Group Operation Assembly Language

- A Flexible Way to Express Collective Communication -

Torsten Hoefler[1], Christian Siebert[2],
Andrew Lumsdaine[1]

[1]Open Systems Lab
Indiana University, Bloomington

[2]NEC Laboratories Europe
Sankt Augustin, Germany

09/25/09
ICPP 2009
Vienna, Austria

Introduction

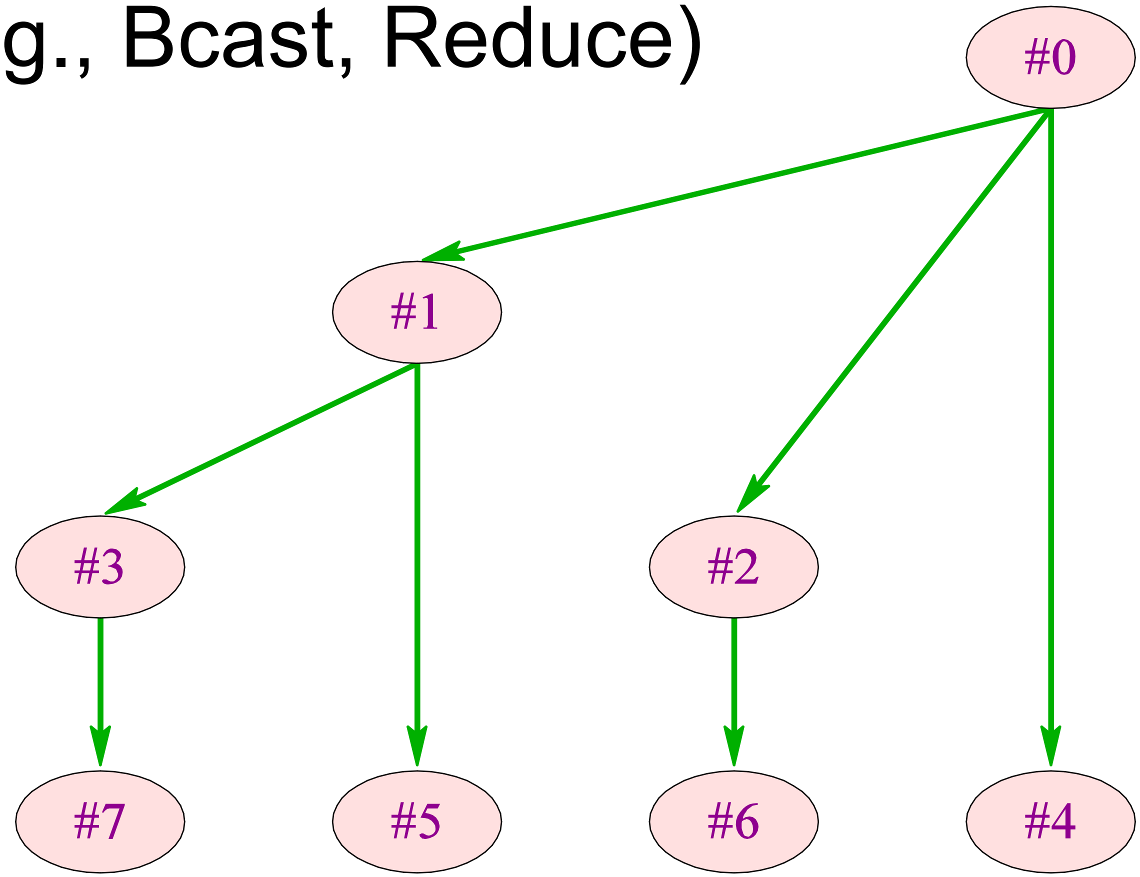
- MPI as de-facto standard in parallel processing
- Collective operations are integral part of MPI
- Large body of research on advanced algorithms
- Multiple implementations in MPI libraries:
  - e.g., MPICH2, MVAPICH, Open MPI
- "Group Operations" are also used in other environments (e.g., MRNet, Multicast)

Motivation

- Group Operations are a general concept
  - e.g., used in MPI, UPC, MRNet
- Nonblocking Collective operations arrived
  - NBC will be in MPI 3.0 (or 2.3?)
- Most implementations are hard-coded
  - Control-flow as static branches in source-code
  - Requires considerable hand-tuning
  - User-defined (sparse) collective operations (?)
- Hardware offload and NBC

Broadcast Tree Examples

- Binomial trees used in many small-message collectives (e.g., Bcast, Reduce)



Our Goals

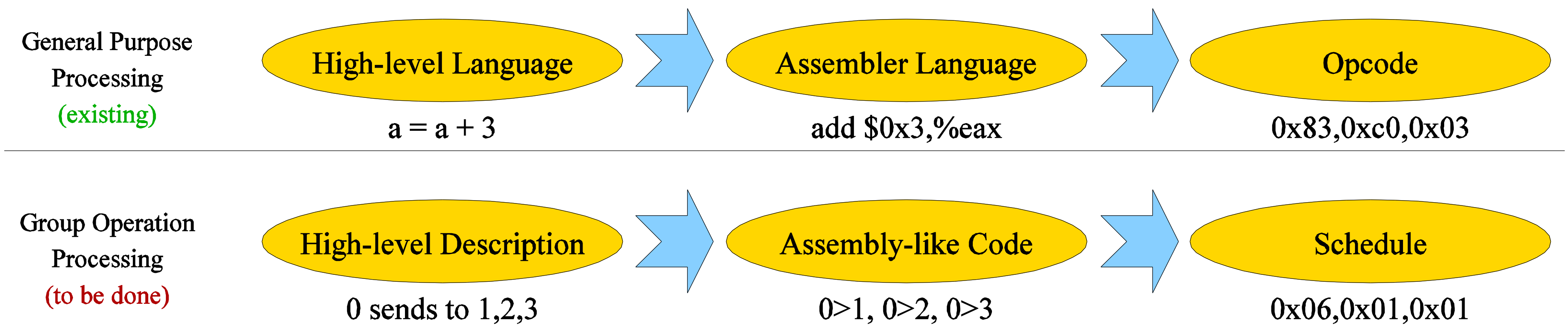
- Define a minimal language to express collective communication to enable:
  - efficient representation for offload
  - fast and simple execution on slow PEs
  - good specification of advanced algorithms
  - execution on resource-constrained environments (NIC)
  - (automatic) transformational optimizations

Abstracting

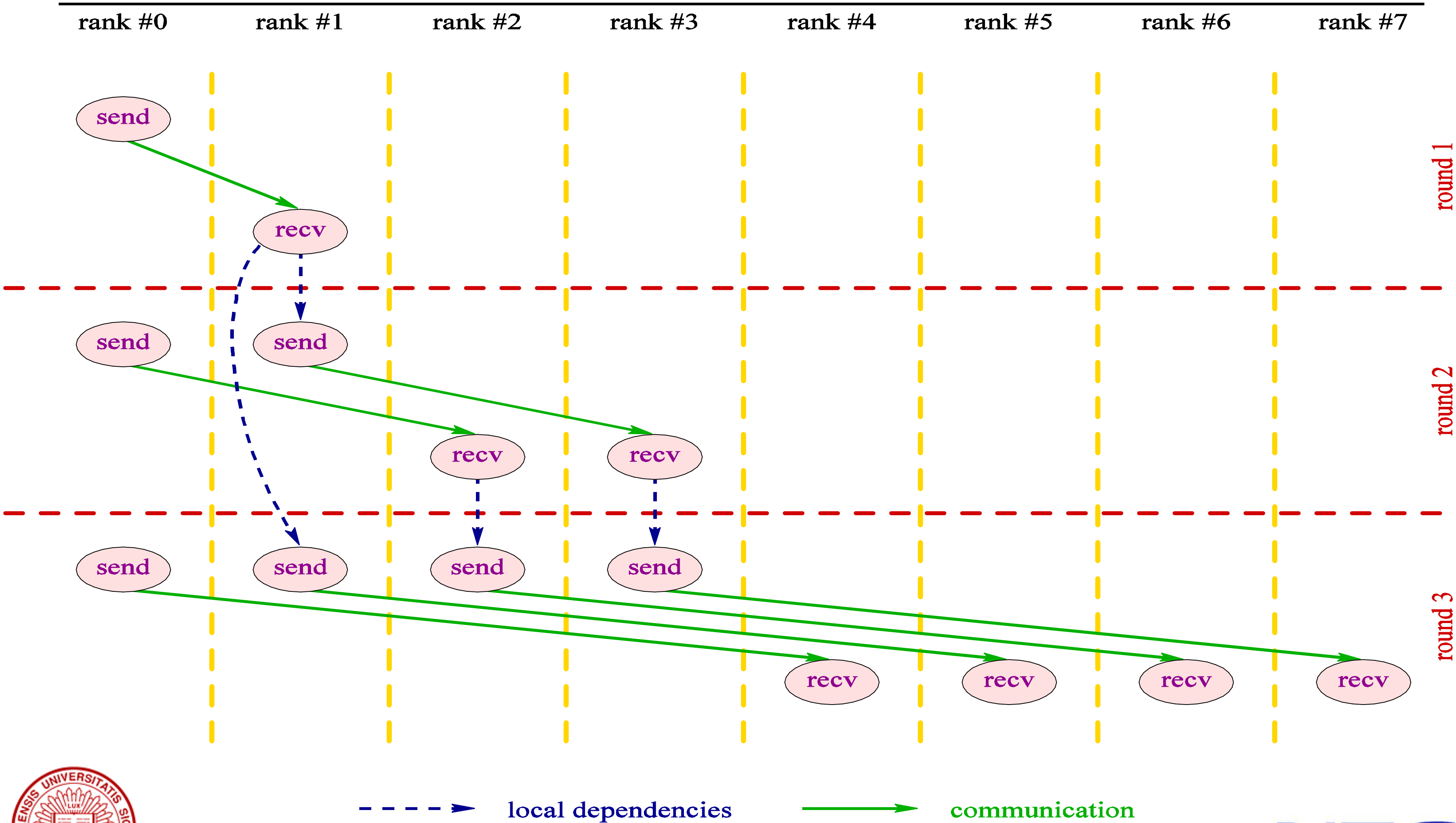
- What is the minimal set of operations needed to perform any collective algorithm?
- Theorem 1 states that send, receive and (local) dependencies are sufficient to model any collective algorithm
  - allows concise definition!
- Theorem 2 states that the order requirement is relative to each single operation
  - allows optimized/adaptive execution!

Group Operation Assembly Language

- Very low-level specification (compilation target)
  - cf. RISC assembler code
- Translated into a machine-dependent form
  - cf. RISC bytecode



A Binomial Tree Example



GOAL Language Interface

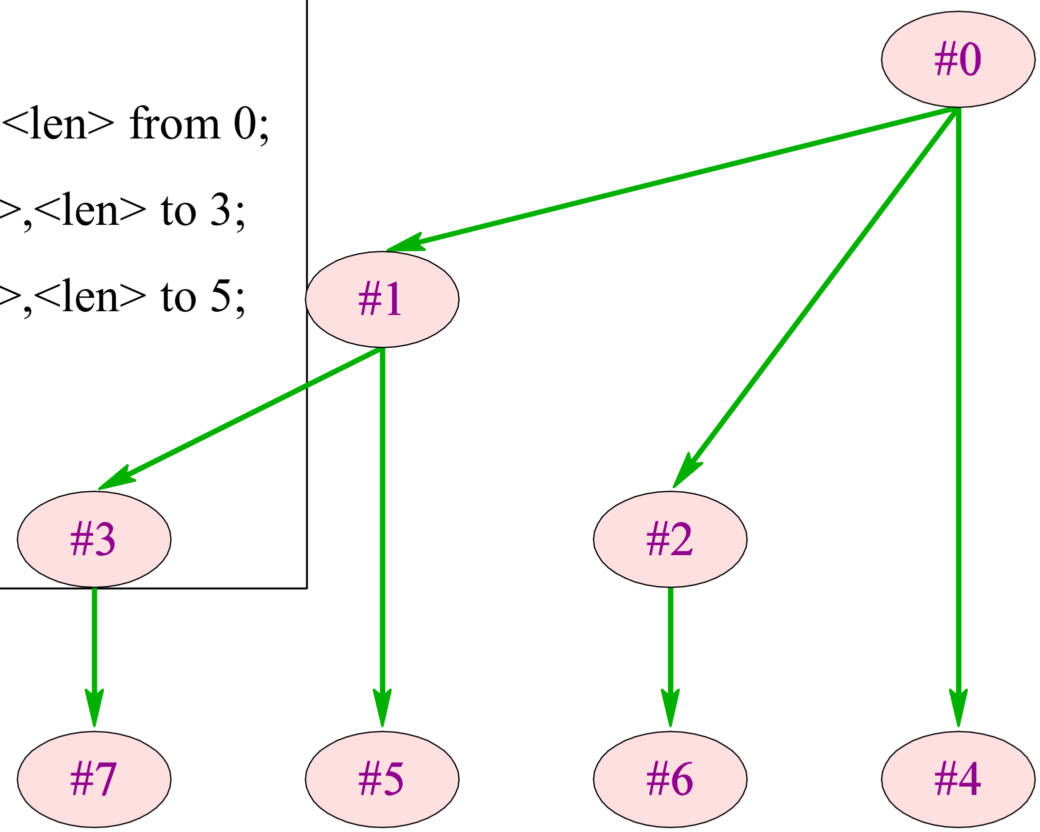
- GOAL Language interface (Bcast example):

```
rank #0 {
  send <msg>,<len> to 1;
  send <msg>,<len> to 2;
  send <msg>,<len> to 4;
}
```

```
rank #1 {
  r: recv <msg>,<len> from 0;
  s1: send <msg>,<len> to 3;
  s2: send <msg>,<len> to 5;
  requ s1 -> r;
  requ s2 -> r;
}
```

```
rank #5 {
  recv <msg>,<len> from 1;
}
```

• • •

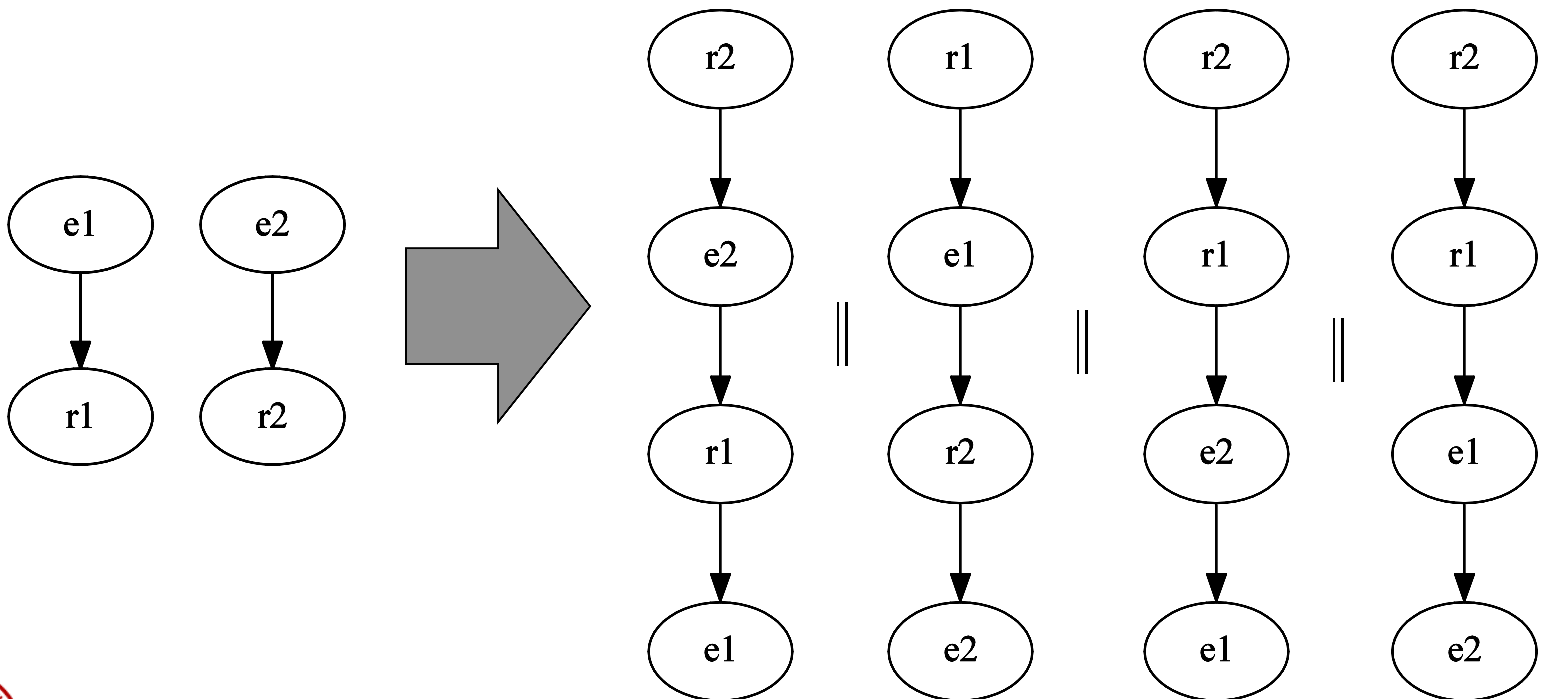


Group Operation Assembly Language

- Alternative schedule creation at runtime:
  - Library interface:
    - gop=GOAL_Create()
    - id=GOAL_Send(sched, buf, size, dest)
    - id=GOAL_Recv(sched, buf, size, dest)
    - GOAL_Exec(sched, func, buf, size)
    - GOAL_Requ(sched, src_id, tgt_id)
    - sched=GOAL_Compile(gop)
- Internal representation reflects a dependency DAG
  - enables transformational optimizations

Optimization possibilities

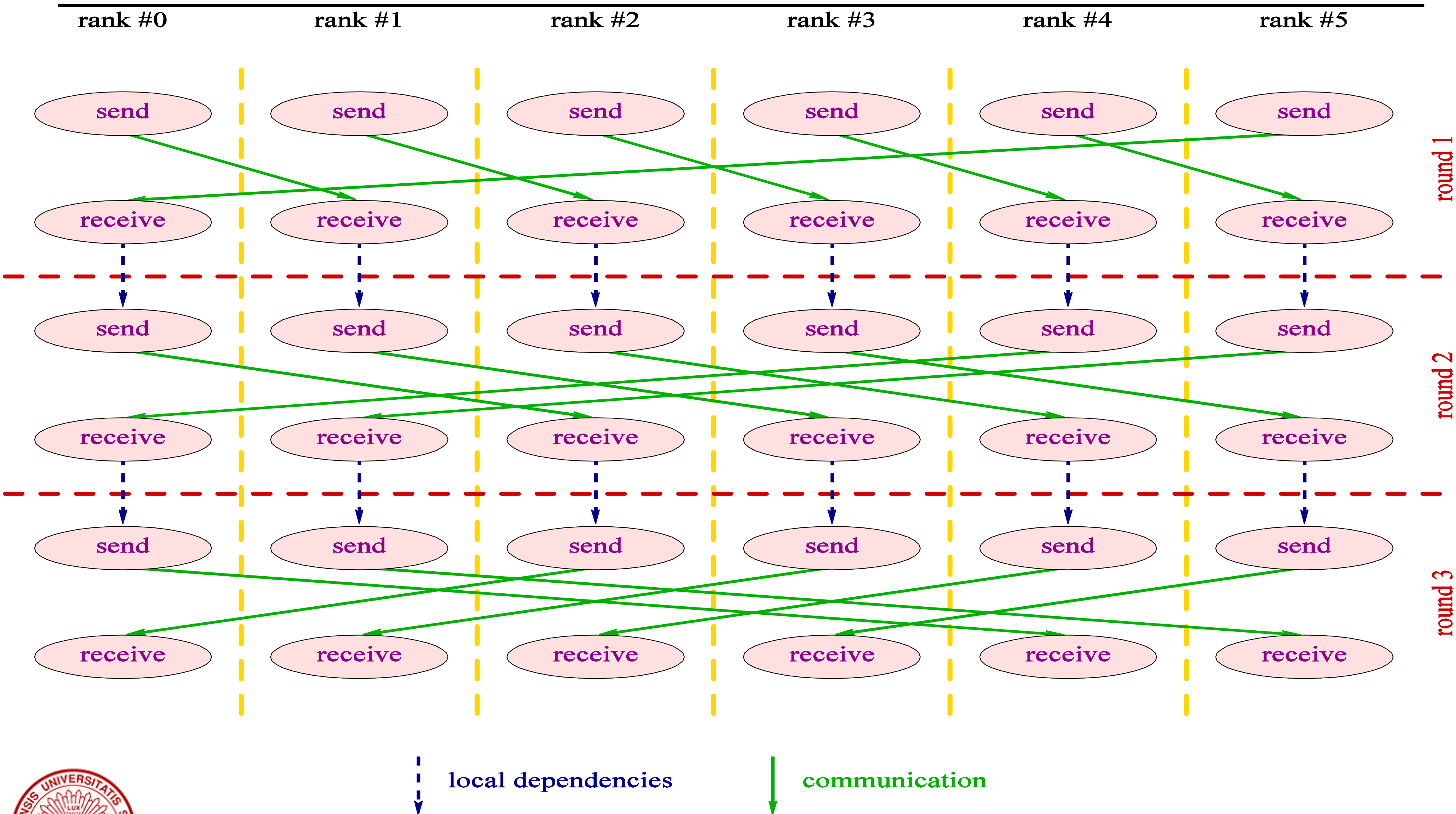
- Adaptive execution
  - Possible to consider process arrival pattern
  - independent ops: sent to ready hosts first



Optimization Possibilities (cont.)

- Parallel execution
  - Schedule (DAG) allows for parallel execution
    - Multiple parallel NICs
  - Same scheduling issues as for multicore task libraries (TBB, Cilk, OpenMP 3.0)
- Static schedule (compiler) optimization
  - e.g., architecture-dependent pipelining
- Scheduler runs in thread or hardware
  - Offload to spare CPU core
  - Offload to NIC (same GOAL specification)

Advanced Example - Dissemination

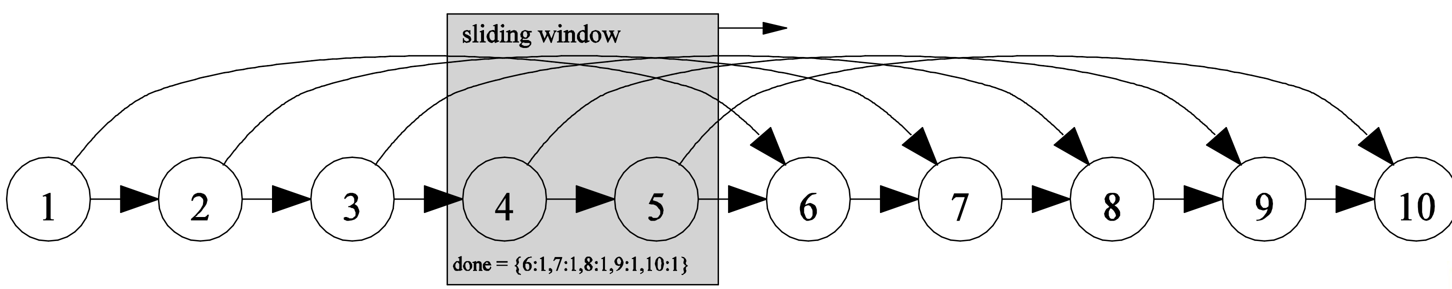


Schedule Details

- Result of GOAL assembly
  - Optimized for each architecture
- Should not lose flexibility
  - Represents dependency/execution graph
- Our machine-dependent representation:
  - We propose binary schedule
  - Linear memory layout (cache/pre-fetch friendly)
  - Executor only 98 SLOC C code in LibNBC
  - Compression possible (not in this work)

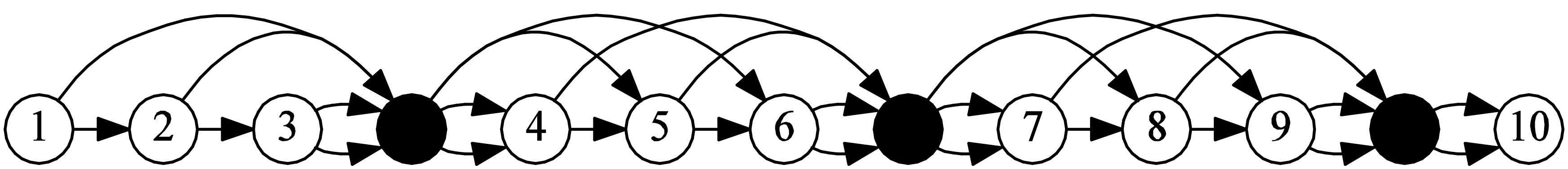
Execution Constraints

- How much memory do we need to execute a schedule?
  - We can use a sliding window (hold only parts of the schedule in a scratchpad memory (NIC))
  - Theorem 3: A schedule of length N can be executed with $\mathcal{O}(N)$ additional memory using a constant-size window.
  - it's actually also $\Omega(N) \rightarrow \Theta(N)$ see:



Execution Constraints (contd.)

- $\Omega(N)$ memory consumption is infeasible
  - SRAM on a NIC is expensive!
- Solution: introduce additional dependencies
  - BUT: additional dependencies ⇨ serialization
- Theorem 4: Each schedule can be executed in $\mathcal{O}(1)$ memory, if dummy actions are added.



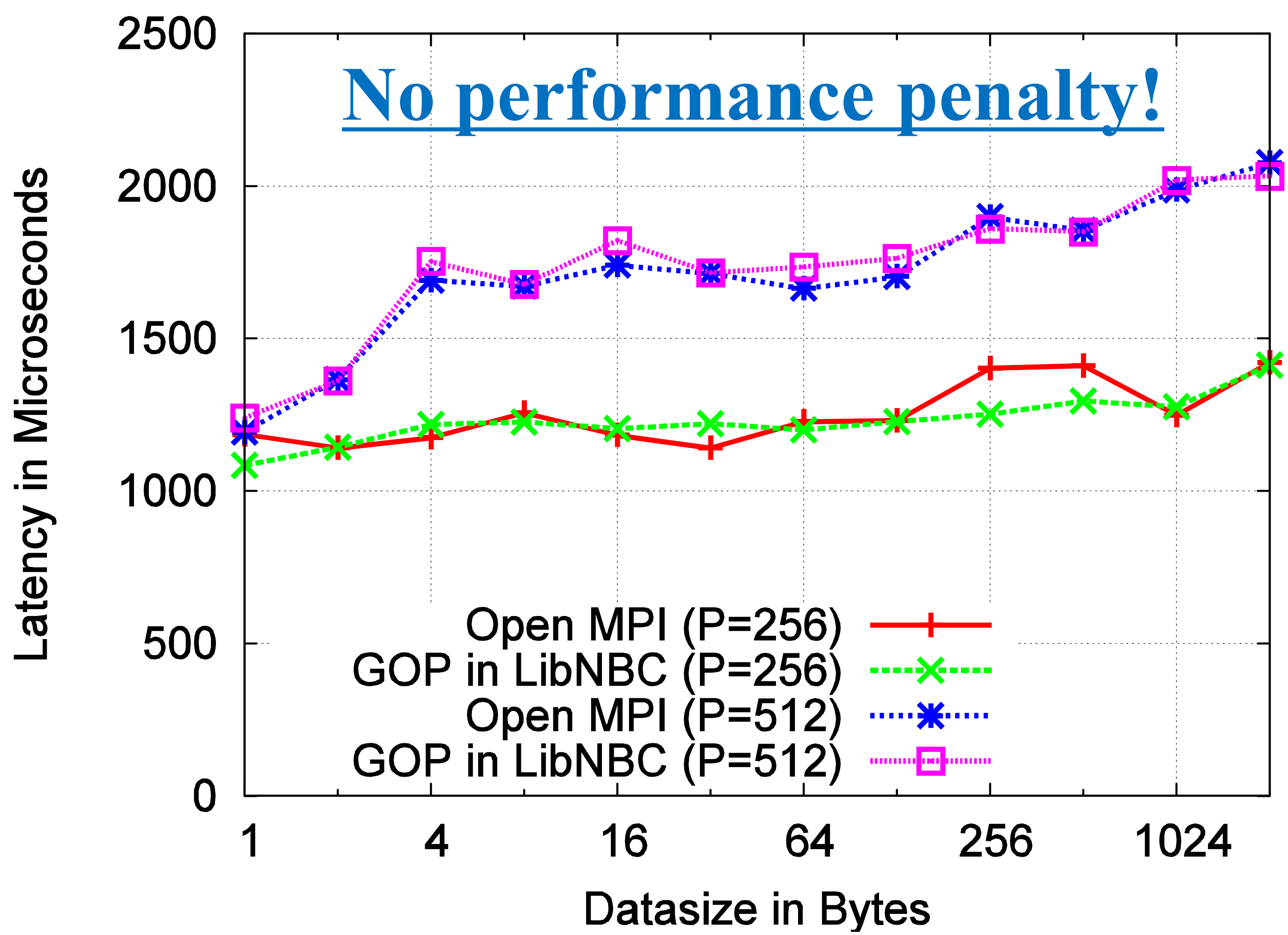
Implementation

- Ernest Rutherford: *"We don't have the money, so we have to think."*
  - no easy access to programmable NIC
  - working with Myricom on Myrinet
  - Mellanox seems to have a similar interface in it's next generation API
- We offloaded to a spare CPU core
  - threading model
  - replacing current implementation in LibNBC
  - less synchronicity than round-based scheme!

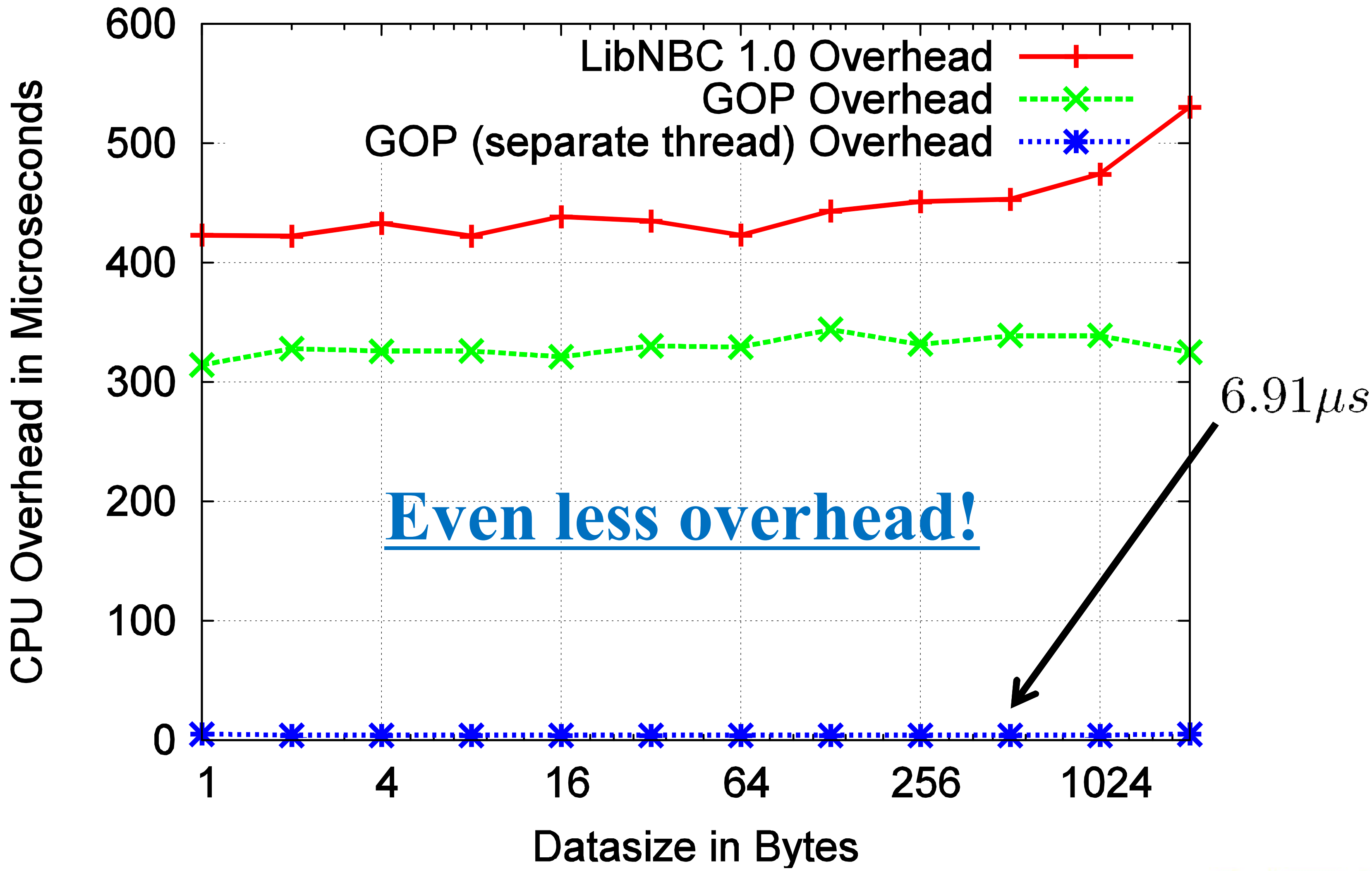
Test System

- Odin Cluster at Indiana University
  - 4x InfiniBand SDR
  - Single 288 port Mellanox switch
  - 128 nodes
  - 4 cores per node -> 512 cores
- Open MPI coll component “tuned”
  - version 1.3
- LibNBC 1.0 (with NBCBench 1.0)
  - OFED-optimized version (uses RDMA-W)

Blocking Collectives



Nonblocking Collectives



Conclusions

- Abstract definition of group communication
  - easy definition of (non-)blocking for offload
  - universal (implements all collectives)
  - small overhead, maximum asynchrony
- Enables compiler-based optimizations and dynamic scheduling
  - e.g., pipelining, coalescing, memory registration
- First step towards high-level communication expression

Future Work

- Investigate compiler optimizations
- Compress schedules (reduce resource needs)
- Implement scheduler on NICs