Unified Parallel C (UPC)

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Idealized Parallel Architectures

Shared Memory

Distributed Memory

interconnect

Programming Models

Cilk
OpenMP
Pthreads

MPI

Key:

Process/Thread
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
Idealized Parallel Architectures of Today

Hybrid Shared + Distributed Memory

Programming Models

- e.g., MPI + OpenMP
- PGAS models
Partitioned Global Address Space Model

- A global address space that is logically distributed
- A collection of threads operating within the address space
- Each thread has affinity with a portion of the address space
- Each thread has private data as well
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 Models

- Unified Parallel C (C) http://upc.wikinet.org
- Titanium (Java) http://titanium.cs.berkeley.edu
- UPC++ (C++) https://bitbucket.org/upcxx

- 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
  — 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
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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[0]</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>y[1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z</td>
</tr>
</tbody>
</table>
## 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]`

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[12]</td>
<td>a[14]</td>
<td></td>
</tr>
</tbody>
</table>

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

![Data Layout Diagram]

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];
}

Each thread executes only its own iterations
Worksharing with `upc_fforall`

- Distributes independent iterations across threads
- Simple C-like syntax and semantics
  - `upc_fforall(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
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 \texttt{upc\_forall}

\begin{verbatim}
//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];
}
\end{verbatim}

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>
Matrix-Vector Multiply (Default Distribution)

// 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];
    }
}
/ 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];
    }
}
Meraculous De Novo Genome Assembler in UPC

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

Fig. 2: Meraculous assembly flow chart.

Fig. 1: Performance and strong scaling of our de Bruijn graph construction & traversal and k-mer analysis steps on Cray XC30 for the human genome. The top three timing curves are with respect to the first y-axis (left) whereas the parallel efficiency curve is with respect to the second y-axis (right). The x-axis uses a log scale.

Imaging the earth’s interior with seismic waves, supercomputers, and PGAS

Scott French\textsuperscript{1}\textsuperscript{*} and Barbara Romanowicz\textsuperscript{1,2}

\textsuperscript{1} Berkeley Seismological Laboratory, UC Berkeley, Berkeley, CA, USA
\textsuperscript{2} Institut de Physique du Globe de Paris, Paris, France
\textsuperscript{*} Now at: Google Inc.
Whole Mantle Waveform Tomography

- **Objective**: 3D model of material properties (elastic wave speed) throughout the earth’s entire mantle (outer 2890 km)
- **Observations**: seismograms of natural earthquakes (100s)
- **Predictions**: numerical simulations of seismic wave propagation

Scientific results: A whole-mantle model

- **Letter**: Whole-mantle radially anisotropic shear velocity structure from spectral-element waveform tomography
  
  S. W. French\textsuperscript{1,7} and B. A. Romanowicz\textsuperscript{1,2,3}

- **Geophysical Journal International**

- **Broad plumes rooted at the base of the Earth’s mantle beneath major hotspots**
  
  Scott W. French\textsuperscript{†} & Barbara Romanowicz\textsuperscript{1,2,3}

  - The first whole-mantle seismic model based on waveform tomography using numerical wavefield simulations
  - Reveals new details of earth structure not seen in previous models based on approximate forward modeling techniques (especially low shear-velocity structures)
Whole Mantle Tomography: Under the Hood

- UPC++ Distributed matrix abstraction (French et al., IPDPS’15) — excellent for distributed data structures + irregular accesses
- Key to implementing one-sided updates optimized for our use case (+= only, associative / commutative, asynchronous)
- Distributed matrices use block-cyclic format

Weak scaling vs. MPI (Hessian estimation)

- Distributed matrix size fixed (180 GB)
- Dataset size scaled w/ concurrency
  - 64 updates per MPI or UPC++ task + thread team (NUMA domain)

Setup:
- NERSC Edison (Cray XC30)
- GNU Compilers 4.8.2 (-O3)
- Cray MPICH 7.0.3
- Up to 12,288 cores
- Matrix size: 180GB
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>
</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></td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></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 nbblocks, size_t nbytes)
    - nbblocks: 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 nbblocks, size_t nbytes)
    - nbblocks: 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
Synchronization - Locks

- **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
  - declarations
strict shared int flag_ready = 0;
shared int result0, result1;

if (MYTHREAD==0){
    results0 = expression1;
    flag_ready=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
References

Slides adapted from

Book

Meraculous De Novo Genome Assembler