New Programming Paradigms: Partitioned Global Address Space Languages
Outline

- Overview of the PGAS programming model
- An overview of the UPC Language
- An overview of Coarray Fortran
- Compiler optimizations for PGAS
- Conclusions
Parallel programming models

**Shared memory**
- Unified address space
- Direct access to all data from all threads
- Targets SMP-level parallelization
- Easier to program

**Distributed memory**
- Separate address spaces
- Communication through messages
  - Coordination between sender and receiver
- Targets cluster-level parallelization

---

**OpenMP**

**Message passing**
Partitioned Global Address Space

- Unified address space
- Direct access to all data from all threads
- Targets SMP-level parallelization
- Targets cluster-level parallelization
- Easier to program

Data is logically partitioned between local and remote
- Large performance delta between local and remote memory accesses (NUMA)
- Explicit data affinity allows programmer to manage access latencies

One-sided communication
- Data transfer between processors does not require intervention from both sender and receiver

Amenable to compiler analysis and optimization
PGAS Execution environment
Distributed model

- PGAS programs can be naturally mapped onto distributed-memory systems
- One or more processes per node
- Runtime system provides illusion shared address space
  - Manage network communication
  - Transparent to the programmer
PGAS Execution environment
Shared memory model

- PGAS programs can be mapped on shared-memory systems
- One or more processes with one or more threads each
- Two-level hierarchy may help model system topology
**PGAS Execution environment**

**Hybrid model**

- PGAS programs can also be mapped to systems with both cluster and SMP level parallelism

- No source changes required, just reconfiguration

- Compiler and runtime system optimizes communication inside each node and across nodes

- Distribution of work across a three-level hierarchy
  - Node
  - Process
  - Thread
An Overview of the UPC Language

- UPC is an extension of the C language
  - Fixed number of threads of execution, all starting main in parallel
  - Number of threads available as program variable THREADS
  - Global variable MYTHREAD specifies current thread index (0..THREADS-1)
  - Number of threads can be compiled-in or set during execution time

- SPMD execution model
  - All threads execute program redundantly
  - Work distribution through explicit check of thread index or upc_forall construct

- Explicit thread communication
  - Direct access to shared variables
  - Primitives for thread synchronization
Shared variables in UPC

- Regular C variables and objects are thread-local
  - Thread-local storage can only be accessed by corresponding thread

- Shared variables are explicitly annotated with the “shared” keyword
  - Only allowed for global scope variables (static, external)
  - Shared scalars are owned by thread 0

- Example

  ```c
  int A;  /* each thread keeps a separate copy of A */
  shared int B;  /* single instance of B accessed by all threads */
  ```

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>THREADS-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>...</td>
<td>A</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning of shared arrays in UPC

- Shared arrays are distributed across all threads
- Elements of shared arrays are distributed in a round-robin fashion
  
  ```
  shared int a[THREADS*2]; /* 2 elements per thread */
  ```
  
  ```
  a  0  1  2  3  ...  N-1  0  1  2  3  ...  N-1
  ```

- Number of elements per thread determined by the blocking factor (default=1)
  
  ```
  shared [2] int b[THREADS*2]; /* 2 elements per thread */
  ```
  
  ```
  b  0  0  1  1  2  2  3  3  4  4  ...  N-1  N-1
  ```

- Language provides mechanism to query data partitioning
  - `upc_threadof(shared void *)` returns id of owner thread
    
    ```
    upc_threadof(&a[2]) == 2
    upc_threadof(&b[2]) == 1
    ```
Pointers to shared in UPC

- UPC provides pointers that point to shared variables, either local or remote
  - Typically larger and more expensive to dereference than regular C pointers
  - Supports pointer arithmetic for array members

- Example

  ```c
  shared int *p; /* declares p as a pointer to shared int */
  struct link { /* shared linked list declaration */
    shared struct link *next;
    int data;
  };
  ```

- Dynamic memory management

  ```c
  shared void *upc_alloc(size_t size);
  shared void *upc_global_alloc(size_t size, size_t blocks);
  void upc_free(shared void *);
  ```
Work sharing in UPC

- UPC adds a special type of loop for work distribution
  ```
  upc_forall(init; test; loop; affinity) {
    ...
  }
  ```

- Affinity expression determines which iterations will be run by each thread
  - Integer
    - Iteration executes if (affinity%THREADS == MYTHREAD)
  - Shared pointer
    - Iteration executes if (upc_threadof(affinity) == MYTHREAD)

- Loop iterations must be independent
- No early exit or entry into the loop are allowed
Thread synchronization in UPC

- UPC provides a weakly-ordered memory model
  - No storage access order on shared variables
  - Optional mode where all shared data accesses are sequential consistent
    **upc_fence**: Ensures all preceding storage accesses are completed before continuing

- Barrier synchronization
  **upc_barrier**: Waits for all threads in the program to reach the barrier

- Split-phase barriers
  - Allows execution of unrelated computation while waiting
    **upc_notify**: Signal to other threads that synchronization point has been reached
    **upc_wait**: Wait until all thread have reached synchronization point

- Simple locking facilities
  **upc_lock(...);**
  **upc_unlock(...);**
UPC utility functions and Collectives

- UPC provides utility functions for data transfer, analogous to libc counterparts
  ```
  upc_memset(...);
  upc_memcpy(...);
  upc_memget(...);
  upc_memput(...);
  ```

- Collectives
  - Collectives are executed by all threads to cooperate completing a certain task
  - Include all required synchronization
  - Data movement: gather, scatter, permute, broadcast, etc..
  - Computation: reductions
Hello World in UPC

```c
#include <upc.h>
#include <stdio.h>

int main() {
    printf("Thread %d of %d: Hello UPC world\n", MYTHREAD, THREADS);
    return 0;
}
```

hello > xlupc helloWorld.upc
hello > env UPC_NTHREADS=4 ./a.out
Thread 1 of 4: Hello UPC world
Thread 0 of 4: Hello UPC world
Thread 3 of 4: Hello UPC world
Thread 2 of 4: Hello UPC world
An Overview of Coarray Fortran

- Natural extension of Fortran
- Part of the Fortran 2008 standard, to be ratified in 2010

- SPMD execution model
  - Fixed number of images, all running independently
  - All images execute program redundantly
  - Intrinsic functions NUM_IMAGES() and THIS_IMAGE()

- Explicit thread communication
  - Coarrays for data exchange between images
  - Built-in functions for synchronization
Coarrays

- Coarrays are Fortran variables that are replicated over multiple images
- Explicit codimension determines data ownership
  - Specified with square brackets after variable declaration and access
  - Images can access coarrays owned by other images
  - If no codimension is specified on access, defaults to current image
  - Can have multiple codimensions, to represent data attributes or system topology
  - Use * on the trailing codimension to extend to number of images
  - Requires SAVE or ALLOCATABLE attributes

- Example

```fortran
REAL :: A[*]
REAL :: B
```

<table>
<thead>
<tr>
<th>Image</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>NUM_IMAGES()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>...</td>
<td>B</td>
</tr>
</tbody>
</table>
Image synchronization in Coarray Fortran

- No implied storage access order between threads
  - `SYNC MEMORY`: Ensures all preceding storage accesses are completed before continuing

- Barrier synchronization
  - `SYNC ALL`: Waits for all images to reach the synchronization point

- Partial synchronization
  - `SYNC IMAGES`: Waits for listed group of images to reach the synchronization point

- Simple locking facilities
  - `CRITICAL`
  - `LOCK`
Hello World in Coarray Fortran

```fortran
program abc
write (*,*) 'Hello world from thread', THIS_IMAGE(), ', of', NUM_IMAGES()
end
```

```bash
$ xlf90_r -qcaf hello.f
** abc   === End of Compilation 1 ===
1501-510  Compilation successful for file hello.f.
$ env CAF_NUM_IMAGES=4 ; poe a.out -hfile ~/.rhosts
Hello world from thread 2 of 4
Hello world from thread 3 of 4
Hello world from thread 4 of 4
Hello world from thread 1 of 4
```
Major differences between UPC and Coarray Fortran

**UPC**
- Data distribution is implied, controlled by blocking factor
- Work distribution through upc_forall construct
- Data movement through utility functions and collectives
- Computation collectives
- Not a language standard

**Coarray Fortran**
- Data distribution is explicit, controlled by separate codimension
- Implicit work distribution through codimension
- Data movement through F90 array language
- Computational collectives on Fortran TR
- Part of Fortran 2008 Standard
Compiler optimization for PGAS languages

- PGAS languages facilitate compiler optimization
  - Extension of the underlying language, with full compiler awareness
  - One-sided communication eliminates hurdle of matching sends and receives
  - High-level collective operations permit automated use of specialized hardware capabilities

- Variety of opportunities for compiler optimization
  - Locality analysis: Bypass runtime overhead on access to local shared variables
  - Communication aggregation: Aggregate remote accesses into blocks
  - Communication/computation overlap: Identify computation that can be executed while communication is taking place
  - Caching: Automatically cache remote shared variables
  - Remote execution: Offload computation to the node that owns the input/output data
The IBM PGAS compilers

- XL UPC Compiler (alpha)
  - Available through alphaworks
  - Supports BG/L, AIX and Linux on Power
  http://www.alphaworks.ibm.com/tech/upccompiler

- IBM is committed to language standard conformance

- Winner of multiple HPC Challenge Class 2 Awards
  - 2009: Best performance
  - 2008: Best performance and most productive implementation
  - 2006: Best productivity and performance

Check often for updates!
Conclusion

- PGAS languages allow parallelization of applications at both SMP and cluster level

- Promise high performance at a lower development cost
  - Easier to develop than message passing
  - Performance optimization assistance from compilation subsystem

- Many parallel programming languages are following this paradigm
  - UPC http://upc.gwu.edu/
  - Fortran 2008 (Coarrays) http://www.nag.co.uk/sc22wg5/

- Many challenges still for these languages to be successful
  - Standarization
  - Quality implementations
  - Integration with current programming frameworks and tools