Design of Kernel-level Asynchronous Collective Communication

University of Tokyo
Akihiro Nomura, Yutaka Ishikawa

Background: MPI Non-blocking Collective Communications

- NBC = Non-blocking + Collective
  - Exploit communication – computation overlap
  - Do complicated communications easily and efficiently
- NBC will be introduced in upcoming MPI 3.0
  - In MPI 2.2, users have to implement collective routines by hand to do non-blocking collectives.
    - In HPL(High-performance Linpack),
      6 implementations of non-blocking bcast are provided
- Existing implementation: LibNBC [Hoefler et al, 06-]
  - Same APIs as MPI 3.0
  - POSIX pthread is used in the implementation

Background: Why threads are needed? (1/3)

- Progression of collective communications
  - Collective communication consists of many point-to-point(P2P) (non-blocking) communications.
  - P2P communications have data dependencies.
    - E.g. send a data AFTER receiving them
  - Progression resolves these dependencies and issues all executable P2Ps as soon as possible.
- How to do progression in existing methods?
  - Call progression explicitly (e.g. HPL's Ibcasts)
  - Communication thread (e.g. LibNBC)

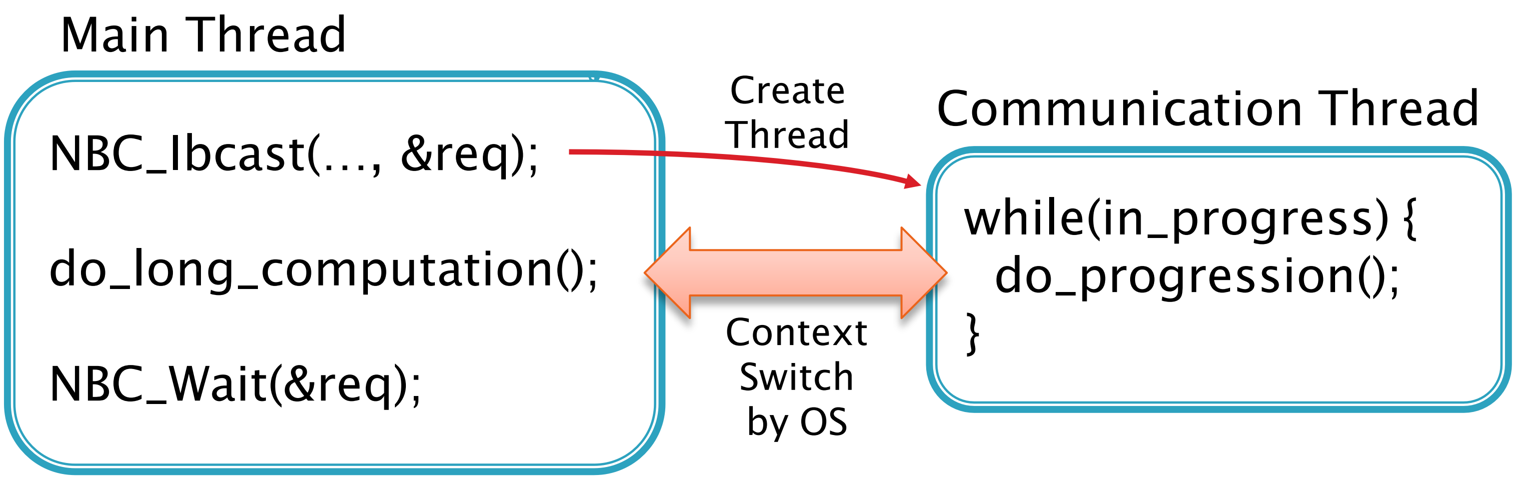
Background: Why threads are needed? (2/3)

- Call progression explicitly
  - HPL's Ibcasts for example
  - MPI users have to call progression routine in MPI library periodically by calling MPI_Test etc.
  - If users don't call progression, non-blocking collective doesn't progress.
  - If users call progression too frequently, CPU time cost of needless MPI_Tests becomes bigger.

```
init_for_collective();
while (1) {
  do_small_computation();
  do_progression();
  if (test_for_collective()) {
    do_rest_of_computation();
    break;
  }
  if (no_work_left()) {
    wait_for_collective();
    break;
  }
}
```

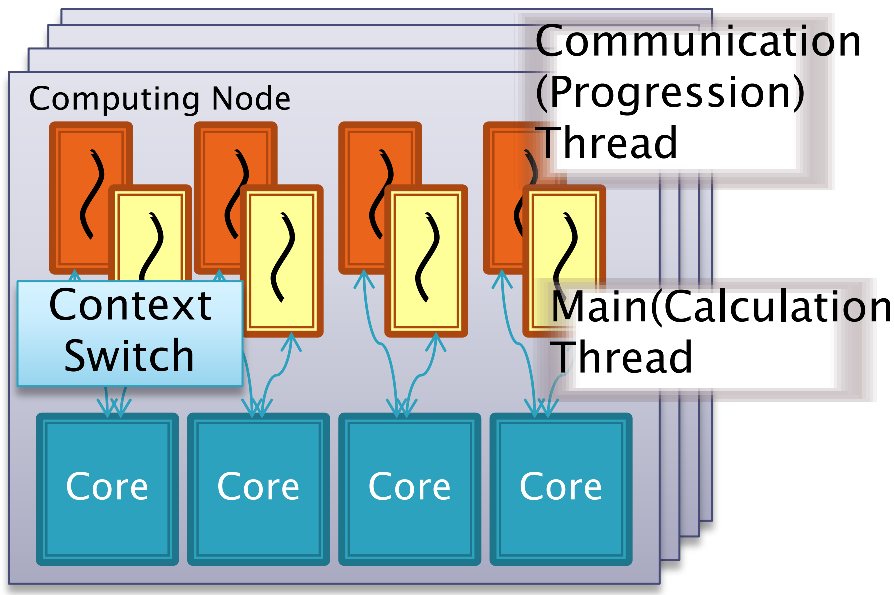
Background: Why threads are needed? (3/3)

- Communication thread (e.g. LibNBC)
  - Creates a thread to perform progression
  - No explicit call required



Issue:
Why thread is problematic?

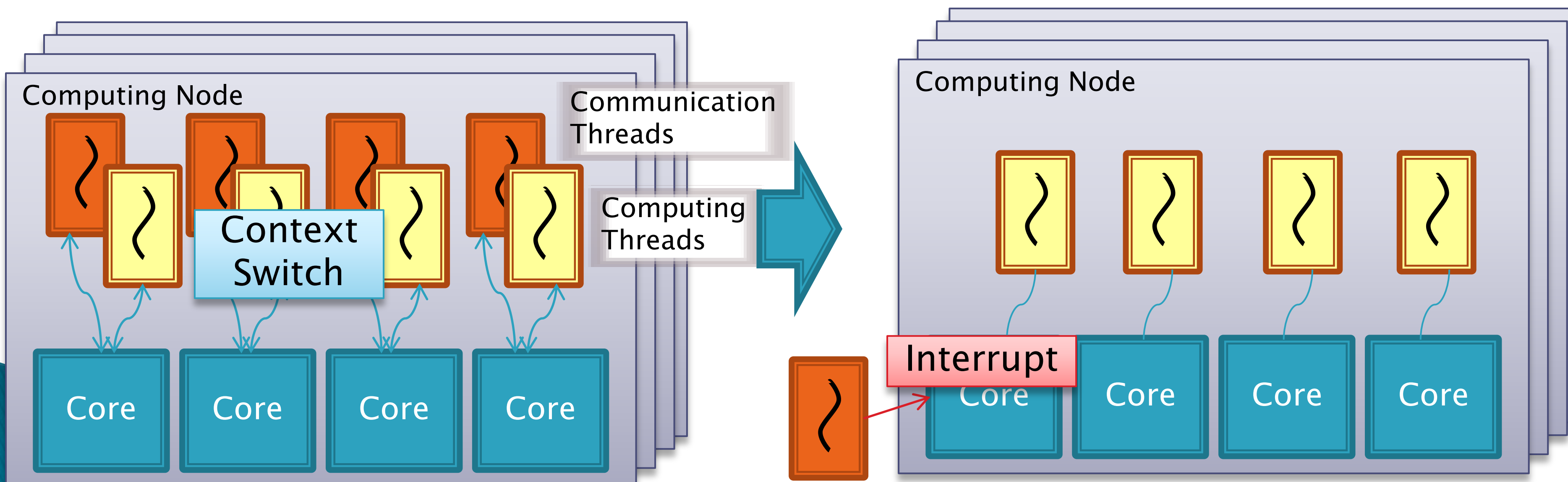
- Gap between MPI users and implementers
  - MPI user usually assumes all CPU cores can be used to calculation
    - User will create 1 process per core.
  - Progression thread is required to implement non-blocking collectives
- In this situation…
  - #threads exceeds #cores
  - Threads steals cores each other
    - Context switching cost
    - Context switch timing may not be optimal



Proposed Method: KACC

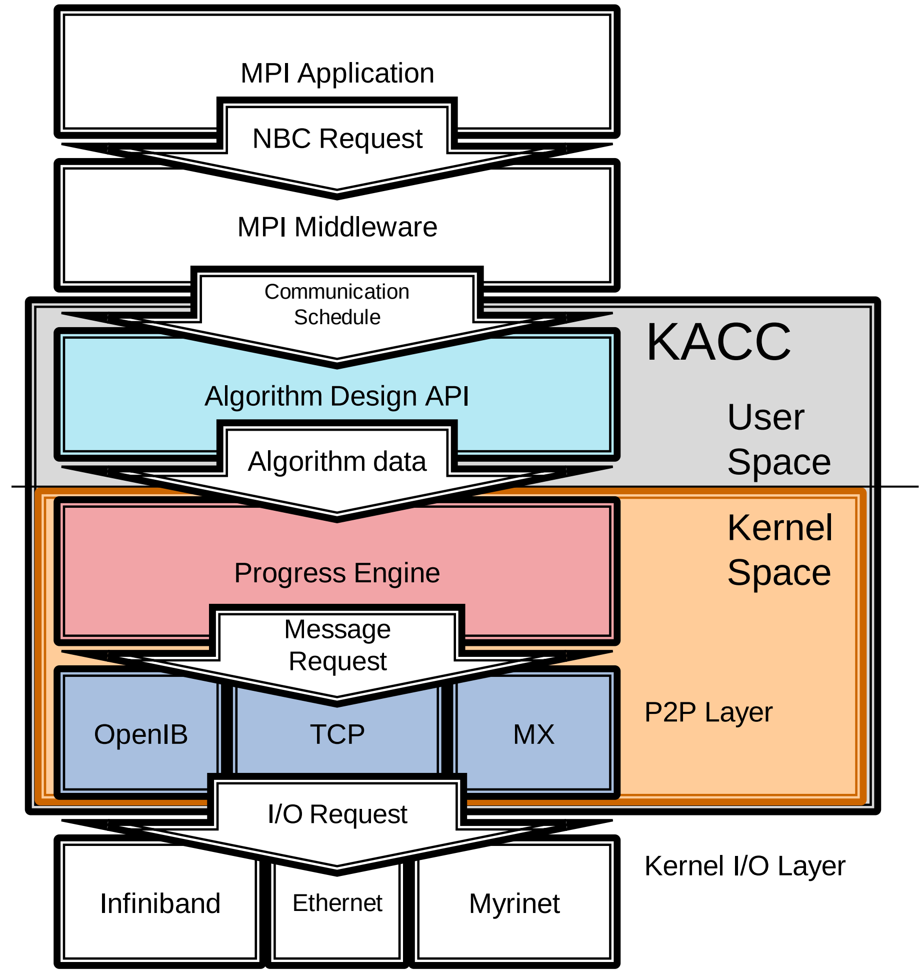
(Kernel-level Asynchronous Collective Communications)

- Progression Engine(PE)
  - Progression is implemented as kernel-mode routines to avoid cost of using threads.
  - PE is invoked by network interrupt handler
  - PE does not have user-mode contexts (memory etc)

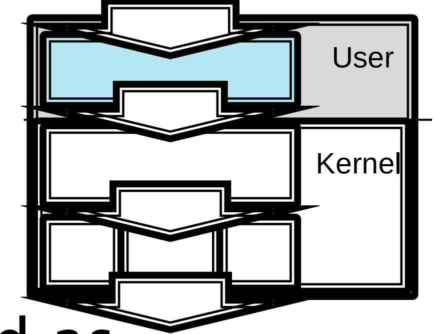


Design and Implementation Overview

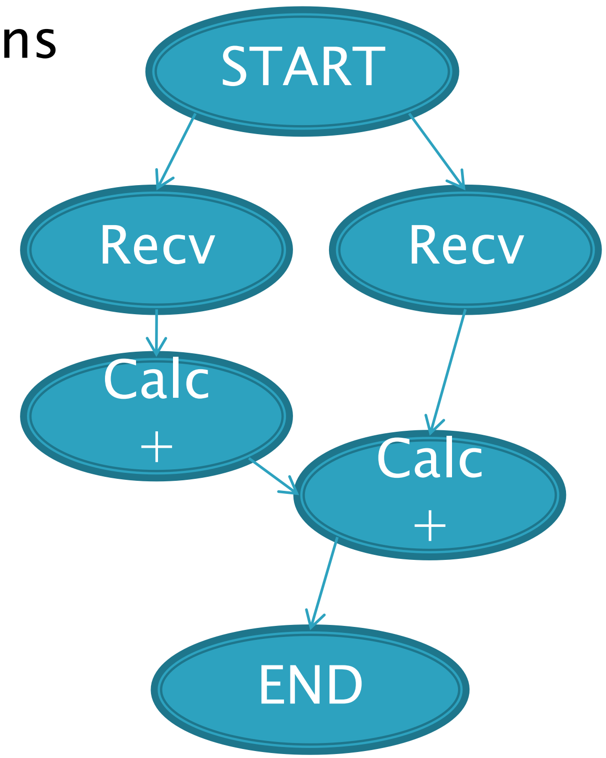
- KACC consists of 3 Layers
  - Algorithm Design
  - Progress Engine
  - P2P Routines
- Implemented on Linux Kernel 2.6
  - As User-level library
  - As OS kernel module



Collective Algorithm Design API



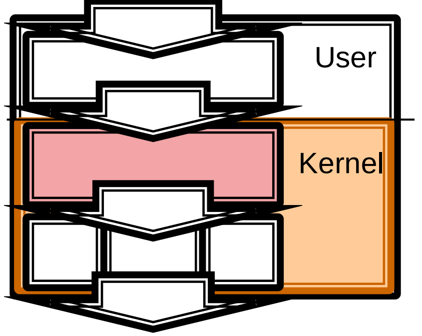
- Collective communications can be described as DAG(Directed Acyclic Graph) [1, 2]
  - Nodes: Communications and Calculations
  - Edges: Dependencies
- Make DAG structures on shared memory with kernel module.
  - MakeSendNode(), ConnectNode()…
  - To avoid passing executables to kernel directly (security issue)
- Call Progress Engine to execute/query algorithms
  - IssueCAD(), QueryCAD()
  - Communication with kernel module is done by using shared memory and system calls



Example of MPI_Reduce on the root node

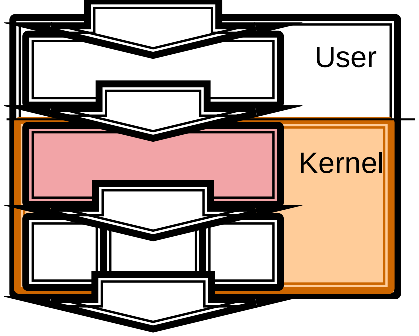
[1]: Hoefler et al, 2007
[2]: MPIplans in MPI-Forum Wiki

Progress Engine (1/2)

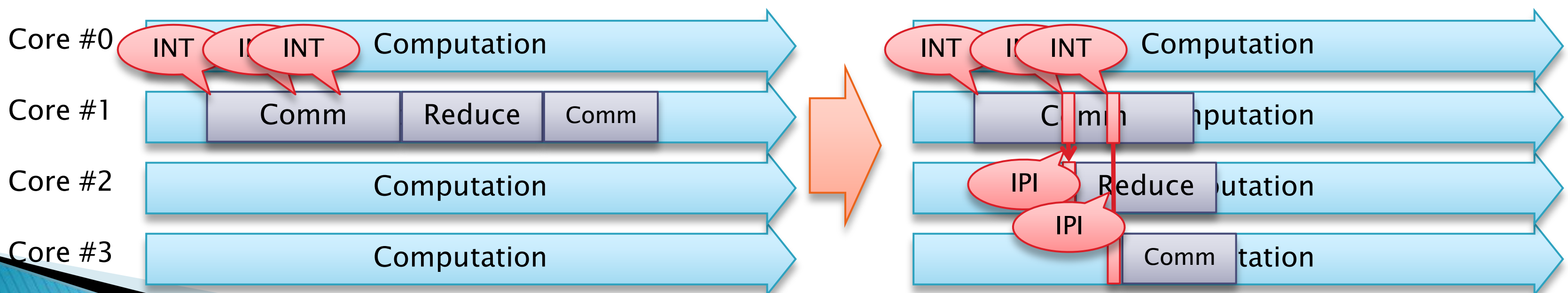


- Process progression
  - = Issue P2P as soon as possible
- Issue communications by requests from other layers
  - From Algorithm Design: Start collective communications (by system call)
  - From P2P Layer: Ready for communication / Complete communication (by Interrupt handler)

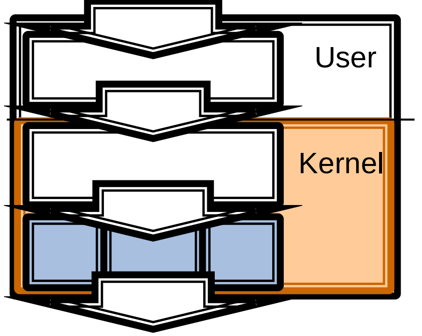
Progress Engine (2/2)



- Implementation
  - Implemented using Linux tasklet
  - Does not have process context (VM address space)
    - Drawback: all data have to be stuck to physical memory before starting collective communication
  - Executed at the end of system calls and interrupt handlers
  - Requires Load balancing using IPI (Inter-processor Interrupt)
    - Tasklet runs on the same CPU as its invoker (=interrupt)
    - Network interrupts is concentrated on one specific core to send packets efficiently



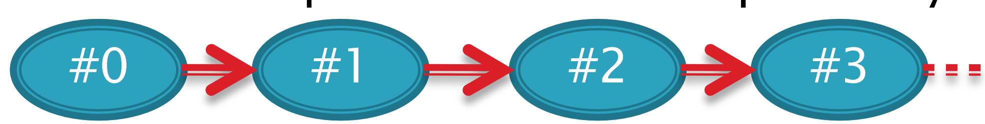
P2P Layer



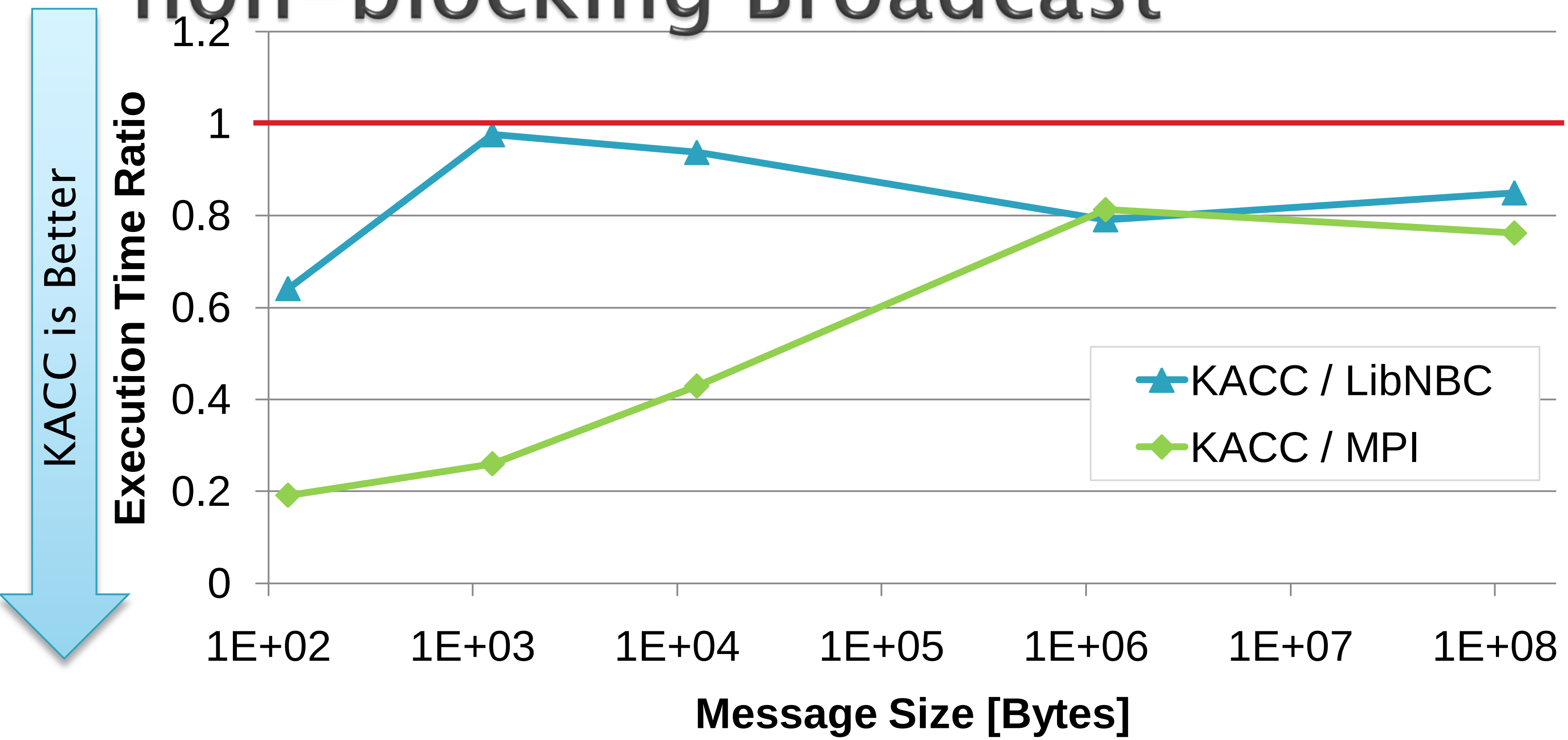
- Abstraction Layer for Progress Engine
  - Executes actual communication in non-blocking manner
  - Runs on Linux tasklet context
- API like MPI's non-blocking P2P (Isend/Irecv)
  - Completion is reported using callback routines
- Implemented non-blocking P2P on kernel-level TCP
  - TCP routine cannot sleep because tasklet doesn't have process context

Evaluation

- We compared execution time and CPU usage of following implementations of non-blocking broadcasts
  - KACC – Proposed Method
  - LibNBC – Using Thread for Progression
  - MPI – Calling Progression Explicitly and Periodically
- CPU usage is calculated using following formula
  - Usage[%] = (1 – Flops(Comm) / Flops(Idle)) x 100
- Environment
  - Dual-core 2Ghz Opteron x 2 (4 core / node)
  - 8 node cluster, connected with 1Gbps Ethernet (Broadcom)
  - Linux kernel 2.6.18 (RedHat EL5)
- Algorithm
  - Pipeline broadcast on 32 MPI process
    - Divide messages into small piece and send sequentially

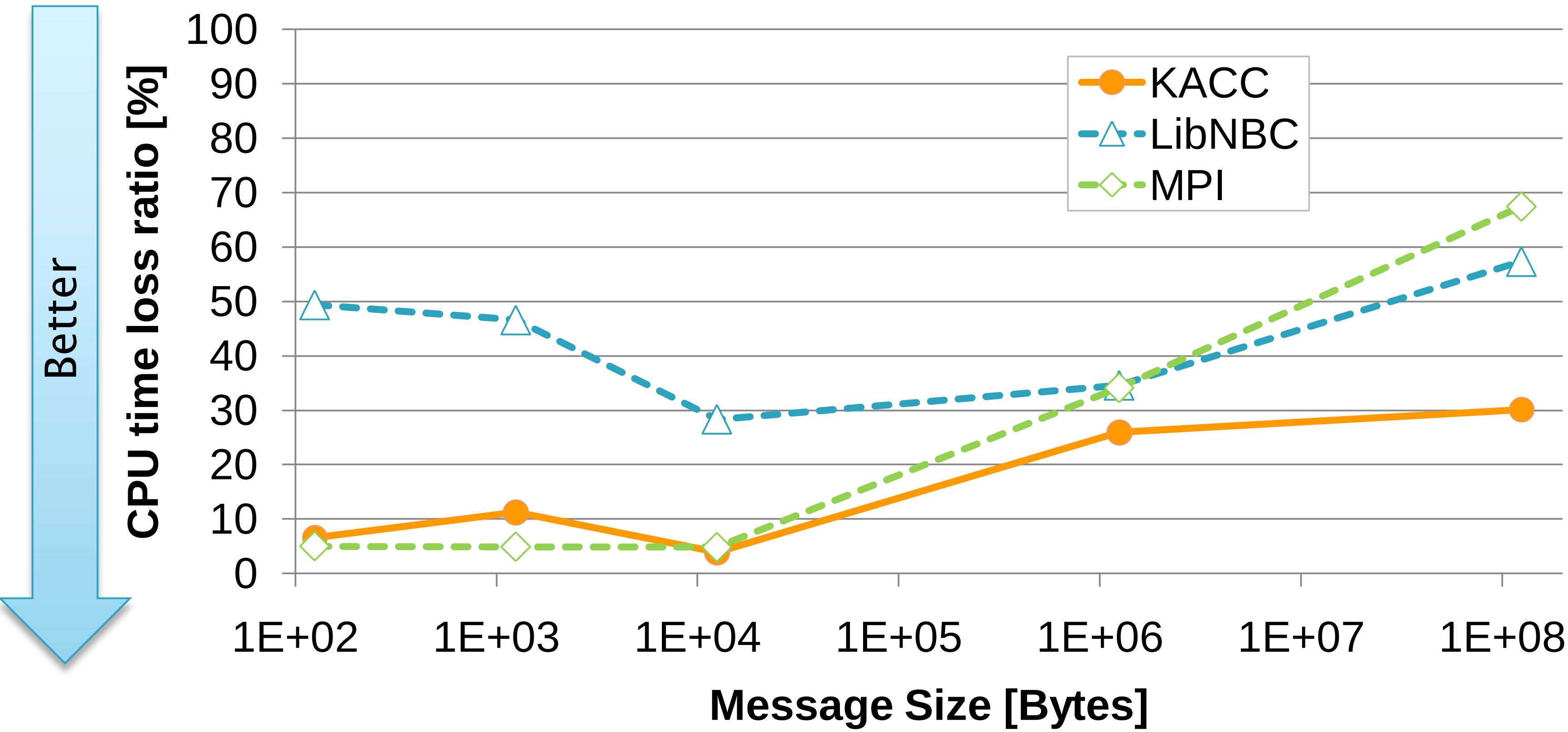


Execution time of non-blocking Broadcast



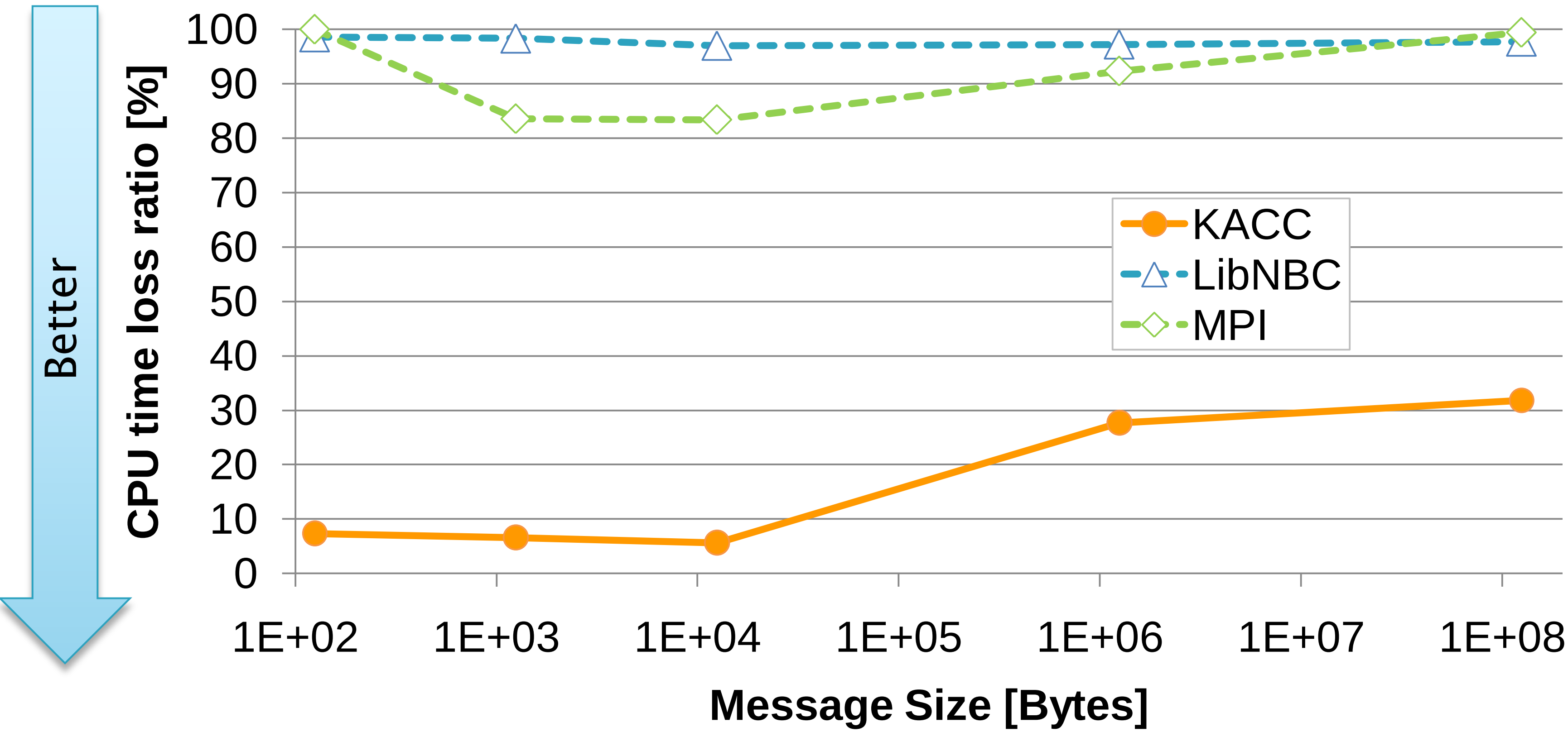
- KACC is faster than existing methods
- If message size is small, KACC slows down due to overheads

CPU Usage



- KACC consumes less CPU time in the same algorithm

CPU Usage (on more frequent Testing)



- If user calls Test/Progression more frequently, most of CPU time is spent on communication under the existing methods.
- On KACC, CPU time consumption ratio is still small.

Conclusions

- We have proposed KACC facility
  - A new method to implement non-blocking collective communications
  - Use kernel's interrupt context to avoid context switching costs of threads
- We evaluated KACC
  - KACC is 21% faster than LibNBC
  - KACC consumes at least 33% less CPU time than LibNBC
- Future work
  - Provide a way to do user-defined operations.
    - Application's signal handler? VM code w/ verification?
  - Provide other P2P layers than TCP
  - Improve performance

Thank you