The Chapel Runtime

Charm++ Workshop 2013
April 15, 2013

Greg Titus
Principal Engineer
Chapel Team, Cray Inc.
Outline

- Introduction
- Compilation Architecture
- Predefined Modules
- Runtime
- Example
- Future Work
Outline

- Introduction
- Compilation Architecture
- Predefined Modules
- Runtime
- Example
- Future Work
What is Chapel?

- An emerging parallel programming language
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry
  - Initiated under the DARPA HPCS program

- Overall goal: Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Support better portability than current programming models
  - Improve the robustness of parallel codes

- A work-in-progress
  - Just released v1.7

- [http://chapel.cray.com/](http://chapel.cray.com/)
Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software

Target Architectures:
- Cray architectures
- multicore desktops and laptops
- commodity clusters
- systems from other vendors
- in-progress: CPU+accelerator hybrids, many-core, …
Chapel Execution Model: Locality

- **Program launches on one or more locales**
  - *Locale*: in Chapel, something that has memory and processors
    - So far, usually a system node
  - User main program executes on locale 0
  - Other locales wait for work, to be delivered by Active Messages

- **User code controls execution locality**

  ```plaintext
  // Move execution to the locale associated with locale-expr
  // for the duration of statement.
  //
  on locale-expr do statement
  ```
User code creates parallelism in the form of tasks

//
// Task-parallel style, no implicit synchronization.
begin statement // one new task

//
// Task-parallel style, implicit barrier at end.
cobegin block-statement // one new task per stmt

coforall idx-var in iter-expr do  // one new task per iter
statement

//
// Data-parallel style, implicit barrier at end.
//
forall idx-var in iter-expr do  // #tasks <= #iterations,
statement // #iterations and locality
// may differ for each task
Outline

- Introduction
- Compilation Architecture
- Predefined Modules
- Runtime
- Example
- Future Work
Compiling Chapel

Chapel Source Code → chpl → Chapel Executable

Standard Modules (in Chapel)
Chapel Compilation Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Standard Modules (in Chapel)
- Internal Modules (in Chapel)

Runtime Support Library (in C)
- Tasks/Threads
- Communication
- Memory
- ...

Chapel Compiler
Outline

- Introduction
- Compilation Architecture
- Predefined Modules
- Runtime
- Example
- Future Work
Chapel Compilation Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Standard Modules (in Chapel)
- Internal Modules (in Chapel)

Runtime Support Library (in C):
- Tasks/Threads
- Communication
- Memory
- …
Predefined Modules

● **A module** in Chapel encapsulates types, variables, and functions
  ● To bring the definitions in a module into a program, you use it (with exception below)
  ● Users can write modules
  ● Some predefined modules come with Chapel

● **Predefined modules**
  ● Internal
    ● Support the language
    ● Arrays, distribution maps, etc.
    ● Implicitly brought in; no use needed
  ● Standard
    ● Support user code
    ● Math, random numbers, time, etc.
    ● Must be explicitly brought in with a use statement
Outline

- Introduction
- Compilation Architecture
- Predefined Modules
- Runtime
- Example
- Future Work
Chapel Compilation Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Standard Modules (in Chapel)
- Internal Modules (in Chapel)
- Runtime Support Library (in C)
  - Tasks/Threads
  - Communication
  - Memory
  - ...
Chapel Runtime

- Lowest level of Chapel software stack
- Supports language concepts and program activities
- Relies on system and third-party services
- Written in C
- Composed of *layers*
  - A misnomer – these are not layers in the sense of being stacked
  - More like *posts*, in that they work together to support a shared load
  - Standardized interfaces
  - Interchangeable implementations
- Environment variables select layer implementations when building the runtime
  - And when compiling a Chapel program, also select which already-built runtime is linked with it
Chapel Runtime Organization

Chapel Runtime Support Library (in C)

- Communication
- Tasking
- Memory
- Launchers
- QIO
- Timers
- Standard

Standard and third-party libraries
Runtime Communication Layer

Chapel Runtime Support Library (in C)

Communication

- none (single locale)
- gasnet
- ugni

GASNet (universal)

Cray uGNI (Cray networks)
Runtime Communication Layer

- Supports inter-locale communication
- PUT operations
  - Single value and multiple values (strided)
- GET operations
  - Single value and multiple values (strided)
  - Blocking and (non-strided, tentative) non-blocking
- Remote fork operations
  - Run a function on some other locale
  - Uses Active Message model; normally starts a task to do the function
  - Blocking
    - Local side waits for remote side to complete; used for Chapel
  - Non-blocking
    - Local side proceeds in parallel with remote side; used internally
  - “Fast”
    - Target function runs directly in AM handler
    - Used for small target functions that will not communicate
Runtime Communication Layer Instantiations

Chapel Runtime Support Library (in C)

- **none** (single locale)
- **GASNet** (universal)
- **Cray uGNI** (Cray networks)

- **No inter-locale communication**
  - Usable only for single-locale execution
Runtime Communication Layer Instantiations

- Highly portable
  - Supports a variety of conduits, the low-level communication technology
  - UDP, MPI, many others (16 in GASNet 1.20)
- Good performance
- Default in most cases
Runtime Communication Layer Instantiations

Chapel Runtime Support Library (in C)

- Very good performance on Cray hardware
  - Especially for applications limited by remote communication latency
  - But could still be improved
- Default with prebuilt Chapel module on Cray systems
Runtime Tasking Layer

Chapel Runtime Support Library (in C)

- Tasking
  - none (serial)
  - fifo
  - muxed

- Threading
  - minimal
  - pthreads
  - soft-threads

- POSIX Threads

- Synchronization
  - Qthreads Tasks (Sandia)
  - Massive-Threads (U Tokyo)
Runtime Tasking Layer

- Supports parallelism
- Local to a single locale

Operations
- Create a group of tasks
- Start a group of tasks
- Start a “moved” task
  - Used to run the body of a non-fast remote fork, for an on
- Synchronization support (sync and single variables)

Threading layer
- Aimed to separate Chapel tasking from underlying threading
- Built an interface and a few instantiations
  - Interface known only to the tasking and threading layers (fortunately)
- Didn’t turn out so well
  - Third-party tasking layers have internal threading interfaces already
  - Threading turned out to be hard to generalize, especially with performance
- Leaning toward removing this as a separate interface
Runtime Tasking Layer Instantiations

- **Chapel Runtime Support Library (in C)**

  ![Diagram]

  - **Tasking**
    - **none (serial)**
    - **fifo**
    - **muxed**
    - Qthreads Tasks (Sandia)
    - Massive-Threads (U Tokyo)

  - **Synchronization**

**Simplest tasking implementation**
- **No true concurrency**
- **Each task must run to completion without blocking**
  - Otherwise: deadlock

**POSIX Threads**
Chapel Runtime Support Library (in C)

- Chapel tasks tied to POSIX threads
  - When a task completes, its host pthread finds another to run
  - Acquire more pthreads as needed
  - Don’t ever give pthreads up
- Default in most cases

Threading

- fifo
- muxed
- Qthreads Tasks (Sandia)
- Massive-Threads (U Tokyo)
- minimal
- pthreads
- soft-threads

POSIX Threads
Chapel Runtime Support Library (in C)

- Tasks are tied to lightweight threads managed in user space
  - When task blocks or terminates, switch threads on processor
- Good performance
- Likely future default

Tasks

- none (serial)
- fifo
- muxed
- minimal
- pthreads
- soft-threads

Qthreads Tasks (Sandia)

Massive-Threads (U Tokyo)

POSIX Threads

Threading

Synchronization
Run TIME TASKING LAYER INSTANTIATIONS

Chapel Runtime Support Library (in C)

- Tasks are tied to lightweight threads managed in user space
  - When task blocks or terminates, switch threads on processor
  - Still pretty new

- none
  - (serial)
- fifo
- muxed
- Qthreads
  - Tasks
    - (Sandia)
- minimal
- pthreads
- soft-threads

POSIX Threads

Massive-Threads
- (U Tokyo)
Runtime Tasking Layer Instantiations

- **Tasks are multiplexed on threads**
  - When task blocks or terminates, switch tasks on thread
- **Threading layer manages lightweight threads in user space**
  - Small fixed number of threads per system node
- **Very good performance**
- **Default with prebuilt Chapel module on Cray systems**

- None (serial)
- Fifo
- Muxed
- Soft-threads
- POSIX Threads
- Minimal
- Pthreads
- Qthreads
- Tasks (Sandia)
- Massive-Threads (U Tokyo)
Runtime Memory Layer Instantiations

Chapel Runtime Support Library (in C)

Memory

- default
- dlmalloc
- tcmalloc

libc malloc(), free(), etc.

dlmalloc (Doug Lea)
tcmalloc (Google perftools)
Outline

● Introduction
● Compilation Architecture
● Predefined Modules
● Runtime
● Example
● Future Work
Example (from HPCC RemoteAccess)

```chapel
var T: [TableSpace] atomic elemType;
...
forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
```

Do atomic `xor` updates to random elements of an array that spans locales. `Updates` is a `distributed domain`, representing iterations and where they should run. `RAStream()` is a stream of random numbers. `Zippered iteration` combines these, pairwise, into a `tuple` per iteration. (Then: toss the iteration; use the random number.)
Example (distribute the work)

Assume in general \#updates >> \#locales, and thus we can run several tasks per locale and still do many updates per task.
Example (distribute the work globally)

```chapel
coforall loc in Locales do on loc do // across locales
  forall (_, r) in localUpdates, RAStream() do // single locale
    T[r & indexMask].xor(r);
```

(corresponding runtime calls)

```chapel
for (int i = 0; i < numLocales; i++)
  if (i != myLocale)
    chpl_comm_fork_nb(i, onWrapper, forkArgs);
onFn1TaskBody(taskArgs);
```

This is such a common code idiom that we don’t actually create local tasks to do the on statements; we just launch remote tasks directly.
Example (distribute the work locally)

```c
coforall loc in Locales do on loc do // across locales
    forall (_, r) in localUpdates, RAStream() do // single locale
        T[r & indexMask].xor(r);
```

(for corresponding runtime calls)

```c
for (int t = 0; t < nLocalTasks; t++)
    chpl_task_addToTaskList(taskBodyFn, taskList, // make descriptors
                             taskArgs);
    chpl_task_processTaskList(taskList); // initiate tasks
    chpl_task_executeTasksInList(taskList); // ensure started
    barrierAfterCoforall;
    chpl_task_freeTaskList(taskList); // cleanup
```

```c
void taskBodyFn(taskArgs) {
    for (int i = 0; i < nTaskIters; i++) T[r & indexMask].xor(r);
}
```

Here we are creating the tasks that will do all the local iterations.
Example (do the updates)

```
forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
```

Compiler must find a definition in the internal modules for an `xor()` method on atomic data. As it turns out, there are more than one.
Example (do updates, no network atomics)

```chapel
forall (_, r) in zip(Updates, RAStream()) do
t[r & indexMask].xor(r);
```

```chapel
inline proc xor(value:int(64),...):int(64) {
  on this do atomic_fetch_xor_explicit_int_least64_t(_v, value, ...);
}
```

modules/internal/Atomics.chpl

If we don’t have network atomic support, then we do an on to move execution to the locale that owns the data, and do the update using a processor atomic operation.
Example (do updates, no network atomics)

The originating locale uses the comm layer to do a blocking remote fork. The remote locale’s Active Message handler creates a task to run the body of the `on`. That task does the user’s work, then sends a completion acknowledgement to let the fork on the originating locale proceed. (Note: we might actually use a “fast” fork here.)
Example (do updates, with network atomics)

```chapel
forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);

inline proc xor(value:int(64)):int(64) {
    var v = value;
    chpl_comm_atomic_xor_int64(v, this.locale.id:int(32), this._v, ...);
}
```

modules/internal/comm/ugni/NetworkAtomics.chpl

If the network can do atomics (and the communication layer supports them), then it’s simpler. Just call the communication layer directly to do the `xor` in the network, given the operand and the atomic datum’s remote locale and address there.
Outline

- Introduction
- Compilation Architecture
- Predefined Modules
- Runtime
- Example
- Future Work
Future Work

**Currently working on hierarchical locales**
- To support hierarchical, heterogeneous architectures such as NUMA nodes, traditional CPUs with attached GPUs, many-core CPUs
  - Adds (sub)locale-aware memory management
  - Sublocale task placement
- New architecture internal module will read an architectural description
- Compiler-emitted memory and tasking calls will go to module code.
  - Though for some architectures will effectively collapse to direct runtime calls at user program compile time.

**Other things we hope to get to soon**
- Task teams (for collectives, etc.)
- Eurekas (for short-circuiting searches, etc.)
- Task private data
Questions?