Parallel Architectures

Shared Memory

Distributed Memory

Programming Models

Habanero-Java
Java Threads
Cilk
OpenMP
Pthreads

Process/Thread
Memory

interconnect

MPI
Map-Reduce
UPC
CAF
Performance and Programmability Concerns

Data movement and synchronization are expensive

To minimize overheads

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

⇒ Significant programmability challenges with addressing these overheads in a shared-nothing programming model like MPI
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 lacking in thread-based models
  - computation partitioning
  - communication placement HJ places help with locality control but not data distribution

- Data movement and synchronization as language primitives
  - amenable to compiler-based communication optimization

- Global view rather than local view

![Diagram of Global and Local Views](image-url)
Partitioned Global Address Space (PGAS) Languages

- Unified Parallel C (extension of C)
- Coarray Fortran (extension of Fortran)
- Titanium (extension of early version of Java)

Related efforts: newer languages developed since 2003 as part of the DARPA High Productivity Computing Systems (HPCS) program
  - IBM: X10 (starting point for Habanero-Java)
  - Cray: Chapel
  - Oracle/Sun: Fortress
Data Distributions

- Motivation for distributions: partitioning and mapping arrays elements to processors
- In HJlib, distributions are used to map computations to places for affinity
- For Unified Parallel C (UPC), distributions map data onto distributed-memory parallel machines (Thread = Place)

Like shared vs. private/local data in HJ, except now each datum also has an “affinity” with a specific thread/place
Unified Parallel C (UPC)

- An explicit parallel extension of ISO C
  - 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 CPU
    - 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 working independently in a SPMD fashion
  - MYTHREAD specifies thread index (0..THREADS-1)
    - Like MPI processes and ranks
    - # threads specified at compile-time or program launch

- Partitioned Global Address Space (different from MPI)

- 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
  - by default, elements of shared arrays are allocated “round robin” among memory modules co-located with each thread (cyclic distribution)
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 cyclic layout

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>y[0]</td>
<td>y[1]</td>
<td>y[2]</td>
</tr>
<tr>
<td>y[6]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Multi-dimensional Shared Array

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

For THREADS = 3, we get the following cyclic 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>
Shared and Private Data

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><strong>x</strong></td>
<td><strong>x</strong></td>
<td><strong>x</strong></td>
</tr>
<tr>
<td><strong>y[0]</strong></td>
<td><strong>y[1]</strong></td>
<td><strong>y[2]</strong></td>
</tr>
<tr>
<td><strong>z</strong></td>
<td><strong>z</strong></td>
<td><strong>z</strong></td>
</tr>
</tbody>
</table>
Controlling the Layout of Shared Arrays

- Can specify a blocking factor for shared arrays to obtain block-cyclic distributions
  - default block size is 1 element ⇒ cyclic distribution

- 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 Multi-dimensional Data

- Consider the data declaration
  
  ```
  #shared [3] int A[4][THREADS];
  ```

- When THREADS = 4, this results 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>

The mapping is not pretty for most blocking factors.
A Simple UPC Program: Vector Addition

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

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

void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD == i % THREARDS)
            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 \texttt{upc\_forall}

- Distributes independent iterations across threads
- Simple C-like syntax and semantics
  \begin{verbatim}
  upc forall (init; test; loop; affinity)
  \end{verbatim}
- 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])
  // Execute iteration i at a[i]'s thread/place
  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)
  // Execute iteration i at place i%THREADS
  a[i] = b[i] * c[i];
```

- Both yield a round-robin distribution of iterations
Vector Addition Using upc_forall

Thread 0  Thread 1

Iteration #: 0  1  2  3

<table>
<thead>
<tr>
<th></th>
<th>v1[0]</th>
<th>v1[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v1plusv2[0]</td>
<td>v1plusv2[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each thread executes subset of global iteration space as directed by the affinity clause

//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];
}
Work Sharing + Affinity with \texttt{upc forall}

- Example 3: implicit affinity by chunks

\begin{verbatim}
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
a[i] = b[i-1] * c[i+1];
\end{verbatim}

- Assuming 4 threads, the following results

\begin{tabular}{|c|c|c|}
\hline
i     & i*THREADS   & i*THREADS/100 \\
\hline
0..24 & 0..96       & 0     \\
\hline
25..49 & 100..196    & 1     \\
\hline
50..74 & 200..296    & 2     \\
\hline
75..99 & 300..396    & 3     \\
\hline
\end{tabular}
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];
    }
}

```
 Th. 0
 Th. 1
 Th. 2
```

=  

```
0 1 2
Th. 0
Th. 1
Th. 2
```

```
 Th. 0
```

*  

```
 Th. 0
```

```
 Th. 0
```

```
 Th. 0
```
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];
    }
}
Synchronization

- **Barriers (blocking)**
  - `upc_barrier`
    - like “next” operation in HJ

- **Split-phase barriers (non-blocking)**
  - `upc_notify`
    - like explicit (non-blocking) signal on an HJ phaser
  - `upc_wait`
    - `upc_wait` is like explicit wait on an HJ phaser

- **Lock primitives**
  - `void upc_lock(upc_lock_t *l)`
  - `int upc_lock_attempt(upc_lock_t *l) // like trylock()`
  - `void upc_unlock(upc_lock_t *l)`
Application Work in PGAS

- Network simulator in UPC (Steve Hofmeyr)
- Barnes-Hut in UPC (Marc Snir et al)
- Landscape analysis
  - “Contributing Area Estimation” in UPC (Brian Kazian, UCB)
- Gyrokinetic Tokamak Simulation Shifter code in CoArray Fortran (CAF)
  - Preissl, Wichmann, Long, Shalf, Ethier, Koniges (LBNL, Cray, PPPL)

Slide credit: Kathy Yelick, January 2011