Introduction to UPC

Presenter: Paul H. Hargrove (LBNL)

Joint work with Berkeley UPC and Titanium Groups at Lawrence Berkeley Nat’l Lab & UC Berkeley

Some slides adapted from Katherine Yelick and Tarek El-Ghazawi
Context

• Most parallel programs are written using either:
  – Message passing with a SPMD model
    • Usually for scientific applications with C++/Fortran
    • Scales easily
  – Shared memory with threads in OpenMP, Threads+C/C++/F or Java
    • Usually for non-scientific applications
    • Easier to program, but less scalable performance
• Global Address Space (GAS) Languages take the best of both
  – global address space like threads (programmability)
  – SPMD parallelism like MPI (performance)
  – local/global distinction, i.e., layout matters (performance)
Partitioned Global Address Space Languages

- Explicitly-parallel programming model with SPMD parallelism
  - Static - Fixed at program start-up, typically 1 thread per core
- Global address space model of memory
  - Allows programmer to directly represent distributed data structures
- Address space is logically partitioned
  - Local vs. remote memory (two-level hierarchy)
- Programmer control over performance critical decisions
  - Data layout and communication
- Performance transparency and tunability are goals
  - Initial implementation can use fine-grained shared memory
- Multiple PGAS languages: UPC (C), CAF (Fortran), Titanium (Java)
  - Newer generation: Chapel, X10 and Fortress
Global Address Space Eases Programming

- The languages share the global address space abstraction
  - Shared memory is logically partitioned by thread
  - Remote memory may stay remote: no automatic caching implied
  - One-sided communication: reads/writes of shared variables
  - Both individual and bulk memory copies
- Languages differ on details
  - Some models have a separate private memory area
  - Distributed array generality and how they are constructed
State of PGAS Languages

• A successful language/library must run everywhere
• UPC
  – Commercial compilers available on Cray, SGI, HP machines
  – Open source compiler from LBNL/UCB (source-to-source)
  – Open source gcc-based compiler from Intrepid
• CAF
  – Commercial compiler available on Cray machines
  – Open source compiler available from Rice
• Titanium
  – Open source compiler from UCB runs on most machines
• Common tools
  – Open64 open source research compiler infrastructure
  – ARMCI, GASNet for distributed memory implementations
  – Pthreads, POSIX shared memory
UPC Overview and Design

• Unified Parallel C (UPC) is:
  – An explicit parallel extension of ANSI C
  – A partitioned global address space language
  – Sometimes called a GAS language

• Similar to the C language philosophy
  – Programmers are clever and careful, and may need to get close to hardware
    • to get performance, but
    • can get in trouble
  – Concise and efficient syntax

• Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C

• Based on ideas in Split-C, AC, and PCP
One-Sided vs. Two-Sided Messaging

- **Two-sided messaging**
  - Message does not contain information about final destination
  - Have to perform look up at the target or do a rendezvous
  - Point-to-point synchronization is implied with all transfers

- **One-sided messaging**
  - Message contains information about final destination
  - Decouple synchronization from data movement

- **What does the network hardware support?**

- **What about when we need point-to-point sync?**
  - Hold that thought…
GASNet Latency Performance

- GASNet implemented on top of Deep Computing Messaging Framework (DCMF)
  - Lower level than MPI
  - Provides Puts, Gets, AMSend, and Collectives
- Point-to-point ping-ack latency performance
  - N-byte transfer w/ 0 byte acknowledgement
    - GASNet takes advantage of DCMF remote completion notification
      - Minimum semantics needed to implement the UPC memory model
      - Almost a factor of two difference until 32 bytes
      - Indication of better semantic match to underlying communication system
GASNet Multilink Bandwidth

- Each node has six 850MB/s* bidirectional link
- Vary number of links from 1 to 6
- Initiate a series of nonblocking puts on the links (round-robin)
  - Communication/communication overlap
- Both MPI and GASNet asymptote to the same bandwidth
- GASNet outperforms MPI at midrange message sizes
  - Lower software overhead implies more efficient message injection
  - GASNet avoids rendezvous to leverage RDMA

* Kumar et. al showed the maximum achievable bandwidth for DCMF transfers is 748 MB/s per link so we use this as our peak bandwidth.
See “The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer”, Kumar et al. ICS08.
UPC (PGAS) Execution Model
UPC Execution Model

- A number of threads working independently in a SPMD fashion
  - Number of threads specified at compile-time or run-time; available as program variable `THREADS`
  - `MYTHREAD` specifies thread index \((0..THREADS-1)\)
  - `upc_barrier` is a global synchronization: all wait
  - There is a form of parallel loop that we will see later
- There are two compilation modes
  - Static Threads mode:
    - `THREADS` is specified at compile time by the user
    - The program may use `THREADS` as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads
Hello World in UPC

• Any legal C program is also a legal UPC program (well, almost)
• If you compile and run it as UPC with P threads, it will run P copies of the program.
• Using this fact, plus the identifiers from the previous slides, we can write a parallel hello world:

```c
#include <upc.h>    /* needed for UPC extensions */
#include <stdio.h>

main() {
    printf("Thread %d of %d: hello UPC world\n",
            MYTHREAD, THREADS);
}
```
Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} \pi r^2 = \pi/4$
- Randomly throw darts at $x,y$ positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - $\pi = 4 \times \text{ratio}$
Pi in UPC

• Independent estimates of pi:

```c
main(int argc, char **argv) {
    int i, trials, hits = 0;
    double pi;
    if (argc != 2) trials = 1000000;
    else trials = atoi(argv[1]);
    srand(MYTHREAD*17);
    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
}
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in C library

Each thread calls “hit” separately
Helper Code for Pi in UPC

• Required includes:

```
#include <stdio.h>
#include <stdlib.h>
#include <upc.h>
```

• Function to throw dart and calculate where it hits:
```
int hit()
{
    double x = ((double) rand()) / RAND_MAX;
    double y = ((double) rand()) / RAND_MAX;
    if ((x*x + y*y) <= 1.0) {
        return(1);
    } else {
        return(0);
    }
}
```
Shared vs. Private Variables
Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared non-array variables are allocated only once, with thread 0

```c
shared int ours;  // use sparingly: performance
int mine;
```

- Shared variables may not have dynamic lifetime: may not occur in a function definition, except as static.
Pi in UPC: Shared Memory Style

- Parallel computing of pi, but with a bug

```
shared int hits = 0;  // shared variable to record hits

main(int argc, char **argv) {
    int i, my_trials = 0;

    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;

    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        hits += hit();  // accumulate hits

    upc_barrier;
    if (MYTHREAD == 0) {
        printf("PI estimated to \%f.\n", 4.0*hits/trials);
    }
}
```

What is the problem with this program?
UPC Synchronization
UPC Global Synchronization

- UPC has two basic forms of barriers:
  - Barrier: block until all other threads arrive
    ```c
    upc_barrier
    ```
  - Split-phase barriers
    ```c
    upc_notify; // this thread is ready for barrier
    upc_wait;   // wait for others to be ready
    ```

- Optional labels allow for debugging
  ```c
  #define MERGE_BARRIER 12
  if (MYTHREAD%2 == 0) {
    ...
    upc_barrier MERGE_BARRIER;
  } else {
    ...
    upc_barrier MERGE_BARRIER;
  }
  ```
Synchronization - Locks

• Locks in UPC are represented by an opaque type:
  
  ```c
  upc_lock_t
  ```

• Locks must be allocated before use:
  
  ```c
  upc_lock_t *upc_all_lock_alloc(void);
  ```
  collective call - allocates 1 lock, same pointer to all threads

  ```c
  upc_lock_t *upc_global_lock_alloc(void);
  ```
  non-collective - allocates 1 lock per caller

• To use a lock:
  
  ```c
  void upc_lock(upc_lock_t *l)
  ```

  ```c
  void upc_unlock(upc_lock_t *l)
  ```

  use at start and end of critical region

• Locks can be freed when not in use
  
  ```c
  void upc_lock_free(upc_lock_t *ptr);
  ```
Pi in UPC: Shared Memory Style

- Parallel computing of pi, without the bug

```c
shared int hits = 0;
main(int argc, char **argv) {
  int i, my_trials, my_hits = 0;
  upc_lock_t *hit_lock = upc_all_lock_alloc();
  int trials = atoi(argv[1]);
  my_trials = (trials + THREADS - 1)/THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
    my_hits += hit();
  upc_lock(hit_lock);
  hits += my_hits;
  upc_unlock(hit_lock);
  upc_barrier;
  if (MYTHREAD == 0)
    printf("PI: %f", 4.0*hits/trials);
}
```
Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```c
shared int all_hits [THREADS];
main(int argc, char **argv) {
  ...
  for (i=0; i < my_trails; i++)
    all_hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++)
      hits += all_hits[i];
    printf("PI estimated to %f.", 4.0*hits/trials);
  }
}
```

all_hits is shared by all processors, just as hits was
update element with local affinity
Collectives

- UPC has support for many standard collectives (in latest language spec)
  - Data Movement: Broadcast, Scatter, Gather, Allgather, Exchange (i.e. Alltoall)
  - Computational: Reductions and Prefix Reductions
- Shared data semantics complicates when data is considered safe to read or modify
- Language lets user specify looser synchronization requirements (i.e. when is source data readable by the collective or modifiable)
  - Looser synchronization allows better implementation in runtime
  - Loose (NO): Data will not be touched within the current barrier phase
  - Medium (MY): Thread will not access remote data associated to collective without point-to-point synchronization or a barrier
  - Strict (All): Can access any and all data associated with a collective without synchronization (i.e. handled w/in the collective)
  - Defaults are to use “strict” – safety over speed
Pi in UPC: Data Parallel Style

- The previous versions of Pi works, but is not scalable:
  - On a large # of threads, the summation will be a bottleneck
- Use a reduction for better scalability

```c
shared int all_hits [THREADS], hits;
main(int argc, char **argv) {
    ... declarations an initialization code omitted
    for (i=0; i < my_trials; i++)
        all_hits[MYTHREAD] += hit();
    upc_all_reduceI(&hits, all_hits, UPC_ADD,
                    THREADS, 1, NULL,
                    UPC_IN_MYSYNC|UPC_OUT_MYSYNC);
    // upc_barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*hits/trials);
}```
Recap: Private vs. Shared Variables in UPC

- We saw several kinds of variables in the pi examples
  - Private scalars (`my_hits`)
  - Shared scalars (`hits`)
  - Shared arrays (`all_hits`)

<table>
<thead>
<tr>
<th>Global address space</th>
<th>Thread$_0$</th>
<th>Thread$_1$</th>
<th>Thread$_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_hits:</td>
<td>my_hits:</td>
<td>my_hits:</td>
<td>my_hits:</td>
</tr>
<tr>
<td>hit_lock:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all_hits[0]:</td>
<td>all_hits[1]:</td>
<td>all_hits[n]:</td>
<td></td>
</tr>
</tbody>
</table>

where: $n=\text{THREADS}-1$

Shared

Private
Work Distribution Using
upc_forall
Example: Vector Addition

- Questions about parallel vector additions:
  - How to layout data (here it is cyclic, more info later)
  - Which processor does what (here it is “owner computes”)

```c
/* vadd.c */
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD == i%THREADS)
            sum[i]=v1[i]+v2[i];
}
```
Work Distribution with upc_forall

- The idiom in the previous slide is very common
  - Loop over all; work on those owned by this thread
- UPC adds a special type of loop
  
  ```
  upc_forall(init; test; loop; affinity)
  statement;
  ```
- Programmer is asserting that the iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression indicates which iterations will run on each thread.
  It may have one of two types:
  - Integer: `(affinity % THREADS) == MYTHREAD`
  - Pointer: `upc_threadof(affinity) == MYTHREAD`
- Syntactic sugar for loop on previous slide
  - Some compilers may do better than this, e.g.,
    ```
    for(i=MYTHREAD; i<N; i+=THREADS) stmt;
    ```
  - Rather than having all threads iterate N times:
    ```
    for(i=0; i<N; i++) if (MYTHREAD == i%THREADS) stmt;
    ```
Vector Addition with upc_forall

• The vadd example can be rewritten as follows

```c
#define N 100*THREADS

shared int v1[N], v2[N], sum[N];

void main() {
    int i;
    upc_forall(i=0; i<N; i++; i)
        sum[i]=v1[i]+v2[i];
}
```

• Affinity of “&sum[i]” or “sum+i” are equivalent to “i”
• The code would still be correct (but potentially slow) if the affinity expression were “i+1” rather than “i”.

The cyclic data distribution may perform poorly on some machines.
Distributed Arrays in UPC
Shared Arrays Are Cyclic By Default

- Shared scalars (when allocated statically) always live in thread 0
- Shared arrays are spread over the threads
- Shared array elements are spread across the threads
  ```
  shared int x[THREADS]  /* 1 element per thread */
  shared int y[3][THREADS] /* 3 elements per thread */
  shared int z[3][3]    /* 2 or 3 elements per thread here*/
  ```
- In the pictures below, assume THREADS = 4
  - Red els have affinity to thread 0
    ![Diagram](y)
    As a 2D array, y is logically blocked by columns
    Think of linearized C array, then map it round-robin
    z is not, since THREADS!=3
Layouts in General

• All static non-array objects have affinity with thread zero.
• Array layouts are controlled by layout specifiers:
  – Empty or [1] (cyclic layout)
  – [*] (blocked layout)
  – [0] or [] (indefinite layout, all on 1 thread)
  – [b] (fixed block size, aka block-cyclic)
• The affinity of an array element is determined by:
  – block size, a compile-time constant
  – and THREADS.
• Element $i$ has affinity with thread
  
  $$(i / \text{block\_size}) \% \text{THREADS}$$
• In 2D and higher, linearize the elements as in a C representation, and then use above mapping
More on Shared Arrays

- Shared arrays are just data allocated on different processors
  - Can be cast into *any* block size
  - Casting just renumbers indices of shared array (data doesn’t move!)
  - Example with 4 threads
    - Allocate an array:
      ```
      shared int *A = upc_all_alloc(THREADS, sizeof(int)*4)
      ```

    - Example:
      ```
      p=(shared [4] int*) A
      q=(shared [2] int*) A
      r=(shared [1] int*) A
      ```

```plaintext

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>T1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>T2</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>T3</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>5</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>T3</td>
<td>6</td>
<td>7</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>T1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>T2</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>T3</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>
```
UPC Matrix Vector Multiplication
Code

• Matrix-vector multiplication with matrix stored by rows
• Contrived example: matrix is square & multiple of THREADS

```c
#define N 1024
shared [N*N/THREADS] int A[N][N]; /*blocked row-wise*/
shared [N/THREADS] int b[N], c[N]; /*blocked row-wise*/

void main (void) {
    int i, j, l;
    upc_forall( i = 0 ; i < N ; i++ ; &A[i][0]) {
        /*affinity means I own row i of A*/
        c[i] = 0;
        for ( l= 0 ; l< THREADS ; l++)
            c[i] += a[i][l]*b[l];
        /*no communication since all data accessed is local*/
    }
```

UPC Matrix Multiplication Code

```c
#include <upc_relaxed.h>
#define N 1024
#define P 1024
#define M 1024

/* a and c are row-wise blocked shared matrices*/
shared [N*P/THREADS] int a[N][P];
shared [M*N/THREADS] int c[N][M];
shared [M/THREADS] int b[P][M]; /*column-wise blocking*/

void main (void) {
    int i, j, l; /* private variables*/
    upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
        for (j=0 ; j<M ; j++) {
            c[i][j] = 0;
            /*access remote data for matrix multiply: */
            for (l=0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
Domain Decomposition for UPC

- Exploits locality in matrix multiplication
- \(A (N \times P)\) is decomposed row-wise into blocks of size \((N \times P) / \text{THREADS}\) as shown below:

\[
\begin{array}{cccc}
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\text{Thread 0} & \text{Thread 1} & \text{Thread THREADS-1} & \text{Thread 0} \\
\end{array}
\]

\(0 \ldots (N\times P / \text{THREADS}) - 1\)

\((N\times P / \text{THREADS}) \ldots (2N\times P / \text{THREADS}) - 1\)

\((\text{THREADS}-1)\times N\times P / \text{THREADS} \ldots \)

\((\text{THREADS}\times N\times P / \text{THREADS}) - 1\)

- \(B(P \times M)\) is decomposed column-wise into \(M/ \text{THREADS}\) blocks as shown below:

\[
\begin{array}{cccc}
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\text{Thread 0} & \text{Thread THREADS-1} & \text{Thread 0} & \text{Thread THREADS-1} \\
\end{array}
\]

Columns 0: \((M/ \text{THREADS}) - 1\)

Columns \((\text{THREADS}-1) \times M / \text{THREADS}:(M-1)\)

- Note: \(N\) and \(M\) are assumed to be multiples of \(\text{THREADS}\)
Pointers to Shared vs. Arrays

• In the C tradition, array can be accessed through pointers.
• Here is the naïve vector addition example using pointers.

```
#define N 100*THREADS
shared int v1[N], v2[N], sum[N]; /*cyclic layout*/
void main() {
    int i;
    shared int *p1, *p2;
    p1=v1; p2=v2;
    for (i=0; i<N; i++, p1++, p2++)
        if (i % THREADS == MYTHREAD)
            sum[i] = *p1 + *p2;
}
```
int *p1; /* private pointer to local memory */
shared int *p2; /* private pointer to shared space */
int *shared p3; /* shared pointer to local memory */
shared int *shared p4; /* shared pointer to shared space */

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.
Common Uses for UPC Pointer Types

```c
int *p1;
```
• These pointers are fast (just like C pointers)
• Use to access local data in part of code performing local work
• Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

```c
shared int *p2;
```
• Use to refer to remote data
• Larger and slower due to test-for-local + possible communication

```c
int *shared p3;
```
• Legal, but rarely useful. Not recommended

```c
shared int *shared p4;
```
• Use to build shared linked structures, e.g., a linked list
Bulk Data Movement and Nonblocking Communication

• Loops to perform element-wise data movement could potentially be slow because of network traffic per element
• Language introduces variants of memcpy to address this issue:
  • upc_memcpy (source and destination are in shared space)
  • upc_memput (source is in private / destination is in shared)
  • upc_memget (source is in shared / destination is in private)
• Berkeley UPC extensions also provide nonblocking variants
  – Allows comm/comp or comm/comm overlap
  – Unlike MPI_Isend and MPI_Irecv, they are completely one sided and are a better semantic fit for Remote Direct Memory Access (RDMA)
  – Expected to be part of future UPC language standard
Extensions and Tricks of the Trade
Pointer Directory

• Want each processor to dynamically allocate an array of \( k \) doubles of data on every processor that is remotely addressable.

• We want the \( k \) doubles to be contiguous so that they can be cast into local pointers and passed into C-library functions without extra copies
  
  - If \( k \) is a compile constant: shared [\( k \)] double \( A[\text{THREADS}*k] \) else

    shared [ ] double **my_dir; /*local array of UPC pointers*/
    shared double *global_array; /*cyclic by default*/

    my_dir = (shared [ ] double**)  
    malloc(sizeof(shared[ ] double*)*\text{THREADS})

    global_array = upc_all_alloc(\text{THREADS}, k*sizeof(double));

  for (i=0; i<\text{THREADS}; i++) { /*cyclic dist. implies elem i is on proc i so cast gets all memory w/ affinity to that proc*/
    my_dir[i] = (shared [ ] double*) &global_array[i];
  }

To access element \( i \) on proc \( p \) (\( i \) can range from 0 to \( k-1 \))

  my_dir [p][i] or *(my_dir [p]+i)
Berkeley UPC Extensions

• Nonblocking communication
  – Ability to have comm/comp or comm/comm overlap
  – Like MPI_Isend and Irecv, uses explicit handles that need to be synched.

• Semaphores and Point-to-Point synchronization
  – Many applications need point-to-point synchronization
  – Provide mechanisms to allow it in UPC without making it default
  – Interface provides a one-sided signaling put which notifies remote processor when data has arrived

• Value-based collectives
  – Simplify collective interface when you need collectives on scalar values: `hits = bupc_allv_reduce(int, my_hits, 0, UPC_ADD);`

• Remote atomics
  – Perform atomic operations on 32 or 64 bit ints in shared space
Point-to-Point Sync

• 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 using the memory model
• Our Proposed Extension
  – Use semaphores in shared space and provide “signalling put”
  – User specifies remote semaphore to signal on completion of put
  – Point-to-point synchronization is provided only when needed
Point-to-Point Synchronization (cont):

- Simple extension to upc_memput interface
  
  ```c
  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
  - Also provide a non-blocking variant

- Target side calls wait on the same semaphore
  - When the semaphore gets tripped the data has arrived and the target can safely use the buffer
  - Interface: `bupc_sem_wait(bupc_sem_t *s)`

```c
bupc_memput_signal(...,sem);  bupc_memput_signal(...,sem);  bupc_memput_signal(...,sem);
/* overlap compute */          /* consume data */
```

---

Thread 1

```c
bupc_memput_signal(...,sem);
/* overlap compute */
```

Thread 0

```c
bupc_memput_signal(...,sem);
/* consume data */
```

**memput_signal:**
- latency ~0.5 round-trips
- allows overlap
- easy to use
Application Examples and Performance
Dense LU Factorization in UPC

• Direct methods have complicated dependencies
  • Especially with pivoting (unpredictable communication)
  • Especially for sparse matrices (dependence graph with holes)

• LU Factorization in UPC
  • Use overlap ideas and multithreading to mask latency
  • Multithreaded: UPC threads + user threads + threaded BLAS
    • Panel factorization: Including pivoting
    • Update to a block of U
    • Trailing submatrix updates
  • Written in a Data-centric way
    • Shared address space and one-sided communication allows remote enqueue of work w/o interrupting the remote processors

• Dense LU done: HPL-compliant
• Sparse version underway

• Ref: “Multi-Threading and One-Sided Communication in Parallel LU Factorization” by Parry Husbands and Kathy Yelick [SC’07]
• Comparison to ScaLAPACK on an Altix, a 2 x 4 process grid
  – ScaLAPACK (block size 64) 25.25 GFlop/s (tried several block sizes)
  – UPC LU (block size 256) - 33.60 GFlop/s, (block size 64) - 26.47 GFlop/s
• n = 32000 on a 4x4 process grid
  – ScaLAPACK - 43.34 GFlop/s (block size = 64)
  – UPC - 70.26 Gflop/s (block size = 200)

• MPI HPL numbers from HPCC database
• Large scaling:
  • 2.2 TFlops on 512p,
  • 4.4 TFlops on 1024p (Thunder)
Other Dense Linear Algebra Performance on BG/P

Parallel Matrix Multiplication (256 core BlueGene/P)

- PBLAS (MPI): 458 GFlops
- UPC hand-roll: 580 GFlops
- UPC collective: 625 GFlops

Parallel Cholesky Factorization (256 core BlueGene/P)

- ScaLapack (MPI): 202 GFLOps
- UPC hand-roll: 212 GFLOps
- UPC collective: 220 GFLOps
Case Study: NAS FT Benchmark

• Perform a large 3D FFT
  – Molecular dynamics, CFD, image processing, signal processing, astrophysics, etc.
  – Representative of a class of communication intensive algorithms
    • Requires parallel many-to-many communication
    • Stresses communication subsystem
    • Limited by bandwidth (namely bisection bandwidth) of the network
• Building on our previous work, we perform a 2D partition of the domain
  – Requires two rounds of communication rather than one
  – Each processor communicates in two rounds with $O(\sqrt{T})$ threads in each
Strong Scaling

- Fix problem size at 2k x 1k x 1k and run in VN mode
  - upto 4 racks of BG/P with 4 processes per node
- Analytic upper bound calculates megaflop rate based on time needed to transfer domain across the bisection
  - Kink at 2048 cores indicates where 3D Torus is completed
- MPI Packed Slabs scales better than MPI Slabs
  - Benefit of comm/comp. overlap outweighed by extra messages
- UPC (i.e. GASNet) Slabs consistently outperforms MPI
  - Lower software overhead enables better overlap
  - Outperforms Slabs by mean of 63% and Packed Slabs by mean of 37%
Weak Scaling

- Scale problem size with the number of cores
  - Computation for FFT scales as \(O(N \log N)\) so thus flops don’t scale linearly
- UPC Slabs scales better than strong scaling benchmark
  - Message size gets too small at high concurrency for strong scaling and becomes hard to utilize overlap
- MPI Packed Slabs outperforms MPI Slabs (most of the time)
  - Again indicates that overlapping communication/computation is not a fruitful optimization for MPI
- UPC achieves 1.93 Teraflops while best MPI achieves 1.37 Teraflops
  - 40% improvement in performance at 16k cores.
Latest FFT Performance on BG/P (strong scaling)

- Slabs
- Slabs (Collective)
- Packed Slabs (Collective)
- MPI Packed Slabs

<table>
<thead>
<tr>
<th>Slabs</th>
<th>Slabs (Collective)</th>
<th>Packed Slabs (Collective)</th>
<th>MPI Packed Slabs</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>512</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4096</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8192</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16384</td>
</tr>
</tbody>
</table>

GFlops
Latest FFT Performance on BG/P (weak scaling)

HPC Challenge Peak as of July 09 is ~4.5 TFlops on 128k Cores

GFlops

<table>
<thead>
<tr>
<th>D/8</th>
<th>D/4</th>
<th>D/2</th>
<th>D</th>
<th>D*2</th>
<th>D*4</th>
<th>D*8</th>
<th>D*16</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>16384</td>
<td>32768</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
</tr>
</tbody>
</table>
Thanks!
Any Questions?
Backup Slides
The cyclic layout is typically stored in one of two ways
- Distributed memory: each processor has a chunk of memory
  - Thread 0 would have: 0, THREADS, THREADS*2,... in a chunk
- Shared memory machine: each thread has a logical chunk
  - Shared memory would have: 0, 1, 2,... THREADS, THREADS+1,...

What performance problem is there with the latter?

What if this code was instead doing nearest neighbor averaging (1D stencil)?

Vector addition example can be rewritten as follows

```c
#define N 100*THREADS

shared int v1[N], v2[N], sum[N];

void main() {
    int i;
    upc_forall(i=0; i<N; i++;
        sum[i]=v1[i]+v2[i];
```
UPC Collectives in General

• The UPC collectives interface is available from:
  – http://www.gwu.edu/~upc/documentation.html

• It contains typical functions:
  – Data movement: broadcast, scatter, gather, …
  – Computational: reduce, prefix, …

• Interface has synchronization modes:
  – Avoid over-synchronizing (barrier before/after is simplest semantics, but may be unnecessary)
  – Data being collected may be read/written by any thread simultaneously
2D Array Layouts in UPC

• Array a1 has a row layout and array a2 has a block row layout.
  
  ```
  shared [m] int a1 [n][m];
  shared [k*m] int a2 [n][m];
  ```

• If \((k + m) \mod \text{THREADS} = 0\) then a3 has a row layout
  
  ```
  shared int a3 [n][m+k];
  ```

• To get more general HPF and ScaLAPACK style 2D blocked layouts, one needs to add dimensions.

• Assume \(r \times c = \text{THREADS}\);
  
  ```
  shared [b1][b2] int a5 [m][n][r][c][b1][b2];
  ```
  
  • or equivalently
  
  ```
  shared [b1*b2] int a5 [m][n][r][c][b1][b2];
  ```
Notes on the Matrix Multiplication Example

• The UPC code for the matrix multiplication is almost the same size as the sequential code
• Shared variable declarations include the keyword shared
• Making a private copy of matrix B in each thread might result in better performance since many remote memory operations can be avoided
• Can be done with the help of upc_memget
## UPC Pointers

<table>
<thead>
<tr>
<th>Where does the pointer point?</th>
<th>Local</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>PP (p_1)</td>
<td>PS (p_3)</td>
</tr>
<tr>
<td>Shared</td>
<td>SP (p_2)</td>
<td>SS (p_4)</td>
</tr>
</tbody>
</table>

Where does the pointer reside?

- **Private**: PP \(p_1\) (private pointer to local memory), PS \(p_3\) (private pointer to shared space)
- **Shared**: SP \(p_2\) (shared pointer to local memory), SS \(p_4\) (shared pointer to shared space)

```c
int *p1;    // private pointer to local memory
shared int *p2; // private pointer to shared space
int *shared p3; // shared pointer to local memory
shared int *shared p4; // shared pointer to shared space
```

Shared to private is not recommended.
(FT) IPDPS ‘06 Talk
Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap

Christian Bell$^{1,2}$, Dan Bonachea$^1$, Rajesh Nishtala$^1$, and Katherine Yelick$^{1,2}$

$^1$UC Berkeley, Computer Science Division
$^2$Lawrence Berkeley National Laboratory
Conventional Wisdom

• Send few, large messages
  – Allows the network to deliver the most effective bandwidth

• Isolate computation and communication phases
  – Uses bulk-synchronous programming
  – Allows for packing to maximize message size

• Message passing is preferred paradigm for clusters

• Global Address Space (GAS) Languages are primarily useful for latency sensitive applications

• GAS Languages mainly help productivity
  – However, not well known for their performance advantages
Our Contributions

• Increasingly, cost of HPC machines is in the network

• One-sided communication model is a better match to modern networks
  – GAS Languages simplify programming for this model

• How to use these communication advantages
  – Case study with NAS Fourier Transform (FT)
  – Algorithms designed to relieve communication bottlenecks
    • Overlap communication and computation
    • Send messages early and often to maximize overlap
**UPC Programming Model**

- **Global address space**: any thread/process may directly read/write data allocated by another
- **Partitioned**: data is designated as local (near) or global (possibly far); programmer controls layout

---

Global arrays: Allows any processor to directly access data on any other processor

---

- **3 of the current languages**: UPC, CAF, and Titanium
  - Emphasis in this talk on UPC (based on C)
  - However programming paradigms presented in this work are not limited to UPC
Advantages of GAS Languages

• Productivity
  – GAS supports construction of complex shared data structures
  – High level constructs simplify parallel programming
  – Related work has already focused on these advantages

• Performance (the main focus of this talk)
  – GAS Languages can be faster than two-sided MPI
  – One-sided communication paradigm for GAS languages more natural fit to modern cluster networks
  – Enables novel algorithms to leverage the power of these networks
  – GASNet, the communication system in the Berkeley UPC Project, is designed to take advantage of this communication paradigm
One-Sided vs Two-Sided

- A one-sided put/get can be entirely handled by network interface with RDMA support
  - CPU can dedicate more time to computation rather than handling communication

- A two-sided message can employ RDMA for only part of the communication
  - Each message requires the target to provide the destination address
  - Offloaded to network interface in networks like Quadrics

- RDMA makes it apparent that MPI has added costs associated with ordering to make it usable as an end-user programming model
Latency Advantages

• Comparison:
  – One-sided:
    • Initiator can always transmit remote address
    • Close semantic match to high bandwidth, zero-copy RDMA
  – Two-sided:
    • Receiver must provide destination address

• Latency measurement correlates to software overhead
  – Much of the small-message latency is due to time spent in software/firmware processing

One-sided implementation consistently outperforms 2-sided counterpart
Bandwidth Advantages

• One-sided semantics better match to RDMA supported networks
  – Relaxing point-to-point ordering constraint can allow for higher performance on some networks
  – GASNet saturates to hardware peak at lower message sizes
  – Synchronization decoupled from data transfer
• MPI semantics designed for end user
  – Comparison against good MPI implementation
  – Semantic requirements hinder MPI performance
  – Synchronization and data transferred coupled together in message passing

Over a factor of 2 improvement for 1kB messages
Bandwidth Advantages (cont)

- GASNet and MPI saturate to roughly the same bandwidth for “large” messages.
- GASNet consistently outperforms MPI for “mid-range” message sizes.
A Case Study: NAS FT

• How to use the potential that the microbenchmarks reveal?

• Perform a large 3 dimensional Fourier Transform
  – Used in many areas of computational sciences
    • Molecular dynamics, computational fluid dynamics, image processing,
      signal processing, nanoscience, astrophysics, etc.

• Representative of a class of communication intensive algorithms
  – Sorting algorithms rely on a similar intensive communication pattern
  – Requires every processor to communicate with every other processor
  – Limited by bandwidth
Performing a 3D FFT (part 2)

- Perform an FFT in all three dimensions
- With 1D layout, 2 out of the 3 dimensions are local while the last Z dimension is distributed

Step 1: FFTs on the columns
(all elements local)

Step 2: FFTs on the rows
(all elements local)

Step 3: FFTs in the Z-dimension
(requires communication)
Performing the 3D FFT (part 3)

• Can perform Steps 1 and 2 since all the data is available without communication
• Perform a Global Transpose of the cube
  – Allows step 3 to continue
The Transpose

• Each processor has to scatter input domain to other processors
  – Every processor divides its portion of the domain into P pieces
  – Send each of the P pieces to a different processor

• Three different ways to break it up the messages
  1. Packed Slabs (i.e. single packed “Alltoall” in MPI parlance)
  2. Slabs
  3. Pencils

• An order of magnitude increase in the number of messages
• An order of magnitude decrease in the size of each message
• “Slabs” and “Pencils” allow overlapping communication and computation and leverage RDMA support in modern networks
Algorithm 1: Packed Slabs

Example with P=4, NX=NY=NZ=16

1. Perform all row and column FFTs
2. Perform local transpose
   - data destined to a remote processor are grouped together
3. Perform P puts of the data

- For $512^3$ grid across 64 processors
  - Send 64 messages of 512kB each
Bandwidth Utilization

• NAS FT (Class D) with 256 processors on Opteron/InfiniBand
  – Each processor sends 256 messages of 512kBytes
  – Global Transpose (i.e. all to all exchange) only achieves
    67% of peak point-to-point bidirectional bandwidth
  – Many factors could cause this slowdown
    • Network contention
    • Number of processors that each processor communicates with

• Can we do better?
Algorithm 2: Slabs

- Waiting to send all data in one phase bunches up communication events

- Algorithm Sketch
  - for each of the NZ/P planes
    - Perform all column FFTs
    - for each of the P “slabs” (a slab is NX/P rows)
      - Perform FFTs on the rows in the slab
      - Initiate 1-sided put of the slab
  - Wait for all puts to finish
  - Barrier

- Non-blocking RDMA puts allow data movement to be overlapped with computation.

- Puts are spaced apart by the amount of time to perform FFTs on NX/P rows

- For $512^3$ grid across 64 processors
  - Send 512 messages of 64kB each
Algorithm 3: Pencils

- Further reduce the granularity of communication
  - Send a row (*pencil*) as soon as it is ready

- Algorithm Sketch
  - For each of the NZ/P planes
    - Perform all 16 column FFTs
    - For $r=0; r<NX/P; r++$
      - For each slab $s$ in the plane
        - Perform FFT on row $r$ of slab $s$
        - Initiate 1-sided put of row $r$
  - Wait for all puts to finish
  - Barrier

- Large increase in message count
- Communication events finely diffused through computation
  - Maximum amount of overlap
  - Communication starts early

- For $512^3$ grid across 64 processors
  - Send 4096 messages of 8kB each
Communication Requirements

- $512^3$ across 64 processors
  - Alg 1: Packed Slabs
    - Send 64 messages of 512kB
  - Alg 2: Slabs
    - Send 512 messages of 64kB
  - Alg 3: Pencils
    - Send 4096 messages of 8kB

GASNet achieves close to peak bandwidth with Pencils but MPI is about 50% less efficient at 8k.

With Slabs GASNet is slightly faster than MPI.

With the message sizes in Packed Slabs both comm systems reach the same peak bandwidth.
## Platforms

<table>
<thead>
<tr>
<th>Name</th>
<th>Processor</th>
<th>Network</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opteron/Infiniband “Jacquard” @ NERSC</td>
<td>Dual 2.2 GHz Opteron (320 nodes @ 4GB/node)</td>
<td>Mellanox Cougar InfiniBand 4x HCA</td>
<td>Linux 2.6.5, Mellanox VAPI, MVAPICH 0.9.5, Pathscale CC/F77 2.0</td>
</tr>
<tr>
<td>Alpha/Elan3 “Lemieux” @ PSC</td>
<td>Quad 1 GHz Alpha 21264 (750 nodes @ 4GB/node)</td>
<td>Quadrics QsNet1 Elan3 /w dual rail (one rail used)</td>
<td>Tru64 v5.1, Elan3 libelan 1.4.20, Compaq C V6.5-303, HP Fortra Compiler X5.5A-4085-48E1K</td>
</tr>
<tr>
<td>Itanium2/Elan4 “Thunder” @ LLNL</td>
<td>Quad 1.4 Ghz Itanium2 (1024 nodes @ 8GB/node)</td>
<td>Quadrics QsNet2 Elan4</td>
<td>Linux 2.4.21-chaos, Elan4 libelan 1.8.14, Intel ifort 8.1.025, icc 8.1.029</td>
</tr>
<tr>
<td>P4/Myrinet “FSN” @ UC Berkeley Millennium Cluster</td>
<td>Dual 3.0 Ghz Pentium 4 Xeon (64 nodes @ 3GB/node)</td>
<td>Myricom Myrinet 2000 M3S-PCI64B</td>
<td>Linux 2.6.13, GM 2.0.19, Intel ifort 8.1-20050207Z, icc 8.1-20050207Z</td>
</tr>
</tbody>
</table>
Comparison of Algorithms

• Compare 3 algorithms against original NAS FT
  – All versions including Fortran use FFTW for local 1D FFTs
  – Largest class that fit in the memory (usually class D)

• All UPC flavors outperform original Fortran/MPI implantation by at least 20%
  – One-sided semantics allow even exchange based implementations to improve over MPI implementations
  – Overlap algorithms spread the messages out, easing the bottlenecks
  – ~1.9x speedup in the best case
Time Spent in Communication

- Implemented the 3 algorithms in MPI using Irecvs and Isends

- Compare time spent initiating or waiting for communication to finish
  - UPC consistently spends less time in communication than its MPI counterpart
  - MPI unable to handle pencils algorithm in some cases
Conclusions

• One-sided semantics used in GAS languages, such as UPC, provide a more natural fit to modern networks
  – Benchmarks demonstrate these advantages

• Use these advantages to alleviate communication bottlenecks in bandwidth limited applications
  – Paradoxically it helps to send more, smaller messages

• Both two-sided and one-sided implementations can see advantages of overlap
  – One-sided implementations consistently outperform two-sided counterparts because comm model more natural fit

• *Send early and often* to avoid communication bottlenecks
Try It!

- Berkeley UPC is open source
  - Download it from http://upc.lbl.gov
Contact Us

• Authors
  – Christian Bell
  – Dan Bonachea
  – Rajesh Nishtala
  – Katherine A. Yelick
  – Email us:
    • upc@lbl.gov

• Associated Paper: IPDPS ‘06 Proceedings
• Berkeley UPC Website: http://upc.lbl.gov
• GASNet Website: http://gasnet.cs.berkeley.edu

Special thanks to the fellow members of the Berkeley UPC Group
• Wei Chen
• Jason Duell
• Paul Hargrove
• Parry Husbands
• Costin Iancu
• Mike Welcome
P2P Sync (PGAS’06)
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

- 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

```
shared [] int data[...];
int f = 0;
upc_lock_t *L = …;
upc_lock(&L);
upc_memput(&data,…);
f = 1;
upc_unlock(&L);
while (1) {
    upc_lock(&L);
    if (f) break;
    upc_unlock(&L);
}
/* consume data */
```

Thread 0

```
shared [] int data[...];
int f = 0;
upc_lock_t *L = …;
upc_lock(&L);
upc_memput(&data,…);
f = 1;
upc_unlock(&L);
while (1) {
    upc_lock(&L);
    if (f) break;
    upc_unlock(&L);
}
/* consume data */
```

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

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

Berkeley UPC: http://upc.lbl.gov
Titanium: http://titanium.cs.berkeley.edu
Point-to-Point Sync Data Xfer in UPC

Thread 1

strict int f = 0;
upc_memput(&data,…);
f = 1;
while (!f) bupc_poll(); /* consume data */

Thread 0

strict int f = 0;
memput + strict flag: latency ~1.5 round-trips
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

Berkeley UPC: http://upc.lbl.gov
Titanium: http://titanium.cs.berkeley.edu
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_memput_signal(...,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
  ```c
  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
  ```c
  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
  - `void bupc_sem_free(bupc_sem_t *s);` creates a `sem_t` object with affinity to caller
  - Flags specify a few different usage flavors
    - eg one or many producer/consumer threads, integral/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

- 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

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

- memput (roundtrip) + strict put: Latency is $\approx 1\frac{1}{2}$ RDMA put roundtrips
- bupc_sem_t: Latency is $\approx 1\frac{1}{2}$ RDMA put roundtrip
  - sem_t and MPI both using a single RDMA put, at least up to 1KB

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

RDMA put latency:
$\approx 10.5$us round-trip
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
75% improvement in synchronous communication time
28% improvement in total runtime (relative to barrier)
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