Performance Evaluation of Chapel's Task Parallel Features

Michèle Weiland, *EPCC and* **Thom Haddow**, *Imperial College London*

ABSTRACT: *Chapel, Cray's new parallel programming language, is specifically designed to provide built-in support for high-level task and data parallelism. This paper investigates the performance of the task parallel features offered by Chapel, using benchmarks such as N-Queens and Strassen's algorithm, on a range of different architectures, including a multi-core Linux system, an SMP cluster and MPP.*

KEYWORDS: Chapel, task parallelism, benchmarks.

1. Introduction

Over the past few years the United States Department of Defence research agency, DARPA, has been funding several diverse HPC projects under the heading of its High Productivity Computer Systems (HPCS) [1] research and development program. The HPCS program aims to increase supercomputer productivity by 2010 by fostering progress across the whole HPC domain, from computer hardware and architecture development, to software tools and programming environments. The stated aims of this program is to develop systems with "Performance, Programmability, Portability and Robustness" as key properties. One of the component projects of the HPCS program is the development of new programming languages for HPC which focused on these properties. Three new programming languages, from three supercomputing companies, were sponsored by this project: Sun Microsystems developed *Fortress*, a language aiming to map mathematical and algorithmic concepts into a programming language, IBM developed *X10*, a high-performance subset of Java, and Cray Inc., under its Cascade project, developed *Chapel*, the **C**ascade **H**igh **P**roductivity **L**anguage.

Chapel

Chapel [2] is a new high-level parallel programming language primarily aimed at the programmability aspect required by the HPCS program. It aims to provide a more expressive interface to parallel programming, in which algorithmic details can be abstracted from underlying optimisation details, and it takes inspiration from both existing HPC languages such as HPF and parallel dialects of C, as well as implementing high level language concepts typical of more modern languages, such as Java and Python. In terms of parallelism, Chapel implements a global-view interface (*i.e.* data is not tied to any particular parallel entity), and it provides high-level abstractions for both the data and task parallel programming paradigms.

The Chapel programming language is still in its development stage - the primary focus in Chapel has been on creating a functionally complete and correct language implementation, rather than on specific optimisations. The version of the compiler used for the present work, v0.9, was released on the 16th April 2009.

The aims of our work are to characterise the performance of the task-parallel constructs in the Chapel language by means of developing a suite of benchmarks suitable for the task parallel model. The aim was to objectively analyse compare the performance of these benchmarks across a range of platforms, with a view to identifying aspects of the language implementation that were performing sub-optimally and to isolate targets for optimisation.

Implementation status

Chapel is very much still work in progress. While the language now supports distributed memory architectures, certain features are not yet (fully) implemented. These include distributed arrays and domains, as well as some data parallel statements, which are currently made serial

by default by the compiler. The task parallel statements are much more mature, which is why this work has concentrate on that aspect of the language.

2. Chapel's Task Parallel Features

While Chapel supports implicit (through whole-array operations) and explicit (through the `forall` loop) data parallelism, the main focus on this study is on the task parallel aspects of the Chapel language. The underlying task-parallel entity is the concept of a "task", an independent thread of potentially parallel computation. Chapel manages these in task lists, and uses a specified number of physical threads (as specified by the `--maxThreads` parameter) to process these tasks. When these threads become idle or are blocked by Chapel's synchronisation constructs1, then they use these task lists in order to find new unprocessed tasks in order to progress the computation.

"begin" and "sync" statements

The `begin` structure is the most basic of Chapel's task parallel constructs. It spawns a new parallel task to handle the statement block and immediately continues execution beyond it. It is classed as an *unstructured* parallel construct because it only acts to introduce parallelism, and requires external synchronisation in order to cooperate with other threads.

The sync statement acts as a means of synchronisation for begin statements, essentially acting as a join that waits for completion of all the dynamically encountered begin statements dispatched within the sync block.

"cobegin" statement

The `cobegin` statement essentially acts as a compound `begin/sync` statement block. Each statement or block within the `cobegin` block is dispatched as a separate task, and at the end of the block the dispatching thread will wait until all the tasks dispatched within until returned.

"coforall" statememt

The `coforall` statement is the task-parallel variant of the data-parallel `forall` statement, which is in turn a parallel variant of the plain `for` loop. Whereas in the `forall` statement implies that independent loop operations *may* be dispatched in parallel, the `coforall` statement guarantees this, which may be necessary to enforce correct behaviour in concurrent programs.

"serial" statement

In the task parallel programming model, and especially in the case of nested parallelism, it is fairly easy to expose too much *potential* parallelism in algorithm implementations. This can result in suboptimal performance due to threading overheads. Chapel provides the `serial` statement as a convenient means of suppressing parallelism — its effect is to disable the spawning of parallel tasks within its scope.

The statement takes a conditional as an argument, and if that conditional resolves to true, then any dynamic parallelism (*i.e.* any of the above parallel statements) reached inside its scope will instead be dispatched serially. Notably, the scope of the `serial` statement extends beyond the local scope to that of any statements called from within it, and it is in turn possible to have an inner `serial` statement inside the scope of some outer `serial` statement which could re-enable dynamic parallelism.

Synchronisation variables

Chapel supplies the type modifier keywords `sync` and `single` to enable concurrent tasks to synchronise and communicate over specialised variables. Their interface as presented to the programmer is interesting in that they are manipulated in the exact same manner as regular variables, except that they have special read and write semantics. Synchronised variables carry extra state information that classes them as either *full* of *empty*, depending on whether they contain a value or not. Attempts to read from an empty variable will cause the reading thread to block until another thread fills (*i.e.* writes to) the variable, and similarly, an attempt to write to a full variable will block until another thread empties it. These variables essentially act as classes, and thus have a selection of instance methods available which allow the semantics of read and write, in terms of whether they block or not, and whether they leave the variable full or empty, to be chosen fairly arbitrarily. The `single` is just a specialised case of the *sync* variable which is only allowed to be *filled* once, and it causes a runtime exception to attempt otherwise. While the functionality (aside from the single-write semantics) of the `single` variable type can be easily emulated with a `sync` variable, it is imagined that the `single` variable type leaves room for future compiler optimisation (for instance, not checking the full/empty status of a single variable twice within the same block).

It's also notable that the form of blocking employed by these synchronisation variables can cause a Chapel program to deadlock if the allowable number of threads running synchronously has already been reached—this is especially likely if the `-maxThreads` option has been set to 1. This is in contrast to the form of blocking employed by the explicit and implicit synchronisation statements, in which a blocked thread will check if there are an unprocessed tasks present on the task list that could

be executed, and thus progress the computation (However, enabling this kind of continuous progression computation on the context-less synchronisation variables would be very complicated indeed, so this is understandable).

3. Benchmarks

This work has aimed to cover a range of task parallel benchmarks, starting with some very small and trivial benchmarks designed to stress or demonstrate particular language features, while the remaining benchmarks are more based around computational kernels, ending with something resembling a very small application-level benchmark.

Some of the benchmarks were used on shared-memory nodes only (N-Queens, Strassen, Mandelbrot), others were aimed at distributed memory architectures (Pi and Black-Scholes). The latter benchmarks were chosen to avoid the use of distributed datastructures, which, although supported, are in very early stages of optimisation.

Microbenchmarks

Chapel's parallel constructs there are different ways to achieve the same thing, but essentially only a few forms of underlying mechanisms. In a bid to quantify the difference in performance that might arise from the choice of parallel construct employed, a few short microbenchmarks were written. The benchmarks show five different implementations to call an arbitrary function eight times. The functions are repeated 5000 times in order to gather more reasonable timing statistics. The five implementations use: `cobegin`; `begin` with `single` variables; `begin` with a `sync` block; `begin` with a `sync` block and a parameterised `for` loop; `coforall` loop.

N-Queens

The eight queens puzzle is the problem of enumerating the number of configurations in which eight queens can be placed on a standard `8×8` chess board in a non-conflicting fashion, *i.e.* no two queens are on the same row, column or diagonal on the board. There are over 4 billion possible ways of placing eight queens on a chess board, and yet there are only 92 possible solutions.

The n-queens problem is a generalised formulation of the eight queens puzzle in which `n` queens have to be placed on an `n×n` chessboard. While serving little practical purpose, it is a fairly well developed problem in the field of computer science as it serves as an example of many forms of algorithms and concepts, such as the cost of brute-force search and the use of heuristics. As such, there exist many standard implementations of the problem in a large number of computer programming models.

As a benchmark, the problem is interesting because of the explosive size of the search space. Viewed naïvely, there are $\prod_{i=0}^{n}(n^2 - i)$ possible solutions for an `n×n` chessboard, providing a very high computational complexity but the configurations of the problem can be expressed in an n-tuple of integers between 1 and `n`, meaning the problem has a very low memory complexity and additionally the problem requires no floating point calculations. In this project we are aiming to benchmark the parallel capabilities of the language, not the memory or floating point properties of any particular machine (which are generally the targeted factors of HPC benchmarks), so this benchmark presents an ideal testing ground, with a large range of different problem sizes to work with, and a variety of possible programming approaches to solving the problem.

The most relevant algorithmic approach here is the recursive backtracking search, due to its natural mapping onto task parallel constructs. Restricting the implementation to be based on this particular algorithm is not particularly limiting as here the focus is on the means of parallelisation, in which there are several options, and not the details of the algorithm itself.

Implementation in Chapel

The serial algorithm, which also forms the core of the parallel algorithm, is represented as a recursive function which takes as arguments the current row number and the current configuration of queens, and returns the number of solutions found as an integer.

```
def nqueens_solv(row : int, queens : [] int) : int {
 var solutions = 0;

 // iterate over columns
 for col in 1..n
 {
  // test current config
  if (isSafe( col, row, queens) )
  {
    //place queen in row
    queens[row] = col;

    // not complete -> recurse & accumulate
    if (row < n) then
     solutions += nqueens_solver(row + 1, queens);
    // complete -> increment count
    else solutions+=1;
  }
 }
 return solutions;
}
```

Figure 1: Recursive serial n-queens solver in Chapel

The naive form of task parallel decomposition of this algorithm is to simply assign each first row queen

position to a parallel task (of which there will be n in total) and sum the results. This however presents severe load balance issues because the algorithm only searches through safe configurations and the various starting columns each give rise to different numbers of safe configurations, thus differing quantities of computational work. In order to supply a non trivial amount of computational work for benchmarking, n is chosen to be in the 12-14 range, and this approach will only scale while n (and thus the number of potential parallel tasks) is much larger than the number of processors.

A slightly more advanced solution involves spawning a parallel task for each of the possible configurations for the first *two* rows, thus providing $O(n^2)$ possible parallel tasks, which is much greater than the potential number of processors. This will involve a substantial overhead due to dispatching significantly more parallel tasks, but providing a much finer granularity of decomposition means the load balance issue is alleviated and the algorithm will scale to a greater number of processors.

```
def parallel_nqueens(n: int) {

 var partialSolutions : [1..n, 1..n] int;

 sync {

  // for each possible configuration
  // for row 1  and 2 (r1,r2)
  for (r1,r2) in [1..n, 1..n] {

   // if the configuration is safe
   // (ie queens do not conflict)
   if( r1!=r2 && r1!=r2+1 && r1!=r2-1) {

    begin {

     // form row 1 and 2 as a
     // configuration array
     var qconfig : [1..n] int;
     qconfig[1..2] = (r1,r2);

     partialSolutions[r1,r2]=nqueens_solver(3,qconfig);

    }
   }
  }
 }

 var totalSolutions = + reduce partialSolutions;
 return totalSolutions;
}
```

Figure 2: Dispatching the n-queens solver in parallel

Strassen

Matrix multiplication forms the dominant aspect of the runtime for many computer applications because of it's relatively high complexity — $O(n^3)$ in the naive case. Strassen's algorithm is a reformulation of matrix multiplication which reduces its complexity for certain specialised cases, specifically square matrices of size $2^n \times 2^n$. While this is a not insignificant improvement in runtime, it should be noted that Strassen's algorithm does suffer from certain numerical stability problems — however, it serves as an excellent benchmarking example because it naturally exhibits both recursive parallelism and task decomposition, while introducing a fairly significant amount of floating point calculation.

Strassen's algorithm is based around decomposing each of the matrices, say A and B, to be multiplied into four equally-sized block matrices, each representing a quadrant of the original. Seven new matrices M_{1-7}, each the size of a quadrant, are calculated based on these decompositions and are used to define a new matrix C, the result of the multiplication.

In the ideal case this algorithm is applied recursively in order to perform the matrix multiplications necessary to calculate M_{1-7} until the size of the matrix quadrants have degenerated into scalars. In practical terms it is more optimal to use naive matrix multiplication once the matrix quadrants become smaller than a certain threshold. The notable feature of this algorithm is that it is a divide and conquer algorithm in which the work is done in the divide stage, and it is not trivial to pre-generate the set of tasks required for expressing this in the task-decomposition style.

Chapel implementation

The most notable (relevant) factors in the Chapel implementation are the use of both scalar-promotion whole-array operations and the use of *domains*, which are a new Chapel concept. Domains are first-class values in Chapel which define sets and subsets of array indices for arbitrary numbers of dimensions. By generating domains for each quadrant of each array being worked with it becomes possible to use whole-array operations on specific quadrants of these arrays. The first half of the Strassen algorithm generates the quadrants of the two source matrices, which are then are used to index into the arrays for whole-array operations that pre-compute the factors of the multiplications that form the second part of the algorithm. In theory, these pre-computations should not be necessary because it should be possible to use whole-array operations as arguments to the later matrix multiplications. The actual pre-computations are done within a `cobegin` statement, but this is largely opportunistic parallelism because these submatrix operations complete very quickly.

Strassen's algorithm is a recursive algorithm, designed to operate on successively smaller submatrices. In the ideal case, this recursion would continue until the submatrix computations required are just unit operations

(*i.e.* `1×1` matrix operations), but in practical implementations it is generally most efficient to switch to using naive multiplication once the size of the submatrices reaches a certain threshold. Regardless of which algorithm is used to perform the submatrix computations, in the Chapel implementation all these computations are performed in parallel using a `cobegin` statement, and it is here that the bulk of the parallel computation is performed. Finally, once these operations are complete, another set of whole-array operations are used to assemble the quadrants of the result matrix, C, which is returned as the result.

Mandelbrot

The Mandelbrot benchmark represents a benchmark with a fairly substantial memory and floating point requirement, but without the complicated hybrid nested-task-farm decomposition illustrated in the Strassen benchmark.

Chapel implementation

The kernel of the Mandelbrot benchmark is obviously very well known, and the Chapel implementation is shown in Figure 3. This function takes a 2D array representing the image, and performs the standard Mandelbrot algorithm with the external variables `maxIters` and `escapeLimit` taking their usual roles. It's most notable feature is at the start of the main loop, where it uses a Chapel tuple combined with iteration over a domain.

```
def mandelbrot(image : [?D] real) where D.rank==2
{
 for (px,py) in D {

  //Map from pixel-space into real-space
  var (x,y) = map(px,py);
  var (xt,yt) = (x,y);
  var iter = 0;

  while (xt*xt + yt*yt <= escapeLimit
         && iter < maxIters)
  {
   var xtemp = xt*xt - yt*yt + x;
   yt = 2*xt*yt + y;
   xt = xtemp;
   iter += 1;
  }
  if ( iter == maxIters ) then image(px,py)=0;
  else image(px,py)=iter;
 }
}
```

Figure 3: The (abridged) kernel of the Mandelbrot benchmark, in Chapel.

Because the kernel is defined to operate on an array, parallelising the Mandelbrot benchmark was trivial. An iterator function `decomposeDomain()` was developed, which takes a Chapel domain and two dimensions as arguments, and then it returns a stream of subdomains which divide the source domain as requested. The parallelisation loop is shown in Figure 4, in which a `coforall` loop is used to iterate over this stream of subdomains in parallel.

```
coforall d in decomposeDomain(imageD,xdecomp,ydecomp)
   do mandelbrot(image[d]);
```

Figure 4: Parallelising the Mandelbrot benchmark.

Pi

The number π can be approximated using a simple formula:

$$\frac{\pi}{4} = \int_0^1 \frac{dx}{1+x^2} \approx \frac{1}{N}\sum_{i=1}^{N} \frac{1}{1+\left(\frac{i-0.5}{N}\right)^2}$$

The larger the value of *N*, the more accurate the approximation will be. All *N* iterations of the sum are independent of each other, making the problem trivial to parallelise.

This benchmark was chosen primarily to look at the performance of a small, embarrassingly parallel problem on distributed memory architectures. No large arrays or data structures are being used and the algorithm only uses a small amount of floating point calculation.

Chapel implementation

The parallel implementation of this benchmark in Chapel was straightforward. In order for the code to run on a distributed memory system, to work load needs to be distributed evenly between locales. This is done using a coforall statement. A second coforall statement is then used to further break up the work based on the number of threads that are available on each locale. Each task calculates its portion of the overall sum, which then needs to be added to the global total – this is done by using a `sync` variable on `Locales(0)`, which ensures that only one tasks at a time can write to the variable. Figure 5 shows the implementation of the π approximation algorithm.

```
var total$ : sync real = 0.0;
var pi : real = 0.0;
var totalTasks : int = numLocales * maxThreads;

coforall loc in Locales{
 on loc {

  coforall tid in 0..maxThreads-1 do{

   var sum : real = 0.0;
   var overallTaskID : int = here.id * maxThreads + tid;
   var lower, upper : int = 0;

   lower = overallTaskID * (N / totalTasks);
   upper = lower + (N / totalTasks - 1);

   for n in lower..upper{
```

```
    sum += (1 / (1 + ((n - 0.5)/N)**2));
   }

   on Locales(0) do {
    total$ += sum;
   }
  }
 }
}
pi = total / N * 4;
```

Figure 5: Chapel implementation of π approximation algorithm.

Black-Scholes

The Black-Scholes algorithm is a well known model from the world of finance theory. It was chosen as a benchmark because it represents a real-life application which can easily be parallelised using a Monte Carlo technique and which dies not rely on distributed data structures. The Black-Scholes model simulates the variation of stock prices over a certain period of time, based on current stock price, risk and volatility rates, as well as random fluctuations of prices. The simulations are independent of each other and can be executed concurrently.

The parallelisation is similar to the π approximation benchmark, yet the Black-Scholes algorithm is much more compute intensive and will thus give a more accurate reflection of the real performance of the Chapel compiler.

Chapel implementation

The simulations are distributed among locales and threads using two `coforall` statements. Each thread then uses nested for loops to represent the number of simutations and the duration of each simulation. Factors such as the volatility or the risk are global and live on `Locales(0)` – however every remote thread needs to access them continuously during every simulation. These factors are therefore defined as `params` – the compiler will replicate the constants on every locale, thus optimising access. Figure 6 shows the abridged Chapel implementation of the Black-Scholes algorithm.

```
coforall loc in Locales{
 on loc{

  coforall tid in 0..maxThreads-1{

   /* instantiate variables etc. */
   …

   cobegin {
    a = 1.0 + (rc * dt);
    b = volatility * sqrt(dt);
    invnsteps = 1.0 / (ntimesteps:real);
   }
```

```
   /* loop over iterations */
   for i in lower..upper{

    /* initialise stock price and sum */
    …
    /* loop over time steps */
    for n in 0..ntimesteps-1 {

     /* use second random number on odd iterations */
     if(n & 1) then gr(1) = gr(2);

     /* new random numbers on even iterations */
     else fillRandom(gr, seed);

     /* next stock price */
     s = s * (a + (gr(1) * b));

     ssum += s; /* add it to sum */
    }

    /* avg stock price for simulation */
    sav = ssum * invnsteps;

    locsavsum += sav; /* add it to sum */

    if (sav > k) then loccsum += (sav-k);
    else  locpsum += (k-sav);
   }

   on Locales(0) do {
    savsum$ += locsavsum;
    csum$ += loccsum;
    psum$ += locpsum;
   }
  }
 }
}

/* calculate average stock price, call and put */
sbar = savsum$ * invnruns;
cfinal = csum$ * invnruns;
pfinal = psum$ * invnruns;
```

Figure 6: Abridged Black-Scholes implementation in Chapel.

4. Description of Hardware

One of the major aims of Chapel is portability and performance on different types of hardware architectures. We therefore chose to run our benchmarks on three distinct systems, which are introduced below.

4.1 Ness

EPCC's compute service *Ness*, which is mainly used for teaching purposes, is a small Linux system consisting of two 16-way SUN X4600 compute servers. It uses 8 dual-core AMD Operton (AMD64e) processors per node which have a clock speed of 2.6GHz and 2GB memory per chip.

The OS on *Ness* is Scientific Linux and the SUN Grid Engine is used as a batch scheduler. The GNU compiler (version 4.1.1) was used to build the Chapel compiler itself (which runs on the front-end of the system), whereas the Portland Croup Compiler Suite (PGI

version 7.0.7) was used for the compilation of the Chapel-generated C code, which runs on the backend.

The largest queue on Ness is 16 cores, therefore the only shared-memory performance of Chapel could be test on this system.

4.2 HPCx

HPCx [3] was the UK's national supercomputing service until the end of 2007 and remains in service until 2010. *HPCx* consists of 160 IBM eSERVER 575 compute nodes, which are set up as a shared-memory cluster with dedicated interconnect, in this case IBM's own *"Federation"* High Performance Switch. *HPCx* has a total of 2560 processing cores. HPCx uses Power5 chips at 1.5GHz clock speed. There are 8 dual-core chips per shared-memory node, with a total of 32GB memory per node.

The Power5 architecture allows for *simultaneous multi-threading* (SMT): each processor can execute two instruction streams simultaneously and thus run two threads concurrently. Each physical processor is split into two logical processors. Therefore, with SMT enabled, the the 16 processor nodes can in fact execute 32 threads.

The OS on HPCx is AIX version 5.3. The Parallel Operating Environment (POE) and LoadLeveler are used for batch scheduling. The XL compiler suite (version 8.0) was used to compile both the front-end and the back-end code. Chapel code can be run across nodes on HPCx by using GASNet's LAPI conduit.

4.3 HECToR

HECToR (High-End Computing Terascale Resource) [4] is the UK's current national supercomputing service. *HECToR* is a Cray XT4 based MPP, with an X2 vector unit. *HECToR* consists of 1416 compute blades, each housing four dual-core AMD Opteron chips (2.8GHz, with 6GB main memory per chip) – this amounts to a total of 11,328 processing cores. The interconnect used on HECToR is Cray's SeaStar2 – each compute chip controls a router chip – and is set up as a 3D torus.

The Chapel benchmarks were run both on the HECToR TDS (test and development system), which runs the Cray Linux Environment (CLE) version2.1.50HD, and the main HECToR service, which runs CLE version 2.0.62. The GNU compiler (version 4.1.2) and the PGI compiler (version 8.0.3) were used for the login and compute nodes respectively.

5. Shared Memory Performance

This section will look at the performance of our Chapel benchmarks on shared memory nodes, comparing them to equivalent implementation in C & Pthreads. The implementation of the parallel features on single locales is much more mature than on distributed memory, which is why the more challenging benchmarks that include large domains and datatstructures were only run on single nodes. In addition, the microbenchmarks were run to assess the performance of the task parallel features and the new version of the Chapel compiler

Microbenchmarks

The first notable thing about the results here is that there are no results from HPCx as the benchmarks just would not run on this platform (they would fail silently). It is suspected that this is related to stack size limits in the default setup of the AIX operating system and the XL compilers, but this could not be accurately diagnosed.

On Ness, the benchmarks were run on 1, 2, 4 and 8 threads in order to ensure the available threads divide equally into the 8 tasks that were being dispatched. By comparing the results in this fashion, there is a clear disparity in performance between two distinct groups of the microbenchmarks. The ones which used explicit `begin` statements (`begin_sync`, `begin_single`, `begin_param`) universally performed quite poorly, and the co-dispatch (`cobegin`, `coforall`) versions retained constant runtime. The underlying reasons for this are that the co-dispatch style implementations first setup the tasks, and then dispatch them all at once (thus incurring only one dispatch overhead), whereas the begin-style implementations will perform a dispatch every time begin statement is passed in the iteration. It should be noted that the performance character would change if there was any significant work within the synchronised block, since the begin-style benchmarks would get started earlier, rather than waiting for the end of the block to dispatch all the tasks at once.

Table 1 makes a direct comparison of Chapel versions 0.7 and 0.9 running the microbenchmarks on 1 up to 8 threads on Ness. It is clear from the runtimes that the performance on the begin statements was improved. However the co-dispatch statements have suffered a drop in performance – the runtimes are still mostly constant from 2 threads up, yet they are poorer than with the earlier version of the compiler. This could be caused by correctness fixes in the compiler which result in additional overheads.

Ness		1	2	4	8
begin_single	v0.7	-	2180.9	5057.9	7034.7
	v0.9	-	1966.3	4704.6	6219.7
begin_sync	v0.7	364.8	2253.1	4133.8	6830.7
	v0.9	351.9	1679.9	2685.3	6244.9
begin_param	v0.7	362.60	2243.4	5035.4	6772.8
	v0.9	354.72	1693.2	4553.1	5878.1
cobegin	v0.7	323.7	867.3	830.8	868.9
	v0.9	312.9	995.6	1021.1	1268.3
coforall	v0.7	322.56	847.81	806.45	810.82
	v0.9	335.86	1024.8	944.02	1298.5

Table 1: Comparsion of microbenchmark performance (runtimes in ms) on Ness - Chapel v0.7 and v0.9.

N-Queens

In the N-Queens benchmark implementations threads are assigned a tangible amount of work and the performance results follow a somewhat predictable pattern (see Table 2). The C performance is consistently around 4 times better than Chapel's. This reflects the fact that the runtime of the algorithm is dominated by actual computational work, as opposed to any overheads due to threading or memory leaks.

Even as the number of threads is increased, the performance ratio between the C/Pthreads and Chapel implementations remains very consistent. Between Ness and HPCx, the performance difference also remains fairly consistent as the number of threads is scaled up. The ratio of runtimes between Ness and HPCx is nearly equal to the clockspeed ratio on the two systems (2.6GHz vs. 1.5GHz). Looking at speedup, it becomes apparent that while the Chapel implementation scales near linearly on both Ness and HPCx (with superlinear scaling on Ness up to 8 threads), the C & Pthreads implementation shows consistently worse scaling. On Ness, the performance ratio between Chapel and C drops from 4.6 on 1 thread to 3.2 on 16 threads.

n=13	1	2	4	8	16
Ness, Chapel	6849ms	3265ms	1624ms	830ms	435ms
	1.00	2.10	4.22	8.25	15.73
HPCx, Chapel	13047ms	7038ms	3406ms	1799ms	920ms
	1.00	1.85	3.83	7.25	14.17
Ness, C	1477ms	779ms	433ms	248ms	133ms
	1.00	1.91	3.44	6.00	11.17

Table 2: Runtimes and speedup for N-Queens with n=13 on Ness and HPCx, up to 16 threads.

Strassen

Strassen's algorithm forms the most heavyweight of the benchmarks, exhibiting both task parallelism and nested parallelism, as well as a significant degree of floating point calculation. Additionally, it is the benchmark with the most complicated implementation - its kernel consists of around 100 lines of recursive array manipulation.

n=512	1	2	4	8	16
Ness, Chapel	1890ms	1116ms	531ms	307ms	208ms
	1.00	1.69	3.56	6.15	9.09
HPCx, Chapel	1975ms	1463ms	1246ms	1735ms	1935ms
	1.00	1.35	1.59	1.14	1.02
HECToR, Chapel	1779ms	926ms	-	-	-
	1.00	1.92	-	-	-
Ness, C	1660ms	1010ms	749ms	672ms	741ms
	1.00	1.64	2.22	2.47	2.24

Table 3: Runtimes and speedup for Strassen's algorithm (512x512 matrix) up to 16 threads.

Comparing the single threaded results in Chapel and C on different platforms (see Table 3), the performance is roughly equal. The C implementation achieves the fastest runtimes, yet the timings on HECToR, Ness and HPCx are not far off the mark.

Looking at the runtimes and the scaling up to 16 threads, it becomes clear that the behaviour of the Chapel implementation on HPCx and Ness is drastically different. The scaling is very good on Ness up to 4 threads, then dropping to a poorer, but still acceptable level. On HPCx however the code does not even scale up to 4 threads – the runtime on 16 threads is similar to that on a single thread. Additionally of note is the total failure of the C implementation to scale past 2 threads. This was a fairly complicated algorithm to implement in C, especially while trying to map its implementation to that of Chapel's using the pseudo-domain arrays and dynamic memory allocation, so it's quite feasible that this is due to programmer error. This very fact may support the Chapel argument for programmability in parallel languages. Pthreads is a very difficult environment to work with - in this particular case Chapel has provided an appropriate programming environment that made it possible to write code that ran faster than C.

Mandelbrot

The Mandelbrot benchmark is implemented in the task-decomposition style and introduces a significant floating point requirement. In terms of single threaded performance, the runtime ratio between the two systems is consistent and largely accountable to the difference in processors clock speeds (see Table 4).

In terms of time, the Mandelbrot benchmark shows certainly Chapel's best performance—on Ness, the Chapel implementation marginally beats the performance of the C implementation in both runtime and scaling up to 4 threads. The performance up to 16 threads is slightly poorer, but it is still a match for the C implementation.

The reason for the good performance of the Chapel code is that the kernel of the algorithm is based around a triply nested loop (technically, a 2D loop and a 1D loop), and doesn't contain Chapel features other than arithmetic — essentially, there's not much room for the compiler to go wrong in converting this into C. The HPCx result and the Ness results also remain a consistent factor apart, further suggesting this benchmark is entirely bound by floating point speed rather than of any operating system feature.

d=8	1	2	4	8	16
Ness, Chapel	1547ms	777ms	389ms	204ms	111ms
	1.00	1.99	3.97	7.60	13.91
HPCx, Chapel	2427ms	1216ms	612ms	323ms	170ms
	1.00	1.99	3.97	7.52	14.26
Ness, C	1552ms	778ms	395ms	196ms	104ms
	1.00	2.00	3.93	7.93	15.02
HPCx, C	2148ms	1074ms	539ms	282ms	146ms
	1.00	2.00	3.99	7.62	14.74

Table 4: Runtimes and speedup for the Mandelbrot benchmark up to 16 threads (image size 2048x2048, decomposition 8x8).

5. Distributed Memory Performance

Support for multi locale execution of parallel features has recently been added to the Chapel compiler. Two simple, embarrassingly parallel, benchmarks were written to assess Chapel ability to generate code that can run on distributed memory system. Chapel uses GASNet as its low-level communication network. GASNet offers different conduits for a wide range of architectures – in this case, the Portals conduit and the LAPI (Low-level Application Programming Interface) conduit were used on the Cray XT4 and the IBM Power5 respectively. Unfortunately, there were problems with the LAPI conduit and the use of RDMA (Remote Direct Memory Access) – an "unknown error" occurred at runtime, disabling RDMA. Unfortunately we have thus far not been able to fix this problem.

Pi

The Pi approximation algorithm is embarrassingly parallel, with a small number of floating point operations per iteration, and was thus expected to perform well with Chapel. The Chapel code was compared to a C & MPI implementation of the algorithm – see Table 5 for runtimes and scaling on both shared memory (1 locale) and distributed memory (2 locales on HPCx and 2 to 16 locales on HECToR TDS).

Looking at the shared memory performance on Ness, it becomes apparent that the Chapel implementation outperforms the C & MPI code – though neither implementation scales well on this platform. On HPCx, the performance of the multi-locale runs is interesting: all runs were set up to span across two nodes, thus a run with 4 threads puts 2 threads on each node. The scaling for these runs is super linear, because with increasing numbers of threads inside each node, the overheads created by the multi-locale executions become ever smaller. As a result, the ratio between the single and multi locale runtimes on 4 and 32 threads respectively is reduced from 4.8 to 1.3.

	1	2	4	8	16	32
Ness comm=none	112ms	56ms	37ms	18ms	17ms	-
	1.00	1.99	2.98	5.93	6.62	-
Ness, C & MPI	110ms	55ms	43ms	19ms	25ms	-
	1.00	1.98	2.52	5.60	4.39	-
HPCx, comm=none	181ms	91ms	46ms	24ms	23ms	-
	1.00	1.99	3.95	7.69	7.69	-
HPCx SMT, comm=none	-	-	-	-	23ms	17ms
	-	-	-	-	16.00	21.57
HPCx, comm=gasnet	-	-	222ms	75ms	38ms	22ms
	-	-	4.00	11.77	23.06	40.46
HECToR TDS, comm=none	112ms	56ms	-	-	-	-
	1.00	1.99	-	-	-	-
HECToR TDS, comm=gasnet	-	112ms	85ms	43ms	22ms	13ms
	-	2.00	2.64	5.17	14.16	17.68
HECToR TDS, C & MPI	111ms	56ms	28ms	14ms	8.7ms	7ms
	1.00	1.98	3.91	7.98	12.71	15.65

Table 5: Runtimes and speedup for the Pi approximation benchmark, run with 8.4 million iterations.

On HECToR TDS, the performance of the C & MPI code and the single node performance of Chapel are near identical. Compared to the multi locale performance however, the MPI code is twice as fast for all runs up to 32 threads. We suspect this is because of overheads introduced by the use of GASNet.

Black-Scholes

For the Black-Scholes algorithm, we are not only comparing the performance on different architectures, but also the difference in performance of Chapel code written for serial, single and multi locale execution. The serial code contains no parallel statements at all and is basically a direct translation of a serial C code into Chapel. The single locale code does contain parallel statements, but the code that distributes work among locales was removed. Some of the variable instantiations that need to be done on locale and thread level (i.e. inside the `coforall` statements) in the distributed code were moved out of these blocks.

Table 6 shows the runtimes and speedup of the three different versions of the algorithm on Ness, using only one locale. It is noticeable that on one thread both the single and the multi locale implementations are quicker than the serial version. This is possibly due to compiler optimisation that are applied for blocks of independent

computation, which are highlighted in the parallel versions of the code (with a `cobegin` statement for instance). While neither parallel implementation scales well, the single locale code performs considerably better than the multi locale version. This could be explained by overheads that are introduced unnecessarily by the distributed code. The results on Ness are as would be expected.

	1	2	4	8	16
Ness, serial comm=none	35ms	-	-	-	-
	-	-	-	-	-
Ness, single comm=none	29ms	16ms	12ms	7ms	4ms
	1.00	1.81	2.35	3.80	6.74
Ness, multi comm=none	29ms	14ms	10ms	11ms	8ms
	1.00	1.97	2.96	2.46	3.54

Table 6: Runtimes and speedup on Ness for serial, single loacle and multi locale implementation of the Black-Scholes algorithm (32,000 runs over 91 days).

Performance on HPCx (see Table 7) however is surprising. The serial performance is in line with that of Ness, yet on a single node the single locale code runs slower by a factor of nearly 2.5 compared to the multi locale code. It is unclear why this might be the case. On 16 threads the multi locale implementation on HPCx is *faster* than the single locale code on Ness, which is completely unexpected. Further investigation is required to fully understand the reasons behind this.

The performance on the algorithm across nodes (2 locales were used here) is quite poor, yet the code shows good scaling up to 32 threads. The slow runtimes are likely due to the lack of RDMA and the underpopulation of the nodes – the overheads of setting up the work distribution exceeds the benefits of doing so. This becomes by looking at the runtime ratios: on 4 threads, the ratio between the single locale and the distributed timings is a factor of 9, yet on 32 threads this is reduced to 3.

	1	2	4	8	16	32
HPCx, serial comm=none	37ms	-	-	-	-	-
	-	-	-	-	-	-
HPCx, single comm=none	63ms	32ms	17ms	9.4ms	8.8ms	9.2ms SMT
	1.00	1.97	3.68	6.71	7.16	6.88
HPCx, multi comm=none	26ms	13ms	7ms	4.5ms	5.3ms	6.5ms SMT
	1.00	1.94	3.70	5.86	5.01	3.96
HPCx, multi comm=gasnet	-	-	155ms	54ms	28ms	27ms
	-	-	4.00	11.51	22.04	22.61

Table 7: Runtimes and speedup on HPCx for serial, single loacle and multi locale implementation of the Black-Scholes algorithm (32,000 runs over 91 days).

On HECToR, both the serial and the single locale codes perform as expected and in line with what was seen on Ness. The multi locale however shows a rather disappointing performance – the scaling stops at 16 threads and the runtimes are only slightly better than those seen on HPCx. The problem size that was run (32,000 iterations over 91 timesteps each) is a realistic size and the computation should be sufficient to mask any communication overheads. Again, further investigation is needed to be able to explain the behaviour of the multi locale implementation.

	1	2	4	8	16	32
HECToR, serial comm=none	31ms	-	-	-	-	-
	-	-	-	-	-	-
HECToR, single comm=none	28ms	21ms	-	-	-	-
	1.00	1.31	-	-	-	-
HECToR, multi comm=gasnet	-	132ms	100ms	51ms	27ms	17ms
	-	2.00	2.63	5.15	9.74	15.43

Table 8: Runtimes and speedup on HECToR for serial, single loacle and multi locale implementation of the Black-Scholes algorithm (32,000 runs over 91 days).

6. Conclusions

Running both the single locale and the multi locale benchmarks has provided us with an interesting picture of what the current implementation of Chapel is capable of. Overall, the most difficulties were encountered on HPCx, which does not come as a surprise – this is a type of system that the Chapel development team has only recently been given access to and thus they have only been able to put very limited effort into its support.

While the Chapel implementations of the different benchmarks are often outperformed by the C implementations of the same algorithm (whether parallelised with Pthreads or MPI), the general performance patterns of this new language are very encouraging, especially inside shared memory nodes.

As was said at the beginning of this paper, Chapel is still very much work in progress. Performance optimisation has so far not been a priority of the development team and the current implementation still suffers from problems such as memory leaks. However such issues are minor (and fixable). The important thing is that Chapel offers a novel way of approaching parallel programming and the performance that the language shows at this early stage is very positive indeed.

Acknowledgments

The authors would like to thank the Chapel development team for their continued help and invaluable input.

References

[1] HPCS: http://www.highproductivity.org/

[2] Chapel: http://chapel.cs.washington.edu/

[3] HPCx: http://www.hpcx.ac.uk

[4] HECToR: http://www.hector.ac.uk

About the Authors

Michèle Weiland works as an Applications Consultant at EPCC, the supercomputing centre at The University of Edinburgh. She can be reached at EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK; email: m.weiland@epcc.ed.ac.uk or at.

Thom Haddow is currently working on his PhD in the Distributed Software Engineering group at Imperial College in London. Thom can be reached at Department of Computing, 346 Huxley Building, Imperial College, SW7 2AZ; email: thaddow@doc.ic.ac.uk.