Unified Parallel C (UPC)

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@cs.rice.edu
Acknowledgments

• Supercomputing 2007 tutorial on “Programming using the Partitioned Global Address Space (PGAS) Model” by Tarek El-Ghazawi and Vivek Sarkar
Programming Models

Message Passing

Shared Memory

DSM/PGAS

MPI

OpenMP

UPC
The Partitioned Global Address Space (PGAS) Model

- Aka the Distributed Shared Memory (DSM) model

- Concurrent threads with a partitioned shared space
  - Memory partition $M_i$ has affinity to thread $Th_i$

- (+)ive:
  - Data movement is implicit
  - Data distribution simplified with global address space

- (-)ive:
  - Computation distribution and synchronization still remain programmer’s responsibility
  - Lack of performance transparency of local vs. remote accesses

- UPC, CAF, Titanium, X10, …
What is Unified Parallel C (UPC)?

• An explicit parallel extension of ISO C
• Single global address space
• Collection of threads
  — each thread bound to a processor
  — each thread has some private data
  — part of the shared data is co-located with each thread
• Set of specs for a parallel C
  — v1.0 completed February of 2001
  — v1.1.1 in October of 2003
  — v1.2 in May of 2005
• Compiler implementations by industry and academia
• A number of threads working independently in a \textit{SPMD} fashion
  —MYTHREAD specifies thread index (0..THREADS-1)
  —Number of threads specified at compile-time or run-time

• Synchronization when needed
  —Barriers
  —Locks
  —Memory consistency control
### UPC Memory Model

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread (\text{THREADS-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shared</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private 0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private 1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(\cdots)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private (\text{THREADS-1})</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A pointer-to-shared can reference all locations in the shared space, but there is data-thread **affinity**
- A private pointer may reference addresses in its private space or its local portion of the shared space
- Static and dynamic memory allocations are supported for both shared and private memory
A First Example: Vector addition

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

shared int v1[N], v2[N], v1plusv2[N];
/* cyclic distribution by default */

void main() {
    int i;
    for (i=0; i<N; i++)
        if (MYTHREAD==i%THREADS)
            v1plusv2[i]=v1[i]+v2[i];
}
//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];
}
3\textsuperscript{rd} Example: A More Convenient Implementation with upc\_forall

//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];
}

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shared Space
Shared and Private Data

• Shared objects placed in memory based on affinity
• Affinity can be also defined based on the ability of a thread to refer to an object by a private pointer
• All non-array shared qualified objects, i.e. shared scalars, have affinity to thread 0
• Threads access shared and private data
### Examples of Shared and Private Data Layout:

**Assume THREADS = 3**

```c
shared int x;  /*x will have affinity to thread 0 */
shared int y[THREADS];
int z; /* private by default */
```

will result in the layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y[1]</td>
<td>y[2]</td>
</tr>
<tr>
<td>y[0]</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
</tbody>
</table>
shared int A[4][THREADS];

will result in the following data 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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```plaintext
shared int A[2][2*THREADS];

will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>⋮</th>
<th>Thread (THREADS-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td></td>
<td>A[0][THREADS-1]</td>
</tr>
<tr>
<td>A[0][THREADS]</td>
<td>A[0][THREADS+1]</td>
<td></td>
<td>A[0][2*THREADS-1]</td>
</tr>
</tbody>
</table>
```
Block-Cyclic Distributions for Shared Arrays

• Default block size is 1

• Shared arrays can be distributed on a block per thread basis, round robin with arbitrary block sizes.

• A block size is specified in the declaration as follows:
  – shared [block-size] type array[N];
  – e.g.: shared [4] int a[16];
Shared and Private Data

Assume THREADS = 4


will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[3][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[3][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Blocking of Shared Arrays

- Block size and THREADS determine affinity
- The term affinity means in which thread’s local shared-memory space, a shared data item will reside
- Element i of a blocked array has affinity to thread:

$$\left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \mod \text{THREADS}$$
Special Operators

- `upc_localsizeof(type-name or expression);`
  returns the size of the local portion of a shared object
- `upc_blocksizeof(type-name or expression);`
  returns the blocking factor associated with the argument
- `upc_elmsizeof(type-name or expression);`
  returns the size (in bytes) of the left-most type that is not an array
Usage Example of Special Operators

typedef shared int sharray[10*THREADS];
sharray a;
char i;

• upc_localsizeof(sharray)  \rightarrow  10*\text{sizeof}(\text{int})
• upc_localsizeof(a)  \rightarrow  10 *\text{sizeof}(\text{int})
• upc_localsizeof(i)  \rightarrow  1
• upc_blocksizeof(a)  \rightarrow  1
• upc_elementssizeof(a)  \rightarrow  \text{sizeof}(\text{int})
String functions in UPC

• Equivalent of memcpy:
  — upc_memcpy(dst, src, size)
    – copy from shared to shared
  — upc_memput(dst, src, size)
    – copy from private to shared
  — upc_memget(dst, src, size)
    – copy from shared to private

• Equivalent of memset:
  — upc_memset(dst, char, size)
    – initializes shared memory with a character

• Shared blocks above must be contiguous with all of its elements having the same affinity

• Private block must also be contiguous
## UPC Pointers

**Where does it point to?**

<table>
<thead>
<tr>
<th></th>
<th>Private</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td><strong>PP</strong></td>
<td><strong>PS</strong></td>
</tr>
<tr>
<td>Shared</td>
<td><strong>SP</strong></td>
<td><strong>SS</strong></td>
</tr>
</tbody>
</table>
UPC Pointers

Needed for expressive data structures

- Private pointers pointing locally
  - int *p1

- Private pointers pointing into the shared space
  - shared int *p2

- Shared pointers pointing locally (not recommended!)
  - int *shared p3

- Shared pointers pointing to shared space
  - shared int *shared p4
UPC Pointers Challenges

- Handling shared data
- Supporting pointer arithmetic
- Supporting pointer casting
UPC Pointers

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

- 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>

<table>
<thead>
<tr>
<th>Thread #</th>
<th>Block Address</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 private pointer, the thread number of the pointer-to-shared may be lost
- Casting of a pointer-to-shared to a private pointer is well defined only if the pointed to object has affinity with the local thread
Special Functions

- `size_t upc_threadof(shared void *ptr);` returns the thread number that has affinity to the object pointed to by `ptr`.
- `size_t upc_phaseof(shared void *ptr);` returns the index (position within the block) of the object which is pointed to by `ptr`.
- `size_t upc_addrfield(shared void *ptr);` returns the address of the block which is pointed at by the pointer to shared.
- `shared void *upc_resetphase(shared void *ptr);` resets the phase to zero.
- `size_t upc_affinitysize(size_t ntotal, size_t nbytes, size_t thr);` returns the exact size of the local portion of the data in a shared object with affinity to a given thread.
pointer to shared Arithmetic Examples:

Assume THREADS = 4

#define N 16

shared int x[N];

shared int *dp=&x[5], *dp1;

dp1 = dp + 9;
Assume THREADS = 4

shared[3] int x[N], *dp=&x[5], *dp1;

dp1 = dp + 9;

Thread 0

|------|------|------|

Thread 1

|------|------|------|

Thread 2

|------|------|------|

Thread 3

|------|-------|-------|

\[
\begin{align*}
&\text{Thread 0} \\
&X[0] \quad X[1] \quad X[2] \\
\end{align*}
\]

\[
\begin{align*}
&\text{Thread 1} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Thread 2} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Thread 3} \\
\end{align*}
\]
UPC Pointers

Example Pointer Castings and Mismatched Assignments:

• Pointer Casting

```c
shared int x[THREADS];

int *p;

p = (int *) &x[MYTHREAD]; /* p points to x[MYTHREAD] */
```

—Each of the private pointers will point at the x element which has affinity with its thread, i.e. MYTHREAD
UPC Pointers

• Mismatched Assignments

Assume THREADS = 4

`shared int x[N];`

`shared[3] int *dp=&x[5], *dp1;`

`dp1 = dp + 9;`

— The last statement assigns to dp1 a value that is 9 positions beyond dp

— The pointer will follow its own blocking and not that of the array
### UPC Pointers: Mismatched Assignments

Assume THREADS = 4

```c
shared int x[N];
shared[3] int *dp=&x[5], *dp1;

dp1 = dp + 9;
```

—The last statement assigns to dp1 a value that is 9 positions beyond dp

—The pointer will follow its own blocking and not that of the array

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X[16]</td>
<td></td>
<td></td>
<td>X[15]</td>
</tr>
</tbody>
</table>
UPC Pointers

• Given the declarations
  
  \[
  \text{shared}[3] \text{ int } *p; \\
  \text{shared}[5] \text{ int } *q;
  \]

• Then
  
  \[
  p=q; /\ast \text{ is acceptable (an implementation may require an explicit cast, e.g. } p=(*\text{shared}[3])q;) /\ast
  \]

• Pointer p, however, will follow pointer arithmetic for blocks of 3, not 5 !!

• A pointer cast sets the phase to 0
Worksharing with upc_forall

• Distributes independent iteration across threads in the way you wish—typically used to boost locality exploitation in a convenient way

• Simple C-like syntax and semantics

  upc_forall(init; test; loop; affinity)
  statement
  — Affinity could be an integer expression, or a
  — Reference to (address of) a shared object
Work Sharing and Exploiting Locality via \texttt{upc\_forall()}

- \textbf{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];
  ```

- \textbf{Example 2: implicit affinity with integer expressions and distribution in a round-robin fashion}
  
  ```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];
  ```

\textbf{Note:} Examples 1 and 2 result in the same distribution
Work Sharing: upc_forall()

- Example 3: Implicitly with distribution by chunks
  
  ```
  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>
Synchronization

• Explicit synchronization with the following mechanisms:
  — Barriers
  — Locks
  — Memory Consistency Control
  — Fence
Synchronization - Barriers

• No implicit synchronization among the threads

• UPC provides the following barrier synchronization constructs:
  — Barriers (Blocking)
    – upc_barrier expr\text{\_opt}\;
  — Split-Phase Barriers (Non-blocking)
    – upc_notify expr\text{\_opt}\;
    – upc_wait expr\text{\_opt}\;

Note: upc_notify is not blocking upc_wait is
Synchronization - Locks

• In UPC, shared data can be protected against multiple writers:
  — void upc_lock(upc_lock_t *l)
  — int upc_lock_attempt(upc_lock_t *l) //returns 1 on success and 0 on failure
  — void upc_unlock(upc_lock_t *l)

• Locks are allocated dynamically, and can be freed
• Locks are properly initialized after they are allocated
Memory Consistency Models

• Has to do with ordering of shared operations, and when a change of a shared object by a thread becomes visible to others

• Consistency can be *strict* or *relaxed*

• Under the relaxed consistency model, the shared operations can be reordered by the compiler / runtime system

• The strict consistency model enforces sequential ordering of shared operations. (No operation on shared can begin before the previous ones are done, and changes become visible immediately)
Memory Consistency - Fence

• UPC provides a fence construct
  — Equivalent to a null strict reference, and has the syntax
    – upc_fence;
  — UPC ensures that all shared references are issued before the upc_fence is completed
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;
    }

—We 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.
—We could have used a fence in the same places. Affects shared references at all threads!
—The above works as an example of point to point synchronization.
Overview UPC Collectives

- A collective function performs an operation in which all threads participate

- Recall that UPC includes the collectives:
  - upc_barrier, upc_notify, upc_wait, upc_all_alloc, upc_all_lock_alloc

- Collectives library include functions for bulk data movement and computation.
  - upc_all_broadcast, upc_all_exchange, upc_all_prefix_reduce, etc.
Library Calls for Bulk Data Transfer

- `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)`
Library Calls for Non-blocking Data Transfer

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
• Returns a upc_handle_t - an opaque handle representing the initiated asynchronous operation
• Complete using one of two functions:
  —block for completion: void bupc_waitsync(upc_handle_t handle);
  —Non-blocking test: int bupc_trysync(upc_handle_t handle);