Opportunities for Integrating Tasking and Communication Layers

Dylan Stark
Brian Barrett
My Objectives for this Talk

1. Review how Chapel operates over multiple locales

2. Describe our unified runtime attempt

3. Talk about opportunities for Chapel to benefit from such an approach
Chapel Compilation Architecture

Chapel During Run Time

- Runtime initialization
- Data movement
- Work Migration
Parallel Job Launch

- (Skipping the details)
- SPMD to the runtime
- OS Process == Locale

- Start with Chapel-defined main() defined in `runtime/src/main.c`
Comm. Layer Initialization

- (CHPL_COMM=gasnet)
- Shim calls chpl_comm_init()
- Registers active message

handlers
- Sets up shared memory segments
Task Layer Initialization

- (CHPL_TASKS=qthreads)
- Shim calls chpl_task_init()
- Gathers information about the local resources and application requirements
- Forks a Pthread for Qthreads
Task Layer Initialization

- Qthreads is initialized in aux. Pthread context
- Number of worker threads equals number of cores
- Control returns to main Chapel RS thread
Progress Engine Start Up

- Another Pthread for a progress engine
- Loop polling GASNet
- `chpl_task_yield()` converted to OS `sched_yield()`
Application Initiation

- Compiler-generated chpl_main() called to start application code
- Spawned as a task into the tasking layer (from outside)
- Caller “suspends” waiting for that task (really a Pthread mutex block)
**Observations from Runtime Initialization**

- Could do better at managing compute resources
- Calls from to the tasking layer from outside of tasks can have asymmetric performance characteristics
**Data Movement**

- Put and get operations are implemented in the comm. layer
- Direct mapping to GASNet
- Of note: core **blocked** during operation
Work Migration

- 3 types: blocking, non-blocking, and “fast” remote fork
- Calling task loops – polling GASNet for completion and yielding
- Scheduler interference on the call side
A Unified Runtime Example

- **Qthreads: Lightweight threading interface**
  - Scalable, lightweight scheduling on NUMA platforms
  - Supports a variety of synchronization mechanisms, including full/empty bits and atomic operations
  - Potential for direct hardware mapping

- **Portals 4: Lightweight communication interface**
  - Semantics for supporting both one-sided and tagged message passing
  - Small set of primitives, allows offload from main CPU
  - Supports direct hardware mapping

- **Kitten: Lightweight OS kernel**
  - Builds on lessons from ASCI Red, Cplant, Red Storm
  - Utilizes scalable parts of Linux environment
  - Primarily supports direct hardware mapping

### Applications

<table>
<thead>
<tr>
<th>Applications</th>
<th>OpenMP</th>
<th>Chapel</th>
<th>MPI</th>
<th>SHMEM</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable Parallel Runtime (SPR)</td>
<td>Qthreads</td>
<td>Portals</td>
<td>Kitten</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Advanced Architectures Testbeds

- Conv. Sys.
- Simulator
Task & Network Runtime Init.

Process 0

Process 1
Task & Network Runtime Init.

Process 0

Process 1

task magic happens

task magic happens
Progress Engine Start Up

Process 0

Process 1

task magic happens

task magic happens
Application initialization

Process 0

"main task"

task magic happens

Process 1

task magic happens
Data Movement in the SPR

- Blocking and non-blocking put and get operations
- Calling task suspends, only resumes after completion event
- Progress engine only responsible for FEB operation
Work Migration in the SPR

- Added qthread_fork_remote(..., rank)
- Remote synchronization managed through FEB semantics
- Messaging using memory pooling
Chapel with a Unified Runtime

- Replaced Qthreads & GASNet with SPR (Qthreads + Portals4)
  - Single point for initializing both platforms: spr_init(SPMD,...)
  - spr_unify() used to transition to single thread of control before application starts
  - Most other interface functions are no-ops (e.g., chpl_task_init(), chpl_comm_post_task_init(), chpl_comm_rollcall(), ...)
  - Direct mappings for data movement and work migration
Chapel with a Unified Runtime

- Both layers now share ...
  - Platform information discovery (to make room for progress engine)
  - Memory management (for activation records, stacks, network packets)
  - Synchronization mechanisms (such as full-empty support)
Chapel with a Unified Runtime

- But just an early **point design**
  - Could have been MPI, MassiveThreads, SHMEM, etc.
  - Could replace progress engine with prioritized tasks
  - Could have optimized for particular hardware
  - Could have ...
Opportunities Moving Forward

- Let third-party implementers worry about
  - Information management (for incr. platform complexity)
  - Coordinated resource management (1 PE today, ? tomorrow)
  - Integrated local and remote task management (beyond command +payload, optimized for new hardware, task/message aggregation)

- Consider that the runtime options are plentiful and just as independent as the application space
Opportunities Moving Forward

- Reorient Chapel Runtime Support shim interface around unified “locality engine” (CHPL_LE=?)
  - Resist early (de facto) standardization
  - Focus on telling runtime what is needed/expected (declarative not imperative)
  - Open up runtime ecosystem to the increasing assortment of unified runtimes (one size won’t fit all)
  - Add coarse-grain “Chapelle” interface (multi-resolution runtime layers?)

- Start a runtime-centric working group to coordinate efforts between compiler writers and RS implementers