Programming Concurrent Systems: Assignment #5

Due on Monday, December 18, 2014

Alyssa - Ilias

January 4, 2015

Introduction

This assignment asked us to implement heat dissipation in Chapel and perform comparison experiments with our previous implementations, and then to experiment further with distributed arrays (using domain maps) and more low-level distributed methods.

Heat dissipation — Chapel

Solution description

Our Chapel implementation is a pretty straightforward translation of the C reference code. The interesting parts are the code segments which are parallelized by Chapel.

First, let's inspect the main computation loop:

We used the forall keyword to indicate that we want the outer loop parallelized. We also tried using one forall ij loop (which would also be nicer code, going over the domain in a single loop). We would have to loop through the matrix again in order to deal with the wrap-around columns. Compared to our current implementation a single loop was slightly faster for the 1000x1000 matrix, but didn't scale well.

Listing 1: Main loop

```
for iteration in 1..p.maxiter {
        dst <=> src;     //swap buffers
        forall i in 1..p.N {
                for j in 1..p.M {
                    /* computation here */
                }
        }

        forall i in 1..p.N {
                dst[i,0] = dst[i,p.M];
                dst[i,p.M+1] = dst[i,1];
        }
```

The reduction was far more complex than the computation loop, as usual.

We came up with three different implementations:

- The naive forall outer loop, which was unsurprisingly not very efficient.
- The obvious Chapel way to reduce, but that involves going over the same data four times, once for each reduction (apparently Chapel was not clever enough to merge these).
- A custom reduction (for min/max/avg only). Trying to do maxdiff using a custom reduction seemed unhelpful performance-wise.

This is the code of the latter implementation, based on `modules/internal/ChapelReduce.chpl`, and using a custom record to store the results:

Listing 2: Reduction loop

```
class heatReduction : ReduceScanOp
{
```

```
    type eltType;
    var tmin: eltType = max(eltType);
    var tmax: eltType = min(eltType);
    var tsum: chpl__sumType(eltType);
    proc accumulate(val: eltType) {
        if (val < tmin) then tmin = val;
        if (val > tmax) then tmax = val;
        tsum += val;
    }
    proc combine(other: heatReduction) {
        if (other.tmin < tmin) then tmin = other.tmin;
        if (other.tmax > tmax) then tmax = other.tmax;
        tsum += other.tsum;
    }
    proc generate() {
        return new heatReductionResults(eltType, tmin, tmax, tsum);
    }
}

/*  code between  */

r.maxdiff = max reduce [ij in ProblemSpace] abs(dst[ij] - src[ij]);
var reduction = heatReduction reduce dst[ProblemSpace];
r.tmin = reduction.tmin;
r.tmax = reduction.tmax;
r.tavg = reduction.tsum / (p.N * p.M);

/* code continues */
```

We left the other two implementations as commments in our code, for reference.

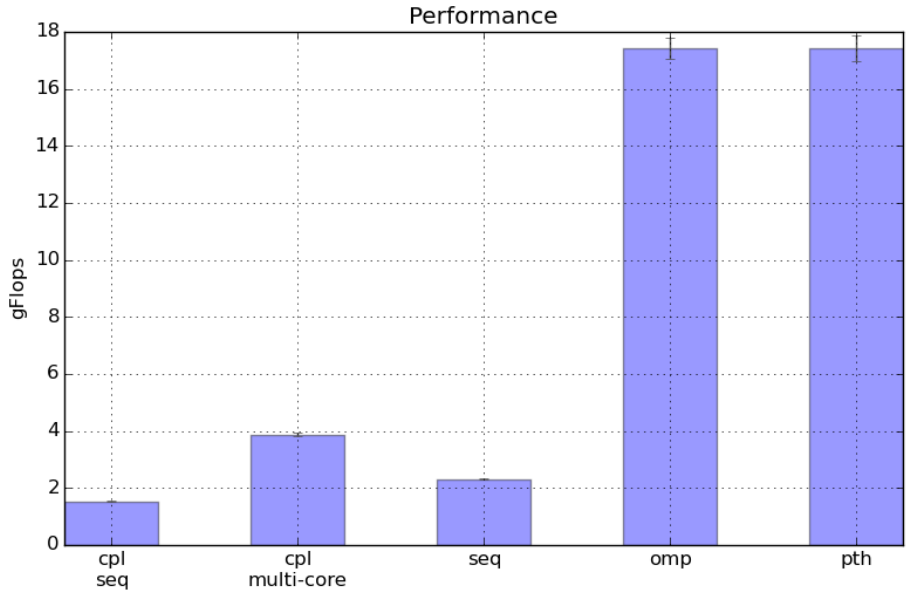
Evaluation - Experiments

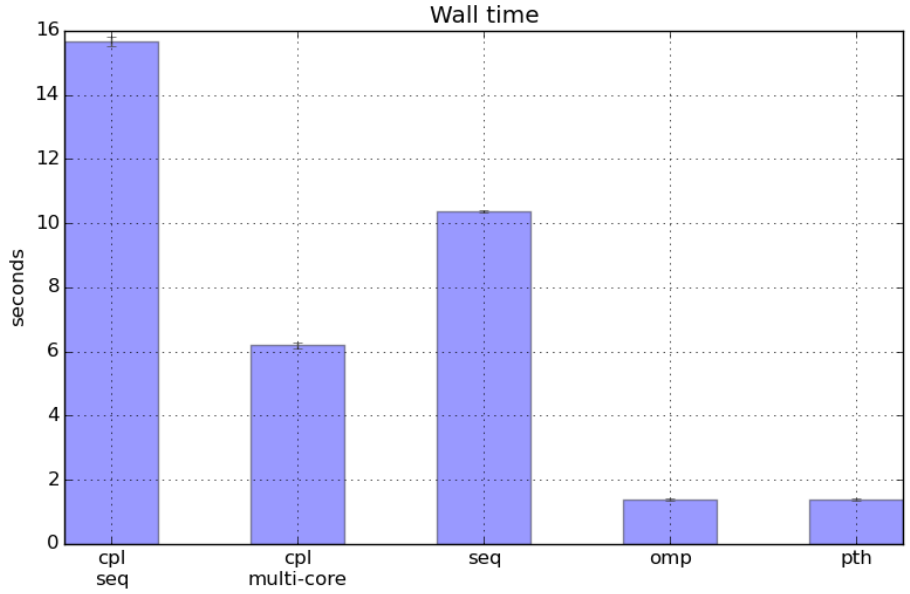
We run our experiments on the DAS-4 system. We used a normal node which has 8 physical cores, and used 8 threads, which we found to perform best in previous assignments.

We only performed experiments on 1000×1000 for now, due to time constraints, and only with one iteration count (see the end), but this is fairly representative for other reasonably-high iteration counts and matrix sizes. We'll present more in our later report update.

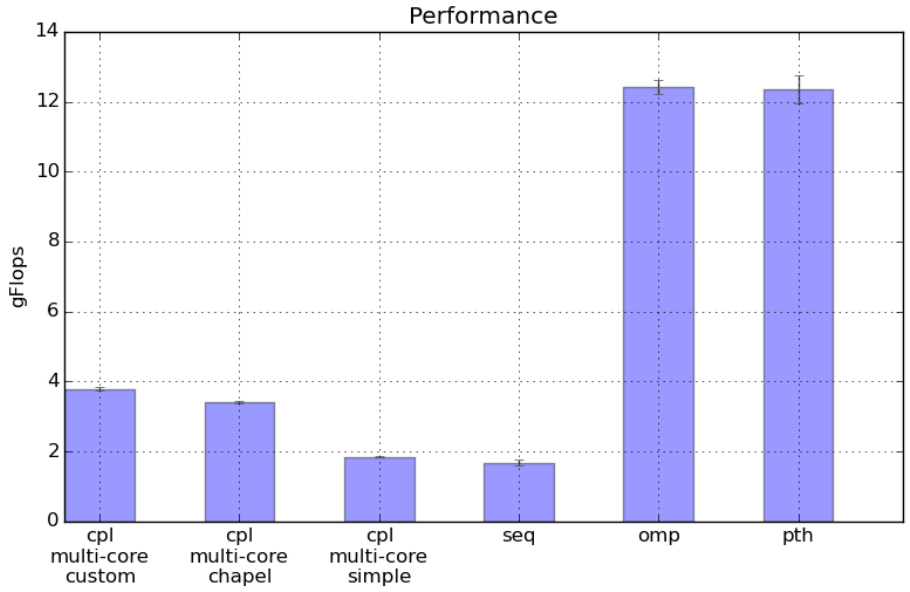
The following graphs depict performance and walltime comparison overview without reductions.

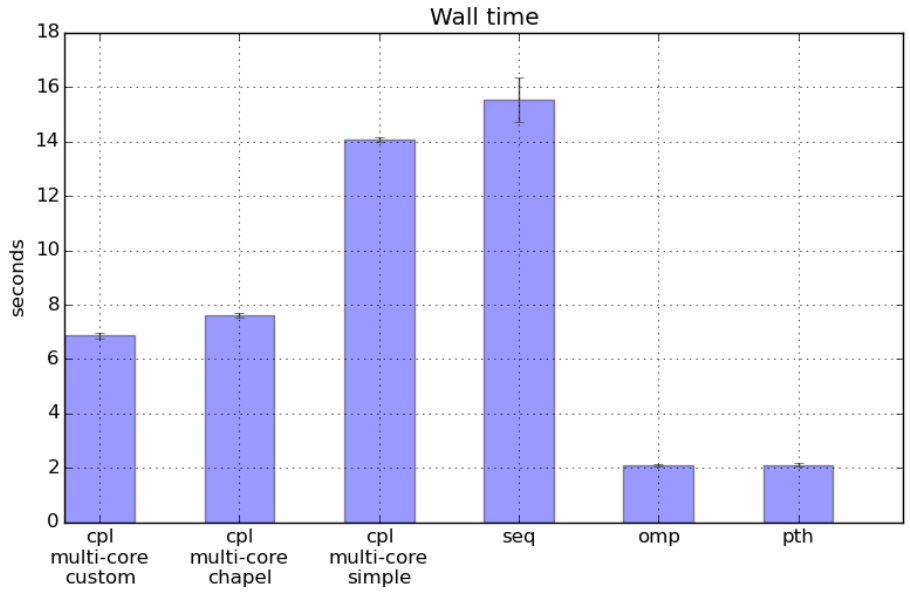
The Chapel implementation has very poor performance compared to our openMP or pthreads implementation. On the other hand, the "sequential" (without forall) version of Chapel is pretty competitive with the sequential version of C, concerning performance.



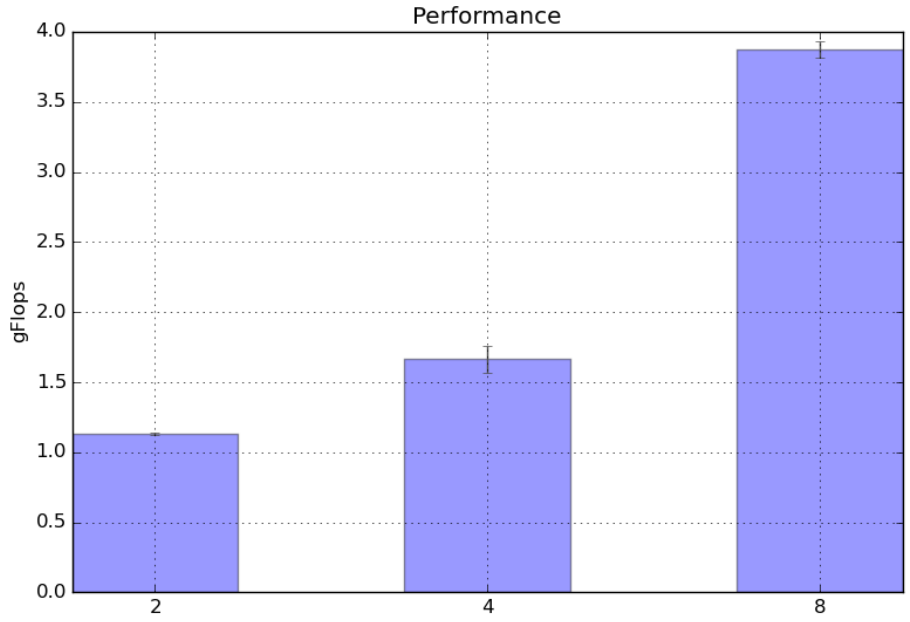


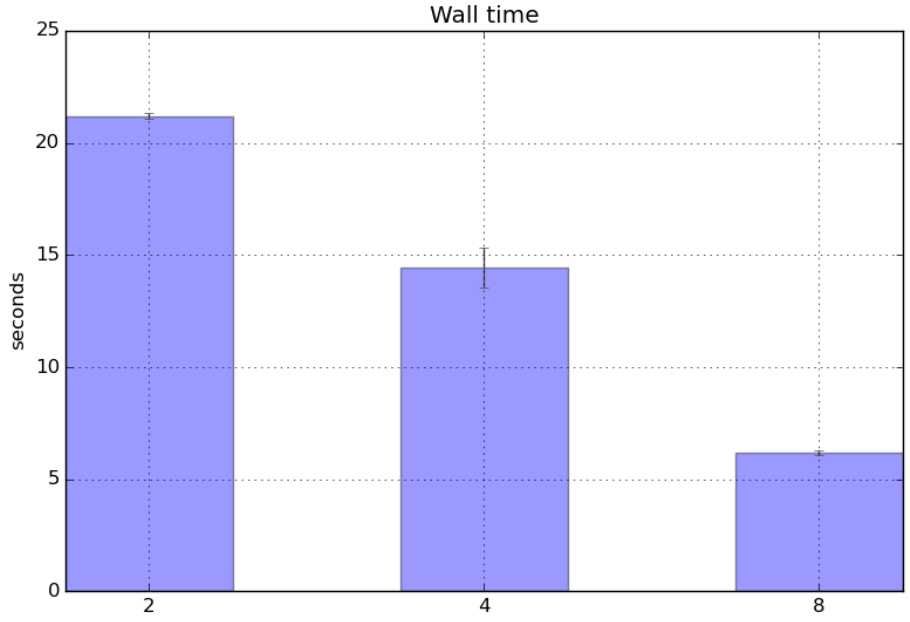
Next, we consider a performance and walltime comparison overview with reductions. We noticed that the performance of Chapel with or without reductions is almost the same which is a result we didn't have on our earlier assignments concerning reductions. This might mean that the parallelization of our main computation look might not be entirely optimized (which seems most likely), or that our reduction implementation performs really well.



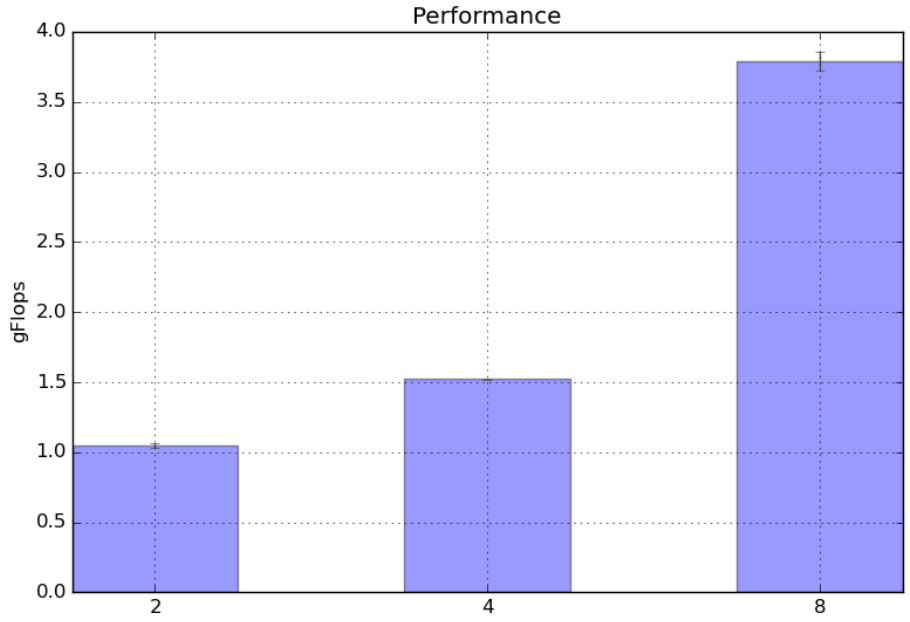


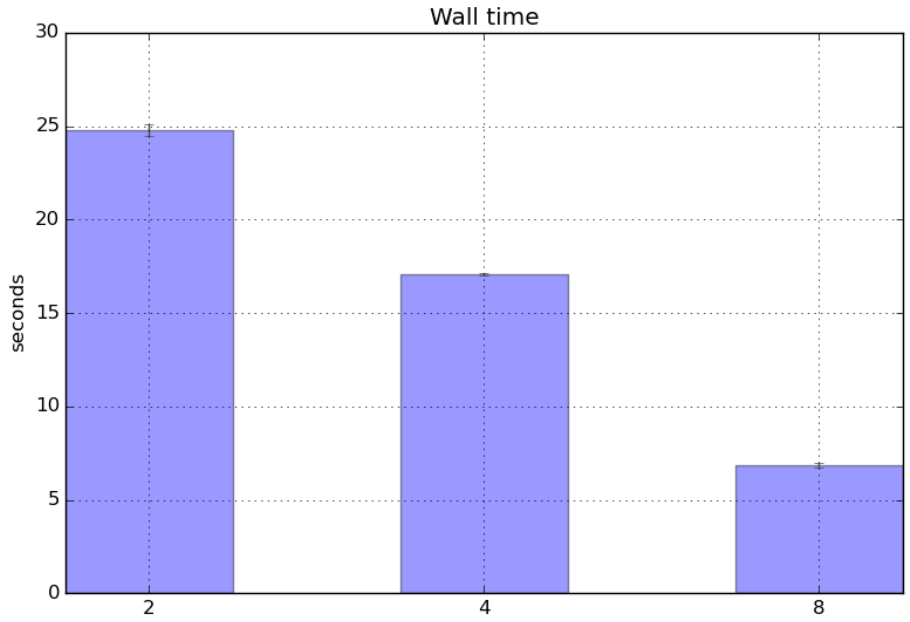
These are graphs of different thread counts without reductions. We can clearly see the performance peak when all 8 threads are used. We also notice poor performance (non linear) when we specify 4 threads.





Finally, here are graphs of different thread counts with reductions. Note that the reduction method used is our custom reduction, which as shown above performed best. As discussed, the speed is similar to the speed obtained without reductions, for Chapel, which is quite impressive compared to our other attempts, but probably indicates weird problems in our code.





9 threads is far slower (unsurprising, due to HT), and 16 threads is still slower than 8, so we didn't include those (yet) in our comparisons.

The experiments without reduction were made with the following parameters:

```
./heat -e 0.0 -i 2000 -k 2001
```

and those with reduction:

```
./heat -e 0.0 -i 2000 -k 1
```

Improvements

As discussed in the last session, large performance improvements can be obtained by using indexes into an additional dimension, rather than swapping the entire contents of the arrays[1]. We implemented this using array aliasing as shown below, and obtained considerable speed improvements (which we won't discuss further here, since it's beyond the deadline of the first sub-assignment).

Listing 3: Using an additional dimension

```
const TwoBigDomains = {0..p.N+1, 0..p.M+1, 0..1};
const BigDomain = TwoBigDomains[0..p.N+1, 0..p.M+1, 0];
var data: [TwoBigDomains] real;
var src = 0, dst = 1;

// buffer swapping
dst <=> src;
var cursrc: [BigDomain] => data[..,..,src];
var curdst: [BigDomain] => data[..,..,dst];
```

We also discovered that it is possible to include all reductions in the same custom reduction operator, with (marginally) improved performance.

Domain mapping — Chapel

For the second part of the assignment, we were asked to use domain maps to run our Chapel code on multiple nodes.

We were not successful in making this work well. Simply adding domain maps to our Chapel code results in a large (more than an order of magnitude) performance decrease.

By printing `TwoBigDomains.dist` and `ProblemSpace.dist` (for example), it is possible to see that the blocks get mapped to the locales in the intended fashion. For example, if we try a naive Block-based distribution with a 1000×1000 matrix and two locales, it is split (as expected) into two parts, with the extra dimension for our 'swap trick' correctly not being distributed:

```
Block
-------
distributes: {1..1000, 1..1000, 0..0}
resulting in:
  [(0, 0, 0)] locale 0 owns chunk: {..500, .., ..}
  [(1, 0, 0)] locale 1 owns chunk: {501.., .., ..}

Block
-------
distributes: {1..1000, 1..1000}
across locales: LOCALE0
LOCALE1
indexed via: {0..1, 0..0}
resulting in:
  [(0, 0)] locale 0 owns chunk: {..500, ..}
  [(1, 0)] locale 1 owns chunk: {501.., ..}
```

Compiling the Chapel code with `-sdebugBlockDist=true` enables debugging for the block distribution, which reveals a huge number of Block distributions being created, but otherwise gives little insight into our performance problems.

References

[1] Raphael Poss, *Pointer Operations in Chapel*.
`https://sourceforge.net/p/chapel/mailman/message/27306614/`

Compiler Award

presented to

Chapel

for allowing us to reach new heights in frustration

4 January 2015

Generated by CertificateMagic.com