Efficient Point-to-Point Synchronization in UPC

Dan Bonachea, Rajesh Nishtala,
Paul Hargrove, Katherine Yelick

U.C. Berkeley / LBNL

http://upc.lbl.gov

Outline

- Motivation for point-to-point sync operations
- Review existing mechanisms in UPC
- Overview of proposed extension
- Microbenchmark performance
- App kernel performance

Point-to-Point Sync: Motivation

- **Many algorithms need point-to-point synchronization**
  - Producer/consumer data dependencies (one-to-one, few-to-few)
    - Sweep3d, Jacobi, MG, CG, tree-based reductions, …
  - Ability to couple a data transfer with remote notification
  - Message passing provides this synchronization implicitly
    - recv operation only completes after send is posted
    - Pay costs for sync & ordered delivery whether you want it or not
  - For PGAS, really want something like a signaling store (Split-C)
- **Current mechanisms available in UPC:**
  - UPC Barriers - stop the world sync
  - UPC Locks - build a queue protected with critical sections
  - Strict variables - roll your own sync primitives
- **We feel these current mechanisms are insufficient**
  - None directly express the semantic of a synchronizing data transfer
    - hurts productivity
    - Inhibits high-performance implementations, esp on clusters
  - This talk will focus on impact for cluster-based UPC implementations

Point-to-Point Sync Data Xfer in UPC

Thread 1		Thread 0
		`shared [] int data[…];`
`upc_memput(&data,…);`	→ / ←	
`upc_barrier;`	↔	`upc_barrier;`
		`/* consume data */`

barrier:
over-synchronizes threads
high-latency due to barrier
no overlap on producer

- **Works well for apps that are naturally bulk-synchronous**
  - all threads produce data, then all threads consume data
  - not so good if your algorithm doesn't naturally fit that model

Point-to-Point Sync Data Xfer in UPC

Thread 1

```
upc_lock(&L);

 upc_memput(&data,…);
 f = 1;

upc_unlock(&L);
```

Thread 0

```
shared [] int data[…];
int f = 0;
upc_lock_t *L = …;

while (1) {
   upc_lock(&L);
   if (f) break;
   upc_unlock(&L);
}
 /* consume data */
```

upc_locks:
latency 2.5+ round-trips
limited overlap on producer

- **This one performs so poorly on clusters that we won't consider it further…**

Point-to-Point Sync Data Xfer in UPC

Thread 1 | **Thread 0**

```
strict int f = 0;

upc_memput(&data,…);
f = 1;

while (!f) bupc_poll();
/* consume data */
```

memput + strict flag:
latency ~1.5 round-trips
no overlap on producer

```
strict int f = 0;

h = bupc_memput_async(&data,…);
  /* overlapped work… */
bupc_waitsync(h);
upc_fence;
h2 = bupc_memput_async(&f,…);
  /* overlapped work… */
bupc_waitsync(h2);

while (!f) bupc_poll();
/* consume data */
```

non-blocking memput + strict flag:
allows overlap
latency ~1.5 round-trips
higher complexity

- **There are several subtle ways to get this wrong**
  - not suitable for novice UPC programmers

Signaling Put Overview

- **Friendly, high-performance interface for a synchronizing, one-sided data transfer**
  - Want an easy-to-use and obvious interface
- **Provide coupled data transfer & synchronization**
  - Get overlap capability and low-latency end-to-end
  - Simplify optimal implementations by expressing the right semantics
  - Without the downfalls of full-blown message passing
    - still one-sided in flavor, no unexpected messages, no msg ordering costs
  - Similar to signaling store operator (`:-`) in Split-C, with improvements

Thread 1

```
bupc_memput_signal(…,sem);
/* overlap compute */
```

Thread 0

```
bupc_sem_t *sem = …;

bupc_sem_wait(sem);
/* consume data */
```

memput_signal:
latency ~0.5 round-trips
allows overlap
easy to use

Point-to-Point Synchronization: Signaling Put Interface

- **Simple extension to upc_memput interface**

```
void bupc_memput_signal(shared void *dst, void *src, size_t nbytes,
                        bupc_sem_t *s, size_t n);
```

  - Two new args specify a semaphore to signal on arrival
  - Semaphore must have affinity to the target
  - Blocks for local completion only (doesn't stall for ack)
  - Enables implementation using a single network message
- **Async variant**

```
void bupc_memput_signal_async(shared void *dst, void *src, size_t nbytes,
                              bupc_sem_t *s, size_t n);
```

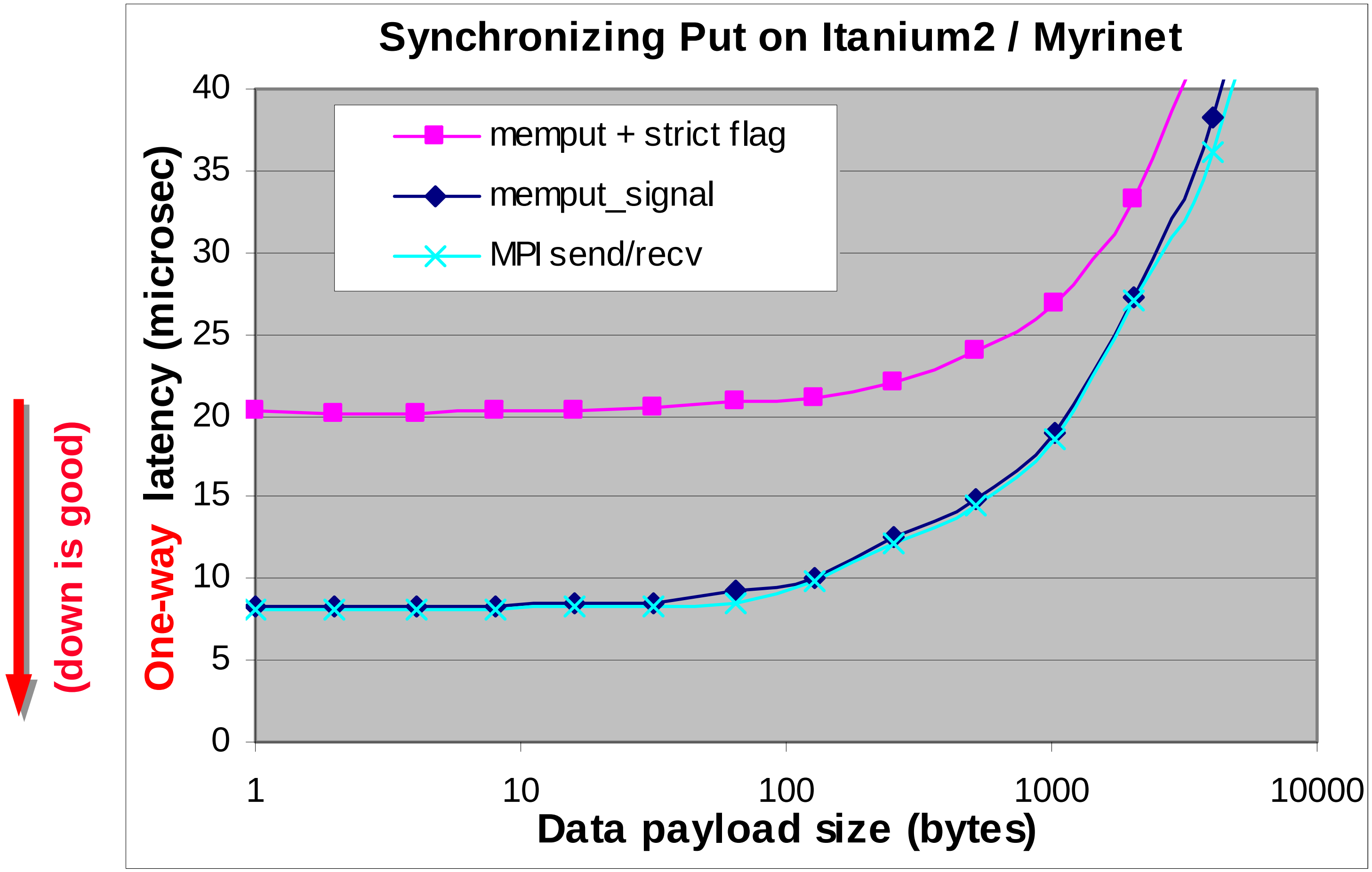
  - Same except doesn't block for local completion
  - Analogous to MPI_ISend
  - More overlap potential, higher throughput for large payloads

Point-to-Point Synchronization: Semaphore Interface

- **Consumer-side sync ops - akin to POSIX semaphores**
  - **`void bupc_sem_wait(bupc_sem_t *s);`** block for signal "*atomic down*"
  - **`int bupc_sem_try(bupc_sem_t *s);`** test for signal "*test-and-down*"
  - Also variants to wait/try multiple signals at once "*down N*"
  - All of these imply a upc_fence
- **Opaque sem_t objects**
  - Encapsulation in opaque type provides implementation freedom
  - **`bupc_sem_t *bupc_sem_alloc(int flags);`** ← non-collectively creates a sem_t object with affinity to caller
  - **`void bupc_sem_free(bupc_sem_t *s);`**
  - flags specify a few different usage flavors
    - eg one or many producer/consumer threads, integral or boolean signaling
- **Bare signal operation with no coupled data transfer:**
  - **`void bupc_sem_post(bupc_sem_t *s);`** signal sem "*atomic up (N)*"
  - post/wait sync that might not exactly fit the model of signaling put

Microbenchmark Performance of Signaling Put

Signaling Put: Microbenchmarks

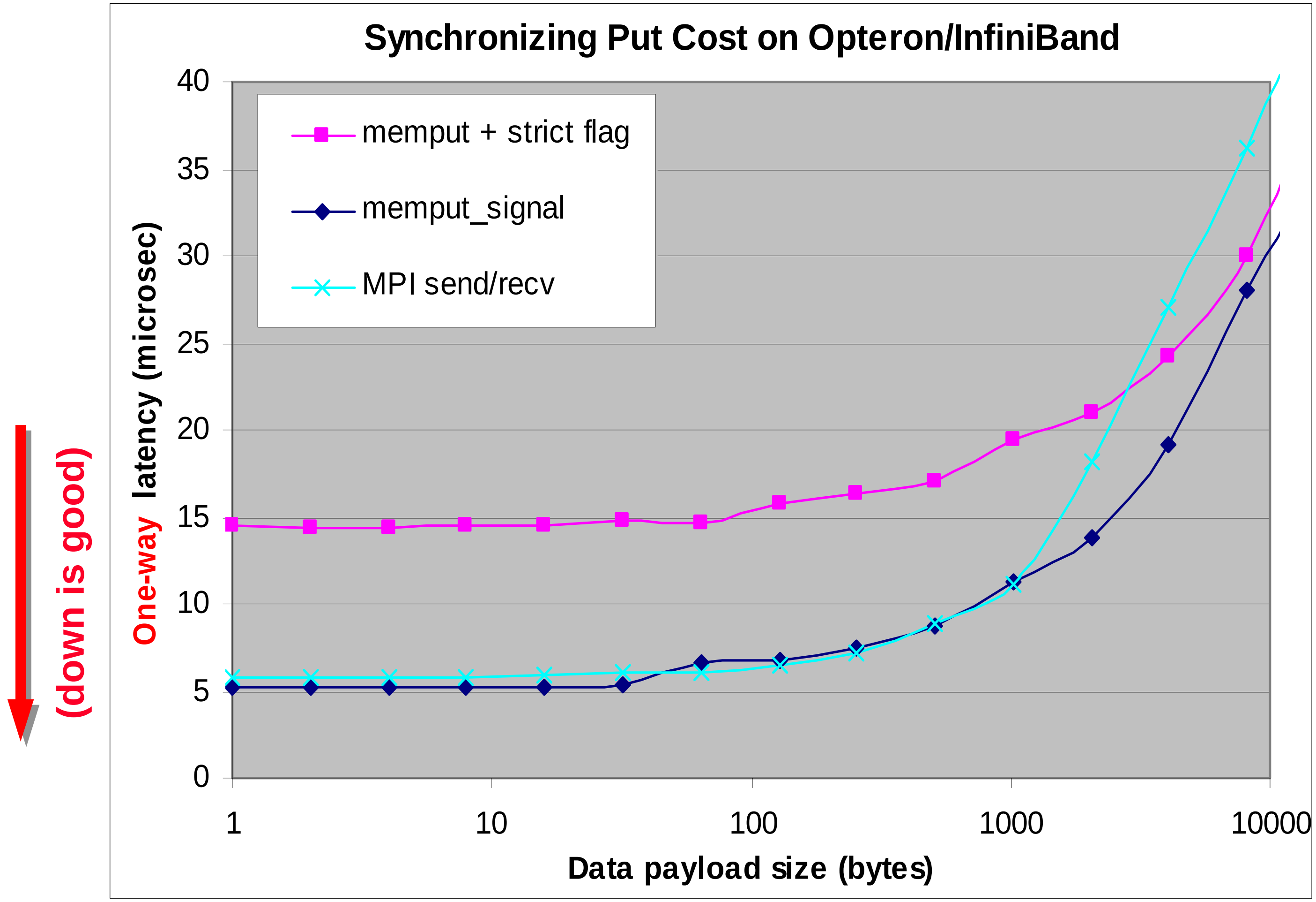


RDMA put or message send latency: ~13 us round-trip

CITRIS @ UC Berkeley
1.3 GHz Itanium-2
Myrinet PCI-XD
MPICH-GM 1.2.6..14a
Linux 2.4.20

- **memput (roundtrip) + strict put: Latency is ~ 1½ RDMA put roundtrips**
- **bupc_sem_t: Latency is ~ ½ message send roundtrip**
  - same mechanism used by eager MPI_Send - so performance closely matches

Signaling Put: Microbenchmarks



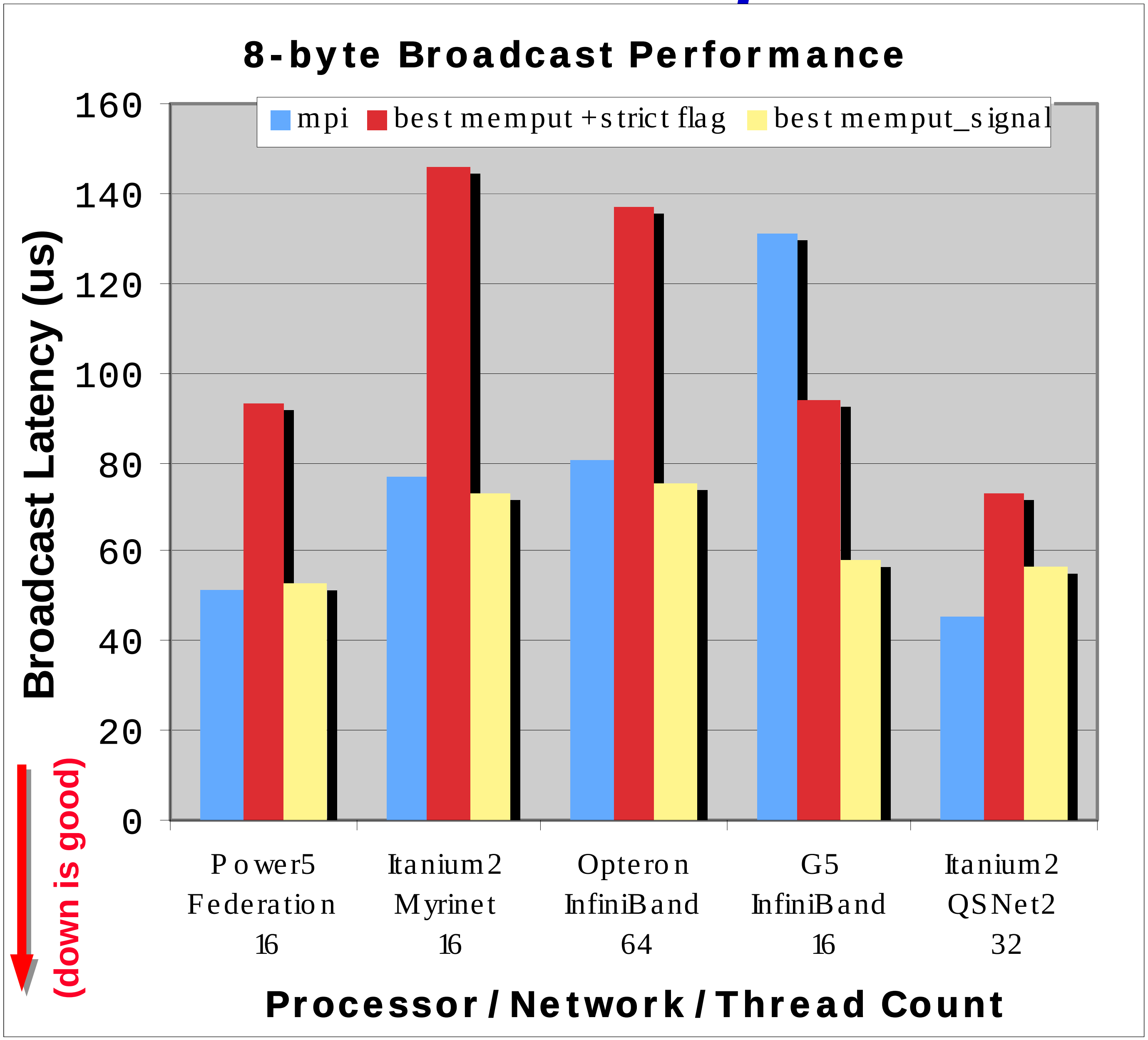
RDMA put latency:
~10.5us round-trip

Jacquard @ NERSC
2.2 GHz Opteron
Mellanox InfiniBand 4x
Linux 2.6.5-7.276
MVAPICH 0.9.5-mlx1.0.3

- **memput (roundtrip) + strict put: Latency is ~1½ RDMA put roundtrips**
- **bupc_sem_t: Latency is ~½ RDMA put roundtrip**
  - sem_t and MPI both using a single RDMA put, at least up to 1KB

Using Signaling Put to Implement Tree-based Collective Communication

Performance Comparison: UPC Broadcast

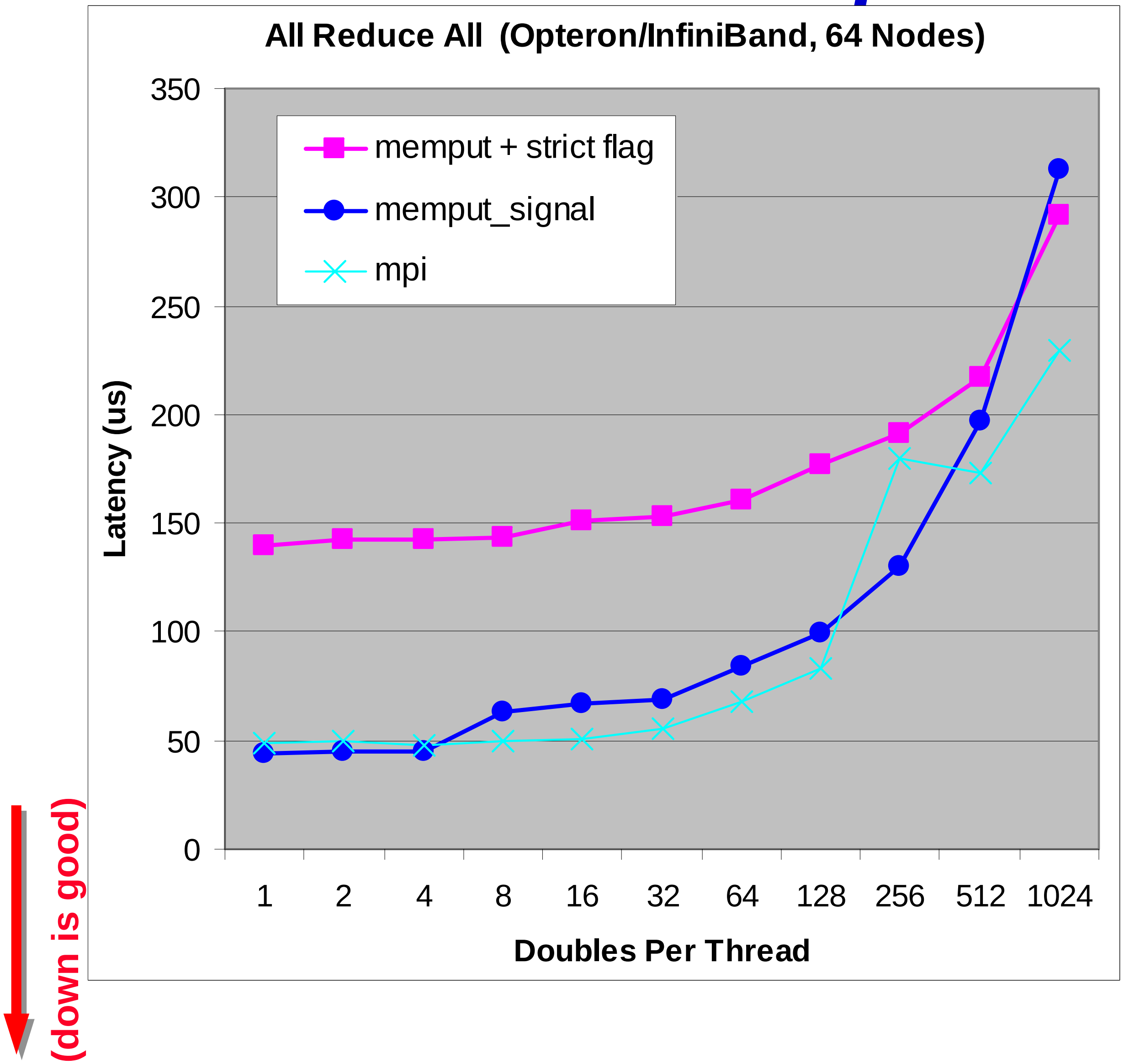


UPC-level implementation of collectives

Tree-based broadcast - show best performance across tree geom.

memput_signal competitive with MPI broadcast (shown for comparison)

Performance Comparison: All-Reduce-All



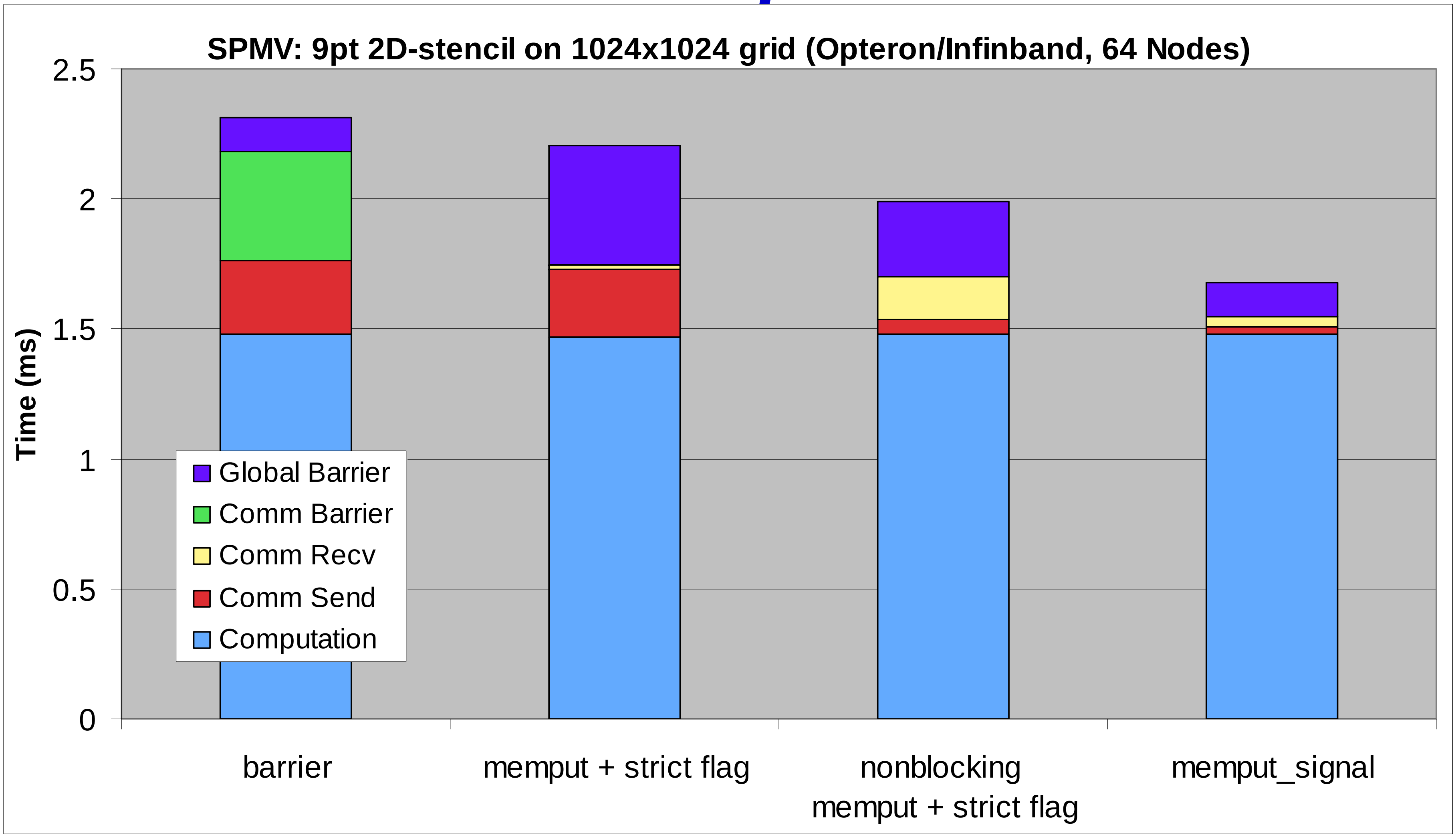
Dissemination-based implementations of all-reduce-all collective

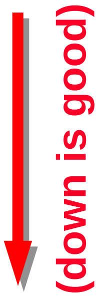
memput_signal consistently outperforms memput+strict flag, competitive w/ MPI

Over a 65% improvement in latency at small sizes

Using Signaling Put in Application Kernels

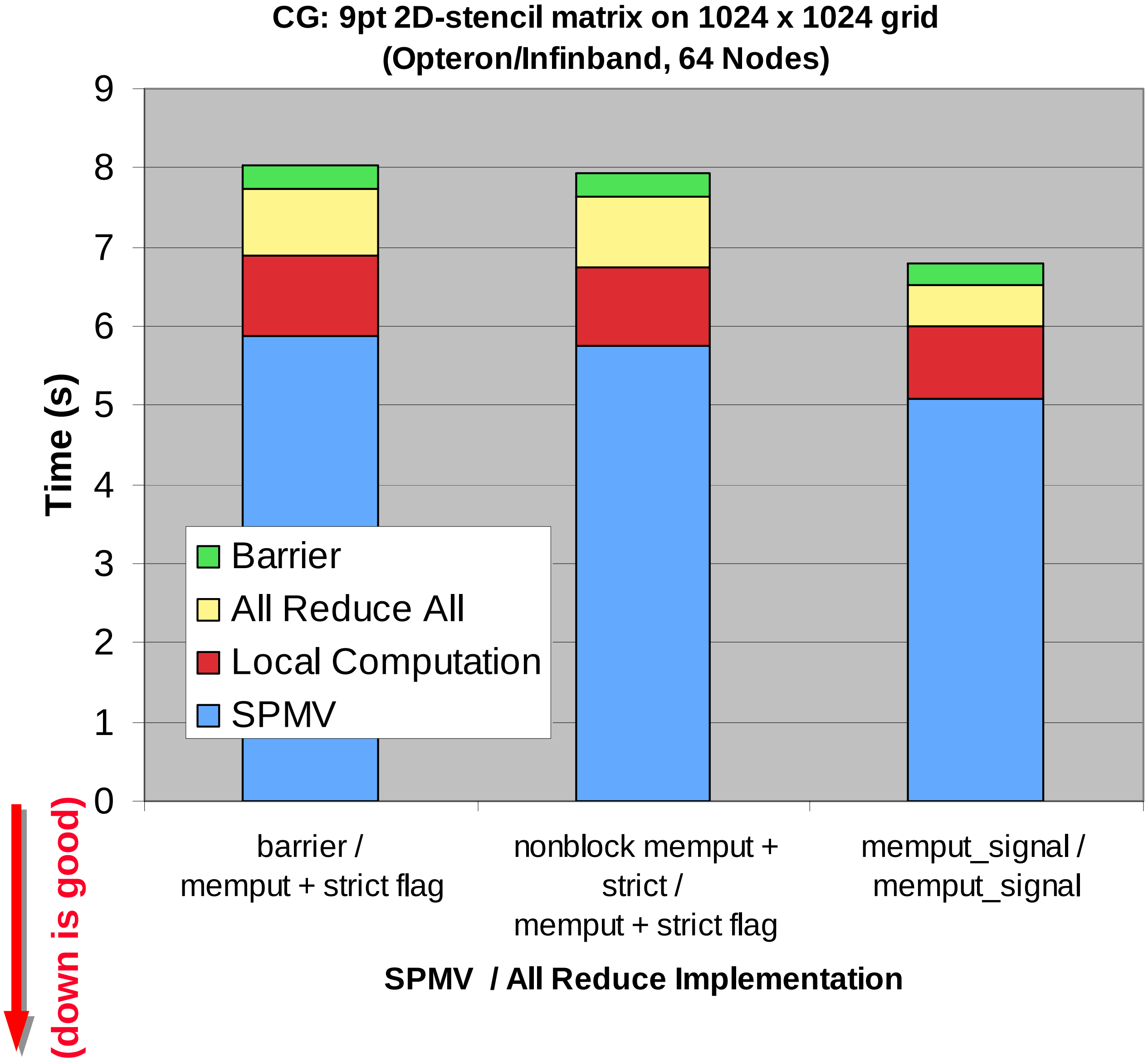
Performance Comparison: SPMV





75% improvement in synchronous communication time
28% improvement in total runtime (relative to barrier)

Performance Comparison: Conjugate Gradient



Incorporates both SPMV and All Reduce into an app kernel

memput_signal speeds up both SPMV and All Reduce portions of the application

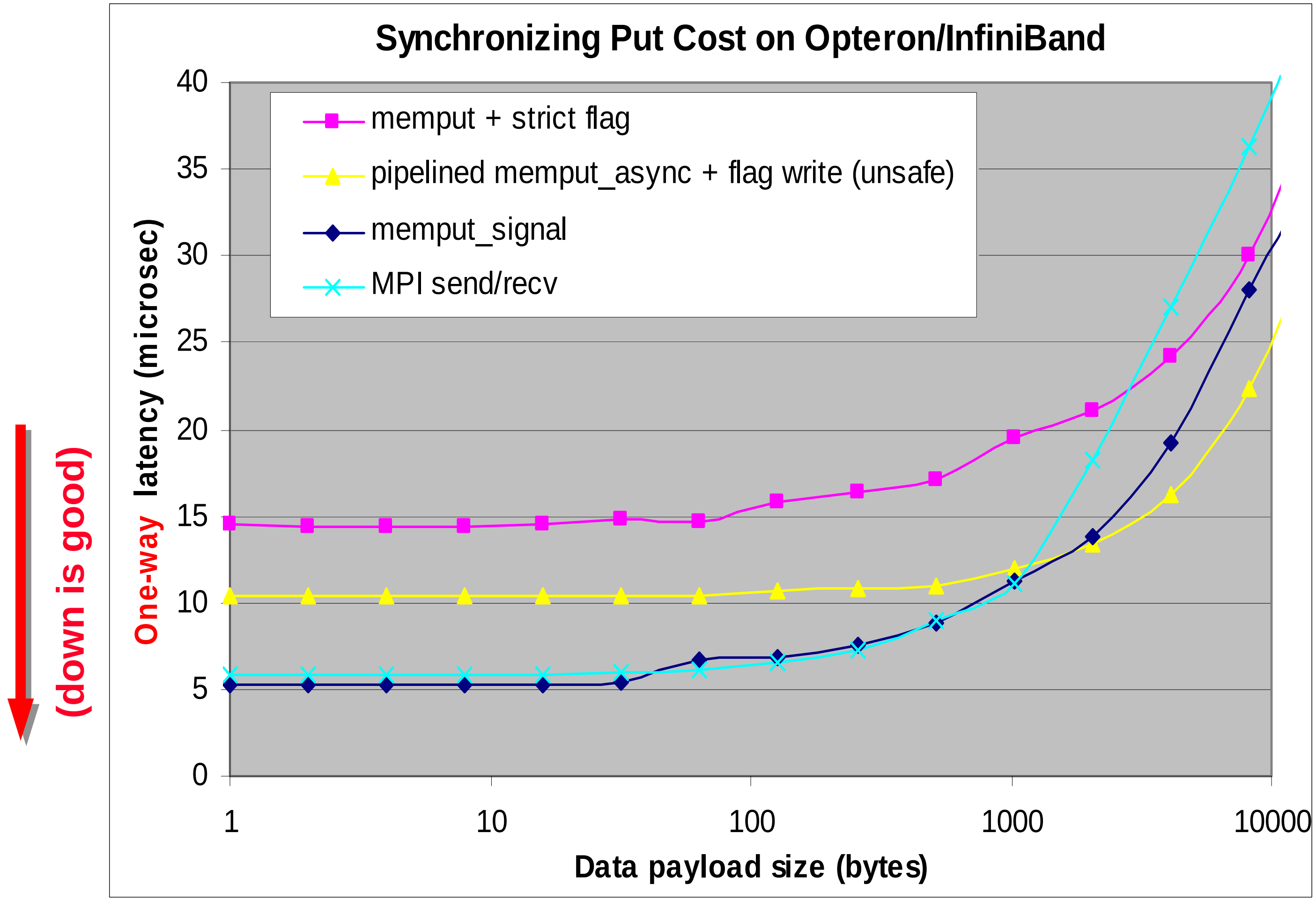
Leads to an 15% improvement in overall running time

Conclusions

- **Proposed a signaling put extension to UPC**
  - Friendly interface for synchronizing, one-sided data transfers
    - Allows coupling data transfer & synchronization when needed
    - Concise and expressive
  - Enable high-perf. implementation by encapsulating the right semantics
    - Allows overlap and low-latency, single message on the wire
  - Provides the strengths of message-passing in a UPC library
    - Remains true to the one-sided nature of UPC communication
    - Avoids the downfalls of full-blown message passing
- **Implementation status**
  - Functional version available in Berkeley UPC 2.2.2
  - More tuned version available in 2.3.16 and upcoming 2.4 release
- **Future work**
  - Need more application experience
  - Incorporate extension in future revision of UPC standard library

BACKUP SLIDES

Signaling Put: Pipelining Notify



RDMA put latency:
~10.5us round-trip

Jacquard @ NERSC
2.2 GHz Opteron
Mellanox InfiniBand 4x
Linux 2.6.5-7.276
MVAPICH 0.9.5-mlx1.0.3

- **Yellow line is two back-to-back RDMA puts (payload, then flag)**
  - Relies on point-to-point ordered delivery guarantees in hardware (unsafe in general)
- **Represents expected performance of an interface that separates put + notify**
  - Still not competitive with best approaches, which win by using only one RDMA put

memput_signal vs Multi-version variables

- memput_signal semantically still a put operation
  - **doesn't manage overwriting of target buffer**
- burden:
  - **user has to decide when can safely overwrite target**
- opportunity:
  - **doesn't impose additional costs for handshaking on target bufs**
    - **algorithm might already provide that sync at a higher level**
  - **fully one-sided**
    - **op can always be retired without any help from target**
  - **zero-copy**
    - **without extra buffer space on order of payload sz**
    - **without rendezvous overheads/delays**
- allows writing to a small stripe of a larger object
- gives you the tools to implement something like MVV?

Split-C Signaling Store

- Signaling Store syntax: g `:- e`
  - **g is a global l-value, e is arbitrary expression.**
  - **initiates a transfer of the value of e into the location g**
  - **does not wait for remote completion**
- Non-collective completion: `store_sync(nbytes)`
  - **wait for nbytes to arrive at this target thread**
- Collective completion: `all_store_sync()`
  - **Global barrier that also waits for all stores to finish system-wide**
- Limitations:
  - **byte-oriented completion: target must know exact payload size**
  - **signaling is anonymous: only allows one logical phase of incoming stores to be outstanding without ambiguity**
  - **no support for layered apps with data abstraction**
  - **bulk/aggregate transfers require a separate (library) interface**
    - **blocks for local completion, limiting overlap + BW for large xfers**

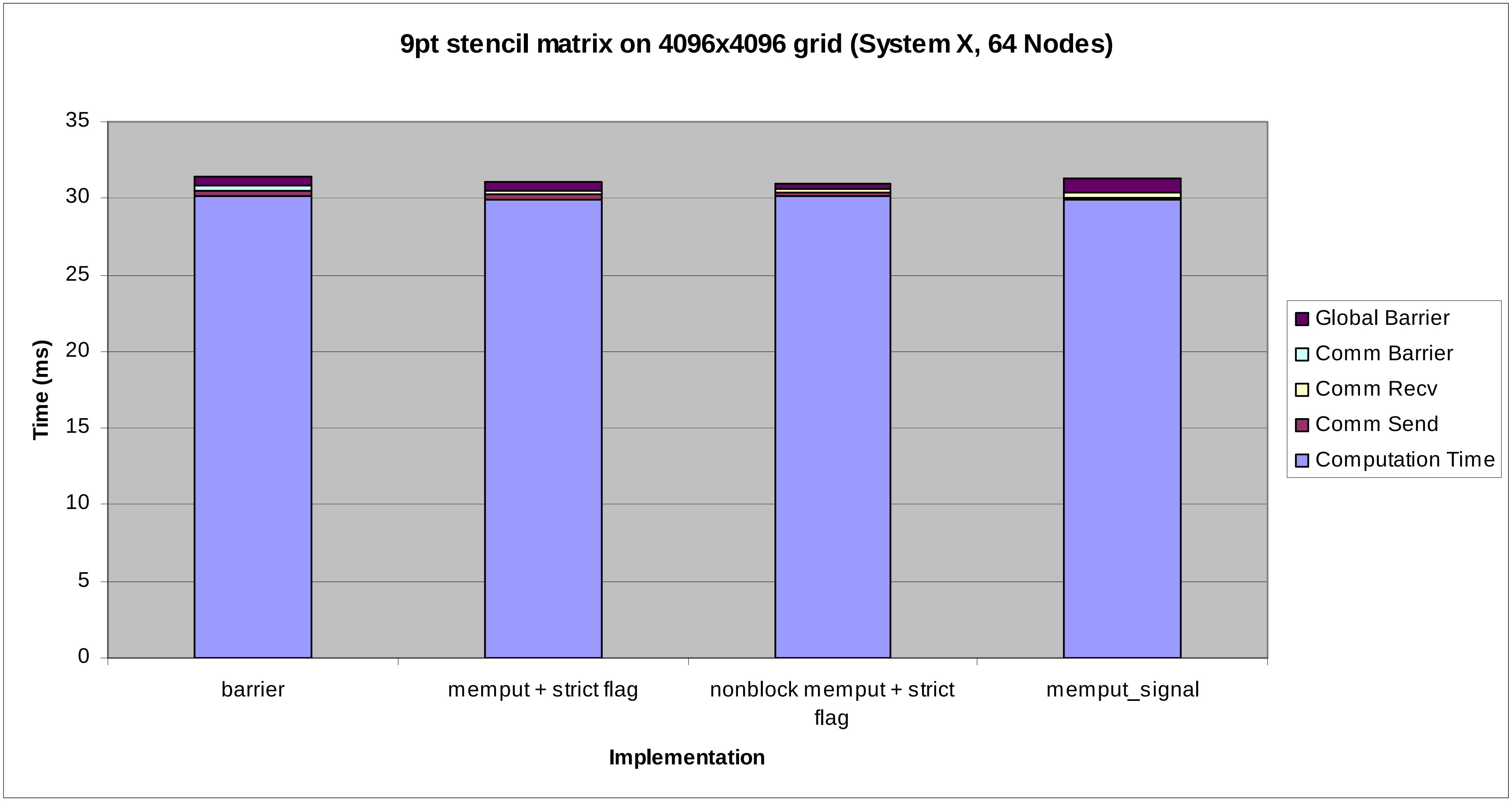
SPMV Expressiveness Comparison

- Common code
  - Sparse matrix mult. kernel and unpacking code
  - 13 lines common to all implementations
- barrier implementation (vanilla upc)
  - 13 common lines + 27 lines of code
- memput + strict flag
  - 13 common lines + 33 lines of code
- nonblock memput + strict flag
  - 13 common lines + 39 lines of code
- memput_signal
  - 13 common lines + 29 lines of code

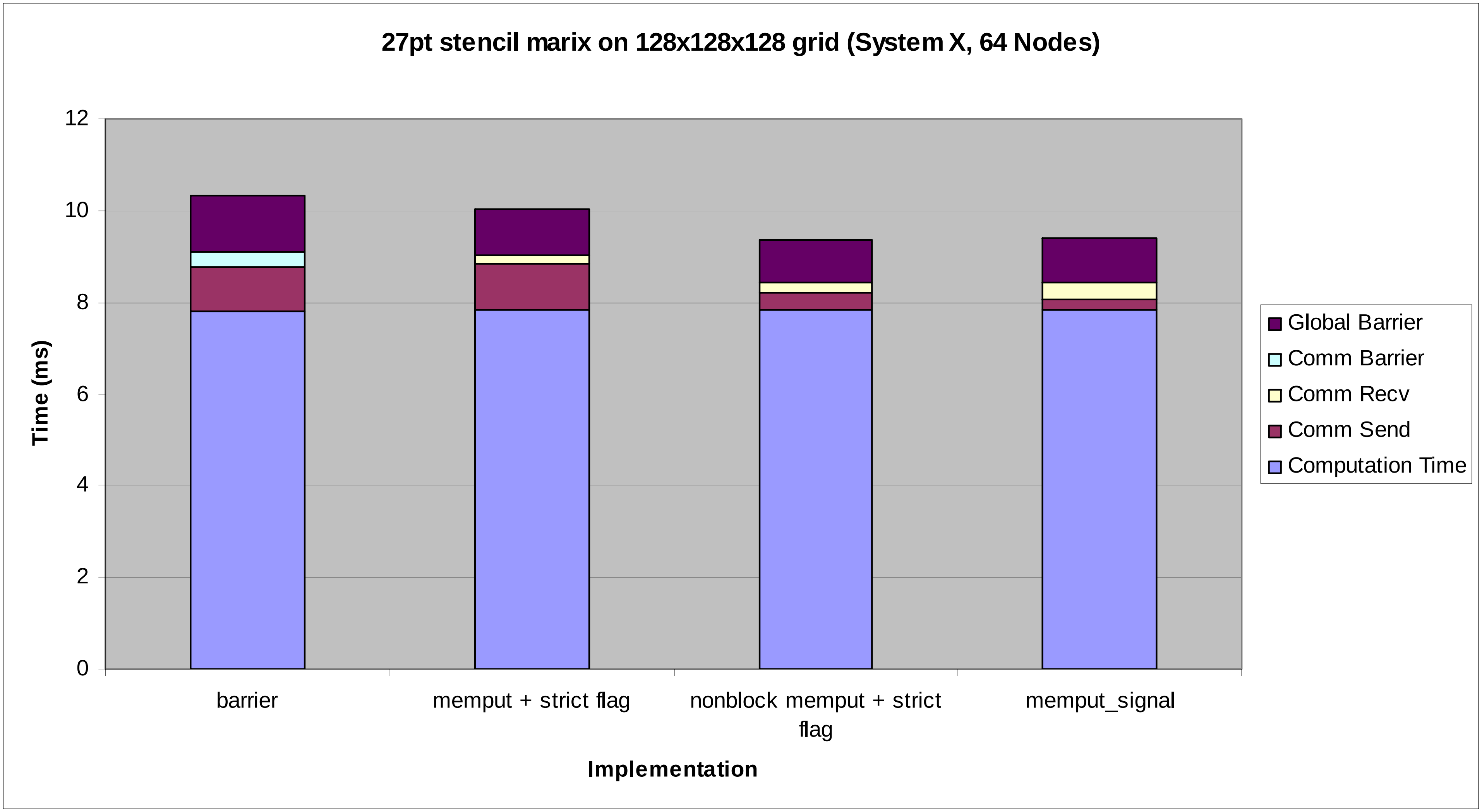
Signaling Put Implementation

- **Berkeley implementation uses a combination of:**
  - GASNet Active Messages - zero-copy transfer
  - Tinysem put - single put optimization via bounce-buffers
- **Tinysem put: minimize latency for small payloads**
  - Some networks (Infiniband, Quadrics) the lowest-latency point-to-point operation is a single RDMA put
  - Problem: need to safely detect completion at target
    - Fastest RDMA puts do not provide target-side notification
    - "waiting for the last byte to change" unsafe on many platforms
  - Approach: single put to a bounce-buffer FIFO at target
    - dynamically establish FIFO's btw threads that communicate
    - put includes payload and a header which contains size & checksum
    - header is sent doubled onto a 0/-1 region to allow reliable reception
    - payload is sent onto a zeroed region and checksum is zero count

SPMV (Computation Dominant)



SPMV (3D 27 point Stencil)

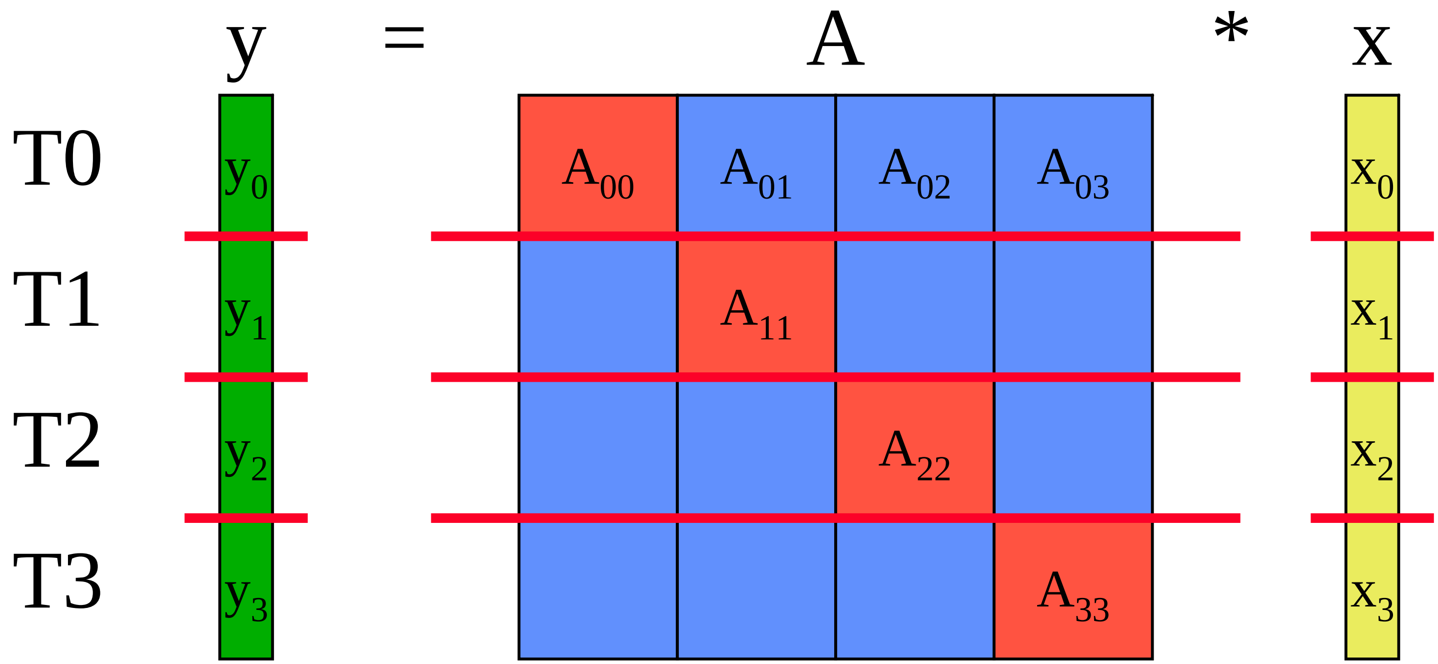


Algorithm Pseudocode

Case Study: Sparse Matrix Vector Multiply

- Sparse Matrix Vector Multiply (SPMV): y = A*x
  - y and x are dense vectors that are partitioned across the threads
    - shared [*] double x[n];   shared [*] double y[m];
  - A is an m x n sparse matrix
    - We use 9pt stencil matrices in our benchmarks
    - Partitioned block row wise such that each thread has a m/THREADS x n block of the matrix
    - Since x is also partitioned we need remote data to perform the multiplication
- Algorithm:
  - Initiate puts of your portion of x to all the other processors that need it
  - Perform local computation on portion of matrix that only requires local pieces of x
  - For each portion of the matrix that requires a remote portion of x
    - Wait for the processor responsible for that remote piece to send it to us
    - Perform computation on that portion of the matrix

SPMV Diagram



$$y_0 = \underbrace{A_{00} * x_0}_{\text{Can be done w/o comm}} + \underbrace{A_{01} * x_1 + A_{02} * x_2 + A_{03} * x_3}_{\text{Needs comm}}$$

Barrier SPMV Algorithm

- **for i=1:THREADS-1**
  - p = (MYTHREAD - i) %THREADS
  - If I need to send anything to p
    - pack src vector destined for p
    - **memput packed data to p**
- **Do Local SPMV on Diagonal Block**
- **BARRIER**
- **for i=1:THREADS-1**
  - p = (MYTHREAD+i) % THREADS
  - If I expect anything from p
    - Unpack data from p
    - Do SPMV on block p

Non-Blocking Barrier SPMV Algorithm

- **for i=1:THREADS-1**
  - p = (MYTHREAD - i) %THREADS
  - If I need to send anything to p
    - pack src vector destined for p
    - **Initiate async memput packed data to p**
- **Do Local SPMV on Diagonal Block**
- **Wait for all memputs to finish**
- **BARRIER**
- **for i=1:THREADS-1**
  - p = (MYTHREAD+i) % THREADS
  - If I expect anything from p
    - Unpack data from p
    - Do SPMV on block p

memput + strict flag SPMV Algorithm

- **for i=1:THREADS-1**
  - p = (MYTHREAD - i) %THREADS
  - If I need to send anything to p
    - pack src vector destined for p
    - **memput packed data to p**
    - **strict put flag to p**
- **Do Local SPMV on Diagonal Block**
- **for i=1:THREADS-1**
  - p = (MYTHREAD+i) % THREADS
  - If I expect anything from p
    - **while (flags[p] ==0) bupc_poll();**
    - Unpack data from p
    - Do SPMV on block p

Non-blocking memput + strict SPMV

- **for i=1:THREADS-1**
  - p = (MYTHREAD - i) %THREADS
  - If I need to send anything to p
    - pack src vector destined for p
    - **async memput packed data to p**
- **Do Local SPMV on Diagonal Block**
- **for i=1:THREADS-1**
  - p = (MYTHREAD - i) %THREADS
  - If I sent anything to p
    - **Wait for memput to finish**
    - upc_fence;
    - **Initiate nonblock flag put to p**
- **for i=1:THREADS-1**
  - p = (MYTHREAD+i) % THREADS
  - If I expect anything from p
    - **while (flags[p] ==0) bupc_poll();**
    - Unpack data from p and do SPMV on block p
- **Wait for all nonblock flags to finish**

Memput_signal SPMV Algorithm

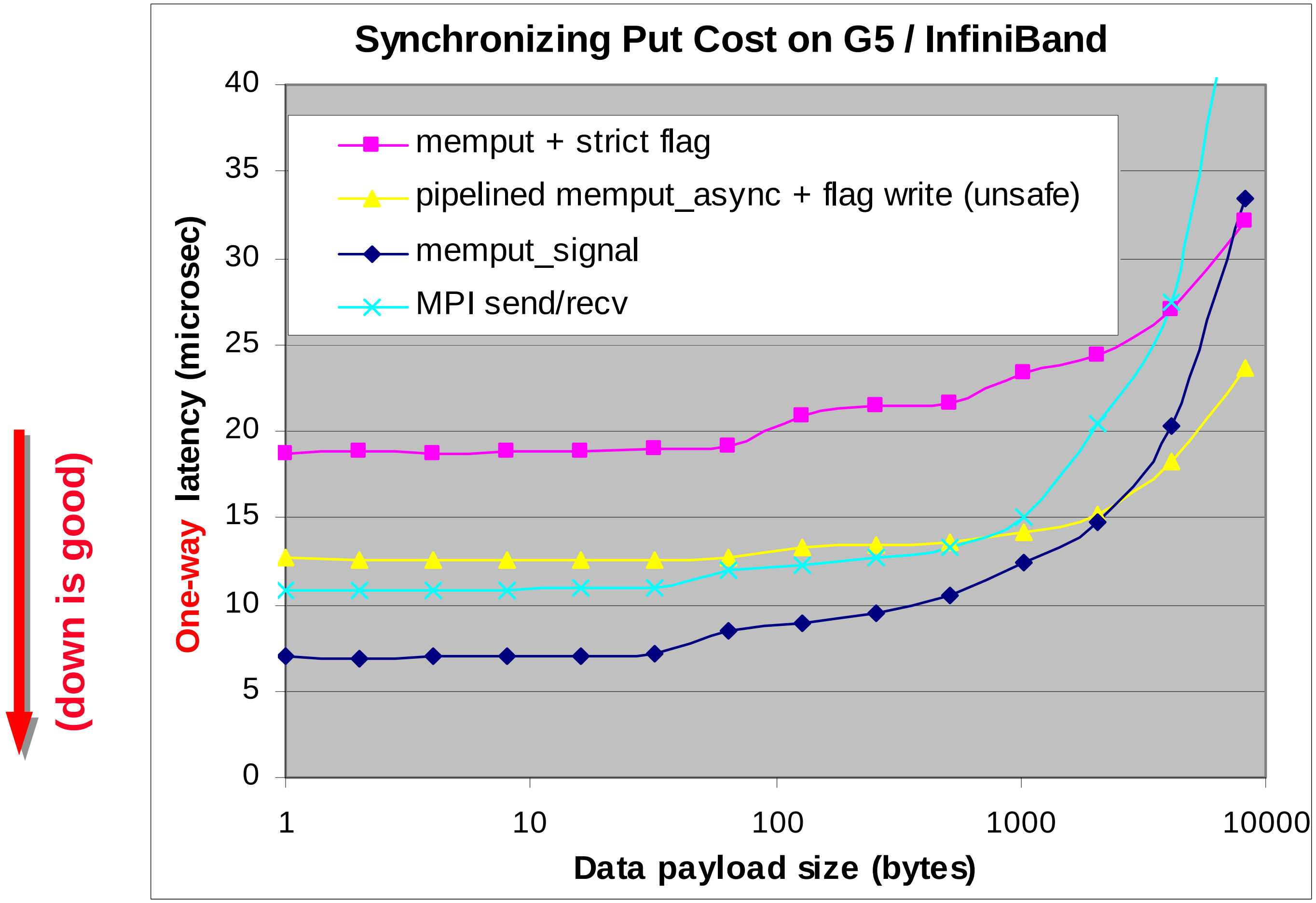
- **for i=1:THREADS-1**
  - p = (MYTHREAD - i) %THREADS
  - If I need to send anything to p
    - pack src vector destined for p
    - **async memput_signal packed data to p**

- **Do Local SPMV on Diagonal Block**

- **for i=1:THREADS-1**
  - p = (MYTHREAD+i) % THREADS
  - If I expect anything from p
    - **sem_wait on data from p**
    - Unpack data from p and do SPMV on block p

SYSX RESULTS

Signaling Put: Microbenchmarks



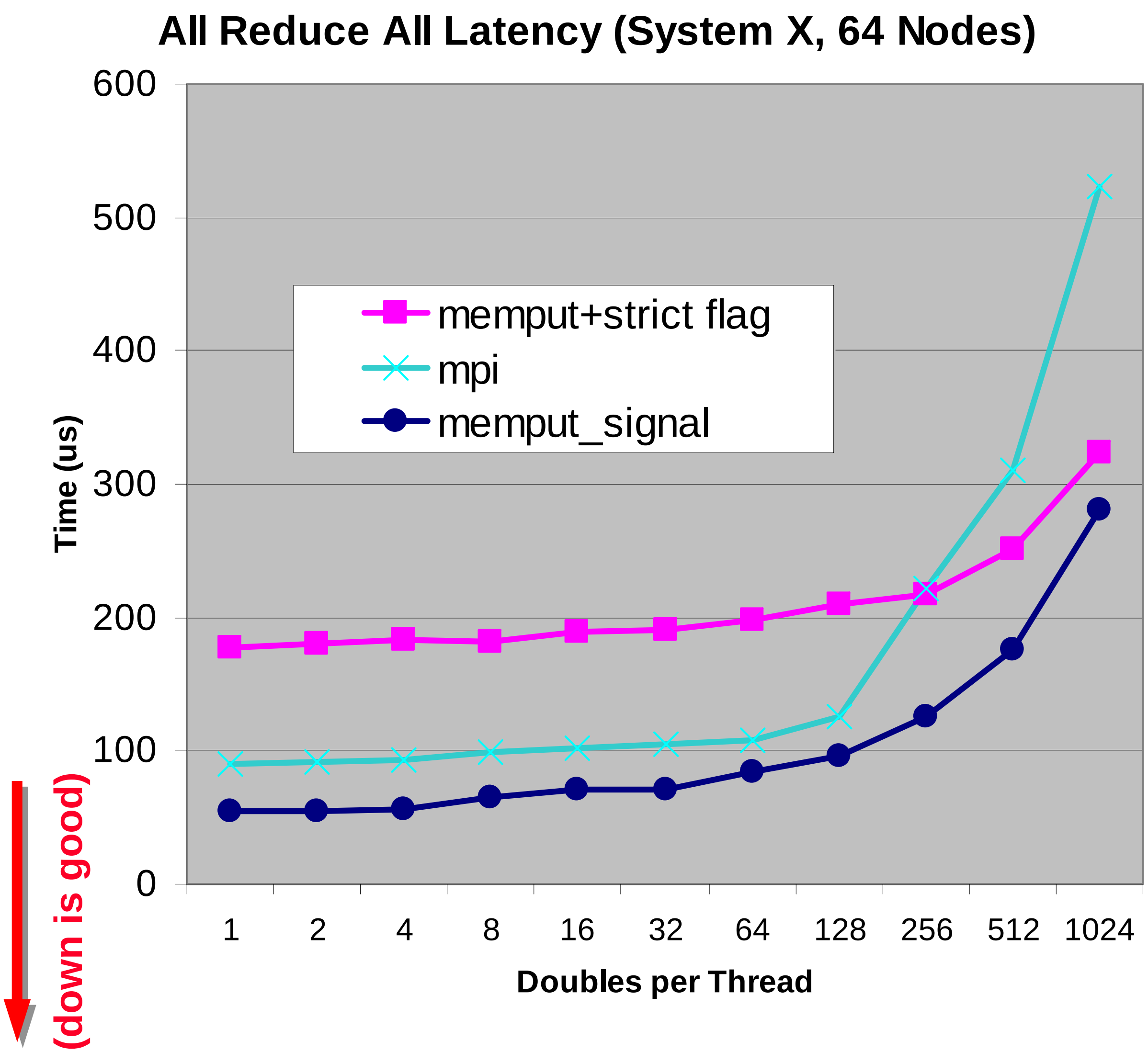
RDMA put latency:
~10.5us round-trip

Message latency:
~18us round-trip

System-X @ Virginia Tech
2.3 GHz G5 PPC
Mellanox Cougar InfiniBand 4x
OS X 10.3.8
MPICH 1.2.5

- **memput (roundtrip) + strict put: Latency is ~1½ RDMA put roundtrips**
- **bupc_sem_t: Latency is ~½ RDMA put roundtrip**
- **MPI is using VAPI msg send, which is slower than RDMA**

Performance Comparison: All-Reduce-All

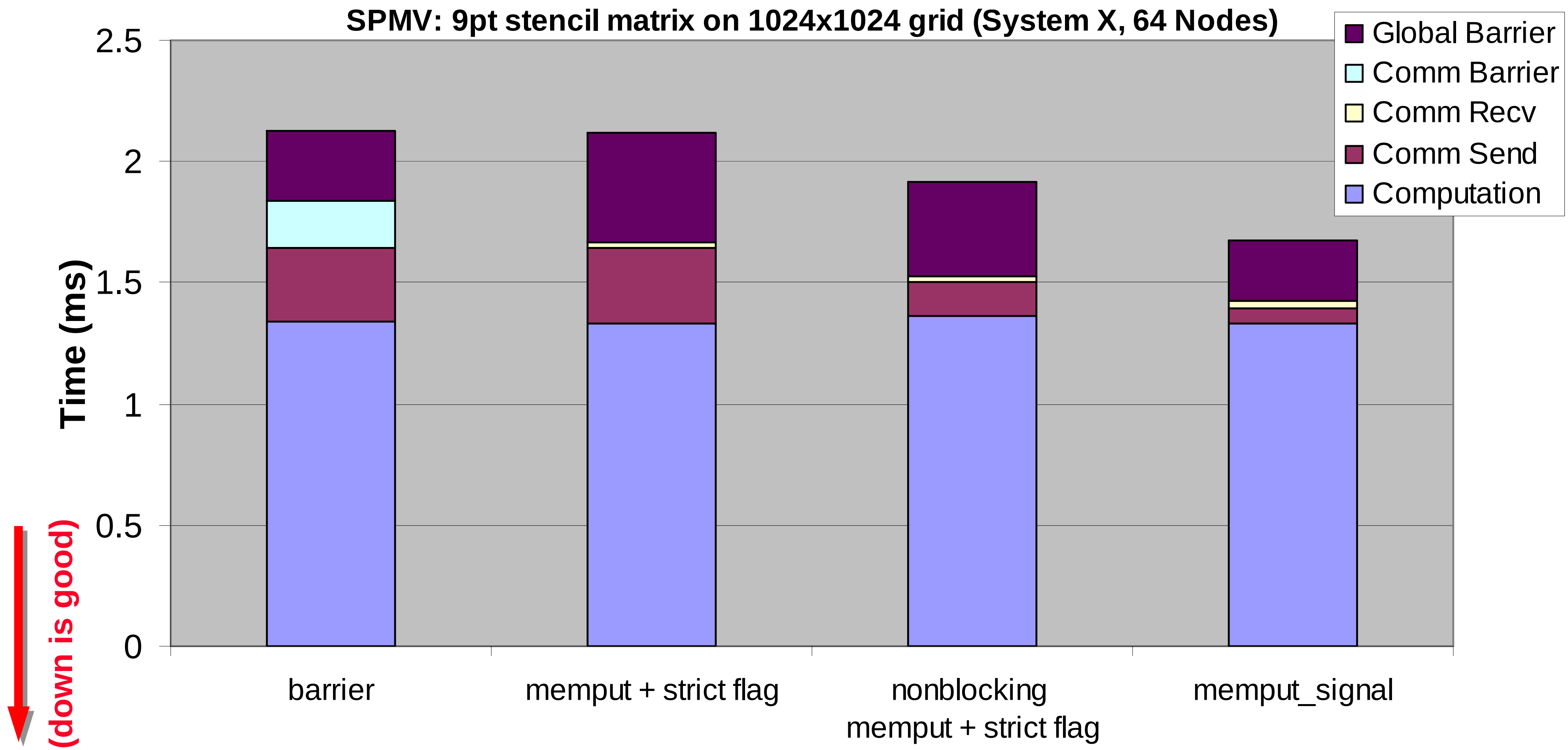


Dissemination-based implementations of UPC all-reduce-all collective

memput_signal consistently outperforms both mpi and memput+strict flag implementations

Over a 70% improvement in latency performance at small message sizes

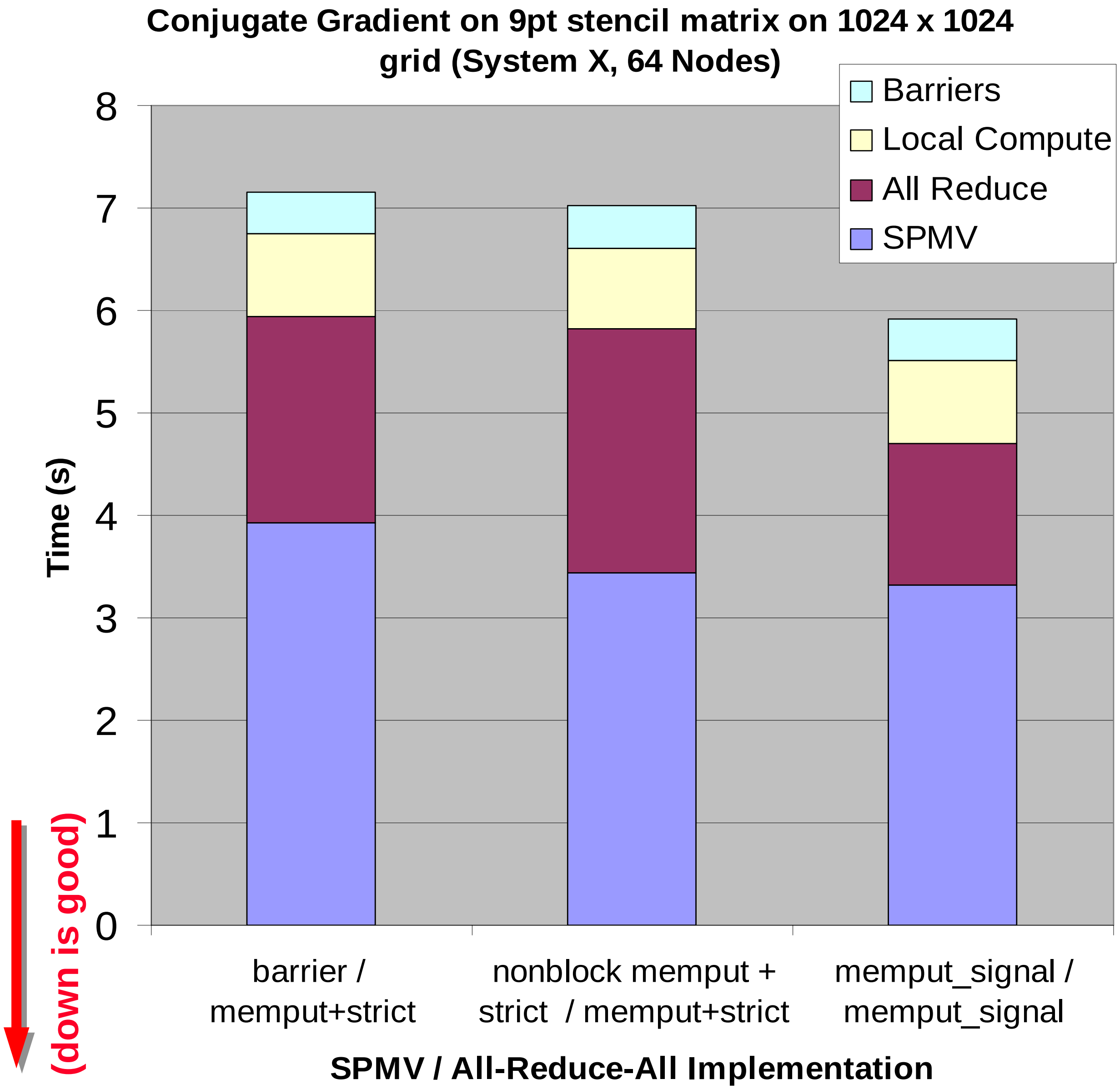
Performance Comparison: SPMV



60% improvement in synchronous communication time

20% improvement in total runtime

Performance Comparison: Conjugate Gradient

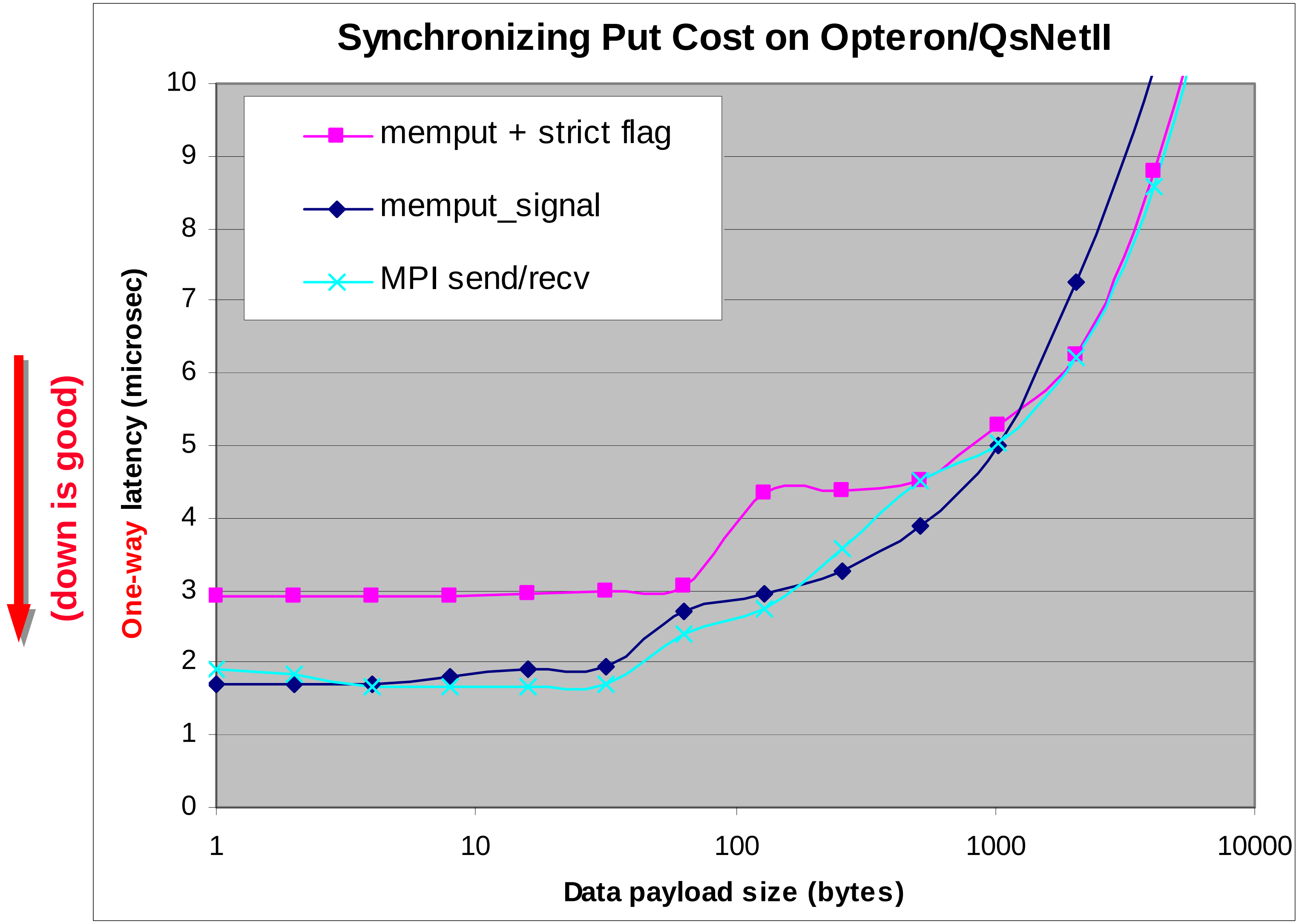


Incorporate both SPMV and All Reduce All into an application

memput_signal speeds up both SPMV and All Reduce portions of the application

Leads to an 18% improvement in overall running time

Signaling Put: on QsNet



RDMA put latency:
~1.4us round-trip

Hive @ LBNL
2.0 GHz Opteron
Quadrics QSNet2
Linux 2.6.8-24.11
Quadrics MPI

Point-to-Point Sync Data Xfer in UPC

Thread 1	Thread 0	
`upc_memput(…);` `upc_barrier;`	`upc_barrier;` `/* consume data */`	**barrier**: over-synchronizes threads, high-latency due to barrier no overlap opportunity

`upc_memput(…);`
`f = 1;`
	`strict int f = 0;`
	`while (!f) bupc_poll();`
	`/* consume data */`
		memput + strict flag: latency ~1.5 round-trips no overlap opportunity

`h = bupc_memput_async(…);`
`/* overlap compute */`
`bupc_waitsync(h);`
`upc_fence;`
`h2 = bupc_memput_async(&f,…);`
`/* overlap compute */`
`bupc_waitsync(h2);`
	`strict int f = 0;`
	`while (!f) bupc_poll();`
	`/* consume data */`
		non-blocking memput + strict flag: latency ~1.5 round-trips allows overlap higher complexity