Hewlett Packard Enterprise

Chapel Runtime Overview

The Chapel Programming Language

February 12, 2025

John H. Hartman
john.hartman@hpe.com

Overview

- Organized into “layers”
- Written in C
- **Memory**
  - **jemalloc**, cstdlib
- **Computation** (tasks)
  - **qthreads**, fifo
- **Communication** ("on", RMA, atomics)
  - ofi, gasnet, ugni, none
- GPU
  - nvidia, amd, cpu
- Storage
  - qio
- **Launchers**
  - slurm-*, pbs-*, gasnet-*, smp, etc.
  - **OOB**: out-of-band communication

Popular Platforms Supported

- HPE Cray EX
- Cray XC
- HPE Apollo (InfiniBand)
- AWS
- Linux
- Mac
- SLURM, PBS

Building the Runtime

- Build controlled by `CHPL_*` variables, e.g., `CHPL_COMM`.

- Compiled and linked into `libchpl.a`
  - Different instances of `libchpl.a` for different `CHPL_*` values
  - E.g., `build/hpe-cray-ex/llvm/x86_64/cpu-x86-rome/loc-flat/comm-ofi-debug/system/pmi2/tasks-qthreads/launch-slurm-srun/tmr-generic/unwind-none/mem-jemalloc/atomics-cstdlib/ofi/gmp-bundled/hwloc-bundled/re2-bundled/llvm-bundled/fs-none/libchpl.a`

- Statically-linked with code generated by the Chapel compiler to produce the executable
  - Proper `libchpl.a` selected by `CHPL_*` variables

Managing Memory

- Support memory allocation by tasks and the runtime itself

- Tasks’ heap and stack must be accessible via RMA in a multi-locale program
  - requires them to be registered with the network fabric
  - some comm layers require allocating them from a fixed, pinned region for the entire locale (process)

- jemalloc
  - fast alloc/free, not so good on fragmentation
    - fragmentation is an issue for pinned heaps required by some comm layers
    - in general, it’s a tradeoff between speed and fragmentation

- Rely on OS "first touch” to map virtual to physical pages

Managing Computation w/ qthreads

- Lightweight user-level thread package from Sandia
- Run to completion (or yield)
- Chapel task == qthread
  - `coforall`, `begin`, etc.
- *Shepherds* run qthreads
- One shepherd per thread (pthread)
- Runtime binds each pthread to a core (or hyperthread if desired)
- Shepherd bound to pthread, qthread bound to shepherd
- qthreads assigned to shepherds in round-robin fashion
  - they are never re-assigned (no work-stealing)
  - this is important to the communication layer
- e.g.,
  - `chpl_task_addTask`

Computation Example

```
var A: [0..<10] int;
coforall a in A {
    a += 42;
}
writeln(A);
```

- The body of the loop is compiled into a function
- A task (qthread) is created for each iteration of the loop
  - use `forall` to limit parallelism
- Each task runs on the same core until it completes
- The main task blocks until all iterations complete

Managing Communication

- HPC network fabrics (InfiniBand, Aries, Slingshot)
  - high bandwidth, low latency
  - user-level access to NICs
    - protection
    - address translation
  - CPU offload
    - RMA, atomics

- Ordering guarantees and/or fences
  - E.g., GET after PUT to same address
  - either option can be expensive

- Visibility concerns
  - when will a subsequent read from memory see the effect of the write?
  - E.g., GET after PUT from a different locale

Messages

- ”Two-sided” communication
  - locales send messages to each other

- Send and Receive
  - remote locale specifies a buffer into which messages are received

- Chapel uses messages to implement *active messages*
  - message contains function to invoke and argument bundle, used to implement `on` statements
  - locale allocates buffer for received active messages
  - active message handler thread removes messages from buffer
    - “fast” active messages it handles itself
    - otherwise creates a task to invoke the function
  - Remote locale GETs argument bundle if it is too large

Active Message Example

```
on Locales[numLocales-1] {
    writeln("Hello World from locale ", here.id);
}
```

- Body of `on` is compiled into a function
- Main task sends an active message to last locale specifying which function to invoke and any arguments
- The function sets a “done” flag when it is complete
- Main task waits for the “done” flag to be set

Complex Active Message Example

```
var A: [0..<10] int;
coforall loc in Locales {
    on loc {
        A[here.id] = here.id + 42;
    }
}
writeln(A);
```

- Logically
  - Combine last two examples – functions for `coforall` and `on` bodies
- Reality
  - function for `on` body
  - main task sends active messages asynchronously to all locales
  - `on` function decrements atomic counter when it completes
  - main task waits for counter to reach zero

Remote Memory Access (RMA/RDMA)

- ”One-sided” communication
  - remote CPU is not involved
  - a locale can PUT to another locale’s memory
  - a locale can GET from another locale’s memory

- Protection via *memory registration*
  - CPU must tell NIC which memory regions are accessible to remote nodes
  - remote node must have a key (capability) to access the region
  - locales exchange registration keys during startup or on demand depending on comm layer

- NIC must do virtual address translation
  - programs refer to virtual addresses
  - ultimately, physical memory is accessed
  - introduces a lot of complexity

RDMA Example

```
var A: [0..<10] int;
coforall loc in Locales {
    on loc {
        A[here.id] = here.id + 42;
    }
}
writeln(A);
```

- Each locale will do a `GET` to fetch the initial value its element
- Each locale will do a `PUT` to write the new value of its element

Network Atomics

- Atomic operations implemented by the NIC
  - e.g., atomically increment a counter without involving the CPU

- Can't mix processor and network atomics
  - currently specified for all atomic variables via `CHPL_NETWORK_ATOMICS` setting

- Unsupported network atomics are implemented via active messages and processor atomics
  - even if only one atomic operation is unsupported
  - processor atomics are also used if there is only a single locale, i.e., `-nl 1`

Network Atomics Example

```
var a: atomic int;
coforall loc in Locales {
    on loc {
        a.add(here.id);
    }
}
writeln(a);
```

- `a.add` will result in a network atomic operation to Locale 0 (on which `a` resides)
- The NIC on Locale 0 will increment `a` atomically without involving the CPU

How is Chapel Unusual?

- Multiple cores per process
  - single process (locale) per node

- Mixture of one-sided and two-sided communication

- Large memory registrations
  - May register (almost) all physical memory

- Memory consistency model requires ordering and visibility guarantees

Launchers: Running Multi-locale Programs

- Written in C (!)
- slurm-srun is a good example
  - slurm: manages a cluster of nodes
  - srun: allocates a set of nodes if necessary and runs a program on them interactively
  - salloc: allocates a set of nodes
  - sbatch: runs a program in batch mode
- Typically, relies on a shared filesystem to distribute the executable
- Compiling `hello.chpl` for multi-locale produces two executables:
  - `hello` – invokes the launcher
  - `hello_real` – the real program
- `hello` invokes srun to launch `hello_real`:
  - `PMI_MAX_KVS_ENTRIES=20 PMI_NO_PREINITIALIZE=y HUGETLB_MORECORE=no srun --job-name=CHPL-hello --quiet --nodes=2 --ntasks=2 --cpus-per-task=256 --exclusive --mem=0 --kill-on-bad-exit /scratch2/hartman/git/chapel/devel/hello_real -nl 2 -v`

Out-of-Band Communication (OOB)

- Communication that occurs before communication layer is initialized
  - e.g., locales need to exchange addresses to communicate

- Examples of OOB communication:
  - locale network addresses
  - memory registration keys
  - barrier information

- Not unique to Chapel

- Rely on an out-of-band mechanism to share information during initialization
  - PMI2 (Process Management Interface)
    - allgather, barrier, broadcast, etc.