Towards Efficient MapReduce Using MPI

Torsten Hoefler[1], Andrew Lumsdaine[1], Jack Dongarra[2]

[1]Open Systems Lab
Indiana University Bloomington

[2]Dept. of Computer Science
University of Tennessee Knoxville

09/09/09
EuroPVM/MPI 2009
Helsinki, Finland

Motivation

- MapReduce as emerging programming framework
  - Original implementation on COTS clusters
  - Other architectures are explored (Cell, GPUs,…)
  - Traditional HPC platforms?
- Can MapReduce work over MPI?
  - Yes, but … we want it fast!
- What is MapReduce?
  - Similar to *functional* programming
    - Map = *map* (`std::transform()`)
    - Reduce = *fold* (`std::accumulate()`)

MapReduce in Detail

- The user defines two functions
  - map: $\mathcal{M} : (K_m \times V_m) \mapsto (K_r \times V_r)$
    - input key-value pairs: $(k, v)\ k \in K_m,\ v \in V_m$
    - output key-value pairs: $(g, w)\ g \in K_r,\ w \in V_r$
  - reduce: $\mathcal{R} : (K_r, V_r^N) \mapsto (K_r, V_r)$
    - input key $\in K_r$ and a list of values $\in V_r^N$
    - output key $\in K_r$ and a single value $\in V_r$
- The framework
  - accepts list $(K_m \times V_m)^N$
  - outputs result pairs (K_r, V_r)

Parallelization

- Map and Reduce are pure functions
  - no internal state and no side effects
  - application in arbitrary order!
- MapReduce done by the framework
  - can schedule map and reduce tasks
  - can restart map and reduce tasks (FT)
- No synchronization
  - implicit barrier between Map and Reduce

MapReduce Applications

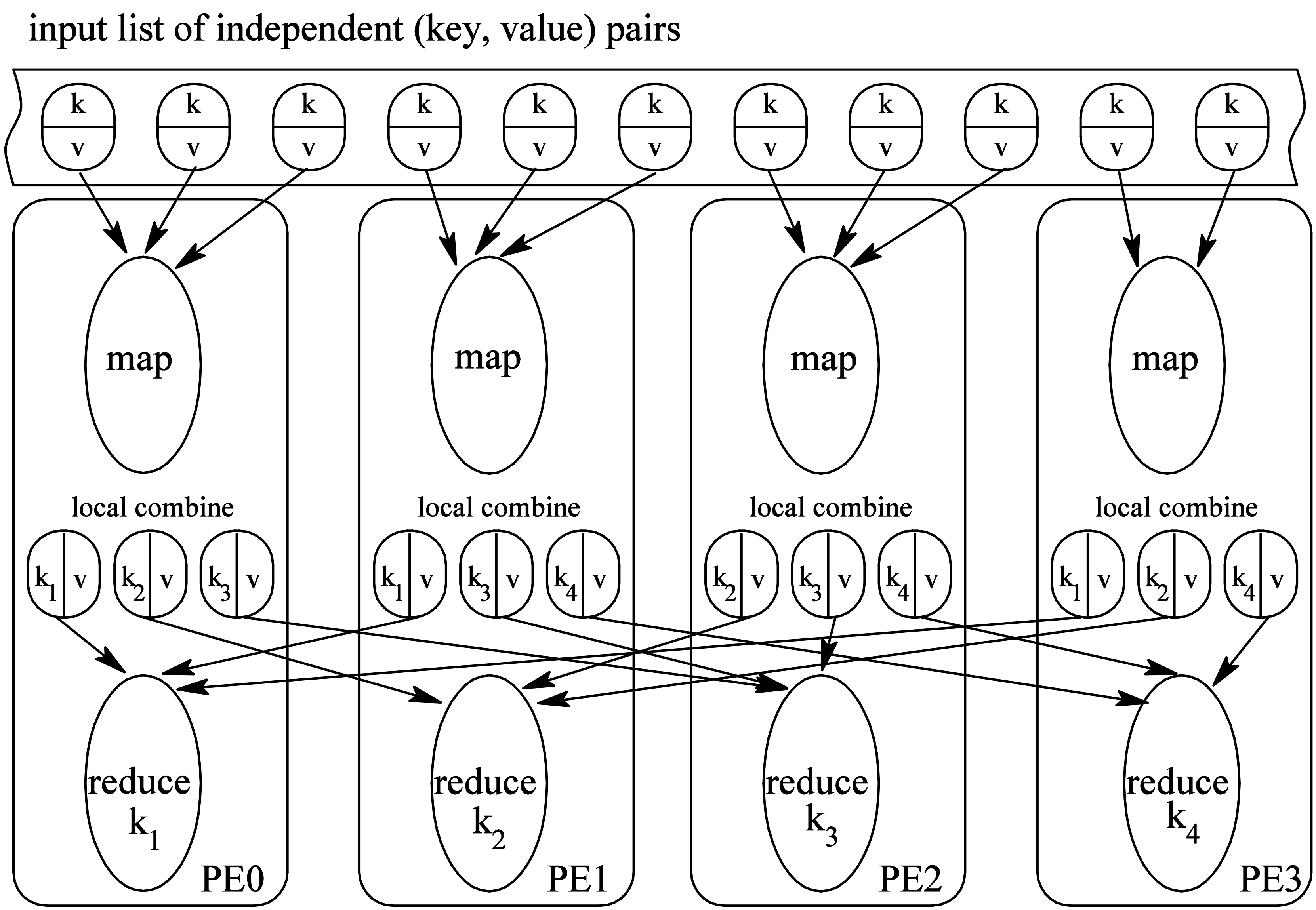
- Works well for several applications
  - sorting, counting, grep, graph transposition
  - Bellman Ford and Page Rank (iterative MR)
- MapReduce has complex requirements
  - express algorithms as Map and Reduce tasks
  - similar to functional programming
  - ignore:
    - scheduling and synchronization
    - data distribution
    - fault tolerance
    - monitoring

Communication Requirements

- two phases, three communication phases
  a) Read input for $\mathcal{M}$
     - read N input pairs: $\Omega(N)$
  b) Build input lists for $\mathcal{R}$
     - order pairs by keys and transfer to $\mathcal{R}$ tasks: $\mathcal{O}(N)$
  c) Output data of $\mathcal{R}$
     - usually negligible $\mathcal{O}(|K_r|)$

- two critical phases: a) and b)

All in one view



Parallelism limits

- map is massively parallel (only limited by N)
  - often $N \gg P$
  - data usually divided in chunks (e.g., 64 MiB)
  - either read from shared FS (e.g., GFS, S3, …)
  - or available on master process
- reduce needs input for a specific key
  - tasks can be mapped close to the data
  - worst-case is an irregular all-to-all
- we assume worst case:
  - input only on master and keys evenly distributed

An MPI implementation

- Straight-forward with point-to-point
  - not focus of this work

- MPI offers mechanisms to optimize:

1) Collective operations
   - optimized communication schemes
2) Overlapping communication and computation
   - requires good MPI library and network

An HPC-centric approach

- Example: word count
  - Map accepts text and vector of strings
  - Reduce accepts string and count
- Rank 0 as master, P-1 workers
- MPI_Scatter() to distribute input data
  - Map like standard MapReduce
- MPI_Reduce() to perform reduction
  - Reduce as user-defined operation
  - ➢ HPC-centric, orthogonal to simple implementation

Reduction in the MPI library

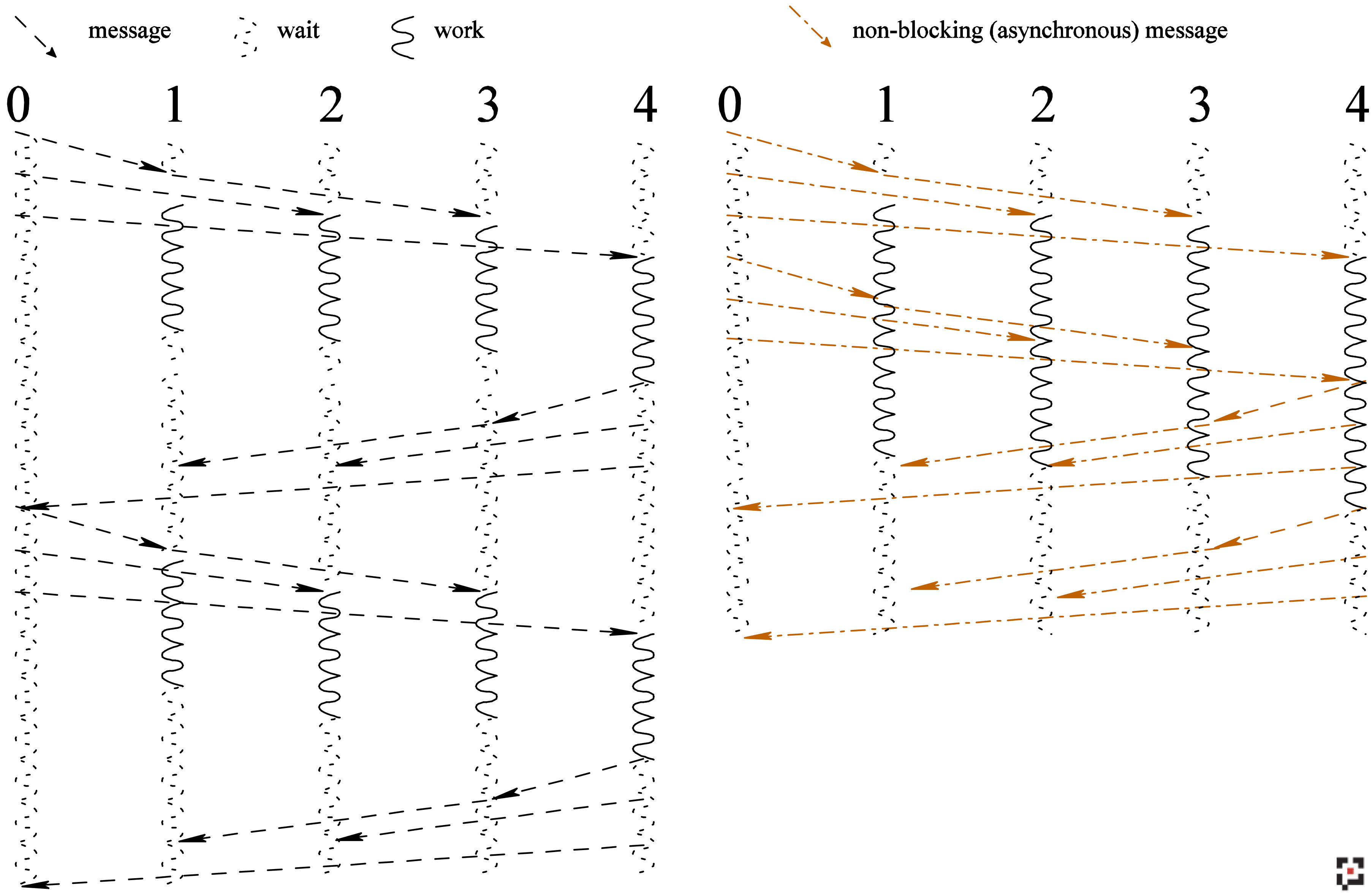
- Built-in or user-defined ops as $\mathcal{R}$
  - $\mathcal{R}$ must be associative (MPI ops are)
  - number of keys $|K_r|$ must be known by all procs
    - can be reduced locally (cf. *combiner*) MPI_Reduce_local
  - keys must have fixed size
  - identity element with respect to $\mathcal{R}$
    - if not all processes have values for all keys

- Obviously limits the possible reductions
  - No variable-size reductions!

Optimizations

- Optimized implementation
  - hardware optimization, e.g., BG/P
  - communication optimization, e.g., MPICH2, OMPI
- Computation/communication overlap?
  - pipelining with NonBlocking Collectives (NBC)
  - accepted for next generation MPI (2.x or 3.0)
  - offered in LibNBC (portable, OFED optimized)

Synchronization in MapReduce

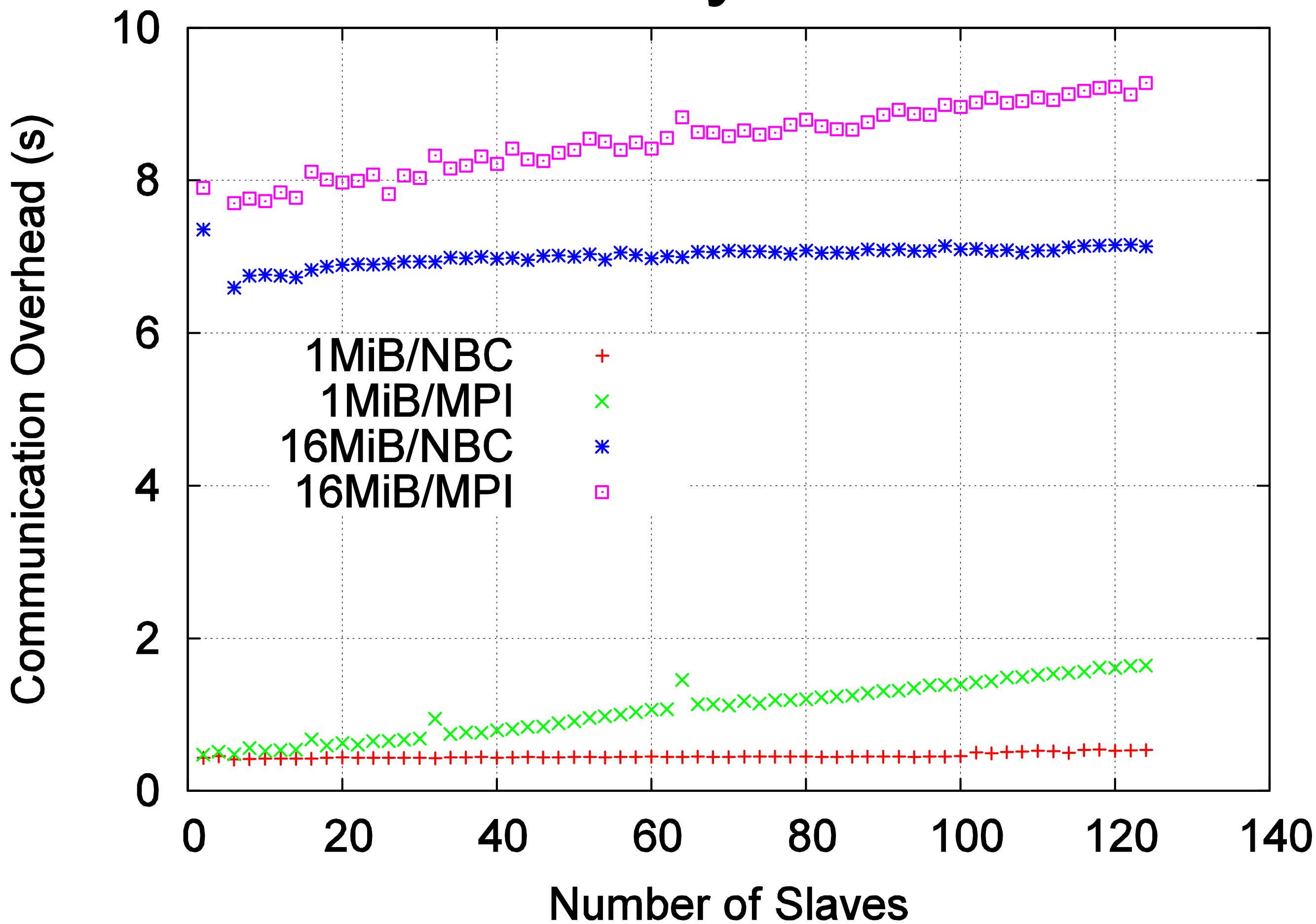


Performance Results

- MapReduce application simulator
  - Map tasks receive specified data and simulate computation
  - Reduce performs reduction over all keys
- System:
  - Odin at Indiana University
  - 128 4-core nodes with 4 GiB memory
  - InfiniBand interconnect
  - LibNBC (OFED optimized, threaded)

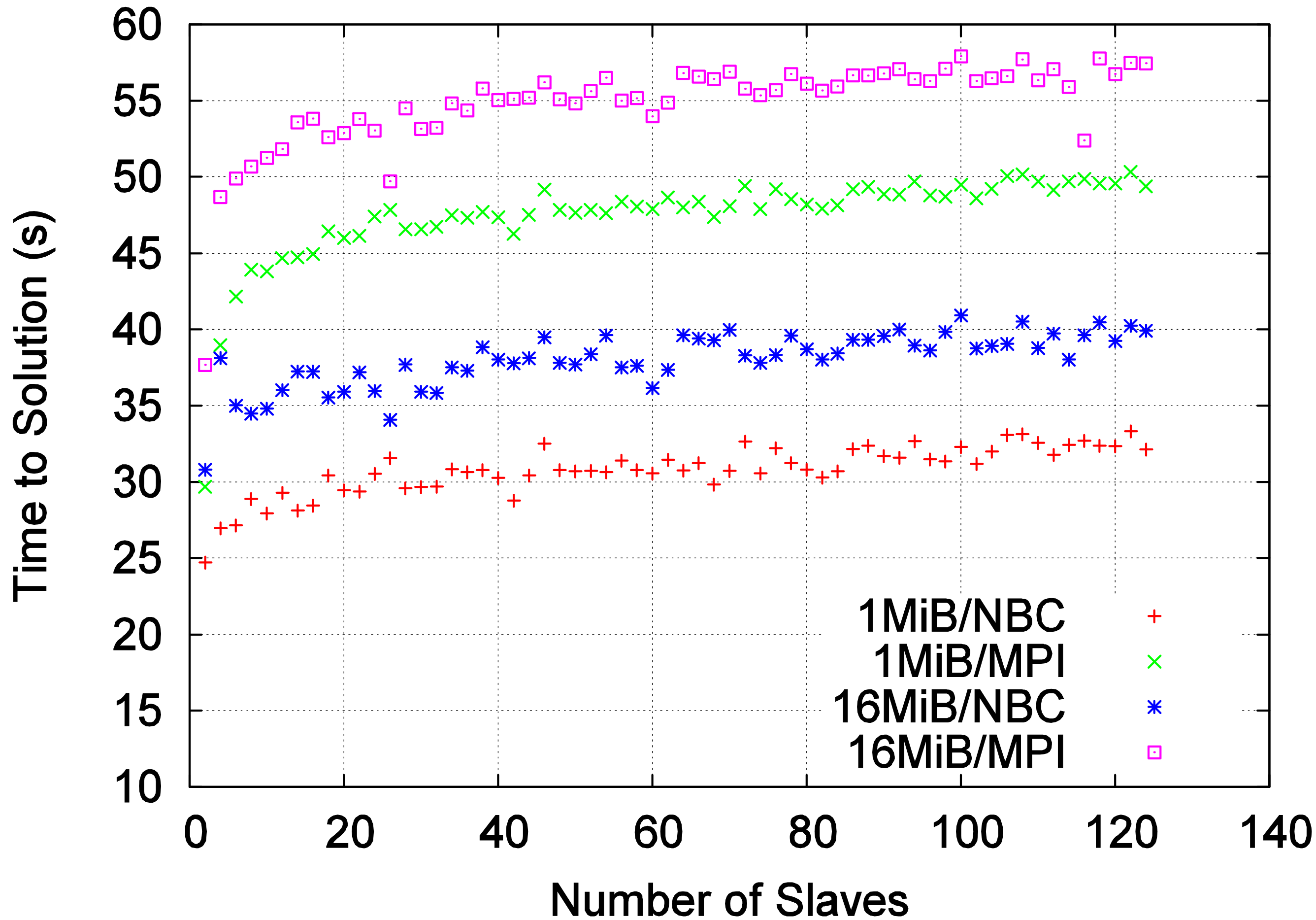
Static Workload

- Fixed workload: 1s per packet
- Reduction of comm/synch overhead of 27%



Dynamic Workload

- Dynamic workload: 1ms-10s
- Reduction of execution time of 25%



What does MPI need?

- Fault Tolerance
  - MPI offers basic inter-communicator FT
  - no support for collective communications
  - checking if a collective was successful is hard
  - collectives might never return (dead-/lifelock)
- Variable Reductions
  - MPI reductions are fixed-size
  - MR needs reductions of growing/shrinking data
  - Also useful for higher languages like C++, C#, or Python

Conclusions

- We proposed an unconventional way to implement MapReduce
  - efficiently uses collective communication
  - limited by MPI interface
  - allows efficient use of nonblocking collectives
- Implementation can be chosen based on properties of Map and Reduce
  - MPI-optimized implementation if possible
  - point-to-point based implementation otherwise

Questions

Questions?