UPC and UPC++: Partitioned Global Address Space Languages

Kathy Yelick
Associate Laboratory Director for Computing Sciences
Lawrence Berkeley National Laboratory
Professor of EECS, UC Berkeley

Amir Kamil
Computer Systems Engineer, LBNL
Lecturer of EECS, University of Michigan
Parallel Programming Problem: Histogram

• Consider the problem of computing a histogram:
  – Large number of “words” streaming in from somewhere
  – You want to count the # of words with a given property

• In shared memory
  – Lock each bucket

  A’s  B’s  C’s  …  Y’s  Z’s

• Distributed memory: the array is huge and spread out
  – Each processor has a substream and sends +1 to the appropriate processor… and that processor “receives”

  A’s  B’s  C’s  D’s  …  Y’s  Z’s
Goals of PGAS Programming

• Applications: convenient programming of irregular codes
  – Graphs
  – Hash tables
  – Sparse matrices
  – Adaptive (hierarchical) meshes

• Machines: expose best available performance on a given machine
  – Low latency for small messages
  – High bandwidth even for medium sized messages
  – High injection bandwidth
PGAS = Partitioned Global Address Space

- **Global address space**: thread may directly read/write remote data
  - Convenience of shared memory
- **Partitioned**: data is designated as local or global
  - Locality and scalability of message passing
Hello World in UPC

• Any legal C program is also a legal UPC program
• If you compile and run it as UPC with P threads, it will run P copies of the program.
• Using this fact, plus a few UPC keywords:

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

main() {
    printf("Thread %d of %d: hello UPC world\n", MYTHREAD, THREADS);
}
```
PGAS means directly accessing remote data

- SPMD: fixed number of threads (e.g., one per core)
- Distributed arrays are built-in

```c
shared int a[100]; // shared array
a[10] = 3;         // put, possibly remote
int x = a[14];     // get, possibly remote
```

- Global pointer are like C pointers:

```c
shared int *p = &a[4]; // can also upc_alloc
*p = 3;               // put
x = *p;               // get
p++;                  // move to next element
```

- UPC has locks and barriers for synchronization and collective communication (broadcast, reduce, etc.)
• Directly read/write remote memory; partitioned for locality
• One-sided communication underneath:

Put:  \( a[i] = \ldots \);  \( *p = \ldots \);  \( \text{upc\_mem\_get(\ldots)} \)
Get:  \( \ldots = a[i] \ldots \);  \( \ldots = *p \);  \( \text{upc\_mem\_put(\ldots)} \)
UPC Non-blocking Bulk Operations

Important for performance:
• Communication overlap with computation
• Communication overlap with communication (pipelining)
• Low overhead communication

#include<upc_nb.h>

upc_handle_t h =
upc_memcpy_nb(shared void * restrict dst,
               shared const void * restrict src,
               size_t n);
void upc_sync(upc_handle_t h);  // blocking wait
int upc_sync_attempt(upc_handle_t h);  // non-blocking
One-Sided Communication in PGAS (e.g., GASnet inside)

- A two-sided message needs to be matched with a receive
  - Ordering requirements on messages can also hinder bandwidth
- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Decouples transfer from synchronization
  - Avoids interrupting the CPU or storing data from CPU (preposts)
Latency on a Cray Aries (NERSC Cori-P1)

Latency on Cori (Haswell with Aries)

- **MPI**
- **GASnet**

Message Size:
- 1B
- 4B
- 16B
- 64B
- 256B
- 1KB
- 4KB
- 16KB
- 64KB
- 256KB
- 1MB
- 4MB

Latency in μs:
- 1
- 10
- 100
- 1000

(Raw Text: Latency on a Cray Aries (NERSC Cori-P1))
Bandwidth on a Cray Aries (NERSC Cori-P1)

Bandwidth on Cori (Haswell with Aries)

- **Message Size**
- **GB/s**

- **GASnet**
- **MPI**

- 16B, 64B, 256B, 1KB, 4KB, 16KB, 64KB, 256KB, 1MB, 4MB
Medium sized “flood” bandwidth across machine

Percent of peak for 4 KB messages in flood bandwidth

<table>
<thead>
<tr>
<th>Machine</th>
<th>GASNet</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elan3</td>
<td>177</td>
<td>231</td>
</tr>
<tr>
<td>Elan4</td>
<td>670</td>
<td>679</td>
</tr>
<tr>
<td>Myrinet</td>
<td>152</td>
<td>223</td>
</tr>
<tr>
<td>G5/IB</td>
<td>252</td>
<td>714</td>
</tr>
<tr>
<td>Opteron/IB</td>
<td>402</td>
<td>580</td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gemini</td>
<td>43%</td>
<td>31%</td>
</tr>
<tr>
<td>Aries</td>
<td>33%</td>
<td>33%</td>
</tr>
<tr>
<td>BG/Q</td>
<td>94%</td>
<td>47%</td>
</tr>
</tbody>
</table>
Application Challenge: Fast All-to-All

Transpose in 3D FFT

• Three approaches:
  • **Chunk:**
    • Wait for 2nd dim FFTs to finish
    • Minimize # messages
  • **Slab:**
    • Wait for chunk of rows destined for 1 proc to finish
    • Overlap with computation
  • **Pencil:**
    • Send each row as it completes
    • Maximize overlap and
    • Match natural layout

Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea
• Avoid congestion at node interface: allow all cores to communicate
• Avoid congestion inside global network: spread communication over longer time period (send early and often)
FFT Performance on BlueGene/P (Mira)

- UPC implementation outperforms MPI
- Both use highly optimized FFT library on each node
- UPC version avoids send/receive synchronization
  - Lower overhead
  - Better overlap
  - Better bisection bandwidth
De novo Genome Assembly

• DNA sequence consists of 4 bases: A/C/G/T
• Read: short fragment of DNA
• De novo assembly: Construct a genome (chromosomes) from a collection of reads
• Sequencers produce fragments called “reads”
• Chop them into overlap fixed-length fragments, “K-mers”
• Parallel DFS (from randomly selected K-mers) → “contigs”

- Hash tables used here (and in other assembly phases)
  - Different use cases, different implementations
- Some tricky synchronization to deal with conflicts
Partitioned Global Address Space Programming

- Store the connections between read fragments (K-mers) in a hash table
- Allows for TB-PB size data sets

DEGAS
Distributed Hash Tables in PGAS

- Remote Atomics, Dynamic Aggregation, Software Caching
- 13x Faster than MPI code (Ray) on 960 cores

Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Lenny Oliker, Dan Rokhsar, and Kathy Yelick. HipMer: An Extreme-Scale De Novo Genome Assembler, SC’15
Comparison to other Assemblers

![Runtime on Assemblers](chart)

- Meraculous: 140 hours
- SGA: 4 minutes
- ABySS 960 Contig only
- Ray 960
- HipMer 960
- HipMer 20K

Equal core counts (960 Edison)
Science Impact: HipMer is transformative

- Human genome (3Gbp) “de novo” assembled:
  - Meraculous: 48 hours
  - HipMer: 4 minutes (720x speedup relative to Meraculous)

- Wheat genome (17 Gbp) “de novo” assembled (2014):
  - Meraculous (did not run):
  - HipMer: 39 minutes; 15K cores (first all-in-one assembly)

- Pine genome (20 Gbp) “de novo” assembled (2014):
  - Masurca: 3 months; 1 TB RAM

- Wetland metagenome (1.25 Tbp) analysis (2015):
  - Meraculous (projected): 15 TB of memory
  - HipMER: Strong scaling to over 100K cores (contig gen only)

Makes unsolvable problems solvable!

Georganas, Buluc, Chapman, Oliker, Rokhsar, Yelick, [Aluru, Egan, Hofmeyr] in SC14, IPDPS15, SC15
**UPC++: PGAS with “Mixins”**

- **UPC++ uses templates** (no compiler needed)
  
  ```
  shared_var<int> s;
  global_ptr<LLNode> g;
  shared_array<int> sa(8);
  ```

- **Default execution model is SPMD**, but

- **Remote procedure calls, async**
  
  ```
  async(place) (Function f, T1 arg1,...);
  wait(); // other side does poll();
  ```

- **Teams for hierarchical algorithms and machines**
  
  ```
  teamsplit (team) { ... }
  ```

- **Interoperability is key; UPC++ can be use with OpenMP or MPI**
UPC++ is a library, not a compiled language, yet performance is comparable.

The difference between UPC++ and UPC is about 0.2 μs (~220 cycles).
Application Challenge: Data Fusion in UPC++

- Seismic modeling for energy applications “fuses” observational data into simulation
- With UPC++ “matrix assembly” can solve larger problems


First ever sharp, three-dimensional scan of Earth’s interior that conclusively connects plumes of hot rock rising through the mantle with surface hotspots that generate volcanic island chains like Hawaii, Samoa and Iceland.
Application Challenge: Data Fusion in UPC++

Distributed Matrix Assembly
- Remote asyncs with user-controlled resource management
- Remote memory allocation
- Team idea to divide threads into injectors / updaters
- 6x faster than MPI 3.0 on 1K nodes
→ Improving UPC++ team support

See French et al, IPDPS 2015 for parallelization overview.
Load Balancing and Irregular Matrix Transpose

- Hartree Fock example (e.g., in NWChem)
  - Inherent load imbalance
- UPC++
  - Work stealing and fast atomics
  - Distributed array: easy and fast transpose
- Impact
  - 20% faster than the best existing solution (GTFock with Global Arrays)

Increase scalability!

<table>
<thead>
<tr>
<th>Distributed Array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Local Array</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Hartree Fock Code in UPC++

Strong Scaling of UPC++ HF Compared to GTFock with Global Arrays on NERSC Edison (Cray XC30)

UPC++ Communication Speeds up AMR

- Adaptive Mesh Refinement on Block-Structured Meshes
  - Used in ice sheet modeling, climate, subsurface (fracking),

Hierarchical UPC++ (distributed / shared style)
- UPC++ plus UPC++ is 2x faster than MPI plus OpenMP
- MPI + MPI also does well
Beyond Put/Get: Event-Driven Execution

- DAG Scheduling in a distributed (partitioned) memory context
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - “memory constrained” lookahead

Uses a Berkeley extension to UPC to remotely synchronize

some edges omitted
Sparse Cholesky using fan-both algorithm in UPC++

- Uses asynchronous tasks with dependencies

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick
symPACK: Sparse Cholesky

Run times on boneS10 for three variants of symPACK

Figure 7: Impact of communication strategy and scheduling on symPACK performance

• Scalability of symPACK on Cray XC30 (Edison)
  – Comparable or better than best solvers (evaluation in progress)
  – Notoriously hard parallelism problem

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick
Common Pattern for Distributed Data Structures

- Many UPC programs avoid the UPC style arrays in factor of directories of objects

```c
typedef shared [] double *sdblptr;
shared sdblptr directory[THREADS];
directory[i]=upc_alloc(local_size*sizeof(double));
```

- These are also more general:
  - Multidimensional, unevenly distributed
  - Ghost regions around blocks
Summary: PGAS for Irregular Applications

• Lower overhead of communication makes PGAS useful for latency-sensitive problems or bisection bandwidth problems

• Specific application characteristics that benefit:
  – Fine-grained updates (Genomics HashTable construction)
  – Latency-sensitive algorithms (Genomics DFS)
  – Distributed task graph (Cholesky)
  – Work stealing (Hartree Fock)
  – Irregular matrix assembly / transpose (Seismic, HF)
  – Medium-grained messages (AMR)
  – All-to-all communication (FFT)

• There are also benefits of thinking algorithmically in this model: parallelize things that are otherwise hard to imagine
Summary: PGAS for Modern HPC Systems

• The lower overhead of communication is also important given current machine trends
  – Many lightweight cores per node (do not want a hefty serial communication software stack to run on them)
  – RDMA mechanisms between nodes (decouple synchronization from data transfer)
  – GAS on chip: direct load/store on chip without full cache coherence across chip
  – Hierarchical machines: fits both shared and distributed memory, but supports hierarchical algorithms
  – New models of memory: High Bandwidth Memory on chip or NVRAM above disk
Installing Berkeley UPC++, UPC, and GASNet

Available on Mac OS X, Linux, Infiniband clusters, Ethernet clusters, and most HPC systems

- UPC++ Open source with BSD license
  https://bitbucket.org/upcxx
- UPC++ installation
  https://bitbucket.org/upcxx/upcxx/wiki/Installing%20UPC++
- GASNet communication
  https://gasnet.lbl.gov
- Examples
  - DAXPY, Conjugate Gradient, FFT, GUPS, MatrixMultiply, Multigrid, Minimum Degree Ordering, Sample Sort, Sparse Matrix-Vector multiply
Using Berkeley UPC at NERSC or ALCF

Load the bupc module via

    module load bupc

Compile code with the upcc

    upcc -V  // shows version

Add the following line to your ~/.soft file:

    PATH += /home/projects/pgas/berkeley_upc-2.22.3/V1R2M2/gcc-narrow/bin/

OR, if using the xl compilers, add:

    PATH += /home/projects/pgas/berkeley_upc-2.22.3/V1R2M2/xlc-narrow/bin/

Run

    resoft

Compile with upcc. To see the version and configuration, run

    upcc -V
**UPC++ V1.0**

A C++ Library for Lightweight PGAS Programming

Led by Scott B. Baden and Paul Hargrove (LBNL)
Presented by Amir Kamil (LBNL/University of Michigan)
UPC++ V1.0 Overview

• A complete redesign of UPC++ that leverages GASNet-EX to deliver better performance and scalability
• A “compiler-free” approach for PGAS
  – Leverage C++ standards and compilers
  – Influence future directions of the C++ standard
• Interoperates with existing programming systems
  – 1-to-1 mapping between MPI rank and UPC++ rank
  – OpenMP and CUDA can be easily mixed with UPC++ in the same way as MPI+X

• Design philosophy:
  All communication is explicit
  Most operations are non-blocking to encourage asynchronous programming
  No non-scalable data structures
Hello World in UPC++

- If you compile and run a UPC++ program with P ranks, it will run P copies of the program
- However, need to initialize UPC++ before calling any UPC++ functions:

```cpp
#include <upcxx/upcxx.hpp>  // UPC++ header
#include <iostream>

int main(int argc, char **argv) {
    upcxx::init();  // Start UPC++ state
    std::cout << "Hello world from rank "
                    << upcxx::rank_me()  // Who am I?
                    << std::endl;
    upcxx::finalize();  // Close down UPC++ state
}
```
The API

- Foundational types
  - Global Pointers
  - Futures (and promises)
  - Distributed Objects

- Communication
  - 1-sided Communication
    - rput/rget (bulk and single element), non-contiguous transfers, memory kinds
  - RPC (remote procedure call)

- Callbacks
- Remote Atomics
- Teams (mechanism for grouping ranks together)
- Progress and the Memory Model
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 = $\frac{1}{4} \times \pi r^2 = \pi/4$
- 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:

```cpp
int main(int argc, char **argv) {
    upcxx::init();
    int hits, trials = 0;
    double pi;
    if (argc != 2) trials = 1000000;
    else trials = atoi(argv[1]);
    srand(upcxx::rank_me() * 17);
    for (int i = 0; i < trials; i++) hits += hit();
    pi = 4.0 * hits / trials;
    cout << "PI estimated to " << pi << endl;
    upcxx::finalize();
}
```
Helper Code for Pi in UPC++

• Required includes:
  ```
  #include <iostream>
  #include <cstdlib>
  #include <upcxx/upcxx.hpp>
  ```

• 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;
      }
  }
  ```
• Required includes and variables:
  ```
  #include <iostream>
  #include <random>
  #include <upcxx/upcxx.hpp>
  default_random_engine generator;
  uniform_real_distribution<> dist(0.0, 1.0);
  ```

• Function to throw dart and calculate where it hits:
  ```
  int hit() {
    double x = dist(generator);
    double y = dist(generator);
    if (x*x + y*y <= 1.0) {
      return 1;
    } else {
      return 0;
    }
  }
  ```

UPC++ allows full use of the C++ Standard Template Library
Private vs. Shared Memory in UPC++

• Normal C++ variables and objects are allocated in the private memory space for each thread.

• Memory from the shared space is allocated explicitly.

```cpp
global_ptr<int> gpstr = new_<int>(rank_me());
int mine;
```

• Shared memory can be accessed from a remote rank.

![Diagram showing private and shared memory allocation across ranks](image-url)
Futures

• UPC++ has no *implicit* blocking
  – We underline blocking operations
• A *future* holds a sequence of values and a state (ready / not ready)
• Waiting on the returned future lets user tailor degree of asynchrony they desire

```cpp
future<T> f1 = rget(gptr1);       // asynchronous op
future<T> f2 = rget(gptr2);       // unrelated work...
bool ready = f1.ready();          // non-blocking poll
wait(f1);                         // block until future is ready
T t = f1.result();                // fails if not ready
```
One-Sided Communication

- Remote read
  \[\text{future}<T> \ rget(\text{global\_ptr}<T> \ src);\]
- Remote write
  \[\text{future}<> \ rput(T \ val, \ \text{global\_ptr}<T> \ dest);\]
  - There is also a signaling version, that runs a handler at the destination \textit{after} rput operation is visible at the target
- Support for non-contiguous transfers
• Parallel computing of pi, but with a bug

```cpp
int main(int argc, char **argv) {
    init();
    int trials = atoi(argv[1]);
    int my_trials = (trials+rank_n()-1)/rank_n();
    global_ptr<int> hits =
        wait(broadcast(new_<int>(0), 0));
    srand(rank_me() * 17);
    for (int i=0; i < my_trials; i++) {
        int old_hits = wait(rget(hits));
        wait(rput(old_hits+hit(), hits));
    }
    barrier();
    if (rank_me() == 0)
        cout << "PI estimated to "
            << 4.0*(hits.local()) / trials;
    finalize();
}
```

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

- UPC++ has two basic forms of barriers:
  - Barrier: block until all other threads arrive
    ```cpp
    barrier();
    ```
  - Asynchronous barriers
    ```cpp
    future<> f = barrier_async(); // this thread is ready for barrier
    // do computation unrelated to barrier
    wait(f); // wait for others to be ready
    ```

- Shared data can be synchronously updated by sending the update to the owner as an RPC (remote procedure call)
Remote Procedure Call

```cpp
future<R> rpc(intrank_t r,
              F func, Args&&... args);
```

- Executes `func(args...)` on rank `r` and returns the result
- `R` is the return type of `func`
  - Empty future if `func` returns `void`
- There is also a ‘fire and forget’ version that returns no result
- Some restrictions apply to what UPC++ operations can be issued in an RPC: the `restricted context`
  - Limits on blocking operations from within an RPC
Pi in UPC++: RPC

- RPC used to synchronize updates

```cpp
int hits = 0;
int main(int argc, char **argv) {
    init();
    int trials = atoi(argv[1]);
    int my_trials = (trials+rank_n()-1)/rank_n();
    srand(rank_me() * 17);
    for (int i=0; i < my_trials; i++) {
        wait(rpc(0, [](int hit) { hits += hit; }, hit()));
    }
    barrier();
    if (rank_me() == 0)
        cout << "PI estimated to " << 4.0*hits/trials;
    finalize();
}
```

RPC can refer to global variable
send update to rank 0
block on the update
Pi in UPC++: Data Parallel Style w/ Collectives

- The previous version of Pi works, but is not scalable:
  - Updates are serialized on rank 0, ranks block on updates
- Use a reduction for better scalability:

```cpp
// int hits;  no global variables or shared memory
int main(int argc, char **argv) {
    ... 
    for (int i=0; i < my_trials; i++)
        my_hits += hit();
    my_hits = // input, binary op
        wait(allreduce(my_hits, std::plus<int>));
    // barrier(); barrier implied by reduction
    if (rank_me() == 0)
        cout << "PI: " << 4.0*my_hits/trials;
    finalize();
}
```
Distributed Objects

- Any C++ type can be made into a *distributed object*
- One instance on every rank of a team

```cpp
class Mesh { public: Mesh(A, B, C); private: ... };
A a; B b; C c;
dist_object<Mesh> dmesh(myTeam, a, b, c);
dist_object<int> counter(0); // over world team

- Collective over team, but not blocking

- Can access remote instances within team

```cpp
auto f1 = rpc(someRank,
    [foo](dist_object<Mesh> &remote) {
        remote->someFunction(foo);
        return remote->recalc(); },
    dmesh);

future<int> f2 = fetch(counter, someRank);
```
Pi in UPC++: Distributed Object Version

• Alternative fix to the race condition
• Have each rank update a separate counter:
  - Do it in a distributed object, have one rank compute sum

```c
int main(int argc, char **argv) {
  ...
  all_hits distributed across all ranks
  dist_object<int> all_hits(0);
  for (int i=0; i < my_trials; i++)
    *all_hits += hit();
  barrier();
  if (rank_me() == 0) {
    for (int i=0; i < rank_n(); i++)
      hits += wait(fetch(all_hits, i));
    cout << "PI estimated to " << 4.0*hits/trials;
  }
  finalize();
}
```

update element with local affinity
collect each rank's contribution
Distributed Objects in Stencil Code

- Communication in 1D stencil (nearest-neighbor computation):

```cpp
int main(int argc, char **argv) {
    ... declarations and initialization code omitted

    global_ptr<double> my_grid =
        new_array<int>(interior+2);
    dist_object<global_ptr<double>> grids(my_grid);

    global_ptr<double> left =
        wait(fetch(grids, (rank_me()+rank_n()-1)%rank_n()));
    global_ptr<double> right =
        wait(fetch(grids, (rank_me()+1)%rank_n()));

    for (int i=0; i < timesteps; i++) {
        future<double> f1 = rget(left+interior);
        future<double> f2 = rget(right+1);

        ... wait on futures and do computation
    }
    ...}
```
Summary

• UPC++ is a PGAS library that supports lightweight communication over GASNet-EX

• Close to the metal performance, lean interface
  – Trade offs to reduce overheads and increase flexibility
    • Asynchronous and explicit communication
    • Reduced consistency guarantees

• Advanced features not covered in talk:
  – Promises, callbacks, remote atomics, progress, memory model, teams

• V1.0 release targeted for September 30, 2017
  – Will include programmer’s guide
Acknowledgements

- Early work with UPC++ involved Yili Zheng, Amir Kamil, Kathy Yelick, and others [IPDPS ‘14]
- Pagoda Project (GASNet-EX and UPC++): Scott B. Baden (PI), Paul Hargrove (co-PI), John Bachan, Dan Bonachea, Steven Hofmeyer, Khaled Ibrahim, Mathias Jacquelin, Amir Kamil, Brian van Straalen
- This research was supported by the Exascale Computing Project (17-SC-20-SC), funded by the U.S. Department of Energy
- UPC++ V1.0 draft specification available at https://bitbucket.org/upcxx/upcxx
LBNL / UCB Collaborators

- Scott Baden
- John Bachan
- Dan Bonachea
- Paul Hargrove
- Steven Hofmeyr
- Khaled Ibrahim
- Mathias Jacquelin
- Amir Kamil*
- Brian van Straalen
- Yili Zheng*
- Eric Roman
- Marquita Ellis
- Costin Iancu
- Michael Driscoll
- Evangelos Georganas

- Penporn Koanantakool
- Leonid Oliker
- John Shalf
- Erich Strohmaier
- Samuel Williams
- Cy Chan
- Didem Unat*
- James Demmel
- Scott French
- Edgar Solomonik*
- Eric Hoffman*
- Wibe de Jong

External collaborators (& their teams!)

- Vivek Sarkar, Rice
- John Mellor-Crummey, Rice
- Mattan Erez, UT Austin

* Former LBNL/UCB

Thanks!