OpenMP and the PGAS Model

CMSC714 – Sept 15, 2015
Guest Lecturer: Ray Chen
Last Time: Message Passing

• Natural model for distributed-memory systems
  – Remote (“far”) memory must be retrieved before use
  – Programmer responsible for specifying:
    • Participants (Single peer, collective communication, etc.)
    • Data types (MPI_CHAR, MPI_DOUBLE, etc.)
    • Logical synchronization

• How about shared-memory systems?
  – All processors can directly access any memory
OpenMP: Open Multi-Processing

- Portable interface for multiple processor systems
  - Agnostic to architecture or language
- Alternative to POSIX threads
  - Thread management too low-level for HPC applications
  - Task parallelism vs. data parallelism
- Not a language
  - Extends existing language (C/C++/Fortran)
  - Compiler responsible for low-level thread management
  - Allows for incremental approach to parallelism
OpenMP Fork/Join Model

- Single master thread executes until parallel region
- When a parallel region is reached
  - At start, N-1 additional threads are spawned (Fork)
  - All N threads execute the same code
    - Different code paths achieved through unique thread ID
  - At end, threads are synchronized via barrier (Join)
OpenMP Syntax

- Compiler directives
  - Express parallel structure of code
  - Clauses modify directives

```
#pragma omp <directive> <clauses>
{
  // Code region affected by directive.
}
```

- Run-time library routines

```
#include <omp.h>
...
int id = omp_get_thread_num();
```
OpenMP parallel Directive

• Begin a new parallel region

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

int main(int argc, char* argv[]) {
    #pragma omp parallel
    {
        printf("Thread %d of %d.\n", omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}
```

```bash
$ gcc -fopenmp a.c
$ ./a.out
Thread 0 of 4
Thread 3 of 4
Thread 2 of 4
Thread 1 of 4
```
OpenMP Data Environment

• All variables have a data-sharing attribute
  – Shared: all threads use the same copy
  – Private: all threads use local storage for this variable
  – Firstprivate: Like private, but value is initialized on fork
  – Lastprivate: Like private, but value updated on join

• In general:
  – Variables defined inside a parallel region are private
  – Variables defined outside a parallel region are shared
  – Not always true: more details available in the spec

• Data-sharing attributes can be overridden
OpenMP Worksharing Constructs

• Used to parallelize loops
  – HPC programs contain computationally intensive loops

```c
#pragma omp loop
for (i = 0; i < MAX; ++i) {
  ...
}
```

• Restrictions
  – Loop iterations must be independent
  – Strict rules for loop initialization, test, and increment
  – Premature termination of loops not supported
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int cap = atoi(argv[1]);
    int count = 0;
    int num = 0;

    #pragma omp parallel loop reduction(max:num) schedule(dynamic)
    for (int i = 0; i < cap; ++i) {
        int prime = 1;
        for (int j = 0; prime && j * j <= i; ++j)
            prime = i % j;

        if (prime) {
            #pragma omp atomic
            ++count;
            num = i;
        }
    }

    printf("Prime #%d is %d
", count, num);
    return 0;
}

Keep a private copy of num for each thread, and copy the maximum value back when the loop is finished.

Parallelize the following loop.

Use a work queue to load-balance threads.

Synchronize the next line; count is a shared variable.
OpenMP Synchronization

#pragma omp barrier
Threads pause here until all reach this point

#pragma omp critical
Threads will execute one at a time

#pragma omp single
Only one thread will ever execute
Other threads wait in an implied barrier at end of region

#pragma omp master
Only the master thread will execute
Other threads skip the region without waiting
Puzzle #1

```c
int x = 1;
#pragma omp parallel num_threads(2) {
    #pragma omp single
    {
        ++x;
    }
    printf("%d\n", x);
}
```
OpenMP Sync Puzzles

- Puzzle #2

```c
int x = 1;
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        ++x;
    }
    printf("%d\n", x);
}
```

or

```
1
2
```

or

```
2
2
```
OpenMP Sync Puzzles

• Puzzle #3

```c
int x = 1;
#pragma omp parallel num_threads(2)
{
  ++x;
  #pragma omp barrier
  printf("%d\n", x);
}
```

Shared variable access is not automatically synchronized. Behavior is undefined.
Limitations of OpenMP

- Not fool-proof
  - Data dependencies, conflicts, race conditions
  - Gives you plenty of rope to hang yourself with

- OpenMP and Amdahl’s Law
  - What is Dagum and Menon take on this issue?

- Computational Problem Size
  - USA #1 supercomputer (Titan) has 32GB/node
  - World #1 supercomputer (Tianhe-2) has 64GB/node
  - What if our datasets are larger than 64GB?
    - “You’re gonna need a bigger boat...”
MPI + OpenMP

- Programming in MPI + OpenMP is hard
  - Necessary for today’s large distributed memory systems
PGAS Programming Model

- **Partitioned Global Address Space**
- Presents an abstracted shared address space
  - Simplifies the complexity of MPI + OpenMP
- Exposes data/task locality for performance
- Several languages use this model
  - UPC, Fortress, HPF, X10, etc.
  - Ever heard of these?
Chapel Themes

• Designed from scratch
  – Blank slate as opposed to a language extension
  – Allows for clearer, more expressive language semantics

• Multi-resolution design
  – Higher-level abstractions built from lower-level ones
  – Allows users to mix and match as needed

• Global-view programming
  – Data can be declared using global problem size
  – Data can be accessed using global indices
Data Parallelism, By Example
Domains

Domain:
- A first-class index set
- The fundamental Chapel concept for data parallelism

```chapel
config const m = 4, n = 8;

const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
```
Data Parallelism by Example: Jacobi Iteration

repeat until max change $< \varepsilon$

\[ \sum \left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array} \right) \div 4 \]
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

Declare domains (first class index sets)

{lo..hi, lo2..hi2} ⇒ 2D rectangular domain, with 2-tuple indices

Dom1[Dom2] ⇒ computes the intersection of two domains

.exterior() ⇒ one of several built-in domain generators
Jacobi Iteration in Chapel

```c
class config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

### Declare arrays

- `var` can be modified throughout its lifetime
- `: [Dom] T` array of size `Dom` with elements of type `T`
- `(no initializer)` values initialized to default value (0.0 for reals)

![Diagram showing BigD, A, and Temp arrays with grid patterns]
Jacobi Iteration in Chapel

\[
\text{config const n = 6,}
\]
\[
\quad \text{epsilon = 1.0e-5;}
\]
\[
\text{const BigD = } \{0..n+1, 0..n+1\},
\]
\[
\quad \text{D = BigD[1..n, 1..n],}
\]
\[
\quad \text{LastRow = D.exterior(1,0);}
\]
\[
\text{var A, Temp : [BigD] real;}
\]
\[
\text{A[LastRow] = 1.0;}
\]

\[
\text{do}
\]
\[
\quad \text{\textbf{Set Explicit Boundary Condition}}
\]
\[
\text{Arr[Dom] } \Rightarrow \text{ refer to array slice (“forall i in Dom do …Arr[i]…”)}
\]
Jacobi Iteration in Chapel

```
config const n = 6,

Compute 5-point stencil
forall ind in Dom \Rightarrow parallel forall expression over Dom's indices,
    binding them to ind
(here, since Dom is 2D, we can de-tuple the indices)

\[ \sum \begin{bmatrix} \text{colors} \end{bmatrix} \div 4 \]

\[ \begin{bmatrix} \text{colors} \end{bmatrix} \]

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
    } while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

Compute maximum change

op reduce \Rightarrow collapse aggregate expression to scalar using op

Promotion: abs() and – are scalar operators; providing array operands
results in parallel evaluation equivalent to:

```forall (a, t) in zip(A, Temp) do abs(a - t)```

```chapel
do {
    forall (i, j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    A = [0..n, 0..n];

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    forall (i,j) in D do

        const delta = max reduce abs(A[D] - Temp[D]);
        A[D] = Temp[D];
} while (delta > epsilon);

writeLn(A);
```
Domain Maps

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...

\[ A = B + \alpha \times C; \]

...to the target locales’ memory and processors:
config const numIters = 100000;

const D = {1..numIters};

forall i in D do
  writeln("Hello, world!",
          "from iteration ", i, " of ", numIters);
config const numIters = 100000;

const D = {1..numIters} dmapped Cyclic(startIdx=1);

forall i in D do
    writeln("Hello, world!",
             "from iteration ", i,
             " of ", numIters);
Layouts and Distributions

Domain Maps fall into two major categories:

*layouts:*
  - target a shared memory
  - examples: row- and column-major order, tilings, compressed sparse row, space-filling curves

*distributions:*
  - map indices/elements to distributed memories
  - examples: Block, Cyclic, Block-Cyclic, Recursive Bisection, …
Sample Distributions: Block and Cyclic

\[ \text{var Dom = \{1..4, 1..8\} dmapped Block( \{1..4, 1..8\})}; \]

\[ \text{var Dom = \{1..4, 1..8\} dmapped Cyclic(startIdx=(1,1))}; \]

distributed to

\[ \text{L0 L1 L2 L3} \]
\[ \text{L4 L5 L6 L7} \]
STREAM Triad: Chapel (multicore)

```plaintext
const ProblemSpace = {1..m};

var A, B, C: [ProblemSpace] real;

A = B + alpha * C;
```

No domain map specified => use default layout
- current locale owns all domain indices and array values
- computation will execute using local processors only
STREAM Triad: Chapel (distributed, blocked)

```chapel
const ProblemSpace = {1..m}
    dmapped Block (boundingBox={1..m});

var A, B, C: [ProblemSpace] real;

A = B + alpha * C;
```
STREAM Triad: Chapel (distributed, cyclic)

\[
\text{const ProblemSpace} = \{1..m\}
\]
\[
\text{dmapped Cyclic(startIdx=1)};
\]

\[
\text{var A, B, C: [ProblemSpace] real;}
\]

\[
\alpha \cdot A = B + \text{alpha} \cdot C;
\]
Jacobi Iteration in Chapel (shared memory)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel (distributed memory)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this simple change, we specify a mapping from the domains and arrays to locales
Domain maps describe the mapping of domain indices and array elements to locales
specifies how array data is distributed across locales
specifies how iterations over domains.arrays are mapped to locales
Jacobi Iteration in Chapel (distributed memory)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);

use BlockDist;
```
Chapel’s Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly

2. Expert users can write their own domain maps in Chapel
   - to cope with any shortcomings in our standard library

3. Chapel’s standard domain maps are written using the same end-user framework
   - to avoid a performance cliff between “built-in” and user-defined cases
Chapel Array Types

- **dense**
- **strided**
- **sparse**

- **associative**
- **unstructured**
All Domain Types Support Domain Maps

- dense
- strided
- sparse

- associative
- unstructured