Parallel programming in Chapel

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

JUAN ZUNIGA
juan.zuniga@usask.ca

Why another language? `http://chapel.cray.com`

- High-level parallel programming language
  - "Python for parallel programming"
  - much easier to use and learn than MPI; few lines of Chapel code typically replace tens of lines of MPI code
  - abstractions for data distribution/parallelism, task parallelism
  - optimization for data-driven placement of subcomputations
  - granular ("multi-resolution") design: can bring closer to machine level if needed
  - everything you can do in MPI (and OpenMP!), you should be able to do in Chapel
- Focus on performance
  - compiled language; simple Chapel codes perform as well as optimized C/C++/Fortran code
  - reportedly, very complex Chapel codes run at ~70% performance of a similar well-tuned MPI code (not bad, but room to improve)
- Perfect language for learning parallel programming for beginners
- Open-source: can compile on all Unix-like platforms, precompiled for MacOS (single-locale via Homebrew), Docker image `http://dockr.ly/2vJbi06` (simulates a multi-locale environment)
- Fairly small community at the moment: too few people know/use Chapel $\iff$ too few HPC centers install and promote it

Useful links

- Slides from `https://chapel-lang.org`
  - *Data parallelism*
  - *Task parallelism*
  - *Locality / Affinity Features*
  - *Domain Maps / Distributions*
- Watch *Chapel: Productive, Multiresolution Parallel Programming* talk by Brad Chamberlain
- *Getting started guide for Python programmers*
- `https://learnxinyminutes.com/docs/chapel`
- Concise *Chapel tutorial* by David Bunde
- Documentation and examples for various Chapel modules in `$CHPL_HOME/modules/`, e.g., `standard/` or `dists/`
- `https://stackoverflow.com/questions/tagged/chapel`

Our workshop

Part 1: Basic language features

- running single-locale Chapel codes on Cedar
  - interactive jobs vs. batch jobs
- quicky on running Chapel on your laptop
- problem description: heat transfer equation
- variables
- ranges and arrays
- conditionals
- `for` loops
- config variables
- timing code execution

See lesson notes

Part 2: Task parallelism

- parallel concepts
  - concurrency vs. true parallelism
  - concurrency vs. task locality
- fire-and-forget tasks
  - `begin` statement
  - `cobegin` statement
  - `coforall` loops
  - `forall` loops
- task synchronization
  - `sync` statement
  - sync variables
  - atomic variables
- task-parallelizing the heat transfer solver (if we have time)

See lesson notes

Part 3: Domain parallelism

- running multi-locale Chapel codes on Cedar
- simple multi-locale codes
- domains and single-locale data parallelism
- distributed domains
- heat transfer solver on distributed domains
- periodic boundary conditions
- writing to files

See lesson notes

Numerical problem: 2D heat transfer equation

- Imagine a metallic plate initially at 25 degrees
- Simple 2D heat (diffusion) equation

$$\frac{\partial T(x,y,t)}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

- Discretize the solution $T(x,y,t) \approx T^{(n)}_{i,j}$ with $i = 1, ..., \text{rows}$ and $j = 1, ..., \text{cols}$
  - upper left corner is `(1,1)`, lower right corner is `(rows,cols)`
- Initial condition: $T^{(0)}_{i,j} = 25$
- Boundary condition: upper side $T^{(n)}_{0,1..\text{cols}} \equiv 0$, left side $T^{(n)}_{1..\text{rows},0} \equiv 0$, bottom side $T^{(n)}_{\text{rows}+1,1..\text{cols}} = 80 \cdot j/\text{cols}$, right side $T^{(n)}_{1..\text{rows},\text{cols}+1} = 80 \cdot i/\text{rows}$ (linearly increasing from 0 to 80 degrees)
- Discretize the equation with forward Euler time stepping

$$\frac{T^{(n+1)}_{i,j} - T^{(n)}_{i,j}}{\Delta t} = \frac{T^{(n)}_{i+1,j} - 2T^{(n)}_{i,j} + T^{(n)}_{i-1,j}}{(\Delta x)^2} + \frac{T^{(n)}_{i,j+1} - 2T^{(n)}_{i,j} + T^{(n)}_{i,j-1}}{(\Delta y)^2}$$

Numerical problem: 2D heat transfer equation (cont.)

- For simplicity assume $\Delta x = \Delta y = 1$
- Use $\Delta t = 1/4$
- The finite difference equation becomes

$$T_{i,j}^{(n+1)} = \frac{1}{4}\left[T_{i+1,j}^{(n)} + T_{i-1,j}^{(n)} + T_{i,j+1}^{(n)} + T_{i,j-1}^{(n)}\right]$$

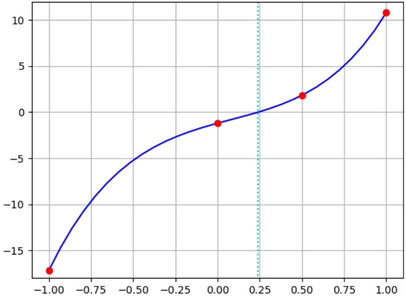
- The objective is to find $T_{i,j}$ after a certain number of iterations, or when the system is in steady state
- Can increase the number of points in the grid to illustrate the advantage of parallelism

Serial exercise: using *procedures* and control flow

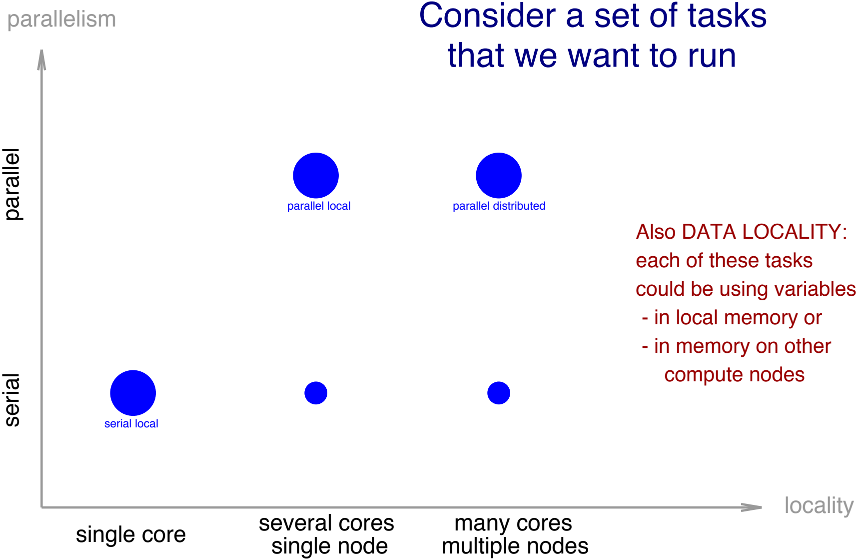
Look up Chapel procedures

Write a Chapel code to find the root of the equation $x^5 + 8x^3 - 2x^2 + 5x - 1.2 = 0$ using the bisection method in the interval [-1,1]

- Calculate the function at the ends and the midpoint of the interval
- Depending on the signs of the three computed values, let the midpoint be either the new left or the new right end
- Repeat until your error is below $\Delta x = 10^{-8}$



Parallelism vs. TASK LOCALITY



Consider a set of tasks that we want to run

Also DATA LOCALITY:
each of these tasks
could be using variables
- in local memory or
- in memory on other
compute nodes

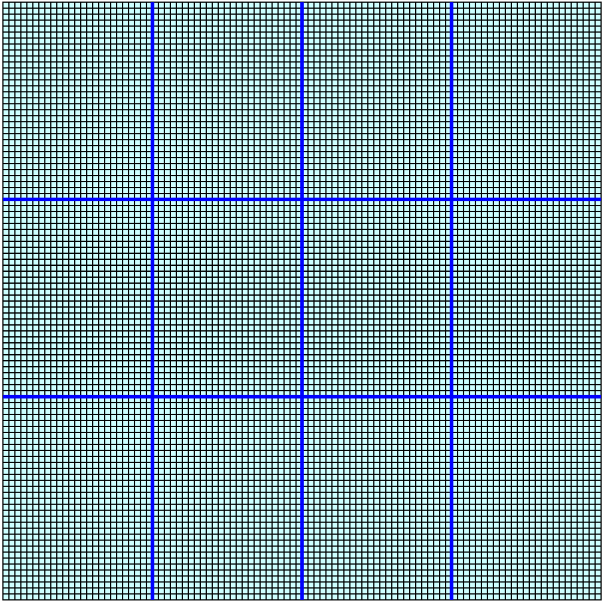
Task- vs. data-parallel

	single locale shared memory parallelism	multiple locales distributed memory parallelism likely shared memory parallelism

task parallel	`config var numtasks = 2;`
	`coforall taskid in 1..numtasks do`
	`  writeln("this is task ", taskid);`
		`forall loc in Locales do`
		`  on loc do`
		`    writeln("this locale is named ", here.name);`

data parallel	`var A, B, C: [1..1000] real;`
	`forall (a,b,c) in zip(A,B,C) do`
	`  c = a + b;`
		`use BlockDist;`
		`const mesh = {1..100,1..100} dmapped`
		`  Block(boundingBox={1..100,1..100});`
		`var T: [mesh] real;`
		`forall (i,j) in T.domain do`
		`  T[i,j] = i + j;`

Array decomposition



Race condition

Task 1	Task 2
lock.add(1)	
lock.waitFor(2)	
"task 1 is done"	lock.add(1)
lock.sub(1)	lock.waitFor(2)
lock.waitFor(0)	

Note: lock.waitFor() is not a collective operation

Data-parallel exercise: compute π with `forall` loop

Write a parallel Chapel code to compute π by calculating the integral numerically through summation

$$\pi = \int_0^1 \frac{4\,dx}{1+x^2}$$

Parallelism cheatsheet

- `for` is a serial loop; `a..#n` means n iterations, `a..b` means b-a+1 iterations
- `forall` loop is executed cooperatively by all local cores in parallel, or by remote locales that own the corresponding indices/elements (subdividing their local iterations among their local cores); number of threads scales to the number of available cores
- `coforall` loop creates a new task per each iteration (cycling through locales or tasks inside a locale)
- `begin { ... }` spins statements inside off into a new task
- `sync { ... }` pauses until the children have synced back up
- `cobegin { line1 line2 line3 }` runs each line in a new task; can be grouped with {}
- Built-in variables and arrays
  - `numLocales` is the number of locales
  - `Locales` stores an array of compute nodes on which the program is executing
  - `locale.id` is the ID of the current locale
  - `locale.maxTaskPar` is the runtime maximum number of tasks on the current local
  - `locale.numCores` is the locale's number of compute cores
  - `locale.name` is a locale's name
  - `here` evaluates to the locale on which the current task is running
- Distributions
  - `BlockDist` partitions indices into blocks according to a boundingBox domain and maps each block onto a separate locale
  - `CyclicDist` maps indices to locales in a round-robin pattern starting at a given index
  - `BlockCycDist`, `DimensionalDist2D`, `PrivateDist`, `ReplicatedDist`, `StencilDist`, `BlockCycDim`, `BlockDim`, `ReplicatedDim`

Distributed domains