CMSC 714
Lecture 5
Chapel and Julia

Alan Sussman

Notes

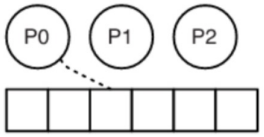
- MPI project due 1 week from Thursday, Sept. 28
  - any questions about project spec, or running on zaratan cluster?
- Readings posted through next week
- Don't forget to send questions for readings

Chapel

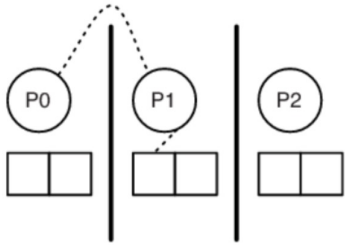
- A parallel programming language
  - a **P**artitioned **G**lobal **A**ddress **S**pace (**PGAS**) language
  - others include UPC/UPC++ (C/C++), Titanium (Java), Co-Array Fortran (part of the current Fortran standard)
- Target Environment
  - Distributed memory machines
  - Cache Coherent multi-processors (so multi-core processors)

PGAS Programming Model

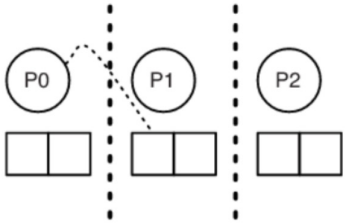
- **P**artitioned **G**lobal **A**ddress **S**pace Model
  - Provides a global view of memory across the nodes
  - Memory is still physically partitioned → local vs. remote accesses
  - But allows for a **shared-memory style of communication**



Shared-memory (e.g., OpenMP)
just "**get**" the data



Message-passing (e.g., MPI)
matching sends/receives



PGAS
just "**get**" the data

Chapel

- Characteristics
  - Goal is programmer productivity of OpenMP but functionality of MPI + OpenMP, so at scale
  - separate low-level parallelization and data distribution details from the algorithm - enable domain scientists to write efficient parallel code
  - Compiler generates communication as needed for non-local accesses
- Example – SpMV – sparse matrix-vector multiply

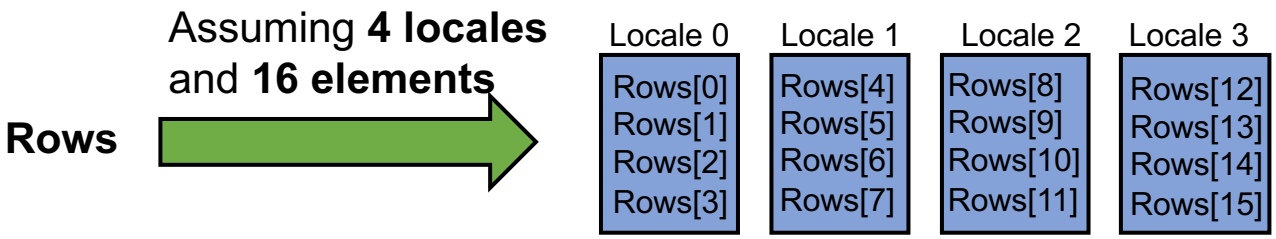
```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```

Chapel Example

- **Sparse Matrix-Vector Multiply (SpMV) - Ax=b**

```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```

- **forall** is a data parallel loop
- **Rows** is a block-distributed array of records (i.e., C structs)



Chapel Basics

- **Tasks, threads, locales, etc.**
  - **tasks:** computations that can conceptually execute concurrently
  - **threads:** mechanisms for executing parallel work
  - **locales:** unit of machine resources (e.g., cores and memory) where tasks execute
    - Usually think of locale as a compute node in a cluster
  - **domain:** represents an index set – for loops and to operate on arrays
    - Chapel supports various domain types including associative, sparse, and unstructured, in addition to ranges of integers (multi-dimensional)
  - Data parallel constructs built on top of task parallel ones:
    - Via the **begin** keyword, or **co-begin**
    - And a **co-forall** loop, where each iteration is a separate task
  - **on** clause - to specify that a statement should execute on a specific locale (the argument to the **on** clause)
- Execution model is similar to OpenMP, but more general
  - One task starts in one locale
  - Tasks created dynamically, using task and data parallel constructs

Chapel Performance

- For single-locale programs, execution is fairly competitive with hand-coded C+OpenMP
- For multiple locales, across multiple machines, depends on the communication patterns
  - For regular patterns (e.g., stencil) performance is competitive with MPI (but maybe not to very large number of locales)
  - For less regular patterns, compiler still needs a lot of optimization work
  - Underlying communication layer on most high-performance networks (e.g., Infiniband) is GASNet – one-sided communication plus active messages

Additional info

- Documentation and more information at https://chapel-lang.org/
- Current version is 1.31, from June 2023

Julia

Overview

- Julia goals: productivity and performance for numerical scientific computing
  - From “careful language design and the right combination of carefully chosen technologies that work very well with each other”
- all basic functionality must be possible to implement in Julia – no escape to C or something else lower level
- Users interact with Julia through a standard REPL (real-eval-print loop environment like Python, R, or MATLAB), by collecting commands in a .jl file, or by typing directly into a Jupyter (JUlia, PYThon, R) notebook

Language Features

- An expressive type system, allowing optional type annotations (section 3 in paper)
- Multiple dispatch using the types to select implementations (section 4 in paper)
- Metaprogramming for code generation (section 5.3 in paper)
- A dataflow type inference algorithm allowing types of most expressions to be inferred
- Aggressive code specialization against run-time types
- JIT compilation using the LLVM compiler framework, which is also used by other compilers such as Clang and Apple's Swift
- Julia's carefully written libraries that leverage the language design

Parallelism in Julia

- Multi-threading
  - able to schedule Tasks simultaneously on more than one thread or CPU core, sharing memory
  - multi-threading is composable - when one multi-threaded function calls another multi-threaded function, Julia will schedule all the threads globally on available resources, without oversubscribing
  - Can set the number of threads via command line argument, or through an environment variable – always start execution in one (main) thread
- Distributed computing
  - multiple Julia processes with separate memory spaces, on the same computer or multiple computers
  - **Distributed** standard library enables remote execution of a Julia function, using remote calls that return *futures* and remote references (of 2 types, *Future* and RemoteChannel)
  - **MPI.jl** and **Elemental.jl** provide access to the existing MPI ecosystem of libraries

Performance

- Can take advantage of multiple types of parallelism
  - SIMD instructions, multi-threading on a single node, multiple nodes, GPUs
- Performance on a single machine/node is "competitive" with C, esp. for numerical computations
  - See https://julialang.org/benchmarks/ for microbenchmarks
- Should be very efficient because of JIT compilation and multiple dispatch
  - Specialize the generated code to the actual types used for each version (combination of parameter types)
  - Generate efficient LLVM intermediate code, then rely on LLVM to generate efficient machine code
- There have been real applications ported to Julia that achieved very high performance (i.e. petaflops)
  - First example was an astronomy application – processing Sloan Digital Sky Survey (SDSS) data using the Celeste Julia code, using 1.3M threads on a DOE supercomputer
  - 178TB of image data processed in 14.6 minutes, in 2017, so about 1.5Petaflops

Summary

- For more info on Julia, see https://julialang.org/
- Current version is 1.9, from April 2023