Have Your Cake and Eat It? Productive Parallel Programming via Chapel's High-level Constructs

Zixian Cai

A report submitted for the course
SCNC2101 Advanced Studies 1
The Australian National University

May 2018

© Zixian Cai 2018

Except where otherwise indicated, this report is my own original work.

Zixian Cai
25 May 2018

To my parents.

Acknowledgments

First, I would like to thank my supervisor, Josh Milthorpe. He helped me turn my vague idea into this project, and guided me with his experiences in high-performance computing and parallel systems.

I would also like to thank Steve Blackburn for helping me understand the architecture and performance of computer systems. His insights into evaluation methodology are invaluable to this work.

I want to extend my thanks to my great friend, Brenda Wang, for moral support and intellectual conversations. Thank you for spending time reading and giving me feedback on my writing, for which I am indebted.

The Programming Languages and Systems Lab has provided me with research machines to perform performance analysis. Some of the experiments were undertaken with the assistance of resources and services from the National Computational Infrastructure (NCI), which is supported by the Australian Government.

Abstract

Explicit parallel programming is required to utilize the growing parallelism in computer hardware. However, current mainstream parallel notations, such as OpenMP and MPI, lack in programmability. Chapel tries to tackle this problem by providing high-level constructs. However, the performance implication of such constructs is not clear, and needs to be evaluated.

The key contributions of this work are: 1. An evaluation of data parallelism and global-view programming in Chapel through the reduce and transpose benchmarks. 2. Identification of bugs in Chapel runtime code with proposed fixes. 3. A benchmarking framework that aids in conducting systematic and rigorous performance evaluation.

Through examples, I show that data parallelism and global-view programming lead to clean and succinct code in Chapel. In the reduce benchmark, I found that data parallelism makes Chapel outperform the baseline. However, in the transpose benchmark, I found that global-view programming causes performance degradation in Chapel due to frequent implicit communication. I argue that this is not an inherent problem with Chapel, and can be solved by compiler optimizations.

The results suggest that it is possible to use high-level abstraction in parallel languages to improve the productivity of programmers, while still delivering competitive performance. Furthermore, the benchmarking framework I developed can aid the wider research community in performance evaluations.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
1.1 Problem Statement 1
1.2 Contributions 2
1.3 Outline 2

2 Parallel Programming **3**
2.1 Background 3
2.2 Performance Models 3
2.2.1 Amdahl's Law 4
2.2.2 Weak Scalability 5
2.2.3 Work-span Model 5
2.3 Parallel Patterns 6
2.4 Sources of Parallel Performance Overheads 8
2.4.1 Coarse-grained Locking 9
2.4.2 Locality Issues 9
2.4.3 Load Imbalance 10
2.5 OpenMP 11
2.6 MPI 11
2.7 Summary 12

3 Chapel **13**
3.1 Global-view Programming 13
3.2 Implicit Data Parallelism 14
3.3 Memory Model 15
3.4 Cache 15
3.5 Architectural Topology and Task Affinity 16
3.6 Issues in Chapel Runtime 16
3.6.1 Miscomputation of the Number of Physical Cores 16
3.6.2 Poor Default Process Mapping 17
3.7 Summary 18

4 Evaluation Methodology **19**
4.1 Software platform . . . 19
4.2 Hardware platform . . . 19
4.3 Warmup . . . 20
4.4 The menthol Framework . . . 21
4.5 The Parallel Research Kernels . . . 21
4.6 Summary . . . 22

5 Case Studies **23**
5.1 Setup . . . 23
5.2 Reduce . . . 23
5.2.1 Result . . . 24
5.2.2 Advantage of Data Parallelism . . . 25
5.2.3 Chapel Performance Anomaly . . . 26
5.2.4 Summary . . . 29
5.3 Transpose . . . 30
5.3.1 Result . . . 30
5.3.2 Limits to Multi-node Scaling . . . 30
5.3.3 Improvement . . . 32
5.3.4 Summary . . . 33
5.4 Summary . . . 33

6 Conclusion **35**
6.1 Future Work . . . 35
6.1.1 Non-temporal Hints for Synchronization Variables . . . 36
6.1.2 Integrating Convergence-based Warmup to More Benchmarks . 36
6.1.3 More Comprehensive Benchmarking . . . 36
6.1.4 Comparison with Other Models . . . 36

Bibliography **39**

List of Figures

2.1 Speedup predicted by Amdahl's Law 4
2.2 Work and span: an example 6
2.3 Map: an example. 7
2.4 Stencil: an example 8
2.5 Two different reduction methods with different spans 8
2.6 Stride of array operations 10

3.1 Performance impact of using the miscalculated number of cores 17

4.1 Variance-based warmup 20

5.1 Two steps of the reduce benchmark 24
5.2 Performance of reduce kernel with number of cores: Chapel vs OpenMP on Raijin 24
5.3 Different implementations of vector addition 25
5.4 Performance of reduce kernel after serializing parts of code on Raijin . 26
5.5 Performance of reduce kernel with 8~16 cores: Chapel vs OpenMP on Raijin 27
5.6 Performance of reduce kernel with number of cores: Chapel vs OpenMP on weasel 28
5.7 Performance of reduce kernel using 12000000 as the vector length . . . 29
5.8 The performance of matrix transpose for all three implementations . . . 31
5.9 The performance of transpose written in Chapel using one vs two locales on a single node 32

Chapter 1

Introduction

Computer systems are increasingly parallel, and Chapel is one of the emerging languages that try to aid in productive parallel programming. This report focuses on high-level constructs of the Chapel language, why they are helpful, and the performance implications they have.

1.1 Problem Statement

Advances in the design and fabrication of semiconductors make computer hardware faster, which benefit different disciplines of science and engineering at large. As a result of this, researchers are able to tackle some computationally expensive problems on their laptops that are not even feasible before.

However, some of the tasks, with notable examples such as chemical and physical simulations, are extremely resource demanding, and their need for computational power, memory and storage simply cannot be met by a single CPU core or machine. To handle these tasks, computer clusters with high level of parallelism are designed and built.

This parallelism cannot be fully utilized by serial code, since there is only so much a compiler can do to automatically parallelize serial code. First, parallelizing code without compromising correctness often involves sophisticated dependency and control flow analysis, which are computationally intractable for large programs. Second, there are certain kinds of information that are only available to programmers, such as the behaviour of native shared libraries. This information may not be expressible in a given programming language. Without such information, compilers are not allowed to make certain optimizations per the semantics of the language.

Therefore, we need explicit parallel programming. There are two types of parallel models, general-purpose ones and domain-specific ones. Domain-specific models, such as Apache Spark[1], are good at their particular areas, but not for general parallel programming. Different general-purpose parallel models are proposed, and some of them, such as OpenMP and MPI, are well adopted. Unfortunately, most of these existing models require expression of algorithms in a task-by-task basis, which leads to fragmentation. That is, code concerning data distribution, synchronization and

[1] https://spark.apache.org

communication is intermingled with the actual algorithms. These fragmented models impose mental burden on the programmers, and thus make them less productive.

In response to these challenges, DARPA launched High Productivity Computing Systems (HPCS) program in 2002 to seek alternatives that would make parallel programming more productive. The Chapel language was developed as part of Cray Inc.'s involvement in the program. It claims to be an good alternative to both traditional multithreaded programming, and distributed memory programming by message-passing. It strives to provide high level abstractions as well as low level primitives. Together these constructs are meant to support expressing parallel computation in general, and optimizing for common cases.

However, the performance of the high level constructs provided by Chapel is not clear. On the one hand, high level abstractions carry the semantics of the operations. This semantics makes it easier for the compiler to reason about the program and perform optimizations. On the other hand, programmers tend not to have fine-grained control of the runtime system with such abstractions. This may make it hard to exploit special characteristics of a particular problem or hardware platform. Therefore, experiments need to be conducted to evaluate the performance of high-level constructs provided by Chapel.

1.2 Contributions

I first selected and implemented benchmark kernels from the widely used Parallel Research Kernels in Chapel. Then, I investigated the performance characteristics of the chosen kernels. I found that global-view programming in Chapel implicitly generates fine-grained communications, and impedes the performance in the transpose kernel. I also discovered that data parallelism gives Chapel performance advantages over OpenMP in the reduce kernel. During this study, I also identified several issues that lead to poor performance of Chapel programs under certain settings.

To support the benchmarking, I designed and developed the benchmarking tool `menthol`. Rigorous measurement principles, including convergence based warmup, are used within the framework. It allows comparison of different implementations of benchmark kernels. It also provides tools that aggregate, normalize and plot results to reveal the scalability trends of different configurations.

1.3 Outline

Chapter 2 summarizes some important aspects of parallel programming in general, with focus on performance theories and pitfalls. Chapter 3 highlights how features of Chapel are used to support parallel programming, and how their implementations relate to the performance.

Chapter 4 documents my experiment methodology. Chapter 5 then presents the result of some case studies of benchmark kernels and related discussions.

Finally, Chapter 6 concludes the report and suggests potential future works.

Chapter 2

Parallel Programming

In this chapter, I will give a high level overview of parallel programming. Section 2.2 summarizes relevant performance theories. Section 2.3 highlights parallel patterns, which are fundamental to the benchmark suites in Section 4.5. Then, Section 2.4 discusses common performance overheads of parallel programs. Finally, Section 2.5 and Section 2.6 introduce widely adopted parallel programming models, OpenMP and MPI. They are used later in Chapter 5 as my benchmarking baselines.

2.1 Background

Serial programs have been able to enjoy the speedup brought by hardware development without much active change, with the possible exception of being recompiled to use instruction set extensions. However, this will no longer be the case. Due to physical limitations, we are getting diminishing improvement brought by increasing clock rates or instruction-level parallelism (ILP) [Olukotun and Hammond, 2005].

To solve this problem, modern computers tend to exhibit different levels of parallelism, such as SIMD instructions and multicore processors. To take this to a larger scale, supercomputers nowadays even consist of thousands of nodes, where each node has multiple cores, and even multiple sockets.

In order to fully utilize these ever growing hardware capabilities, the transition to parallel programming is inevitable.

2.2 Performance Models

Performance models give us an expectation of the performance of parallel programs. A good model will focus on system and program parameters that are key to the performance, abstracting away less important details.

In the following section, I will present three important performance models for parallel systems. With respect to notations, T_P denotes the time it takes to solve an computation problem using P workers, and S_P denotes the speedup we have compared with only using one worker.

2.2.1 Amdahl's Law

Amdahl argued that the sequential execution time T_1 of a program consists of two parts [Amdahl, 1967]:

1. T_{ser}: time spent on serial work
2. T_{par}: time spent on parallelizable work

Then, given P workers, the parallel execution time is

$$T_P \geq T_{\text{ser}} + \frac{T_{\text{par}}}{P} \tag{2.1}$$

The bound on T_P assumes no superlinear speedup, and the equality can only be obtained when the parallelization is perfect.

Let f be the serial proportion of the total work T_{ser}/T_1, then the speedup (see Fig. 2.1) of having P workers is

$$S_P = \frac{T_1}{T_P} \leq \frac{1}{f + (1-f)/P} \tag{2.2}$$

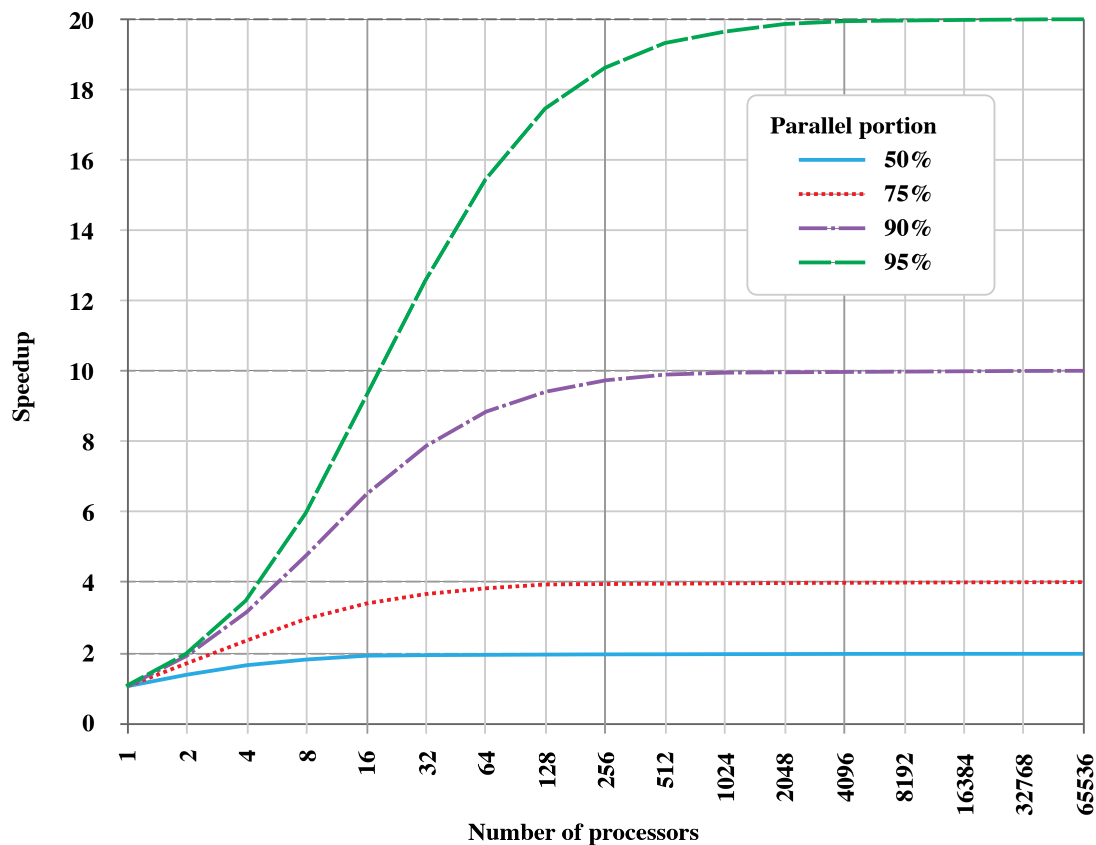


Figure 2.1: Speedup predicted by Amdahl's Law from Daniels220 at English Wikipedia [CC BY-SA 3.0], via Wikimedia Commons

As a consequence, the speedup parallelization is limited by the fraction of serial work, even when using an infinite amount of resources

$$S_\infty = \frac{1}{f} \tag{2.3}$$

Furthermore, the efficiency of the system is quite poor if we just increase the number of workers

$$\eta_\infty = \lim_{P\to\infty} \eta_P = \lim_{P\to\infty} \frac{T_{\text{ser}} + T_{\text{par}}}{PT_{\text{ser}} + T_{\text{par}}} = 0 \tag{2.4}$$

To summarize, Amdahl's law shows that the scalability of parallel programs is bounded from above by the proportion of parallelizable work. In addition, adding more workers gives diminishing returns, and decreases the efficiency of the system.

2.2.2 Weak Scalability

Amdahl's Law considers that the problem size is fixed, but the computational resources grow. This is called strong scalability.

Gustafson observed that the problem sizes tend to grow as there are more resources available on HPC clusters [Gustafson, 1988]. If the serial part of the problem is of constant size, it diminishes as a proportion of the whole (f in Eq. (2.3)). This means that the maximum speedup of the program will be higher as predicted in Eq. (2.3).

Known as weak scalability, Gustafson's Law makes the scalability story more compelling. However, it is in fact just a different view on the same problem, compared with Amdahl's law. Gustafson's Law focuses on the possibility of solving bigger problems in the same time, while Amdahl's Law concentrates on how long it takes to solve the same problem given more resources.

2.2.3 Work-span Model

Amdahl's Law is too optimistic, as it assumes perfect parallelism. In general, parallelism is imperfect, because the parallelizable parts are often of difference lengths, and other workers need to wait until the slowest one finishes (see Fig. 2.2).

Instead, the work-span model is more widely used as a guide, because it provides a tighter upper bound and a lower bound. In the work-span model, a directed graph is used to describe dependencies between tasks, i.e. a task can only begin after all its predecessors finish. T_1 is called the **work** of an algorithm, which is the total time spent to finish all tasks in a serial fashion. T_∞ is called the **span** of an algorithm. It is equivalent to the length of the longest chain of tasks that need to be executed one after another. This chain is also known as critical path, and having more workers does not accelerate its execution.

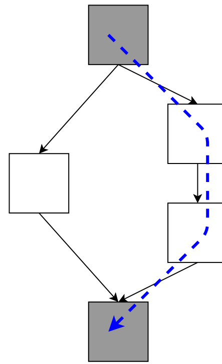


Figure 2.2: Work and span: an example. Black solid arrow denotes the dependency between tasks. Blue arrow denotes the span.

Figure 2.2 shows an example, where each box takes a unit time to finish. The grey boxes represent serial work and the blue dash arrow denotes the span of this problem. The work is 5 and the span is 4. Note that in this example, the upper bound for the speedup obtained from Amdahl's Law is $\frac{5}{2}$, while the upper bound derived using the work-span model is $\frac{5}{4}$. The difference here demonstrates that Amdahl's Law does not consider the fact that the work is not perfectly parallelizable.

The upper bound for the speedup using the work-span model is

$$T_P \geq T_\infty \tag{2.5}$$

$$S_p = \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty} \tag{2.6}$$

Brent's Theorem [Brent, 1974] gives a upper bound for T_P in terms of T_1 and T_∞.

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty \tag{2.7}$$

In practice, $T_\infty \ll T_1$, so the following approximation can be used

$$T_P \approx \frac{T_1}{P} + T_\infty \tag{2.8}$$

And therefore, a lower bound for speedup can be approximated by

$$S_P = \frac{T_1}{T_P} \approx P \tag{2.9}$$

2.3 Parallel Patterns

Some algorithmic structures are commonly seen in efficient parallel programs, which are known as parallel patterns [McCool et al., 2012]. Understanding and applying these constructs appropriately lead to efficient implementation of parallel problems. Section 4.5 describes the Parallel Research Kernels (PRK), which cover these patterns

in a benchmark suite, and is used to evaluate the performance of parallel systems in this work. This section will focus on the semantics of these patterns, i.e. what they achieve.

Map

The **map** pattern transforms the entire collection in an element-wise and uniform fashion (see Fig. 2.3).

The transformation is usually done by a function, which is applied on elements of a collection. In order to be parallelizable, the elemental function needs to behave deterministically despite the execution order. In particular, the functions needs to be side-effect free.

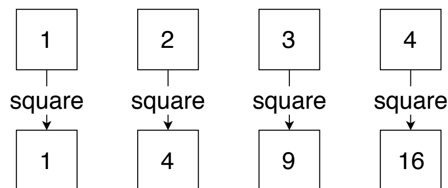


Figure 2.3: Map: an example.

This pattern can replace loops in serial programs, where each iteration of the loop only depends on the iteration index and the element being processed. Examples of usage include transforming each pixels of the image from RGB to greyscale, mapping words of an articles to the count of characters, etc.

Stencil

The **stencil** pattern generalizes the map pattern, with the difference that the transformation of each element not only depends on itself but also its neighbours (see Fig. 2.4). The performance of stencil is more subtle than that of map because the dependency on neighbouring cells has cache effects. And the effects can be significant for higher dimension matrices.

Stencil computations are featured in many scientific simulations, such as cellular automata and the Jacobi method. It is also widely used during imaging processing, like sharpening, where each pixel of the output image is determined by its neighbours pixels.

Reduce

The **reduce** pattern combines elements in a collection into a single value. In general, the combination function does not need to be associative, and the order the combination is carried out matters.

However, if the combination function is associative, different orderings will give the same result. But they might have different spans, and therefore different bounds

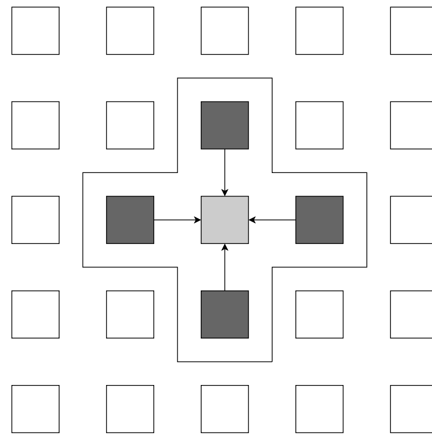
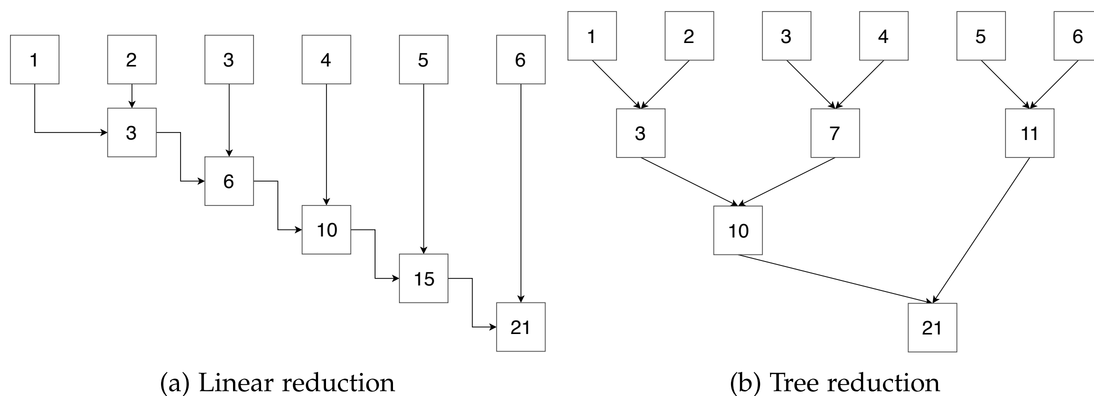


Figure 2.4: Stencil: an example. Dark grey boxes represent the neighbours of the light grey box.

on speedup. Figure 2.5 is an example of how the summation of a list can have different spans.



(a) Linear reduction (b) Tree reduction

Figure 2.5: Two different reduction methods with different spans

As shown in the diagram above, the summation can be evaluated in two different orders. The first method adds numbers from left to right in a sequential way, and the span grows linearly with the problem size. The second one adds numbers in pairs in a tree-like structure. In this way, the calculation of each pair is independent from each other, and therefore is potentially parallelizable. And the span is proportional to the logarithm of the problem size.

2.4 Sources of Parallel Performance Overheads

Real systems are not ideal, and different parts of parallel systems interact with each other in the forms of contention and communication. Mishandling of these situations can lead to both correctness and performance issues. In this section, I will focus on the common sources of parallel performance overheads.

2.4.1 Coarse-grained Locking

Locks are used to enforce mutual exclusion on a critical region. They make parts of the execution serial, and thus introduce Amdahl bottleneck. A highly contended lock is often reveal by profiling results showing a prolonged period of time spent inside locking mechanisms. It can degrade the performance even more than plain serial code due to cache effects, etc.

One common cause for this is coarse-grained locking, as it may cause contention more than necessary. For example, if a big data structure is protected using one mutex, updates to two or more logically independent fields will unnecessarily contend.

One way of solving the problem is replacing a highly contended lock with smaller ones. Furthermore, if mutual exclusion is applied on a single memory location, hardware atomic operations might be used to perform reads and write. Last but not least, we should seek lock-free algorithms where possible.

2.4.2 Locality Issues

Locality consists of two different parts: temporal locality and spatial locality. Both of them refer to how predictable the memory accessing patterns are. Temporal locality means a program is likely to reference recently used memory locations again. Spatial locality means a program is likely to reference nearby locations of a recently referenced memory location.

Nowadays, CPUs exhibit multiple levels of cache hierarchy to compensate the mismatch between the core frequency and the performance of RAM chips. And hardware prefetchers are also used to predict and load data from RAM when those data are likely needed.

In parallel systems, locality plays an important role. Good locality enables utilization of hardware caching and prefetching, and leads to efficient access to distributed data. On contrast, bad locality renders the prefetched data and cache useless, which could mean CPU stalling for hundreds of cycles waiting for memory operations. On modern hardwares, this stalling is much more expensive than computation cost. Therefore, distributing data in a way that maximize cache hits is usually desirable, even at the cost of some extra computation.

In the context of arrays, the stride of array operations affect the spatial locality of a program. When the array does not fit in the cache, operations of non-unit stride will cross cache line boundaries and generate more memory traffic. For example, as shown in Fig. 2.6, if a two-dimensional array is of row-major order, reading from $(1,1)$ and then $(1,2)$ is of unit stride, which exhibits good spatial locality. In contrast, reading from $(1,1)$ and then $(2,1)$ is of non-unit stride, as there is a row of entries in between. When the row does not fit in a cache line, reading from $(2,1)$ will result in a cache miss, since the cache line where it resides needs to be brought into the cache hierarchy.

Tiling and fusion are commonly used to increase the performance by reducing cache misses. These techniques break the problem into working sets that fit in cache,

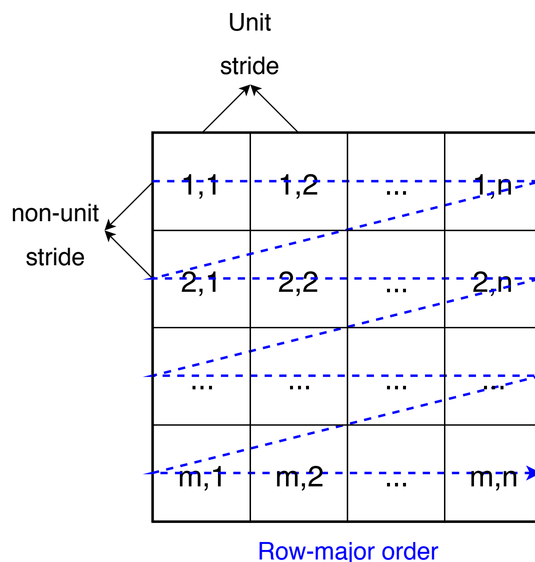


Figure 2.6: Stride of array operations

so that the time spent on numeric computation dominates. Also, the accessing patterns are tuned to avoid accessing too many pages at once to minimize Translation Lookaside Buffer (TLB) misses.

2.4.3 Load Imbalance

When the work is divided and distributed among processing units unequally, the span will be determined by the longest chain of tasks, which limits the speedup (see Section 2.2.3). This can be solved by dividing the work into even smaller chunks, since it is easier to schedule those tasks evenly among workers. This is known as overdecomposition, as we create more potential parallelism than the number of workers, and thus reduce the number of them idling. However, overdecomposition comes with overheads.

First, if the granularity is too small, the scheduling overheads are not negligible. For example, the dynamic loop scheduling in OpenMP[1] assigns work to threads in chunks of the same size. If the work is not substantial, the time spent on scheduling work will be significant.

This problem can be mitigated by not enforcing the division of tasks into uniform chunks. The guided scheduling in OpenMP adjusts the size of chunks dynamically during runtime, and thus makes the granularity fine enough to create potential parallelism, while keeping the overheads small. Work-stealing scheduling can also be used to improve the efficiency. Each processor is working on its local queue, and only "steals" work from other threads when necessary. Therefore, there will be no scheduling overheads if the work among processors are balanced, and the penalty

[1] Table 2.5 in OpenMP 4.5 Complete Specifications

only incurs infrequently.

Second, having too many workers might lead to competition on other hardware resources. For example, in real systems, the memory bandwidth is limited, and having too many workers will cause them to contend on the memory bus. In addition, having more parallel workers leads longer time spent on synchronization and communication before starting the serial section of the work. This cost can no longer by amortized if the number of workers is too big. So, it is crucial to find the sweet-spot between too much overdecompostion and worker underutilization.

2.5 OpenMP

OpenMP stands for open multi-processing, which is standardized by the OpenMP Architecture Review Board. It is a widely adopted programming interface for shared memory programs that bases on a set of compiler pragmas.

OpenMP is often used to parallelize loops, where threads are spawned to collectively execute the region[2]. Thinking in terms of threads does have advantages. It enables programmers to control the runtime behaviour of the program, and get the most out of a peculiar hardware platform. However, it imposes mental burden on programmers to understand the context and semantics of the directives they are using, and how they map to threads. This does not scale well when the problem becomes more complex.

This explicit threading model also causes composability problem. In particular, OpenMP does not even compose well with itself. For example, programmers might opt to use a third-party library which uses OpenMP internally. If the third-party library function is invoked inside an existing parallel region, the number of threads grow quadratically and may heavily oversubscribe the hardware. Therefore, programmers need to scrutinize every single library they are using before attempting nested parallelism. This approach does not scale well for real-world projects that use a wide range of libraries.

2.6 MPI

MPI stands for Message Passing Interface, which is standardized by the Message Passing Interface Forum[3]. Unlike OpenMP, which is limited to shared memory systems, MPI can be and is widely used for distributed parallel programming.

Under Flynn's Characterization [Flynn, 1972], MPI is classified as Single Program, Multiple Data (SPMD). Each MPI program is run as a set of processes that contain identical instructions. However, they independently operate on their local data structures. Data is exchanged among processes by explicitly sending and receiving messages via the MPI API.

[2]OpenMP also supports task parallelism, but it is relatively new and not widely used.
[3]http://www.mpi-forum.org

The MPI API supports different kinds of communication schemes, including point-to-point communication and collective communication. Some MPI implementations also features parallel IO, where small IO requests from different processes are delivered in bulk.

The main problem with MPI, and other SPMD programming models, is fragmentation [Chamberlain et al., 2007]. Since each process can only operate on its own local data, the overall data structure used by a problem is manually fragmented between processes. The actual algorithms are obfuscated by detailed management of communication and synchronization.

2.7 Summary

This chapter discusses patterns and concepts that are bread and butter of parallel programming. The problems present in OpenMP and MPI motivate the features of Chapel, which is presented in Chapter 3. The discussions on performance in this chapter are helpful later when analysing the performance of Chapel.

Chapter 3

Chapel

Since we are stuck with explicit parallel programming, it is crucial that we are able to be productive and write maintainable code while preserving performance. Productivity issues with some mainstream models, such as OpenMP and MPI, are discussed in Chapter 2.

Section 3.2 and Section 3.1 show how features of Chapel [Chamberlain et al., 2007; Cray Inc, 2018] attempt to solve these issues and support parallel programming at scale. Then, Section 3.3, Section 3.4 and Section 3.5 discuss concepts that have impact on the performance of Chapel programs. Finally, Section 3.6 describes two performance issues I found in the Chapel runtime code, and solutions are proposed.

3.1 Global-view Programming

As discussed in Section 2.6, prevalent SPMD models, such as MPI, require explicit fragmenting of data structures, and expression of algorithms are couple with the layout of data. That is, programmers have to scattering code over the the body of algorithms if they want to access data in a particular pattern. This tends to obscure the underlying algorithms, and imposes mental burden on programmers.

Chapel is different from the above mainstream notations that it supports global-view programming via Partitioned Global Address Space (PGAS). Unlike explicit message passing, Chapel programs have data referencing semantics, as distributed data structures can be manipulated in the same way as local ones. As shown in the code snippet below, one locale can use the variable stored on another locale without explicitly writing a `GET` operation.

```
1  var x: int = 2;
2  on Locales[1 % numLocales] {
3      var y: int = 3;
4      // Locale 1 referencing variable x that is declared and stored on locale 0
5      writeln("From␣locale␣", here.id, ",␣x␣is:␣", x, "␣and␣y␣is:␣", y);
6  }
```

The data layout is expressed in an orthogonal part of the code, where programmers can choose partition schemes that suit the application. For example, as the code snippet below demonstrates, an array can distributed among different locales (see

line 7). The distribution of the array can be changed by just modifying line 5.

```
use BlockDist;

const Space = {1..4};
// One dimensional Block-distributed domain
const BlockSpace = Space dmapped Block(boundingBox=Space);
// Distributed array over the above domain
var BA: [BlockSpace] int;
for L in Locales {
    on L {
        // Each locale will store roughly equal portion of the space
        writeln("From␣locale␣", here.id, ":␣", BA.localSubdomain());
    }
}
```

This separates the concerns of the algorithm and the data layout. In this way, programmers can write distributed parallel code in the same fashion despite the different in the underlying data storage.

3.2 Implicit Data Parallelism

Section 2.3 highlights some important parallel patterns that are often found in real-world applications. In parallel notations like OpenMP, these patterns are usually expressed in the form of parallel loops. However, these explicit loops tend to obfuscate the underlying algorithms by scattered control statements. In contrast, Chapel provides succinct syntax for common patterns such as map and reduce through implicit data parallelism.

The **map** pattern is supported by promotions of functions and operators. A function accepting one or more scalar arguments can be called with one or more iterables, including arrays and ranges. With promotion, the same operation is applied uniformly across the entire collection without explicit looping. The code snippet below shows how elements of an array can be scaled by a constant (see line 4).

There is also a keyword for expressing the **reduce** pattern. The combination function is specified as the first operand, and the collections as the second operand. As shown by the code snippet below, the sum of the array can be expressed as a reduction using addition as the combination function (see line 5).

```
const Index = {1..42};
const scale = 3.14;
var realArray: [Index] real;
var scaled = scale * realArray; // * is "promoted"
var sum = + reduce realArray;
```

When encountering these keywords, an arbitrary number of tasks will be used to execute them. The actual number of tasks generated depends on the length of the operands, and the hardware available, etc.

3.3 Memory Model

Being a parallel language, it is important that Chapel exploits the maximum level of parallelism exists on hardware. One important technique to improve parallelism is reordering memory operations, and it can be done at different levels. The compiler might reorder memory loads and stores as long as the overall effects are the same. On the hardware level, ILP and out-of-order execution can also reorder instructions and issue them to different datapaths.

Chapel based its consistency model on sequential consistency for data-race-free programs, which is also used by C11 [ISO, 2011], etc. The memory model defines what kinds of interleaving and reordering are valid when multiple tasks read and write with shared memory [Sorin et al., 2011]. It is crucial to both the correctness and performance of programs. On the one hand, if we disallow the specification of the order of memory operations, hardware and software optimizations can reorder memory reads and writes in such a way that breaks the semantic of the program. On the other hand, if we disallow the interleaving and reordering completely, the performance degradation will be significant, as we are not using the full hardware capacity, like multiple datapaths on CPU. Chapel also extends this model to multi-locale programs. Conceptually, using `GET` and `PUT` to access remote data is just like using load and store for memory on a single node.

3.4 Cache

Besides memory model, caching is critical to both performance and correctness. Section 2.4.2 has already discussed the performance impact of cache misses in the context of locality. This performance penalty is even higher on multi-locale programs, because remote data accessing is of orders of magnitudes slower than that on a single node due to the high internode communication latency. The correctness of programs also depends on cache, because reading stale values in cache would lead to incorrect computation. Chapel runtime uses memory fences to keep a coherent view of the memory among different tasks. Normally, the program reads and write through the cache. When encountering a `release` fence, pending `PUT`s are flushed to the destination locale. When encountering an `acquire` fence, any prefetch value is discarded.

The synchronization variable (quantified by `sync`) is an example of how cache affects the performance of language features. Synchronization variables have empty/full states associated with the values. When a variable is written, its state turns to full, while when it is read, its state changes to empty. A read of a synchronization variable is blocked until the variable is full, and a write of a synchronization variable can only proceed when the variable is empty. As the name suggests, these variables are often used by tasks to synchronize with each other.

Recall that data is transferred in blocks (cache lines) between memory and CPU. If multiple cores are coordinating using a `sync` variable, the entire cache line where the variable resides needs to be moved from one core's cache to another core's cache. This imposes high pressure on the memory bus, as the cache line bouncing back and forth

between cores, even sockets. When more than one locale is involved, the situation is more problematic because all remote operations on synchronization variables involve migrating tasks to another locale.

In the case of shared memory programs, non-temporal hint might be used to directly write the data to memory. On supported hardware, this feature allows bypassing the cache hierarchy, and avoids the corresponding cache line being fetched [Yang et al., 2011]. However, due to my time constraint, this technique is not implemented and evaluated in Chapel.

3.5 Architectural Topology and Task Affinity

Modern hardware, especially the ones used in HPC server farms, tends to exhibit hierarchical topology. And the topology is getting more and more complex, as new technologies emerge, such as non-uniform memory access (NUMA) memory. Awareness of such architecture and scheduling tasks accordingly are crucial to exploiting available hardware capacity efficiently.

One important aspect of task scheduling is task affinity. It restricts the execution of certain tasks to a subset of hardware contexts. Tasks usually have their own memory accessing and communication patterns, and ignoring these facts can sometimes lead to significant performance differences. For example, two independent tasks might need to be placed on two different sockets to maximize utilization of memory bandwidth. In contrast, two highly coupled tasks might be better off by placing them on the same socket so that they can share Last Level Cache (LLC). Furthermore, for communication intensive tasks, they might benefit from being placed on the processors that are closed to interface cards [Moreaud et al., 2010].

3.6 Issues in Chapel Runtime

During the study of the performance of Chapel programs, I encountered the following two bugs in the Chapel runtime. Both of them lead to poor performance in certain settings.

3.6.1 Miscomputation of the Number of Physical Cores

Chapel allows programmers to specify the maximum number of threads used by the runtime. Besides numerical values, one can use symbols `MAX_LOGICAL` and `MAX_PHYSICAL`, which stand for the number of logical and physical cores respectively. This allows programmers to prescribe the overall threading behaviour without hardcoding a value for a specific platform.

I found a bug[1] that the Chapel runtime code may miscompute the number of physical cores on certain platforms. The runtime assumes that we have access to physical cores in the same ratio as we do with logical cores. Therefore, the number

[1] https://github.com/chapel-lang/chapel/issues/9395

of physical cores can be derived from the number of logical cores. For example, if we have access to 6 out of 8 logical cores on a two-way hyper-threaded machine, the Chapel runtime would assume that we have access to three physical cores ($6/2 = 3$). This leads to underestimating the number of physical cores on a machine where not all of the corresponding logical cores are exposed the program.

This does cause performance degradation for certain applications. For example, the platform I used, `Raijin`, does not expose hyper-threads to programs by default. Therefore, the numbers of physical and logical cores are the same, and one would expect to see identical performance trend for both `MAX_LOGICAL` and `MAX_PHYSICAL`.

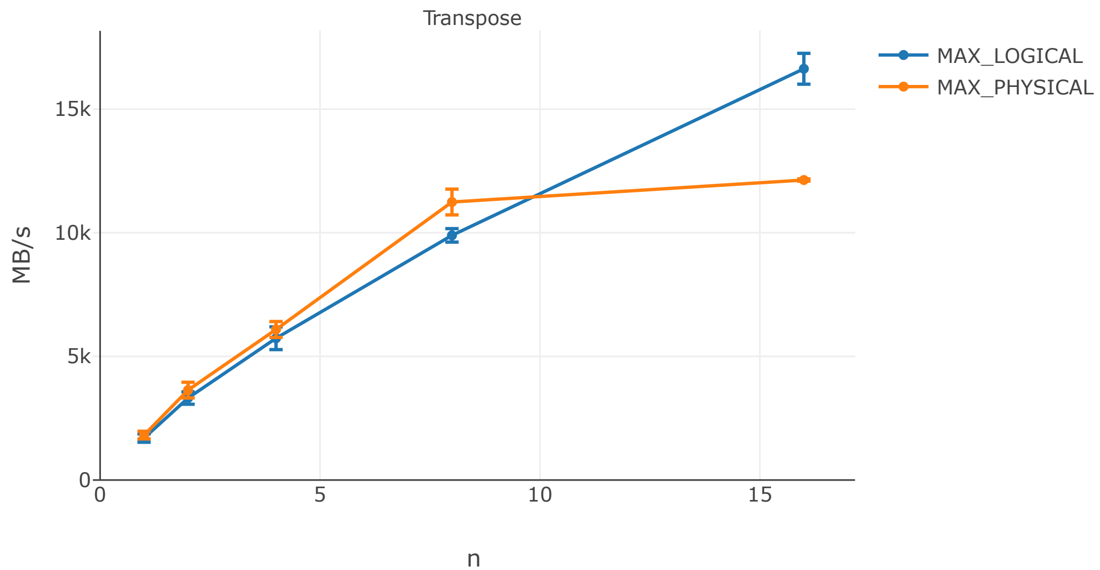


Figure 3.1: Performance impact of using the miscalculated number of cores

However, when running the transpose benchmark (see Section 5.3), `MAX_PHYSICAL` causes near 25% slowdown when the number of available physical cores is 16 (see Fig. 3.1).

One way of solving this problem is to use the `hwloc` library[2]. During my experiments on various hardware platforms, the library can find out the available cores and hardware threads quite reliably. I believe this is a better solution than the original naïve approach used by Chapel, which is reading `/proc/cpuinfo`.

3.6.2 Poor Default Process Mapping

Another bug is that multi-locale Chapel programs exhibit poor default process mapping on non-Cray clusters. For example, when running on a cluster with InfiniBand interconnect, a Chapel program uses GASNet, which in turn uses `mpirun`, to spawn processes on different nodes. By default, processes are placed sequentially on cores available. This leads to an undesired behaviour that two or more Chapel process

[2] https://www.open-mpi.org/projects/hwloc/

are placed and contending on the same socket, while there are other sockets or even nodes idling (also see Section 3.5).

This problem can be fixed by specifying how `mpirun` maps processes. As suggested by the Chapel documentation, one Chapel process should be spawned on each node, which can be implemented as `mpirun --map-by ppr:1:node:PE=<n>`[3] where `<n>` is the number of cores on the node.

3.7 Summary

In this chapter, I demonstrate how Chapel can be a solution to the productivity issues we are facing in parallel programming. In addition, I show how concepts, such as memory models, impact the performance of Chapel programs. Furthermore, two performance bugs I found in Chapel runtime are examined.

Chapter 4 shows my experimental methodology, which is fundamental to the performance evaluation I did.

[3] https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php

Chapter 4

Evaluation Methodology

Evaluation methodology is fundamental to this work, as my goal is to gain understanding of the performance implications of the high-level constructs of the Chapel language.

Section 4.1 and Section 4.2 document the software and hardware of my experimental environment. Next, Section 4.3 shows how variance-based warmup is used to handle non-determinism, and reduce the variance in the results. Then, the `menthol` framework, which is used throughout this work, is presented in Section 4.4. Finally, Section 4.5 discusses the choice of benchmark suite.

4.1 Software platform

The Intel® C++ Compiler of version 18.0.1 20171018 is used. The codes written in C and C++ are compiled with flags `-g -O3 -xHOST -qopt-report=5`. The MPI code is compiled using `mpicc` that wraps `icc` with Open MPI 3.0.1. The Chapel of version 1.17.1[1] is built from source with multi-locale support using GASNet with `ibv` conduit. Programs written in Chapel are compiled with flags `--fast`.

4.2 Hardware platform

The benchmarks are run on the `normal` queue on the `Raijin` cluster managed by NCI. Each node has dual 8-core Intel® Xeon® E5-2670@ 2.60 GHz with 32 GB, 64 GB or 128 GB memory available (depends on how much memory a job requests). The nodes are interconnected using Hybrid FDR/EDR Mellanox Infiniband (up to 100 Gbit s^{-1}) with full fat tree topology. Allocated nodes are chosen by the batch system (PBSPro 14.2.5.20180202014445).

Some shared memory benchmarks are also run on the machine `weasel` with quad 12-core Intel® Xeon® Gold 5118@ 2.30 GHz with 512 GB memory. This is to reproduce the experiments on a NUMA machine with different microarchitecture.

[1] https://github.com/chapel-lang/chapel/tree/1.17.1

4.3 Warmup

When taking measurements during the experiments, we are interested in the steady states, as they are predominant in the total execution time.

However, the application usually takes a while to reach such state for the following reasons. First, the working set of the program takes time to be brought into the cache hierarchy (see Section 3.4). Second, the working state of communication systems is set up on demand [Hoefler and Belli, 2015].

The naïve approach used in the original PRK code is to use a constant number, say one, of iterations to warmup. This rule of thumb might be useful in many cases, but it is not sufficient when the environment takes more than one iteration to warm up. Figure 4.1 is real data from one of the experiment runs. As shown in the graph, the variance of measurements in first 10 iterations is big, and it is apparent that one iteration did not warm up the program sufficiently.

I built a convergence based warmup strategy upon the work in [Blackburn et al., 2006, 2008]. First, several unmeasured iterations are performed, and the moving variance of metrics is recorded. Then, after the variance is below a predefined threshold, another N measured iterations are executed to produce the final result. This ensures that the communication and runtime is sufficiently warmed up, and the variance is low.

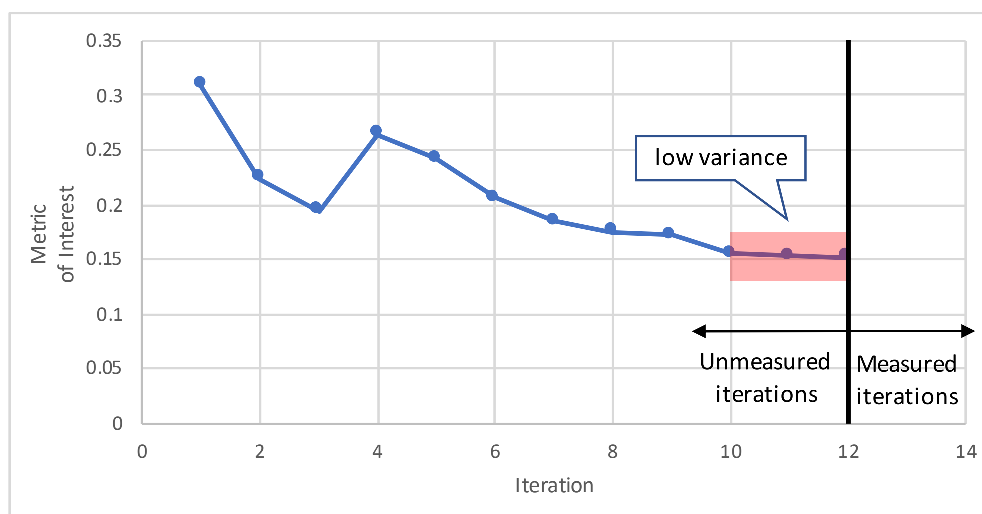


Figure 4.1: Variance-based warmup. Red shade indicates the window where the variance is below a predefined threshold.

However, applying this warmup strategy requires modification to the source code. Due to my time constraints, this was only integrated into the Chapel version of the reduce benchmark.

4.4 The menthol Framework

To conduct the benchmark rigorously, common techniques like warmup and performing multiple invocations of the program are used to reduce the variance in the measurement.

Different implementations of benchmark kernels are also interleaved during the experiment. For example, when comparing three implementations (A, B and C) for two invocations, order $ABCABC$ is used instead of $AABBCC$. This is to reduce the chance that the result of one implementation is highly skewed due to short running background tasks, such as the operating system happens to be rotating logs during invocations of A. And to summarize and analyze the result, graphing and statistic reporting are also needed.

These methods and utilities are not unique to this work, so distilling them into a framework will benefit other researchers in the wider community. Therefore, I designed and developed a framework called menthol, and it is used throughout this work. It is publicly available at https://github.com/caizixian/menthol and licensed under Apache License 2.0.

4.5 The Parallel Research Kernels

To evaluate the performance of a system, we need workloads that are typical of how the system is intended to be used. That is, the workloads we choose need to be diverse and representative enough to help us understand and improve the performance of real-world systems.

Section 2.3 discusses some patterns that usually lead to efficient parallel programs. The Parallel Research Kernels (PRK) [Wijngaart and Mattson, 2014] is a benchmark suite supporting evaluation of parallel systems. Wijngaart and Mattson argue that the typical workload will evolve, and production-level applications included in a benchmark suite will be less relevant. Therefore, full applications are not used in this suite. Instead, a collection of kernels is used to cover parallel patterns, and stress different features of parallel systems, including computation and synchronization. By covering a wide enough range of these patterns, the test suite can be used to support the design and prediction of future parallel systems.

The PRK is used in this work because these kernels are relatively simple to reason about, and be ported to a new platform, namely Chapel. These kernels also come with performance models specified by problem size and the number of process units. With these performance models, it is easier to find out whether the equivalent implementation in Chapel performs as what we expected. Furthermore, the PRK is designed so that the result of computation can be calculated using a set of formulae. This self-verification would allow us to pick out the error in the compilation, runtime, etc.

The reduce and transpose benchmarks from PRK are used in Chapter 5.

4.6 Summary

This chapter shows how various techniques are deployed to make the benchmarking more systematic and rigorous. I also justified my choice of workloads in Section 4.5.

In Chapter 5, I will show the results obtained by using these methodologies. Two kernels in PRK are used as examples to show the performance implication of high-level structures of Chapel.

Chapter 5

Case Studies

In Chapter 3, I demonstrated the programmability of Chapel over MPI and OpenMP through two concrete examples: global-view programming and implicit data parallelism. Nanz et al. [2013] also showed that Chapel has clear advantage in terms of succinctness compared with other models, such as TBB and Cilk. However, Chapel's performance is relatively poor in that study.

High-level constructs can carry the important semantics of operations, whiling abstracting away details of the underlying platform. In this chapter, I will use two case studies to show how these two ideas come into play with the performance of Chapel, why they lead to better or worse performance, and what we might do to improve the performance.

5.1 Setup

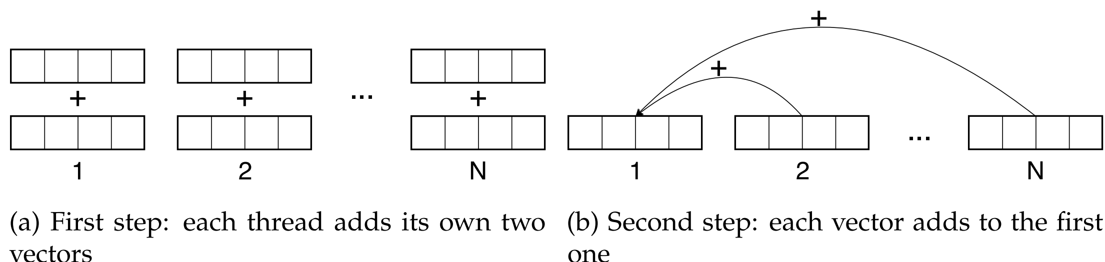
The experimental setup and methodology are discussed in Chapter 4. There are still two points worth mentioning. First, the filesystem setup does not affect the results in my case studies, as all the input data are synthesized instead of loaded from filesystem. Second, weak scaling is used for the case studies, since problem sizes tend to grow when there are more resources in real-world HPC settings (see Section 2.2.2). How problem sizes are calculated will be explained for each benchmark.

In the following sections, the standard deviation is displayed along with data points in the line graphs.

5.2 Reduce

The reduce benchmark from the PRK suite stresses synchronization mechanisms, as well as communication and memory bandwidth. In this benchmark, the element-wise sum of a collection of vectors is performed collectively by a group of workers (see Fig. 5.1). In the first step, as one can see in Fig. 5.1(a), each thread adds the two vectors it owns. Then, workers collectively add the rest of vectors to the first vector (see Fig. 5.1(b)).

Since the reduce benchmark is not included in the official Chapel release, I have to implement it myself. Due to my time constraint, I only implemented the single



(a) First step: each thread adds its own two vectors

(b) Second step: each vector adds to the first one

Figure 5.1: Two steps of the reduce benchmark

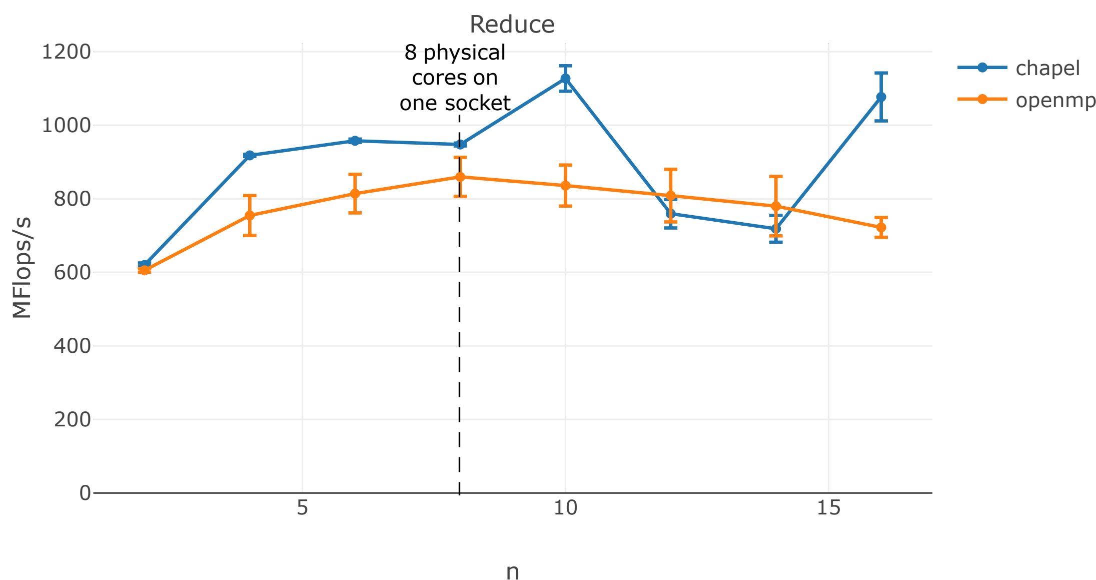


Figure 5.2: Performance of reduce kernel with number of cores: Chapel vs OpenMP on Raijin

locale version, so that the OpenMP implementation is used as our baseline here.

When running the benchmark, the number of threads is set to the total number of physical cores. The length of vectors for each thread is fixed. This means that the problem size, namely the total length of vectors, is proportional to the number of physical cores.

5.2.1 Result

As shown in Fig. 5.2, Chapel outperforms our baseline OpenMP except when the number of cores is 12 or 14. The slowdown around 12 cores for Chapel will be discussed in Section 5.2.3.

It is not immediately clear what is causing the performance advantage of Chapel. By examining the source code of reduce, I noticed that the vector addition is implemented differently in Chapel and OpenMP. The Chapel implementation uses array addition syntax (see Fig. 5.3(a)) which is a form of implicit data parallelism (see Section 3.2), while the OpenMP implementation uses a serial for loop to add entries of

```
for id in 1..nthread - 1 {
    vector[0..vectorLength-1] += vector[id*vectorLength..(id+1)*vectorLength-1];
}
```

(a) Chapel

```
#pragma omp parallel
{
    ...
    #pragma omp barrier
    #pragma omp master
    {
        for (id=1; id<nthread; id++) {
            for (i=0; i<vector_length; i++) {
                VEC0(0,i) += VEC0(id,i);
            }
        }
    }
    ...
}
```

(b) OpenMP

Figure 5.3: Different implementations of vector addition

the vectors (see Fig. 5.3(b)). This difference is presented in both the first and second step of the reduce (see Fig. 5.1). I suspect that this is the source of difference in performance, which will be discussed in Section 5.2.2.

5.2.2 Advantage of Data Parallelism

Recall that the span of a program is the length of the longest chain of tasks that needs to be executed serially. As shown in the work-span model (see Section 2.2.3), the longer the span is, the lower the maximum speedup for a program is. The problem size for reduce, which is the total length of vectors, is proportional to the number of threads. This means that the span of OpenMP reduce grows linearly with the number of threads, and consequently limits its speedup. Therefore, I hypothesize that the Chapel has automatic parallelization for array addition, and therefore benefits from the extra parallelism in second step.

To test out my hypothesis, I created two variants of the Chapel reduce, where the first step and the second step are rewritten respectively. Instead of using array addition syntax, the vector addition is explicitly implemented using a serial for loop, so that the loop will not be parallelized. The code snippet is equivalent to Fig. 5.3(a), but uses a serial for loop.

```
for id in 1..nthread - 1 {
    for i in 0..vectorLength-1 {
        vector[i] += vector[id*vectorLength+i];
    }
}
```

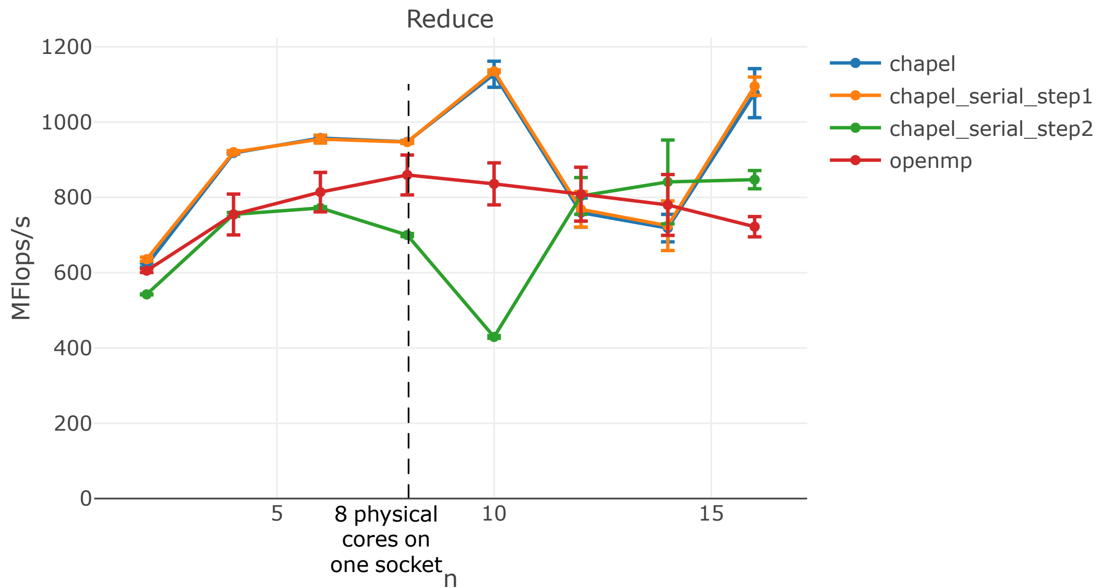


Figure 5.4: Performance of reduce kernel after serializing parts of code on Raijin

These two variants are denoted by `chapel_serial_step1` and `chapel_serial_step2`. Figure 5.4 shows the performance trend of the above two variants. As shown in the graph, serializing the vector addition in the first step does not make any signifiant difference. In contrast, rewriting the second step makes the performance of Chapel consistently worse than OpenMP when the number of cores is smaller than 12. This suggests that parallelizing vector additions in the second step is indeed the source of performance advantage of the original Chapel code.

One could argue that this is not an inherent problem for OpenMP, since Fig. 5.3(b) can be rewritten using `parallel for` directive. This brings back to the issue of OpenMP composibility. As discussed in Section 2.5, OpenMP does not compose well with itself. In order to prevent oversubscription, nested parallelism is turned off by default. Since `parallel` pragma has already been used to create threads for reduction, the `parallel for` applied on the second step would be nested under an existing parallel region. Therefore, the `parallel for` directive will be ignored, and the loop is still serial.

In contrast, Chapel supports high-level data parallelism constructs, and does not expose threading model to the programmer directly. As a result, the compiler and runtime can parallelize vector additions as long as there are available hardware resources.

5.2.3 Chapel Performance Anomaly

As mentioned in Section 5.2.1, there is a significant performance drop around 12 cores. Another set of experiments was performed, with the number of cores ranging from 8 to 16. This is to narrow down the exact number of cores that triggers this

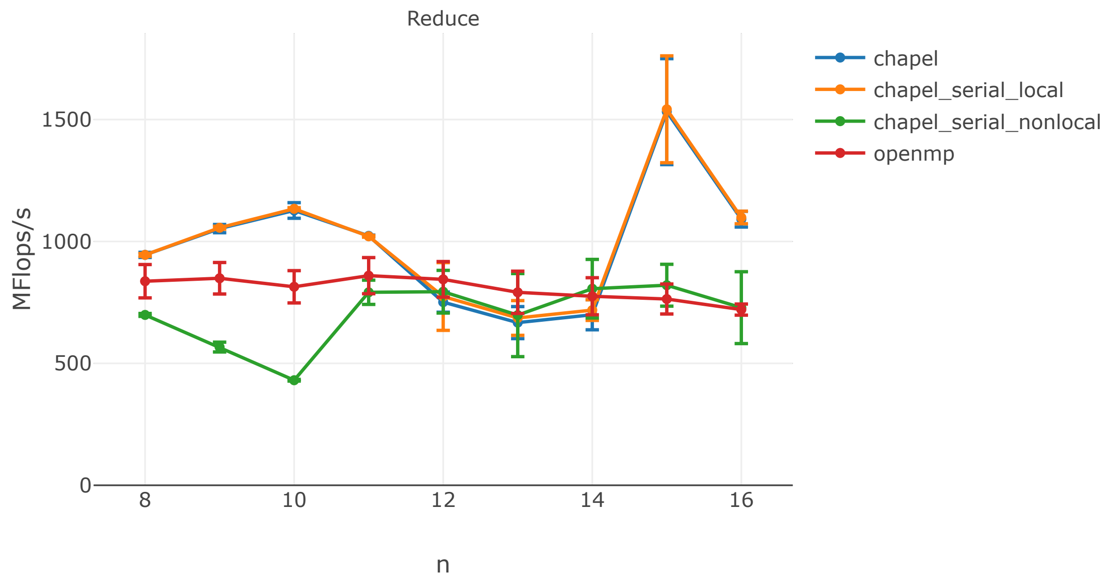


Figure 5.5: Performance of reduce kernel with 8~16 cores: Chapel vs OpenMP on Raijin

behaviour. As shown in Fig. 5.5, the performance of Chapel reduce slows down when the number of cores is bigger than 10. However, the graph does not reveal where the source of the problem is. I tried to use a different microarchitecture and a different problem size to find out the cause.

Different Microarchitecture

I attempted to reproduce the results in Fig. 5.2 using a different machine `weasel` (see Section 4.2). It has 48 physical cores and uses a different microarchitecture Skylake, rather than Sandy Bridge on `Raijin`. I could not reproduce the problem. As shown in Fig. 5.6, the scalability of Chapel reduce is close to linear, and the performance is consistently higher than that of OpenMP. One might notice the performance drop around 16 cores for Chapel. Since the machine has 12 physical cores per socket, using more than 12 threads may lead to communication between sockets, which is likely to be the cause of above problem.

Different Problem Size

Instead of changing the hardware, I also tried to change the problem size. In particular, I changed the length of the vectors owned by individual threads from 15000000 to 12000000. Based on my experience, the size of working set does sometimes affect the performance due to cache effect. From Fig. 5.7, we can see that with this problem size, the performance of Chapel does not suffer from the slowdown seen in Fig. 5.2. Furthermore, its performance is consistently higher than OpenMP for all configurations. When the number of physical cores is 16 (one full node), there is high variance in the measurements of the throughput of Chapel. I did not investigate this issue due

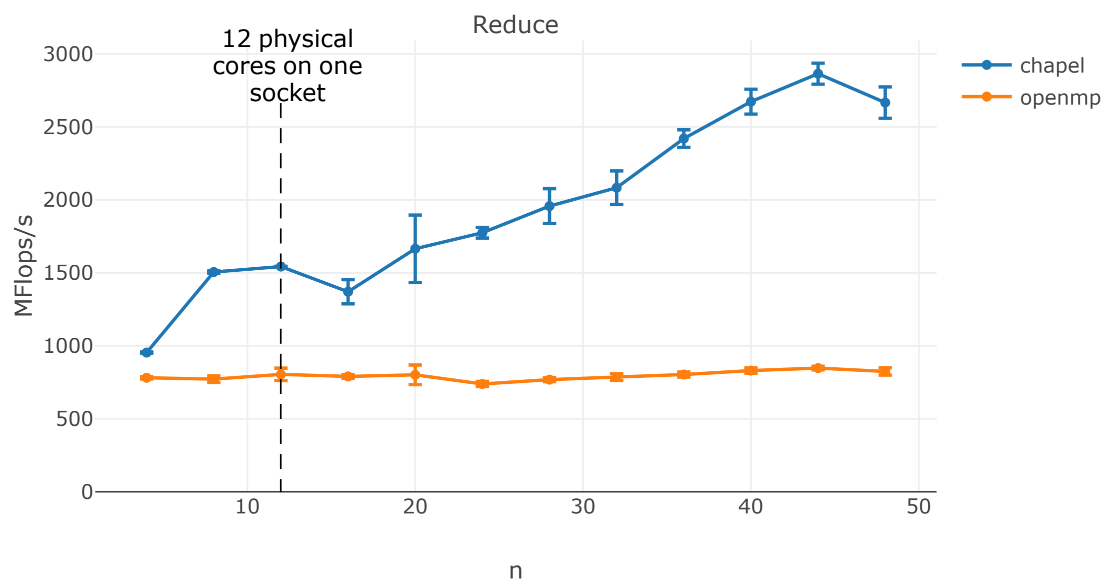


Figure 5.6: Performance of reduce kernel with number of cores: Chapel vs OpenMP on weasel

to time constraints.

Based on the above two experiments, they suggest there are pathological conditions when the runtime, operation system scheduling and the microarchitecture interact in certain way. Due to my time constraints, I could not investigate this further. Here are some of my speculations.

1. **Cache effect** Accessing array elements may generate cache misses. However, it is unlikely to be the cause of the problem. First, the benchmark features intensive sequential reads and writes to arrays, where the data is effectively streamed from memory. It exhibits good spatial locality which can makes good use of the hardware prefetcher. Second, since the array is sufficiently big, having a small number of elements crossing the page boundary might not have big impact on TLB misses.

2. **Operating system scheduling** Another aspect that `weasel` differs from `Raijin` is that they are running different versions of Linux kernel. On both `Raijin` and `weasel`, Chapel uses `qthreads` as the tasking layer. Chapel tasks will be executed by `qthreads`, which in turn uses OS threads. The Linux kernel, which manages these OS threads, uses different scheduling algorithms on different releases. As a result, the difference in the version of Linux kernels might affect the runtime behaviour of Chapel programs. One way of verifying is to make the two machines run the same version of Linux kernel, and redo the experiments. However, this is not very practical. `Raijin` is centrally managed by the NCI, where I cannot specify the version of Linux kernel. `weasel` is running on relatively new CPUs, which may not be supported by the old kernel.

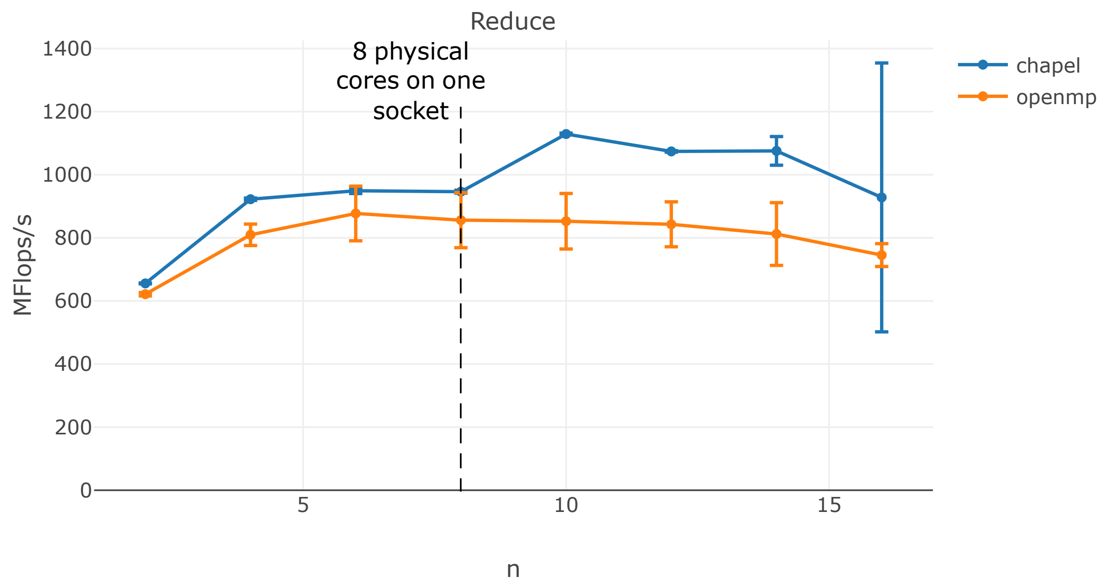


Figure 5.7: Performance of reduce kernel using 12000000 as the vector length

3. **Chapel tasking layer** The task and data parallelism[1] in Chapel are hints to the compiler for generating concurrent tasks. These tasks are scheduled to a number of threads during runtime. However, threads are opaque to programmers, and it is not guaranteed which thread will be used to execute a task.

```
for i in 1..iterations {
    forall j in 0..vectorLength-1 {
        vector[j] += 1;
    }
}
```

   As shown in the code snippet above, for different iterations of the outer loop, different threads might be used to compute the same location of the array, which can leads to bad spatial locality.

5.2.4 Summary

The reduce benchmark shows that by writing code in a natural, high-level fashion, the Chapel compiler and runtime will be better informed and can perform certain optimizations. In contrast, the OpenMP code suffers from serial bottleneck due to explicit threading model. This shows that clean and maintainable code is not always at odds with performance.

[1]The number of tasks generated by data parallelism depends on the size of iterators, available hardware, etc.

5.3 Transpose

The transpose benchmark from the PRK suite stresses communication and memory bandwidth. Each process owns a block of the original matrix and the transposed matrix. Depending on whether the columns or rows are grouped into a block, the benchmark can feature unit-stride reads and non-unit stride writes (see Section 2.4.2), or vice versa.

The MPI and MPI/OpenMP implementations are used as our baseline here. The MPI/OpenMP implementation uses OpenMP for multithreading within a process, and uses MPI for interprocess communication. The problem size, namely the order of the square matrix, is proportional to the square root of the number of physical cores. Tiling is applied when running the benchmark, so that each individual matrix block is accessed as smaller tiles to reduce cache and TLB misses.

5.3.1 Result

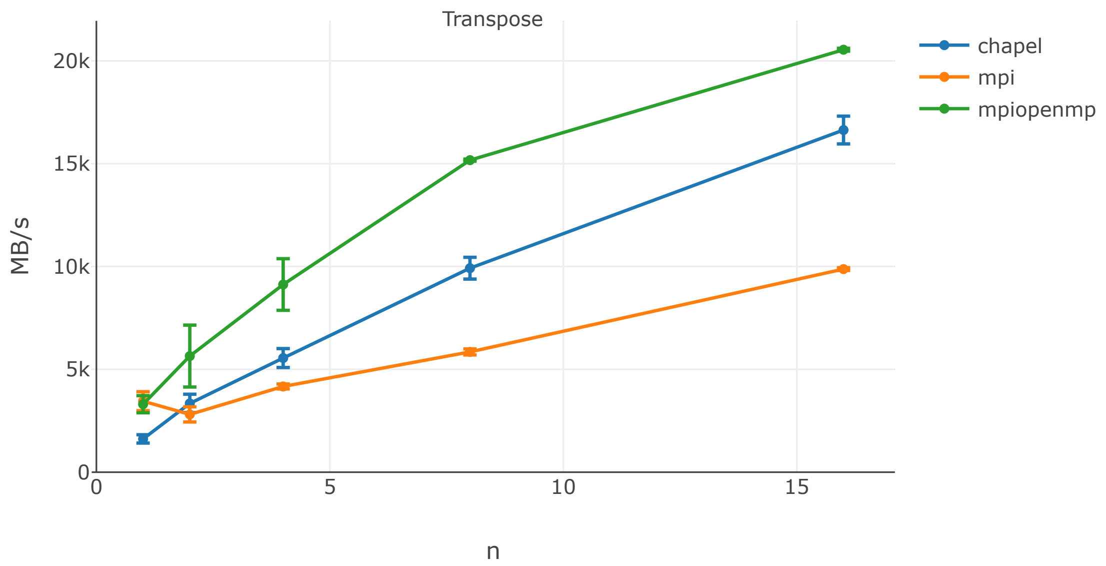
As shown in Fig. 5.8(a), all three of the implementations show good scalability within one node (16 cores). The MPI/OpenMP hybrid implementation is the fastest one, followed by Chapel and then MPI. Both MPI/OpenMP and Chapel only launch one process per node with multiple threads, while MPI spawns one process per code. It shows that Chapel does have competitive performance within a node. Furthermore, using processes instead of threads leads to higher overheads.

As shown in Fig. 5.8(b), MPI keeps scaling beyond one node at nearly the same rate. Both MPI/OpenMP and Chapel do not scale beyond one node. The performance MPI/OpenMP drops significantly after one node. Although it speeds up later when more nodes are added, the throughput of MPI/OpenMP on 8 nodes just matches the throughput on one node. The performance of Chapel is decreasing in general, and its performance on 8 nodes is nearly 50% slower than the performance on one node.

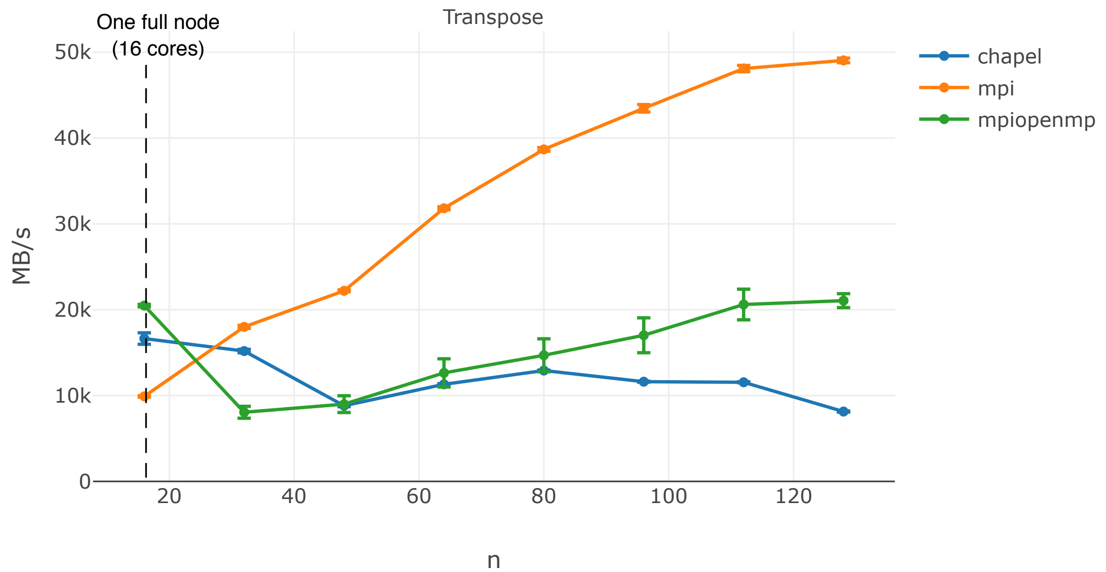
5.3.2 Limits to Multi-node Scaling

By investigating the MPI/OpenMP code, the performance degradation when using more than one node is likely caused by blocking communication. After processing the local data, all other threads are idling while the master thread starts the communication with other processes. This scheme is known as bulk synchronous communication. Whiling sending and receiving data, the entire node is doing nothing other than communication. This serial part of the program introduces an Amdahl bottleneck (see Section 2.2.1), and limits the speedup one can get.

By examining the the generated C code for Chapel, it suggests that the fine-grained communication between locales prevents the scaling. Concretely, accessing an entry of the matrix on another locale results in communication operations, such as `GET`. This is confirmed by using two locales on a single node while keeping the number of physical cores the same. As shown in Fig. 5.9, the performance trend for using one locale is nearly linear. In contrast, the performance boost is not proportional to



(a) Within a node



(b) Multiple nodes

Figure 5.8: The performance of matrix transpose for all three implementations

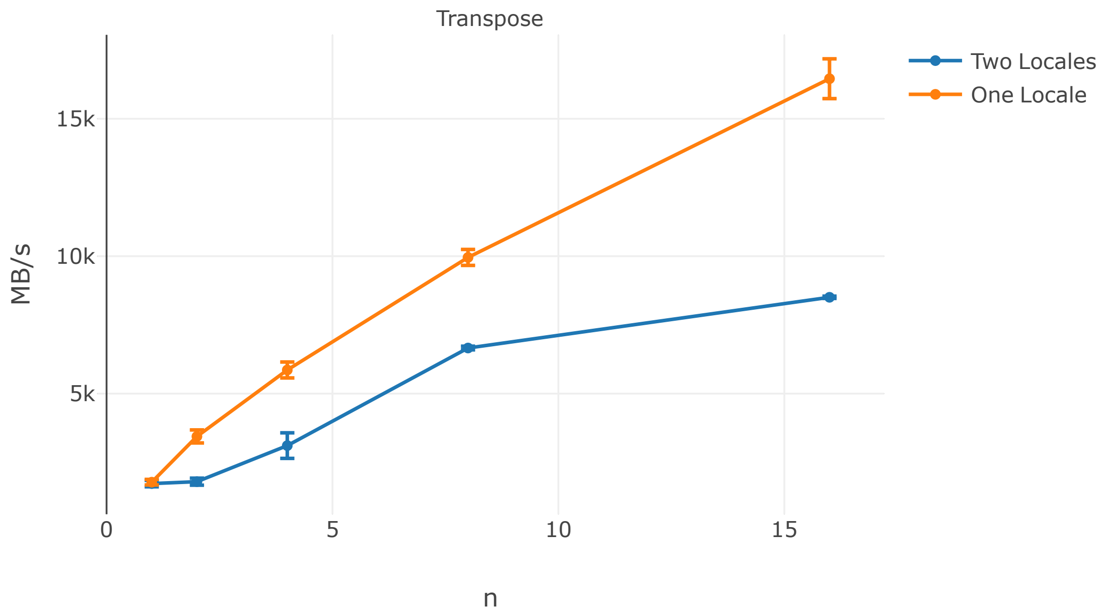


Figure 5.9: The performance of transpose written in Chapel using one vs two locales on a single node

the increase in the number of cores if we are using two locales, which is reflected in Fig. 5.9 as decreasing slopes of blue segments.

5.3.3 Improvement

The performance of MPI/OpenMP can be improved by using asynchronous communication. Currently, the master OpenMP thread is used to communicate and the computation is blocked until the data is completely transferred. However, by having dedicated background threads to handle communication, other threads can continue to do local work while the data is transferred. This approach overlaps computation and communication, and thus relieve the Amdahl bottleneck introduced by blocking communication.

Bulk communication could be used to improved the performance of Chapel in this case. Since each communication action has associated constant cost, increasing the message size will make the throughput higher. In the current implementation, Chapel only optimizes assignments of whole arrays or array slices, but not for single elements even when those elemental assignments are independent. We could extend this approach so that the elemental assignments are collected first, and then transferred to destination locales in bulk. This will reduce the overheads incurred for fine-grained communication, since the overhead is amortized when the communication is done in bulk.

One caveat of this approach is that the sequence of elemental assignments to be optimized needs to be independent, so that the assignments can be reordered. When the dependency analysis is too complex, the compiler might need hints from the

programmers, stating whether the assignments are independent. Chapel currently provides the syntax for stating intents for reduction. Similar syntactic supports can be deployed to express data dependency.

5.3.4 Summary

Global-view programming is not a panacea. The implicit communications created by the runtime are not obvious to programmers, and can lead to performance problems. One potential solution is to let the compiler and runtime collectively batch up communication. How the communication is done is also important. As discussed in Section 5.3.2, blocking communication introduces Amdahl bottleneck. Since it supports asynchronous PGAS, Chapel can overlap communication and computation. However, whether the compiler is generating expected efficient code is yet to be confirmed.

5.4 Summary

This chapter presents my preliminary survey of the performance of Chapel high-level constructs by using a small number of benchmarks. In the reduce benchmark, by using high-level operations, such as array addition, the Chapel implementation is both clean and performant. It shows that high-level constructs can have competitive performance. In the transpose benchmark, PGAS separates the concern of data distribution and the algorithm, and makes the program concise. However, the use of global-view programming in Chapel leads to implicit fine-grained communications, which is not currently optimized by the Chapel compiler. This leads to bad scalability compared with MPI and MPI/OpenMP implementation.

Chapter 6

Conclusion

Computing hardware is becoming increasingly parallel. Improving the efficiency of parallel systems requires us to bridge the gap between the hardware and the programming model we use. Current mainstream parallel models suffer from productivity problems. In this report, I contend that high-level constructs in Chapel are potential solutions to this problem.

In Chapter 3, I showed how Chapel can be used to tackle the above problem through the use of high-level constructs. I showed that how code can be written succinctly in Chapel through the use of data parallelism and global-view programming. I also showed how memory model, caching and task affinity can affect the performance of Chapel programs. During the study, I found two bugs in the runtime code of Chapel, and proposed fixes to them.

Chapter 4 showed how various techniques, such as convergence-based warmup, can reduce the variance in the measurements. These techniques are distilled into a benchmarking framework `menthol`.

Finally, the reduce and transpose kernels were used to examine the performance implications of high-level constructs in Chapel in Chapter 5. The reduce benchmark suggested that by supporting implicit data parallelism, array additions can be expressed succinctly and be automatically parallelized. The global-view programming separates the concern of algorithms from the data layout. However, the transpose benchmark showed that Chapel hides the underlying communication over the network and implicitly generates fine-grained communication that degrades the performance. To fix this problem, I suggested potential optimizations to bulk transfer data between locales. Through these two benchmarks, I showed that the high-level constructs in Chapel can potentially lead to both productive programming and performant programs, provided sufficient compiler optimizations.

6.1 Future Work

Due to my time constraint, there are several directions that I could do better. The following sections focus on potential future extensions to this work.

6.1.1 Non-temporal Hints for Synchronization Variables

Section 3.4 discussed the cache effects of synchronization variables, in particular, constantly flushing cache and false sharing. Non-temporal hints are supported on some hardware platforms to allow bypassing the cache hierarchy when performing memory operations. Future work can use these hints to implement synchronization variables for Chapel, and evaluate its performance impact.

6.1.2 Integrating Convergence-based Warmup to More Benchmarks

Section 4.3 demonstrated how the convergence-based warmup methodology could be use to reduce the variance in benchmark results. However, it is currently only integrated into the Chapel implementation of reduce. To make this method available to other languages and platforms, I have rewritten it in Rust so that a shared library can be built. In the future, benchmarks written in other languages, such as C and C++, can use the same mechanism when performing measurements.

6.1.3 More Comprehensive Benchmarking

Chapter 5 discussed my evaluation on the high-level constructs of the Chapel language. However, only the transpose and reduce benchmarks from PRK were used as examples due to time constraints. In the future, I would like to use a wider selection of benchmarks to cover more real life scenarios, such as synchronization, mutual exclusion, sparse matrix, etc. These benchmarks can exercise more features of the Chapel language, such as synchronization variables and numerical libraries. A more comprehensive evaluation could help application programmers make more informed choice on the language they use. In return, these benchmarks can also affect the design decisions of the Chapel language.

6.1.4 Comparison with Other Models

In this work, mainstream programming models, such as OpenMP and MPI, are used as baselines. However, there are other interesting parallel models to evaluate.

X10[1], which is another asynchronous PGAS language, has a lot of overlapping features compared with Chapel. As discussed in Section 5.3.2, Chapel does not require explicit communication when accessing data on another locale. As a result, programmers might not be aware of the potential cost of the communication. One notable difference between X10 and Chapel is that X10 requires the explicit use of the `at` construct whenever the communication may potentially happen over the network. The productivity and performance implication of the `at` construct is worth investigating for future work.

Unlike most of parallel models, which are statically typed, Julia[2] is a dynamically typed high-performance programming language. The performance implication of

[1] http://x10-lang.org
[2] https://julialang.org

type information, or lack thereof, is also worth exploring.

Bibliography

AMDAHL, G. M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring) (Atlantic City, New Jersey, 1967), 483–485. ACM, New York, NY, USA. doi:10.1145/1465482.1465560. http://doi.acm.org/10.1145/1465482.1465560. (cited on page 4)

BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHANG, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIĆ, D.; VANDRUNEN, T.; VON DINCKLAGE, D.; AND WIEDERMANN, B., 2006. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06 (Portland, Oregon, USA, 2006), 169–190. ACM, New York, NY, USA. doi:10.1145/1167473.1167488. http://doi.acm.org/10.1145/1167473.1167488. (cited on page 20)

BLACKBURN, S. M.; MCKINLEY, K. S.; GARNER, R.; HOFFMANN, C.; KHAN, A. M.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIK, D.; VANDRUNEN, T.; VON DINCKLAGE, D.; AND WIEDERMANN, B., 2008. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51, 8 (Aug. 2008), 83–89. doi:10.1145/1378704.1378723. http://doi.acm.org/10.1145/1378704.1378723. (cited on page 20)

BRENT, R. P., 1974. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21, 2 (Apr. 1974), 201–206. doi:10.1145/321812.321815. http://doi.acm.org/10.1145/321812.321815. (cited on page 6)

CHAMBERLAIN, B.; CALLAHAN, D.; AND ZIMA, H., 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21, 3 (2007), 291–312. doi:10.1177/1094342007078442. https://doi.org/10.1177/1094342007078442. (cited on pages 12 and 13)

CRAY INC, 2018. Chapel language specification. https://chapel-lang.org/docs/master/_downloads/chapelLanguageSpec.pdf. [Online; accessed 24-April-2018]. (cited on page 13)

FLYNN, M. J., 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, 9 (Sept 1972), 948–960. doi:10.1109/TC.1972.5009071. (cited on page 11)

Gustafson, J. L., 1988. Reevaluating amdahl's law. *Commun. ACM*, 31, 5 (May 1988), 532–533. doi:10.1145/42411.42415. http://doi.acm.org/10.1145/42411.42415. (cited on page 5)

Hoefler, T. and Belli, R., 2015. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15 (Austin, Texas, 2015), 73:1–73:12. ACM, New York, NY, USA. doi:10.1145/2807591.2807644. http://doi.acm.org/10.1145/2807591.2807644. (cited on page 20)

ISO, 2011. Information technology – programming languages – c. Standard, International Organization for Standardization, Geneva, CH. (cited on page 15)

McCool, M. D.; Robison, A. D.; and Reinders, J., 2012. *Structured parallel programing: patterns for efficient computation*. Elsevier, Morgan Kaufmann, Amsterdam. ISBN 978-0-12-415993-8. (cited on page 6)

Moreaud, S.; Goglin, B.; and Namyst, R., 2010. Adaptive mpi multirail tuning for non-uniform input/output access. In *Recent Advances in the Message Passing Interface*, 239–248. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 16)

Nanz, S.; West, S.; and da Silveira, K. S., 2013. Benchmarking usability and performance of multicore languages. *CoRR*, abs/1302.2837 (2013). http://arxiv.org/abs/1302.2837. (cited on page 23)

Olukotun, K. and Hammond, L., 2005. The future of microprocessors. *Queue*, 3, 7 (Sep. 2005), 26–29. doi:10.1145/1095408.1095418. http://doi.acm.org/10.1145/1095408.1095418. (cited on page 3)

Sorin, D. J.; Hill, M. D.; and Wood, D. A., 2011. *A primer on memory consistency and cache coherence*. No. 16 in Synthesis lectures on computer architecture. Morgan & Claypool, San Rafael, Calif. ISBN 978-1-60845-564-5 978-1-60845-565-2. OCLC: 930741576. (cited on page 15)

Wijngaart, R. F. V. d. and Mattson, T. G., 2014. The Parallel Research Kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–6. doi:10.1109/HPEC.2014.7040972. (cited on page 21)

Yang, X.; Blackburn, S. M.; Frampton, D.; Sartor, J. B.; and McKinley, K. S., 2011. Why nothing matters: The impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11 (Portland, Oregon, USA, 2011), 307–324. ACM, New York, NY, USA. doi:10.1145/2048066.2048092. http://doi.acm.org/10.1145/2048066.2048092. (cited on page 16)