Unified Parallel C (UPC)

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Idealized Parallel Architectures

Programming Models

- Cilk
- OpenMP
- Pthreads

- MPI
- UPC
- CAF

Key:
- Process/Thread
- Memory
Idealized Parallel Architectures of Today

Hybrid Shared + Distributed Memory

Programming Models

e.g., MPI + OpenMP
PGAS models
Performance Concerns for Distributed Memory

Data movement and synchronization are expensive

To minimize overheads

• Co-locate data with processes
• Aggregate multiple accesses to remote data
• Overlap communication with computation

Distributed Memory
Partitioned Global Address Space Languages

• Global address space
  — one-sided communication (GET/PUT)  simpler than msg passing

• Programmer has control over performance-critical factors
  — data distribution and locality control
  — computation partitioning
  — communication placement

• Data movement and synchronization as language primitives
  — amenable to compiler-based communication optimization
Partitioned Global Address Space Languages

- Unified Parallel C (C) http://upc.wikinet.org
- Titanium (Java) http://titanium.cs.berkeley.edu
- Coarray Fortran 2.0 (Fortran) http://caf.rice.edu

- Related efforts: HPCS Languages
  - X10 (http://x10-lang.org)
  - Chapel (http://chapel.cray.com)
  - Fortress (https://projectfortress.java.net)
Unified Parallel C (UPC)

• An explicit parallel extension of the C language
  — a few extra keywords
    – shared, MYTHREAD, THREADS, upc_forall

• Language features
  — partitioned global address space for shared data
    – part of shared data co-located with each thread
  — threads created at application launch
    – each bound to a hardware thread / core
    – each has some private data
  — a memory model
    – defines semantics of interleaved accesses to shared data
  — synchronization primitives
    – barriers
    – locks
    – load/store
UPC Execution Model

- Multiple threads work independently in a SPMD fashion
  - MYTHREAD specifies thread index (0..THREADS-1)
  - # threads specified at compile-time or program launch

- Address Space

- Threads synchronize as necessary using using
  - synchronization primitives
  - shared variables
Shared and Private Data

- Static and dynamic memory allocation of each type of data
- Shared objects placed in memory based on affinity
  - shared scalars have affinity to thread 0
    - here, a scalar means a singleton instance of any type
  - elements of shared arrays are allocated round robin among memory modules co-located with each thread
A One-dimensional Shared Array

Consider the following data layout directive

```c
shared int y[2 * THREADS + 1];
```

For THREADS = 3, we get the following layout:

Thread 0
- y[0]
- y[3]
- y[6]

Thread 1
- y[1]
- y[4]

Thread 2
- y[2]
- y[5]
A Multi-dimensional Shared Array

```c
shared int A[4][THREADS];
```

For THREADS = 3, we get the following layout

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td>A[0][2]</td>
</tr>
</tbody>
</table>
Consider the following data layout directives

```c
shared int x; // x has affinity to thread 0
shared int y[THREADS];
int z; // private
```

For THREADS = 3, we get the following layout

- **Thread 0**
  - `x` (shared)
  - `y[0]`
  - `z`

- **Thread 1**
  - `y[1]` (private)
  - `z`

- **Thread 2**
  - `y[2]`
  - `z`
Controlling the Layout of Shared Arrays

- Can specify a blocking factor for shared arrays
  - default block size is 1 element
- Shared arrays are distributed on a block per thread basis, round robin allocation of block size chunks
- Example layout using block size specifications
  - e.g., shared [2] int a[16]

```
Thread 0
a[0]
a[1]
a[6]
a[7]
a[12]
a[13]

Thread 1
a[2]
a[3]
a[8]
a[9]
a[14]
a[15]

Thread 2
a[4]
a[5]
a[10]
a[11]
```
Blocking of Shared Arrays

- Block size and THREADS determine *affinity*—with which thread will a datum be co-located
- Element $i$ of a blocked array has affinity to thread:

$$\left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \mod \text{THREADS}$$
Blocking Multi-dimensional Data I

- Manage the interaction between
  - contiguous memory layout of C multi-dimensional arrays
  - blocking factor for shared layout

- Consider layouts for different block sizes for
  - shared [BLOCKSIZE] double grids[N][N];

For the case where $N = K \times \text{THREADS}$:

- Default
  - BLOCKSIZE=1

- Column Blocks
  - BLOCKSIZE=N/THREADS

- Distribution by Row
  - BLOCKSIZE=N
Consider the data declaration

\[
\text{shared } [3] \text{ int A}[4][\text{THREADS}];
\]

When THREADS = 4, this results in the following data layout

The mapping is not pretty when the rightmost dimensions aren’t a multiple of THREADS
A Simple UPC Program: Vector Addition

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

shared int v1[N], v2[N], v1plusv2[N];

void main() {
  int i;
  for(i=0; i<N; i++)
    if (MYTHREAD == i % THREADS)
      v1plusv2[i] = v1[i] + v2[i];
}

Each thread executes each iteration to check if it has work
A More Efficient Vector Addition

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

shared int v1[N], v2[N], v1plusv2[N];

void main() {
    int i;
    for(i = MYTHREAD; i < N; i += THREADS) {
        v1plusv2[i]=v1[i]+v2[i];
    }
}

Iteration #:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

| v1[0] | v1[1] |
|       |       |

| v2[0] | v2[1] |
|       |       |

| v1plusv2[0] | v1plusv2[1] |

Each thread executes only its own iterations
Worksharing with **upc**\_**forall**

- Distributes independent iterations across threads
- Simple C-like syntax and semantics
  - `upc_forall(init; test; loop; affinity)`
- Affinity is used to enable locality control
  - Usually, map iteration to thread where the iteration’s data resides
- Affinity can be
  - An integer expression, or a
  - Reference to (address of) a shared object
Work Sharing + Affinity with `upc_forall`

- **Example 1:** explicit affinity using shared references
  ```c
  shared int a[100], b[100], c[100];
  int i;
  upc_forall (i=0; i<100; i++; &a[i])
    a[i] = b[i] * c[i];
  ```

- **Example 2:** implicit affinity with integer expressions
  ```c
  shared int a[100], b[100], c[100];
  int i;
  upc_forall (i=0; i<100; i++; i)
    a[i] = b[i] * c[i];
  ```

**Note:** both yield a round-robin distribution of iterations
Vector Addition Using `upc forall`

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

shared int v1[N], v2[N], v1plusv2[N];

void main()
{
    int i;
    upc forall (i = 0; i < N; i++; i)
        v1plusv2[i]=v1[i]+v2[i];
}
```

Each thread executes subset of global iteration space as directed by affinity clause.
• Example 3: implicit affinity by chunks

```c
shared int a[100], b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
  a[i] = b[i] * c[i];
```

• Assuming 4 threads, the following results

<table>
<thead>
<tr>
<th>i</th>
<th>i*THREADS</th>
<th>i*THREADS/100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.24</td>
<td>0.96</td>
<td>0</td>
</tr>
<tr>
<td>25.49</td>
<td>100.196</td>
<td>1</td>
</tr>
<tr>
<td>50.74</td>
<td>200.296</td>
<td>2</td>
</tr>
<tr>
<td>75.99</td>
<td>300.396</td>
<td>3</td>
</tr>
</tbody>
</table>
// vect_mat_mult.c
#include <upc_relaxed.h>

shared int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];
void main (void) {
    int i, j;
    upc_forall(i = 0; i < THREADS; i++; i) {
        c[i] = 0;
        for (j = 0; j < THREADS; j++)
            c[i] += a[i][j]*b[j];
    }
}
Matrix-Vector Multiply (Better Distribution)

// vect_mat_mult.c
#include <upc_relaxed.h>

shared[THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];
void main (void) {
    int i, j;
    upc_forall(i = 0 ; i < THREADS ; i++; i) {
        c[i] = 0;
        for (j = 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
UPC Pointers

- Needed for expressive data structures
- Flavors
  - private pointers pointing to local
    - int *p1
  - private pointers pointing to shared
    - shared int *p2
  - shared pointers pointing to local
    - int *shared p3
  - shared pointers pointing to shared
    - shared int *shared p4
UPC Pointer Implementation Requirements

- Handle shared data
- Support pointer arithmetic
- Support pointer casting
UPC Pointer Representation

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

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

Example: Cray T3E 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>48</td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>0</td>
</tr>
</tbody>
</table>
UPC Pointer Features

• 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 private pointer, the thread # of pointer-to-shared may be lost

• Casting of a pointer-to-shared to a private pointer
  —well defined only if the target object has affinity to local thread
Dynamic Memory Allocation of Shared Memory

- Dynamic memory allocation of shared memory is available
- Functions can be collective or not
- Collective function
  — called by every thread
  — returns the same value to each of them
- Collective function names typically include “all”
Global Allocation of Shared Memory

• Collective allocation function: must be called by all threads
  —shared void *upc_all_alloc(size_t nblocks, size_t nbytes)
    nblocks: number of blocks
    nbytes: block size
  —each thread gets the same pointer
  —equivalent to:
    shared [nbytes] char[nblocks * nbytes]

• Non-collective version that yields the same layout
  —shared void *upc_global_alloc(size_t nblocks, size_t nbytes)
    nblocks: number of blocks
    nbytes: block size
shared [N] int *ptr;
ptr = (shared [N] int *)
upc_all_alloc( THREADS, N*sizeof( int ) );
Local-Shared Memory Allocation

`shared void *upc_alloc (size_t nbytes)`

—`nbytes`: block size

- Non collective; called by one thread
- Calling thread allocates a contiguous memory region in its local shared space
- Space allocated per calling thread is equivalent to `shared [] char[nbytes]`
- If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer
Synchronization

• No implicit synchronization among the threads

• UPC provides the following synchronization mechanisms:
  — barriers
  — locks
Synchronization - Barriers

• Barriers (blocking)
  — upc_barrier expr_opt;

• Split-phase barriers (non-blocking)
  — upc_notify expr_opt;
  — upc_wait expr_opt;
  – note: upc_notify is not blocking upc_wait is
• Lock primitives
  —void upc_lock(upc_lock_t *l)
  —int upc_lock_attempt(upc_lock_t *l) // success returns 1
  —void upc_unlock(upc_lock_t *l)

• Locks are allocated dynamically, and can be freed
• Locks are properly initialized after they are allocated
Dynamic Lock Allocation

• Collective lock allocation (à la upc_all_alloc)
  — upc_lock_t * upc_all_lock_alloc(void);

• Global lock allocation (à la upc_global_alloc)
  — upc_lock_t * upc_global_lock_alloc(void);

• Lock deallocation
  — void upc_lock_free(upc_lock_t *ptr);
Memory Consistency Model

- Dictates the ordering of shared operations
  - when a change to a shared object by a thread becomes visible to others

- Consistency can be strict or relaxed

- Relaxed consistency model
  - compiler & runtime can reorder accesses to shared data

- Strict consistency model
  - enforce sequential ordering of operations on shared data
    - no operation on shared can begin before previous ones are done
    - changes become visible immediately
Memory Consistency

- Default behavior can be altered for a variable definition in the declaration using:
  - Type qualifiers: strict & relaxed

- Default behavior can be altered for a statement or a block of statements using
  - #pragma upc strict
  - #pragma upc relaxed

- Precedence order for memory consistency specifications
  1. declarations
  2. pragmas
  3. program level
strict shared int flag_ready = 0;
shared int result0, result1;

if (MYTHREAD==0){
    results0 = expression1;
    flagReady=1; //if not strict, it could be
    // switched with the above statement
} else if (MYTHREAD==1){
    while(!flag_ready); //Same note
    result1=expression2+results0;
}

- Could have used a barrier between the first and second statement in the if and the else code blocks
  - expensive: affects all operations at all threads
- Above works as an example of point to point synchronization
Forcing Memory Consistency via **upc_fence**

- **What is a memory fence?**
  - All memory operations initiated before a fence operation must complete before the fence completes

- **UPC provides a fence construct**
  - **Syntax**
    - `upc_fence;`
  - **Semantics**
    - Equivalent to a null strict reference
Library Operations for Bulk Data

- No flexible way to initiate bulk transfer operations in UPC
- Rely on library operations for bulk data transfer and set
  - void upc_memcpy(shared void * restrict dst, shared const void * restrict src, size_t n)
  - void upc_memget(void * restrict dst, shared const void * restrict src, size_t n);
  - void upc_memput(shared void * restrict dst, const void * restrict src, size_t n);
  - void upc_memset(shared void * dst, int c, size_t n);
Explicit Non-blocking Data Movement

• Get, Put, Copy
  
  — upc_handle_t bupc_memcpy_async(shared void *dst, shared const void *src, size_t nbytes)
  
  — upc_handle_t bupc_memget_async(void *dst, shared const void *src, size_t nbytes)
  
  — upc_handle_t bupc_memput_async(shared void *dst, const void *src, size_t nbytes)
    
    – same args and semantics as blocking variants
    
    – upc_handle_t: opaque handle representing the operation initiated

• Synchronize using one of two new functions
  
  – void bupc_waitsync(upc_handle_t handle)
    
    – blocking test for completion

  – int bupc_trysync(upc_handle_t handle)
    
    – non-blocking test for completion
• Chop reads into k-mers that overlap by k-1
• Store k-mers in distributed hash table.
  \((\text{key}, \text{value}) = (\text{k-mer}, \text{2-char fwd/bwd extension [AGCT] [AGCT]})\)
• From selected k-mers, perform forward and reverse traversals to construct contigs

Fig. 2: Meraculous assembly flow chart.

Fig. 5: Parallel de Bruijn graph construction.

Ongoing Work in UPC

• Interoperability support for multicore
  — UPC always has run on clusters of shared memory
  — Desire to run UPC on 1-n nodes for small n, within MPI

• Adding hierarchy to memory (socket, node, cab?,...)

• Use unique network features (atomics, DMA, etc.)

• PGAS/UPC for GPUs (P = CPU/GPU partition + nodes)
  — First step: GASnet on CPU clusters
  — Can put/get from GPU or CPU to/from GPU or CPU
  — (Illusion is a single address space; ugly on current hw)

• SPMD PGAS + dynamic load balancing
  — Dynamic load balancing is mostly useful for load imbalance in
    the apps, and cost across nodes is high
  — So apps choose to add tasking on top of UPC

• Teams and autotuned collectives in UPC
References

Slides adapted from


Meraculous De Novo Genome Assembler

E. Georganas et al., Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly, SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pp.437-448, Nov. 2014 doi: 10.1109/SC.2014.41
URL: http://www.eecs.berkeley.edu/~egeor/sc14_genome.pdf