Partitioned Global Address Space Programming with Unified Parallel C (UPC)

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

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
  - Make an array of counts
    - A's  B's  C's  ...  Y's  Z's
  - Each processor works on a subset of the stream and lock each entry before incrementing
- 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”

PGAS Languages

- Global address space: thread may directly read/write remote data
  - Hides the distinction between shared/distributed memory
- Partitioned: data is designated as local or global
  - Does not hide this: critical for locality and scaling

NERSC Facility
Computational Research

ESnet Facility
Applied Mathematics

Computational Science
Computer Science

Opportunities for summer internship, joint projects, etc.
Programming Challenges and Solutions

**Message Passing Programming**
- Divide up domain in pieces
- Each compute one piece
- Exchange (send/receive) data

**Global Address Space Programming**
- Each start computing
- Grab whatever you need whenever

**PVM, MPI, and many libraries**
- ~10% of NERSC apps use some kind of PGAS-like model

Mixed Model Programming

- We can run 1 MPI process per core, but there are problems with 6-12+ cores/socket:
  - Insufficient memory: user level data and internal buffers
  - Runtime overheads: copying and synchronization
- OpenMP, Pthreads, or other shared memory models
  - No control over locality, e.g., Non-Uniform Memory Access
  - No explicit memory movement, e.g., accelerators or NVRAM
- Tuning is non-obvious
  - Tradeoff between speed and memory footprint

Shared Memory vs. Message Passing

**Shared Memory**
- Advantage: Convenience
  - Can share data structures
  - Just annotate loops
  - Closer to serial code
- Disadvantages
  - No locality control
  - Does not scale
  - Race conditions

**Message Passing**
- Advantage: Scalability
  - Locality control
  - Communication is all explicit in code (cost transparency)
- Disadvantage
  - Need to rethink entire application / data structures
  - Lots of tedious pack/unpack code
  - Don’t know when to say “receive” for some problems

Science Across the “Irregularity” Spectrum

- Massive Independent Jobs for Analysis and Simulations
- Nearest Neighbor Simulations
- All-to-All Simulations
- Random access, large data Analysis
Programming Models for De Novo Assembly

• Current approach: buy large shared memory machines
• For many problems, these are not large enough
• UPC De Bruijn graph construction: 2 days to 1 minute
  ~3K speedup on 3K cores (relative to Perl) and scalability!

Work by Evangelos Georganas, Jarrod Chapmanz, Khaleed Ibrahim, Daniel Rokhsar, Leonid Oliker, and Katherine Yelick

Bringing Users Along: UPC Experience

1991 Active Msgs are fast
1992 First Split-C (compiler class)
1992 First AC (accelerators + split memory)
1993 Split-C funding
1993 DOE

• Ecosystem:
  – Users with a need (fine-grained random access)
  – Machines with RDMA (not full hardware GAS)
  – Common runtime; Commercial and free software
  – Sustained funding and Center procurements

• Success models:
  – Adoption by users: vectors → MPI, Python and Perl, UPC/CAF
  – Influence traditional models: MPI 1-sided; OpenMP locality control
  – Enable future models: Chapel, X10, …

History of UPC

• Initial Tech. Report from IDA in collaboration with LLNL and UCB in May 1999 (led by IDA).
  – Based on Split-C (UCB), AC (IDA) and PCP (LLNL)
• UPC consortium participants (past and present) are:
  – “UPC is a community effort, well beyond UCB/LBNL”
• Design goals: high performance, expressive, consistent with C goals, …, portable
• UPC Today
  – Multiple vendor and open compilers (Cray, HP, IBM, SGI, gcc-upc from Intrepid, Berkeley UPC)
  – “Pseudo standard” by moving into gcc trunk
  – Most widely used on irregular / graph problems today

Tutorial Outline

1. Overview of UPC
2. Serial Optimizations in UPC
3. Shared Memory Optimizations in UPC
4. Distributed Memory Tuning in UPC
5. Beyond UPC
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
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Using this fact, plus the 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);
}
```

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
  - \( \pi = 4 \times \text{ratio} \)
Each thread calls “hit” separately

Helper Code for Pi in UPC

- Required includes:

```c
#include <stdio.h>
#include <math.h>
#include <upc.h>
```

- Function to throw dart and calculate where it hits:

```c
def hit()
{
    int const rand_max = 0xFFFFFFFF;
    double x = ((double) rand()) / RAND_MAX;
    double y = ((double) rand()) / RAND_MAX;
    if ((x*x + y*y) <= 1.0) {
        return(1);
    } else {
        return(0);
    }
}
```

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared 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. Why?
Pi in UPC: Shared Memory Style

- Parallel computing of pi, but with a bug
  ```
  shared int hits;
  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();
  upc_barrier;
  if (MYTHREAD == 0) {
    printf("PI estimated to %f.\n", 4.0*hits/trials);
  }
  ```

What is the problem with this program?

Pi in UPC: Shared Array Version

- Alternative fix to the race condition
  ```
  shared int all_hits [THREADS];
  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++)
    all_hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0) {
    for (i=0; i < THREADS; i++) hits += all_hits[i];
    printf("PI estimated to %f.\n", 4.0*hits/trials);
  }
  ```

Shared Arrays Are Cyclic By Default

- Shared scalars 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 */
```

- In the pictures below, assume THREADS = 4

Think of linearized C array, then map in round-robin

As a 2D array, y is logically blocked by columns

z is not

UPC Synchronization
UPC Global Synchronization

- UPC has two basic forms of barriers:
  - Barrier: block until all other threads arrive
    `upc_barrier`
  - Split-phase barriers
    `upc_notify`: this thread is ready for barrier
    `do computation unrelated to barrier`
    `upc_wait`: wait for others to be ready
- Optional labels allow for debugging
  
  ```
  #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:
  `upc_lock_t`
- Locks must be allocated before use:
  `upc_lock_t *upc_all_lock_alloc(void);`
  allocates 1 lock, pointer to all threads
  `upc_lock_t *upc_global_lock_alloc(void);`
  allocates 1 lock, pointer to one thread
- To use a lock:
  `void upc_lock(upc_lock_t *l)`
  `void upc_unlock(upc_lock_t *l)`
  use at start and end of critical region
- Locks can be freed when not in use
  `void upc_lock_free(upc_lock_t *ptr);`

Pi in UPC: Shared Memory Style

- Like pthreads, but use shared accesses judiciously

```c
shared int hits;    // one shared scalar variable
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;    // other private variables
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    int trials = atoi(argv[1]);        // create a lock
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++) {    // accumulate hits locally
        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);
}
```

Recap: Private vs. Shared Variables in UPC

- We saw several kinds of variables in the pi example
  - Private scalars (`my_hits`)
  - Shared scalars (`hits`)
  - Shared arrays (`all_hits`)
  - Shared locks (`hit_lock`)
UPC Collectives

UPC (Value-Based) Collectives

- A portable library of collectives on scalar values (not arrays)
- Example: `x = bupc_allv_reduce(double, x, 0, UPC_ADD)`

Computational Collectives:
- reductions and scan (parallel prefix)
- Portable implementation available from:
  - [http://upc.lbl.gov/download/dist/upcr_preinclude/bupc_collectivev.h](http://upc.lbl.gov/download/dist/upcr_preinclude/bupc_collectivev.h)

Data movement collectives:
- broadcast, scatter, gather
- UPC also has more general collectives over arrays
  - [http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf](http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf)

Pi in UPC: Data Parallel Style

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

```c
#include <bupc_collectivev.h> // shared int hits

main(int argc, char **argv) {
    ... for (i=0; i < my_trials; i++)
        my_hits += hit();
    my_hits = bupc_allv_reduce(int, my_hits, 0, UPC_ADD); // type, input, thread, op
    if (MYTHREAD == 0) printf("PI: %f", 4.0*my_hits/trials);
}
```

UPC Collectives in General

- The UPC collectives interface is in the language spec:
  - [http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf](http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf)
- 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
- Simple interface for collecting scalar values (int, double, …)
  - Berkeley UPC value-based collectives
  - Works with any compiler
  - [http://upc.lbl.gov/docs/user/README-collectivev.txt](http://upc.lbl.gov/docs/user/README-collectivev.txt)
Full UPC Collectives
- Value-based collectives pass in and return scalar values
- But sometimes you want to collect over arrays
- When can a collective argument begin executing?
  - Arguments with affinity to thread \( i \) are ready when thread \( i \) calls the function; results with affinity to thread \( i \) are ready when thread \( i \) returns.
  - This is appealing but it is incorrect: In a broadcast, thread 1 does not know when thread 0 is ready.

UPC Collective: Sync Flags
- In full UPC Collectives, blocks of data may be collected
- A extra argument of each collective function is the sync mode of type `upc_flag_t`
- Values of sync mode are formed by or-ing together a constant of the form `UPC_IN__X_SYNCH` and a constant of the form `UPC_OUT__Y_SYNCH`, where \( X \) and \( Y \) may be `NO`, `MY`, or `ALL`.
- If `sync_mode` is `(UPC_IN__X_SYNCH | UPC_OUT__Y_SYNCH)`, then if \( X \) is:
  - `NO` the collective function may begin to read or write data when the first thread has entered the collective function call,
  - `MY` the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call, and
  - `ALL` the collective function may begin to read or write data only after all threads have entered the collective function call
- and if \( Y \) is
  - `NO` the collective function may read and write data until the last thread has returned from the collective function call,
  - `MY` the collective function call may return in a thread only after all reads and writes of data with affinity to the thread are complete, and
  - `ALL` the collective function call may return only after all reads and writes of data are complete.

Example: Vector Addition
- Questions about parallel vector additions:
  - How to layout data (here it is cyclic)
  - 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 Sharing with upc forall()

- A common idiom:
  - Loop over all elements; work on those owned by this thread
- UPC adds a special type of loop
  ```c
  upc forall(init; test; loop; affinity)
  statement;
  ```
- Programmer indicates the iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression indicates which iterations to run on each thread.
  It may have one of two types:
  - Integer: `affinity%THREADS` is MYTHREAD
  - Pointer: `upc_threadof(affinity)` is MYTHREAD
- Syntactic sugar for:
  ```c
  for(i=0; i<N; i++) if (MYTHREAD == i%THREADS)
  ```
- Compilers will sometimes do better than this, e.g.,
  ```c
  for(i=MYTHREAD; i<N; i+=THREADS)
  ```

Vector Addition with upc forall

- Vector addition can be written 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];
    }
  }
  ```
- The code would be correct but slow if the affinity expression were `i+1` rather than `i`.
- Equivalent code could use `&sum[i]` for affinity and would still work if you change the layout of `sum`

Blocked Layouts in UPC

- Array layouts are controlled by blocking factors:
  - Empty (cyclic layout)
  - `[*]` (blocked layout)
  - `[b]` (fixed block size)
  - `[0]` or `[]` (indefinite layout, all on 1 thread)
- Vector addition example can be rewritten as follows using a cyclic or (maximally) blocked layout
  ```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];
    }
  }
  ```
Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:
  - Empty (cyclic layout)
  - [*] (blocked layout)
  - [0] or [] (indefinite layout, all on 1 thread)
  - [b] or [b1,b2]...[bn] = [b1*b2*...bn] (fixed block size)
- The affinity of an array element is defined in terms of:
  - block size, a compile-time constant
  - and THREADS.
- Element i has affinity with thread
  \[(i / \text{block	extunderscore size}) \mod \text{THREADS}\]
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping

2D Array Layouts in UPC

- Array a1 has a row layout and array a2 has a block row layout.
  
  \[
  \text{shared} [\text{m}] \text{ int} \ a1 [\text{n}][\text{m}]; \\
  \text{shared} [\text{k*}\text{m}] \text{ int} \ a2 [\text{n}][\text{m}]; \\
  \]
- If \((\text{k} + \text{m}) \mod \text{THREADS} = = 0\) then a3 has a row layout
  
  \[
  \text{shared int} \ a3 [\text{n}][\text{m+k}]; \\
  \]
- To get more general HPF and ScaLAPACK style 2D blocked layouts, one needs to add dimensions.
  
  \[
  \text{Assume r*c = THREADS;} \\
  \text{shared [b1][b2] int a5 [m][n][r][c][b1][b2];} \\
  \text{or equivalently} \\
  \text{shared [b1*b2] int a5 [m][n][r][c][b1][b2];} \\
  \]

Pointers to Shared vs. Arrays

- In the C tradition, array can be access through pointers
- Here is the vector addition example using pointers

```c
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    shared int *p1, *p2;    // v1
    p1=v1; p2=v2;
    for (i=0; i<N; i++, p1++, p2++)
        if (i %THREADS = = MYTHREAD)
            sum[i] = *p1 + *p2;
}
```

UPC Pointers

Where does the pointer point?

<table>
<thead>
<tr>
<th>Where does the pointer reside?</th>
<th>Local</th>
<th>Global (to shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>Shared</td>
<td>p3</td>
<td>p4</td>
</tr>
</tbody>
</table>

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 local memory (p3) is not recommended.
UPC Pointers

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block)

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

- 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

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

- int *shared p3;
  - Not recommended

- shared int *shared p4;
  - Use to build shared linked structures, e.g., a linked list

UPC Pointers

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not vice versa!
- When casting a pointer-to-shared to a pointer-to-local, the thread number of the pointer to shared may be lost
- Casting of shared to local is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast

Example implementation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Thread</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>49</td>
<td>38</td>
</tr>
<tr>
<td>37</td>
<td>48</td>
<td>38</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Phase is needed to implement p++ within/ between threads
Special Functions

- `size_t upc_threadof(shared void *ptr);` returns the thread number that has affinity to the pointer to shared.
- `size_t upc_phaseof(shared void *ptr);` returns the index (position within the block) field of the pointer to shared.
- `shared void *upc_resetphase(shared void *ptr);` resets the phase to zero.

Global Memory Allocation

- `shared void *upc_alloc(size_t nbytes);`  
  nbytes: size of memory in bytes  
  - Non-collective: called by one thread  
  - The calling thread allocates a contiguous memory space in the shared space with affinity to itself.

- `shared [] double [n] p2 = upc_alloc(n*sizeof(double));`

- `void upc_free(shared void *ptr);`  
  - Non-collective function; frees the dynamically allocated shared memory pointed to by ptr

Distributed Arrays Directory Style

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

- The consistency model defines the order in which one thread may see another thread's accesses to memory.
  - If you write a program with unsynchronized accesses, what happens?
  - Does this work?
    ```c
    data = ... while (!flag) { };  
    flag = 1; ... = data;  // use the data
    ```
- UPC has two types of accesses:
  - Strict: will always appear in order
  - Relaxed: May appear out of order to other threads
- There are several ways of designating the type, commonly:
  - Use the include file:
    ```c
    #include <upc_relaxed.h>
    ```
  - Which makes all accesses in the file relaxed by default
  - Use strict on variables that are used as synchronization (flag)

Properties of UPC memory model

- Definitions:
  - A data race is:
    - Two concurrent memory operations from two different threads to the same memory location in which at least one is a write.
  - A race-free program is one in which:
    - All executions of the program are free of data races (would be nice if the user could only worry about naïve implementations)
  - And states that programs will be sequentially consistent (behave as if all operations from each thread execute in order) if either of the following holds:
    - The program is race-free
    - The program contains no relaxed operations

Intuition on Strict Orderings

- Each thread may “build” its own total order to explain behavior
- They all agree on the strict ordering shown above in black, but
  - Different threads may see relaxed writes in different orders
    - Allows non-blocking writes to be used in implementations
  - Each thread sees own dependencies, but not those of other threads
    - Weak, but otherwise there would place consistency requirements on some relaxed operations (e.g., local cache control insufficient)
    - Preserving dependencies requires usual compiler/hw analysis

Synchronization- Fence

- Upc provides a fence construct
  - Equivalent to a null strict reference, and has the syntax
    ```c
    upc_fence;
    ```
  - UPC ensures that all shared references issued before the upc_fence are complete
UPC Performance Features

UPC Compiler Implementation

**UPC-to-C translator**
- Pros: portable, can use any backend C compiler
- Cons: may lose program information between the two compilation phases
- Example: Berkeley UPC

**UPC-to-object-code compiler**
- Pros: better for implementing UPC specific optimizations
- Cons: less portable
- Example: GCC UPC and most vendor UPC compilers

Exemplar Programming System Stack on Cray

Berkeley UPC Software Stack

Tip: you can choose your favorite C compiler (e.g., clang, icc, gcc, nvcc, xlc) as the backend compiler with B UPC.
GASNet Software Stack

- PGAS Programming Systems (e.g., BUPC, CAF 2.0, Chapel, OpenSHMEM, Titanium, and DEGAS)
- GASNet
  - Active Messages
  - One-sided Communication
  - Collective Communication
- Low-level communication APIs (e.g., Cray GNI, IBM PAMI, IB Verbs, Portals 4, UDP, shared-memory)

When Address Translation Overheads Matter?

**Case 1: access local data**
1. Get the partition id of the global address (1 cycle)
2. Check if the partition is local (1 cycle)
3. Get the local address of the partition (1 cycle)
4. Access data through the local address (1 cycle)

3 CPU cycles for address translation vs. 1 cycle for real work (Bad: 3X overhead)

**Case 2: access remote data**
1. Get the partition id of the global address (1 cycle)
2. Check if the partition is local (1 cycle)
3. Get the local address of the partition (1 cycle)
4. Access data through the network (~10^4 cycles)

3 CPU cycles for address translation vs. ~10^4 cycles for real work (Good: 0.3% overhead)

Implementing UPC Shared Data Access

```c
shared int s;
s = 5;
```

**UPC-to-C Translator**

```
UPCR_PUT_PSHARED_VAL(s, 0, 5, 4);
```

**UPC Runtime**

Runtime Address Translation Overheads

GASNet

Tip: try "upcc -trans test.upc" to see the translated C code for Berkeley UPC.

Performance: Pointer-to-local vs. Pointer-to-shared

<table>
<thead>
<tr>
<th>Data Size (bytes)</th>
<th>Shared Data Access Time on 32-core AMD</th>
<th>Shared Data Access Time on 8-core Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>64</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>128</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>256</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>512</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

Tip: Cast a pointer-to-shared to a regular C pointer for accessing the local portion of a shared object. E.g., `int *p = (int *)ptr; p[0] = 1;`
How to Amortize Address Translation Overheads

- Move data in chunks
  upc_mem(cpy|put|get)(...)
  non-blocking upc_mem(cpy|put|get) are even better

- Cast pointer-to-shared to pointer-to-local
  #include<upc_castable.h> // in UPC 1.3
  void *upc_cast(const shared void *ptr);

Thread 1's perspective

Source Node

T1

src

T2

tgt

Physical Shared-memory

Virtual Address Space

Non-blocking Memcopy is crucial to performance

Hardware can reorder operations to improve performance (e.g., network adaptive routing), but possible data dependencies may prohibit it.

Put 1 to *p1

Put 2 to *p2

Src Node

These two Put operations may be completed out-of-order if p1 and p2 are different addresses.

By using non-blocking memcpy, the user gives the permission to complete memory operations in arbitrary order.

Tip: UPC 1.3 enables you to cast a pointer-to-shared with affinity to another UPC thread to a pointer-to-local if both threads share the same physical node.

UPC 1.3 Non-blocking Memcopy

#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

// Implicit handle version, no handle management by user
void upc_memcpy_nbi(...); // parameters the same as upc_memcpy
void upc_synci(); // sync all issued implicit operations
int upc_sync_attempti(); // test the completion status of // implicit operations

UPC 1.3 Atomic Operations

- More efficient than using locks when applicable
  upc_lock();
  update();
  upc_unlock();
  vs
  atomic_update();

- Hardware support for atomic operations are available, but
  Only support limited operations on a subset of data types. e.g.,
  Atomic_ops from different processors may not be atomic to each other

Atomic_CAS on uint64_t

Atomic_Add on double
UPC 1.3 Atomic Operations (cont.)

- Key new idea: atomicity domain
  Users specify the operand data type and the set of operations over which atomicity is needed

  // atomicity domain for incrementing 64-bit integers
  upc_atomicdomain_t *domain =
  upc_all_atomicdomain_alloc(UPC_INT64, UPC_INC, 0);

  upc_atomic_strict(upc_atomicdomain_t *domain,
  void * restrict fetch_ptr,
  upc_op_t op,
  shared void * restrict target,
  const void * restrict operand1,
  const void * restrict operand2);

  upc_atomic_relaxed(...); // relaxed consistency version

Performance of UPC

PGAS Languages have Performance Advantages

- Strategy for acceptance of a new language
  - Make it run faster than anything else

- Keys to high performance
  - Parallelism:
    - Scaling the number of processors
    - Maximize single node performance
    - Generate friendly code or use tuned libraries (BLAS, FFTW, etc.)
  - Avoid (unnecessary) communication cost
    - Latency, bandwidth, overhead
    - Berkeley UPC and Titanium use GASNet communication layer
  - Avoid unnecessary delays due to dependencies
    - Load balance; Pipeline algorithmic dependencies

Berkeley UPC Compiler

- Used by bupc and gcc-upc
- Compiler-generated C code
- UPC Runtime system
- GASNet Communication System
- Network Hardware
- Network-independent
- Compiler-independent
- Language-independent
- Platform-independent
- Network-independent

UPC Code → UPC Compiler

Used by Cray
UPC, CAF, Chapel, Titanium, and others
One-Sided vs Two-Sided

- A one-sided put/get message can be handled directly by a network interface with RDMA support.
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data.
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
  - Ordering requirements on messages can also hinder bandwidth

Bandwidths on Cray XE6 (Hopper)

- GASNet better for latency across machines

GASNet: Portability and High-Performance

- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point (N ½ ) differs by one order of magnitude
- This is not a criticism of the implementation!
GASNet: Portability and High-Performance

GASNet at least as high (comparable) for large messages

Joint work with UPC Group; GASNet design by Dan Bonachea

Communication Strategies for 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

Overlapping Communication

- Goal: make use of “all the wires all the time”
  - Schedule communication to avoid network backup
- Trade-off: overhead vs. overlap
  - Exchange has fewest messages, less message overhead
  - Slabs and pencils have more overlap; pencils the most
- Example: Class D problem on 256 Processors

<table>
<thead>
<tr>
<th>Communication Strategy</th>
<th>Example Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exchange (all data at once)</td>
<td>512 Kbytes</td>
</tr>
<tr>
<td>Slabs (contiguous rows that go to 1 processor)</td>
<td>64 Kbytes</td>
</tr>
<tr>
<td>Pencils (single row)</td>
<td>16 Kbytes</td>
</tr>
</tbody>
</table>

Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea
NAS FT Variants Performance Summary

- Slab is always best for MPI; small message cost too high
- Pencil is always best for UPC; more overlap

FFT Performance on BlueGene/P

- UPC implementation consistently outperform MPI
- Uses highly optimized local FFT library on each node
- UPC version avoids send/receive synchronization
  - Lower overhead
  - Better overlap
  - Better bisection bandwidth
- Numbers are getting close to HPC record on BG/P

FFT Performance on Cray XT4

- 1024 Cores of the Cray XT4
  - Uses FFTW for local FFTs
  - Larger the problem size the more effective the overlap

Event Driven LU in UPC

- DAG Scheduling before it’s time
- 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

Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea

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

Event Driven LU in UPC
UPC HPL Performance

- 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)

MILC (QCD) Performance in UPC

- MILC is Lattice Quantum Chromo-Dynamics application
- UPC scales better than MPI when carefully optimized

Summary

- UPC designed to be consistent with C
  - Ability to use pointers and arrays interchangeably
- Designed for high performance
  - Memory consistency explicit; Small implementation
  - Transparent runtime
- gcc version of UPC: http://www.gccupc.org/
- Berkeley compiler http://upc.lbl.gov
- Vendor compilers: Cray, IBM, HP, SGI,…
Application Development in UPC

Topics

• Starting a project
  – Choosing the right SDK
  – Interoperability with other programming models
    • OpenMP, MPI, CUDA...

• Shared memory programming
  – Data layout and allocation
  – Computational efficiency ("serial" performance
  – Synchronization
  – Managing parallelism – data parallel & dynamic
    tasking
  – UPC and OpenMP

Topics (2)

• Distributed memory programming
  – UPC and MPI

• Tuning communication performance
• Hybrid parallelism

UPC SDKs

• Multiple SDKs are available
  – Portable
    • BUPC provided by LBL is portable – available at
      http://upc.lbl.gov
    • GUPC provided by Intrepid, gcc based, portable, uses
      BUPC runtime
  – Vendor SDKs – Cray UPC XT/XE

• UPC has been shown to interoperate with
  • MPI, OpenMP, CUDA, Intel TBB, Habanero-C
  • Any pthreads based library e.g. MKL

• Some interoperability aspects are implementation
  specific, e.g. who owns main()
  • E.g. http://upc.lbl.gov/docs/user/interoperability.shtml
Shared Memory Programming

- Performance determined by:
  - Locality – placement, data initialization
  - Computational efficiency
  - Synchronization performance
  - Management of parallelism

When should memory be shared (shared)?
When should memory be blocked (shared [])?

Pointer Arithmetic and Data Placement

- Memory is allocated with upc_alloc, upc_all_alloc with affinity to a certain thread
- The pointer type determines the address arithmetic rules and the "locality" of access

```cpp
shared double *p1;
shared [*] double *ps;
shared [] double *pi;
for(i=0; i < N; i++) {
    p1[i] = i;  1 3 2 4
    ps[i] = i;  1 2 3 4
    pi[i] = i;  1 2 3 4
}
```

2-D Stencil – Laplace Filter – block cyclic

```cpp
shared double matrix[ROWS][COLS];
...
main() {
    for(i=0; i < ROWS; i++)
        for(j = 0; ; j < COLS; j++) {
            up = (i == 0) ? 0 : matrix[i-1][j];
            down = (i == ROWS-1) ? 0 : matrix[i+1][j];
            left = (j == 0) ? 0 : matrix[i][j-1];
            right = (j == COLS - 1) ? 0 : matrix[i][j+1];
            tmp[i][j] = 4 * matrix[i][j] - up - down - left - right;
        }
}
```

Block cyclic layout easy to choose when porting codes, bad for locality
2-D Stencil – Laplace Filter – block layout

shared [*] double matrix[ROWS][COLS];

main() {
    for(i=0; i < ROWS; i++) {
        for(j = 0; j < COLS; j++) {
            up = (i == 0) ? 0 : matrix[i-1][j];
            down = (i == ROWS-1) ? 0 : matrix[i+1][j];
            left = (j == 0) ? 0 : matrix[i][j-1];
            right = (j == COLS - 1) ? 0 : matrix[i][j+1];
            tmp[i][j] = 4 * matrix[i][j] - up - down - left - right;
        }
    }
}

Blocked layout easy to choose when porting codes, good for locality, code not portable

2-D Stencil – Laplace Filter – directory

typedef shared [] double * SDPT;
shared SDPT matrix[ROWS];
SDPT local_dir[ROWS];

main() {
    main() {
        matrix[my_row] = upc_alloc(..); //allocate ptrs to rows
        upc_barrier;
        local_dir[i] = matrix[i]; //local copies of dir entries
        for(i=0; i < ROWS; i++) {
            for(j = 0; j < COLS; j++) {
                up = (i == 0) ? 0 : local_dir[i-1][j];
                right = (j == COLS - 1) ? 0 : local_dir[i][j+1];
                tmp[i][j] = 4 * local_dir[i][j] - up - down - left - right;
            }
        }
    }
}

Directory based approach provides locality and portability

Computational Efficiency
(ALWAYS Cast to C)

Computational Intensity – ALWAYS cast to C

Cast a pointer-to-shared to a regular C pointer for accessing the local portion of a shared object.
E.g., int *p = (int *)pts; p[0] = 1;
Application Examples

LULESH - https://codesign.llnl.gov/lulesh.php

- Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
- Models explicit hydrodynamics portion of ALE3D
- Particular application is a Sedov blast wave problem
- Used to explore various programming models, e.g. Charm++, Chapel, Loci, Liszt
- Solves equations on a staggered 3D spatial mesh
- Most communication is nearest neighbor on a hexahedral 3D grid

LULESH OMP

- Doesn’t scale beyond 12 cores (2 NUMA nodes)

LULESH OMP Parallel Initialization

- Parallel initialization helps only slightly
- Still doesn’t scale beyond 18 cores
- Uses temporary arrays with malloc and free in many calls
LULESH OpenMP to UPC

• LULESH authors advise: “Do not make simplifications”
• None-the-less, I made some simplifications:
  - Primarily for readability and clarity
  - Why follow certain impl. choices? (e.g. temp arrays)
• Performance improvements in UPC at scale
  - Primarily due to locality management, not simplifications
• UPC with one thread is slower than C++ serial
  - Best UPC 298s, best C++ serial 283s

LULESH Naïve UPC – block cyclic distribution

• Shared arrays distributed cyclically (default)
• Replicate data to make it private where possible
• Poor compared to OMP

LULESH UPC Blocked Memory Layout

• Cyclic layout poor fit for communication pattern
• Contiguous layout (blocked) reduces communication

```
shared[*] double x[N * THREADS];
```

LULESH UPC Communication

Cyclic layout                         Contiguous layout
**LULESH UPC Cast Shared to Private**

- Use private pointer to the thread block in shared array
  
  ```c
  double* my_x = (double*)(x + MYTHREAD * BSIZE)
  ```

**XSBench - Embarrassingly parallel**

- Monte Carlo simulation of paths of neutrons traveling across a reactor core
  - 85% of runtime in calculation of macroscopic neutron cross sections

  ```
  random_sample
  binary_search
  for each nuclide
    lookup_bounding_micro_xs
    interpolate
    accumulate_macro_xs
  ```

- Uses a lot of memory

**XSBench OMP Doesn’t Scale**

- Option to add flops; according to README:
  “Adding flops has so far shown to increase scaling, indicating that there is in fact a bottleneck being caused by the memory loads”

**XSBench OMP Initialization**

- But memory locality is the problem (on NUMA)
- Adding parallel initialization makes it scale
XSBench UPC

- Private replication of data
- Except: make largest memory array shared

Synchronization Performance

XSBench UPC No Shared Memory

- Improves if all memory is private
- Can’t do for large problems, e.g. 355 isotopes requires 60GB for full replication on 48 cores

Barriers, locks, atomics, collectives....

- OpenMP provides an implicit model of synchronization
- The UPC language provides rich synchronization primitives
  - e.g. UPC 1.3 atomics
- Some are well optimized for multicore performance
  "Optimizing Collective Communication on Multicore". Nishtala&Yelick, HotPar'09
- In general, UPC synchronization performs much better than OpenMP synchronization or other pthreads based libraries (implementation does matter)

#pragma omp critical
  -> bupc_allv_reduce_all()
LULESH UPC Procs vs Pthreads

- At 48 cores, pthreads takes 33s, processes only 22s
- Top non-app code functions with pthreads:
  - upcr_wait_internal 15% (barrier)
  - gasnete_coll_broadcast 2%
  - gasnete_coll_gather 2%
- Top non-app code functions with pinned procs:
  - gasnete_pshmbARRIER_wait 5%
- For comparison, collectives with pinned procs:
  - gasnete_coll_broadcast 0.2% (15x)
  - gasnete_coll_gather 0.04% (75x)

Lessons Learned

- On a large NUMA system, managing remote memory access is key
  - Good preparation for distributed memory?
- UPC
  - Contiguous blocking is effective at reducing communication
  - Explicitly cast to private whenever possible
  - Procs can be significantly faster than pthreads
    Hybrid PGAS Runtime Support for Multicore Nodes
    Blagojevic, Hargrove, Iancu, Yelick. PGAS 2010
  - Replication to private can help, but limited by available memory -> replicate fixed amount?

Managing Parallelism

- Data parallel constructs in UPC – upc forall
  - SPMD, shorthand for filtering the computation performed by a task
  - Not real equivalent of #pragma omp for..
- Task parallelism in OMP: #pragma omp task
- Written in stock UPC, works on
  - shared memory - comparable to OpenMP tasking
  - distributed memory – akin to Charm++
- Provides:
  - Init, termination
  - Locality aware distributed work-stealing
  - Synchronization for dependent task graphs
Task Library API

taskq_t *taskq_all_alloc(int nFunc, void *func1, int input_size1, int output_size1, ...);

int taskq_put(taskq_t *taskq, void *func, void *in, void *out);

int taskq_execute(taskq_t *taskq);

int taskq_steal(taskq_t *taskq);

void taskq_wait(taskq_t *taskq);

void taskq_fence(taskq_t *taskq);

int taskq_all_isEmpty(taskq_t *taskq);

Hierarchical Work Stealing on Manycore Clusters
Min, lancu,Yelick. PGAS 2011

UPC Task Library – Shared Memory

Performance of Victim Selection Policies on 8 Core Nehalem SMP

Exec. Time Normalized to gcc‐OpenMP (Lower the Better)

Performance of Victim Selection Policies on 256 cores on Carver Cluster

Speedup Normalized to the Random Policy

Distributed Memory Programming
**UPC and MPI**

- Send/Recv carry both data and synchronization
- One-sided carries only data
- When porting codes from MPI two-sided to one sided, a Send/Recv pair needs to be replaced with Put/Get and producer-consumer semantics

- There are also performance differences
  - UPC can saturate the network with fewer cores active per node
  - It alleviates the need for packing messages

---

**Cray XE6 Application Performance**

- Bar chart comparing 64 and 256 procs
- Percentage UPC over MPI speedup

**Cray XE6 BW Saturation (hopper @ NERSC)**

- Heatmaps showing bandwidth saturation

---

**Tuning Communication Performance**
**UPC Trends**

- In MPI, large messages or large message concurrency (messages per core, ranks per node) is required for performance.
- In UPC, communication overlap is beneficial:
  - with other communication
  - with other computation
- In UPC:
  - Pays to think about increasing the message concurrency
  - Sometimes need to take care to avoid congestion

*Congestion Avoidance on Manycore HPC Systems*  
Luo, Panda, Ibrahim, Iancu. ICS’12

- Again, avoiding pthreads improves performance

**Saturation IB**

- Messages < 1024 benefit from concurrent injection
- Messages > 8K benefit from throttling

**Throughput and Message Concurrency**

- Limiting the number of outstanding messages provides 5X speedup (expected 32X slower)

**When To Use It?**

- With irregular parallelism with “natural small messages”
- When hybrid parallelism makes packing complex
- Need to mix with pthreads based libraries and want to perform communication from within pthreads
  - Implementation specific, but available
- Do not want to worry about matching communication concurrency to intra-node concurrency…

- Challenges:
  - Exporting data, do not want to modify data structures
  - One-sided is different, need to understand it…
Beyond UPC

C++ is Important in Scientific Computing

Languages use at NERSC: 75% Fortran, 45% C/C++, 10% Python with C++ at least as important as C

- DOE’s Exascale Co-Design Centers
  - ExaCT: Combustion simulation (uniform and adaptive mesh)
  - ExMatEx: Materials (multiple codes)
  - CESAR: Nuclear engineering (structures, fluids, transport)
  - NNSA Center: umbrella for 3 labs

- “Proxy apps” to represent them
  - 10 codes
  - 4 in C++

A “Compiler-Free” Approach for PGAS

- Leverage the C++ standard and compilers
  - Implement UPC++ as a C++ template library
  - C++ templates can be used as a mini-language to extend the C++ grammar

- New features in C++ 11 makes UPC++ more powerful
  - E.g., async, auto type inference, lambda functions
  - C++ 11 is well-supported by major compilers
**C++ Generic Programming for PGAS**

- C++ templates enable generic programming
  - Parametric template definition
    ```cpp
template<class T>
    struct array { T *elements; size_t sz; };
    ```
  - Template instantiation
    ```cpp
    array<double> A; array<complex> B;
    ```
- UPC++ uses templates to express shared data
  ```cpp
  shared_var<int> s; // shared int s in UPC
  shared_array<int> sa(8); // shared int sa[8] in UPC
  ```

**Shared Variable Example**

```
#include <upcxx.h>
using namespace upcxx;

shared_var<int> s = 0; // shared by all threads
shared_lock l;     // SPMD
void update() {
  for (int i=0; i<100; i++)
    s = s + 1; // a race condition
  barrier();
  // shared data accesses with lock protection
  for (int i=0; i<100; i++)
    { l.lock(); s = s + 1; l.unlock(); }
}
```

**UPC++ Translation Example**

```
shared_array<int, 1> sa(100); sa[0] = 1; // "[]" and "+" overloaded
C++ Compiler

tmp_ref = sa.operator[] (0);
tmp_ref.operator = (1);
UPC++ Runtime

Yes Is tmp_ref local? No

Local Access Remote Access
```
Dynamic Global Memory Management

- Global address space pointers (pointer-to-shared)
  \[ \text{global_ptr}<\text{data_type}\> \text{ pts}; \]
- Dynamic shared memory allocation
  \[ \text{allocate}<\text{T}\>(\text{uint32_t} \text{ place}, \text{size_t count}); \]
  \[ \text{allocate}<\text{void}\>(\text{uint32_t} \text{ place}, \text{size_t nbytes}); \]

Example: allocate space for 512 integers on thread 2
\[ \text{global_ptr}<\text{int}\> p = \text{allocate}<\text{int}\>(2, 512); \]

One-Sided Data Transfer Functions

// Copy \text{count} elements of \text{T} from src to dst
\[ \text{upcxx}::\text{copy}<\text{T}>(\text{global_ptr}<\text{T}\> \text{ src}, \text{global_ptr}<\text{T}\> \text{ dst}, \text{size_t count}); \]

// Non-blocking version of copy
\[ \text{upcxx}::\text{async\_copy}<\text{T}>(\text{global_ptr}<\text{T}\> \text{ src}, \text{global_ptr}<\text{T}\> \text{ dst}, \text{size_t count}); \]

// Synchronize all previous async\_copy's
\[ \text{upcxx}::\text{async\_copy\_fence}(); \]

UPC++ Equivalents for UPC Users

<table>
<thead>
<tr>
<th>UPC</th>
<th>UPC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of threads</td>
<td>THREADS</td>
</tr>
<tr>
<td>My ID</td>
<td>MYTHREAD</td>
</tr>
<tr>
<td>Shared variable</td>
<td>shared Type s</td>
</tr>
<tr>
<td>Shared array</td>
<td>shared [BS] Type A[sz]</td>
</tr>
<tr>
<td>Pointer-to-shared</td>
<td>shared Type *pts</td>
</tr>
<tr>
<td>Dynamic memory allocation</td>
<td>upc_alloc(nbytes)</td>
</tr>
<tr>
<td>Bulk data transfer</td>
<td>upc_memcpy(dst, src, nbytes);</td>
</tr>
<tr>
<td>Affinity query</td>
<td>upc_threadof(ptr)</td>
</tr>
<tr>
<td>Synchronization</td>
<td>upc_lock</td>
</tr>
<tr>
<td></td>
<td>upc_barrier</td>
</tr>
</tbody>
</table>

Homework: how to translate upc\_forall?

Asynchronous Task Execution

- C++ 11 async function
  \[ \text{std}::\text{future}<\text{T}> \text{ handle} \]
  \[ = \text{std}::\text{async}(\text{Function>& f, \text{Args}&&... \text{ args}); \text{ handle}.\text{wait}(); \]

- UPC++ async function
  // Remote Procedure Call
  \[ \text{upcxx}::\text{async}\_call}(\text{Function} f, \text{T1 arg1}, \text{T2 arg2},...); \text{ upcxx}::\text{wait}(); \]

  // Explicit task synchronization
  \[ \text{upcxx}::\text{event} e; \text{ upcxx}::\text{async}\_call}(e, \text{Function} f, \text{T1 arg1}, ...); \text{ e}.\text{wait}(); \]
Async Task Example

```c
#include <upcxx.h>
#include <forkjoin.h> // using the fork-join execution model

void print_num(int num) {
  printf("myid %u, arg: %d\n", MYTHREAD, num);
}

int main(int argc, char **argv) {
  upcxx::range tg(1, THREADS, 2); // threads 1,3,5,...
  // call a function on a group of remote processes
  upcxx::async(tg)(print_num, 123);
  upcxx::wait(); // wait for the remote tasks to complete
  return 0;
}
```

Async with Lambda Function

```c
// Thread 0 spawns async tasks
for (int i = 0; i < THREADS; i++) {
  // spawn a task at place “i”
  // the task is expressed by a lambda (anonymous) function
  upcxx::async(i)[](int num) { printf("num: %d\n", num); },
  1000+i); // argument to the A function
  upcxx::wait(); // wait for all tasks to finish
}

mpirun -n 4 ./test_async
```

Output:
```
num: 1000
num: 1001
num: 1002
num: 1003
```

X10-style Finish-Async Programming Idiom

```c
using namespace upcxx;

// Thread 0 spawns async tasks
finish {
  for (int i = 0; i < THREADS; i++) {
    async(i)[](int num) {
      printf("num: %d\n", num); },
      1000+i);
  }
} // All async tasks are completed
```

How We Did It?

```c
// finish { => macro expansion =>
  for (f_scope _fs; _fs.done == 0; _fs.done = 1) {
    // f_scope constructor call generated by compiler
    // push the current scope in a stack
    f_scope() { push_event(&_fs.e); }

    for (int i = 0; i < THREADS; i++) {
      // register the async with the current scope
      async(i, e = peek_event())(...);
    }
  }
  // f_scope destructor call generated by compiler
  ~f_scope() { pop_event(); __fs.e.wait(); }
} // All registered tasks are waited for completion

Leverage C++ Programming Idiom Resource
Acquisition Is Initialization (RAII)
```
### Random Access Benchmark (GUPS)

```c
// shared uint64_t Table[TableSize]; in UPC
shared_array<uint64_t> Table(TableSize);

void RandomAccessUpdate()
{
    uint64_t ran, i;
    ran = starts(NUPDATE / THREADS * MYTHREAD);
    for(i = MYTHREAD; i < NUPDATE; i += THREADS)
    {
        ran = (ran << 1) ^ ((int64_t)ran < 0 ? POLY : 0);
        Table[ran & (TableSize-1)] ^= ran;
    }
}
```

### Manycore - A Good Fit for PGAS

**Logical data layout**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 2</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 3</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Physical data layout**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| Thread 0 | 0 | 4 | 8 | 12 |
| Thread 1 | 1 | 5 | 9 | 13 |
| Thread 2 | 2 | 6 | 10 | 14 |
| Thread 3 | 3 | 7 | 11 | 15 |

### GUPS Performance on MIC

**Random Access Latency**

- UPC++ vs. UPC

**Giga Updates Per Second**

- UPC++ vs. UPC

Difference between UPC++ and UPC is only about 0.2 µs (~220 cycles)

### GUPS Performance on BlueGene/Q

**Random Access Latency**

- UPC++ vs. UPC

**Giga Updates Per Second**

- UPC++ vs. UPC

Difference is negligible at large scale
**UPC++ Application: Embree**

- Low resolution
- High resolution
  - Intel open-source ray tracing toolkit written in C++
  - Ported to UPC++ by Michael Driscoll
  - Performance scaled on Edison (Cray XC30)

**Embree Performance on Edison**

Hybrid UPC++ for internode communication and OpenMP within a NUMA node

**LULESH Proxy Application**

- Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
- Proxy App for UHPC, ExMatEx, and LLNL ASC
- Written in C++ with MPI, OpenMP, and CUDA versions

https://codesign.llnl.gov/lulesh.php

**LULESH 3-D Data Partitioning**
**LULESH Communication Pattern**

Cross-section view of the 3-D processor grid

- 26 neighbors
  - 6 faces
  - 12 edges
  - 8 corners

**Data Layout of Each Partition**

- 3-D array $A[x,y,z]$
- Row-major storage
- $z$ index goes the fastest

- Blue planes are contiguous
- Green planes are stride-$N^2$ chunks
- Red planes are stride-$N$ elements

---

**Convert MPI to UPC++**

Pseudo code

```c
// Post Non-blocking Recv
MPI_Irecv(RecvBuf1);
... MPI_Irecv(RecvBufN);
Pack_Data_to_Buf();
// Get neighbors' RecvBuf addresses
// Post Non-blocking Copy
upcxx::async_copy(SendBuf1, RecvBuf1);
... upcxx::async_copy(SendBufN, RecvBufN);
MPI_Wait();
async_copy_fence();
Unpack_Data();
```

---

**LULESH Performance on Cray XC30 (Edison)**

Take advantage of PGAS without the pain of adopting a new language
Problem 2: Combining Data Sets

- Merge measurement data into simulation and evaluate fit
- Matrix is too large for single shared memory
- Assembly: Strided writes into a global array
- Goal is scalability in context of full code

Summary of UPC++

- Minimally Invasive Technique for providing PGAS features to existing C/C++ apps
- UPC++ draws ideas from existing PGAS languages: UPC, Phalanx, Titanium and X10
- Source code available on Bitbucket
  https://bitbucket.org/upcxx/upcxx
- Questions and comments:
  upcxx@googlegroups.com

A Family of PGAS Languages

- UPC based on C philosophy / history
  - http://upc-lang.org
  - Free open source compiler: http://upc.lbl.gov
  - Also a gcc variant: http://www.gccupc.org
- Java dialect: Titanium
  - http://Titanium.cs.berkeley.edu
- Co-Array Fortran
  - Part of Stanford Fortran (subset of features)
  - CAF 2.0 from Rice: http://caf.rice.edu
- Chapel from Cray (own base language better than Java)
  - http://chapel.cray.com (open source)
- X10 from IBM also at Rice (Java, Scala,…)
- Phalanx from Echelon projects at NVIDIA, LBNL,…
  - C++ PGAS languages with CUDA-like features for GPU clusters
- Coming soon…. PGAS for Python, aka PyGAS

Productivity of the Titanium Language

- Titanium is a PGAS language based on Java
- Line count comparison of Titanium and other languages:

  **NAS Parallel Benchmarks**

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>NPB-CG</th>
<th>NPB-FT</th>
<th>NPB-MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>2000</td>
<td>1500</td>
<td>1000</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>1500</td>
<td>1000</td>
<td>500</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>1000</td>
<td>500</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AMR Chombo</th>
<th>C++/Fortran/MPI</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR data structures</td>
<td>35000</td>
<td>2000</td>
</tr>
<tr>
<td>AMR operations</td>
<td>6500</td>
<td>12000</td>
</tr>
<tr>
<td>Elliptic PDE Solver</td>
<td>4200*</td>
<td>1500</td>
</tr>
</tbody>
</table>

* Somewhat more functionality in PDE part of C++/Fortran code
## Productive Features in Titanium

- UPC++ already provides many of Titanium’s productivity features
  - Basic high-level language features (e.g. object orientation, memory management)
  - Templates and operator overloading
  - SPMD execution model and PGAS memory model
- Titanium features we want to implement in UPC++
  - True multidimensional rectangular arrays
    - Not distributed, but may be located on a remote thread
  - Hierarchical teams
  - Global object model (future work)

## C and UPC Arrays

- C/C++ arrays are limited in many ways
  - Multidimensional arrays must specify sizes of all but first dimension as compile-time constants
    - These sizes are part of the type, which makes it hard to write generic code
  - Easy to get view of contiguous subset of an array, but non-contiguous view must be handled manually
- UPC shared arrays have their own limitations
  - Can only be distributed in one dimension
    - User must manually linearize a multidimensional array, use a directory structure, or both
  - Blocking factor must be a compile-time constant
    - `upc_memcpy` only supports contiguous source and destination

## Example: Ghost Zones

- Copying ghost zones requires manually packing/unpacking elements at source/destination
  - In effect, turns one-sided operation into two-sided
- Strided copy is not enough for ghost cell thickness > 1
  - Need “side factors” to specify how many elements to skip at end of each dimension

## Multidimensional Arrays in Titanium

- True multidimensional arrays
  - Supports subarrays without copies
    - Can refer to rows, columns, slabs, interior, boundary, etc.
  - Indexed by Points (tuples of ints)
  - Built on a rectangular set of Points, RectDomain
  - Points and RectDomains are built-in immutable classes, with useful literal syntax
- Support for AMR and other grid computations
  - Domain operations: intersection, shrink, border
- Arrays are located on a single thread, but can be a remote thread
Points, RectDomains, Arrays in General

- Points specified by a tuple of ints
  \[
  \text{Point}<2> \ lb = [1, 1]; \\
  \text{Point}<2> \ ub = [10, 20];
  \]
- RectDomains given by 3 points:
  - lower bound, upper bound (and optional stride)
  \[
  \text{RectDomain}<2> \ r = [\text{lb} : \text{ub}];
  \]
- Array declared by number of dimensions and type
  \[
  \text{double} [2d] \ a;
  \]
- Array created by passing RectDomain
  \[
  a = \text{new} \ \text{double} [r];
  \]

Unordered Iteration

- Motivation:
  - Memory hierarchy optimizations are essential
  - Compilers sometimes do these, but hard in general
- Titanium has explicitly unordered iteration
  - Helps the compiler with analysis
  - Helps programmer avoid indexing details
- \[
  \text{foreach} (p \ \text{in} \ r) \ { \ \ldots \ A[p] \ \ldots \ }
  \]
  - \(p\) is a Point (tuple of ints), can be used as array index
  - \(r\) is a RectDomain
- Note: \textit{foreach} is not a parallelism construct

Simple Array Example

- Matrix sum in Titanium
  \[
  \text{Point}<2> \ lb = [1,1]; \\
  \text{Point}<2> \ ub = [10,20]; \\
  \text{RectDomain}<2> \ r = [\text{lb} : \text{ub}]; \\
  \text{double} [2d] \ a = \text{new} \ \text{double} [r]; \\
  \text{double} [2d] \ b = \text{new} \ \text{double} [1:10,1:20]; \\
  \text{double} [2d] \ c = \text{new} \ \text{double} [1b:ub:[1,1]]; \\
  \text{for} (\text{int} \ i = 1; \ i <= 10; \ i++) \\
    \text{for} (\text{int} \ j = 1; \ j <= 20; \ j++) \\
      c[i,j] = a[i,j] + b[i,j];
  \]
  \[
  \text{foreach} (p \ \text{in} \ c.\text{domain}()) \ { \ c[p] = a[p] + b[p]; \ }
  \]

More Array Operations

- Titanium arrays have a rich set of operations
  - \textit{translate}, \textit{restrict}, \textit{slice (n dim to n-1)}
  - None of these modify the original array, they just create another view of the data in that array
  - Most important array operation: one line copy between any two arrays with same element type and arity
    \[
    \text{dst.copy(src)}
    \]
    - Copies all elements in intersection of source and destination domains
    - Both source and destination can be located on any thread
**Example: Setting Boundary Conditions**

```cpp
foreach (l in local_grids.domain()) {
    foreach (a in all_grids.domain()) {
        local_grids[l].copy(all_grids[a]);
    }
}
```

- Can allocate arrays in a global index space
- Let compiler compute intersections

**Implementation of Titanium Arrays in UPC++**

- UPC++ implementation built using C++ templates and operator overloading
  - Template parameters specify arity and element type
  - Overload element access operator []
- Macros provide simple syntax for domain/array literals
  - Titanium
    ```cpp
    RectDomain<3> rd = [[1, 1, 1] : [3, 3, 3]];
    int[3d] local_arr = new int[[1, 1, 1] : [3, 3, 3]];
    ```
  - UPC++
    ```cpp
    POINT(1, 3) rectdomain<3> rd = RECTDOMAIN((1, 1, 1), (3, 3, 3));
    ndarray<int, 3> arr = ARRAY(int, ((1, 1, 1), (3, 3, 3)));
    ```

**Foreach Implementation**

- Macros also allow definition of `foreach` loops

```cpp
#define foreach(p, dom) \    
    foreach_(p, dom, UNIQUIFY(FOREACH_PTR_, p))
#define foreach_(p, dom, ptr_) \    
    for (auto ptr_ = (dom).iter(); !ptr_.done; \    
        ptr_.done = true) \    
        for (auto p = ptr_.start(); ptr_.next(p));
```

**Preliminary Results**

- Currently have full implementation of Titanium-style domains and arrays in UPC++
- Additionally have ported useful pieces of the Titanium library to UPC++
  - e.g. timers, higher-level collective operations
- Four kernels ported from Titanium to UPC++
  - 3D 7-point stencil, NAS conjugate gradient, Fourier transform, and multigrid
  - Minimal porting effort for these examples
    - Less than a day for each kernel
    - Array code only requires change in syntax
    - Most time spent porting Java features to C++
  - Larger applications will require global object model to be defined and implemented in UPC++
Performance Tuning

- Since UPC++ is a library, cannot rely on compiler to optimize array accesses
  - Array library is very general, but generality results in overhead in simple cases
- Preliminary approach is to provide template specializations that allow users to bypass inefficient, general code
- In the future, we plan to explore automatic dynamic specialization
  - Potentially leverage SEJITS work at UCB

Example: CG SPMV

- Unspecialized local SPMV in conjugate gradient kernel
  ```cpp
  void multiply(ndarray<double, 1> output, 
  ndarray<double, 1> input) {
    double sum = 0;
    foreach(i, lrowRectDomains.domain()) {
      sum = 0;
      foreach(j, lrowRectDomains[i]) {
        sum += la[j] * input[lcolidx[j]];
      }
      output[i] = sum;
    }
  }
  ```
  - 3x slower than hand-tuned code (sequential PGCC on Cray XE6)

- Specialized local SPMV
  ```cpp
  void multiply(ndarray<double, 1, simple> output, 
  ndarray<double, 1, simple> input) {
    double sum = 0;
    foreach1(i, lrowRectDomains.domain()) {
      sum = 0;
      foreach1(j, lrowRectDomains[i]) {
        sum += la[j] * input[lcolidx[j]];
      }
      output[i] = sum;
    }
  }
  ```
  - Comparable to hand-tuned code (sequential PGCC on Cray XE6)

Hierarchical Programming

- Applications can reduce communication costs by adapting to machine hierarchy
- Applications may also have inherent, algorithmic hierarchy
  - Recursive algorithms
  - Composition of multiple algorithms
  - Hierarchical division of data
Algorithm Example: Merge Sort

• Task parallel
  ```java
  int[] mergeSort(int[] data) {
    int len = data.length;
    if (len < threshold)
      return sequentialSort(data);
    d1 = fork mergeSort(data[0:len/2-1]);
    d2 = mergeSort(data[len/2:len-1]);
    join d1;
    return merge(d1, d2);
  }
  ```
  • Cannot fork threads in SPMD
    – Must rewrite to execute over fixed set of threads

Hierarchical Teams

• Thread teams are basic units of cooperation
  – Groups of threads that cooperatively execute code
  – Collective operations over teams
• Structured, hierarchical teams provide many benefits over flat teams
  – Expressive: match structure of algorithms, machines
  – Safe: eliminate many sources of deadlock
  – Composable: enable existing code to be composed without being rewritten to explicitly use teams
  – Efficient: allow users to take advantage of machine structure, resulting in performance gains

Team Data Structure

• Threads comprise teams in tree-like structure
• First-class object to allow easy creation and manipulation

Work in progress: add ability to automatically construct team hierarchy from machine structure
Team Usage Construct

- Syntactic construct specifies that all enclosed operations are with respect to the given team
- Collectives and constants such as MYTHREAD are with respect to currently scoped team

```cpp
teamsplit(row_team) {
  Reduce::add(mtmp, myresults, rpivot);
}
```

Team Construct Implementation

- `teamsplit` implemented exactly like `finish`

```cpp
// teamsplit(row_team) { => macro expansion =>
for (ts_scope _ts(row_team); _ts.done == 0;
    _ts.done = 1) {
  // ts_scope constructor call generated by compiler
  // descend one level in team hierarchy
ts_scope(team &t) { descend_team(t->mychild()); }
  // collective operation on current team
  Reduce::add(mtmp, myresults, rpivot);

  // ts_scope destructor call generated by compiler
  ~ts_scope() { ascend_team(); }
}
```

Merge Sort Team Hierarchy

- Team hierarchy is binary tree
- Trivial construction

```cpp
void divide_team(team &t) {
  if (THREADS > 1) {
    t.split(MYTHREAD % 2, MYTHREAD / 2);
    teamsplit(t) {
      divide_team(t.mychild());
    }
  }
}
```

Merge Sort Implementation

- Control logic for sorting and merging

```cpp
void sort_and_merge(team &t) {
  if (THREADS == 1) {
    allres[myidx] = sequential_sort(mydata);
  } else {
    teamsplit(t) {
      sort_and_merge(t.mychild());
    }
    barrier();
  }
}
```

Leverage C++ Programming Idiom

Resource Acquisition Is Initialization (RAII)

- Leverage C++ Programming Idiom
- Resource Acquisition Is Initialization (RAII)
Hierarchical Teams Results (Titanium)

- Titanium has full hierarchical team implementation, including machine model
- Hierarchical sort algorithm has both algorithmic hierarchy (merge sort) and machine-level hierarchy (mixed sample sort and merge sort)

![Distributed Sort (Cray XE6)]

Summary

- Many productive language features can be implemented in C++ without modifying the compiler
  - Macros and template metaprogramming provide a lot of power for extending the core language
- Many Titanium applications can be ported to UPC++ with little effort
  - UPC++ can provide the same productivity gains as Titanium
- However, analysis and optimization still an open question
  - Can we build a lightweight standalone analyzer/optimizer for UPC++?
  - Can we provide automatic specialization at runtime in C++?

Future Work

- Arrays
  - Investigate dynamic optimization using just-in-time specialization
  - Design and build distributed array library on top of current library
- Hierarchical teams
  - Design hierarchical machine model for UPC++
  - Add ability to query machine structure at runtime
- Global object model
  - Explore template metaprogramming techniques for implementing a global object interface
  - Build a tool for generating global analogs from local class definitions

Application Work in PGAS

- Network simulator in UPC (Steve Hofmeyr, LBNL)
- Real-space multigrid (RMG) quantum mechanics (Shirley Moore, UTK)
- Landscape analysis, i.e., "Contributing Area Estimation" in UPC (Brian Kazian, UCB)
- GTS Shifter in CAF (Preissl, Wichmann, Long, Shalf, Ethier, Koniges, LBNL, Cray, PPPL)
Two Distinct Parallel Programming Questions

- What is the parallel control model?
  - SPMD “default” plus data parallelism through collectives and dynamic tasking within nodes or between nodes through libraries
    - Data parallel, dynamic, single program

- What is the model for sharing/communication?
  - Synchronization may be coupled (implicit) or separate (explicit)
  - Data parallel (single thread of control)
  - Dynamic threads
  - Single program, multiple data (SPMD)

PyGAS: Combine two popular ideas

- Python
  - No. 6 Popular on http://langpop.com and extensive libraries, e.g., Numpy, Scipy, Matplotlib, NetworkX
  - 10% of NERSC projects use Python
- PGAS
  - Convenient data and object sharing
  - PyGAS: Objects can be shared via Proxies with operations intercepted and dispatched over the network:
    - num = 1+2*j
    - print pxy.real # shared read
    - print pxy.imag = 3 # shared write
    - print pxy.conjugate() # invoke
  - Leveraging duck typing:
    - Proxies behave like original objects.
    - Many libraries will automatically work.

Arrays in a Global Address Space

- Key features of Titanium arrays
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to identify ghosts and iterate:
  - foreach (p in gridA.shrink(1).domain()) ...
- Array copies automatically work on intersection
  - gridB.copy(gridA.shrink(1));

Languages Support Helps Productivity

C++/Fortran/MPI AMR
- Chombo package from LBNL
  - Bulk-synchronous comm.
  - Pack boundary data between procs
  - All optimizations done by programmer
- Titanium AMR
  - Entirely in Titanium
  - Finer-grained communication
    - No explicit pack/unpack code
    - Automated in runtime system
    - General approach
      - Language aware programmer optimizations
      - Compiler/runtime does some automatically

Work by Tong Wen and Philip Colella; Communication optimizations joint with Jimmy Su
Particle/Mesh Method: Heart Simulation

- Elastic structures in an incompressible fluid.
  - Blood flow, clotting, inner ear, embryo growth, ...
- Complicated parallelization
  - Particle/Mesh method but “Particles” connected into materials (1D or 2D structures)
  - Communication patterns irregular between particles (structures) and mesh (fluid)

2D Dirac Delta Function

Code Size in Lines

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8000</td>
<td>4000</td>
</tr>
</tbody>
</table>

Note: Fortran code is not parallel

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen

Compiler-free “UPC++” eases interoperability

- Global array
  - L-value reference (write/put)
    \[ A[1] = 1; \]
  - R-value reference (read/get)
    \[ n = A[1] + 1; \]

Hierarchical SPMD (demonstrated in Titanium)

- Thread teams may execute distinct tasks
  - 
  

- Hierarchy for machine / tasks
  - Nearby: access shared data
  - Far away: copy data
- Advantages:
  - Provable pointer types
  - Mixed data / task style
  - Lexical scope prevents some deadlocks
Hierarchical machines → Hierarchical programs

- Hierarchical memory model may be necessary (what to expose vs hide)
- Two approaches to supporting the hierarchical control

- Option 1: Dynamic parallelism creation
  - Recursively divide until… you run out of work (or hardware)
  - Runtime needs to match parallelism to hardware hierarchy
- Option 2: Hierarchical SPMD with “Mix-ins”
  - Hardware threads can be grouped into units hierarchically
  - Add dynamic parallelism with voluntary tasking on a group
  - Add data parallelism with collectives on a group

Option 1 spreads threads, option 2 collect them together

Vertical PGAS

- New type of wide pointer?
  - Points to slow (off-chip memory)
  - The type system could get unwieldy quickly

One-sided communication works everywhere

PGAS programming model

```c
*p1 = *p2 + 1;
A[i] = B[i];
upc_memput(A, B, 64);
```

It is implemented using one-sided communication: put/get

Support for one-sided communication (DMA) appears in:
- Fast one-sided network communication (RDMA, Remote DMA)
- Move data to/from accelerators
- Move data to/from I/O system (Flash, disks,..)
- Movement of data in/out of local-store (scratchpad) memory

HPC: From Vector Supercomputers to Massively Parallel Systems

- Programmed by “annotating” serial programs
- Programmed by completely rethinking algorithms and software for parallelism
- 25% industrial use
- 50% industrial use

New type of wide pointer?
- Points to slow (off-chip memory)
- The type system could get unwieldy quickly
**PGAS Languages**

- **Global address space**: thread may directly read/write remote data
- Hides the distinction between shared/distributed memory
- **Partitioned**: data is designated as local or global
- Does not hide this: critical for locality and scaling

**A Brief History of Languages**

- **When vector machines were king**
  - Parallel "languages" were loop annotations (IVDEP)
  - Performance was fragile, but there was good user support

- **When SIMD machines were king**
  - Data parallel languages popular and successful (CMF, "Lisp, C*, …)
  - Quite powerful: can handle irregular data (sparse mat-vec multiply)
  - Irregular computation is less clear (multi-physics, adaptive meshes, backtracking search, sparse matrix factorization)

- **When shared memory multiprocessors (SMPs) were king**
  - Shared memory models, e.g., OpenMP, POSIX Threads, were popular

- **When clusters took over**
  - Message Passing (MPI) became dominant
- With multicore building blocks for clusters
  - Mixed MPI + OpenMP is the preferred choice