PRACTICAL MATTERS

Debugging

While developing the programs for this tutorial, we've learned several tricks for debugging that might come in handy: how to run a Chapel program in a debugger, how to look at the intermediate C code to find compiler bugs, how to improve parallel performance (with a discussion of Chapel's tasking model), and submitting test cases for bugs and feature requests.

Exceptions and Segfaults

If a program crashes with no error message printed to the command line, then you'll need a debugger to find the cause. The compiler has command line options to help do this.

When compiling, use the `-g` and `--savec` flags. `-g` adds debugging information (similar to a C compiler). `--savec` takes a directory (which will be created if it doesn't exist) and preserves the intermediate C files for line references.

```
chpl -g --savec=/tmp/chpl -o prog prog.chpl
```

Note that this will increase the compile time.

Then when running the program, use the `--gdb` flag to start the program in GDB.

```
./prog --gdb [other arguments]
```

Within GDB, just type run. The program will halt at the error and print the stack trace with the line of the error. You can inspect the program's state normally within the debugger. The compiler may have changed the names of your symbols, as we'll see next when we look at the generated code.

Generated Code

While debugging the ransac_rst program, we ran into a runtime error complaining about memory being freed twice. We found the problem by looking at the C code that Chapel generated from the program.

You'll see in the `pick_seeds()` procedure that we build a list of potential matches three times, for all but the first seed. Before picking one, we check that there is at least one possibility. If not, we abandon the attempt.

```
nmatch = 0;
for i in rng2 {
  if are_corners_similar(corners1(ind11), corners2(i)) {
    nmatch += 1;
    matches(nmatch) = i;
  }
}
if (0 == nmatch) then return false;
```

It was in the last line that the program was crashing with the message

```
ransac_rst.chpl:192: error: halt reached - array reference count is negative!
```

where line 192 points to the definition of the `matches` array.

The compiler has two options of interest. As we said above, the `--savec` option takes a directory for saving the generated C code. You can also add `--no-munge-user-idents`. The compiler will modify your symbol names to prevent collisions with internal symbols by appending _chpl. This option will (mostly) prevent that from happening. You'll find though that even with munging the symbols are still readable.

That (0 == nmatch) test compiled to:

```
call_tmp_chpl54 = (INT64(0) == nmatch_chpl);
if (call_tmp_chpl54) {
  ret_chpl = false;
  chpl__autoDestroy5(matches_chpl, INT64(192), "ransac_rst.chpl");
  chpl__autoDestroy2(call_tmp_chpl6, INT64(194), "ransac_rst.chpl");
  chpl__autoDestroy2(call_tmp_chpl7, INT64(195), "ransac_rst.chpl");
  goto _end_pick_seeds_chpl;
}
/* much code deleted */
_end_pick_seeds_chpl:;
chpl___ASSIGN_17(try_chpl, &_formal_tmp_try_chpl);
chpl__autoDestroy2(call_tmp_chpl7, INT64(195), "ransac_rst.chpl");
chpl__autoDestroy2(call_tmp_chpl6, INT64(194), "ransac_rst.chpl");
chpl__autoDestroy5(matches_chpl, INT64(192), "ransac_rst.chpl");
return ret_chpl;
```

The pattern we see the compiler using is to jump to a label at the end of the procedure to clean-up local allocations (which is the style we talked about in the Image Interface chapter). The problem is obvious: there are two calls to chpl__autoDestroy5(matches_chpl, ...), which raises the error. Even with the symbol mangling it was easy to find the problem code by searching for the array name given in the error message. The work-around is to use nested `if`'s instead of returning so that the clean-up only occurs at the end of the procedure. You'll find this in the current version of ransac_rst.chpl.

```
if (0 != nmatch) {
  /* determine third seed */
  if (0 != nmatch) {
    /* determine fourth seed */
    if (0 != nmatch) {
      return true;
    }
  }
}
return false;
```

Tasks

A system monitor that can show the load on each (virtual) core is a good tool for checking if your program is being run in parallel. While running ransac_rst, we noticed that the corner alignment was only running on two cores, despite a `forall` in the main loop.

You'll see a difference in behavior depending on which threading package you've compiled into Chapel. The quickstart version uses pthreads, the full version qthread (plus there are a couple other third-party options that we won't discuss). pthreads is a kernel-level library: threads are created and managed by the system, which is relatively expensive. In the qthread library the threads are managed in user space. Technically Chapel doesn't see threads, it works with an abstraction called a task that maps down to threads by each library. A task is created by a `begin`, `cobegin`, or `coforall`. There is some control over the resources allocated for threads - their number and stack size, for example - but in practice you should never count on being able to affect how

threads execute, or how tasks are mapped to them.

In the pthreads approach, there is a single task pool. Threads are created as needed for the tasks (up to a configurable limit), and each thread runs its task to completion. Because threads involve system calls to create, they are not destroyed when their work is done. Instead, they look for another task to pick up. Thus, tasks will start quickly if there is an idle thread available, or less so if one must be created. If a task blocks (on a synchronized variable, for example) then the system scheduler idles its thread. Each thread guards against stack overflow by adding a guard page at the end of its stack that will raise an error if accessed.

For the qthreads library, the task pool is split into different queues. The library creates two kinds of threads, workers for task execution and shepherds for distributing work. When a task blocks the shepherds can re-assign a worker to another task. This is done in user space. There is one worker per core, and often multiple workers per task queue. qthreads also uses a guard page at the end of the stack to guard against overflows.

In practice the qthreads library performs better than the pthreads. There may be a higher cost at start-up because it creates more worker threads initially, but the pthread-by-pthread startup based on task demand may lead to a slower overall program. If the system creates many pthreads, then they will hang around waiting for new tasks, and this can lead to a deceptively high system usage as they poll for new work. You can see this if your system monitor distinguishes between the time spent in system calls (pthreads overhead) or userland (actual work). We have seen higher loads spread evenly across the cores with pthreads than qthreads so that it looks like more work is being done, but the program still runs slower.

The Chapel development team recommends splitting work coarsely if possible, with big loops, to minimize the thread overhead. They also suggest not using a `coforall` with many more iterations than the number of cores, because this just creates tasks that will have to wait for resources to free up. A `forall` is in general a better choice. Make sure the hardware configuration is in line with the runtime environment - don't create more locales than compute resources, for example. Until you need to fine tune a large application, there's no need to play around with the many environment variables and command line options that the libraries and Chapel support to tweak the number of threads or stack size. Details about these options are scattered through many files in CHPL_HOME/doc, like executing.rst and tasks.rst.

With the help of Greg Titus at Cray we were able to track down the problem and find a work-around. The top level matching loop in the program looked like:

```
proc align_corners(const corners1 : [] corner, const corners2 : [] corner,
                   out bestmap : mapinfo,
                   map1to2  : [] int, map2to1 : [] int) {
  var tries : [1..ntry] tryinfo;
  var rand :  new RandomStream(real);
  const minlen = min(corners1.domain.dim(1).last, corners2.domain.dim(1).last);
  const mincnt = nearbyint(matchfrac * minlen) : int;
  var tree = new kdtree(corners2.domain.size, int, 2);

  /* Build the kd-tree for quick lookup of mapped corners during match. */
  for i in corners2.domain {
    tree.add_node(corners2(i).center, i);
  }
  tree.assemble_tree();

  forall i in 1..ntry {
    do {
      if pick_seeds(corners1, corners2, rand, tries(i)) {
        var mapped1 : [corners1.domain] corner;
        map_corners(corners1, tries(i).map, mapped1);
```

```
          const matchcnt = count_matches(mapped1, corners2, tree);
          if (mincnt <= matchcnt) {
            tries(i).nmap = matchcnt;
            break;
          }
        }
        tries(i).nfail += 1;
      } while (tries(i).nfail < nfail);
    }

    /* Regenerate the best mapping and refine it with a linear regression.
       This is the final result. */
    const besttry = select_besttry(tries);
    var mapped1 : [corners1.domain] corner;
    map_corners(corners1, tries(besttry).map, mapped1);
    match_corners(mapped1, corners2, tree, map1to2, map2to1);
    refine_mapping(corners1, corners2, map1to2, map2to1, bestmap);

    delete rand;
    delete tree;
  }
```

(This code compiled in Chapel 1.12. Later versions require some changes, for example for the random number generator. We haven't checked if these fixes are still necessary.)

We started by looking at a histogram of the timing of each trial, which is measured and stored in the tryinfo record. We noticed that although most trials took on the order of seconds to run, there were two that waited hundreds of seconds (and two that took tens of milliseconds). We do not expect orders of magnitude difference in the times.

We converted the loop to a work queue, which has a fixed number of worker threads that start a trial when they're free. The `coforall` guarantees the number of workers. The `nextIter` variable is incremented as each trial starts, making sure we cover the number specified.

```
  var nextIter : atomic int;
  nextIter.poke(1);
  coforall worker in 1..here.maxTaskPar {
    var i = nextIter.fetchAdd(i);
    while (i <= ntry) {
      /* perform trial */
      i = nextIter.fetchAdd(i);
    }
  }
```

`here` is a constant that Chapel creates. It is the locale in which the current task (the function call in this case) is running. `maxTaskPar` is a member of the locale class with the maximum parallelism available at the site.

This too only ran on two cores under qthreads, however. With the pthreads library it would use all four cores, albeit with a higher system load. This implies the problem lies with the threading library, which is not something we could debug. The timing still showed a large spread in values. With the work queue we could count how many trials each task in the `coforall` executed. Three were balanced, but one ran only one pass. That thread was blocking until the others finished all the trials. The next step was to isolate what was causing the block. We scattered timing statements throughout the loop and found the `pick_seeds()` call was where it would freeze. Checking the timing of `pick_seeds()` almost line by line, we found that the call to the random number generator was what was blocking.

We made one change to `align_corners()`, to move the random number generator into the loop. In principle this shouldn't be necessary; it's supposed to be parallel safe. But something in its implementation, perhaps an interaction with the synchronization variable the generator uses as a lock and the thread scheduler, causes the qthreads library to block. Because pthreads are scheduled by the kernel, we would expect their behavior to be different. The change was made in the process of trying different code paths, not deliberately. The loop you'll find in the program now is

```
forall 1..ntry {
  var rand = new RandomStream();
}
```

with everything else in the procedure unchanged. This costs an instantiation of the random number generator per trial, but avoids contention for its lock, and this restores the correct parallel performance. As with all the problems we've found, we submitted the problem to the development team and continue to use the work-around. (We haven't checked if the behavior has changed with the latest release.)

Bug Submission and Test Cases

Chapel is hosted on Github and uses its issue tracker for bugs. The New Issue button opens a ticket. You should provide a summary of the problem, how to reproduce it, and your Chapel setup and environment. See CHPL_HOME/doc/rst/usingchapel/privatebugs.rst for the guideline. You can also prepare a test case that becomes part of the nightly test run.

Procedurally this is a bit involved. Because the test case will become part of the release, you'll be asked to sign a Contributor's Agreement. The Chapel repository is found on Github at

https://github.com/chapel-lang/chapel

You'll notice it has several directories that do not appear in the release version. The agreement is found in one, at doc/developer/contributerAgreements. There are two versions, one for individuals and one for companies.

Once the agreement has been signed and sent to the development team, you'll need a Github account to be able to check in code. Make a clone of the master branch on your local machine. You'll find complete instructions in CHPL_HOME/developer/bestPractices/ContributorInfo.rst

Chapel uses a test system that automatically runs code samples it finds. In the development version, you'll find the test cases in CHPL_HOME/test. Many of these are copied into CHPL_HOME/examples for the release version. Not only does the system verify the correct behavior of the language and bug fixes, but it can also track the performance of programs and is used to track feature requests. The main test script scans the top directory recursively looking for files with a .chpl suffix. A test contains at least two files: a program to run and the output it should generate.

<file>.chpl - test program

<file>.good - expected output

The test system compiles the program, runs it, and compares the actual output to the expected. Mismatches count as a failure. If your program requires options to the compiler, anything other than a `chpl -o <tmpfile> <file>` command, then you can put them in specification files with the options on one line. If your program requires options to execute, they go in another file.

<file>.compopts - command-line options to compiler

<file>.execopts - command-line options to run program

The system also supports a directory-wide, instead of per-program, specification. For these two files the global versions are COMPOPTS and EXECOPTS. We prefer using the per-test version.

To create a test case, you'll need a place to put it in under the test directory. The sub-directory CHPL_HOME/test/users/<your name> seems to be the preferred location for your sample program and good output. Then run the program `start_test` (found in CHPL_HOME/util) on your directory. You should run `start_test` both with and without the `-performance` option; the nightly tests run with both. Once you've verified the test case is correct, sync your branch with Github and submit a pull request to the development team. They will verify the change and test case, ask you for edits if any are needed, and will finally do the merge with the master branch.

The development team also uses this system for feature requests. In addition to the two files above, you'll need to submit a description of the request. The first line of <file>.future must contain the type of change (feature, bug, edit to an error message) and a succinct summary. The rest of the file can contain details about the proposal. If the change raises a compiler or runtime error, then you can include another file to capture that. This will catch other changes to Chapel that break the incorrect behavior we currently have.

<file>.future - explanation of feature request

<file>.bad - output generated on a failing test

Note that the output in <file>.bad may contain information specific to a release: line numbers inside the compiler, for example. If you add a script <file>.prediff, it will be run on the output before comparisons are made. This might include sed commands to remove variable line. Make sure the file is flagged executable.

<file>.prediff - pre-processing of output before comparison to .good/.bad

Feature requests are not included in the daily regression testing. The team monitors them separately.

The test system can also be used to monitor the performance of programs. The system will examine the program's output for keywords or strings. A value after the keyword/string will be extracted from the output and appended to a file that tracks the value over time. There is support for graphically viewing the progress in a browser, as described in the documentation of the test system in CHPL_HOME/doc/developer/bestPractices/TestSystem.txt (on Github, not in the official release).

<file>.perfkeys - keyword(s) in output that gives a performance metric

There are also variations of the compiler and execution options files, since they may differ from the functional tests. Note that `--fast --static` are automatically added before the options you define.

<file>.perfcompopts - compiler command-line options during performance tests

<file>.perfexecopts - execution command-line options during performance tests

Let's see the test system in action. We've added the necessary files to the ransac directory to verify the output of ransac_rst. This required a change in the code, adding the config param `--fixrng` to use a known seed for the random number generator. This removes the variability when running the program: we should see the same output, with only the average time per try changing.

In ransac_rst.chpl:

```
forall i in 1..ntry {
  var rand : RandomStream();
  if (fixrng) {
    rand = makeRandomStream((2*i) + 1);   /* must be an odd seed */
  } else {
    rand = makeRandomStream();
  }
  /* rest of loop unchanged */
}
```

Our compiler options are

ransac_rst.compopts:

```
--fast -sfixrng=true ip_color_v3.chpl ip_corner.chpl kdtree.chpl img_png_v3.h
build/img_png_v3.o -lpng
```

ransac_rst.perfcompopts:

```
-sfixrng=true ip_color_v3.chpl ip_corner.chpl kdtree.chpl img_png_v3.h
build/img_png_v3.o -lpng --dynamic
```

You'll notice that most of the line contains the files we put on the normal compilation command. We add `--fast` so that the normal test bench runs quickly. `--dynamic` is needed to cancel the `--static` option that's added automatically and which on our system does not link.

For the execution options we list the input files and threshold.

ransac_rst.execopts and ransac_rst.perfexecopts:

```
--inname1=bonds1.png --inname2=bonds2_rst.png --thr=10
```

We will want to remove the variable timing result in the output for the functional test bench. A sed command to delete the line is enough.

ransac_rst.prediff:

```
#!/bin/sh
testname=$1
outfile=$2
sed '/avg time per try/d' $outfile > $outfile.2
mv $outfile.2 $outfile
```

This is also part of the line with the timing value we want to track. The full line is

ransac_rst.perfkeys:

```
avg time per try [ms]
```

Now in the ransac directory you can run the test bench.

```
cd ransac
start_test ransac_rst.chpl
```

You'll see the program setting up the test environment, compiling the program, and running it. It prints a summary with the pass at the end. A copy of the output is put in a log file. You can see which is used if you type

```
start_test -h
```

and look for the -logfile line. The default location is in the Chapel distribution directory. If you're working elsewhere, you'll want to change the location.

The performance test bench wants to write its results into the environment variable CHPL_TEST_PERF_DIR, which defaults to CHPL_HOME/test/perfdata/<machine>. Change this if you're not working directly in the distribution. You'll also need to rename the prediff file so that system does not run it and delete the timing data. (The development team recommends a config constant to turn off the variable output for functional tests, because the system can't skip the prediff script.)

```
mkdir /tmp/perfdat
export CHPL_TEST_PERF_DIR=/tmp/perfdat
mv ransac_rst.prediff ransac_rst.prediff.func
start_test -performance ransac_rst.chpl
```

Once this runs, you'll find a file /tmp/perfdat/ransac_rst.dat with the timing value. If you run the program repeatedly (and start_test has an option `-num-trials` to make multiple runs with only one compilation) you'll see them added to the end.

Resource Usage

There are several tools available to help track resource usage by programs.

For debugging memory usage, Chapel programs will run under valgrind. There are also command line options when you run a program. `--memStats` prints a table with the amounts that have been allocated and freed, `--memLeaksByType` the source of memory that's not been freed when the program ends, and `--memLeaks` a full list of all unfreed memory, including where in the program it was allocated.

If you compiled Chapel with the hwloc package, it contains several programs to discover the system's layout. They're found in CHPL_HOME/third-party/hwloc/install/<arch>/bin. `lstopo` will show the system's topology graphically. `hwloc-info` and `hwloc-ls` will print the information on the console.

The October 2015 Chapel 1.12 release contains a new monitoring tool called `chplvis`, found in CHPL_HOME/tools/chplvis. To use it you add two calls to start and stop the service, and can add tags to monitor smaller sections of your program.

```
use VisualDebug;
startVdebug(dirname);
/* do work */
tagVdebug(tagname);
/* do more work */
stopVdebug();
```

As the program runs it will write information about the tasks that are created and when they start and finish to a file in the given directory. The chplvis program graphically displays the resource usage on each node, such as CPU usage, number of tasks, and inter-locale communication. The graphical display doesn't seem very useful for our setup with its single CPU, but the raw data the VisualDebug module generates does look interesting. For the ransac programs it shows that the kdtree generates more tasks that there are computing units, meaning that many will be idled, while the forall loops used during alignment are tailored to the number of cores.