Nonblocking and Sparse Collective Operations on Petascale Computers

Torsten Hoefler

Presented at Argonne National Laboratory on June 22nd 2010

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Disclaimer

- The views expressed in this talk are those of the speaker and not his employer or the MPI Forum.
- Appropriate papers are referenced in the lower left to give co-authors the credit they deserve.
- All mentioned software is available on the speaker's webpage as "research quality" code to reproduce observations.
- All pseudo-codes are for demonstrative purposes during the talk only ☺

Introduction and Motivation

Abstraction == Good!

Higher Abstraction == Better!

- Abstraction can lead to higher performance
  - Define the "*what*" instead of the "*how*"
  - *Declare* as much as possible *statically*
- Performance portability is important
  - Orthogonal optimization (separate network and CPU)
- Abstraction simplifies
  - Leads to easier code

Abstraction in MPI

- MPI offers persistent or predefined:
  - Communication patterns
    - Collective operations, e.g., MPI_Reduce()
  - Data sizes & Buffer binding
    - Persistent P2P, e.g., MPI_Send_init()
  - Synchronization
    - e.g., MPI_Rsend()

What is missing?

- Current persistence is not sufficient!
  - Only predefined communication patterns
  - No persistent collective operations
- Potential collectives proposals:
  - Sparse collective operations (pattern)
  - Persistent collectives (buffers & sizes)
  - One sided collectives (synchronization)

Sparse Collective Operations

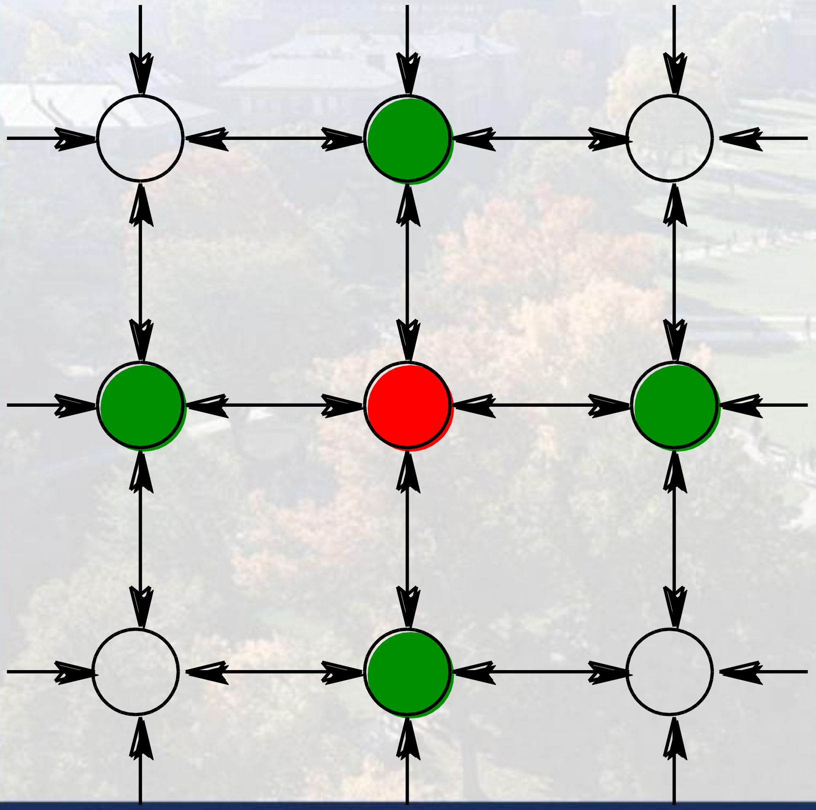
- User-defined communication patterns
  - Optimized communication scheduling
- Utilize MPI process topologies
  - Optimized process-to-node mapping

```
MPI_Cart_create(comm, 2 /* ndims */, dims,
  periods, 1 /*reorder*/, &cart);
MPI_Neighbor_alltoall(sbuf, 1, MPI_INT,
  rbuf, 1, MPI_INT, cart, &req);
```

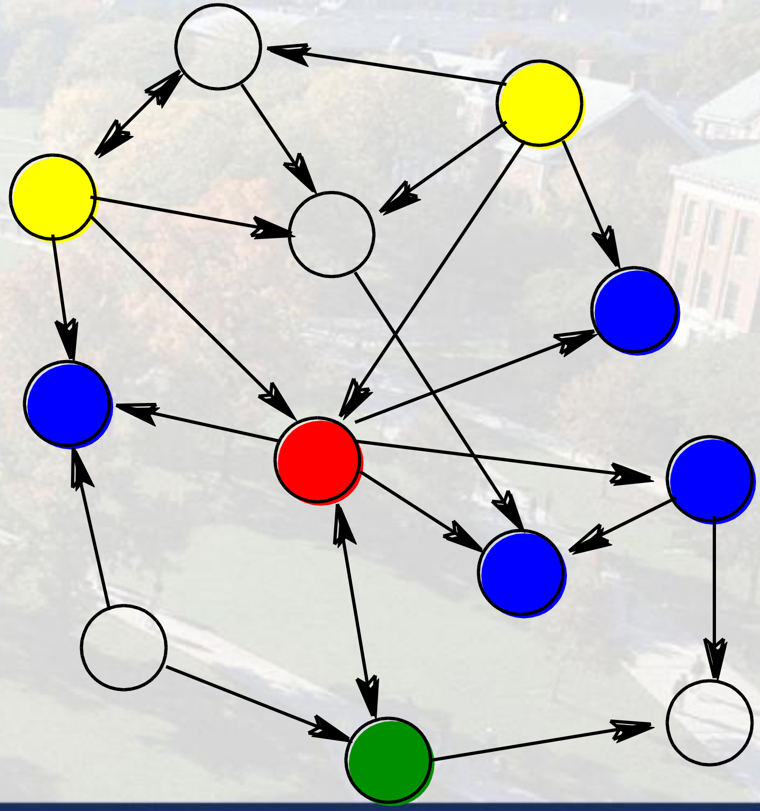
HIPS'09: "Sparse Collective Operations for MPI"

What is a Neighbor?

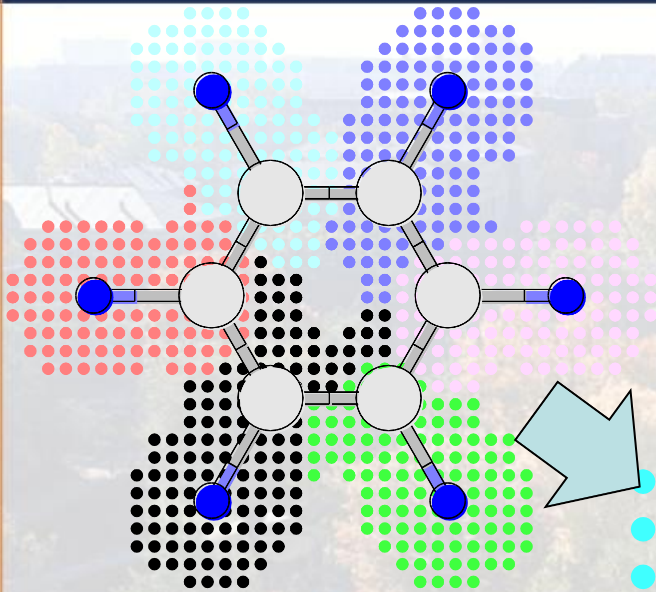
MPI_Cart_create()



MPI_Dist_graph_create()

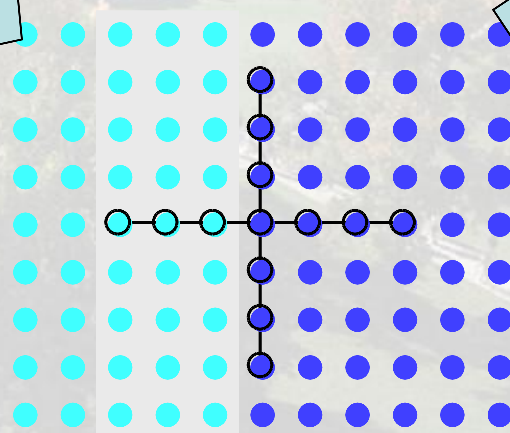


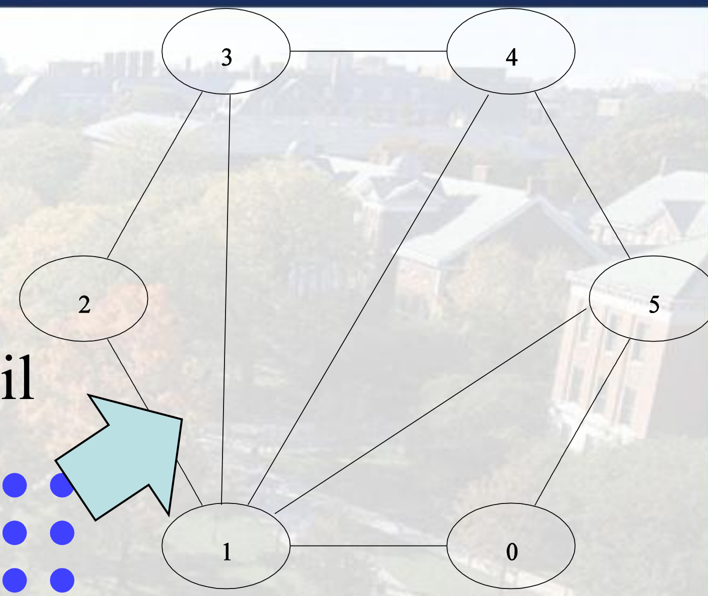
Creating a Graph Topology



Decomposed Benzene (P=6)

+13 point stencil





=Process Topology

All Possible Calls

- MPI_Neighbor_reduce()
  - Apply reduction to messages from sources
  - Missing use-case
- MPI_Neighbor_gather()
  - Sources contribute a single buffer
- MPI_Neighbor_alltoall()
  - Sources contribute personalized buffers
- Anything else needed … ?

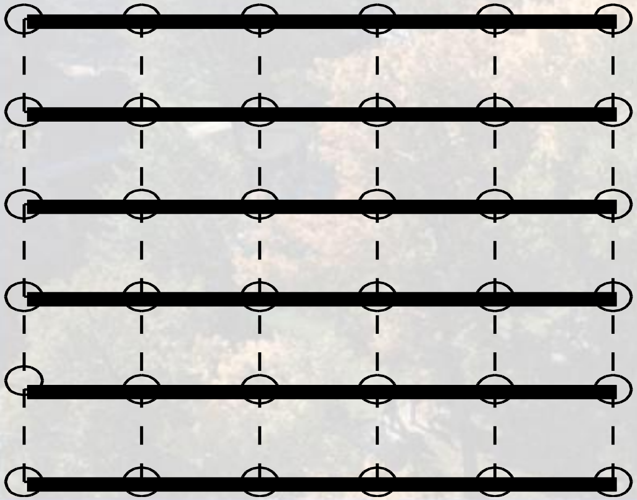
Advantages over Alternatives

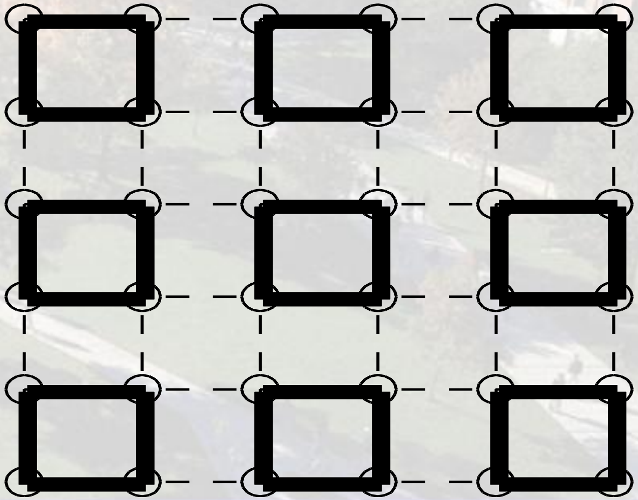
1. MPI_Sendrecv() etc. – defines “*how*”
   – Cannot optimize message schedule
   – No static pattern optimization (only buffer & sizes)

2. MPI_Alltoallv() – not scalable
   – Same as for send/recv
   – Memory overhead
   – No static optimization (no persistence)

An simple Example

- Two similar patterns
  - Each process has 2 heavy and 2 light neighbors
  - Minimal communication in 2 heavy+2 light rounds
  - MPI library can schedule accordingly!

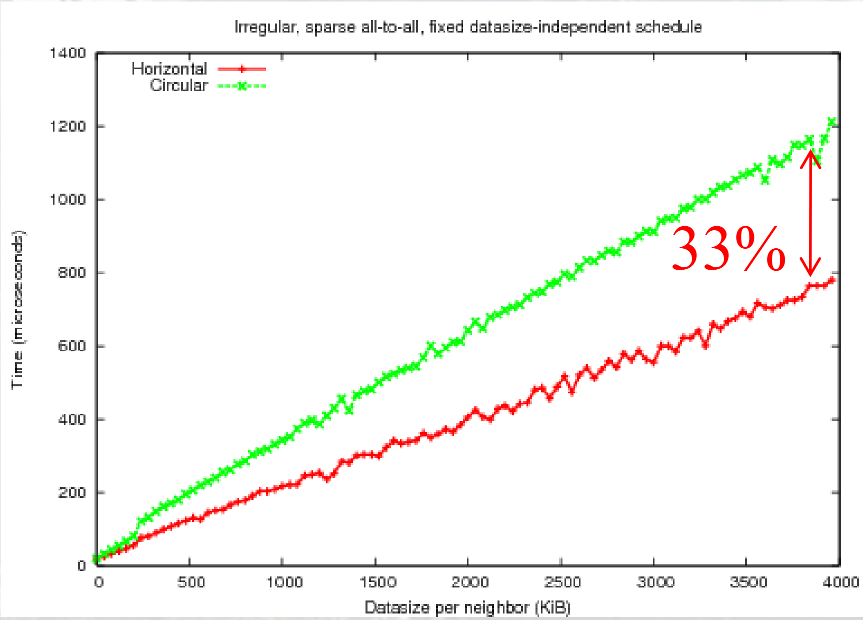




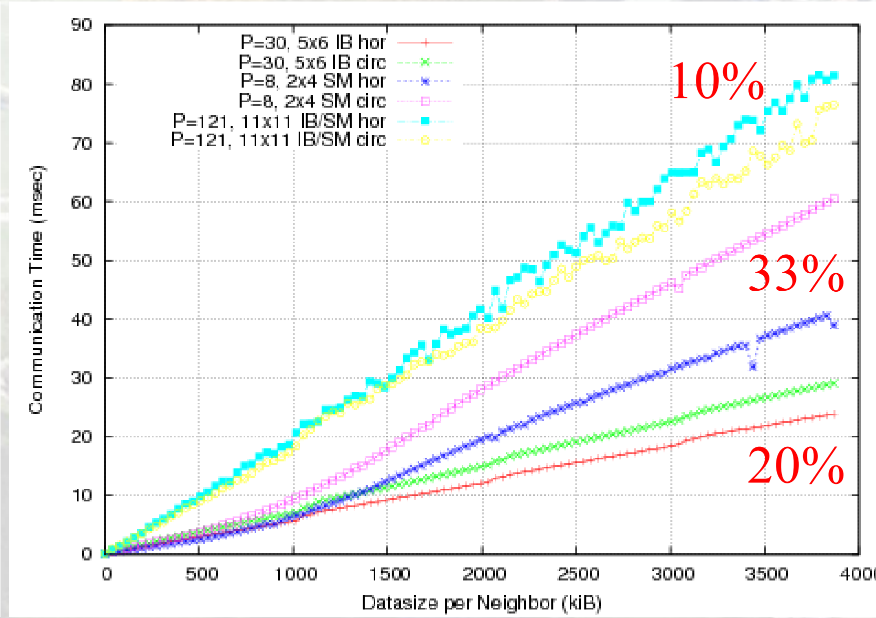
HIPS'09: "Sparse Collective Operations for MPI"

A naïve user implementation

```
for (direction in (left,right,up,down))
  MPI_Sendrecv(…, direction, …);
```



NEC SX-8 with 8 processes



IB cluster with 128 4-core nodes

HIPS'09: "Sparse Collective Operations for MPI"

More possibilities

- Numerous research opportunities in the near future:
  - Topology mapping
  - Communication schedule optimization
  - Operation offload
  - Taking advantage of persistence (sizes?)
  - Compile-time pattern specification
  - Overlapping collective communication

Nonblocking Collective Operations

- … finally arrived in MPI ☺
  - I would like to see them in MPI-2.3 (well …)
- Combines abstraction of (sparse) collective operations with overlap
  - Conceptually very simple:

```
MPI_Ibcast(buf, cnt, type, 0, comm, &req);
/* unrelated comp & comm */
MPI_Wait(&req, &stat)
```

  - Reference implementation: libNBC

SC'07: "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI"

"Very simple", really?

- Implementation difficulties
  1. State needs to be attached to request
  2. Progression (asynchronous?)
  3. Different optimization goals (overhead)
- Usage difficulties
  1. Progression (prefer asynchronous!)
  2. Identify overlap potential
  3. Performance portability (similar for NB P2P)

Collective State Management

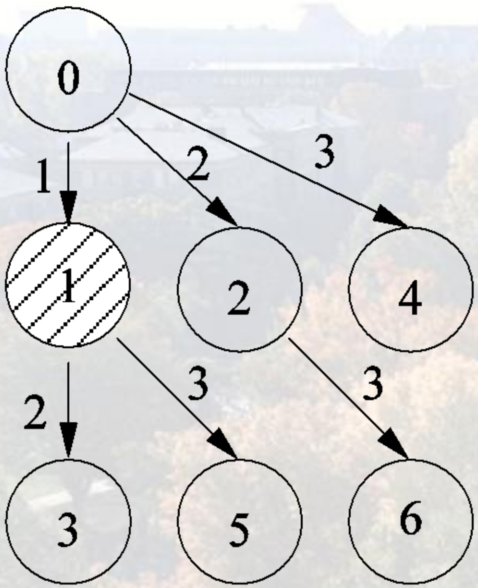
- Blocking collectives are typically implemented as loops

```
for (i=0; i<log_2(P); ++i) {
  MPI_Recv(…, src=(r-2^i)%P, …);
  MPI_Send(…, tgt=(r+2^i)%P, …);
}
```

- Nonblocking collectives can use schedules
  – Schedule records send/recv operations
  – The state of a collective is simply a pointer into the schedule

SC'07: "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI"

NBC_Ibcast() in libNBC 1.0



Pseudocode for schedule at rank 1:

```
NBC_Sched_recv(buf, count, dt, 0, schedule);
NBC_Sched_barr(schedule);
NBC_Sched_send(buf, count, dt, 3, schedule);
NBC_Sched_barr(schedule);
NBC_Sched_send(buf, count, dt, 5, schedule);
```

compile to
binary schedule

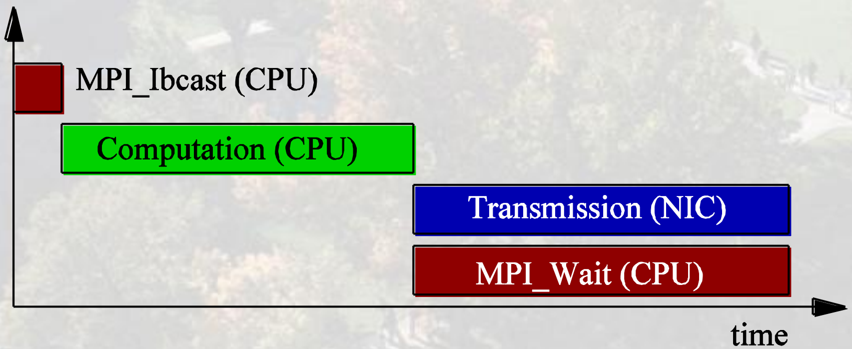
recv from 0	end	send to 3	end	send to 5

SC'07: "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI"

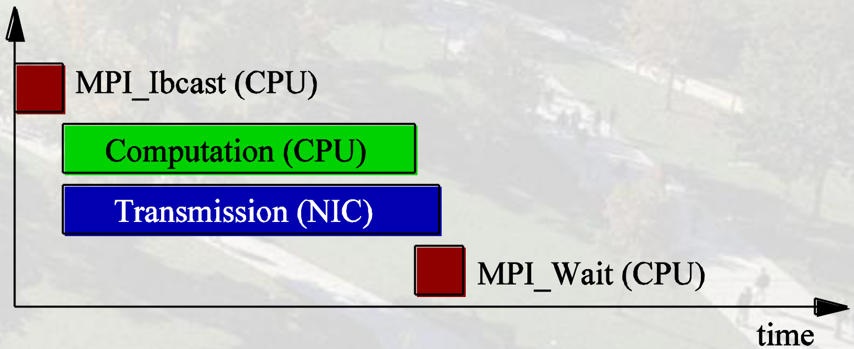
Progression

```
MPI_Ibcast(buf, cnt, type, 0, comm, &req);
/* unrelated comp & comm */
MPI_Wait(&req, &stat)
```

Synchronous Progression



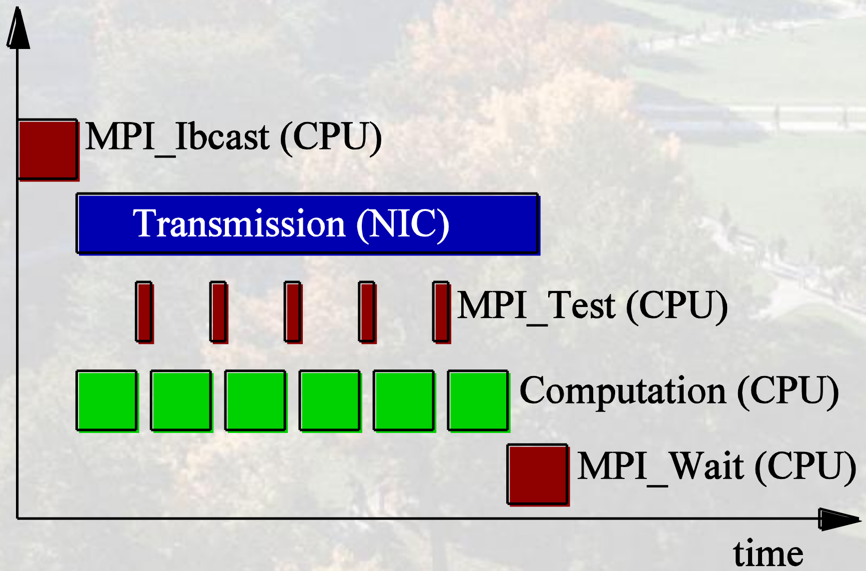
Asynchronous Progression



Cluster '07: "Message Progression in Parallel Computing – To Thread or not to Thread?"

Progression - Workaround

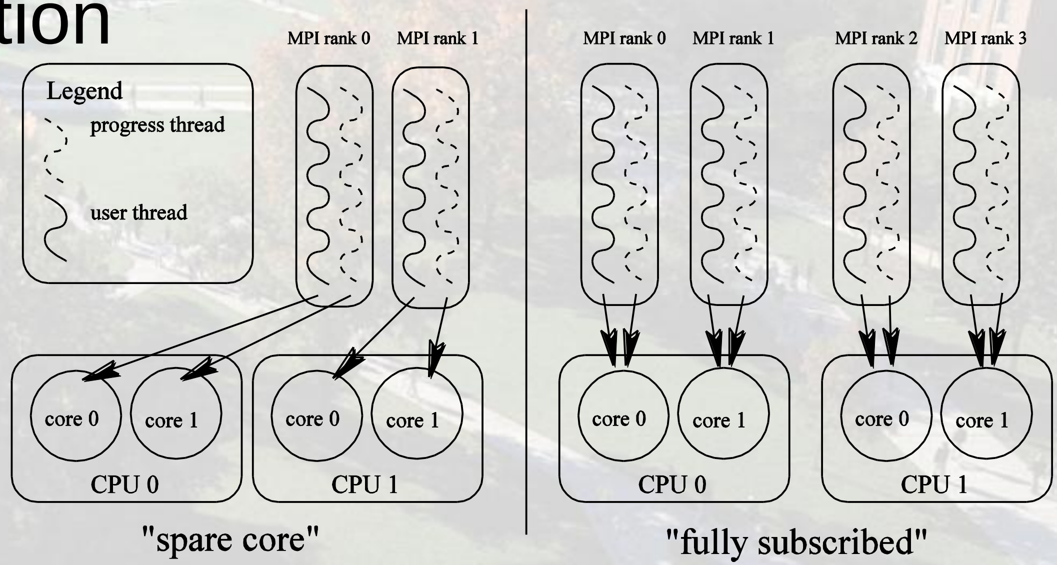
```
MPI_Ibcast(buf, cnt, type, 0, comm, &req);
/* comp & comm with MPI_Test() */
MPI_Wait(&req, &stat)
```



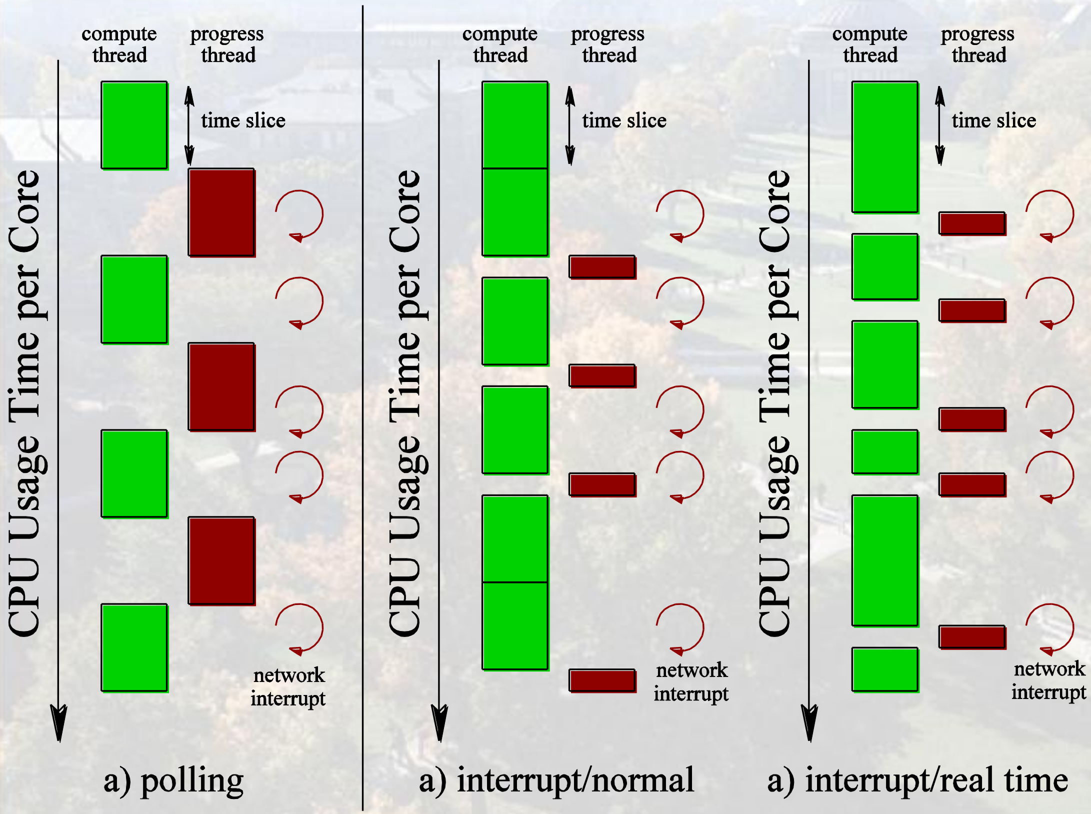
- Problems:
  - How often to test?
  - Modular code ☹
  - It's ugly!

Threaded Progression

- Two obvious options:
  - Spare communication core
  - Oversubscription
- It’s hard to spare a core!
  - might change



Oversubscribed Progression



- Polling == evil!
- Threads are not suspended until their slice ends!
- Slices are >1 ms
  - IB latency: 2 us!
- RT threads force Context switch
  - Adds costs

Cluster'07: "Message Progression in Parallel Computing – To Thread or not to Thread?"

A Note on Overhead Benchmarking

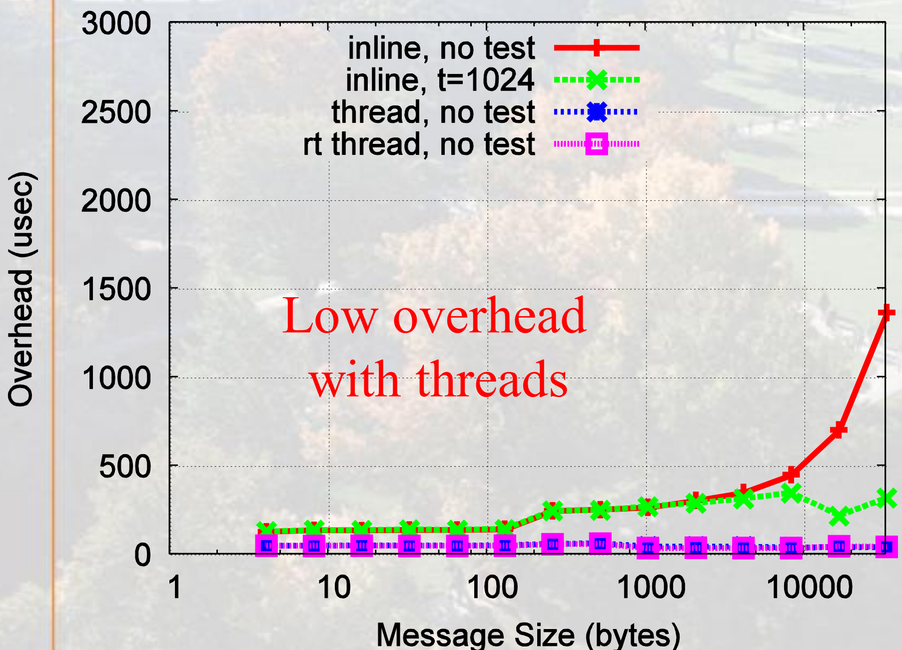
- Time-based scheme (bad):
  1. Benchmark time t for blocking communication
  2. Start communication
  3. Wait for time t (progress with MPI_Test())
  4. Wait for communication
- Work-based scheme (good):
  1. Benchmark time for blocking communication
  2. Find workload w that needs t to be computed
  3. Start communication
  4. Compute workload w (progress with MPI_Test())
  5. Wait for communication

K. McCurley: *"There are lies, damn lies, and benchmarks."*

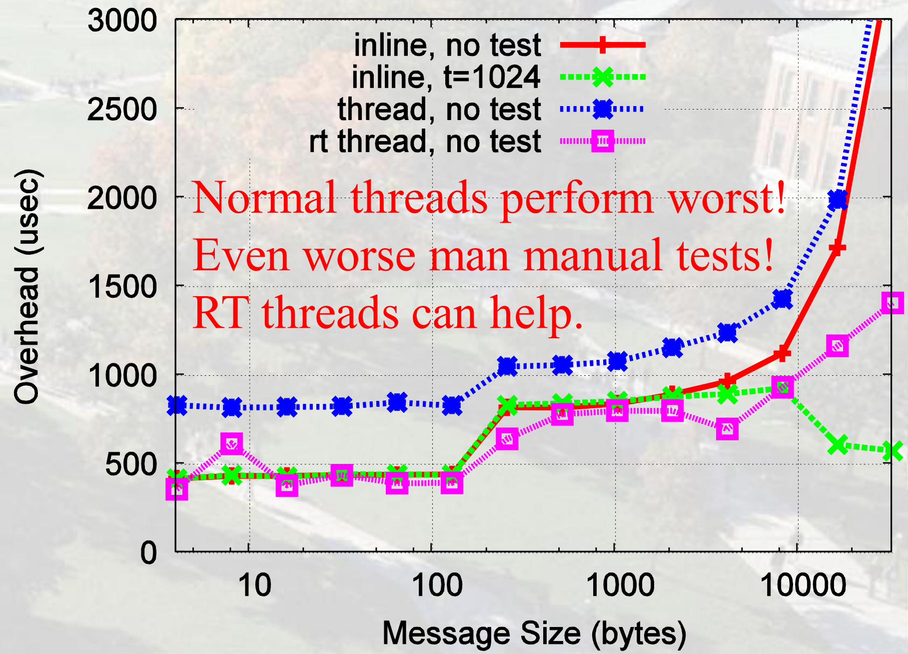
Work-based Benchmark Results

32 quad-core nodes with InfiniBand and libNBC 1.0

Spare Core



Oversubscribed



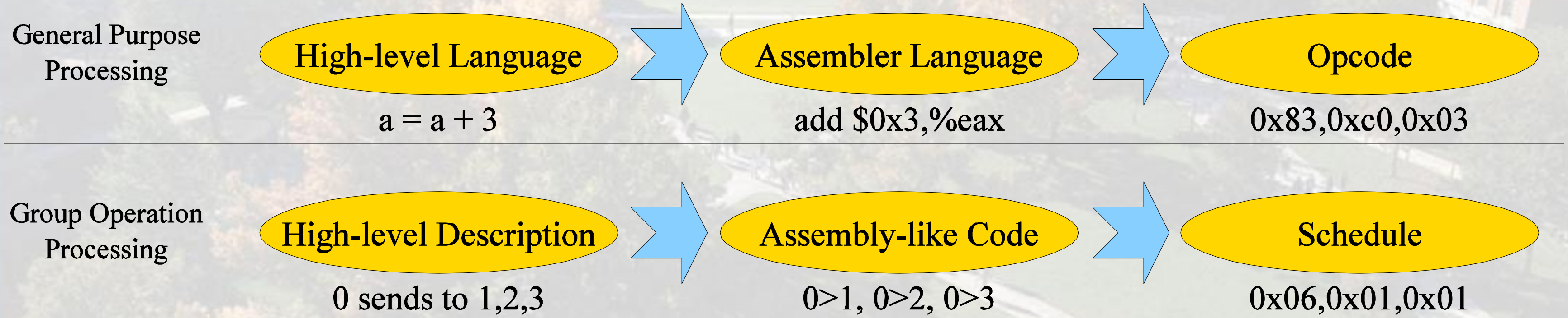
CAC'08: "Optimizing non-blocking Collective Operations for InfiniBand"

An ideal Implementation

- Progresses collectives independent of user computation (no interruption)
  - Either spare core or hardware offload!
- Hardware offload is not that hard!
  - Pre-compute communication schedules
  - Bind buffers and sizes on invocation
- Group Operation Assembly Language
  - Simple specification/offload language

Group Operation Assembly Language

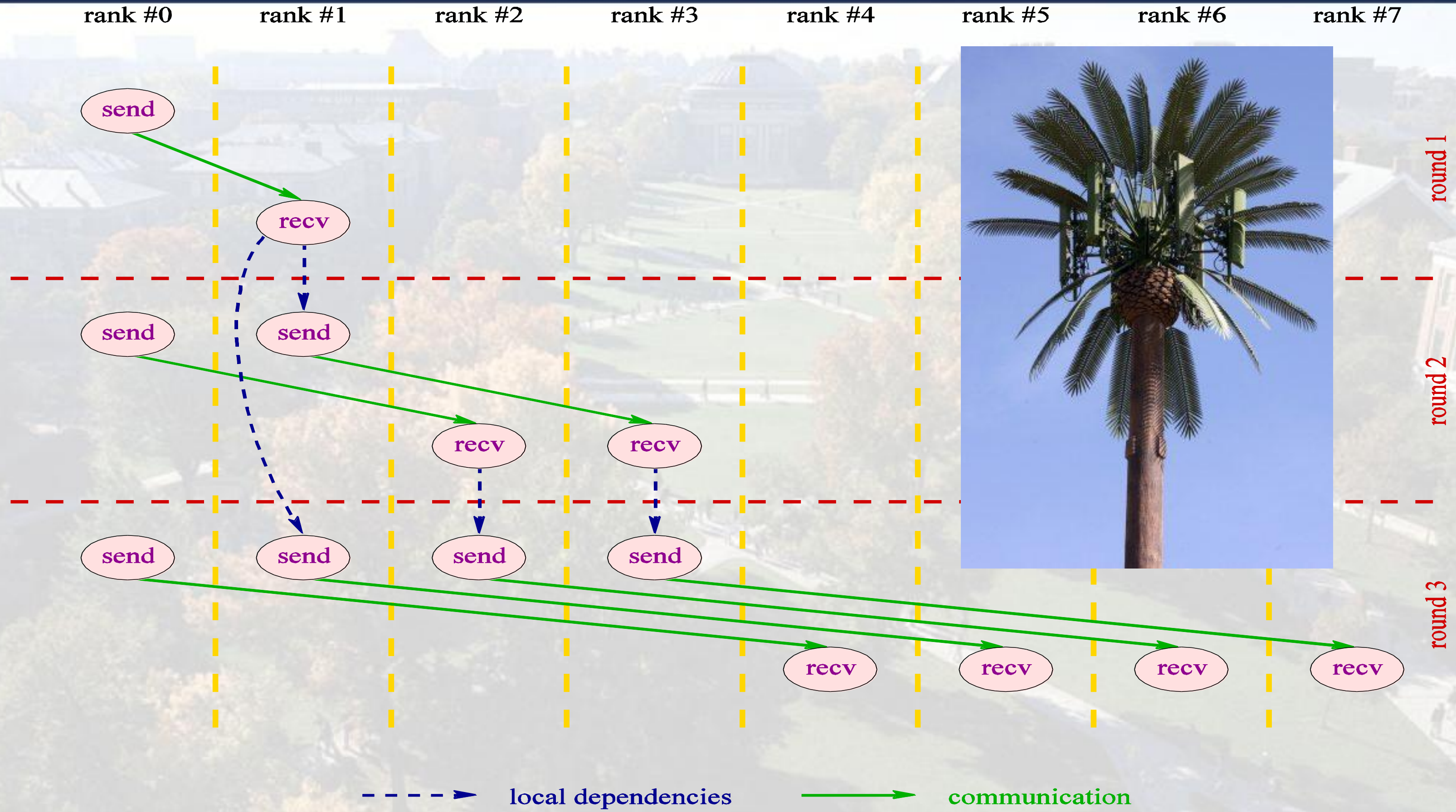
- Low-level collective specification
  - cf. RISC assembler code
- Translate into a machine-dependent form
  - i.e., schedule, cf. RISC bytecode



- Offload schedule into NIC (or on spare core)

ICPP'09: "Group Operation Assembly Language - A Flexible Way to Express Collective Communication"

A Binomial Broadcast Tree



ICPP'09: "Group Operation Assembly Language - A Flexible Way to Express Collective Communication"

Optimization Potential

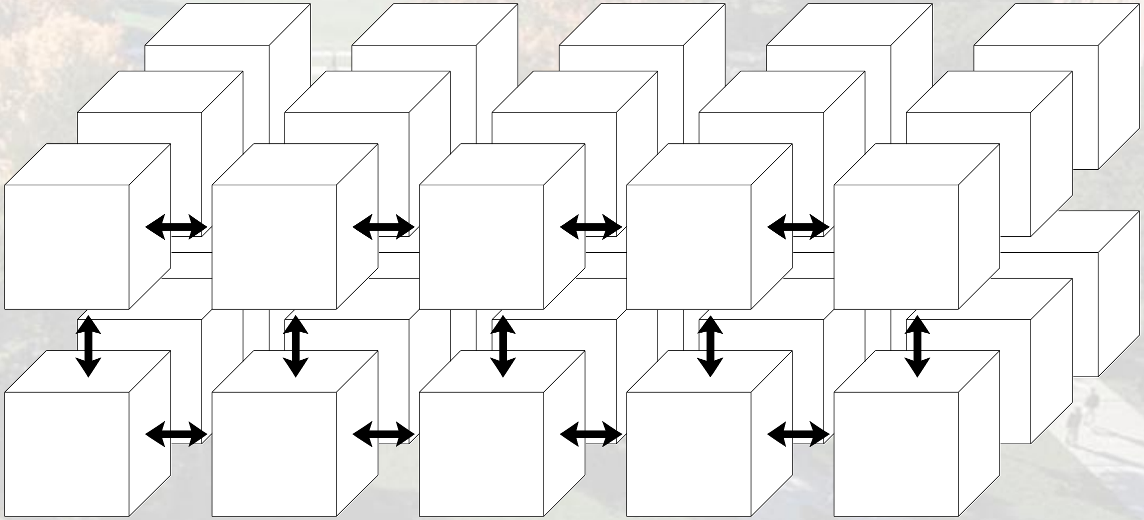
- Hardware-specific schedule layout
- Reorder of independent operations
  - Adaptive sending on a torus network
  - Exploit message-rate of multiple NICs
- Fully asynchronous progression
  - NIC or spare core process and forward messages independently
- Static schedule optimization
  - cf. sparse collective example

A User's Perspective

1. Enable overlap of comp & comm
   – Gain up to a factor of 2
   – Must be specified manually though
   – Progression issues ☹
2. Relaxed synchronization
   – Benefits OS noise absorption at large scale
3. Nonblocking collective semantics
   – Mix with p2p, e.g., termination detection

Patterns for Communication Overlap

- Simple code transformation, e.g., Poisson solver various CG solvers
  - Overlap inner matrix product with halo exchange

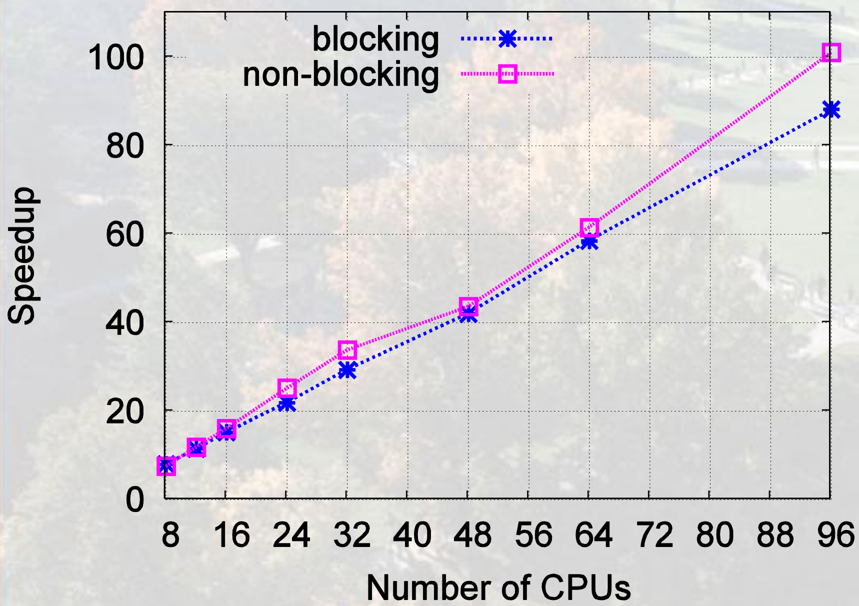


PARCO'07: "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations"

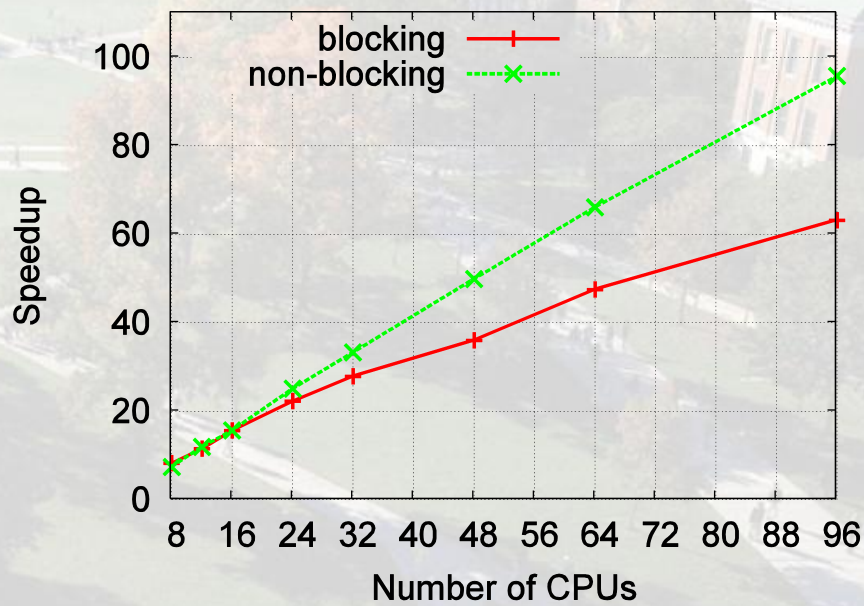
Poisson Performance Results

128 quad-core Opteron nodes, libNBC 1.0 (IB optimized, polling)

InfiniBand (SDR)



Gigabit Ethernet



PARCO'07: "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations"

Simple Pipelining Methods

- Parallel linear array transformation:

```
for(i=0; i<N/P; ++i) transform(i, in, out);
MPI_Gather(out, N/P, …);
```

- With pipelining and NBC:

```
for(i=0; i<N/P; ++i) {
  transform(i, in, out);
  MPI_Igather(out[i], 1, …, &req[i]);
}
MPI_Waitall(req, i, &statuses);
```

Problems

- Many outstanding requests
  - Memory overhead
- Too fine-grained communication
  - Startup costs for NBC are significant
- No progression
  - Rely on asynchronous progression?

Workarounds

- Tile communications
  - But aggregate **how many** messages?
- Introduce windows of requests
  - But limit to **how many** outstanding requests?
- Manual progression calls
  - But **how often** should MPI be called?

Final Optimized Transformation

```
for(i=0; i<N/P; ++i) transform(i, in, out);
MPI_Gather(out, N/P, …);
```

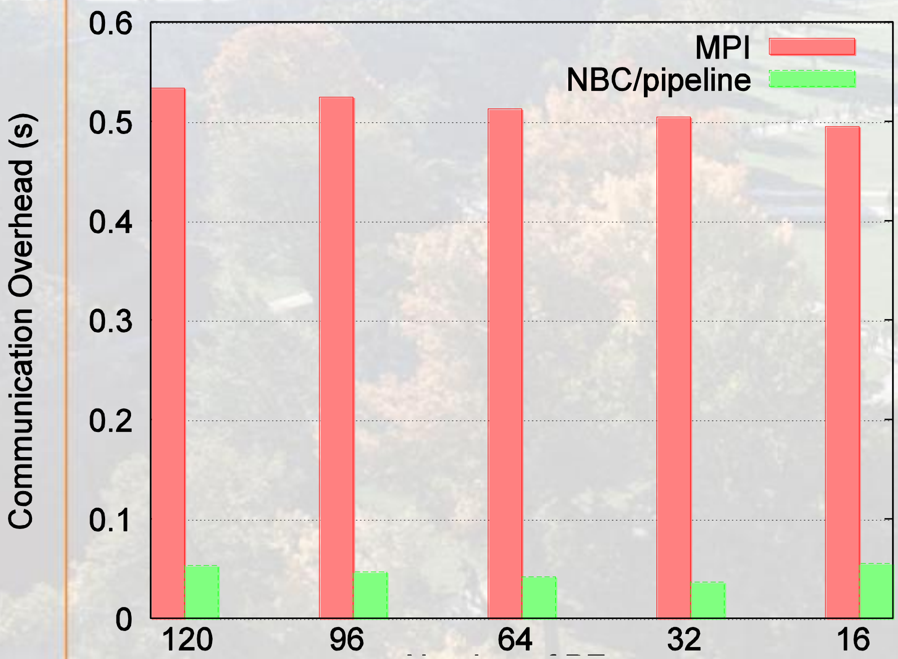
Inputs: t – tiling factor, w – window size, f – progress frequency

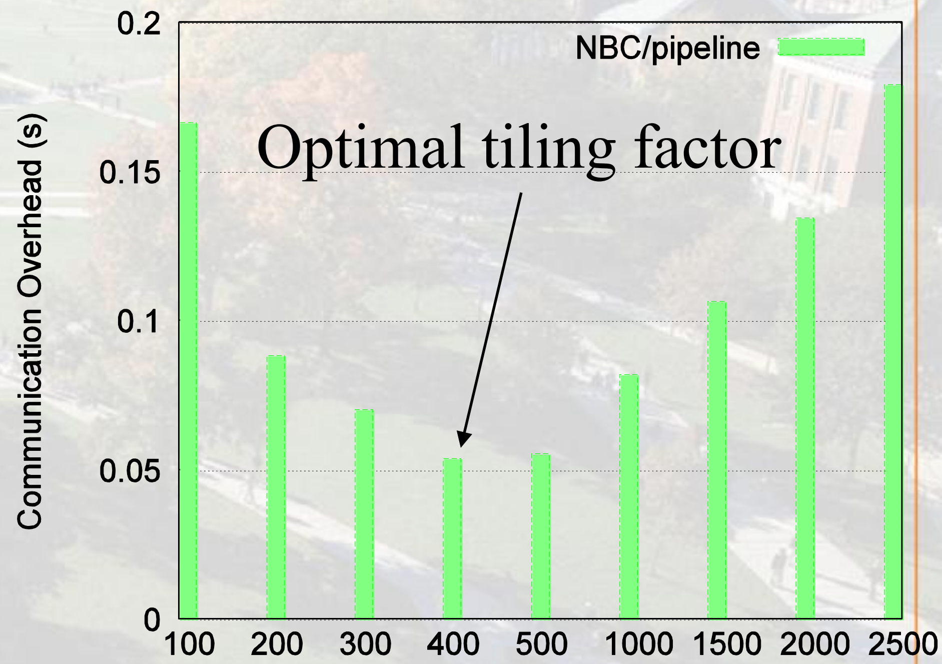
```
for(i=0; i<N/P/t; ++i) {
  for(j=i; j<i+t; ++j) transform(j, in, out);
  MPI_Igather(out[i], t, …, &req[i]);
  for(j=i; j>0; j-=f) MPI_Test(&req[i-f], &fl, &st);
  if(i>w) MPI_Wait(&req[i-w]);
}
MPI_Waitall(&req[N/P-w], w, &statuses);
```

SPAA'08: "Leveraging Non-blocking Collective Communication in High-performance Applications"

Parallel Compression Results

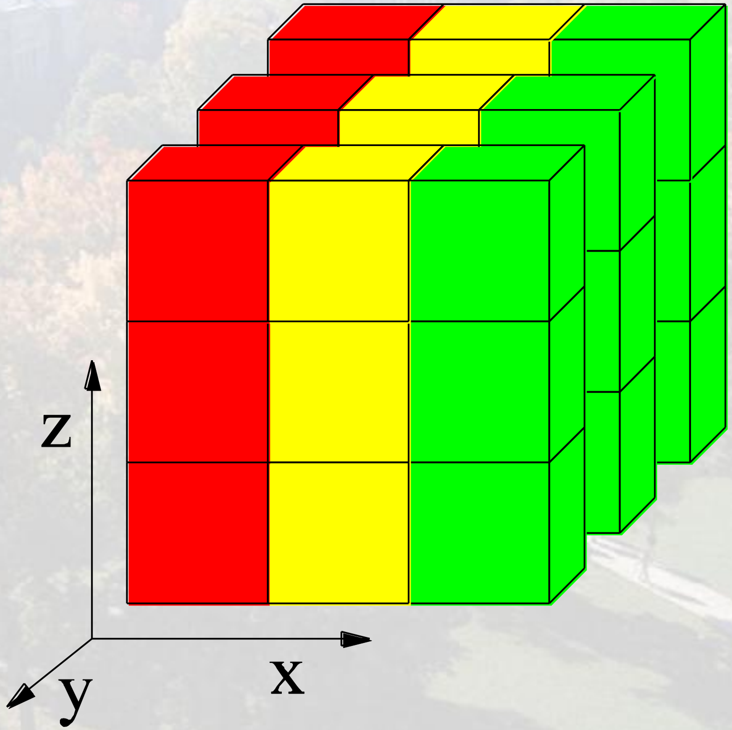
```
for(i=0; i<N/P; ++i) size += bzip2(i, in, out);
MPI_Gather(size, 1, …, sizes, 1, …);
MPI_Gatherv(out, size, …, outbuf, sizes, …);
```





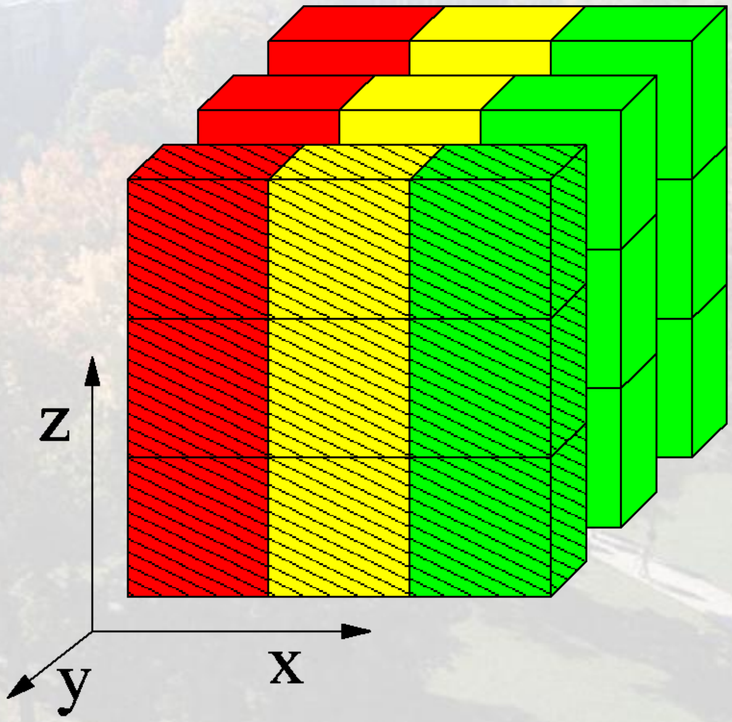
Parallel Fast Fourier Transform

- Data already transformed in y-direction



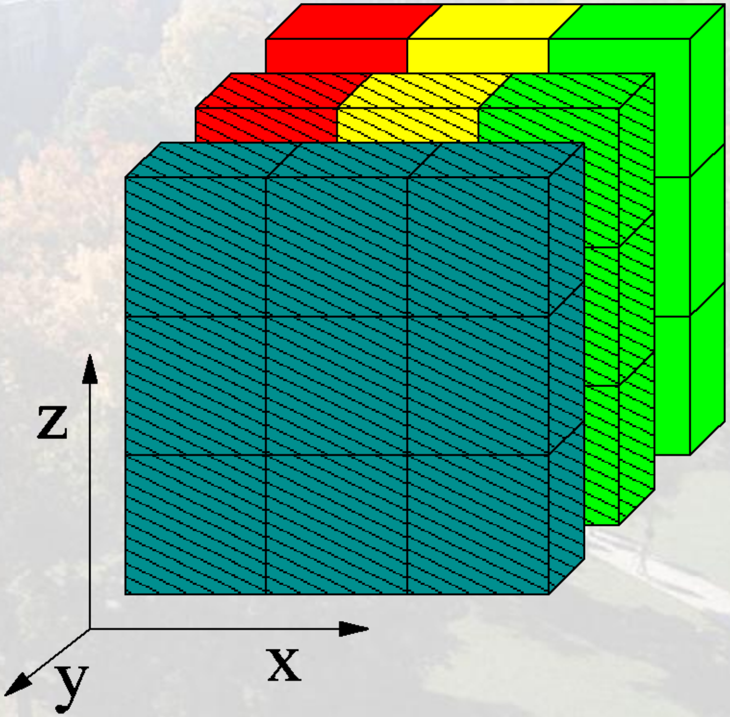
Parallel Fast Fourier Transform

- Transform first y plane in z



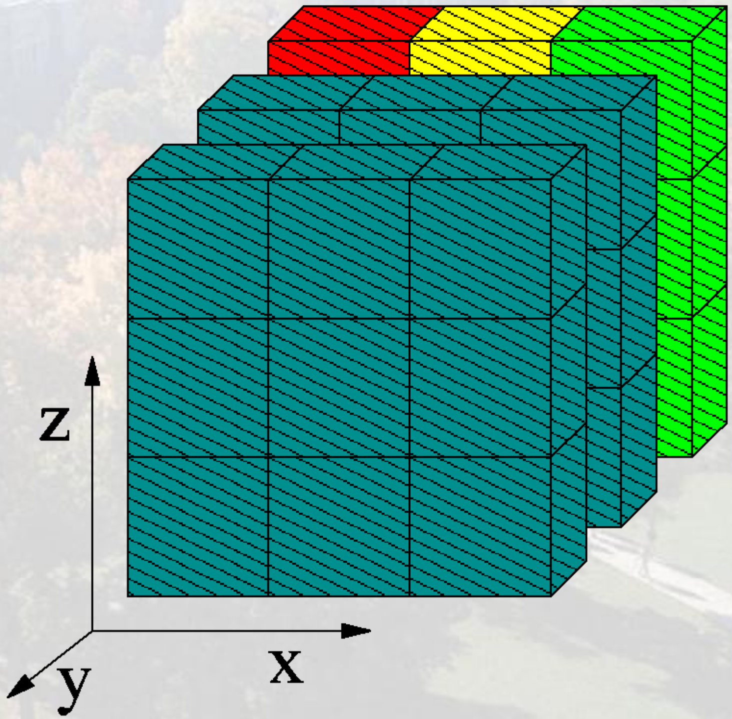
Parallel Fast Fourier Transform

- Start ialltoall and transform second plane



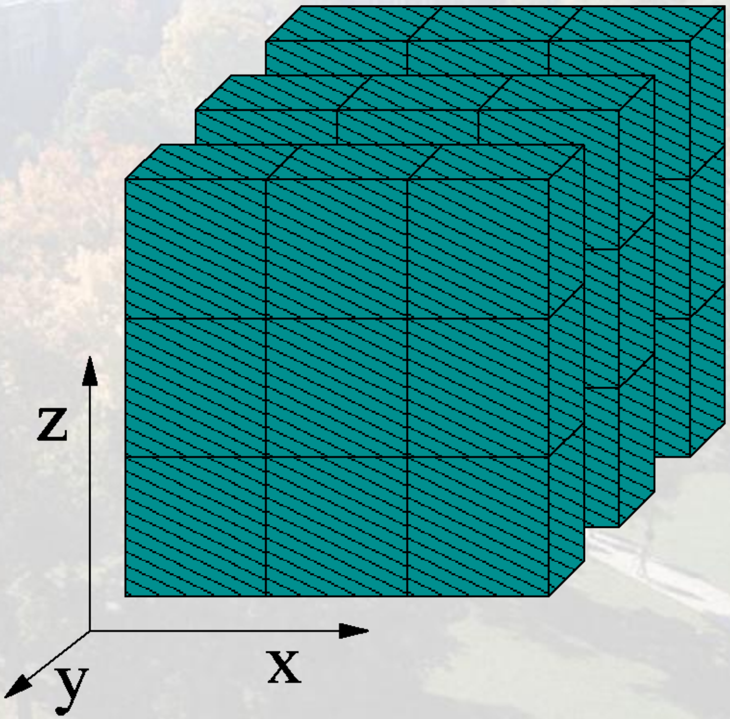
Parallel Fast Fourier Transform

- Start ialltoall (second plane) and transform third



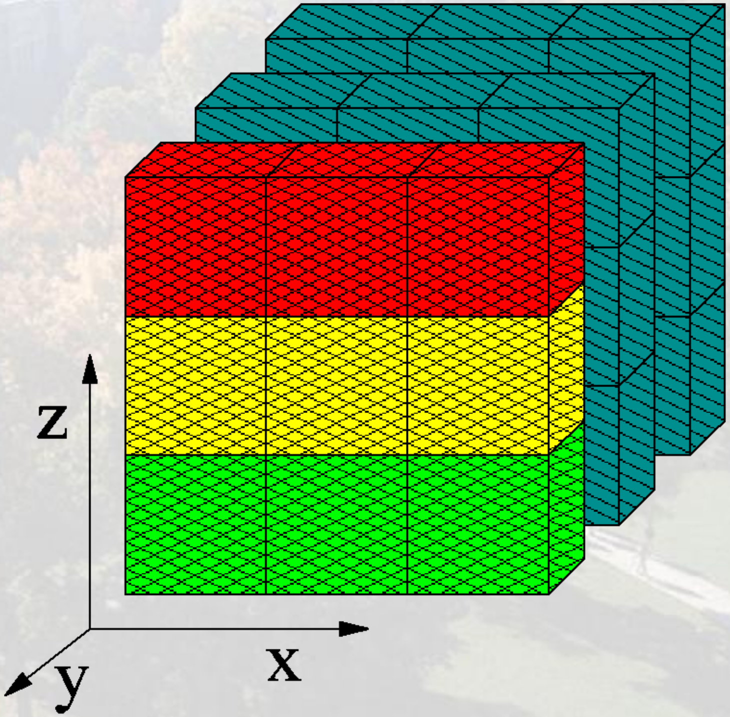
Parallel Fast Fourier Transform

- Start ialltoall of third plane and …



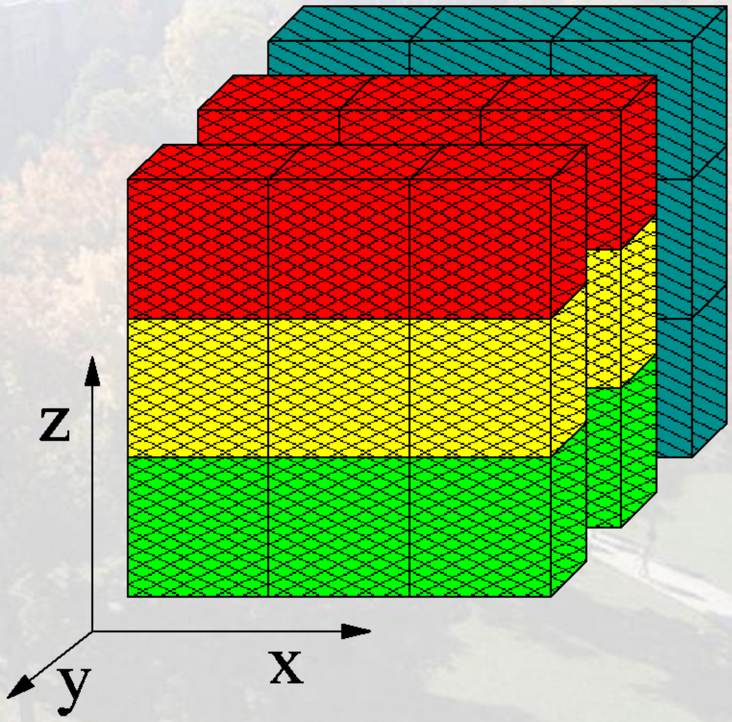
Parallel Fast Fourier Transform

- Finish ialltoall of first plane, start x transform



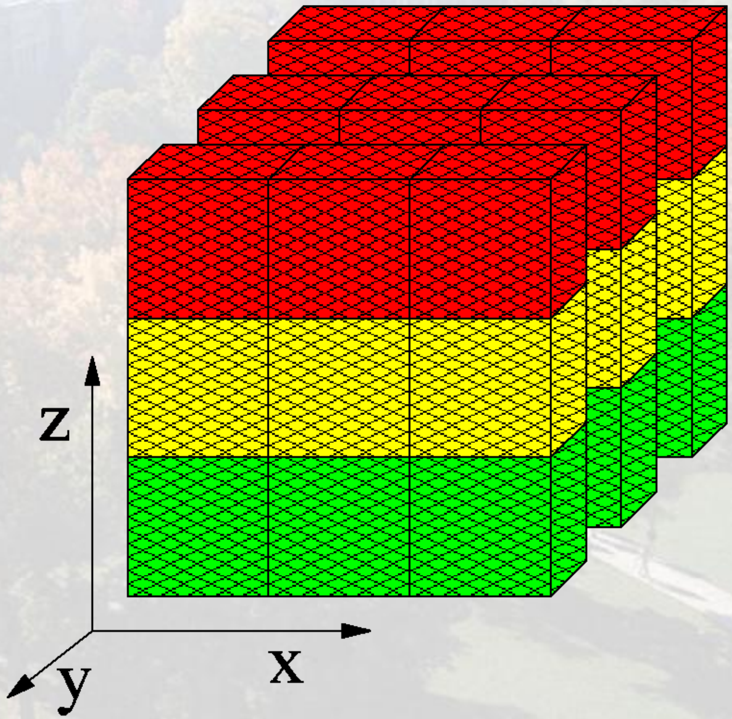
Parallel Fast Fourier Transform

- Finish second ialltoall, transform second plane



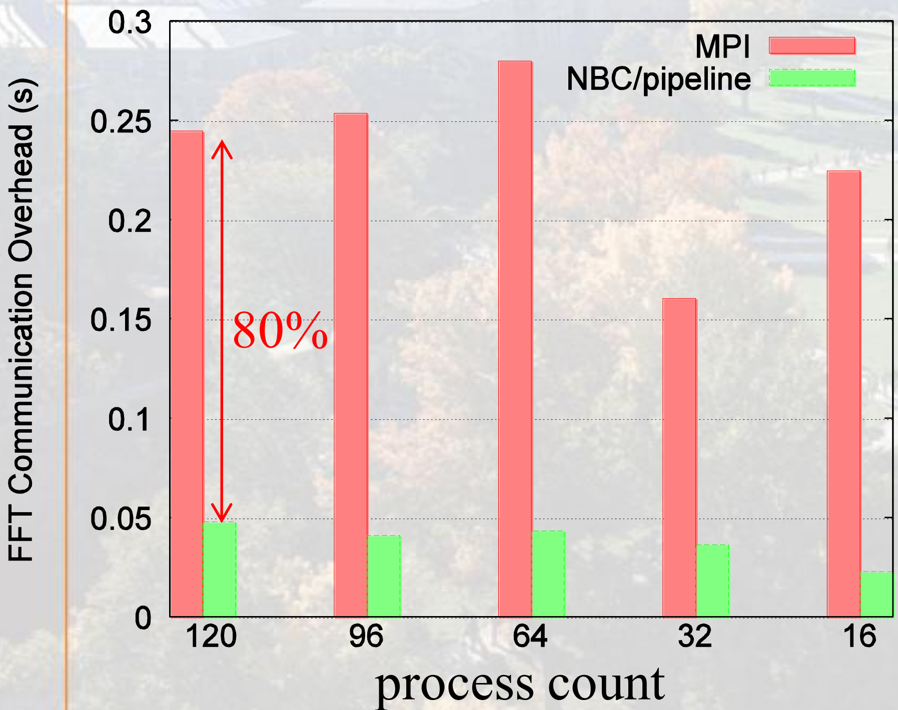
Parallel Fast Fourier Transform

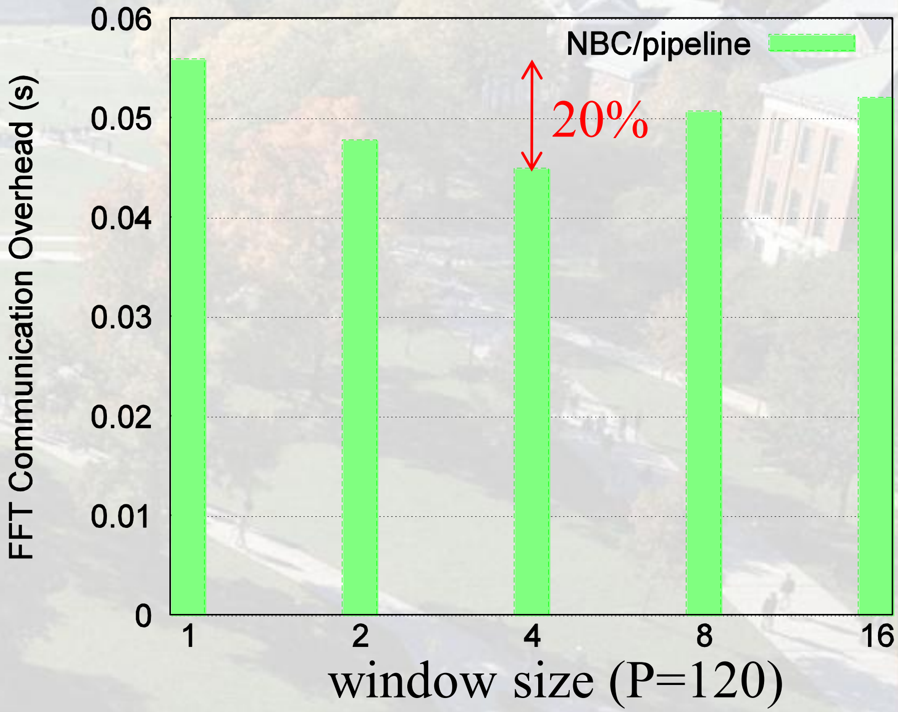
- Transform last plane → done



Performance Results

- Weak scaling 400^3-720^3 double complex





Again, why Collectives?

- Alternative: One-Sided/PGAS implementation

```
for(x=0; x<NX/P; ++x) 1dfft(&arr[x*NY], ny);
for(p=0; p<P; ++p) /* put data at process p */
for(y=0; x<NY/P; ++y) 1dfft(&arr[y*NX], nx);
```

- This trivial implementation will cause congestion
  - An MPI_Ialltoall would be scheduled more effectively
    - e.g., MPI_Alltoall on BG/P uses pseudo-random permutations
- No support for message scheduling
  - e.g., overlap copy on same node with remote comm
- One-sided collectives are worth exploring

Bonus: New Semantics!

- Quick example: Dynamic Sparse Data Exchange
- Problem:
  - Each process has a set of messages
  - No process knows from where it receives how much
- Found in:
  - Parallel graph computations
  - Barnes Hut rebalancing
  - High-impact AMR

PPoPP'10: "Scalable Communication Protocols for Dynamic Sparse Data Exchange"

DSDE Algorithms

- Alltoall ($\mathcal{O}(P)$)
- Reduce_scatter ($\mathcal{O}(P)$)
- One-sided Accumulate ($\mathcal{O}(\log(P))$)
- Nonblocking Barrier ($\mathcal{O}(\log(P))$)
  - Combines NBC and MPI_Ssend()
  - Best if numbers of neighbors is very small
  - Effectively constant-time on BG/P (barrier)

The Algorithm

Algorithm 1: $\mathcal{NBX}$—Nonblocking Consensus.

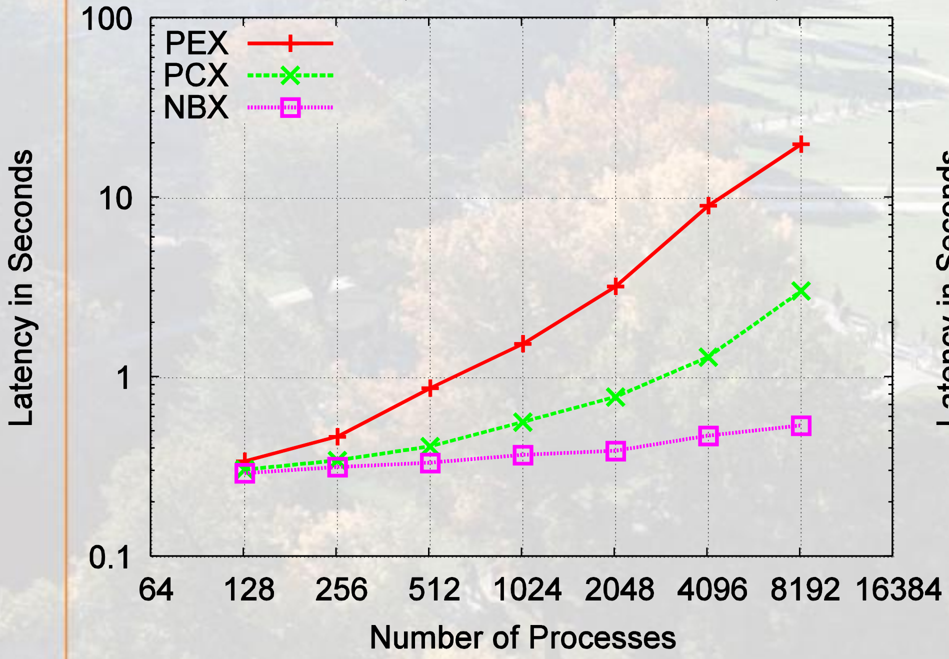
Input: List I of destinations and data
Output: List O of received data and sources

```
done=false;
barr_act=false;
foreach i ∈ I do
    start nonblocking synchronous send to process dest(i);
while not done do
    msg = nonblocking probe for incoming message;
    if msg found then
        allocate buffer, receive message, add buffer to O;
    if barr_act then
        comp = test barrier for completion;
        if comp then done=true;
    else
        if all sends are finished then
            start nonblocking barrier;
            barr_act=true;
```

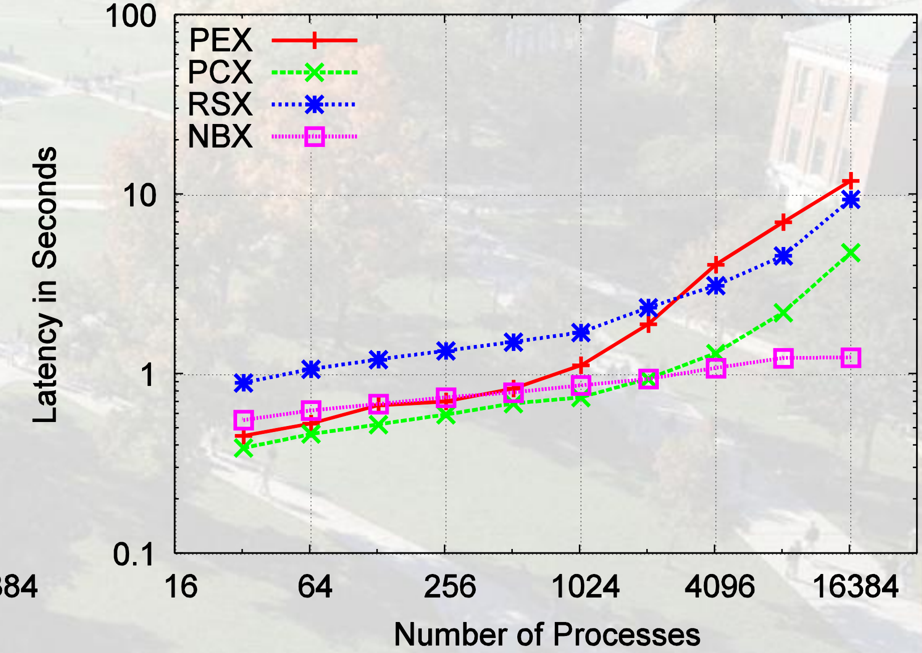
Some Results

Six random neighbors per process:

BG/P (DCMF barrier)



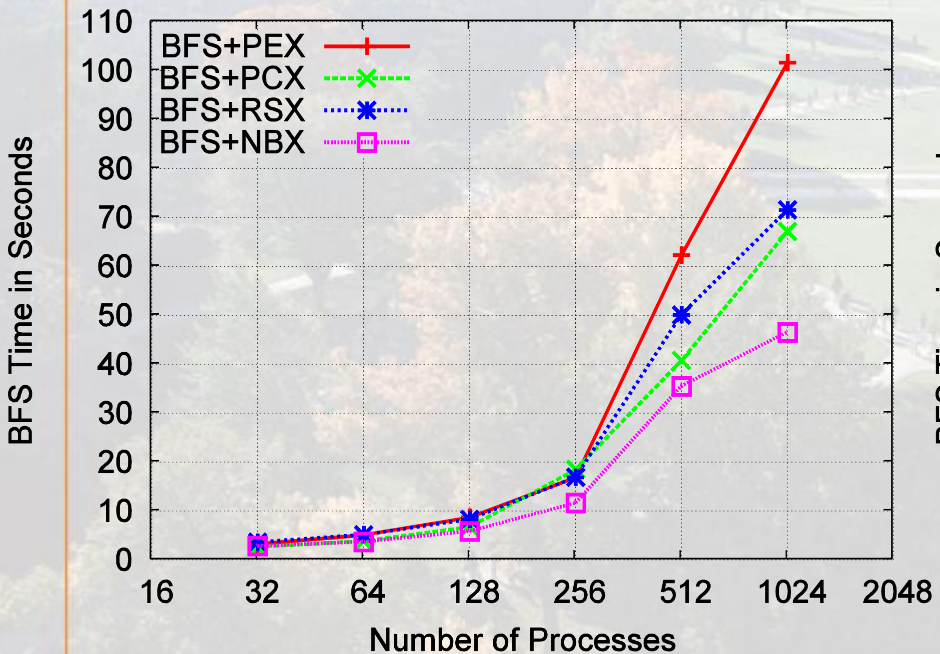
Jaguar (libNBC 1.0)



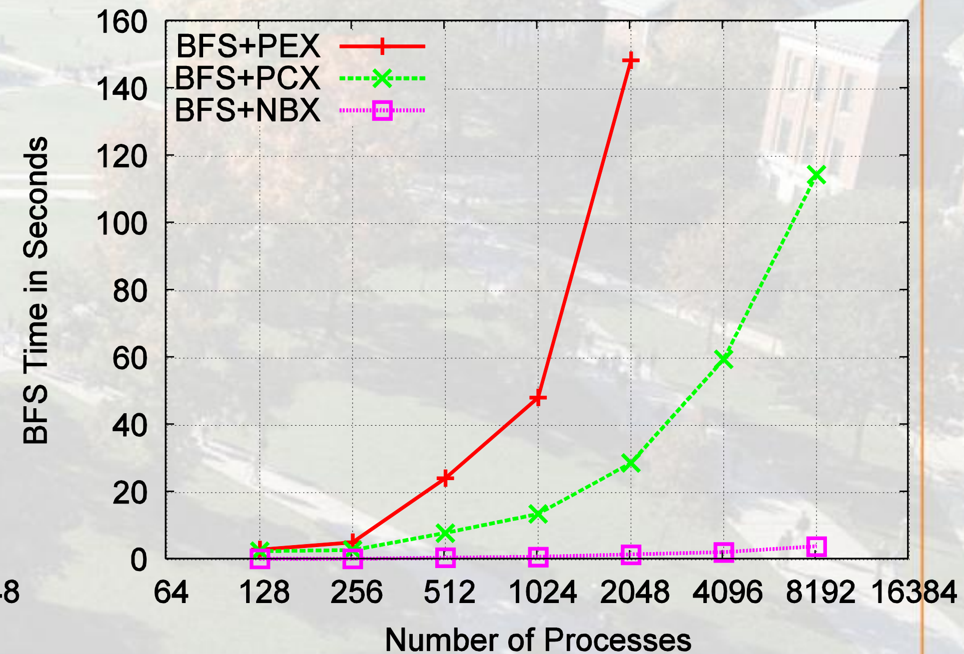
Parallel BFS Example

Well-partitioned clustered ER graph, six remote edges per process.

Big Red (libNBC 1.0)



BG/P (DCMF barrier)



Perspectives for Future Work

- Optimized hardware offload
  - Separate core, special core, NIC firmware?
- Schedule optimization for sparse colls
  - Interesting graph-theoretic problems
- Optimized process mapping
  - Interesting NP-hard graph problems ☺
- Explore application use-cases
  - Overlap, OS Noise, new semantics

Thanks and try it out!

- LibNBC (1.0 stable, IB optimized)
  http://www.unixer.de/NBC
- Some of the referenced articles:
  http://www.unixer.de/publications

Questions?

Bonus: 2nd note on benchmarking!

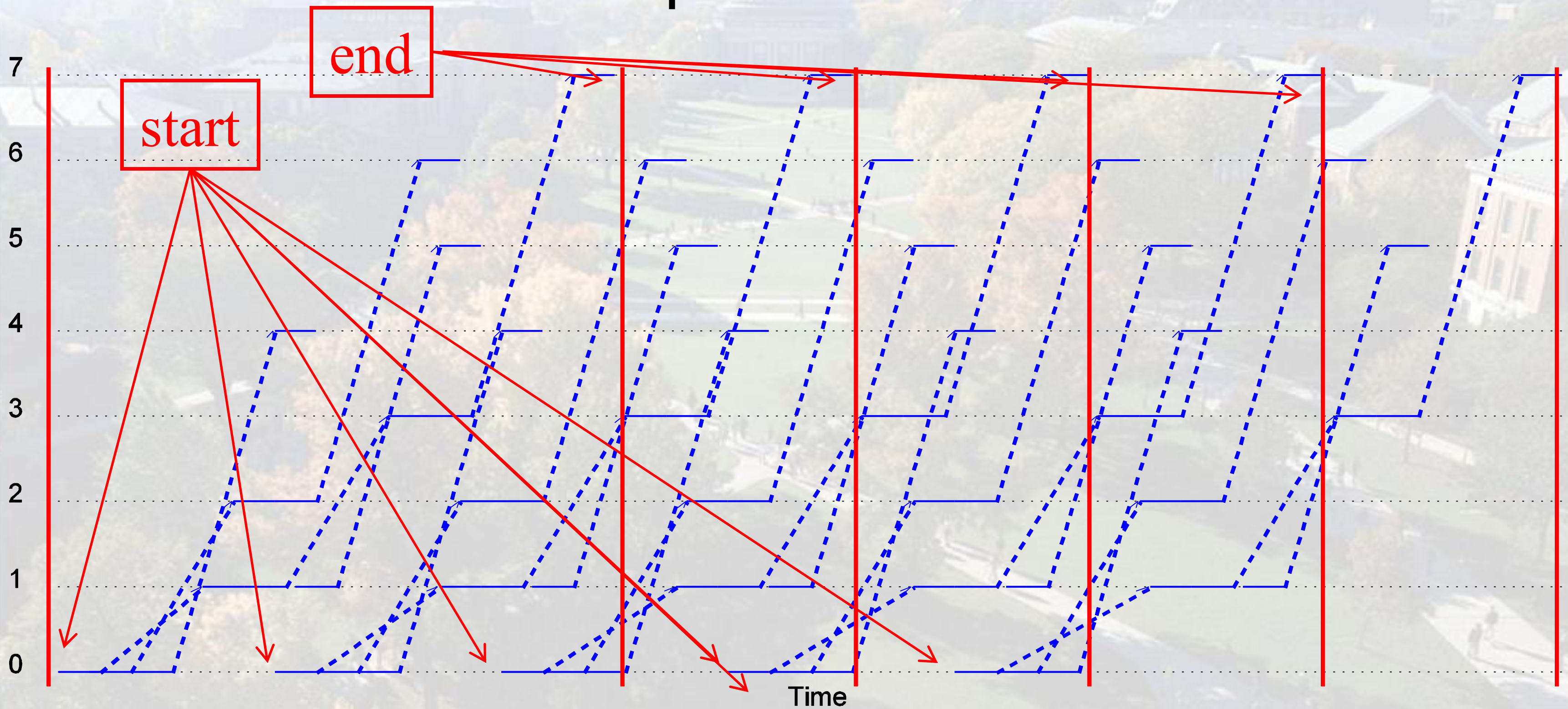
- Collective operations are often benchmarked in loops:

```
start= time();
for(int i=0; i<samples; ++i) MPI_Bcast(…);
end=time();
return (end-start)/samples
```

- This leads to pipelining and thus wrong benchmark results!

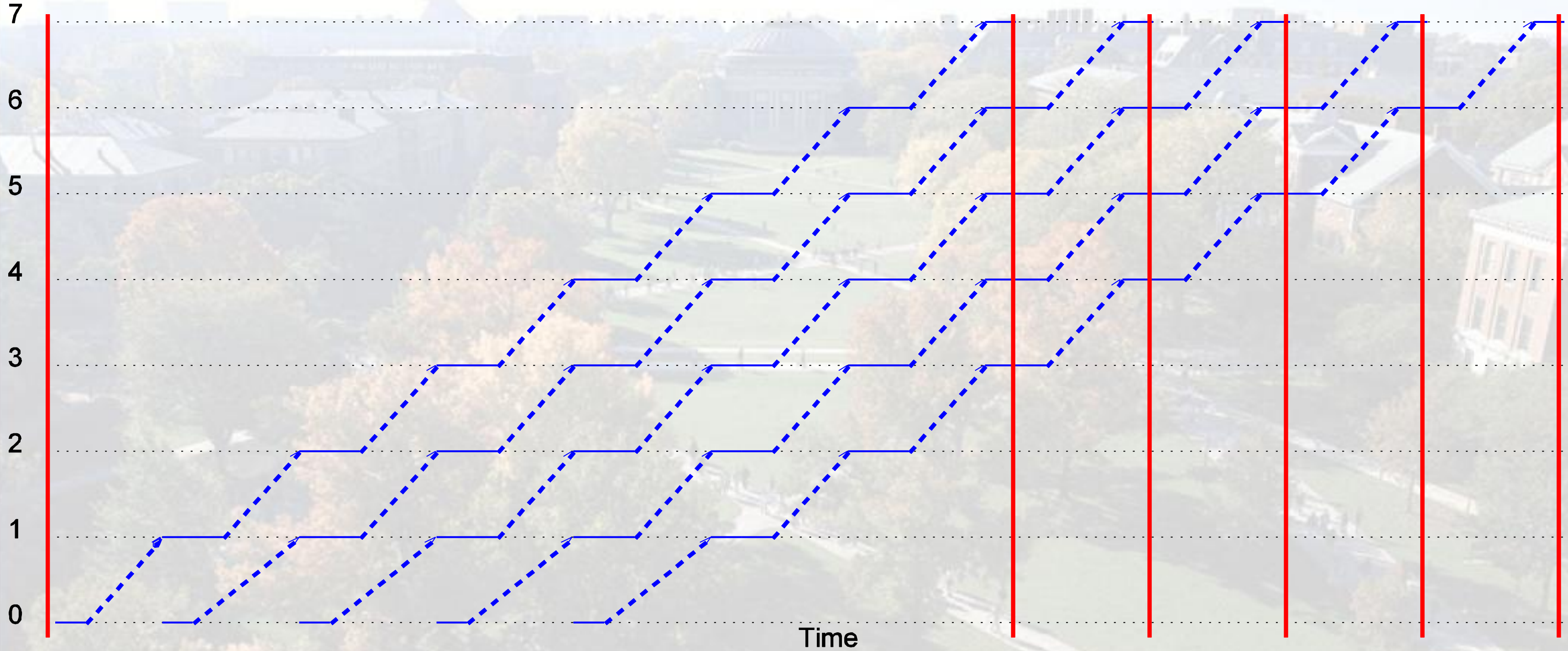
Pipelining? What?

Binomial tree with 8 processes and 5 bcasts:



SIMPAT'09: "LogGP in Theory and Practice […]"

Linear broadcast algorithm!



This bcast must be really fast, our benchmark says so!

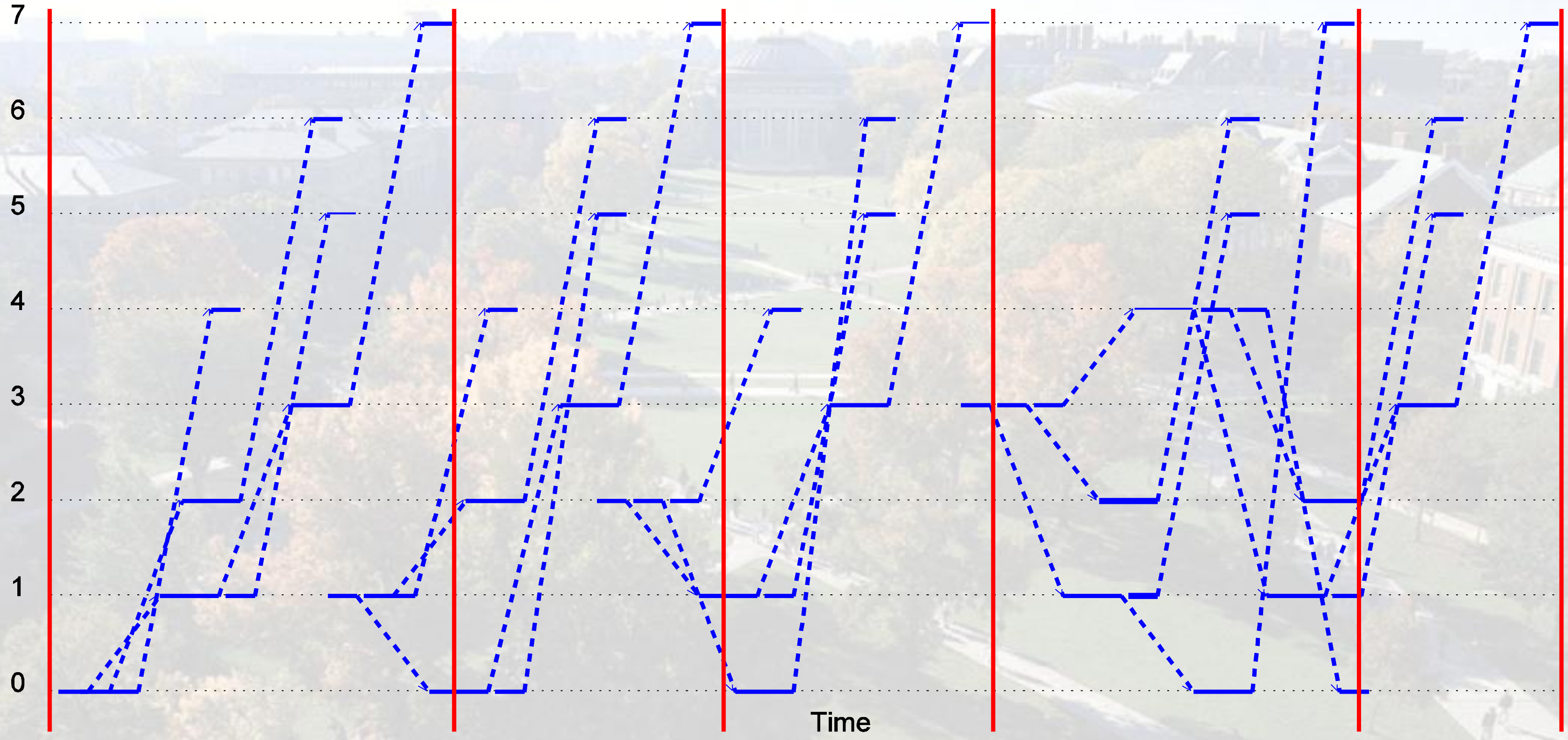
Root-rotation! The solution!

- Do the following (e.g., IMB)

```
start= time();
for(int i=0; i<samples; ++i)
  MPI_Bcast(…,root= i % np, …);
end=time();
return (end-start)/samples
```

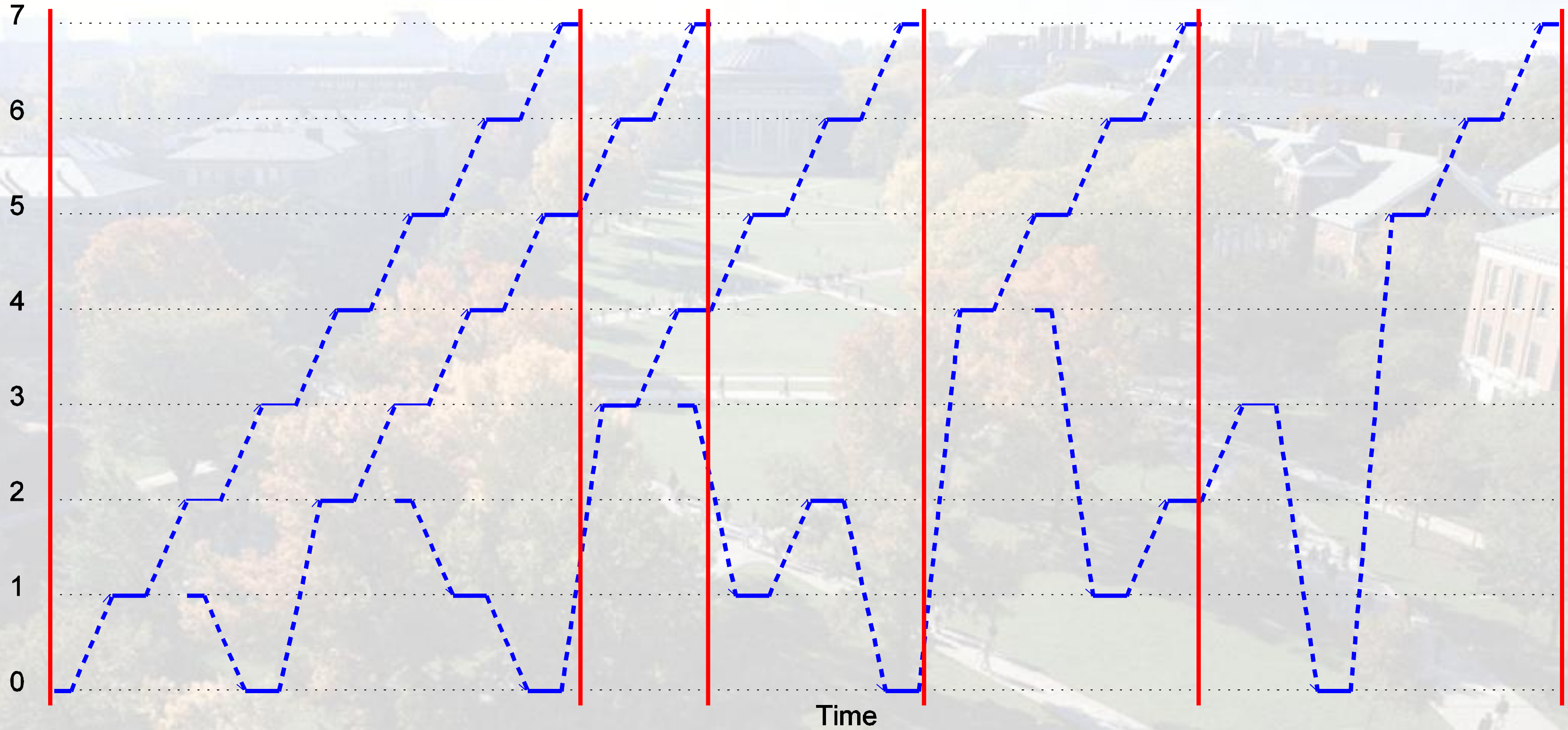
- Let’s simulate …

D’oh!



- But the linear bcast will work for sure!

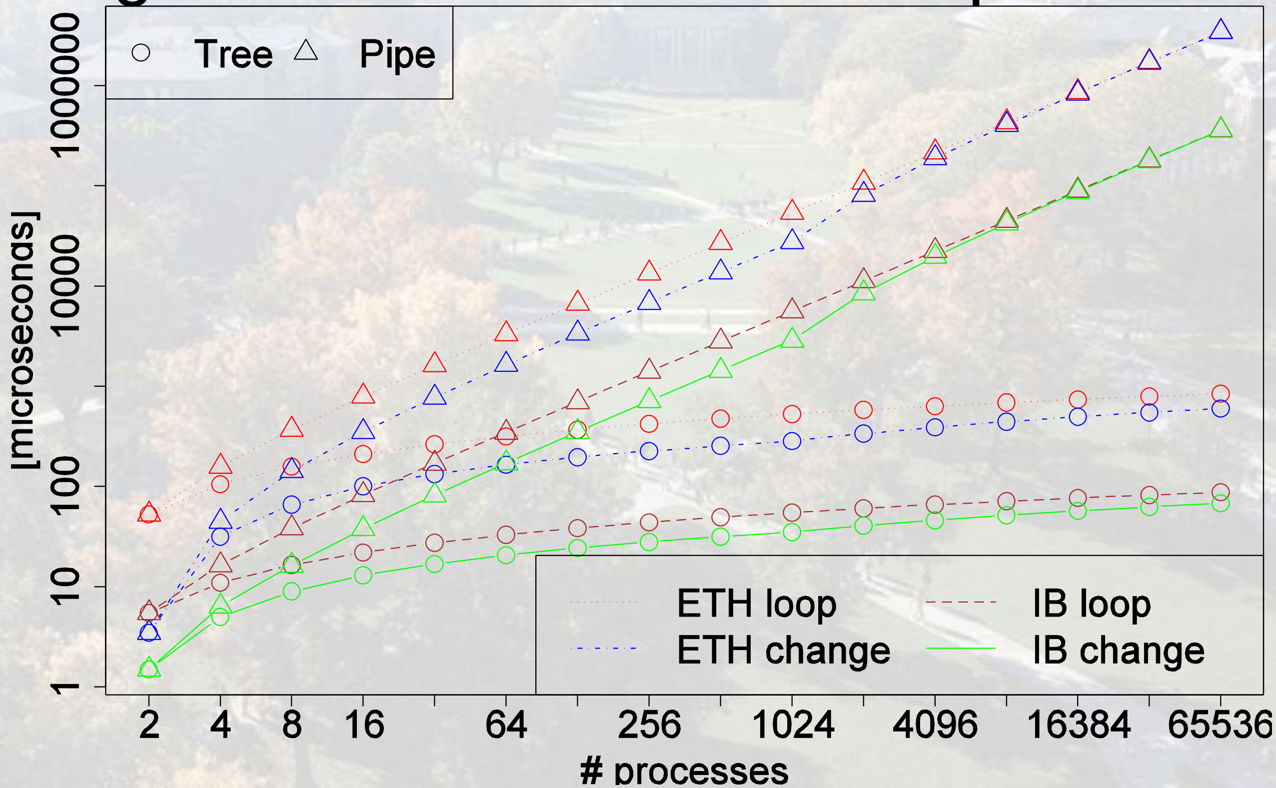
Well … not so much.



But how bad is it really? Simulation can show it!

Absolute Pipelining Error

- Error grows with the number of processes!



SIMPAT'09: "LogGP in Theory and Practice […]"