Unified Parallel C

UPC

Slides based on those found at http://www2.hpcl.gwu.edu/pgas09/tutorials/upc_tut.pdf

see upc.gwu.edu for more UPC information, or the Berkeley site
# Contrast with MPI

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>MPI</th>
<th>OpenMP</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message passing</td>
<td>Shared Memory</td>
<td>Dist. Shared Memory</td>
<td></td>
</tr>
<tr>
<td>Expressing Parallelism</td>
<td>Library</td>
<td>Library Directives</td>
<td>C Extension</td>
</tr>
<tr>
<td>Architecture supported</td>
<td>MIMD</td>
<td>SMP</td>
<td>MIMD</td>
</tr>
<tr>
<td>Incremental parallelizing</td>
<td>Not really</td>
<td>Yes</td>
<td>Not really</td>
</tr>
<tr>
<td>Locality exploitation</td>
<td>Yes, using dist and comm</td>
<td>NO</td>
<td>Yes: blocking and affinity</td>
</tr>
<tr>
<td>Collective operations</td>
<td>Yes</td>
<td>Compiler/Pgmmer input</td>
<td>Compiler/Pgmmer Input</td>
</tr>
<tr>
<td>Data distribution in Declarations</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Consistency</td>
<td>N/A</td>
<td>Strict</td>
<td>Strict or relaxed</td>
</tr>
<tr>
<td>Dynamic memory alloc</td>
<td>Private only</td>
<td>Yes</td>
<td>Private or shared, w/or without blocking</td>
</tr>
<tr>
<td>pointers to dist. data</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Barriers/Wait</td>
<td>critical sect, locks, etc.</td>
<td>barriers, locks, cons. ctrl</td>
</tr>
</tbody>
</table>
UPC -- design philosophy

- Start with C
- Keep C low-level control features (addresses, pointers, etc.)
- Add parallelism, learn from previous languages
- Take input and suggestions from the developer’s community
- Integrate and work with experts from government, vendors, academia
UPC design philosophy

• Assume programmers know what they are doing
• Put programmers close to the hardware, let them exploit hardware properties
  • Can get good performance without super-powerful compilers
• Can also get into trouble
• Concise and efficient syntax like C
• Easy to implement on different architectures
• High performance at the system and node levels
UPC a PGAS language

- PGAS is Partitioned Global Address Space
- Unlike MPI, one address space covers all objects ("objects" in the sense of something in memory, not in the OO sense)
- Unlike OpenMP/Pthreads/Java, address space is assumed to be partitioned across multiple nodes and (perhaps) physically disjoint address spaces
UPC Execution Model

- A number of threads working independently in a **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
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.
Vector add in UPC

#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];
    }
}
A better implementation using upc_forall

#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++;
        v1plusv2[i] = v1[i] + v2[i];
    }

• upc_forall handles distribution of work
• Easier since UPC keeps track of assignment of work to threads
Assume THREADS=3

shared int x; /* x has an affinity for process 0 */
share int y[THREADS]
int z; // private -- one per thread

<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></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
</tbody>
</table>

Friday, April 1, 16
Shared and private data (2)

**shared int A[4][THREADS]**

Data is spread across the processors in a round-robin fashion
Different distributions of data

• Default block size is 1 -- gives a cyclic distribution

• Other block sizes can be specified to achieve a block or block-cyclic distribution

• Thread affinity is given by \[ \left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \mod \text{THREADS} \]

shared [block-size] type array-name[N]

shared[4] int a[16]
Shared and private data

- Assume \textbf{THREADS} = 4
- will result in the following layout:

UPC provided string functions can be used to move blocks of data among threads

Query operators provided to determine blocking information and affinity information of arrays and types
UPC Pointers

- Two attributes
  - location of storage pointed to
  - location of the pointer itself

<table>
<thead>
<tr>
<th></th>
<th>private</th>
<th>shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>data location/</td>
<td>PP</td>
<td>PS</td>
</tr>
<tr>
<td>pointer location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared</td>
<td>SP</td>
<td>SS</td>
</tr>
</tbody>
</table>

Where pointer is

Where pointed to data is
UPC Pointers

int *p1; /* private pointer pointing to local storage */
shared int *p2; /* private pointer pointing into shared space. A “pointer to shared” */
int *shared p3; /* shared pointer pointing locally */
shared int* shared p4; /*shared pointer pointing into the shared space */

People sometimes use *shared pointer* to mean a pointer pointing into the shared space (*p2*), but could also be a pointer residing in the shared space (*p4*)
UPC Pointers

private pointer (no shared)
int *p1; /* private pointer pointing to local storage */

shared int *p2; /* private pointer pointing into shared space. A “pointer to shared” */

shared ptr
int *shared p3; /* shared pointer pointing locally */

shared ptr
shared int* shared p4; /*shared pointer pointing into the shared space */

shared int
shared ptr

a bad idea. Why?
int *p1; /* private point pointing to local storage */
shared int *p2; /* private point pointing into shared space */
int *shared p3; /* shared pointer pointing locally */
shared int *shared p4; /* shared pointer pointing into the shared space */
Implementation of pointer to shared objects

- UPC pointers to shared objects have three fields:
  - **thread number**: thread whose storage that contains to the object being pointed to
  - **Block address**: local address of the block that contains the object
  - **Phase**: contains the location of the object within the block
UPC Pointers manipulations

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private is legal, **but not vice versa**
- Casting from pointer-to-shared to pointer-to-private may lose the “owning” thread number information
- Casting from pointer-to-shared to pointer-to-private is well defined **only** if the private pointer resides on the same thread as the data
UPC pointer arithmetic

Assume **THREADS=4**

```c
#define N 16
shared int x[N];
shared int *dp=&x[5], *dp1;
dp1 = dp+9;
```
blocking is part of type -- can lead to interesting pointer arithmetic

Assume **THREADS=4**

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

\[ dp1 = dp + 9; \]
Why do this?

- Allows pointer arithmetic to be used to scan elements within a block and to exploit locality
- Pointer follows its own declared blocking and not that of what it points to

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

• Given the declarations

  shared[3] int *p;

  shared[5] int *q;

• Then

  p=q is acceptable (may need an explicit cast w/some implementations)

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

- Distributes work across threads
- Simple C-like syntax and semantics
- `upc_forall(init; test; incr; affinity);`
  - affinity can be an integer expression or
  - a reference to (address of) a shared object
Exploiting locality with upc_forall

Example 1:

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];

Iteration $i$ executes on the processor that $a[i]$ resides on.

Example 2:

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

_expression_ mod THREADS gives the thread iteration executes on. Same distribution as in example 1.
More working sharing with upc_forall

Example 3: distribute 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];

<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>
Other supported functionality

- Dynamic memory allocation
- Synchronization
- Memory consistency models
Dynamic memory allocation

- Collective operations executed by every thread, allocates a contiguous chunk in the shared space, all threads get the same value

```c
shared [N] int *ptr;
ptr = (shared [N] int *)
upc_all_alloc( THREADS, N*sizeof( int ) );
```
• **non-collective operations** allocate a contiguous region in the shared space

• each thread invoking this allocates a different region and gets a pointer to that region

```c
shared [N] int *ptr;

ptr =
    (shared [N] int *)
upc_global_alloc( THREADS, N*sizeof( int ));
```

![Diagram](image-url)
Local shared memory allocation

```
shared [] int *ptr;
ptr = (shared [] int *)upc_alloc(N*sizeof(int));
```
UPC free

- UPC_free frees up storage **pointed-to** by a shared pointer
- local allocations to local pointers can be done by malloc
UPC synchronization

- barriers that can involve various subsets of threads
- Can be blocking or non-blocking
  - With non-blocking, hit the barrier, do some work, then wait for the barrier when data written before the barrier is needed
- Locks, lock_attempt allows a lock to be checked
Memory models

- Can either have a relaxed or strict memory model
- Strict is SC
- Can be specified on a per-variable basis
- The compiler enforces this
Reduced Coding Effort is Not Limited to Random Access—NPB Examples

<table>
<thead>
<tr>
<th></th>
<th>SEQ1</th>
<th>UPC</th>
<th>SEQ2</th>
<th>MPI</th>
<th>UPC Effort (%)</th>
<th>MPI Effort (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB-CG</td>
<td>#lines</td>
<td>665</td>
<td>710</td>
<td>506</td>
<td>1046</td>
<td>6.77</td>
</tr>
<tr>
<td></td>
<td>#chars</td>
<td>16145</td>
<td>17200</td>
<td>16485</td>
<td>37501</td>
<td>6.53</td>
</tr>
<tr>
<td>NPB-EP</td>
<td>#lines</td>
<td>127</td>
<td>183</td>
<td>130</td>
<td>181</td>
<td>44.09</td>
</tr>
<tr>
<td></td>
<td>#chars</td>
<td>2868</td>
<td>4117</td>
<td>4741</td>
<td>6567</td>
<td>43.55</td>
</tr>
<tr>
<td>NPB-FT</td>
<td>#lines</td>
<td>575</td>
<td>1018</td>
<td>665</td>
<td>1278</td>
<td>77.04</td>
</tr>
<tr>
<td></td>
<td>#chars</td>
<td>13090</td>
<td>21672</td>
<td>22188</td>
<td>44348</td>
<td>65.56</td>
</tr>
<tr>
<td>NPB-IS</td>
<td>#lines</td>
<td>353</td>
<td>528</td>
<td>353</td>
<td>627</td>
<td>49.58</td>
</tr>
<tr>
<td></td>
<td>#chars</td>
<td>7273</td>
<td>13114</td>
<td>7273</td>
<td>13324</td>
<td>80.31</td>
</tr>
<tr>
<td>NPB-MG</td>
<td>#lines</td>
<td>610</td>
<td>866</td>
<td>885</td>
<td>1613</td>
<td>41.97</td>
</tr>
<tr>
<td></td>
<td>#chars</td>
<td>14830</td>
<td>21990</td>
<td>27129</td>
<td>50497</td>
<td>48.28</td>
</tr>
</tbody>
</table>

\[
\text{UPC}_{\text{effort}} = \frac{\#\text{UPC} - \#\text{SEQ1}}{\#\text{SEQ1}} \\
\text{MPI}_{\text{effort}} = \frac{\#\text{MPI} - \#\text{SEQ2}}{\#\text{SEQ2}}
\]

SEQ1 is C
SEQ2 is from NAS, all FORTRAN except for IS