1D Heat Equation in Chapel

Jeremiah Corrado
HPE

jeremiah.corrado@hpe.com

Daniel Fedorin
HPE

daniel.fedorin@hpe.com

***Abstract*—This paper describes an assignment in the Chapel programming language for creating a 1D heat equation solver. Two methods are used to solve the problem, exposing a variety of parallel programming concepts. The first portion of the assignment uses high-level parallel constructs, namely Chapel's `forall` loop and `Block` distribution, to create a simple distributed-memory solver. Here, students are asked to think about what it means for an array to be split across the memory in multiple compute nodes while relying on the language to handle the details of communication and synchronization. The second portion of the assignment uses low-level parallel constructs such as Chapel's `coforall` loop (used to manually spawn threads), `barriers`, and explicit communication. Here, the goal is to create a more efficient solver by reducing overhead, while also introducing students to the ideas of explicit communication and synchronization. In both parts, students are provided with a non-distributed version of the solver and are asked to create a modified version that runs across multiple compute nodes. See the full materials for the assignment at this link: https://github.com/jeremiah-corrado/Chapel-Heat1D-PPA**

***Index Terms*—chapel; heat equation; distributed**

I. Motivation for Assignment

The heat equation is a simple partial differential equation (PDE) that can be solved using the finite difference method. It is highly amenable to parallelization and distribution while exposing enough complexity to motivate interesting discussion of parallel programming concepts.

Here, we will solve the PDE using the Chapel Programming Language. Chapel is a modern general-purpose language designed to make parallel and distributed programming highly productive while achieving performance similar to other HPC stacks such as C++/MPI/OpenMP. As such, it allows us to introduce some intermediate level distributed/parallel programming concepts to beginner High Performance Computing (HPC) programmers. Specifically, this assignment is aimed at students who have some solid programming experience, are interested in scientific computing, have a need for HPC, and who are familiar with at least one high-level programming language such as Python or Matlab. Students should expect to learn about parallelizing order-independent loops, how to write distributed memory patterns, and how to reason about synchronization and communication in a multi-node setting.

II. Assignment

For the purpose of this assignment, we will use the following form of the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha * \Delta u = \alpha * \frac{\partial^2 u}{\partial x^2}$$

which can be converted into the following discretized form:

$$u^{n+1}[x] = u^n[x] + \alpha * (u^n[x-1] - 2 * u^n[x] + u^n[x+1])$$

Here, u is an array of values defined at discrete points in space ($x \in \Omega$) and time (n, $n+1$, ...). Given some initial conditions and forcing values on the edges to be static (Dirichlet boundary conditions), we will approximate u for some number of time steps using the following algorithm:

1. define Ω to be a set of discrete points on the x-axis, and $\hat{\Omega} \subset \Omega$ to not include boundary points ($\partial\Omega \not\subset \hat{\Omega}$)
2. define an array: `u` over Ω with some initial conditions
3. create a temporary copy of `u` named `un`
4. for `nt` time steps do the following:
    1. swap `u` and `un`
    2. compute `un` in terms of `u` over $\hat{\Omega}$

A. Local Parallel Solver

The associated file: `Example1.chpl`, has a direct implementation of this algorithm. Several key concepts are introduced: configurable constants (*line 2*), domains (*line 7*), arrays (*line 11*), and forall loops (*line 26*).

An important aspect of this computation is that `un` can be computed in parallel, as each of its values depends strictly on the previous time step's values. In Example 1, we use a `forall` loop to automatically split step 4.2 across multiple tasks. Chapel's runtime will execute tasks on all the available cores concurrently. In a distributed setting, the `forall` loop can also be used to parallelize computations across multiple compute nodes and across the cores of each node with a single loop.

Students can look at the associated file: `ChplGuide.md`, for instructions on compiling and running Chapel programs. Example 1 should produce the following output:

```
mean: 1.50024 stdDev: 0.498622
```

B. Distributed Parallel Solver

The first part of the assignment is to convert Example 1 into a distributed code, i.e., a version that can split the problem across multiple compute nodes, taking advantage of the collective memory and processing capacity of a cluster or supercomputer. To do this, we first need to introduce Chapel's concept of a distribution.

A distribution maps the entries in a `domain` (a set of indices, like Ω) to a memory layout across a group of nodes (called `locales` in Chapel). See the following snippet that uses the `Block` distribution to create a distributed 1D domain. It initializes an array `a` over that domain, where the value of each element corresponds to its index, then squares each value in parallel using a `forall` loop. Running this program across multiple compute-nodes (or locales) will split `a` into evenly-sized contiguous "blocks" in the memory across those nodes. The computation on each block will be executed on their respective locales.

```
use BlockDist;
config const n = 1000;
const D = Block.createDomain({0..<n});
var a = [i in D] i;
forall i in D do a[i] *= i;
```

Assignment: Using these concepts and `Example1.chpl`, fill in the blanks in `Assignment1.chpl`. When compiled and run, the code should produce the same output as the single-locale version, but run faster on a multi-node machine.

C. Explicit Task-Parallel Solver

One drawback of the above approach is that new tasks are created and destroyed by the `forall` loop at each time step, which incurs a some overhead. To avoid this, we'll introduce a modified version of the code that creates a set of tasks at the start of the program and re-uses them throughout.

The associated file, `Example2.chpl`, makes use of several new concepts, namely: barriers[1] (*line 28*), `coforall` loops (*line 31*) procedures (*line 38*), array slices (*line 51*), range slices (*line 45*), and foreach loops (*line 72*). A barrier is used to manage synchronization among tasks — something the `forall` loop did for us automatically. The `coforall` loop spawns exactly one task per iteration (unlike the `forall` loop that evenly splits work across available cores). A procedure (`taskSimulate`) is used to abstract away the computation for a single task. Array and range slices are used to copy the initial conditions into each task's local array. The `foreach` loop is used to express order-independent parallelism without creating new tasks.

In addition synchronizing via the barrier, we also need to manage the sharing of edge-values between tasks that own neighboring regions of the global array. `Example 2` does this by creating a global array of "halo" cells. At each time step, tasks store the values along their edges in their neighbors' halo cells. They then copy the neighbors' values into their own local array. See `Diagram 1` for a detailed visual description of this pattern.

[1] A construct used to synchronize program execution among a group of threads (called tasks in Chapel)

D. Explicit Distributed Task-Parallel Solver

The next assignment is to convert `Example 2` into a distributed code. To do this, we need to understand how to specify where a computation should be executed (`taskSimulate` in our case) and where memory should be allocated (`uLocal1` and `uLocal2`). In Assignment 1, locality was specified behind the scenes by the `Block` distribution's implementation. For this example, we'll need to use an `on`-statement to explicitly designate which locale each task should run on, and by extension, where each of its declared variables should be stored in memory.

See the following code snippet that concurrently executes 10 tasks on each locale in the global `Locales` array (an array of all the locales available to the program). The outer `coforall` loop, creates one task per locale, and uses an `on`-statement to specify that the body of the loop should run on the given locale. The inner loop spawns 10 more tasks, declares an array, and prints out a message.

```
coforall loc in Locales do on loc {
  coforall tid in 0..<10 {
    var a: [1..5] int = loc.id * tid;
    writeln("Task ", tid,
        " of 10 on Locale", here.id,
        ". 'a' is on locale: ", a.locale.id);
  }
}
```

Note that `a.locale.id`, which returns the ID of the locale where `a` is located, will always match `here.id` (the ID of the locale where the code is executed).This program would print out one line for each task (ten times the number of locales) in any order, where each line would look something like:

```
Task 4 of 10 on Locale 2. 'a' is on locale: 2
```

Assignment: use Example 2 and the above concepts to fill in the blanks in `Assignment2.chpl`. Keep in mind that the `halos` array should also be distributed so that each task's halo cells will be stored on the same locale where the task is running. If it were not distributed, each access into that array would involve communication with locale 0 (instead of the neighboring locale), creating a communication bottleneck. When the completed program is run, it should produce the same output as the other programs and should run faster than them on a multi-node machine. Up to a point, it should also run faster as more nodes are used for the computation.