Implementing Stencil Problems in Chapel: An Experience Report

Per Fuchs
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
per.fuchs@gmail.com

Pieter Hijma
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
p.hijma@vu.nl
Universiteit van Amsterdam
Amsterdam, The Netherlands
p.hijma@uva.nl

Clemens Grelck
Universiteit van Amsterdam
Amsterdam, The Netherlands
c.grelck@uva.nl

Abstract

Stencil operations represent a fundamental class of algorithms in high-performance computing. We are interested in what level of performance can be expected from a high-productivity language such as Chapel. To this effect we discuss four different implementations of a generic stencil operation with a convergence check after each iteration. We start with a sequential implementation followed by a global-view implementation that we experiment with both on a 16-core multi-core system as well as on a cluster with up to 16 such nodes using domain maps. We finish with a local-view implementation that explicitly encodes all design decisions with respect to parallel execution. This paper is set up as a two stage experience report: We mainly report our findings from the users' perspective without any feedback from the Chapel implementers. We then report additional analysis performed under guidance of the Chapel team. Our experimental findings show that Chapel performs as expected on a single node. However, it does not achieve the expected levels of performance on our multi-node system, neither with the data-parallel global-view approach, nor with the task-parallel local-view code. We discuss the root causes of our reduced performance in detail and report possible solutions.

CCS Concepts • **Software and its engineering → Parallel programming languages**.

Keywords stencil operation, experience report, Chapel, performance study, cluster computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHIUW '19, June 22, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6800-1/19/06…$15.00
https://doi.org/10.1145/3329722.3330146

ACM Reference Format:
Per Fuchs, Pieter Hijma, and Clemens Grelck. 2019. Implementing Stencil Problems in Chapel: An Experience Report. In *Proceedings of the ACM SIGPLAN 6th Chapel Implementers and Users Workshop (CHIUW '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3329722.3330146

1 Introduction

Chapel [6, 7] is a high-productivity programming language that promises multi-scale parallel programming from multi-core systems to large-scale compute clusters. Support for distributed memory architectures is based on the *Partitioned Global Address Space (PGAS)* model. Chapel addresses parallel execution and the corresponding issues in an explicit way, but offers programmers the choice between a rather explicit task-based programming style and a more implicit data-parallel programming style. Data distribution is governed by user-defined *domain maps*, and it is a design characteristic of Chapel to use the very same interface for standard and custom data distributions.

We have been teaching Chapel programming in Amsterdam for many years, initially at the University of Amsterdam within the course *Programming Concurrent Systems* and currently in the courses *Programming Large-Scale Parallel Systems* and *Parallel Programming Practical* at the VU Amsterdam as part of a joint degree program of the two Amsterdam universities. In these courses students implement a fairly simple stencil operation, namely a simplified variant of heat diffusion on the surface of a cylinder, using a variety of parallel programming approaches including Chapel. Stencil codes represent a fundamental class of algorithms with endless applications in the compute-intensive application domains that Chapel aims at.

In the classroom we advocate Chapel as making the step from sequential code to parallel code nearly non-existent and the step from single node parallel code to multi-node parallel code particularly simple. However, experience across a number of iterations has shown initial enthusiasm to turn into disappointment as soon as performance measurements show their results. In this paper, we aim to share our experiences with stencil computations in Chapel and we perform a deep

analysis of the obtained performance with help from the Chapel team. In contrast to most previous research (Sec. 2), we particularly target multi-locale scenarios, the main target for a PGAS language. We show a total of four Chapel implementations of our flavor of the heat dissipation problem (Sec. 3) and investigate their performance:

a) a carefully engineered sequential version as a baseline (Sec. 4);
b) a textbook style single-node implementation (Sec. 5);
c) a global-view data-parallel implementation with domain maps for multi-locale execution (Sec. 6); and
d) a local-view task-parallel implementation (Sec. 7).

We discuss our results in Sec. 8, and conclude that although Chapel offers promising and highly powerful constructs to design parallel applications, Chapel's multi-locale performance is below our expectations as a result of reduced support for our hardware configuration and lack of investment in the Chapel compiler for the locality assertions that we use.

2 Related Work

Bertolacci et al. show that Chapel's parallel iterators, a programming construct that allows one to define iterators for parallel loops, can provide a way to separate the execution schedule from the computation [4]. This paper compares the effectiveness of Chapel compared to other languages such as C or C++ without taking multi-locale execution into account, while our work investigates several Chapel versions of stencil operations for single- and multi-locale execution.

Burkhart et al. [5] compare stencil operations implementations (among others based on work by Barrett et al. [3]) with different languages such as Java, OpenMP, and MPI in a classroom setting. In addition, they compare multiple versions of the Chapel implementation with other PGAS languages (UPC and Co-array Fortran). Our experiences are also based on the classroom and similar to our work, they compare various versions but only evaluate single-node versions, while we compare multi-node versions.

Mittal explores Successive Over-relaxation in the languages Chapel, D, and Go [12]. All versions have modest speed-ups on a single node. We study four different versions on single node and multiple nodes.

The Chapel team has been investigating the performance of stencil operations since 2017 using the Intel Parallel Research Kernel benchmark [13]. Over the past years, the performance of stencil operations has steadily increased approaching MPI/OpenMP performance. To achieve this, it was necessary to apply more fine-grained locality control in the code, and to improve the compiler and runtime, for example for NUMA affinity in the Cray-specific ugni communication layer. In contrast to the Chapel team, we do not have intricate knowledge about the compiler and runtime and we do not use the Cray-specific ugni communication layer. The reports on Stencil performance can be accessed in the performance release notes of version 1.16, 1.17, and 1.19 [8].

3 Problem Definition

We use a simplified version of heat diffusion on the surface of a cylinder. Our focus is not on the correct modeling of the underlying physics, but on stencil operation that exposes various characteristics of common stencil operations.

We discretize the cylinder surface as a temperature matrix of $N \times M$ grid points. Each grid point is associated with a temperature value, specified as a double precision floating point number. In each iteration the temperature value of each grid point is recomputed as the weighted average of the previous iteration's value, its four direct and its four diagonal neighbor grid points. In addition we use an $N \times M$ read-only conductivity coefficient matrix of double precision floating point numbers to model the material's conductivity.

The simulation is finished after a given maximum number of iterations or when the simulation has sufficiently converged, i.e. all absolute differences between the old and new temperature values across the entire temperature matrix remain below a given threshold ϵ, whichever occurs first.

3.1 Experimental Setup

All experiments have been performed on the DAS-5 cluster computer [2] of the VU Amsterdam. Each node in the cluster contains two sockets with an *Intel Xeon E5-2630 CPU* with 2.4 GHz with 8 cores with 2 hyperthreads per core. The nodes are connected with a 48 Gbps InfiniBand network. The operating system is 64bit CentOS 7.4 with a 3.10.0 kernel. The Chapel version used is the latest stable release (1.19.0) and all code was compiled with the `--fast` flag that turns on all optimizations.

Chapel itself and the C code generated by Chapel is compiled with GCC 6.4.0, the latest version available on our system. Two run-times were created: one for single-locale execution and one for multi-locale execution. We set the environment variable for the target CPU to `native` for both run-times. For the multi-locale run-time, we set the environment variables such that we use the GASNet library with substrate `ibv` and maximum of physical memory of `1G`. Additionally, we set some environment variables to make use of the Slurm launch code generated by Chapel, but these environment variables do not influence the performance.

All numbers of the performance experiments are averaged over three runs executing all 500 iterations. The standard deviations have been verified for each average and in all cases, the standard deviation was low for the numbers that we present in this paper. In other words, all our results are highly reproducible. The code repository is available with a permissive license [1] with a permanent link to the specific versions used in this paper [9] in the form of a DOI.

```
config const N = 8192;
config const M = 8192;
config const I = 500;
config const E = 0.01;

const HaloDomain: domain(2) = {0..N+1, 0..M+1};
const CylinderDomain: subdomain(HaloDomain) = {1..N, 1..M};

const LeftHalo: subdomain(HaloDomain) = {1..N, 0..0};
const RightHalo: subdomain(HaloDomain) = {1..N, M+1..M+1};
const UpperHalo: subdomain(HaloDomain) = {0..0, 0..M+1};
const LowerHalo: subdomain(HaloDomain) = {N+1..N+1, 0..M+1};

const LeftColumn: subdomain(HaloDomain) = {1..N, 1..1};
const RightColumn: subdomain(HaloDomain) = {1..N, M..M};
const UpperRow: subdomain(HaloDomain) = {1..1, 0..M+1};
const LowerRow: subdomain(HaloDomain) = {N..N, 0..M+1};

class Cylinder {
  var temperature : [HaloDomain] real;
}

const factor_direct_neighbors : real =
  sqrt(2.0) / (sqrt(2.0) + 1) / 4;
const factor_diagonal_neighbors : real =
  1 / (sqrt(2.0) + 1) / 4;
```

Figure 1. Global definitions of configuration constants, domains, and classes.

4 Sequential Version

We begin our Chapel implementation of the heat dissipation problem with a number of (configuration) constants that are shared across all our different variants. Fig. 1 shows the most relevant ones.

To serve as a performance baseline we implement a completely sequential version of our code. Chapel is a language with inherent parallel constructs, so creating a sequential version is not natural. Therefore, we show in Fig. 2 the computational kernel of our heat dissipation problem in the way one would write a parallel version in Chapel naturally.

Within the sequential time iteration loop we consistently use `forall`-loops to exploit the abundant data parallelism in our code. Following the update of the halo columns we see a classical stencil code with the weight handling as described in the previous section. We implement the convergence check with a Chapel reduction, again in a data-parallel way, and break out of the loop if sufficient convergence is detected.

A potential pitfall for novice Chapel programmers is the swap operator application that completes the loop body. We deliberately wrap our array buffers in a class `Cylinder`, as shown in Fig. 1, to achieve a pointer swap here. This could easily be overlooked with the effect that the two (usually large) buffers are swapped call-by-value.

In principle, we could simply run our experiments with setting the `CHPL_RT_NUM_THREADS_PER_LOCALE` environment variable to ensure that no data parallelism is used. In the absence of knowledge about Chapel compiler implementation internals we suspect that only dynamically de-activating

```
for iteration in 1..I {

  forall (i,j) in zip(LeftHalo, RightColumn) {
    src.temperature[i] = src.temperature[j];
  }
  forall (i,j) in zip(RightHalo, LeftColumn) {
    src.temperature[i] = src.temperature[j];
  }

  forall (i, j) in CylinderDomain {
    var weight = conductivity[i, j];
    var remaining_weight = 1 - weight;

    dst.temperature[i, j] =
      weight * src.temperature[i, j] +
      // four direct neighbors
      remaining_weight * factor_direct_neighbors *
      (src.temperature[i-1, j] +
       src.temperature[i, j+1] +
       src.temperature[i+1, j] +
       src.temperature[i, j-1]) +
      // four diagonal neighbors
      remaining_weight * factor_diagonal_neighbors *
      (src.temperature[i-1, j-1] +
       src.temperature[i-1, j+1] +
       src.temperature[i+1, j+1] +
       src.temperature[i+1, j-1]);
  }

  max_difference = max reduce [ij in CylinderDomain]
    abs(dst.temperature[ij] - src.temperature[ij]);

  if max_difference < E break;

  src <=> dst;
}
```

Figure 2. Global-view single-locale textbook implementation.

parallel execution potentially leaves binary code for the organization of parallel program execution behind that may cause overhead. Hence, we decided against this simple solution and replaced all implicit and explicit data parallel constructs with sequential ones.

Concretely, we replaced all `forall`-loops by corresponding `for-loops` (lines 3, 6, and 10). Likewise, the Chapel reduction is replaced by yet another straightforward `for-loop` in line 30.

At this point we need to make a clear design decision. It is possible to reduce the `max_difference` value outside of the `forall`-loop as is shown in Fig. 2 or we could reduce the value within the `forall`-loop as is shown in Fig. 3. To find out what works best, we experimented with both options.

4.1 Expected Results

In our opinion, the latter version with the reduction in the loop is the most natural version for a sequential implementation, because it would make better use of the cache as all elements of the two arrays only have to be accessed once at the cost of a small amount of unpredictable control-flow in the main loop. In contrast, having a separate reduce loop

```
1  for iteration in 1..I {
2
3    // exchange columns
4
5    for (i, j) in CylinderDomain {
6      const weight = conductivity[i, j];
7      const remaining_weight = 1 - weight;
8      const oldTemp = src.temperature[i, j];
9
10     const newTemp =
11       weight * oldTemp +
12       // neighbor computations
13
14     max_difference =
15       max(max_difference, abs(newTemp - oldTemp));
16     dst.temperature[i, j] = newTemp;
17   }
18
19   if max_difference < E break;
20
21   src <=> dst;
22 }
```

Figure 3. Sequential implementation with an in-loop reduction.

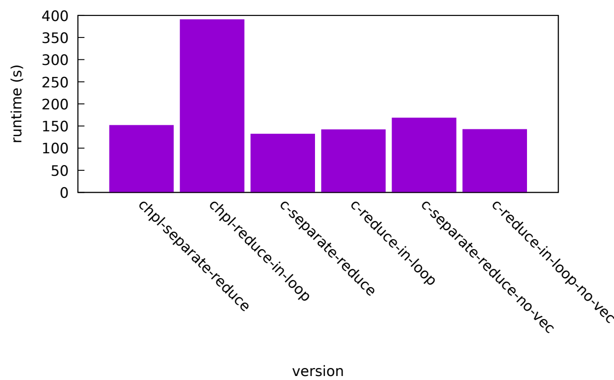

Figure 4. Run-times of the Chapel and C versions having a reduction within the loop or a separate reduction on an input of 8192×8192.

would make the stencil loop more predictable, but the reduction would need to access all elements of the two arrays a second time.

4.2 Actual Results

To our surprise, the two versions show a large performance gap. The version with a separate reduction outside the stencil loop needs about 151 s while the version with the reduction inside the stencil loop, the one we expected to perform better, needs about 390 s.

To get a sense of "ground truth", we compare it against a C version as well. The separate reduce version needs about 131 s while the in-place reduction version needs 141 s, again against our expectations.

We investigated why the performance of the reduction within the loop is so low. After careful inspection of the compiler output of the Chapel and C compilers, we learned that the C compiler is able to vectorize the stencil loop, but only with the reduction outside of the stencil loop.

To better understand the effects of vectorization, we turned it off in the C versions. This yields 168 s for the version with the separate reduction and 142 s for the version with the reduction inside of the stencil loop, which is in line with our expectations. Figure 4 puts all measurements into context.

Although we now understand why the version with a separate reduction is faster than the version with the reduction inside the stencil loop, we still cannot explain why the performance drops by a factor of more than two while the C version does not have this problem. Disabling vectorization for the Chapel version with the separate reduction leads to a performance similar to the 'in-loop' version. Apparently, vectorization is an important optimization for Chapel code.

Since the compiler on our system is two years old, we tried a more recent compiler (GCC 9.1.0) on a different machine to investigate whether the vectorization capabilities of the C compiler are an important factor or whether the Chapel compiler generates code that is difficult to vectorize in general. We can confirm that the more recent compiler is able to successfully vectorize the code both for the reduction inside the loop and outside the loop. This tells us that our experience over the last few years could have been improved by using the most recent compiler and we recommend the Chapel team to add a note about the influence of the C compiler on the performance in their documentation.

We choose version `chpl-separate-reduce` as our baseline. It can be argued that this is not a true sequential version because the C compiler applies vectorization in the most important loop. However, in spite of this, we decided to use this version as a baseline based on the following arguments: From the perspective of Chapel, this version is purely sequential as no parallelism is expressed in the code.

Furthermore, the C compiler is external to the Chapel ecosystem similar to the way that the hardware on which the code runs, is external to the Chapel ecosystem. The fact that a C compiler is smart enough to extract parallelism from a sequential loop is outside of the control of the Chapel compiler, similar to the fact that it is outside of the control of the Chapel compiler that hardware extracts parallelism from an instruction stream on superscalar architectures. With modern architectures truly sequential execution does simply not exist.

Finally, the fastest Chapel version is close to the performance of the C version, regardless of the way the reduction is done and regardless of whether vectorization is turned on or off. Therefore, we choose this as our baseline as it helps us to come to more precise conclusions. We understand that there are also arguments against choosing this baseline.

Table 1. Execution times in seconds and speedup of the single-locale version compared to the sequential version for an input size of 8192×8192. Below that the required bandwidth in GB/s for this execution time and the maximum bandwidth from the STREAM benchmark (GB/s).

Threads	1	2	4	8	16
par-global-single	152.20	75.23	45.58	35.85	24.99
Speedup	0.99	2.01	3.32	4.22	6.06
BW STREAM	18.8	40.4	71.1	79.9	78.0
Required BW	22.9	46.4	76.6	97.3	139.6

5 Global View Single Locale

Figure 2 shows the computational kernel of our data-parallel global-view version for a single locale. In contrast to the sequential version we use `forall`-loops instead of `for`-loops in lines 3, 6, and 10, and a Chapel reduce expression in line 30 as is shown in the figure.

5.1 Expected Results

Since a node in our cluster has 2 CPUs with 8 cores each, we measured the execution for this version with 1 to 16 threads controlled by setting the `CHPL_RT_NUM_THREADS_PER_LOCALE` environment variable. The number of threads of 16 is also chosen as a default by the Chapel run-time when the number of threads is not specified. This is sensible for our use-case because we should not benefit from hyperthreading as the code employs a large amount of floating point instructions.

Since our main loop touches three large arrays and the order of magnitude of array accesses and computation is the same, it is likely that our program is bandwidth limited. To understand what a realistic speedup is, we apply the Roofline model [14] by counting the number of accesses (lines 4, 7, 11, 14, 15, 18-21, 24-27, and 31 in Fig. 2), measuring the maximum bandwidth with the STREAM benchmark [11] for 1 to 16 threads, and deriving execution times from these numbers. For example, 1 thread achieves a maximum of 18.8 GB/s and 16 threads a maximum of 78.0 GB/s on the STREAM benchmark which corresponds to execution times 186 s and 44.7 s respectively. Since the sequential version achieves 151 s, there must be effect from the cache and as such, these estimated execution times can be considered pessimistic.

We also applied an optimistic scenario where all stencil accesses are considered to be in the cache (lines 18-21 and 24-27 left out). This leads to an execution time of 71.5 s for 1 thread and 17.2 s for 16 threads, which means that the maximum speedup we can expect is 8.8 in the optimistic scenario.

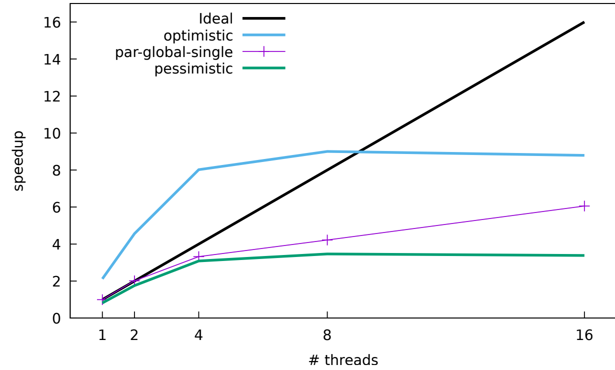

Figure 5. Speedup and pessimistic and optimistic bounds for `par-global-single` compared to the sequential version on an input of 8192×8192.

5.2 Actual Results

Table 1 presents the execution time and speedup for each configuration. It also shows the maximum bandwidth measured with the STREAM benchmark and the bandwidth that would be required to achieve the measured execution time. The difference between these numbers gives an indication of the effect of the cache (if the required bandwidth is greater than the maximum bandwidth, this must be due to caching).

Figure 5 visualizes the speedups and shows that the measured speedup scales up to about 4 threads and then decreases, precisely where the required bandwidth approaches the maximum bandwidth possible on our system. The measured speedup is also in between our optimistic and pessimistic scenario, where the cache effect becomes larger for more threads. This is logical as with an increased number of cores, more cache becomes available. This gives us high confidence that Chapel achieves the maximum performance possible and we will use the execution time with 16 threads as a baseline for the multi-locale versions.

6 Global View Multi Locale

Following Chapel's multi-resolution approach to parallelism, we next aim at running our code on a cluster of machines. This requires the distribution of our three arrays (temperature source and destination as well as conductivity) over the distributed memories of the cluster machines. The design of Chapel very much facilitates this step as it boils down to annotating the domains with a predefined distribution using the `dmapped` keyword. The predefined block distribution seems like a logical choice for our problem because stencil operations benefit from both temporal and spatial locality.

The Chapel library offers a special version of the block distribution for stencil operations, named `Stencil`. This distribution provides a cache for a configurable amount of rows and columns around the blocks that are owned by a cluster node. These rows and columns can be accessed locally

```
1  use StencilDist;
2
3  var MyLocaleView = {0..numLocales-1, 0..0};
4  var MyLocales =
5    reshape(Locales[0..numLocales-1], MyLocaleView);
6
7  const HaloDomain: domain(2) dmapped Stencil(
8    boundingBox = {0..N+1, 0..M+1},
9    targetLocales = MyLocales,
10   fluff=(1,1)) = {0..N+1, 0..M+1};
```

Figure 6. Data distribution for the global-view-multi-locale version.

```
1  for iteration in 1..I {
2
3    ref s = src.temperature;
4    ref d = dst.temperature;
5
6    // exchange columns
7
8    forall (i, j) in CylinderDomain {
9      local {
10       var weight = conductivity[i, j];
11       var remaining_weight = 1 - weight;
12
13       d[i, j] =
14         weight * s[i, j] +
15         // neighbor computations
16     }
17   }
18   d.updateFluff();
19
20   max_difference = max reduce [ij in CylinderDomain]
21     abs(d[ij] - s[ij]);
22
23   if max_difference < E break;
24
25   src <=> dst;
26 }
```

Figure 7. Global-view multi-locale implementation.

during the stencil operation while updates are meant to happen once after the stencil operation has completed with an `updateFluff()` operation.

A true block-wise distribution however, would hurt the performance, since it would mean that columns should be exchanged between the various machines. Therefore, we decided to distribute the arrays row-wise to prevent column access in the update. Since columns are not aligned in the cache, it may trigger many remote messages, while rows are aligned in the cache and can be sent with one message.

Figure 6 shows that we use the `StencilDist` distribution. We create a view on the locales with one column and `numLocales` rows and then use it to reshape the locales. The `HaloDomain` is domain mapped with a stencil distribution with our view on the locales, resulting in a row-wise distribution of our main domains.

Figure 7 shows that the stencil distribution allows us to assert that the stencil computation can be performed completely locally (line 9) and that the rows can be exchanged after the `forall`-loop with the `updateFluff()` call (line 18). Please note that because of our row-wise distribution, the column exchanges are also done completely locally (line 6, identical to lines 3–8 in Fig. 2).

In conclusion, it is enough to change a few lines of code to get a program that is able to run on multiple machines with Chapel. Under the hood, the compiler adds a global address space layer and with that the ability for remote writes and reads.

Table 2. Execution times in seconds, speedup and efficiency of the global view, multi-locale version compared to the single-locale version with 16 threads for an input size of 8192×8192.

Nodes	1	2	4	8	16
par-global-multi	158.68	83.13	40.08	21.10	11.03
Speedup	0.16	0.30	0.62	1.18	2.27
Efficiency	0.16	0.15	0.16	0.15	0.14

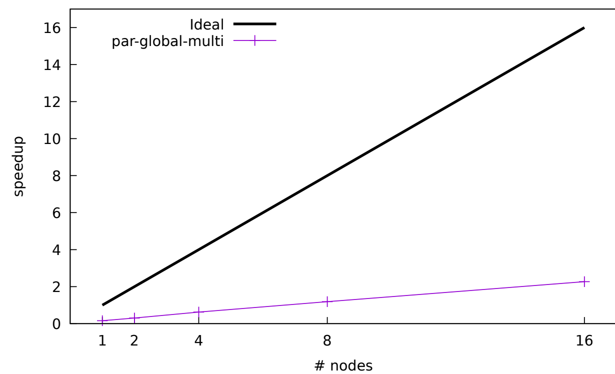

Figure 8. Speedup of the global view multi locale version compared to the single locale version with 16 threads on an input of 8192×8192.

6.1 Expected Results

Theoretically, this should result in reasonable speedups over the par-global-single version: The workload can be perfectly statically balanced and the amount of computation is an order of magnitude larger than the communication. Additionally, the communication increases only linearly with more locales, can be implemented as a bulk operation and is performed using our low-latency, high-throughput InfiniBand network.

The main point of synchronization, the reduction, can efficiently be performed in parallel, as explained in Sec. 5. Furthermore, by choosing large data sizes, we can mitigate the effects of the fact that part of our main computation is serial.

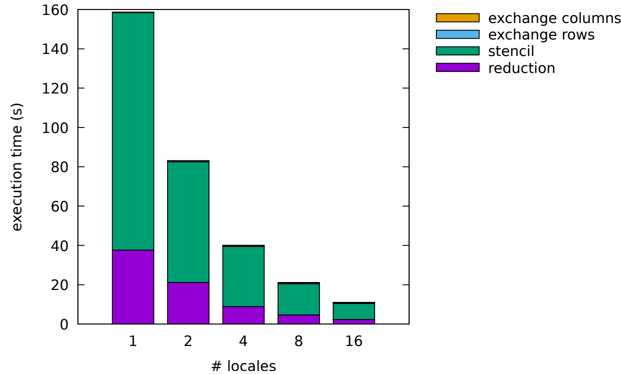

Figure 9. Breakdown of the execution times of the four stages of `par-global-multi` on an input of 8192×8192 with 16 parallel tasks per locale.

6.2 Actual Results

Unfortunately, the results are below expectations. Table 2 shows the execution times, speedups and efficiencies of the multi-locale version compared to the single-locale version with 16 threads. Figure 8 shows the speedup graphically.

On first glance, an execution time of 158.7 s for one locale seems reasonable compared to the sequential and single-locale version, but this version should also use 16 threads per node to compute the stencil. So in fact, we would at best expect an execution time of 25 s (the performance of the single-locale version with 16 threads, see Table 1) and because of overhead a possibly somewhat higher execution time.

Although the performance is below expectations, this version scales very well. Figure 9 shows in a breakdown of the execution times for various stages that with more locales, the highly compute-intensive parts scale very well and communication of the rows requires virtually no time in comparison.

The stencil computations contribute for a large part to the reduced performance despite the fact that 16 threads are available and that the "stencil" stage is defined completely within a local block. This should mean that it is not necessary to perform run-time checks for remote memory in this part and in principle the code should perform as well as the single-locale versions from the point-of-view of a user.

6.3 Feedback from the Chapel team

After analysis with help from the Chapel team, we learned that the local block does not do what we expected but that we can achieve what we wanted by removing the local block and use the undocumented feature of asserting that an access is local by using for example `d.localAccess[i,j]` in line 13 in Fig. 7. Using this feature improved the performance by a factor of 3.5 for 1 locale to a factor 2.4 for 16 locales compared to results in Table 2.

```
1 var tmp : [CylinderDomain] real;
2
3 local {
4   tmp = [ij in CylinderDomain] abs(d[ij] - s[ij]);
5 }
6
7 max_difference = max reduce tmp;
```

Figure 10. A local block for the reduce expression.

We want to note that we prefer the `.localAccess()` feature over the `local` blocks, because it gives more fine-grained control. For instance, in line 21 in Fig. 7 it allows us to assert that the `d` and `s` accesses are local whereas we refrained from the solution shown in Fig. 10 where we create a local block with a temporary variable to assert that the implicit `forall` loop contains only local accesses.

In addition, our performance is hampered by the fact that our cluster has two sockets per node. GASNet is not NUMA aware and registers remote memory on one socket resulting in a penalty for the cores on the other socket. One suggestion was to use GASNet's MPI conduit that does not have the NUMA problem because memory need not be registered with the network. Another suggestion was to either configure Chapel to use GASNet's 'fast' segment or use the OpenFabrics [10] communication layer. Both methods may improve network performance although these solutions are also not NUMA aware. However, all methods require us to use an MPI based launcher and unfortunately, for an unknown reason we are unable to configure this launcher to use more than one core per process on a node in our system, resulting in reduced performance.

7 Local View Multi Locale

This program written in the local view programming paradigm allows us to control how and when we want to distribute data between locales. In Fig. 11 we declare a domain `Row`, a class `Communicator` that contains the first and last row for a locale, and the array of communicators with domain `LocaleSpace`, automatically distributing the values of the array.

We then start our main `coforall` loop in which we iterate over the locales and execute the locale-specific code. In the first phase, we create the local communicators and distribute them. After that, each locale accesses the communicators of their neighboring locales. With this mechanism, locales can exchange rows with their neighbors by just writing to the row arrays in their respective communicators.

The local view programming paradigm also allows us to isolate the compute-intensive regions of the code to ensure that these regions only work on local data. In our version we make use of a domain mapping of the `HaloDomain` in the same way we did this in the global-view version (see Fig. 6) but in this case we use the `BlockDist` distribution instead of the `StencilDist`. Given this data distribution we

```
1  const Row : domain(1) = {0..M+1};
2
3  class Communicator {
4    var firstRow: [Row] real;
5    var lastRow: [Row] real;
6  }
7
8  var communicators: [LocaleSpace] Communicator;
9
10 var globalIterations = 0;
11 var globalMaxDiff: real = min(real);
12
13 coforall l in Locales with
14     (ref globalIterations, ref globalMaxDiff) {
15   on l {
16     const myCommunicator = new unmanaged Communicator();
17     communicators[here.id] = myCommunicator;
18
19     allLocalesBarrier.barrier();
20     const isLast = here.id == LocaleSpace.last;
21     const isFirst = here.id == LocaleSpace.first;
22
23     const beforeCommunicator = if !isFirst
24       then communicators[here.id - 1]
25       else new unmanaged Communicator();
26     const nextCommunicator = if !isLast
27       then communicators[here.id + 1]
28       else new unmanaged Communicator();
```

Figure 11. Definition of communicators to exchange rows between locales for the multi-locale, local view version.

```
1 const LocalCylinderDomain = CylinderDomain.localSubdomain();
2 const LocalHaloDomain = LocalCylinderDomain.expand(1, 1);
3
4 const LocalUpperRow = LocalHaloDomain.dim(1).first;
5 const LocalLowerRow = LocalHaloDomain.dim(1).last;
```

Figure 12. Definition of local domains.

can obtain the local dimensions automatically by calling `localSubdomain()` on the distributed global domain as can be seen in Fig. 12. This functionality is very convenient in Chapel as it means that we do not need to compute the domains manually.

Figure 13 shows the main loop for the iterations. Line 5 shows a local block with the main `forall`-loop that performs the stencil computation and performs the exchange of the columns locally (similar to lines 3–8 in Fig. 2).

The communication after the stencil operation is straightforward but its implementation involves some important decisions. First, it should be noted that we measured remote writes to be much faster than remote reads. Intuitively, this can be explained because a remote write can be implemented in a "fire and forget" way while a remote read has to wait for an answer. Resulting from this knowledge, the neighboring row exchanges are implemented as remote writes (lines 24, 25 and 34–43) and all local difference values are stored on locale 0, obtained from all locales (line 30) which allows the reduce operation to work locally without a remote read (line

```
1  do {
2    ref s = src.temperature;
3    ref d = dst.temperature;
4
5    local {
6      local_max_difference = min(real);
7
8      forall (i, j) in LocalCylinderDomain with
9          (max reduce local_max_difference) {
10       const weight = localConductivity[i, j];
11       const remaining_weight = 1 - weight;
12       const oldTemp = s[i, j];
13
14       const newTemp = weight * oldTemp +
15         // neighbor computations
16
17       d[i, j] = newTemp;
18       local_max_difference = max(local_max_difference,
19           abs(new_temp - old_temp));
20     }
21     // exhange columns
22   }
23
24   nextCommunicator.firstRow = d[LocalLowerRow - 1, ..];
25   beforeCommunicator.lastRow = d[LocalUpperRow + 1, ..];
26
27   src <=> dst;
28
29   // located on locale 0
30   max_diffs[here.id] = local_max_difference;
31
32   allLocalesBarrier.barrier();
33
34   if (!isLast) {
35     forall col in Row {
36       d[LocalLowerRow, col] = myCommunicator.lastRow[col];
37     }
38   }
39   if (!isFirst) {
40     forall col in Row {
41       d[LocalUpperRow, col] = myCommunicator.firstRow[col];
42     }
43   }
44
45   local_max_difference = max reduce max_diffs;
46
47   local_iteration += 1;
48 } while (local_max_difference > E && local_iteration < I);
```

Figure 13. Computational kernel for the local-view multi-locale version.

45). Note that we synchronize this communication with a barrier (line 32).

This implementation demonstrates the ability of local view programming in Chapel and it shows a big advantage of Chapel: the possibility to choose and combine different levels of abstraction to gain performance where needed but use productive and safe abstractions for most parts.

We are pleased with how well global view programming and local view programming combine in a single Chapel program. The initialization can be done using global view programming and does not need to change while the compute-intensive part of the code can use local view programming.

Table 3. Execution times in seconds, speedup and efficiency of the local view, multi-locale version compared to single-locale version with 16 threads for an input size of 8192×8192.

Nodes	1	2	4	8	16
par-local-multi	47.15	24.74	14.55	7.62	4.33
Speedup	0.53	1.01	1.72	3.28	5.78
Efficiency	0.53	0.51	0.43	0.41	0.36

Furthermore, it is possible to use much of Chapel's supporting functionality even for the local view part of the program. For example, the distribution can be done using the `localSubdomain()` and array slicing functionality which is more convenient and less error-prone than manual distribution.

In addition, the communication in Chapel via global arrays leaves less room for mistakes than explicit communication via messaging. However, very subtle code changes can lead to big performance changes: A programmer in Chapel might fall down a big performance cliff without having a clue what caused this cliff to exist.

An example for this is the difference between remote writing or reading which might depend on syntactically nearly invisible changes as using `LocaleSpace` or `{0..LocalesSpace.size}` as domain for the global array.

Additionally, we noticed that if remote communication is involved, array slicing beats a semantically equivalent `forall`-loop performance-wise. Intuitively this seems logical because array slicing tells the compiler in a more declarative way what should be done and can be easier optimized. However, if no remote communication is involved the `forall`-loop seems to be faster. Generally such behavior is bad, but especially in a high-performance language that offers a workflow to incrementally introduce more concrete concepts to gain performance, this kind of changing performance behavior leads to frustration and unnecessary work.

7.1 Expected Results

We expect that the version in Fig. 13 has high performance and scales well. We control the computation such that each locale works on its own local data and the stencil operation and reduction can be parallelized well locally. We also perform the reduction in two phases, first locally and then globally on one locale with a limited amount of values. Furthermore, the iterations are synchronized by only one barrier and we make use of remote writes that we have found to be faster. Given the large inputs that we have, from the user's perspective, there should be no reason why this version would not perform and scale well.

7.2 Actual Results

Table 3 shows that the performance is about a factor 2 for one node to a factor 3 for 16 nodes lower than we expect.

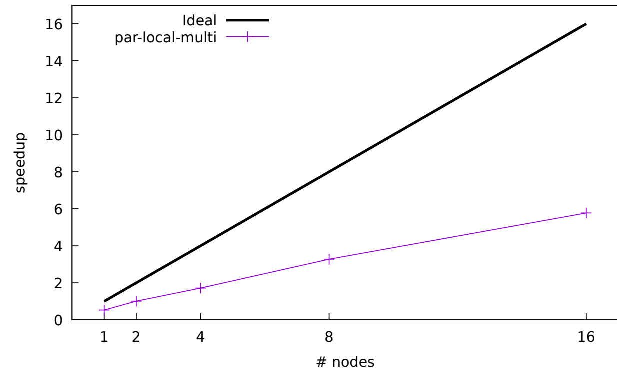

Figure 14. Speedup of the local view multi locale versions compared to the single-locale version with 16 threads on an input of 8192×8192.

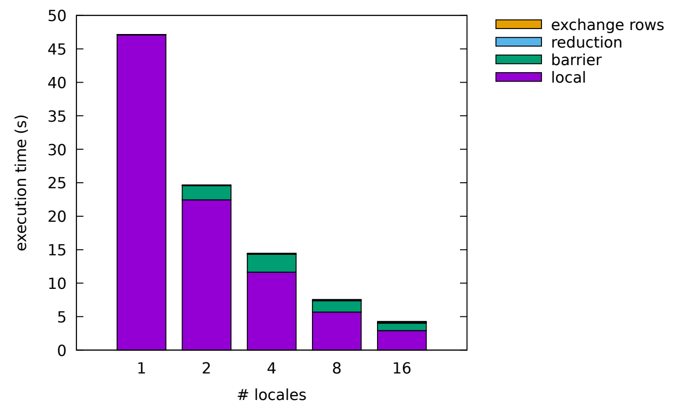

Figure 15. Breakdown of the execution times of the four stages of `par-local-multi` on an input of 8192×8192 with a varying number of nodes.

The scalability is not ideal, but this is likely caused by the fact that the total run-time is less than 5 seconds where the overhead of synchronization becomes more pronounced. We estimate that the scalability is better when we increase the input size.

Figure 14 shows graphically that the speedup lags behind our expectation, although this version achieves better results than `par-global-multi`. Figure 15 shows that the slowdown is hardly due to communication, but mostly caused by the computations that are in the local block with a small share of synchronization.

The implementation that we describe in this section has the largest potential for speed-up, so for this version we tried as much as we could think of to let the performance increase. The performance numbers reported here are for the implementation that performs the reduction inside of the loop as this gave better overall performance. We also experimented with having a separate reduction and we turned

on all performance-related compiler options. We verified the output of the GCC vectorization that reported that none of the important loops were vectorized. From a user-perspective, this is the maximum that can be expected in our opinion.

7.3 Feedback from the Chapel team

The information we received from the Chapel team also applies here: The reduced performance measured for this version can be improved by replacing the local blocks with the undocumented `.localAccess()` array accesses. In that case, the execution time improves by 1.46 times for 1 locale and by 1.36 times for 16 locales. The additional problem of the lack of NUMA awareness in GASNet is likely a large contributor.

8 Discussion and Conclusion

Chapel was designed with several principles and concepts in mind [6, 7]. We share our view on the relevant principles and concepts given our experiences. One of the principles is that it should be comfortable to express parallelism. Chapel certainly satisfies this with stencil computations. Stencil operations can be comfortably expressed and moving to multi-locale execution needs minimal adjustments. This is also conform to the principle that explicit orchestration should not be needed.

Particularly impressive is the way higher-level abstractions can be carefully applied when programming on a lower level as we did in our local-view version. The so-called "multi-resolution" design principles where all higher-level constructs are implemented in terms of lower-level constructs also available to programmers, allowed us to separate local code conveniently from multi-locale code.

Furthermore, Chapel focuses on separation of concerns between application developers, parallelism experts, and the compiler. Users are responsible for identifying parallelism and locality to achieve high performance but do not need a deep understanding of the optimizations the compiler applies.

In our experience, this principle has not been completely satisfied. The performance of the compiler-generated C code and the binary that the C compiler generates can vary significantly based on a small change in the code. We have seen sequential performance of 151 s and 390 s on the same input.

A final important principle is that the Chapel project sets ambitious goals to be a high-performance language for distributed execution. We applaud this, but unfortunately it appears that there is currently no satisfying way to achieve high multi-locale performance on our system using the GASNet communication layer, the recommended default for InfiniBand networks. However, the new OpenFabrics communication layer seems promising for our system once it becomes NUMA aware.

Acknowledgments

We want to thank the anonymous reviewers for their valuable feedback and Brad Chamberlain, Ben Harshbarger, and Elliot Ronaghan for their valuable performance insights. This research has received partial funding from the project "A methodology and ecosystem for many-core programming" funded by the Netherlands eScience Center, filenr. 27016G06.

References

[1] 2019. Software repository for the heat-dissipation problem in Chapel. https://github.com/JungleComputing/heat-dissipation-chapel.

[2] Henri E. Bal, Dick Epema, Cees de Laat, Rob V. van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 5 (May 2016), 54–63.

[3] Richard F Barrett, Philip C Roth, and Stephen W Poole. 2007. Finite Dfference Stencils Implemented Using Chapel. *Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122* (2007).

[4] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. 2015. Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/2751205.2751226

[5] Helmar Burkhart, Madan Sathe, Matthias Christen, Olaf Schenk, and Max Rietmann. 2012. Run, stencil, run! HPC productivity studies in the classroom. In *Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS 2012)*.

[6] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

[7] B. L. Chamberlain. 2015. Chapel. In *Programming Models for Parallel Computing*, Pavan Balaji (Ed.). MIT Press, Chapter 6, 129–159.

[8] Chapel Team Cray Inc. 2019. Release notes: Benchmarks and Performance Results. https://chapel-lang.org/releaseNotes.html.

[9] Per Fuchs, Pieter Hijma, and Clemens Grelck. 2019. Heat-dissipation in Chapel. https://doi.org/10.5281/zenodo.2713721

[10] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. 2015. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI '15)*. IEEE Computer Society, Washington, DC, USA, 34–39.

[11] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Comp. Soc. Tech. Comm. on Comp. Arch. (TCCA) Newsletter* (Dec. 1995), 19–25.

[12] Sparsh Mittal. 2014. A Study of Successive Over-relaxation (SOR) Method Parallelization Over Modern HPC Languages. *Int. Jour. of High Perf. Computing and Networking* 7, 4 (2014), 292–298.

[13] R. F. Van der Wijngaart and T. G. Mattson. 2014. The Parallel Research Kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2014.7040972

[14] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52 (April 2009), 65–76. Issue 4.