Message Progression in Parallel Computing - To Thread or not to Thread?

Torsten Hoefler and Andrew Lumsdaine

Open Systems Lab
Indiana University
Bloomington, USA

IEEE Cluster 2008
Tsukuba, Japan
September, 30th 2008
Introduction

Non-blocking Interfaces

- can help to hide latency
- mitigate effects of process skew
- reported application speedup up to 1.9
- requires much effort at the algorithm and implementation levels

Examples

- MPI offers non-blocking point-to-point
- non-blocking collectives are discussed for MPI-3
- GASNet is fully non-blocking
- Asynchronous I/O
Non-blocking does not mean asynchronous!

Isend/Irecv (CPU)
Computation (CPU)
Transmission (NIC)
Wait (CPU)

Isend/Irecv (CPU)
Computation (CPU)
Transmission (NIC)
Wait (CPU)
Non-blocking Middleware Implementation

**Non-blocking Send/Receive**
- eager protocol/copy for small messages
  - uses a single message
- rendezvous protocol/synchronize for large messages
  - uses multiple messages (two to three)
- OS bypass networking
  - does not involve the kernel in send/receive
  - polling to check for messages

**Non-blocking Collectives**
- similar issues as send/receive
- much more complex tasks and protocols
- multiple send/receive operations and dependencies in a single collective operation
Progression Strategies I/II

Manual Progression

- simplest to implement in a middleware
- user has to progress (e.g., calling MPI_Test)
- number of necessary progress calls depends on protocol
- best case: eager, worst case: pipelined protocols
- our proposed black-box scheme: \( N = \left\lfloor \frac{\text{size}}{\text{interval}} \right\rfloor + 1 \)

---

Torsten Hoefler and Andrew Lumsdaine

To Thread or not to Thread?
<table>
<thead>
<tr>
<th>Hardware-based Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>need to run full protocol in NIC</td>
</tr>
<tr>
<td>complicated to implement</td>
</tr>
<tr>
<td>full benefits to the user</td>
</tr>
<tr>
<td>mostly not supported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threaded Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous threads</td>
</tr>
<tr>
<td>often stated as “silver bullet” but not widely used (?)</td>
</tr>
<tr>
<td>problem with manual progression: “fire at the right time”</td>
</tr>
<tr>
<td>threads could solve this problem (woken up correctly)</td>
</tr>
<tr>
<td>could enable fully asynchronous progression</td>
</tr>
<tr>
<td>OS influence (scheduler) is significant</td>
</tr>
</tbody>
</table>
Threading Configurations

Spare Core vs. Fully Subscribed

- **two extreme scenarios**
  - **spare core**: min P/2 cores are idle (one per process)
    → used in I/O or memory-bound applications
  - **fully subscribed**: no cores idle
    → used in compute-bound applications
Polling vs. Interrupt vs. Real Time

- **Polling**
  - CPU Usage Time per Core
  - Compute thread
  - Progress thread
  - Time slice
  - Network interrupt
  - a) polling

- **Interrupt/Normal**
  - CPU Usage Time per Core
  - Compute thread
  - Progress thread
  - Time slice
  - Network interrupt
  - a) interrupt/normal

- **Interrupt/Real Time**
  - CPU Usage Time per Core
  - Compute thread
  - Progress thread
  - Time slice
  - Network interrupt
  - a) interrupt/real time

---

Torsten Hoefler and Andrew Lumsdaine

To Thread or not to Thread?
### Issues with Non-blocking Collectives

- NBCs introduce dependencies
  → e.g., sending a message in a tree
- Dependencies might lead to synchronization

### Case-study InfiniBand

- Supports polling and interrupt
- Polling bad without non-spare core, else fastest
- Interrupts are slow and cause overhead
- Scheduler issues! (timeslice 4 ms, latency 3 $\mu$s)

### Real-Time Threads in Linux

- Highest priority
- Scheduled immediately
- No preemption
Overhead of Threading

Real time vs. Normal

- normal threads: interrupts coalesce, low (no) progression
- RT-threads: each interrupt pays full overhead

Sender | Receiver
--- | ---
SQ | RQ
Data | Data

a) eager
b) rendezvous

Send completion event
Receive completion event
Point-to-point overhead

Open MPI/65536
LibOF/no thread
LibOF/thread

Overhead (usec)

Message Size (bytes)
NBC_Iallreduce Overhead on 32 Nodes - spare core

Overhead (usec)

Message Size (bytes)

no thread, no test
no thread, t=1024
thread, no test
rt thread, no test

To Thread or not to Thread?

Torsten Hoefler and Andrew Lumsdaine
To Thread or not to Thread?

Torsten Hoefler and Andrew Lumsdaine
Wrong Metric?

- often used time-based benchmark
- hides interrupt overhead costs!
- overhead $\approx 3.4\mu s$ per interrupt
- many many interrupts in an NBC; 1016 in pipelined case

Work-based benchmark

- compute fixed work quantum
- results account for interrupt overhead
- should be used for any threaded progression analysis!
Work-based results - fully subscribed - not so nice

Overhead (usec)

Message Size (bytes)

no thread, no test
no thread, t=1024
thread, no test
rt thread, no test

Torsten Hoefler and Andrew Lumsdaine
To Thread or not to Thread?
Summary

- we developed fully threaded LibNBC for IB → high overhead
- tested implementation with RT threads → lower overhead (better than theory?)
- implemented new work-based benchmarking metric → realistic (high) overhead

Conclusions and Future Work

- threaded implementation makes sense with spare cores
- very tricky without spare cores → manual again?
- investigate other options
  → signalled progression (not safe/realistic!)
  → OS involvement (opposite to OS bypass)
  → hardware progression
Summary and Future Work

Summary

- we developed fully threaded LibNBC for IB → high overhead
- tested implementation with RT threads → lower overhead (better than theory?)
- implemented new work-based benchmarking metric → realistic (high) overhead

Conclusions and Future Work

- threaded implementation makes sense with spare cores
- very tricky without spare cores → manual again?
- investigate other options → signalled progression (not safe/realistic!)
  → OS involvement (opposite to OS bypass)
  → hardware progression