# Mochi affinity and threading

Mochi Bootcamp
September 24-26, 2019

## Process affinity

## Not all cores are created equal

Computer architectures are increasingly complex, particularly the nodes we typically see in HPC systems:

- ➢ Multi-core, multi-socket
- ➢ Numa nodes
- ➢ Multiple NICs, GPUs
- ➢ …

To make most efficient use of these systems, it is important to take note of the locality of these devices and to allocate resources accordingly

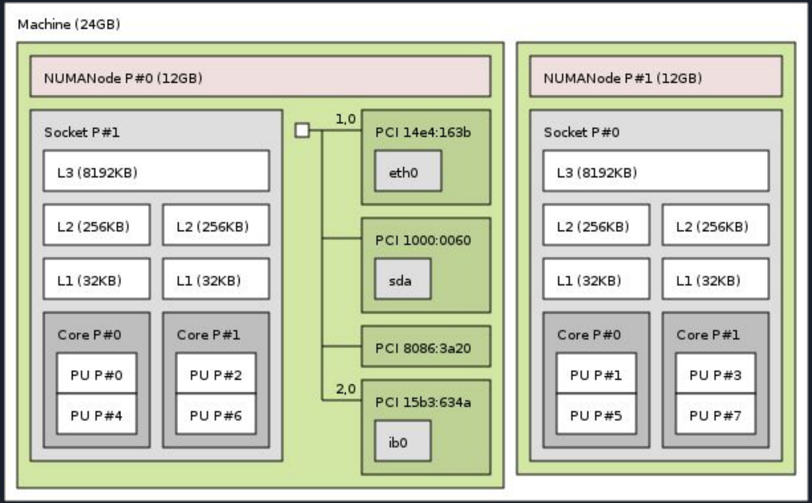
This problem is complicated by multi-threaded applications or by multiple applications/services sharing a node

## How can we learn about our target system?

*hwloc*

- ➢ Provides a portable abstraction of the hierarchical topology of modern computer architectures

Using hwloc’s ‘lstopo’ command, we can generate graphical representations of our system architecture:



## Controlling processor affinity for Mochi apps

Now that you’ve learned more about your architecture, how can you actually take advantage of it?

*Numactl*

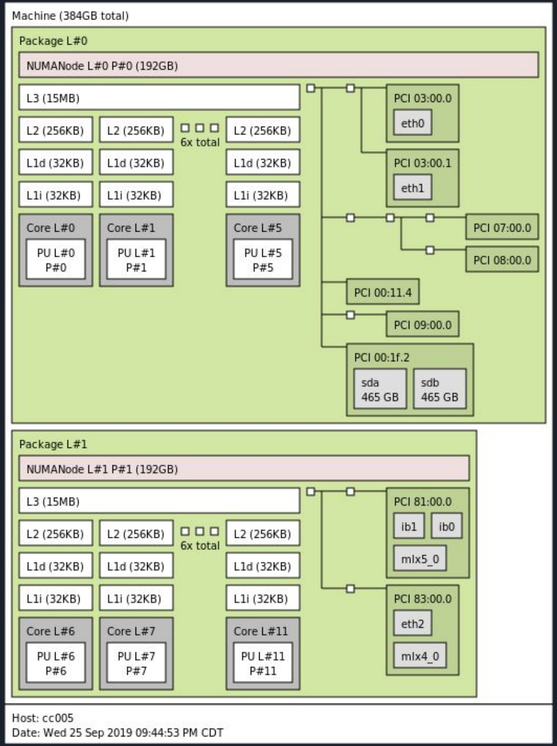
- ➢ Allows control of the NUMA scheduling and memory policies for a given executable
- ➢ Can specify which cores/sockets to run processes on, as well as which NUMA memory domain these processes allocate memory from
    - --cpunodebind=sockets, -N sockets : only execute on sockets
    - --physcpubind=cpus, -C cpus : only execute on cpus
    - --membind=sockets, -m sockets : only allocate memory from sockets

## hwloc/numactl example

Cooley Linux cluster @ ALCF, which uses an IB network

We prefer to pin  a Mochi service on socket 1, since this socket shares locality with the IB controller for this node

➢ numactl  -N 1 -m 1 <executable>



## Threading

## Threading

Getting comfortable with Argobots threading is critical to achieving desired performance under high-concurrency

Keep in mind some key Argobots terminology:

- *Execution stream (ES)* - sequential execution streams, essentially an OS thread
- *User-level threads (ULTs)* - an execution unit associated with a specific function
  - Scheduled on an ES, must yield to allow other ULTs execute on the ES
- *Scheduler* - Chooses what to execute from one or more associated pools
- *Pools* - set of schedulable work units for 1 or more ES

```
ABT_pool pool;
ABT_pool_create_basic(ABT_POOL_FIFO_WAIT, ABT_POOL_ACCESS_MPMC,
        ABT_TRUE, &pool);
```

## Threading

Getting comfortable with Argobots threading is critical to achieving desired performance under high-concurrency

Keep in mind some key Argobots terminology:

- *Execution stream (ES)* - sequential execution streams, essentially an OS thread
- *User-level threads (ULTs)* - an execution unit associated with a specific function
  - Scheduled on an ES, must yield to allow other ULTs exe
- *Pools* - set of s ES
- r - Ch mor

Automatically free pool

First-in, first-out pool with ability to wait gracefully

Units in pool can be produced on any ES and consumed on any ES

```
ABT_pool pool;
BT_pool_create_basic(ABT_POOL_FIFO_WAIT, ABT_POOL_ACCESS_MPMC,
    ABT_TRUE, &pool);
```

# Threading

Getting comfortable with Argobots threading is critical to achieving desired performance under high-concurrency

Keep in mind some key Argobots terminology:

- *Execution stream (ES)* - sequential execution streams, essentially an OS thread
- *User-level threads (ULTs)* - an execution unit associated with a specific function
  - Scheduled on an ES, must yield to allow other ULTs execute on the ES
- *Scheduler* - Chooses what to execute from one or more associated pools
- *Pools* - set of schedulable work units for 1 or more ES

```
...
ABT_sched sched;
ABT_sched_create_basic(ABT_SCHED_BASIC_WAIT, 1, &pool,
        ABT_SCHED_CONFIG_NULL, &sched);
```

## Threading

Getting comfortable with Argobots threading is critical to achieving desired performance under high-concurrency

Keep in mind some key Argobots terminology:

- ➢ *Execution stream (ES)* - sequential execution streams, essentially an OS thread
- ➢ *User-level threads (ULTs)* - an execution unit associated with a specific function
  - ○ Scheduled on an ES, must yield to allow other ULTs execute on the ES
- ➢ *Scheduler* - Chooses what to execute from one or more associated pools
- ➢ *Pools* - set of schedul

Scheduler with ability to wait gracefully

1 or more pools to schedule work from

```
...
ABT_sched sched;
ABT_sched_create_basic(ABT_SCHED_BASIC_WAIT, 1, &pool,
        ABT_SCHED_CONFIG_NULL, &sched);
```

## Threading

Getting comfortable with Argobots threading is critical to achieving desired performance under high-concurrency

Keep in mind some key Argobots terminology:

- *Execution stream (ES)* - sequential execution streams, essentially an OS thread
- *User-level threads (ULTs)* - an execution unit associated with a specific function
    - Scheduled on an ES, must yield to allow other ULTs execute on the ES
- *Scheduler* - Chooses what to execute from one or more associated pools
- *Pools* - set of schedulable work units for 1 or more ES

```
...
ABT_xstream xstream;
ABT_sched_create_basic(sched, &xstream);
```

## Threading

Getting comfortable with Argobots threading is critical to achieving desired performance under high-concurrency

Keep in mind some key Argobots terminology:

- *Execution stream (ES)* - sequential execution streams, essentially an OS thread
- *User-level threads (ULTs)* - an execution unit associated with a specific function
  - Scheduled on an ES, must yield to allow other ULTs execute on the ES
- *Scheduler* - Chooses what to execute from one or more associated pools
- *Pools* - set of schedulable work units for 1 or more ES

```
...
ABT_xstream xstream;
ABT_sched_create_basic(sched, &xstream);
```

Create an xstream associated with the given scheduler

## Providing xstreams/pools for Margo

At Margo init time, we have the opportunity to specify a couple of threading options:

Using regular margo_init:

- *use_progress_thread* - boolean value, 1 to use dedicated progress, 0 to use calling thread
- *rpc_thread_count - number of ESs to allocate for RPC handlers, 0 to use calling thread, -1 to use progress thread*

```
margo_instance_id margo_init_opt(const char *addr_str, int mode,
    int use_progress_thread, int rpc_thread_count);
```

## Providing xstreams/pools for Margo

At Margo init time, we have the opportunity to specify a couple of threading options:

Using regular margo_init_pool:

- ➢ *progress_pool* - ABT_pool to use for the progress thread
- ➢ *handler_pool* - ABT_pool to use for running RPC handlers

```
margo_instance_id margo_init_pool(ABT_pool progress_pool, ABT_pool handler_pool,
   hg_context_t *hg_context);
```

Note you need to also pass in an HG context, rather than an address string. This call is meant to provide caller most control over Margo init

## Providing xstreams/pools for Margo

At Margo RPC registration time, we can override the default handler pool we have specified at init time:

➢ Last argument is an ABT_pool to use for executing handlers for the RPC type being registered

```
...
MARGO_REGISTER_PROVIDER(mid, "operation_name", void, void,
        operation_ult, provider_id, pool);
```