUTED MEMORY

Messages to copy data

Why Messaging?
INTEL SCC: SHARED MEMORY, PRIVATE CACHES
On-Chip Network and Message Passing Buffers
INTEL SCC: SHARED MEMORY, PRIVATE CACHES

Large messages push out cache contents $\Rightarrow$ inefficient

1) read from memory

2) write to MPB

3) read from MPB

4)
INTEL SCC: SHARED MEMORY, PRIVATE CACHES

Shared memory for data and messages

1) modify data + flush
2) send address + "needs flush"
3) flush + read
INTEL MIC: SHARED L2 CACHE
Shared L2$ for data; Messages stored in L2$

1) modify data
2) send address
3) read
SUMMARY: MANY SMALL MESSAGES

- initiate tasks
- transfer addresses, parameters, results
- propagate consistency events
- synchronize running tasks

⇒ very small messages sufficient
  but high number

- examples: LRPC, Barreelfish, X10, ...
OUTLINE

1. Why Messaging?

2. How we use it

3. Protocols on the Intel SCC

4. Benchmarks and Experiences
SOFTWARE STACK: TACO

- aPGAS similar to X10, but pure C++
- global object pointers, remote method calls
- collective operations on groups
- based on cooperative user-level threads and active messages

Detail: Protocol Interface

- send(dest, msg-ptr, length) asynchronous
- probe(receive-callback) receive from any source callback is applied on each message
SOFTWARE STACK: MEMORY-EFFICIENT SHARING

- SCC: LUT remapping, manual L2$ flushing
  ⇒ 2.5GB for shared objects
- hides coherence, replication, . . .
  using RMIs & collective operations internally
  ⇒ shared arrays, sets, graphs, . . .
- coming soon: parallel graph partitioning

Observation
protocol performance limits level of parallelism

2 · How we use it
OUTLINE

1. Why Messaging?

2. How we use it

3. Protocols on the Intel SCC

4. Benchmarks and Experiences
NVP: NOTIFICATION VECTOR PROTOCOL

sender

$M_s(1)$

$\cdots$

$M_s(d)$

$\cdots$

$M_s(48)$

$A_s(1)$

$A_s(d)$

$A_s(48)$

$\cdots$

$N_s(\cdot)$

$\cdots$

unused

receiver

$M_d(1)$

$\cdots$

$M_d(48)$

$A_d(\cdot)$

$\cdots$

$N_d(1)$

$N_d(s)$

$N_d(48)$

unused
NVP: NOTIFICATION VECTOR PROTOCOL

sender

receiver

message slots

acknowledge flags

notification flags

1) is free?

2) write msg

unused

Unused

Unused

sender

receiver

$M_s(1)$

$M_s(d)$

$M_s(48)$

$A_s(1)$

$A_s(d)$

$A_s(48)$

$N_s(\cdot)$

$N_s(\cdot)$

$N_s(48)$

$M_d(1)$

$\cdots$

$M_d(48)$

$A_d(\cdot)$

$A_d(\cdot)$

$N_d(1)$

$N_d(s)$

$N_d(48)$
NVP: NOTIFICATION VECTOR PROTOCOL

Sender:
- $M_s(1)$
- $M_s(d)$
- $M_s(48)$
- $A_s(1)$
- $A_s(d)$
- $A_s(48)$
- $N_s(\cdot)$
- $\cdot$
- Unused

Receiver:
- $M_d(1)$
- $\cdot$
- $M_d(48)$
- $A_d(\cdot)$
- $\cdot$
- $N_d(1)$
- $N_d(s)$
- $N_d(48)$
- Unused

Message slots
Acknowledgment flags
Notification flags

3) Mark used
4) Write notify (byte)
NVP: NOTIFICATION VECTOR PROTOCOL

sender

\[ M_s(1) \ldots M_s(d) \ldots M_s(48) \]

\[ A_s(1) \ldots A_s(d) \ldots A_s(48) \]

\[ N_s(\cdot) \ldots \]

unused

receiver

\[ M_d(1) \ldots \]

\[ M_d(48) \ldots \]

\[ A_d(\cdot) \ldots \]

\[ N_d(1) \ldots N_d(s) \ldots N_d(48) \]

unused

message slots

acknowledge flags

notification flags

5) read flags (2 lines)

6) read msg
NVP: NOTIFICATION VECTOR PROTOCOL

sender

Mₕ(1)

Mₕ(d)

Mₕ(48)

Aₕ(1)

Aₕ(d)

Aₕ(48)

Nₕ(\cdot)


receiver

Mₜ(1)

Mₜ(d)

Mₜ(48)

Aₜ(\cdot)

Aₜ(s)

Aₜ(48)

Nₜ(1)

Nₜ(s)

Nₜ(48)

message slots

acknowledge flags

notification flags

unused

3 - Protocols on the Intel SCC
NVP: NOTIFICATION VECTOR PROTOCOL

sender

M_s(1)
M_s(d)
M_s(48)
A_s(1)
A_s(d)
A_s(48)
N_s(·)

receiver

M_d(1)
M_d(·)
M_d(48)
A_d(·)
N_d(1)
N_d(s)
N_d(48)

message slots
acknowledge flags
notification flags
unused

3 · Protocols on the Intel SCC
SRBP: SYNCHRONIZED RINGBUFFER PROTOCOL

sender

receiver

message slots
as ring buffer

write positions
atomic counters
(off-chip FPGA)

read position

3 · Protocols on the Intel SCC
SRBP: SYNCHRONIZED RINGBUFFER PROTOCOL

Sender

- $M_s(1)$
- $M_s(N)$

Receiver

- $M_d(1)$
- $M_d(N)$

Message slots as ring buffer

Read position

Write positions atomic counters (off-chip FPGA)

1) Atomic increment write pos.

2) Buffer full?
SRBP: SYNCHRONIZED RINGBUFFER PROTOCOL

1. Send message to sender.
2. Message is stored in the ring buffer.
3. Write message at write position.
4. Read message at read position.

Sender:
- Messages: $M_s(1)$, $M_s(N)$
- Write positions: $W_1$, $W_d$, $W_{48}$
- Atomic counters (off-chip FPGA)

Receiver:
- Messages: $M_d(1)$, $M_d(N)$
- Read position

Message slots as ring buffer.

Protocols on the Intel SCC
SRBP: SYNCHRONIZED RINGBUFFER PROTOCOL

Sender

\[ M_s(1) \]
\[ M_s(N) \]

Message slots as ring buffer

Read position

Receiver

\[ M_d(1) \]
\[ M_d(N) \]

Write positions

Atomic counters (off-chip FPGA)

3 · Protocols on the Intel SCC
SRBP: SYNCHRONIZED RINGBUFFER PROTOCOL

sender

receiver

message slots as ring buffer

write positions
atomic counters (off-chip FPGA)

read position

message slots

3 - Protocols on the Intel SCC
OUTLINE

1. Why Messaging?
2. How we use it
3. Protocols on the Intel SCC
4. Benchmarks and Experiences
- SRBP >200 cycles faster than NVP
- distance has only little impact (10%)
COLLECTIVE OPERATIONS: LOGARITHMIC GROWTH

- constant overhead per core, interleaves with other tasks
- SRBP slower than NVP (counter congestion)
- constant overhead per core, interleaves with other tasks
- SRBP slower than NVP (counter congestion)
**ROUNDTRIPS: 15X FASTER THAN RCKMPI**

- MPI: supports large messages, tags, source-dest matching, …
- NVP/SRBP: offloading 15x smaller workloads still efficient
ROUNDTRIPS: 15X FASTER THAN RCKMPI

- MPI: supports large messages, tags, source-dest matching, ...
- NVP/SRBP: offloading 15x smaller workloads still efficient
MAIN COST FACTOR: POLLING OVERHEAD

RCKMPI: polling one line per core, takes 200 cycles
NVP: polling one line per 32 cores
SRBP: polling on single word
MAIN COST FACTOR: POLLING OVERHEAD

RCKMPI: polling one line per core, takes 200 cycles

NVP: polling one line per 32 cores

SRBP: polling on single word
CONCLUSION: EFFICIENCY THROUGH SPECIALIZATION

- many-cores are hybrid systems
⇒ combine shared memory and message passing
  - just small & simple messages
⇒ simpler & faster protocols, more parallelism
  - analysis ≈ 70% overhead by notification and probe
⇒ great potential for power savings by better HW support
5. Appendix
### WGR COST MODEL

<table>
<thead>
<tr>
<th></th>
<th>UC</th>
<th>bypass</th>
<th>MPBT</th>
<th>bypass</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>5</td>
<td>5</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>g</td>
<td>51–84</td>
<td>17</td>
<td>78–108</td>
<td>39</td>
</tr>
<tr>
<td>r</td>
<td>54–87</td>
<td>19</td>
<td>60–90</td>
<td>21</td>
</tr>
</tbody>
</table>
SCC ADDRESS TRANSLATION

- 32b logical address
- mmap

- page table (4kB pages)
- 32b physical address and access mode

- L1 Cache

- CL1INVMB
- normal access read & write
- only on cache miss
- UC MPBT write
- MPBT read

- L2 Cache
- mesh interface & addr. translation
- on cache miss
- on filled buffer

- Write Combine Buffer (32B)
- Write Combine Buffer (32B)

- LUT (16MB pages)

- 34b system address (includes mesh destination)
PROTOCOL SCALABILITY ON XEON E7

![Graph showing protocol scalability on XEON E7](image-url)
PROTOCOLS: SRBP

Sender
Core

Receiver's
Counter

Receiver's
MPB

Receiver
Core

reserve slot
check free
write msg
+ nfy

MPBT

UC

poll + read
reset nfy
incr. read pos.