Chapel 1.33 / 2.0 Release Notes: Runtime / Portability / Performance

Chapel Team
December 14, 2023 / March 21, 2024
Outline

• Co-locale Improvements
• One Billion Row Challenge
• AWS Portability and Performance
• Other Improvements
Co-locale Improvements
Co-locales
Background

- Traditional Chapel multi-locale configuration:
  - One locale per node
  - Multithreading across cores in a locale
  - One NIC per locale

- Modern hardware performs best with a process per socket or even NUMA domain
  - High cost of getting NUMA affinity wrong
  - Benefit of targeting multiple NICs using distinct processes
Generalized Co-locale
Background and This Effort

**Background:**
- Previously, supported only one co-locale configuration
  - One locale per socket
  - NICs must be in sockets

**This Effort:**
- Allow one locale per socket, NUMA domain, L3 cache, or core
  - Also support simple partitioning of cores between the co-locales
- Automatically bind locales to architectural features
  - Option “-nl 1x2” will run each co-locale in its own socket on a node with two sockets
  - Option “-nl 1x8” will run each co-locale in its own NUMA domain on a node with eight NUMA domains
  - Option “-nl 1x6” will run each co-locale on 1/6 of the cores if no architectural feature has six instances
Generalized Co-locales

Impact

**Impact:** Improved NUMA affinity

- Stream benchmark results (no communication)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>GB/s</th>
<th>Improvement</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nl 2</td>
<td>357</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>-nl 2x2</td>
<td>460</td>
<td>28.9%</td>
<td>Socket</td>
</tr>
<tr>
<td>-nl 2x8</td>
<td>466</td>
<td>30.5%</td>
<td>NUMA</td>
</tr>
<tr>
<td>-nl 2x16</td>
<td>470</td>
<td>31.7%</td>
<td>L3 cache</td>
</tr>
<tr>
<td>“first touch”</td>
<td>470</td>
<td>31.7%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- Measured on dual-socket node, Milan CPUs, 64 cores/CPU
Generalized NIC Selection
Background, This Effort, and Next Steps

**Background:** Previously, bound each co-locale to the NIC in its socket

**This Effort:**
- Bind an arbitrary number of co-locales to NICs, possibly not in sockets
- Greedy algorithm:
  
  Repeat
  
  Create distance matrix between all unbound co-locales and all NICs
  
  Repeat
  
  Bind co-locale and NIC with shortest distance
  
  Until all co-locales are bound to a NIC, or there are no more NICs
  
  Until all co-locales are bound to a NIC

**Next Steps:** Evaluate impact on communication-intensive programs
**Explicit Binding to Architectural Features**

**Background and This Effort**

**Background:**
- By default, co-locales are implicitly bound to architectural features of which there are the same number
  - e.g., “-nl 2x2” will bind each co-locale to a socket on a dual-socket machine

**This Effort:**
- Added suffixes to explicitly force the binding to an architectural feature
  - e.g., “-nl 2x6numa” will bind each co-locale to a NUMA domain, leaving any extra domains unused

- Primarily useful for testing and benchmarking, e.g. “-nl 2x1s” to run a locale in one socket
Explicit Binding to Architectural Features

Status

**Status:** 
- `-nl` argument accepts an optional suffix that specifies the binding
  - E.g., “-nl 2x8numa”

<table>
<thead>
<tr>
<th>Binding</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>s or socket</td>
</tr>
<tr>
<td>NUMA domain</td>
<td>numa</td>
</tr>
<tr>
<td>L3 cache</td>
<td>llc</td>
</tr>
<tr>
<td>Core</td>
<td>c or core</td>
</tr>
</tbody>
</table>
Co-locales
Next Steps

• Evaluate impact of co-locales on large shared-memory systems like HPE Superdome Flex

• Shared-memory bypass
  • Co-locales on the same node communicate using shared memory, instead of the network
  • Requires moderate amount of coding
  • Minimal benefit if there isn’t intra-node communication or caching

• Automatically determine ideal number of co-locales
  • Requires extensive refactoring of the launchers
One Billion Row Challenge
One Billion Row Challenge

Background:
- The One Billion Row Challenge is a "fun exploration of how quickly 1B rows from a text file can be aggregated"
  - It became viral on social media; several implementations exist in various languages
  - It avoids measuring IO overhead by first preloading data onto a RAM disk
- For our purposes, we find it more interesting and practical to use for measuring and addressing IO overhead

This Effort: create a Chapel implementation focused on readability and parallelism
- We use the 'ParallelIO' and 'ConcurrentMap' package modules
- The implementation uses a simple 'forall' loop as well as a custom aggregator and deserialization functions
- This is what the main loop looks like:

```chapel
var stats = new ConcurrentMap(bytes, tempData);
forall ct in readDelimited(fileName, t = cityTemp) with (var token = stats.getToken()) {
    stats.update(ct.city, new aggregator(ct.temp), token);
}
```
**One Billion Row Challenge**

**Impact:** the concise Chapel code performs well on a 64-core (AMD EPYC 7513) machine

- A naïve Python version takes 1390s (23m, 10s)
- A naïve, serial Chapel version takes 755s (12m, 35s)
- The parallel version further improves performance:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Time (s)</th>
<th>Time (m:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>588</td>
<td>9:48</td>
</tr>
<tr>
<td>2</td>
<td>292</td>
<td>4:52</td>
</tr>
<tr>
<td>4</td>
<td>147</td>
<td>2:27</td>
</tr>
<tr>
<td>8</td>
<td>90</td>
<td>1:30</td>
</tr>
<tr>
<td>16</td>
<td>51</td>
<td>0:51</td>
</tr>
<tr>
<td>32</td>
<td>34</td>
<td>0:34</td>
</tr>
<tr>
<td>64</td>
<td>24</td>
<td>0:24</td>
</tr>
</tbody>
</table>

**Next Steps:**

- Create a multi-node (distributed) version
- Publish blog post about this (work-in-progress)
AWS Portability and Performance
Background: Past uses of Chapel on AWS have been one-off efforts by heroic users or developers

This Effort:

- Evaluated Chapel correctness and performance with AWS ParallelCluster
  - Allows users to create their own HPC-like clusters in the cloud

- Validated Arkouda correctness with AWS Parallel Cluster

- Refreshed Chapel AWS documentation
  - Provided step-by-step guide to use Chapel and AWS ParallelCluster

- Compared performance of different AWS networks
  - Ethernet (tcp)
  - Elastic Fabric Adapter (efa)
AWS Performance

Intel 8252C (m5zn.12xlarge) vs. AWS Graviton3 (c7g.16xlarge)

- Chart 1: Stream performance with locales-per-node=1 and threads-per-node=24.
- Chart 2: Stream performance with locales-per-node=1 and threads-per-node=32.

The diagrams illustrate the performance comparison between the two architectures for different numbers of nodes, showing the GB/s achieved with tcp and efa protocols.
AWS

Performance

Intel 8252C (m5zn.12xlarge)

AWS Graviton3 (c7g.16xlarge)

isx-hand-optimized
locales-per-node=1 threads-per-node=32

isx-hand-optimized
locales-per-node=1 threads-per-node=24

Better

Number of Nodes

Number of Nodes
**AWS Performance**

Intel 8252C (m5zn.12xlarge)

- locales-per-node=1 threads-per-node=24

AWS Graviton3 (c7g.16xlarge)

- locales-per-node=1 threads-per-node=32

Graphs showing performance comparison between Intel 8252C and AWS Graviton3 under different configurations.
AWS
Next Steps

• AWS Packaging
  • Currently, the easiest way for users to use Chapel on AWS is to build from source

• Remove EFA memory restrictions
  • EFA heap registration is currently limited to 96GB per node
Other Improvements
Other Improvements

See the following sections in the `CHANGES.md` file for a full list of changes:

- Performance Optimizations / Improvements
- Runtime Library Changes
- Portability / Platform-specific Improvements
- Bug Fixes for the Runtime
- Launchers
- Developer-oriented changes: Platform-specific bug fixes
- Developer-oriented changes: Launcher Improvements
Thank you

https://chapel-lang.org
@ChapelLanguage