

---

# ***Berkeley UPC***

## ***SESSION 2: Data Movement & Synchronization***

**Dan Bonachea**

Christian Bell, Wei Chen, Jason Duell, Paul Hargrove,  
Parry Husbands, Costin Iancu, Rajesh Nishtala,  
Mike Welcome, Kathy Yelick  
**U.C. Berkeley / LBNL**

**<http://upc.lbl.gov>**



## ***SESSION 2: Data Movement & Synchronization***

- **Explicitly non-blocking memcpy library**
  - How to overlap bulk communication with other work
- **Point-to-point synchronization library**



## ***Explicitly Non-blocking Memcpy Motivation***

- **Distributed-memory NIC hardware is naturally non-blocking**
  - Provide significant speedups by overlapping communication with computation or other communication
  - This feature is *crucial* to our UPC-FT impl., which outperforms MPI!
- **Allow programmer to directly express the lack of data dependencies using explicitly non-blocking bulk transfer operations**
  - The programmer often knows that given bulk data movements are completely independent, but it's often very difficult for a compiler to infer this info
    - due to buffer aliasing, data dependent operations, unexpressed restrictions on the set of legal inputs, etc
  - Proposed interface is easy to understand and use
    - Also trivial to implement on most network hardware



## *Explicitly Non-blocking Memcpy Interface*

- **Very simple extension to existing library:**

- New "flavors" of `upc_mem{put,get,copy}` with "`_async`" suffix:

```
upc_handle_t bupc_memcpy_async(shared void *dst, shared const void *src,  
                               size_t nbytes);
```

```
upc_handle_t bupc_memget_async(void *dst, shared const void *src,  
                               size_t nbytes);
```

```
upc_handle_t bupc_mempush_async(shared void *dst, const void *src,  
                                size_t nbytes);
```

- Same args and semantics as blocking variants, returns a `upc_handle_t`
  - an opaque handle representing the initiated asynchronous operation
  - analogous to `MPI_Request` object for `MPI_Isend/MPI_Irecv`
- Synchronized using one of two new functions:

- Block for completion (after overlapped work):

```
void bupc_waitsync(upc_handle_t handle);
```

- Non-blocking test for completion (useful for event-driven algorithms):

```
int bupc_trysync(upc_handle_t handle);
```



## *Explicitly Non-blocking Memcpy Example*

**Example of a nearest-neighbor data fetch on a regular 1-D blocked decomposition using `upc_memget_async`:**

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ], rightdata[BLKSZ]; /* local temporary buffers */
upc_handle_t leftfetch_handle = UPC_COMPLETE_HANDLE; /* handles */
upc_handle_t rightfetch_handle = UPC_COMPLETE_HANDLE;

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
    leftfetch_handle = bupc_memget_async(leftdata, &(A[BLKSZ*(MYTHREAD-1)]),
                                         BLKSZ*sizeof(double));
if (MYTHREAD < THREADS-1) /* initiate fetch of data from right neighbor */
    rightfetch_handle = bupc_memget_async(rightdata, &(A[BLKSZ*(MYTHREAD+1)]),
                                           BLKSZ*sizeof(double));

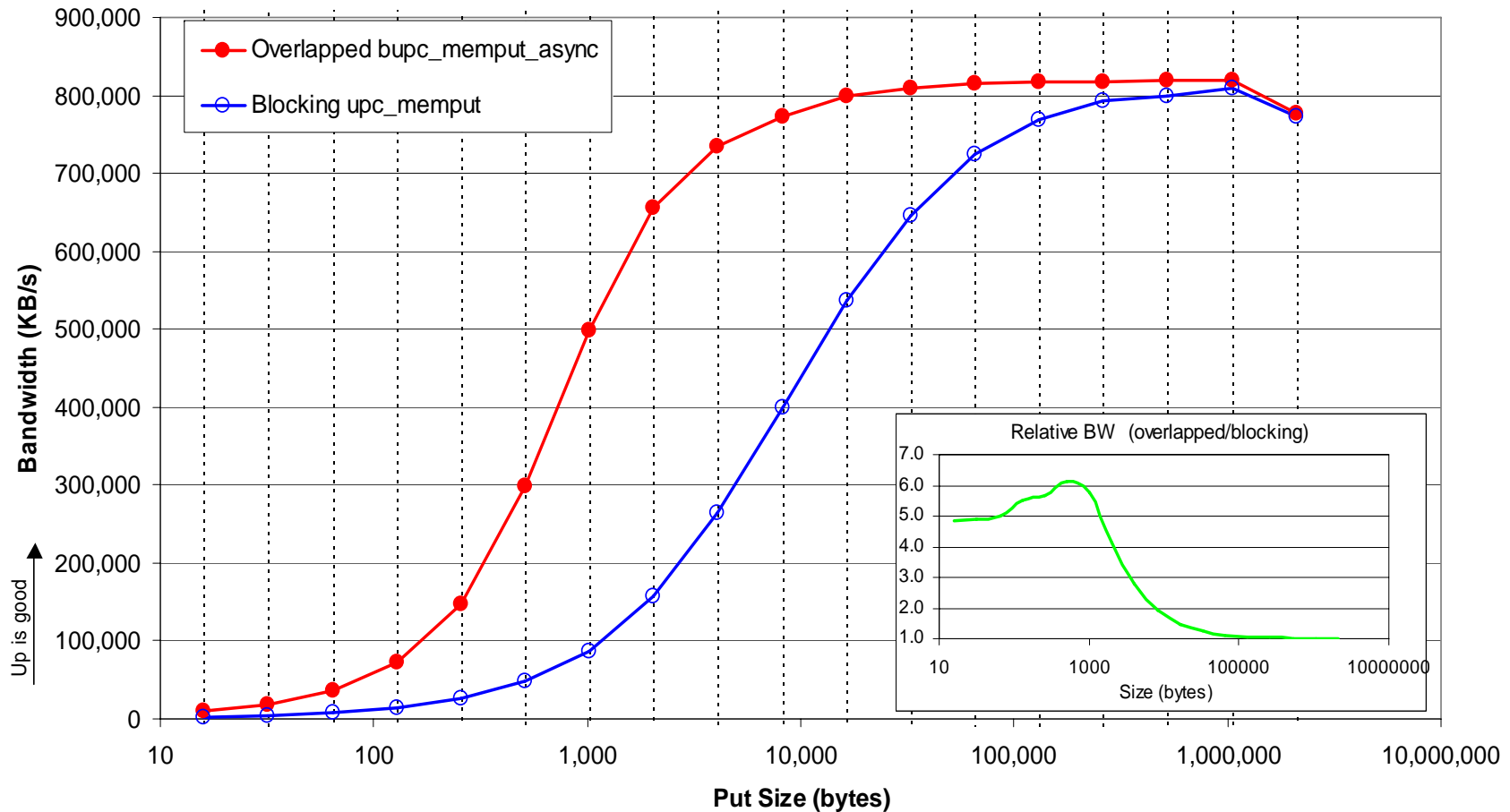
/* perform some independent overlapped computations here */

bupc_waitsync(leftfetch_handle); /* block for completion, if necessary */
bupc_waitsync(rightfetch_handle);

/* now safe to operate on leftdata and rightdata */
```



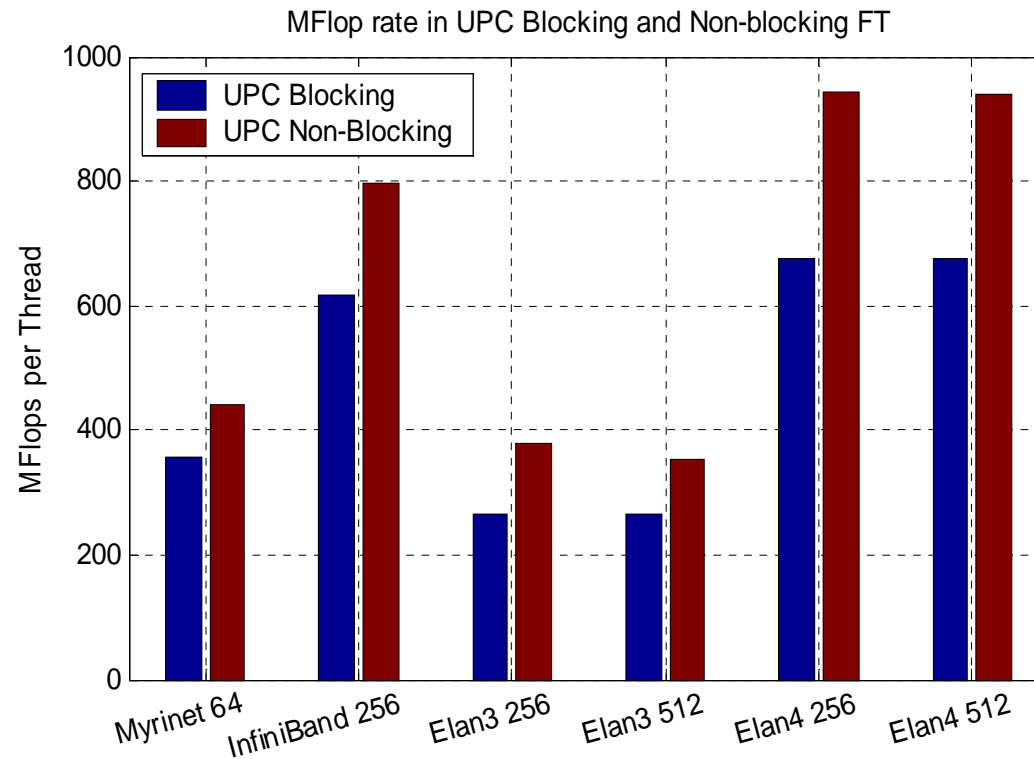
# Comm/Comm Overlap - Microbenchmark Performance



Machine: NERSC Jacquard 2.2 GHz Dual Opteron/ 4x Infiniband

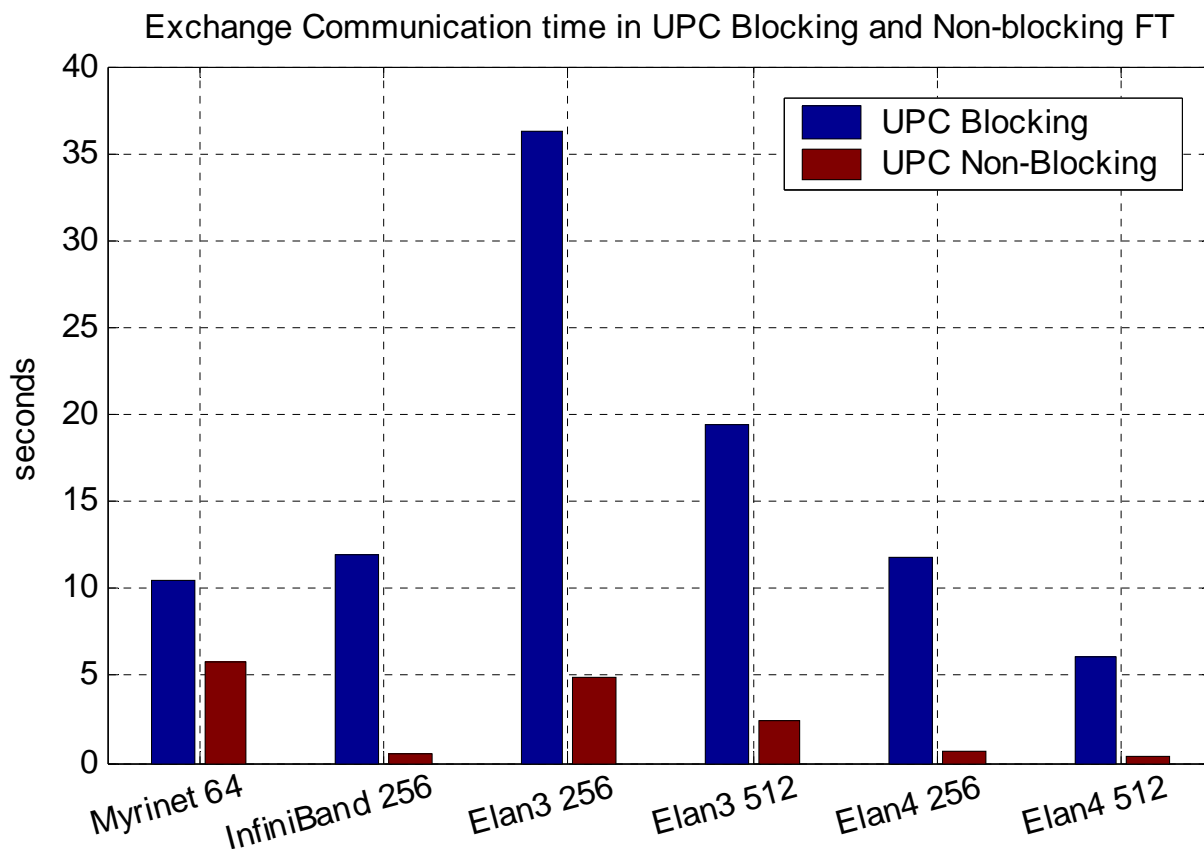


# NAS FT: UPC Non-blocking MFlops



- **UPC Non-blocking extensions produce 15-45% speedup over best UPC Blocking version**
- **Non-blocking version (`bupc_memput_async()`) of the algorithm requires about 30 extra lines of UPC code**

# NAS FT: UPC Communication time



- **15-45% Speedups in MFlops come from little to significantly less amounts of time spent in non-blocking communication initiation and completion (`bupc_memput_async()/bupc_waitsync()`) rather than a blocking exchange**

## *SESSION 2: Data Movement & Synchronization*

- **Explicitly non-blocking memcpy library**
- **Point-to-point synchronization library**
  - How to perform pairwise synchronization between threads



# *Point-to-Point Synchronization: Motivation*

- **Many algorithms need pairwise sync.**
  - Ability to couple a data transfer with remote notification
    - eg signalling store, for implementing producer/consumer codes
  - Message passing provides this sync. implicitly (whether you want it or not)
- **Need friendly interface for pairwise sync.**
  - Strict variables are sufficient, but not ideal
    - correctness can be subtle, limited overlap hurts performance
  - Want an easy-to-use and obvious interface
  - Allow more optimal network-specific implementations of signalling store than can be achieved using only strict



# Point-to-Point Synchronization: Semaphore Interface

- **Creation - opaque objects analogous to upc\_lock**
  - `bupc_sem_t *bupc_sem_alloc(int flags);`
  - `void bupc_sem_free(bupc_sem_t *s);`
  - flags specify a few different usage flavors
    - eg single or multiple producer/consumer threads, integral or boolean signaling
- **Sync operations - like POSIX semaphores**
  - Bare synchronization with no coupled data transfer:
  - `void bupc_sem_post(bupc_sem_t *s);` signal sem "*atomic up*"
  - `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 post/wait multiple signals at once "*up/down N*"
  - All of these imply a `upc_fence`



# *Point-to-Point Synchronization: Signaling Put Interface*

- **Provide coupled data transfer & synchronization**
  - without the downfalls of full-blown message passing
- **Simple extension to memput interface**

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

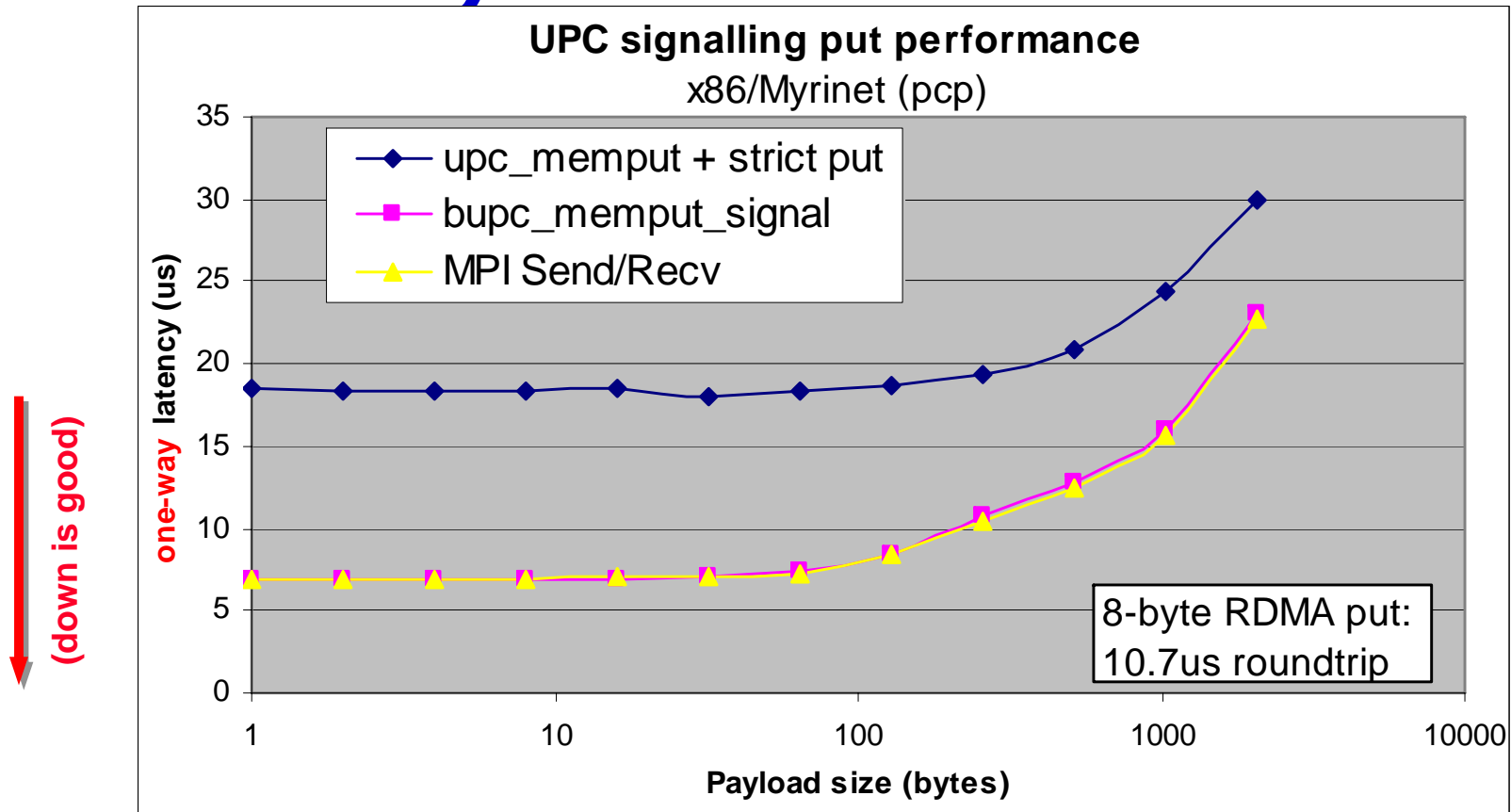
- Two new args specify a semaphore to signal on arrival
  - Semaphore must have affinity to the target
  - In many cases can be implemented using a single message
  - Blocks for local completion only (doesn't stall for ack)
- **Async variant**

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

- Same except doesn't even block for local completion



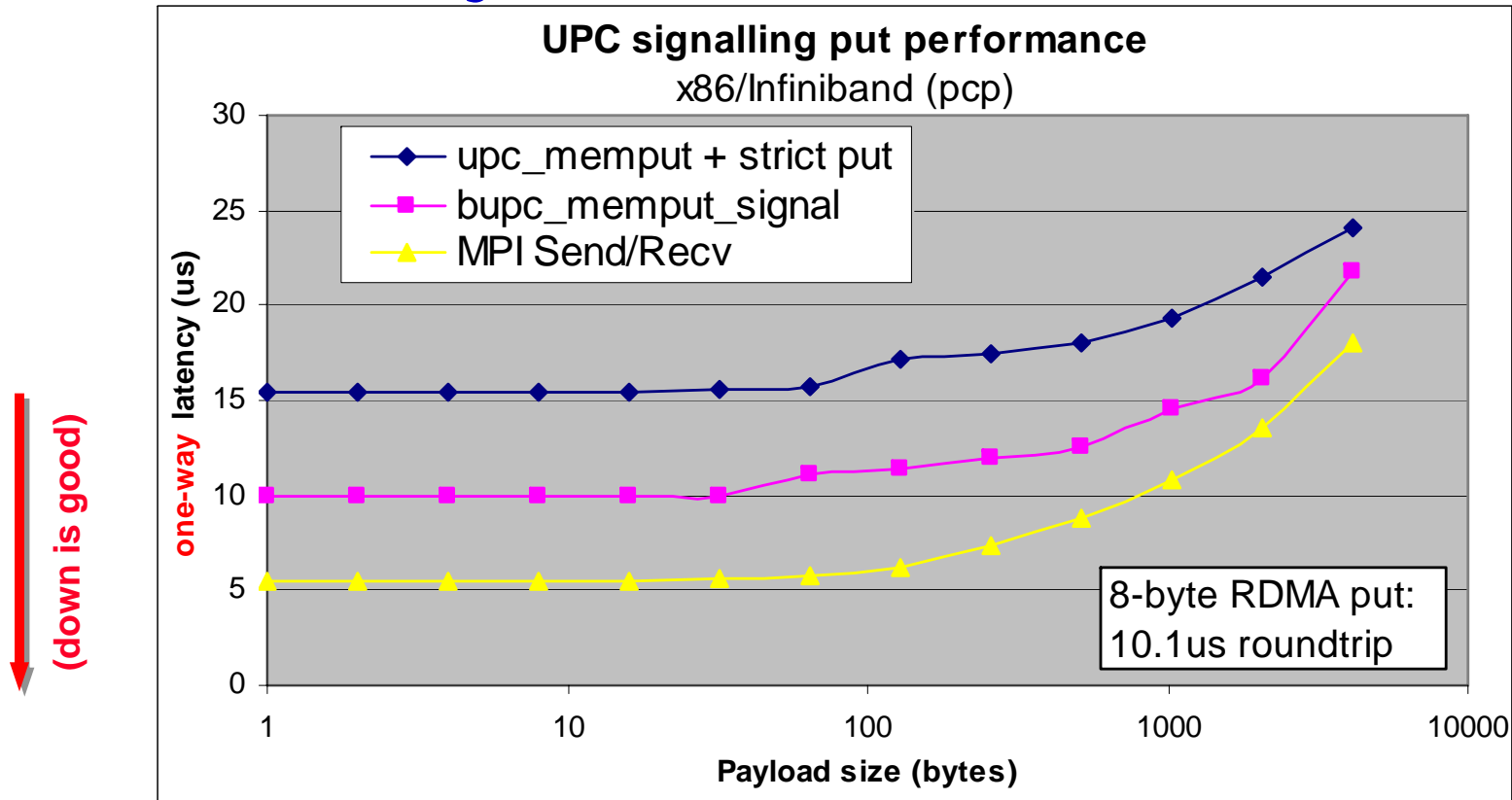
# Point-to-Point Synchronization: Preliminary Microbenchmark Results



- **memput (roundtrip) + strict put: Cost is roughly 1½ RDMA put roundtrips**
- **bupc\_sem\_t: Cost is ½ message send roundtrip**
  - same mechanism used by eager MPI\_Send - so performance closely matches



# Point-to-Point Synchronization: Preliminary Microbenchmark Results



- **memput (roundtrip) + strict put: Cost is 1½ RDMA put roundtrips**
- **bupc\_sem\_t: Cost is ½ message send roundtrip**
  - MPI wins by using single RDMA put for eager MPI\_Send
  - looking into adding a put-based algorithm to match MPI on vapi, further tuning underway



**Berkeley UPC**

<http://upc.lbl.gov>

**Dan Bonachea**

