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

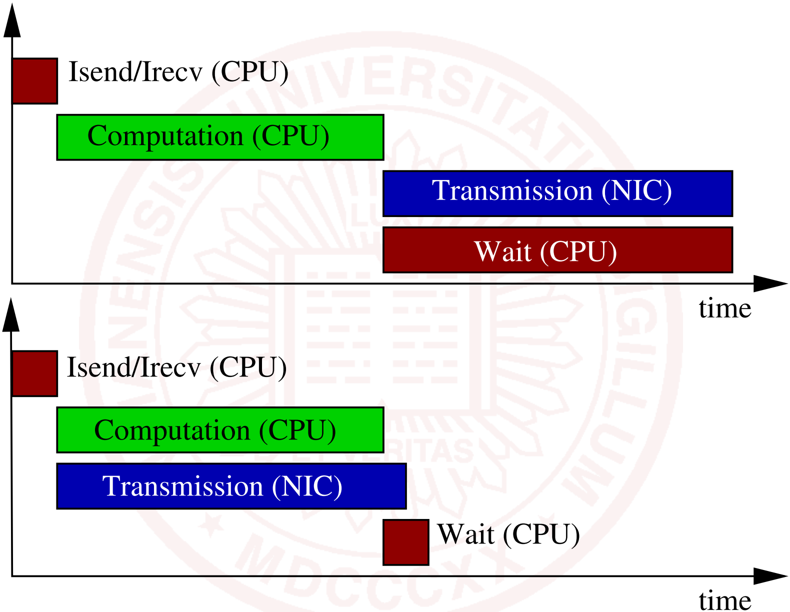
Non-blocking Interfaces

- 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!



Non-blocking Middleware Implementation

Non-blocking Send/Receive

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

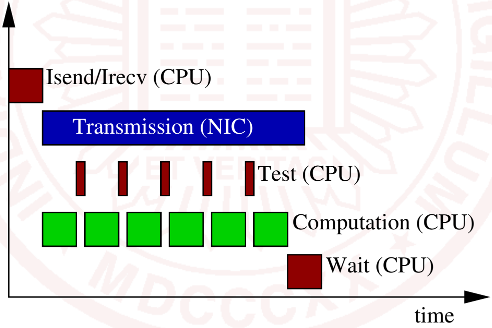
Non-blocking Collectives

- 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: $N = \lfloor \frac{size}{interval} \rfloor + 1$



Progression Strategies II/II

Hardware-based Progression

- need to run full protocol in NIC
- complicated to implement
- full benefits to the user
- mostly not supported

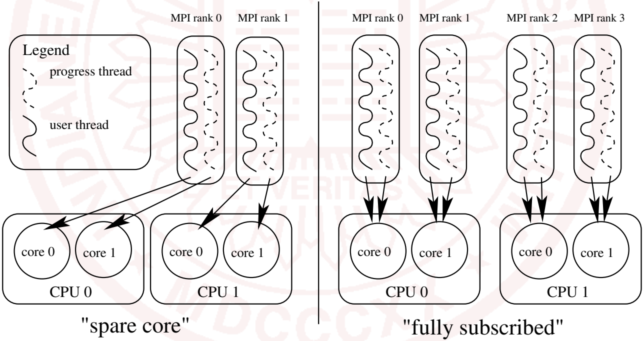
Threaded Progression

- asynchronous threads
- often stated as “silver bullet” but not widely used (?)
- problem with manual progresssion: “fire at the right time”
- threads could solve this problem (woken up correctly)
- could enable fully asynchronous progression
- OS influence (scheduler) is significant

Threading Configurations

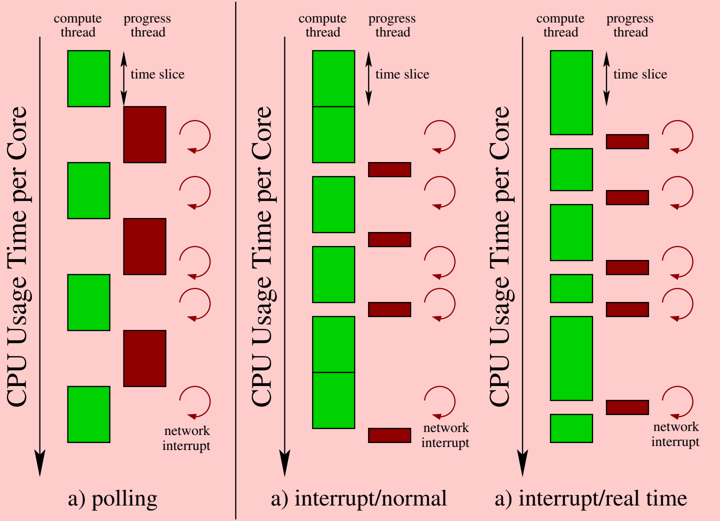
Spare Core vs. Fully Subscribed

- 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



Implementation Possibilities

Polling vs. Interrupt vs. Real Time



Non-blocking Collectives and InfiniBand

Issues with Non-blocking Collectives

- NBCs introduce dependencies
  $\rightarrow$ 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 4ms, latency 3μs)

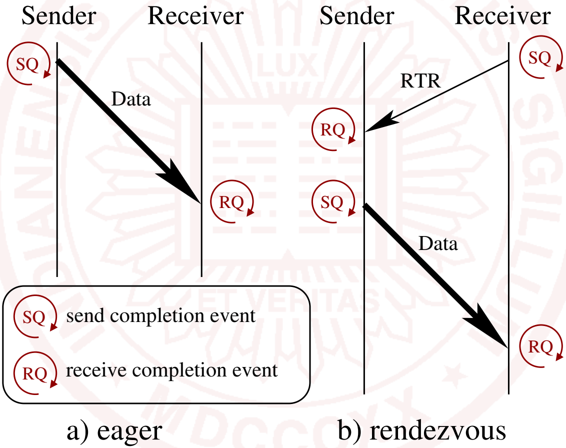
Real-Time Threads in Linux

- highest priority
- scheduled immediately
- no preemption

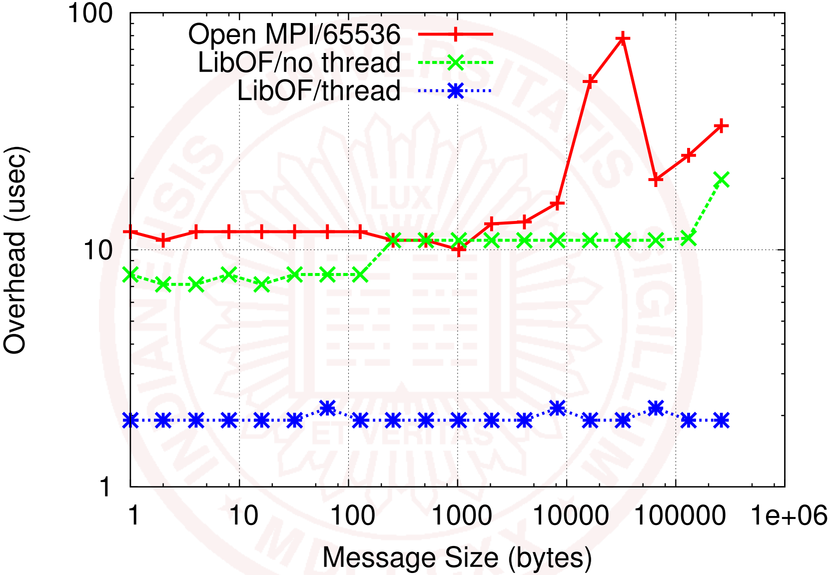
Overhead of Threading

Real time vs. Normal

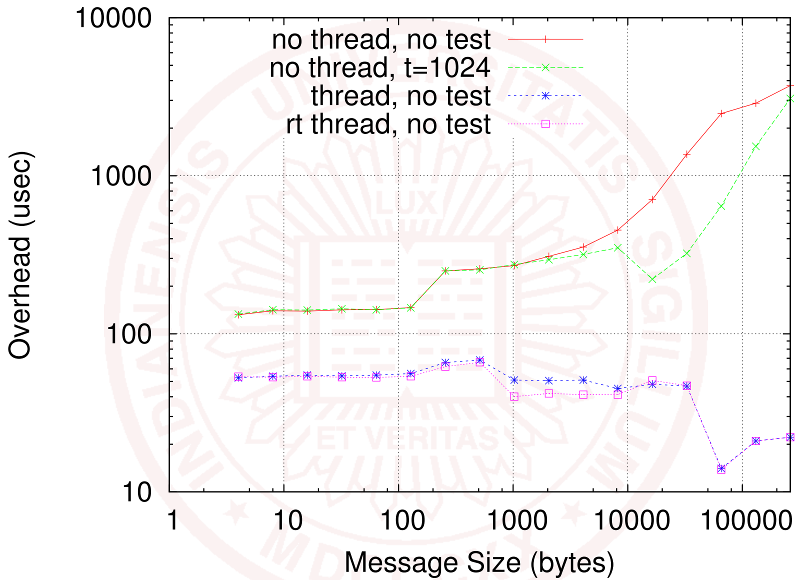
- normal threads: interrupts coalesc, low (no) progression
- RT-threads: each interrupt pays full overhead



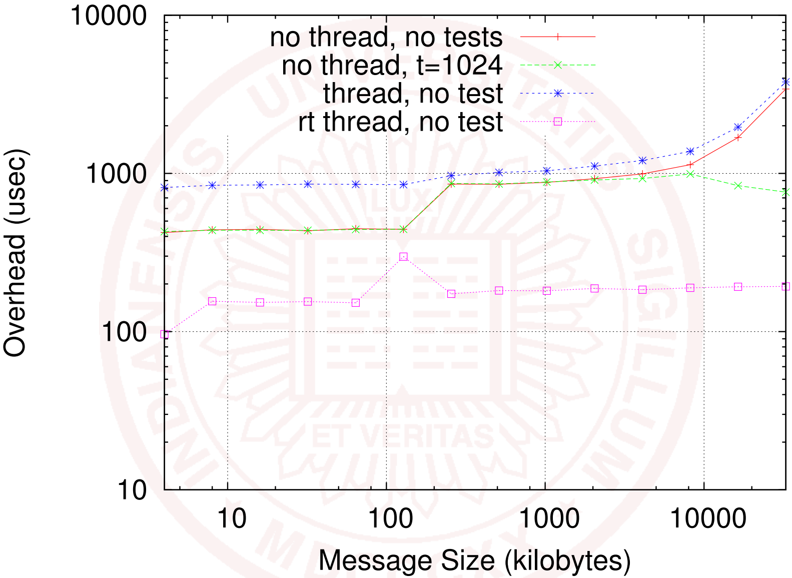
Point-to-point overhead



NBC_Iallreduce Overhead on 32 Nodes - spare core



NBC_Iallreduce Overhead on 32 Nodes - fully subscribed



Nice Results ... but wrong Metric

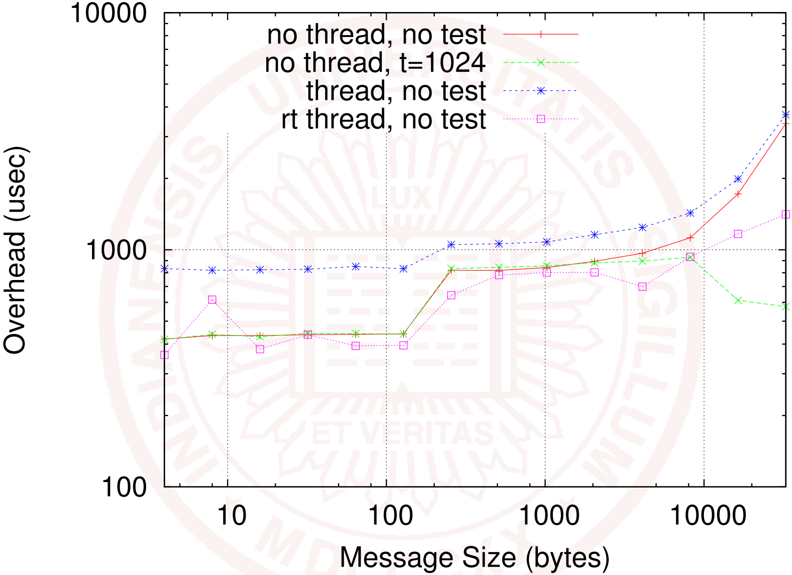
Wrong Metric?

- 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



Summary and Future Work

Summary

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

Conclusions and Future Work

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

Summary and Future Work

Summary

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

Conclusions and Future Work

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