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.

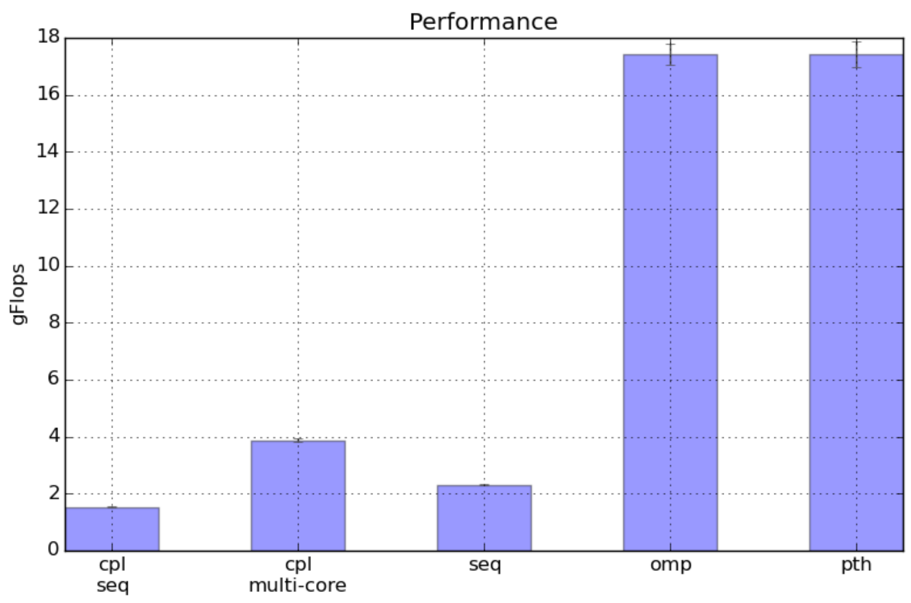
Evaluation - Experiments

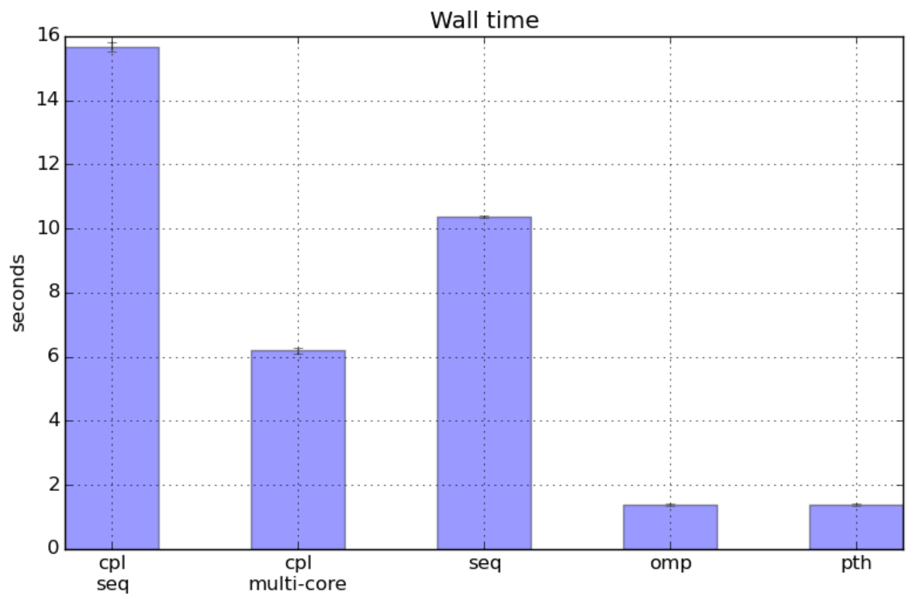
We run our experiments on the DAS-4 system. We used a normal node which has 8 physical cores, and used 8 threads, which we found to perform best in previous assignments.

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.

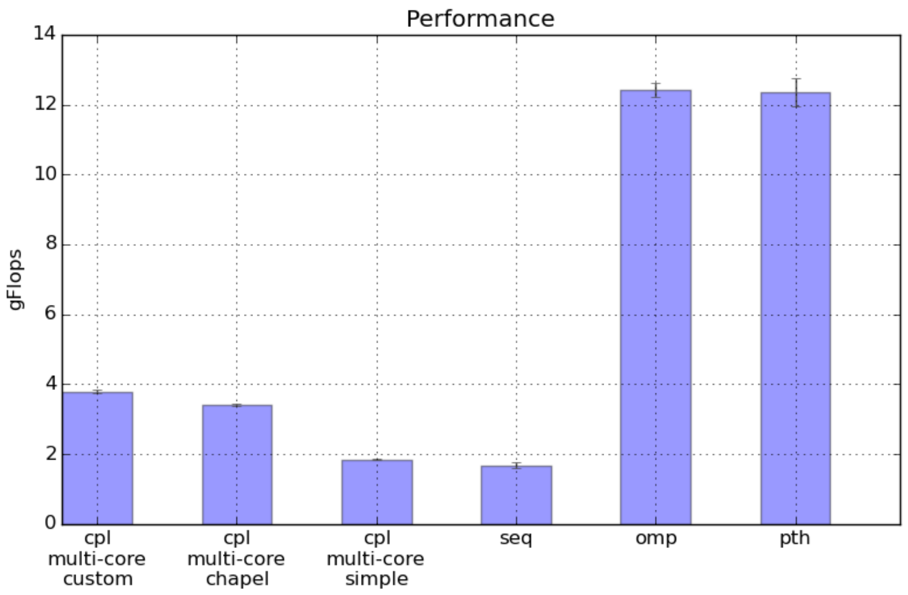
The following graphs depict performance and walltime comparison overview without reductions.

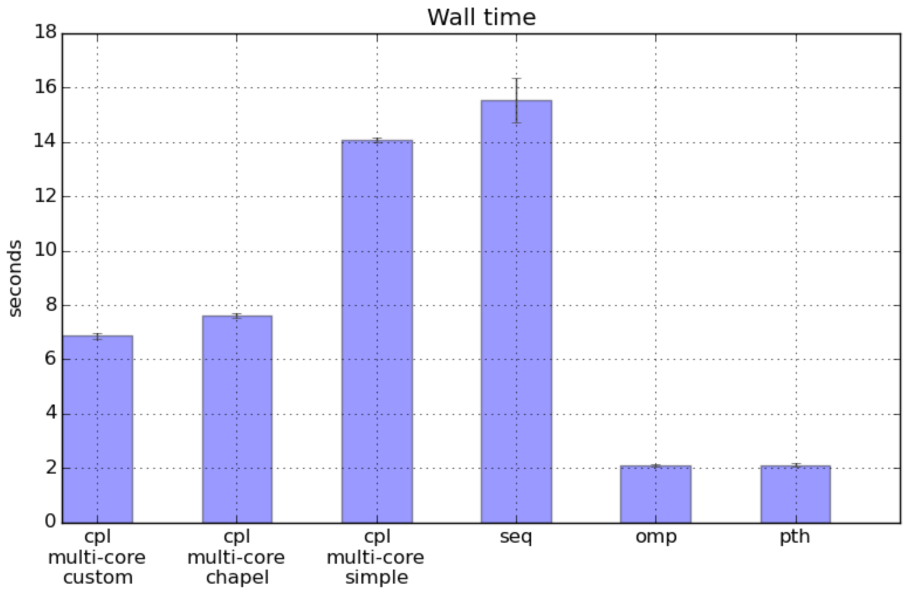
The Chapel implementation has very poor performance compared to our openMP or pthreads implementation. On the other hand, the "sequential" (without forall) version of Chapel is pretty competitive with the sequential version of C, concerning performance.



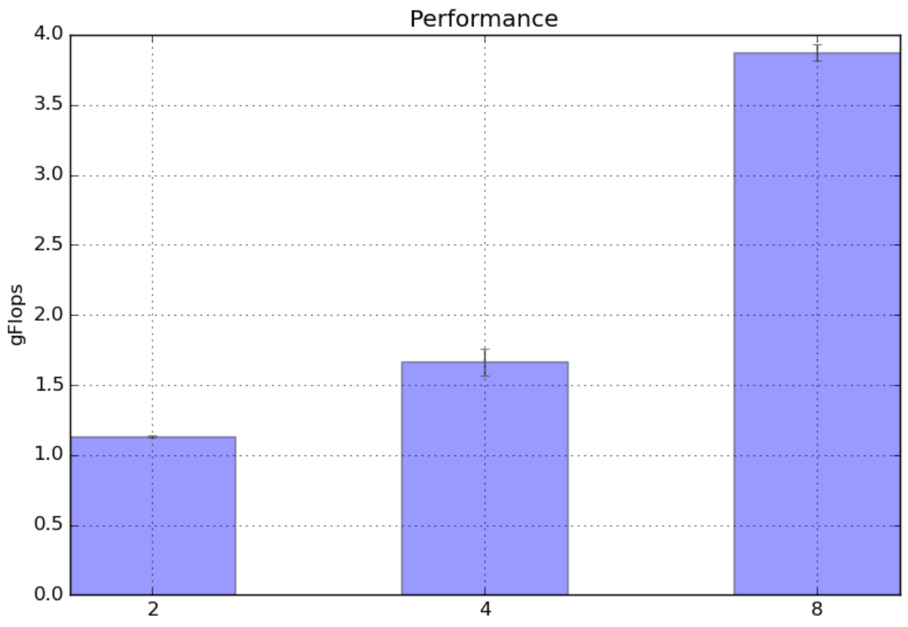


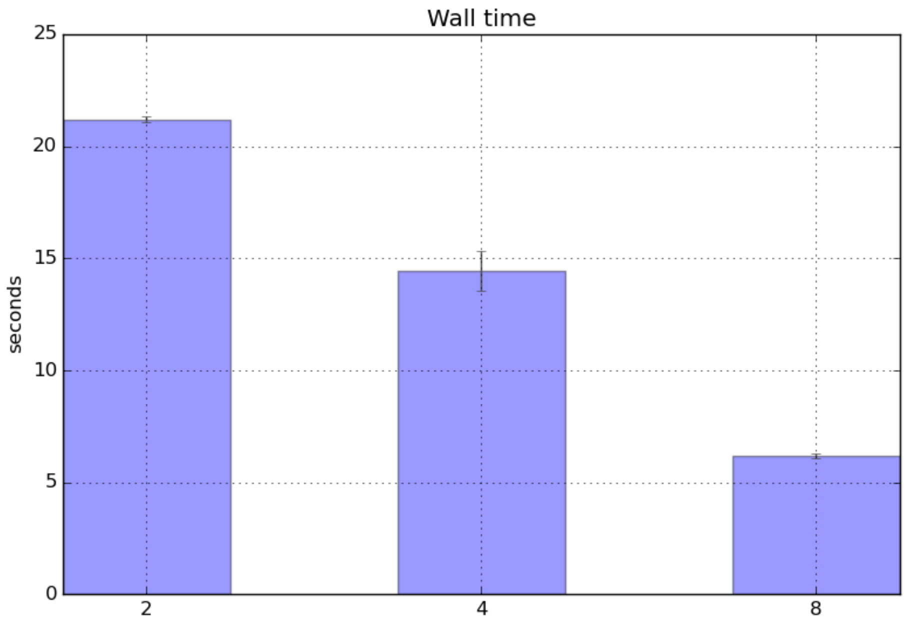
Next, we consider a performance and walltime comparison overview with reductions. We noticed that the performance of Chapel with or without reductions is almost the same which is a result we didn't have on our earlier assignments concerning reductions. This might mean that the parallelization of our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



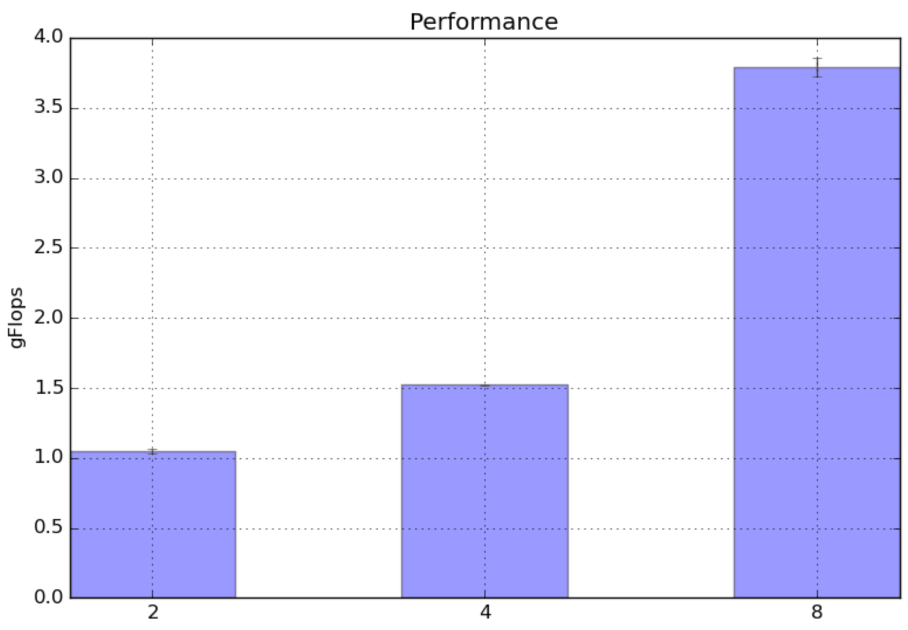


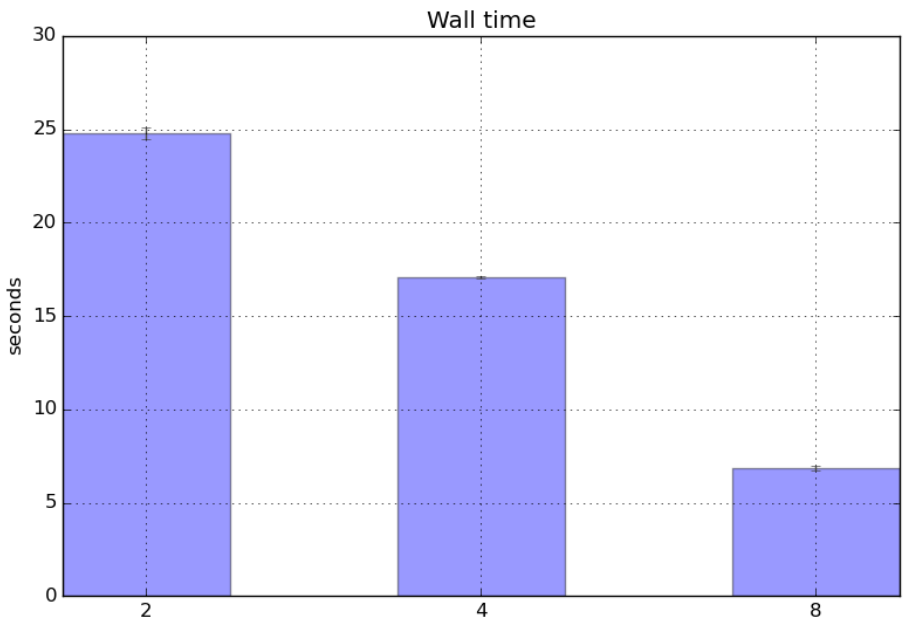
These are graphs of different thread counts without reductions. We can clearly see the performance peak when all 8 threads are used. We also notice poor performance (non linear) when we specify 4 threads.





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
```

Further Discussion

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.

Finally, we learnt from the last session that it's possible to set the number of threads per locale to use from within Chapel, rather than making use of `CHPL_RT_NUM_THREADS_PER_LOCALE`, but we haven't implemented this.

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 implemented this by using `dmapped` on the main domain (`TwoBigDomains` in our current code) and then making the other domains subdomains (which should, in theory, inherit the domain mapping, and indeed we see no difference if we specify this manually for every variable/domain, so hopefully this works).

We were not successful in making this work well. Simply adding any domain map to our Chapel code results in a large (more than an order of magnitude) performance decrease, without even moving to a multi-locale version compiled with gasnet support.

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.

After we excluded some obvious potential causes for the performance problems (such as disabling the smearing of the start/end columns, which added some extra complexity and also caused code to only run on two locales for the cases where we split the locales up along the columns dimension), we decided to give up on further investigation and move onto the next sub-assignment, since the fact that the performance is so bad for the non-multi-locale version indicates some fundamental failure. It's possible that the additional dimension approach we implemented earlier is also causing problems, although attempting to simply remove this was not helpful.

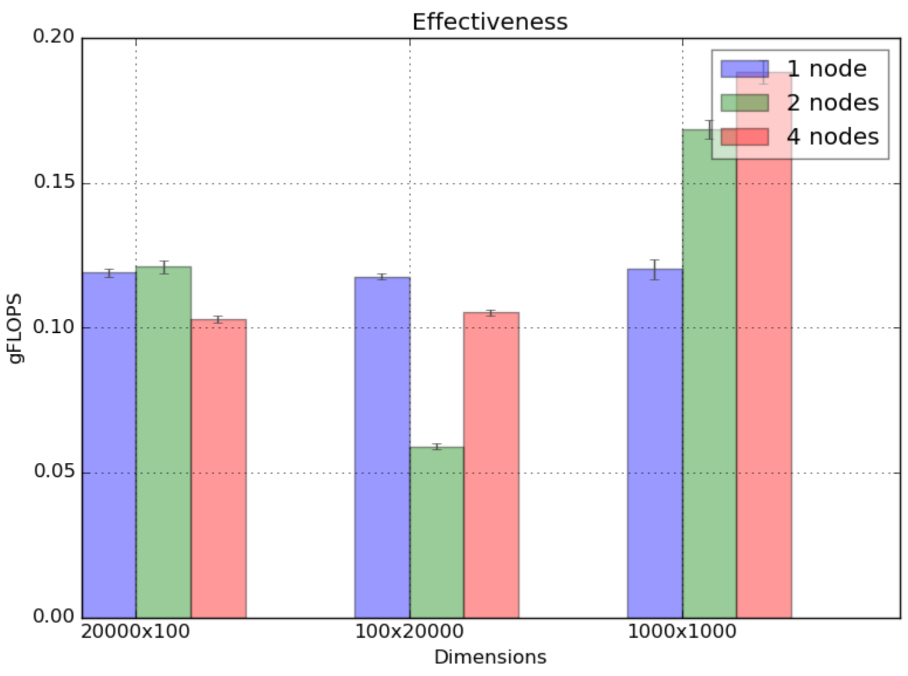
Since our problems seemed to be present even when running locally, we didn't investigate the remote accesses (using the comm diagnostics features of Chapel).

The results we present here are using the Block distribution with the default values; we didn't get improvements from other parameters and certainly not for different distributions (unsurprisingly, since for example Cyclic would be crazy given the access patterns involved, and we need the system to be consistent so replicated non-synchronized versions are unhelpful).

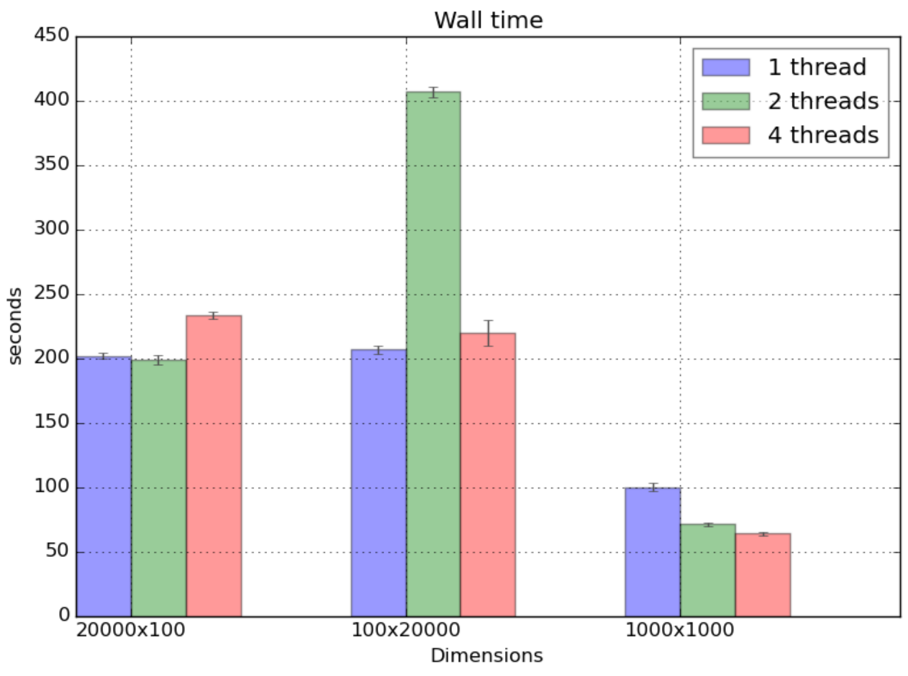
We couldn't get useful results for the 8 and 16 node versions; when running with non-trivial numbers of iterations, the jobs did not finish correctly on DAS-4, we suspect because they took far too long (even with job times increased to 45 minutes), and again we felt that our time was not best used debugging this issue beyond an hour or so of work.

That said, we have a figure depicting the performance of different matrices and nodes, using 1000 iterations:

Domain mapping — Chapel continued on next page. . .



And here, the same results, but presented using time:



Unfortunately, given our problems, we don't think there is much meaningfulness to be attached to this code; we'd hoped to be able to discuss how they might provide evidence that the entire dataset is being copied excessively, and that no smart halo-cell type copying is going on, but instead the results mostly serve to confuse us further. The 100×20000 matrix results for 2 nodes in particular confuse us terribly (and as you can see from the error bar, this is not just a hiccup in a single run, but rather something structural). You can, however, clearly see that for the 1000×1000 case we do get improved performance when adding more nodes, and that is at least as expected/hoped. Hopefully with larger matrix sizes this difference would be far more significant..

Low-level implementation — Chapel

The limitations of the Chapel compiler itself became quickly apparant while trying to implement this code; we constantly encountered compiler internal errors with no further information, making development a very time-consuming process.

Our relatively simple final attempt, which tries to minimise the changes to the design of the other Chapel code, is provided as `compute_manual.chpl`; it splits the data up into columns, with one column per locale, much like the block map concept.

For each iteration of the main loop, each locale runs computation code in parallel. This code copies data out of the 'global' data matrix, not only the column which it will be working on but also a certain amount of overlap on each side (specifiable using a parameter) which provides the requested *halo nodes*. It then performs the calculation repeatedly on the column, in a loop, as many times as there is overlap available (i.e. if we have an overlap of 1 then we can only run one calculation, because then we will need data from other nodes, but if we have an overlap of 2 then in the second iteration we can still do the calculation on the inner halo cells based on the values of the outer halo cells calculated at the first iteration).

Finally, each locale copies data back into the main matrix. This is not at all efficient (each locale only really needs to copy the overlapping data to/from its neighbour locales), but in our final attempt we implemented this in this fashion since we hoped it would avoid having to adapt the reduction code.

It does not work at all for a single locale (because it doesn't support copying from both sides at once), and gives incorrect results anyway. In hindsight, it would have worked better if we'd split into groups of rows (since we wouldn't have to deal with the wraparound problems), but unfortunately we didn't succeed in making this work in the time available to us.

Conclusion — Chapel

As warned, we had very little success trying to implement distributed heat diffusion efficiently in Chapel, and we didn't approach the single-node performance for our problem. It's clear that trying to implement the algorithm manually without using Chapel's built-in parallelisation features is not a promising approach for many reasons.

In conclusion, Chapel is an interesting language which clearly has much potential for efficient development of distributed code. However, the implementation is sorely lacking, and the limited documentation has been a significant problem. It might be useful for rapid prototyping of code (especially with features such as the built-in parameter mapping to variables), but unfortunately the slow compile turnaround times also limit its usefulness in this sense.

However, the development team appear to be actively modifying and improving both the language and the toolchain, and we look forward to seeing what Chapel will be like after a few more years of development.

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