Preliminary Investigation of Productivity Tools for Memory Profiling in Parallel Programs

by

Elizabeth Zou
S.B. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 6, 2022

Certified by .....
Charles E. Leiserson
Professor
Thesis Supervisor

Certified by .....
Tim Kaler
Postdoctoral Associate
Thesis Supervisor

Certified by .....
Alexandros-Stavros Iliopoulos
Postdoctoral Associate
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Preliminary Investigation of Productivity Tools for Memory Profiling in Parallel Programs

by

Elizabeth Zou

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As computing efficiency becomes constrained by hardware scaling limitations, code optimization grows increasingly important as an area of research. The impact of certain optimizations depends on whether a program is compute-bound or memory-bound. Memory-bound computations especially benefit from program transformations that improve their data locality, to better exploit modern memory hierarchies. Reuse distance is a useful measure for analyzing data locality in an architecture-agnostic way, i.e., independent of specific cache sizes. Previous work has researched different ways to calculate reuse distance, ranging from deterministic to probabilistic and using different definitions of reuse distance.

This thesis investigates the use of static compiler instrumentation tools to implement memory analysis tools for parallel programs. I show how the comprehensive static instrumentation (CSI) framework can be used to compute the reuse-distance of memory locations in a sequential execution of a program. For analyzing parallel programs, it is necessary to contextualize the memory access patterns with the logical parallel structure of the code. To this end, I show how reuse distance calculations can be organized according to the logical parallel structure of the program by building a series-parallel tree using CSI. I present several potential algorithms for using this instrumentation to calculate statistics for average and peak memory bandwidth in parallel codes. Although these instrumentation tools remain prototypes, they constitute a compelling proof-of-concept for the use of CSI to perform memory analysis in parallel codes.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Thesis Supervisor: Tim Kaler
Title: Postdoctoral Associate

Thesis Supervisor: Alexandros-Stavros Iliopoulos
Title: Postdoctoral Associate

Research Acknowledgments

This research was sponsored in part by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Acknowledgments

Thanks to my supervisor, Professor Charles E. Leiserson of MIT CSAIL and the Supertech Research Group for their support of this thesis work. Specifically, huge thanks to my advisers Tim Kaler and Alexandros-Stavros Iliopoulos for their weekly advice, the large amount of time spent in the last week helping me edit the thesis before submission, and for being great people to work with. Thanks to Tao B. Schardl as well for his helpful comments.

Thanks to fellow Supertechie and MEng student Wanlin Li for helping me fix figures at 4:59AM, among other things. Thanks to my family, my boyfriend Michael Huang, and all my other friends for their support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming today is unlike programming fifty years ago. Whether it be for software, simulations, or data science, codebases are growing increasingly larger and more complicated. As Moore's Law scaling comes to an end [18, 20, 29], optimizing memory access becomes increasingly important and necessary to maintain and enable further improvements in computing efficiency.

The cost of memory accesses in a program depends both on the spatial and temporal locality. Modern hardware architectures employ a hierarchical cache structure. Computations with good data locality are served by the smaller, faster levels of the hierarchy. Conversely, computations which exhibit poor data locality tend to benefit less from hardware caches and consequently have worse performance. With advances in processor speed outpacing those of memory [34], the importance of optimizing a program's memory locality has grown over time.

Memory analysis tools and profilers have proven to be highly effective at helping programmers understand and optimize the performance of their programs. One area of research focuses on code analysis tools that help programmers understand and instrument the memory accesses of both serial and parallel programs. An example is Cachegrind, a tool suite part of Valgrind [23] for simulation based analysis of memory access behavior. Another example is ThreadSanitizer [30], a dynamic detector of data races.

These tools are typically unaware of the logical parallel structure of a program, but

can still provide useful insights by profiling each thread in the program independently. Tools like this can analyze parallel executions to measure memory locality, memory bandwidth, cache behavior, cache contention, and detect data races. Such tools, however, are often only able to measure a particular execution of a parallel program because they lack visibility into the program's logical parallel structure.

For certain problems, such as race detection, there is precedent for using the logical parallel structure of the program to provide more useful program analysis tools. For example, the Cilksan CSI tool [27] implements a provably correct race detector that detects race conditions based on the logical parallel structure of the program. This enables Cilksan to provide stronger guarantees than other tools, like ThreadSanitizer, which only detect race conditions that actually occur during a particular parallel execution of a code. Another example is Cilkmem [16], which computes the worst-case p-processor memory high-water mark for parallel programs that perform dynamic memory allocations.

In my thesis, I explore the potential of comprehensive static instrumentation (CSI) [26] as a framework for implementing memory analysis tools that are able to understand the logical parallel structure of a task-parallel program. I begin by showing how to implement a reuse distance calculator within CSI and demonstrate that reuse distance correlates with program runtime performance using a set of microbenchmarks with common patterns of memory-access reorganization. Next, I show how the reuse distance calculator can be adapted to detect, dynamically at runtime, memory accesses that are likely to result in data movement across different levels of a machine's cache hierarchy. These statistics based on reuse distance can be organized within a series-parallel tree, also termed spawn tree [12], that is built using CSI. Lastly, I propose algorithms for analyzing the series-parallel tree, augmented with reuse distance statistics, to analyze the average and worst-case p-processor memory bandwidth of a parallel program. These tools and algorithms remain prototypes, but they serve to demonstrate the viability of performing memory analysis based on logical parallel structure using CSI.

The remainder of this chapter is structured as follows. Section 1.1 provides addi-

tional motivation for memory analysis tools for parallel codes. Section 1.2 explains instrumentation decisions and background on the framework I chose to use. Section 1.3 defines reuse distance and provides examples showing how reuse distance measures memory locality, and Section 1.3.1 walks through an example program that demonstrates how reuse distance analysis can provide insight to the memory usage of a program. Section 1.4 gives an overview of the prototype reuse distance calculator I implemented as part of my investigation, and Section 1.5 introduces the memory bandwidth analysis work I completed. Finally, Section 1.6 describes the structure of the remaining chapters of this thesis.

1.1 Motivation

Performance bottlenecks due to poor data locality or suboptimal access patterns are often difficult to diagnose. The typical programmer will often not have a detailed understanding of the memory behavior of their code, or how it impacts performance. Even among expert programmers, it is often difficult to pinpoint memory-related bottlenecks and understand how these bottlenecks impact the overall performance of complex codes or improve the memory usage.

Simple program transformations can impact memory behavior and hence performance. For example, reordering loops, given that correctness is preserved, can benefit a program's performance [1, 2, 22]. A well-known example is matrix multiplication, where memory locality can be significantly improved with a loop reordering. In a similar way, reordering function calls or fields in a struct may also have a significant effect on program performance due to better memory locality. One more example of better memory locality resulting from simple changes to code is reorganizing data from an array of structs to a struct of arrays, or vice versa. But how to we determine which components of a program benefit from the application of these transformations? My thesis investigates technologies that make it easier for programmers to identify memory-related bottlenecks in their code, allowing them to target critical sections of their program for further optimization.

1.2 Instrumentation

This section describes the instrumentation technologies I used to implement prototypes of the memory analysis tools described in this thesis. The specific instrumentation framework I use is the Comprehensive Static Instrumentation framework (CSI) which has been used to implement other tools for parallel programs written using Cilk [6], such as the Cilksan race detector [27], Cilkscale scalability profiler [27], and Cilkmem memory high-water mark analyzer [16]. CSI is a framework that can be used to inject code into programs so that a variety of profiling tools, such as race detectors, cache simulators, code-coverage analyzers, etc. can "observe and investigate runtime behavior" [26]. My work serves as a demonstration of memory analysis techniques for measuring data locality and parallel memory bandwidth using CSI.

CSI enables writing instrumentation tools as C libraries. Under CSI, profiling tools can insert custom instrumentation on top of the compiler, with standard hooks available at many parts of the execution, such as before and after each memory access, before and after each function call, etc. CSI provides this flexibility without much of a tradeoff in terms of efficiency and performance. CSI is also accessible – it is very easy for other researchers to take an existing tool and make changes to it, or extend its functionality.

CSI allows instrumentation of the logical structure of a task-parallel program. The way CSI accomplishes this is by adding hooks around specific elements of the LLVM IR. The Tapir/LLVM extension [28] augments LLVM's intermediate representation with 3 additional instructions (detach, reattach, and sync) that are used to express logical parallelism in a program. Consequently, a CSI tool can instrument the logical parallel structure of a program by implementing hooks for these Tapir/LLVM instructions. All of the above are readily available with OpenCilk [27], making it simple and straightforward for Cilk programmers to create sophisticated and helpful profiling tools while keeping the overhead low.

There are alternative instrumentation approaches such as the use of hardware performance counters [35], binary instrumentation techniques [4, 40], and compiler

passes [38], each with its own pros and cons. Ultimately, I chose to explore the feasibility of CSI as a framework for memory analysis because it allows you to investigate the logical parallel structure of a program, and because I believe that the ease with which tools written using CSI can be extended facilitates further research.

1.3 Reuse Distance

In this section, I introduce reuse distance and explain why it is a useful program property to analyze. Reuse distance is a useful program property due to its relation to memory or reference locality, the tendency of a processor to access the same set of memory locations repetitively over a short period of time [32]. Because modern machines have hierarchical memory, with the most frequently accessed data being the fastest to access, analyzing memory locality is crucial to understanding the memory behavior of programs.

There are two basic types of memory locality:

- *Temporal locality* refers to the reuse of specific data within a relatively small time duration.
- *Spatial locality*, also termed data locality [11], refers to the use of data elements close in memory location.

One of the common metrics used to analyze program memory locality is the concept of reuse distance, also known as stack distance [5]. Generally speaking, in a sequential execution, reuse distance is the number of *distinct* data elements accessed between two consecutive references to the same element. Reuse distance is a useful measurement due to its relationship to cache miss ratios (given some cache size) [31] and various other locality measures [39].

A hypothetical analysis of reuse distance's relationship to cache miss ratios despite being inherently cache-oblivious is as follows. If a memory access has reuse distance of 33000, and the machine the program is running on has L1 cache size of 32000, then under the naive LRU stack algorithm, accessing that data value misses the L1

cache. More realistically, we can measure reuse distance based on cache lines, rather than individual memory locations. This way, if cache line size is 64 and the L1 cache size of a machine is 32000, then a memory access to a cache line with reuse distance $505 > 500 = 32000 \div 64$ will incur a miss to the L1 cache and thus higher memory costs.

1.3.1 An Illustrative Example with Matrix Multiplication

In this section, I show how reuse distance calculations can measure the locality of a program's memory accesses. Specifically, I illustrate the use of a reuse distance calculator on an example program that performs matrix multiplication. We will walk through two versions of the example program, what memory behavior we expect to see in each version, and how that relates to program runtime, along with the output from actually computing the metrics we are looking for.

A simple example I analyzed is matrix multiplication. It is well known that for the naive algorithm with triple nested for loops, different nesting orderings of the loop index can lead to drastically different program runtimes. Figure 1-1 shows one implementation. If N is the size of the matrix and we set $N = 1000$, this first and slower implementation runs in 4.791s. However, a simple loop reordering as in Figure 1-2 gives a much faster algorithm: with $N = 1000$ as well, this second and faster implementation, after loop reordering, runs in 0.426s.

Figure 1-1 Matrix Multiplication (`ijk`)

```
1: for each i < N do
2:     for each j < N do
3:         for each k < N do
4:             c[i][j] ← a[i][k] × b[k][j]
5:         end for
6:     end for
7: end for
```

The differences in performance of these two codes for matrix multiplication can be better understood by analyzing each implementation's memory access patterns. If we look at the line of code within the triple nested for loops, we notice that if each

Figure 1-2 Matrix Multiplication (`ikj`)

```
1: for each i < N do
2:     for each k < N do
3:         for each j < N do
4:             c[i][j] ← a[i][k] × b[k][j]
5:         end for
6:     end for
7: end for
```

loop iteration uses a different value for i, then `c[i][j]` and `a[i][k]` will incur high memory costs; similarly, if each iteration uses a different value for k, then `b[k][j]` will incur a high memory cost. This is because a change in index for the first dimension of the matrix means that a new cache-line is likely accessed for each value. Since both implementations have `i` in the outer-most loop, `c[i][j]` and `a[i][k]` incur relatively low memory costs in both implementations. However, in the first implementation, the access to `b[k][j]` is high memory cost, as the inner-most loop iterates over `k`; in contrast, in the second implementation, the inner-most loop iterates over `j`, so values of `b[k][j]` on the same cache line are accessed one after another.

This difference in memory locality for the accesses to `b[k][j]` explains the large speedup we see from optimizing the memory usage. If we were to measure total reuse distance summed across the entire program execution, we expect matrix multiplication (`ikj`) to have significantly lower total reuse distance than matrix multiplication (`ijk`).

I ran an experiment to measure the reuse distances of these two codes, and present the results as follows. As we expected, when adjusting the parameter N and running the reuse distance calculator, we see that the total reuse distance summed across the program execution scale an order faster for the slower implementation than the faster one. The measured numbers are graphed in Figure 1-3, shown below.

The difference in total reuse distance in the two programs comes almost entirely from the memory usage in the line of code

```
c[i][j] += a[i][k] * b[k][j].
```

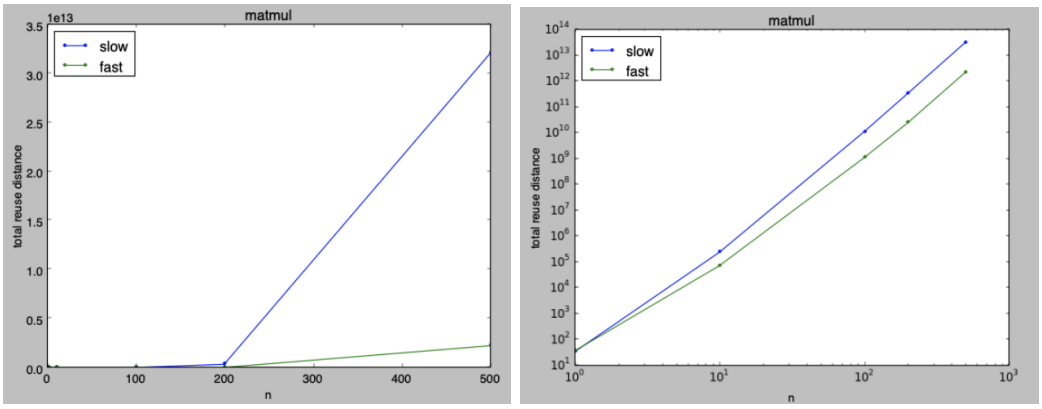

Figure 1-3: Total reuse distance vs. matrix size for two different implementations of matrix multiplication (pseudocodes in Figure 1-1 and Figure 1-2); linear scale (left) and log scale (right).

If we measure reuse distance metrics associated with specific parts of the code, down to a line number and a column number, we expect the memory accesses from `c[i][j]` to have shorter reuse distances than those from `a[i][k]` and `b[k][j]` in matrix multiplication (`ijk`), since `i` and `j` are the two outer loops; similarly, we expect the memory accesses from `a[i][k]` to have shorter reuse distances than those from `c[i][j]` and `b[k][j]` in matrix multiplication (`ikj`), since `i` and `k` are the two outer loops.

Using $N = 500$, tables 1.1 and 1.2 below summarize the reuse distance patterns of the slower and faster implementations respectively.

Memory Access	Average Reuse Distance	RMS Reuse Distance
`c[i][j]`	3.00	3.00
`a[i][k]`	1501.00	1501.00
`b[k][j]`	126625.63	178105.57

Table 1.1: Reuse distance calculations (in number of distinct memory locations between two accesses to the same memory location) for matrix multiplication (`ijk`).

Overall, the output of this reuse distance calculator is sensible and correlates well with what we would expect to see. As expected, the slower implementation has patterns of longer reuse distance, and we can conclude that reuse distance is an effective metric that correlates well with program runtime and can provide insight

Memory Access	Average Reuse Distance	RMS Reuse Distance
`c[i][j]`	129.35	182.34
`a[i][k]`	63.50	124.01
`b[k][j]`	65257.74	65257.74

Table 1.2: Reuse distance calculations (in number of distinct memory locations between two accesses to the same memory location) for matrix multiplication (`ikj`).

to potential speedups in programs. To understand precisely why these numbers are what they are, see Section 4.2. In next sections, I give an overview of the work I completed for my thesis, along with any artifacts developed as part of the study.

1.4 Reuse Distance Calculator

The first part of my thesis study involves examining reuse distance metrics and their relation to program memory behavior. To do so, I created a prototype CSI tool that calculates reuse distance. In Section 1.3, I defined what reuse distance is and why I chose to analyze it. In this section, I briefly discuss the implementation of this reuse distance calculator.

A simple reuse distance calculator may take the form of a linked list [21]. Using the definition of reuse distance presented in Section 1.3, a possible implementation is as follows (Figure 1-4):

Figure 1-4 Reuse Distance Calculator

```
1: for each address do
2:     addrnode ← createNode(address)
3:     if addrnode ∈ list then
4:         reusedistance ← list.index(addrnode)
5:         list.remove(addrnode)
6:     list.insertFirst(addrnode)
7: end for
```

The algorithm in Figure 1-4 has per memory access runtime of $O(N)$, where N is the size of program data, and thus an overall runtime of $O(TN)$, where T is the length of execution. Instead of using this algorithm, I improve the per memory

access runtime to $O(\log N)$ in Section 3 by using a self-balancing binary search tree [3, 25, 33], bringing the overall tool runtime down to $O(T \log N)$.

1.5 Memory Bandwidth Analysis

The second part of my thesis is concerned with analyzing the memory bandwidth requirements of parallel programs. If the memory bandwidth of a program exceeds the theoretical machine upper bound of memory bandwidth, then the program's execution becomes limited by this memory bandwidth, as opposed to lack of computation power. In this section, I discuss how I approached the problem of analyzing a parallel program's memory bandwidth requirements using the code's logical parallel structure.

I propose that a program's average and peak memory bandwidth requirements can be useful measures for understanding the scalability of task-parallel programs and investigate an approach for measuring average and peak bandwidth requirements for scaling a computation using multiple processors. My approach utilizes the series-parallel tree representation of a parallel computation, together with reuse distance measurements within and across subtrees. This makes it possible to collect measurements with a serial execution of a program and use them to reason about the expected effects of parallelism.

The data bandwidth requirement of a program can be defined as the total data movement divided by time. My approach to measuring data movement is based on calculations performed by the reuse distance calculator. Given a reuse distance K, I consider any memory access that has reuse distance greater than K to result in data movement between different levels of cache. Different values of K can be selected to target different cache sizes or different levels of the cache hierarchy. My approach to measuring time is similar to the approach taken in other CSI tools such as Cilkscale [27] where time is measured by the number of LLVM pseudo instructions executed, which are used by the compiler to estimate the cost of program instructions.

The memory bandwidth analysis tool for parallel programs employs the reuse distance calculator's measurements, which are organized within a series-parallel tree.

This is a data structure, maintained at runtime, that captures series-parallel relationships in program execution. Both data movement and time are recorded in this data structure, where program execution is represented with a tree structure of computation nodes, which are either serial or parallel. The children of serial nodes run in series, while the children of parallel nodes run in parallel. The series-parallel tree is described in more detail in Section 5.

I developed prototypes of analysis algorithms for calculating memory bandwidth. The cases covered by these prototype algorithms are the average memory bandwidth of a program assuming infinite processors, the peak memory bandwidth assuming infinite processors, and the peak memory bandwidth given a limited number of processors p. The details of these algorithms are elaborated on in Chapter 6. I also propose ideas for an algorithm computing the average memory bandwidth for p processors in Section 7.2.

1.6 Thesis Structure

The remainder of this thesis is divided as follows. First, I shall explain the reuse distance calculator and analyze benchmark programs empirically. Chapter 2 discusses the benchmark programs I created and selected to use for my evaluations. Chapter 3 describes the reuse distance calculator, and Chapter 4 defines metrics I measure and presents my empirical studies. Second, I modify the tool to perform memory bandwidth analysis. Chapter 5 talks about the integration of a series-parallel tree to model the logical parallel structure of parallel code, and Chapter 6 discusses different algorithms for performing memory bandwidth analysis and presents preliminary results applying these prototypes to simple programs. Finally, Chapter 7 concludes with extensions and ideas I didn't get a chance to work on but could be interesting.

Chapter 2

Microbenchmarks

In this chapter, I describe benchmark programs for testing and demonstrating the functionality of the prototype reuse distance calculator I developed. In the introduction, we used matrix multiplication as an example to demonstrate the functionality of our calculator and how it can be used to give programmers insight as to where the memory bottlenecks of a program may be located. For this example, we created two different implementations of the same algorithm; one with better memory locality than the other, and hence better reuse distance patterns and faster program runtime. I propose a set of microbenchmark programs like this matrix multiplication example, and apply my prototype tool to them, to assess the potential of a reuse distance CSI profiling tool as an aid towards identifying program regions or structures where data locality can be improved. Numerical results are presented in Section 4.2.

I discuss four simple code changes that may lead to different memory usage patterns, and microbenchmarks I came up with to exhibit such changes in memory usage patterns. Specifically, I designed these microbenchmarks to exhibit different reuse distance patterns before and after the code changes, and verified that their runtimes were positively correlated with reuse distance. This is done by creating two different implementations for each case, one before the code change, and one after. I then evaluate the performance and output of these benchmarks on the reuse distance tool we create in Section 4.2.

The microbenchmarks presented here are simple, but they correspond to common

patterns and design decisions which are often found in real programs. They serve to validate the premise of my prototype tool, as well compare the performance and results of different versions of the same tool. This was helpful as I implemented an inefficient but straightforward reuse distance calculator based on the algorithm presented in Figure 1-4, and used it to confirm the accuracy of the more complicated but more efficient one as described in Chapter 3.

2.1 Reordering Nested Loops

One simple code change we aim to target is a reordering of nested for loops. This can lead to a significant improvement in the memory locality of a program when the loop reordering allows data values on the same cache line to be accessed one after another in the inner-most loop. The microbenchmark we proposed for this is the matrix multiplication example, described in more detail in Section 1.3.1.

2.2 Reordering Struct Fields

In a large struct with many fields, putting fields that are loaded or accessed together in loop iterations adjacent to each other in the struct can improve memory locality. This is because by doing so, these loops can touch fewer cache lines. For example, if all fields touched by a loop can be rearranged to fit on one cache line, as opposed to being spread apart on multiple different cache lines, then the total number of distinct cache lines needed to be accessed one after another by that loop decreases, and the peak memory bandwidth as well.

Similarly, we may also want to reorder fields to enable vectorization opportunities. Rather than having a struct with fields of the data types [`char`, `int`, `char`, `int`, `char`, `int`, `char`] in that order, where `char`s are 1 byte and `int`s are 4 bytes, it may be better for performance to reorganize the fields into the order [`int`, `int`, `int`, `char`, `char`, `char`, `char`].

2.3 Reordering Function Calls

Similar to reordering nested loops, assuming that program correctness is preserved, reordering function calls such that those which access overlapping memory addresses are adjacent in program execution may help improve memory locality. For example, if we have arrays $arr1$ and $arr2$, and functions myFunc1 and myFunc2 that operate on arrays, then a program with alternating accesses to the two arrays (Figure 2-1) can achieve better memory locality by reordering the function calls (Figure 2-2).

Figure 2-1 Function Calls 1

```
1: myFunc1(arr1)
2: myFunc1(arr2)
3: myFunc2(arr1)
4: myFunc2(arr2)
```

Figure 2-2 Function Calls 2

```
1: myFunc1(arr1)
2: myFunc2(arr1)
3: myFunc1(arr2)
4: myFunc2(arr2)
```

In this scenario, we expect the code snippet presented in Figure 2-2 to run faster than the code snippet presented Figure 2-1 due to the more efficient memory access patterns. The significance of the memory locality improvement and thus the speedup is related to the cache sizes of the machine and the number cache misses at each level incurred by the two programs. For the maximum difference in performance between the two implementations, the arrays are sized to perfectly fill up a given cache level.

2.4 Array of Structs vs. Struct of Arrays

The last code change we propose microbenchmarks for is changing data organization from an array of structs to a struct of arrays, or vice versa. In many software applications, we end up having data similar to a matrix of values, e.g. a list of students each with associated data like name and age. In these scenarios, we can have a `Student`

struct with fields `name` and `age` (Figure 2-3), and store data as an *array of structs*. Alternatively, we can have a `Students` struct with fields `names` and `ages`, where `names` and `ages` are both arrays and `Students` is thus a *struct of arrays* (Figure 2-4).

Figure 2-3 Array of Structs

```
1: struct STUDENT
2:     name : STRING
3:     age : INT
4:     height : INT
5:     weight : INT
6: end struct
```

Figure 2-4 Struct of Arrays

```
1: struct STUDENTS
2:     names : STRING ARRAY
3:     ages : INT ARRAY
4:     heights : INT ARRAY
5:     weights : INT ARRAY
6: end struct
```

But which of these is better? Depending on the application, the answer may swing either way. If we only select a few students but go through all the fields associated with each student, we expect to see the array of structs perform better. An example program where we expect to see this result is shown in Figure 2-5.

In this program, we expect GETAVGSARRAYOFSTRUCTS to perform better than GETAVGSSTRUCTOFARRAYS. This is intuitive from the memory access patterns triggered by lines 6 through 8 and 16 through 18. Because we are only querying one out of every one hundred students, the array of structs data structure, which keeps the data values associated with each individual student adjacent in memory, is expected to have better memory locality and thus perform better than the struct of arrays data structure, where the queries jump back and forth in memory.

On the other hand, if we access all the students' ages, but not the names or other data of the students, then we expect the struct of arrays to perform better. An example program where we expect to see this result is shown in Figure 2-6.

In this program, we expect GETAVGAGESTRUCTOFARRAYS to perform better

Figure 2-5 Array of Structs Program

```
1: function getAvgsArrayOfStructs(n, people)
2:     totalage ← 0
3:     totalheight ← 0
4:     totalweight ← 0
5:     for each i < n, 100 | i do
6:         totalage ← totalage + people[i].age
7:         totalheight ← totalheight + people[i].height
8:         totalweight ← totalweight + people[i].weight
9:     end for
10:    return totalage ÷ n, totalheight ÷ n, totalweight ÷ n
11:
12: function getAvgsStructOfArrays(n, people)
13:    totalage ← 0
14:    totalheight ← 0
15:    totalweight ← 0
16:    for each i < n, 100 | i do
17:        totalage ← totalage + people.ages[i]
18:        totalheight ← totalheight + people.heights[i]
19:        totalweight ← totalweight + people.weights[i]
20:    end for
21:    return totalage ÷ n, totalheight ÷ n, totalweight ÷ n
```

Figure 2-6 Struct of Arrays Program

```
1: function getAvgAgeArrayOfStructs(n, people)
2:     totalage ← 0
3:     for each i < n do
4:         totalage ← totalage + people[i].age
5:     end for
6:     return totalage ÷ n
7:
8: function getAvgAgeStructOfArrays(n, people)
9:     totalage ← 0
10:    for each i < n do
11:        totalage ← totalage + people.ages[i]
12:    end for
13:    return totalage ÷ n
```

than getAvgAgeArrayOfStructs. This is because of the memory access patterns triggered by lines 4 and 10. The ages are stored consecutively in memory in the struct of arrays data structure, but not in the array of structs data structure; so if we were to count the total number of cache lines touched by these memory accesses,

which is correlated to the amount of data moved between cache levels, we will see that the memory accesses in line 10 touch significantly fewer cache lines than those in line 4, thus giving the struct of arrays better memory locality.

Chapter 3

The Reuse Distance Calculator

This chapter describes my implementation of a prototype reuse distance calculator, which is a tool that observes the execution of a program and tracks memory usage with reuse distance (as defined in Section 1.3). The analysis of a program's execution in my tool takes place online, where each memory access is processed as it happens during program execution, and the collected reuse distance information is aggregated at the end of the program to report metrics such as the total reuse distance, average reuse distance, or the number of memory accesses with reuse distance $> N$.

In the introduction, I presented the algorithm for a simple reuse distance calculator given an address trace (Figure 1-4), which calculates the reuse distance of each individual memory access using a linked list [21]. Although inefficient with a per memory access runtime of $O(N)$ and overall runtime of $O(TN)$, where N is the size of program data and T is the length of execution, this algorithm is simple and straightforward, and I implemented it to provide a benchmark reuse distance calculator to compare a second version of the tool against for debugging purposes.

The remainder of this chapter is structured as follows. Section 3.1 describes the second implementation of the reuse distance calculator, which compared to the first implementation, trades off complexity for efficiency. Section 3.2 suggests extensions to the prototype tool that can enable more complicated analyses and expand its use cases.

3.1 Tree Implementation

A more efficient implementation of the reuse distance calculator is a self-balancing binary search tree of unique memory accesses, arranged by time of access [3, 25, 33]. This brings the per memory access runtime from $O(N)$ down to $O(\log N)$, where N is the size of program data, and the overall reuse distance calculator runtime from $O(TN)$ down to $O(T \log N)$, where T is the length of execution. This approach simplifies the interval tree of holes method proposed by Almási et al. [3] with the observation that for the sake of calculating reuse distances of each individual memory access, we do not need to track holes at all.

The Binary Search Tree

In this subsection, we describe the data structures utilized for the reuse distance calculator. We implement a self-balancing binary search tree, where each node in the tree represents a memory address. In addition, we maintain a mapping from each memory address to its corresponding `TreeNode`. This way, given a memory address, we can easily find its place in the tree. Figure 3-1 below shows the declaration of the `TreeNode` struct.

Figure 3-1 TreeNode

```
1: struct TreeNode
2:     left : TreeNode
3:     right : TreeNode
4:     parent : TreeNode
5:     size : int
6:     addr : uintptr_t
7: end struct
```

In our implementation, we also maintain two invariants:

- *The memory addresses in the tree are unique.* For example, if we have address trace $ABBCBDA$, the set of memory addresses in the tree will be $\{A, B, C, D\}$.
- *The memory addresses in the tree are sorted by order of last access.* For example, if we have address trace $ABBCBDA$, the memory addresses in the tree will be

$\{C, B, D, A\}$, in that order.

Reuse distance is easy to calculate assuming these two invariants. Given a memory access and the corresponding memory address x, the reuse distance of that access is the number of nodes in the tree that come after the node corresponding to x. We can calculate this by find the node corresponding to x and its place in the tree using the mapping we maintain, then using the augmented subtree sizes to calculate the number of nodes that come after it. This calculation is performed in $O(\log n)$ time. If no `TreeNode` corresponding to x has been created yet, then we know this memory address has never been accessed before, and can treat it as a new memory access. This algorithm for processing each memory access is shown in Figure 3-2. If an address has been accessed before, we calculate the reuse distance using subtree sizes and move that `TreeNode` to the end; if not, we create a new `TreeNode` and insert it at the end of the tree.

Figure 3-2 Example Program (Series-Parallel Tree in Figure 5-2)

```
1: function processMemoryAccess(tree, mapping, addr)
2:     if addr ∈ mapping then
3:         reusedistance ← tree.numAfter(mapping[addr])
4:         tree.remove(mapping[addr])
5:     else
6:         reusedistance ← None
7:         mapping[addr] ← createNode(addr)
8:     tree.insertLast(mapping[addr])
9:     return reusedistance
```

We walk down the tree to rebuild unbalanced subtrees whenever the tree is modified by either an insertion or a deletion. This keeps the runtime logarithmic instead of linear. We define the imbalance ratio r of subtree s as

$$r = \frac{\min(s.left.size, s.right.size)}{s.size},$$

and rebuild subtrees that have imbalance ratios below 0.3, excluding base cases. This is done by performing rotations of these subtrees while maintaining the correct order of the nodes, similar to AVL trees.

Instrumentation and CSI Hooks

This subsection introduces the CSI hooks used for the reuse distance calculator I developed and the data collection that happens. Besides the standard `csi_init` hook that gets called before program execution, where we initialize our data structures and set up reporting functions on program exit, the reuse distance calculator part of my tool only uses two CSI hooks: `csi_before_load` and `csi_after_store`. These hooks instrument memory accesses and pass to our tool the memory addresses queried, which we put through processMemoryAcesses from the algorithm shown in Figure 3-2.

As reuse distance data is calculated inside these hooks, the tool records the reuse distances for each memory access and maps them to the corresponding source code locations. The calculator can track all of the following performed by a specific location in the code, down to the line and column numbers:

- The number of accesses to new memory addresses.
- The total number of memory accesses.
- The list of the reuse distances of memory accesses, or any value that can be computed given this list, such as the total, average, or root mean square of the reuse distances of memory accesses.

As a proof-of-concept for the use of CSI to perform memory analysis, this prototype tool is written in a way such that it is easy to add or remove other information for the tool to track, associate with source code locations, and compute metrics with. The current metrics supported using this collected data and outputted by the tool are described more in Chapter 4.2.

3.2 Extension Ideas

In addition to the tool being easily extendable in terms of data collected online and output metrics, below are some ideas for further extensions:

- *Creating a version of the tool where analysis is performed offline, separate from program execution.* This could enable much more complicated analysis with an address trace, such as probability-based quantitative analysis of the nature studied by Gupta et al. [14].

- *Compute non-linear cost function of whole program reuse distance with step functions based on cache sizes for tool output.* The idea behind this extension is coming up with a single, one-value metric that the tool can output to give initial insight to program memory usage for any code.

- *Integrating the tool into an IDE.* This allows this profiling tool to be published and used by other programmers and researchers.

- *Generalizing the tool's usage to languages other than C.* An easy argument can be made as to why the insights generated by this tool can be useful for programmers for other languages as well; however, we would need to find alternative instrumentation to CSI.

Chapter 4

Metrics and Results

In this chapter, I describe the different metrics my reuse distance calculator measures. I also present numerical results collected by running the tool on the benchmark programs introduced in Chapter 2.

4.1 Metrics

This section breaks down the design of the reuse distance metrics into three parts: what the tool measures, how it aggregates these measurements, and what is actually reported during tool output. Each of these three parts is explained in more detail below.

Measurement

The reuse distance calculator I created is capable of measuring reuse distance with different granularities. This is the same idea as the simple example given in Section 1.3, where I gave hypothetical numbers for reuse distances both when measured for individual memory locations and based on cache lines. Specifically, reuse distance can be calculated on a:

- *Per memory address basis.* In other words, each specific memory address is its own unit, treated as distinct from all other memory addresses. A reuse distance

calculation is only triggered when the exact memory address is accessed again.

- *Per cache line basis.* When measuring in this way, memory addresses from the same cache line are treated as the same memory location. Assuming a cache line size of 64, this can be done by zero-ing out the last 6 bits of each memory address. Other cache line sizes can be modeled with the same approach using a different number of bits.

- *Any other neighborhood size* [14]. Just like the per cache line basis measurements, multiple memory addresses can be lumped into one group and treated as a singular memory location. However, the neighborhood size does not need to be a power of two, and may be anything useful for analysis.

Any of the above different measurement granularities can be applied to a variety of aggregation and reporting methods. These are presented in the following subsections.

Aggregation

Another point of flexibility in my reuse distance tool is the scope of aggregation. This can either be a span of code whose collected reuse distance data is aggregated and reported together, or be based on the location of a data access in memory relative to other data accesses. There are three main scopes of aggregation:

- *The entire program.* As it suggests, this scope is simply an aggregation across the entire program execution and the entire memory space. Example outputs include a sum of all the reuse distances incurred by the program, the average reuse distance of all data accesses, and the number of distinct memory locations accessed throughout the entire program.

- *A section of the source code.* This can be anything from a specific location in the code, such as `a[i][j]`, to the body of a specific function. The tool can also support anything in between, such as aggregating the reuse distance data associated with data accesses incurred by a specific line of code or by code wrapped in a specific loop.

- *A consecutive chunk of memory.* At the lowest level, this may be a single memory address or a cache-line. However, this is different from the measurement granularities introduced in the previous section: aggregating the reuse distance numbers collected for memory addresses in a cache-line is different from treating all those memory addresses as the same memory location, and thus collecting different reuse distance numbers.

  At higher levels, this may take the form of aggregating for memory locations found in the same cache block, or those belonging to the same allocation.

These aggregation methods may be used with any of the different measurement granularities described in the previous subsection. However, some combinations may not make sense – for example, treating memory addresses in the same cache line as the same memory location but aggregating reuse distance data for each individual memory address.

Reported Metrics

After aggregating the reuse distance data, the tool is capable of reporting a variety of different metrics. Since the purpose of the tool is to generate output that is insightful for each specific program use case, there is no single numerical metric it reports as a "one metric fits all". Instead, the tool reports a wide selection of metrics that the user may customize and draw insight from whichever seems most meaningful. An overview of the metrics the tool can report is as follows:

- *Aggregated reuse distance numbers.* The tool currently supports the total reuse distance summed across a scope of aggregation, along with the average or RMS reuse distance across the same scopes. The tool is also able to support the reporting of any metric that can be calculated using the list of reuse distances.

- *Number of memory accesses.* The reuse distance calculator is able to aggregate and report information such as the total number of memory accesses performed by the program, or the number of distinct memory locations accessed.

- *Information on cache hits and misses.* If provided with numbers on cache-sizing, the tool can estimate the number of hits and misses for each cache level based on the reuse distances, using the LRU stack model. Although this number is not necessarily accurate, it's a reasonable measurement and can provide important insight into program memory usage.

Now that we've described what the tool measures, how it's aggregated, and what is actually reported, the next section gives results acquired from running the tool on the benchmark programs introduced in Chapter 2.

4.2 Results

In this section, we present the results collected from running the reuse distance calculator from Chapter 3 on the benchmarks described in Chapter 2. For each of the target benchmark properties listed in Chapter 2, we created a program with two versions to demonstrate that the proposed code changes can improve runtime, and that this effect can be seen in the tool's reuse distance metrics. The following programs were created:

- For *reordering nested loops*, we created `matmul`. This is a matrix multiplication program, where the two versions have different loop orders. The pseudocode is shown and the analysis is explained in detail in the Section 1.3.1.

- For *reordering struct fields*, we created `music`. The most important part of the program is two versions of a struct, where the suboptimal version has alternating `int` and `int[]` fields, while the reordered version has all the `int` fields adjacent to each other and all the `int[]` fields adjacent to each other.

- For *reordering function calls*, we created `cash`. This program maintains two arrays of values, and isolated operations are performed on the two arrays. The suboptimal version of the program takes the form shown in Figure 2-1, where the two arrays are operated on in alternating order, and the reordered version of

the program takes the form shown in Figure 2-2, where the multiple operations on each array are performed adjacent in program flow.

- For showing that a *struct of arrays* can outperform an array of structs, we created `ppl`. This benchmark is developed using the template in Figure 2-5.

- For showing that an *array of structs* can outperform a struct of arrays, we created `ppl2`. This benchmark is developed using the template in Figure 2-6.

Table 4.1 lists the program average reuse distances measured for the two versions of each microbenchmark. These numbers are measured using a cache line granularity and aggregated over the entire program.

Benchmark	Average Reuse Distance (slow)	Average Reuse Distance (fast)	Runtime Speedup
`matmul`	12.01	5.54	91.1%
`music`	4.92	3.39	7.00%
`cash`	3.52	2.80	22.8%
`ppl`	4.53	3.34	37.7%
`ppl2`	3.16	0.56	32.21%

Table 4.1: Program average reuse distance measurements (in number of distinct memory locations between two accesses to the same memory location) for microbenchmarks.

Observe that the amount of speedup we achieve from making the adjustments proposed for each microbenchmark doesn't perfectly match the decrease in average reuse distance of a program. This is expected – program average reuse distance is not an end all be all metric, but it successfully shows that reuse distance does give insight into program memory locality, and we can conclude that it is a useful program property.

Notes

There are a few things to note about this prototype tool that results in the aggregated reuse distance metrics outputted not perfectly matching what we would come

up with calculating with pen and paper. According to CSI hooks, data accesses from some parts of the source code trigger multiple memory locations being accessed. One possible explanation is that the data may either be larger than one machine word, or aligned in a way such that it covers multiple machine words. From my preliminary investigations, the larger the problem size, the more memory locations are accessed this way, leading to larger average reuse distances. For example, in the slower implementation of matrix multiplication (pseudocode in Figure 1-1 and aggregated reuse distance metrics in Table 1.1), `b[k][j]` appears to trigger two memory accesses, leading to the reuse distance of memory addresses accessed by `c[i][j]` to be 3 instead of 2. On the other hand, in the faster implementation of matrix multiplication (pseudocode in Figure 1-2 and aggregated reuse distance metrics in Table 1.2), if we use a small matrix size such as $N = 5$, the average and RMS reuse distances of `a[i][k]` both drop to 2, as we would expect with our definition of reuse distance.

Additionally, some memory addresses are referenced to non-existent parts of the source code, e.g. line -1 or column -1. This is an issue related to the compilation to LLVM IR, since CSI directly extracts the source code locations from the LLVM IR. When this happens, a side effect is that aggregated reuse distance metrics can be slightly off for other parts of the source code.

Despite these caveats, the reuse distance calculator can give important insight into program memory locality.

Chapter 5

The Series-Parallel Tree

This chapter describes the series parallel tree that I use to capture the logical parallel structure of an instrumented program. The series-parallel tree is used to implement the prototype memory analysis tools I describe in Chapter 6. With the series-parallel tree structure, we can model the logical parallelism of codes, as opposed to only being able to analyze actualized parallelism. In the past, this property of series-parallel trees has been used to design provably good race detectors [13, 19]. For my thesis, I model and analyze the execution of parallel code with any number of processors p, even if the machine the code is running on only has 1 processor.

This chapter is structured as follows. Section 5.1 defines a series-parallel tree. Section 5.2 describes the information we record in this data structure, and Section 5.3 presents the CSI-specific algorithm used to create the tree. Section 5.4 introduces an alternative representation, the computation DAG, which may better illustrate program flow or be used in different analyses. Finally, Section 5.5 walks through a sample program and its series-parallel tree and computation DAG.

5.1 Specification

A series-parallel tree is a representation of program execution with computation nodes. In the implementation I used, adapted from the work of Kaler et al. [17, 15], there are three types of nodes:

1. **Serial nodes:** the children of serial nodes run in series.

2. **Parallel nodes:** the children of parallel nodes run in parallel.

3. **Data nodes:** these nodes have no children; instead, they represent sequences of computation and hold any data or information logged during this time.

In this structure, each serial and parallel node will have as children one or more data nodes, and a non-negative number of serial and parallel nodes. We show an example program with its corresponding series-parallel tree below, in Figures 5-1 and 5-2 respectively. In this program, a call to DOTPRODUCT will trigger two calls to MULTIPLY in parallel. Aside than this, computation executes serially.

Figure 5-1 Example Program (Series-Parallel Tree in Figure 5-2)

```
1: function MULTIPLY(x, y)
2:     return x × y
3:
4: function DOTPRODUCT(x[2], y[2])
5:     a ← cilk_spawn MULTIPLY(x[0], y[0])
6:     b ← MULTIPLY(x[1], y[1])
7:     cilk_sync
8:     return a + b
```

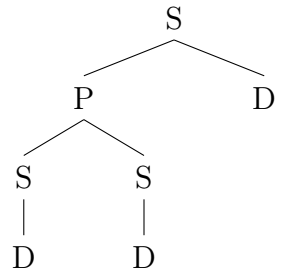

Figure 5-2: Example series-parallel tree of dot product (program in Figure 5-1).

The root serial node represents entrance to the DOTPRODUCT function. Its child parallel node splits into two serial nodes, each representing the two parallel MULTIPLY function calls, and its child data node represents the computation associated with the return statement. The serial nodes on the third level also each have their child data node, which holds any information logged during the execution of their corresponding lines of code.

5.2 Data Logged

In this section, I describe how data is logged to the data nodes during the creation of the tree. Each data node holds the following information about its corresponding sequence of computation:

- *A list of events that occurred.* These include data loads and stores, the allocation and freeing of memory, entering and exiting functions, etc.
- *The list of memory addresses the program loaded from or stored to.* We also track the corresponding CSI IDs for these loads and stores.
- *The amount of data movement with reuse distances greater than K.* This is recorded in bytes, and K can be set to estimate the amount of data moved between cache levels, etc.
- *The number of LLVM pseudo instructions executed.* These serve as timing measurements for the different segments of the computation.

In addition to the above, the implementation structure also allows for simple modifications and extensions. If one wanted to include the specific list of reuse distances incurred by the memory accesses, one could easily update the tool to include that. Some of this information is later accessed to compute memory bandwidth metrics used to further study the behavior of parallel programs, as we shall discuss in Chapter 6.

5.3 Algorithm

In this section, I describe the algorithm for building a series-parallel tree dynamically at runtime and describe its implementation in CSI. Specifically, we examine how the implementation invokes different CSI hooks, and when data logging occurs. An overview of the series-parallel tree implementation is as follows:

- At the start of the program, we initialize the series-parallel tree in the tool and create the root node, which is always a serial node.

- Serial and parallel nodes are opened, closed, or synced during the CSI hooks for Tapir control flow. Specifically, we do the following for each of the 6 hooks:
  - `csi_task`: this is called after entering a spawned task following a `cilk_spawn`. A serial node is opened.
  - `csi_task_exit`: the serial node is closed.
  - `csi_detach`: this is called within the spawn function helper, right before entering a spawned task following a `cilk_spawn`. A parallel node is opened.
  - `csi_detach_continue`: this is called before the continuation of code execution following a `cilk_spawn` and its `csi_detach`. A serial node is opened.
  - `csi_before_sync`: the serial node opened with `csi_detach_continue` is closed.
  - `csi_after_sync` the parallel nodes opened with `csi_detach` are synced.
- We invoke the CSI hooks `csi_after_load` and `csi_after_store` for logging information on memory accesses. For information regarding function calls, we use the CSI hooks `csi_func_entry` and `csi_func_exit`. We also instrumented the tool to log data to the nodes during hooks for memory allocation; specifically, during `csi_after_alloca`, `csi_after_allocfn`, and `csi_after_free`.

This summarizes the essence of our implementation of the series-parallel tree data structure. In Chapter 6, I introduce code that walks through the series-parallel tree and uses the data we collected to compute memory bandwidth metrics.

5.4 Computation DAG

This section describes the process to create a computation DAG, or series-parallel graph [37], based on the series-parallel tree. This alternative representation is helpful for visualization and understanding, especially since a series-parallel tree may end up large and complicated, with many nested levels of serial and parallel nodes.

Data nodes in the series-parallel tree become nodes in the computation DAG, while serial and parallel nodes simply dictate the edge relations between the nodes. Specifically, graph construction is as follows:

- *We create a source terminal s and sink terminal t.* Computation starts at the source, follows the directed edges, and ends at the sink.

- *For a serial node in the series-parallel tree, we string its children in series.* For example, given a single root serial node with child nodes a, b, c, we draw a directed edge from the $s \to a$, from $a \to b$, from $b \to c$, and from $c \to t$. This represents the fact that a must be computed before b, and b before c.

- *For a parallel node in the series-parallel tree, we put its children in parallel in the new graph.* For example, given a single root parallel node with child nodes a, b, c, we draw a directed edge from $s \to a, s \to b, s \to c$, and from $a \to t, b \to t, c \to t$. This represents the fact that the computations of a, b, and c can occur in any order, but all must complete for the program to terminate.

When creating this graph, any subtree can be abstracted with a node. The node can then be replaced with the graph corresponding to that subtree, where the source and sink terminals are collapsed. An example creation of this computation DAG will accompany the series-parallel tree example in the next section.

5.5 Example

This section walks through an example program and the series-parallel tree generated by my prototype tool. Compared to the dot product example from Section 5.1, this example involves a `cilk_for` loop and presents the actual series-parallel tree produced, rather than a theoretical diagram.

We introduce a program that performs a parallel linear scan of an array of size N. The program does so by reading through the array and computing the average of the values: specifically, it starts by initializing the array with random integers, then

scans through the array, reading every value it holds and computing the average of all values in the array. Parallelism is present in the loop that walks through the array to compute the average. The pseudocode is shown below in Figure 5-3. For $N = 4$, this program generates the series-parallel tree in Figure 5-4.

Figure 5-3 Linear Scan Program (Series-Parallel Tree in Figure 5-4)

```
 1: function getAverage(n, arr)
 2:     total ← 0
 3:     cilk for i < N
 4:         total ← total + arr[i]
 5:     end for
 6:     return total ÷ n
 7:
 8: function linearScan(n)
 9:     arr ← int[n]
10:     for each i < n do
11:         arr[i] ← random integer
12:     end for
13:     return getAverage(n, arr)
```

We examine and walk through the generated series-parallel tree. In this prototype tool implementation, the trees generated are unbalanced, as if instrumentation came before Tapir loops. We notice that there are 4 parallel nodes, one for each spawn from the `cilk_for`. Each parallel node spawns two serial nodes a and b, for the two tasks that program execution splits off into after a spawn respectively. The shorter subtree rooted in a corresponds to the spawned off iteration of the for loop, while the taller subtree rooted in b corresponds to the program execution that continues and ends up spawning more parallel tasks for future iterations of the loop. Each serial and parallel node has exactly one data node as children, with the exception of the root serial node – this node has two data nodes as children, one before the parallel node and one after the parallel node. This matches what we expect to see from the program pseudocode in Figure 5-3, since there are instructions executed and computation completed both before the first spawn and after it. On the other hand, we do expect the other serial and parallel nodes to have only one child data node, since program execution continues in one of the child nodes.

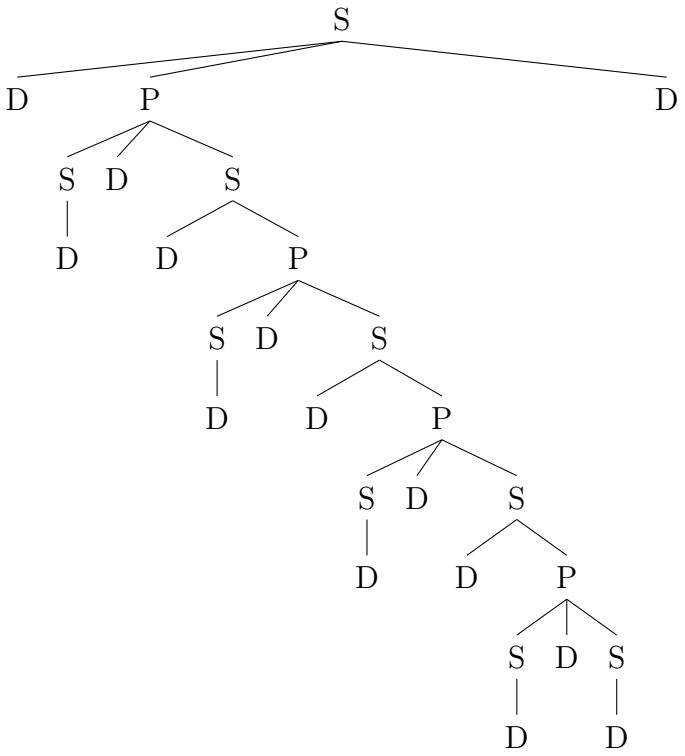

Figure 5-4: Series-parallel tree of linear scan program using array size $N = 4$ (program in Figure 5-3).

We now create the computation DAG for this series-parallel tree. We first label the data nodes with numbers, as shown in Figure 5-5. The constructed computation DAG from this tree, following the previously outlined steps, is shown in Figure 5-6.

The child nodes of a parallel node in the series-parallel tree always diverge from and converge to the same two nodes in the computation DAG. This follows directly from the definition of this graph that program execution flows one-way. In other words, a node cannot be executed until all its ancestors have been executed. Because of this, the graph can also be viewed as a topological sort of the computation nodes, and can thus be analyzed for many other calculations. For example, this is the typical representation used when analyzing the work and span of parallel programs.

Now that we've explained the series-parallel tree integrated into our tool, we discuss the application of memory bandwidth analysis in which we utilize it in the Chapter 6.

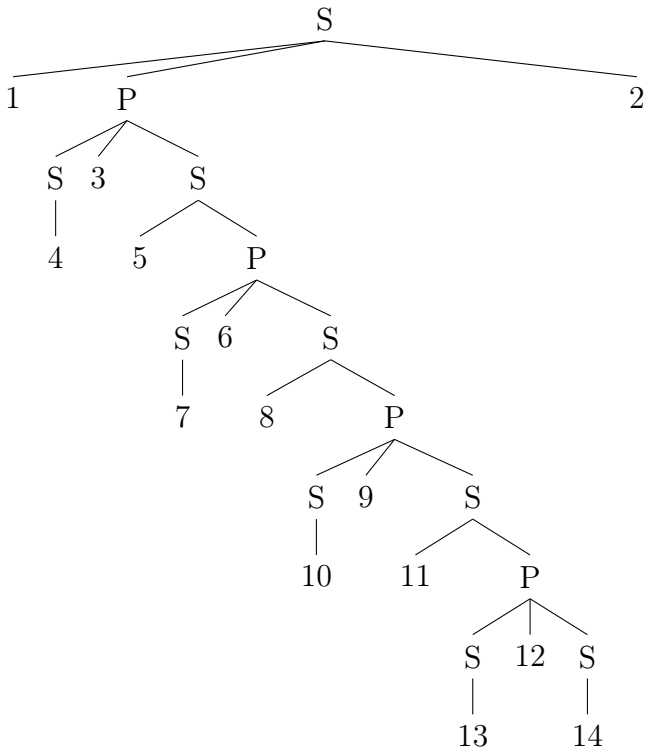

Figure 5-5: Series-parallel tree of linear scan program with numbered data nodes, using array size $N = 4$ (program in Figure 5-3, original series-parallel tree in Figure 5-4).

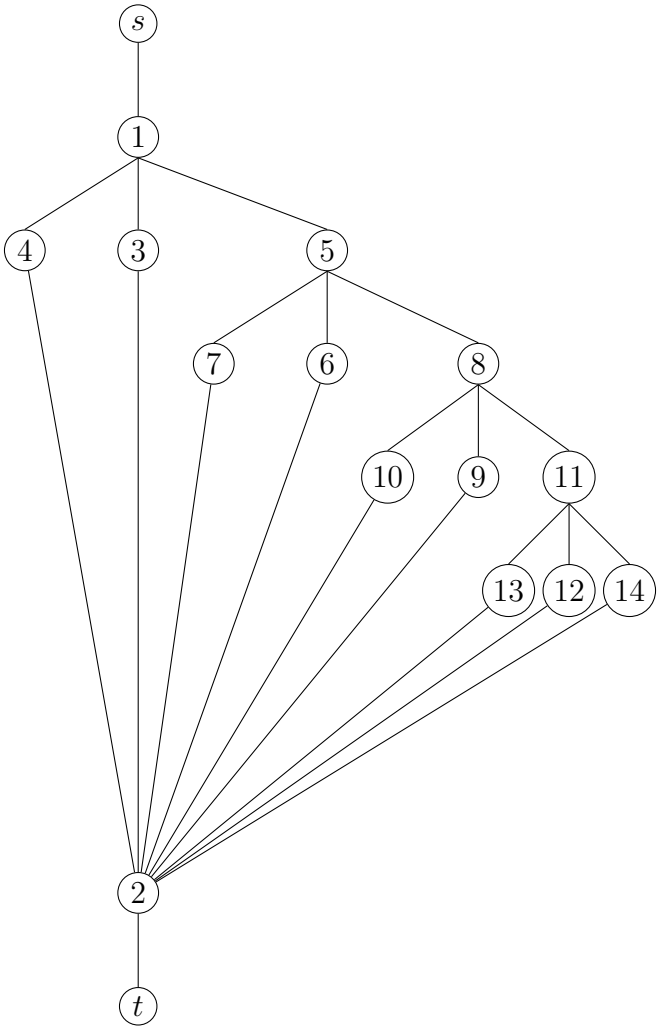

Figure 5-6: Computation DAG of linear scan program using array size $N = 4$ (program in Figure 5-3, original series-parallel tree in Figure 5-4, series-parallel tree with numbered data nodes in Figure 5-5).

Chapter 6

Parallel Memory Bandwidth Analysis

This chapter overviews the memory bandwidth analysis I performed in my study, including metrics I propose to analyze parallel program memory bandwidth, and how to compute them.

For hardware specifications, memory bandwidth is defined as the rate at which data can be read from or stored into a semiconductor memory by a processor [36], usually expressed in units of bytes per second. This is crucially related to the question of whether a program is memory-bound or computation-bound: if the memory bandwidth of a program measured for a specific execution exceeds the theoretical machine upper bound of memory bandwidth, then the program's execution becomes limited by this memory bandwidth, as opposed to lack of computation power.

Following this train of thought, the intuitive definition of actualized memory bandwidth we want to measure should also take on a form related to the rate bytes per second. Introduced in Section 1.5, given a reuse distance K, we can consider any memory access with reuse distance greater than K to result in data movement between different levels of cache. Thus, as memory accesses with reuse distance less than K can be considered to incur very low memory costs, we only measure the data movement, in bytes, of those accesses with reuse distance greater than K. However, program runtime tends to fluctuate with factors outside of the program's control. In searching for a more deterministic measurement of "runtime", we settle on the number of LLVM pseudo instruction executed, which is the compiler's internal cost measure

for program instructions, and correlates well with program runtime.

Thus, our chosen memory bandwidth metric is as follows:

$$\frac{\text{data movement}}{\text{time}} = \frac{\text{data movement}}{\text{\# of LLVM pseudo instructions executed}}.$$

We will be computing this metric for parallel programs, both the average memory bandwidth and the peak memory bandwidth.

6.1 Parallel Memory Bandwidth Calculation Pattern

All of the algorithms I explored follow the same pattern. The format taken is as the following (Figure 6-1):

Figure 6-1 Memory Bandwidth Calculation (Algorithm Pattern)

```
1: function memoryBandwidth(node)
2:     if node is a data node then
3:         base case
4:         return
5:     initialize metric values
6:     for each child in node.children do
7:         memoryBandwidth(child)
8:         if node is serial then
9:             update metric values for the serial case
10:        else if node is parallel then
11:            update metric values for the parallel case
12:    end for
```

Although the format is the same, how each type of node is treated is different across the different algorithms we propose for the different use cases. Each is described in more detail in the following sections.

6.2 Average Memory Bandwidth

This section covers algorithms to calculate the average memory bandwidth of a program, given its series-parallel tree recording the data movement and runtime in number of LLVM pseudo instructions in each part of the program execution.

Infinite Processors

In the theoretical case of having infinite processors, we can assume that all child computation nodes of a parallel node in the series-parallel tree run in parallel with each other. Thus, even though it could be short of realistic for large parallel programs, this measure if asymptotic and useful in helping us understand the program or algorithm's memory limits.

The aggregation of the data recorded in data nodes is as follows (Figure 6-2):

Figure 6-2 Average Memory Bandwidth Calculation (Infinite Processors)

```
 1: function AVERAGEMEMORYBANDWIDTH(node)
 2:     if node is a data node then
 3:         return
 4:     node.datamovement ← 0
 5:     node.time ← 0
 6:     for each child in node.children do
 7:         AVERAGEMEMORYBANDWIDTH(child)
 8:         node.datamovement ← node.datamovement + child.datamovement
 9:         if node is serial then
10:             node.time ← node.time + child.time
11:         else if node is parallel then
12:             node.time ← MAX(node.time, child.time)
13:     end for
```

In other words, we compute the total data movement of the program by summing up all data movement across the different parts of the program, and compute the total runtime in number of LLVM pseudo instructions executed by taking the maximum runtime of the program, given infinite processors. This is done by summing up the number of instructions in serial and taking the maximum of the number of instructions running in parallel. We can then divide the total data movement measured this way by the total runtime measured this way to get the average memory bandwidth as described by our metric.

To test this algorithm, I ran it on the series-parallel tree generated for the linear scan program with array size $N = 4$. Results on this example program are, of course, only useful for didactic purposes to illustrate the behavior of the memory bandwidth analysis algorithms. For such small loops with very little work per-iteration, the

benefits of parallel execution are marginal. To demonstrate the memory bandwidth calculations in this small example, we set the reuse distance threshold to $K = 0$. The data movement and time values calculated for each node is shown in Figure 6-3.

We can immediately see a few things that match our expectations:

- Among the children of the root serial node, the first data node is significantly larger than the last data node. This is what we expect to see since the array allocation and initialization occurs before the parallel part of the program, with very few lines of code after.

- On the same level, the amount of data scanned through is on the same order in that first data node and in the parallel node. This makes sense since we make one pass through the array for initialization, and one pass through the array when scanning through. There are minor differences from other parts of the code using other (non-array) variables.

- Under the four parallel nodes corresponding to the four `cilk_spawn`s, there is a serial node with bandwidth values 40/22. This corresponds to a processor performing the actual code within the parallel loop, and matches our expectations that all four use the same amount of data and cost the same number of instructions.

- As we expect, for small amounts of parallelism such as this example with array size $N = 4$, parallel computing is simply not worth the overhead – both the data movement and time measurements of the subtree rooted at the highest level parallel node is significantly higher than 4 times the measurements of the serial node representing the actual code inside the loop being executed.

Overall, the program's average memory bandwidth is $600/5217 = 0.115$ bytes of data moved per LLVM instruction executed. We compare this number to the measurements we obtain with the other memory bandwidth calculations in future sections.

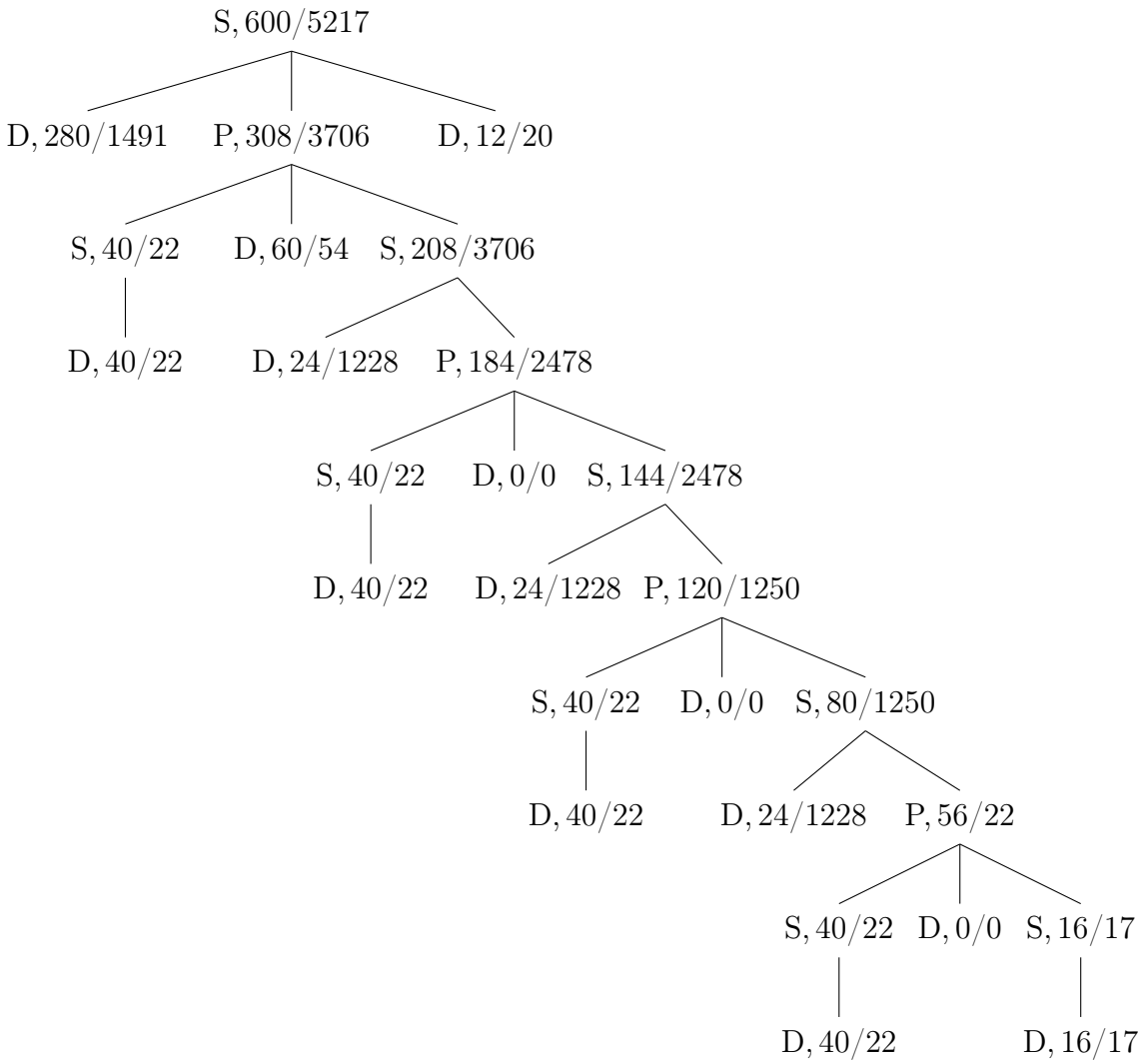

Figure 6-3: Series-parallel tree with average memory bandwidth calculations of linear scan program using array size $N = 4$ (program in Figure 5-3) and reuse distance threshold $K = 0$. Nodes are in the format of $[T, d, t]$, where T is the type of the node (serial, parallel, or data), d is the total data movement of the subtree rooted at that node in bytes, and t is the total time of the subtree rooted at that node in number of LLVM instructions executed.

p Processors

A measure that may have more realistic applications is the same metric, but measured for a specific number of processors p. My idea for this diverges from the general parallel memory bandwidth calculation pattern for the other three calculations presented in Section 6.1; different from this pattern, the implementation for this idea would take a much more complicated form.

I did not implement this as part of my thesis work, so the idea and proposed algorithm are further elaborated on in the conclusion (Chapter 7).

6.3 Peak Memory Bandwidth

Although the average memory bandwidth metric gives us good insight into the memory usage of a program, something more relevant to detecting whether or not a program is memory-bound is the measurement of peak memory bandwidth. As the name suggests, peak memory bandwidth is the maximum memory bandwidth, as a rate, during the entire program execution. If this rate ever exceeds the specified memory bandwidth the hardware provides, then our program is, at least partially, memory-bound.

Infinite Processors

If we assume infinite processors, then the calculation is straightforward – we analyze the computation DAG and perform simple aggregations, just like the average memory bandwidth algorithm for infinite processors.

However, different from that algorithm, we want to sum the numbers for parallel nodes, and take the maximum for the serial nodes. This makes sense since we are now measuring bandwidth as a single number, as opposed to breaking it down into data movement and time – this is because when calculating peak bandwidth, the other measurement that matters is the rate, as opposed to for average, where the amount of time spent moving data at a specific rate also matters. This algorithm is shown in Figure 6-4.

As for the average memory bandwidth algorithm assuming infinite processors, I ran the peak memory bandwidth algorithm assuming infinite processors on the series-parallel tree generated for the linear scan program with array size $N = 4$ and reuse distance threshold $K = 0$. The peak memory bandwidth calculated as a rate of bytes moved per LLVM instruction executed for each node is shown in Figure 6-5.

We can notice many similar things as the ones we did for Figure 6-3. However,

Figure 6-4 Peak Memory Bandwidth Calculation (Infinite Processors)

```
1: function PEAKMEMORYBANDWIDTH(node)
2:     if node is a data node then
3:         node.bandwidth ← node.datamovement ÷ node.time
4:         return
5:     node.bandwidth ← 0
6:     for each child in node.children do
7:         PEAKMEMORYBANDWIDTH(child)
8:         if node is serial then
9:             node.bandwidth ← MAX(node.bandwidth, child.bandwidth)
10:        else if node is parallel then
11:            node.bandwidth ← node.bandwidth + child.bandwidth
12:    end for
```

different from that approach, the peak bandwidth tree also lets us easily trace the parts of the program that incur high instantaneous memory bandwidth – specifically, for each serial node, it is obvious from the diagram which of its child nodes is contributing most to this bandwidth metric. In the same way, for each parallel node, we can look at the peak bandwidth calculations for its child nodes and easily determine if work was split equally in terms of memory usage.

Compared to the overall average memory bandwidth of 0.115 bytes per LLVM instruction, the peak bandwidth is a high 9.325 bytes per LLVM instruction for this program. However, all things considered, this makes sense – since the demonstrative problem size is small, more work and time is spent on initialization and overhead, as opposed to the memory-intensive sections, thus leading to a lower overall average memory bandwidth.

Although the average memory bandwidth metric gives better insight to a program's overall performance and limitations, which can guide asymptotic conclusions, the peak memory bandwidth metric is more useful in determining whether or not there is an instance during runtime where a program is actually memory-bound. In the next section, we examine calculations of peak memory bandwidth, but restricting computation to p processors.

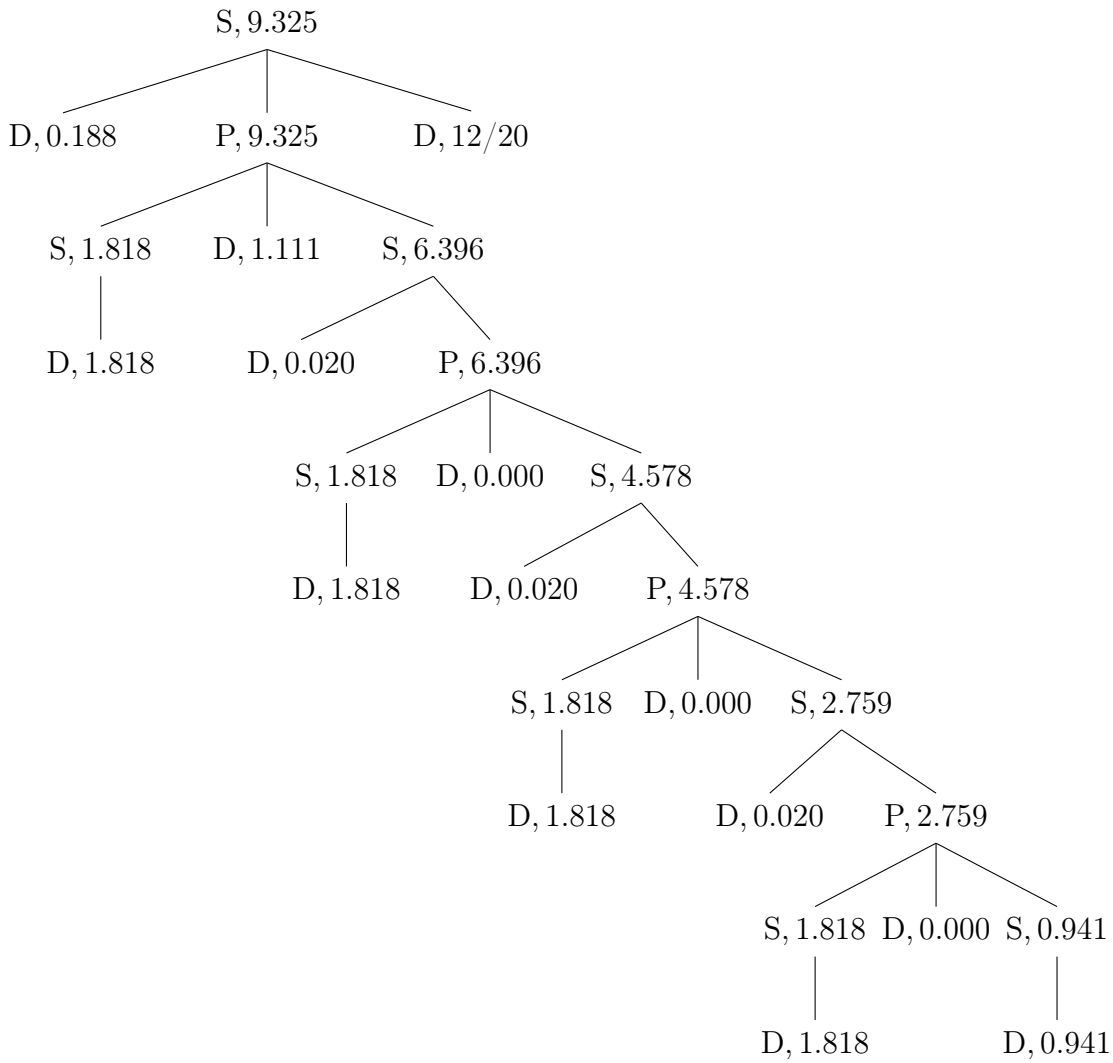

Figure 6-5: Series-parallel tree with peak memory bandwidth calculations of linear scan program using array size $N = 4$ (program in Figure 5-3), reuse distance threshold $K = 0$, and infinite processors. Nodes are in the format of $[T, b]$, where T is the type of the node (serial, parallel, or data), and b is the peak memory bandwidth of the subtree rooted at that node as a rate of bytes of data moved per LLVM instruction executed.

p Processors

Similar to the section on average memory bandwidth, a measure with more realistic applications is the same metric, but measured for a specific number of processors p. We are able to compute this for any reasonable p, because of the ability of the series-parallel tree design to model logical parallelism (Chapter 5).

To calculate the peak memory bandwidth for a program given a specific number

of processors, we can use dynamic programming to allocate processors to the children of parallel nodes.

In the computation DAG, since each numbered node corresponds to a data node, we can assign each node its memory bandwidth as a rate by dividing its data movement by its time in number of LLVM pseudo instructions executed. The peak memory bandwidth of the program is then the maximum rate we can obtain by summing across nodes in parallel, with the restriction that we can sum at most p nodes at once.

Note that if the number of processors available p is greater than or equal to the maximum number of nodes that can be running in parallel, then the peak memory bandwidth calculation is the same as the calculation for infinite processors.

For p less than the maximum number of nodes that can be running in parallel, the calculation is actually simpler with the original series-parallel tree. We maintain a table for each node in the computation DAG, which gives the computed peak memory bandwidth for any number of processors allocated to that node, up to p processors. For any node x, given the tables of its child nodes, we can compute the table for x using dynamic programming. The exact algorithm is shown below (Figure 6-6):

To paraphrase the main part of the algorithm, the peak bandwidth of a parallel node is the maximum sum of peak bandwidth of its child nodes, for some allocation of the available processors. This corresponds to the idea of summing across nodes that can be run in parallel in the computation DAG, and accounts for the nodes not directly in a subtree by maintaining a table that gives the peak bandwidth for any number of processors up to p.

As for the previous two algorithms, I ran the peak memory bandwidth algorithm for p processors on the series-parallel tree generated for the linear scan program with array size $N = 4$, using a range of values for p. The overall peak memory bandwidth computed is shown below in Table 6.1:

Even though the for loop only has 4 iterations, the maximum number of branches that can be running in parallel is 6; this is because each parallel node in the actualized series-parallel tree has 3 children. We can count this maximum number of parallel tasks in any of the series-parallel tree diagrams annotated with memory bandwidth

Figure 6-6 Peak Memory Bandwidth Calculation (p Processors)

```
1: function PEAKMEMORYBANDWIDTH(node, p)
2:     node.table ← DOUBLE[p + 1]
3:     if node is a data node then
4:         for each 0 < i ≤ p do
5:             node.table[i] ← node.datamovement ÷ node.time
6:         end for
7:         return
8:     for each child in node.children do
9:         PEAKMEMORYBANDWIDTH(child, p)
10:        if node is serial then
11:            for each 0 < i ≤ p do
12:                node.table[i] ← MAX(node.table[i], child.table[i])
13:            end for
14:        else if node is parallel then
15:            temp ← DOUBLE[p + 1]
16:            for each 0 ≤ i < p do
17:                for each 0 < j ≤ p − i do
18:                    temp[i + j] ← MAX(temp[i + j], node.table[i] + child.table[j])
19:                end for
20:            end for
21:            for each 0 < i ≤ p do
22:                node.table[i] ← MAX(node.table[i], temp[i])
23:            end for
24:    end for
```

p	Program Peak Memory Bandwidth
1	1.818
2	3.636
3	5.455
4	7.273
5	8.384
≥ 6	9.325

Table 6.1: Peak memory bandwidth calculations (in bytes of data moved per LLVM pseudo instruction executed) of linear scan program using array size $N = 4$ and p processors (program in Figure 5-3).

numbers, excluding data nodes that used 0 bytes of data and executed 0 LLVM instructions. Thus, for $p \geq 6$, the calculated numbers are the same as those calculated using the peak memory bandwidth algorithm assuming infinite processors, as we

would expect.

Using $p = 3$, the peak memory bandwidth calculated as a rate of bytes moved per LLVM instruction executed for each node is shown in Figure 6-7.

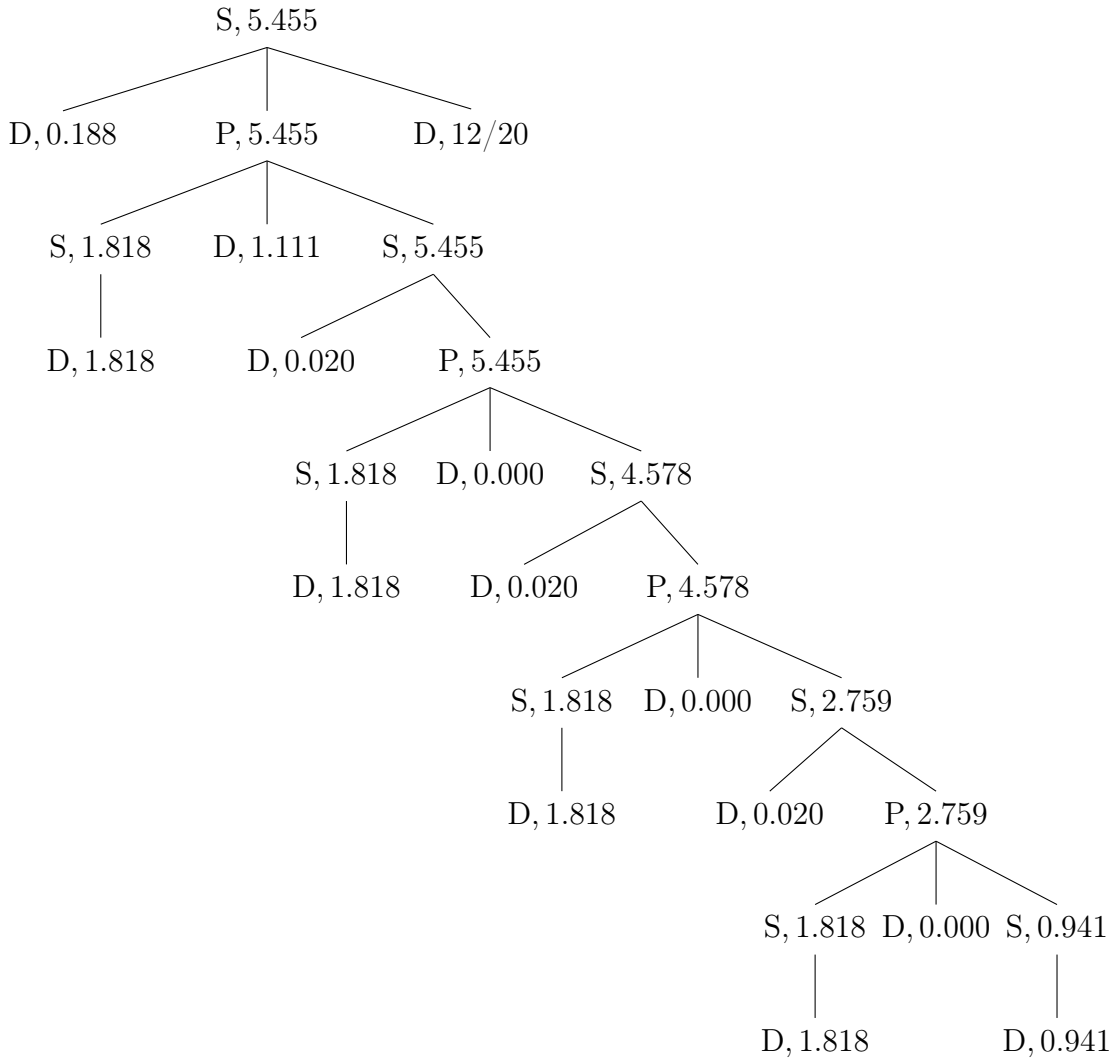

Figure 6-7: Series-parallel tree with peak memory bandwidth calculations of linear scan program using array size $N = 4$ (program in Figure 5-3), reuse distance threshold $K = 0$, and $p = 3$ processors. Nodes are in the format of $[T, b]$, where T is the type of the node (serial, parallel, or data), and b is the peak memory bandwidth of the subtree rooted at that node as a rate of bytes of data moved per LLVM instruction executed.

Observe that nodes whose subtrees use up to at most $p = 3$ processors have the same peak bandwidth as those calculated assuming infinite processors – this makes sense, since for those subtrees, $p = 3$ is equivalent to having infinite processors.

Similar to the diagram for infinite processors, we can also easily trace along this diagram and pinpoint which parts of the program execution are contributing to the overall peak bandwidth.

Chapter 7

Conclusion

Now that I've presented all parts of my thesis work, this chapter wraps up my thesis with discussions on extensions that I did not get a chance to work on and ideas that are not fully flushed out.

Section 3.2 lists a few extension ideas for the reuse distance calculator. If I were to work on this project for longer, it would be cool to implement some of those extensions and reanalyze the tool. In addition, I would want to evaluate the tool on industry standard benchmarks, and fully develop an algorithm for calculating the average memory bandwidth of a parallel program given p processors. These two ideas are discussed in further detail in the following sections.

7.1 Industry Benchmarks

Something I wanted to try but didn't get a chance to is running my reuse distance calculator and performing memory bandwidth analyses on a set of industry standard benchmarks. One set of benchmarks that could serve this purpose is the Rodinia Benchmark Suite.

The University of Virginia Rodinia Benchmark Suite is a collection of parallel programs which targets heterogeneous computing platforms with both multicore CPUs and GPUs [8, 9]. The Rodinia benchmarks cover "a wide range of parallel communication patterns, synchronization techniques and power consumption", and has

given insight on the growing importance of memory-bandwidth limitations and the consequent importance of data layout.

Example applications in this benchmark suite include back propagation, breadth-first search, and k-nearest neighbors; computations cover graph traversal, dense linear algebra, dynamic programming, etc. The domains that utilize such programs include medical imaging, fluid dynamics, physics simulation, image/video compression, and many more.

This benchmark suite is a good choice for my study of reuse distance and memory bandwidth analysis both because of the relevance of memory-bandwidth work related to it, and because of its coverage of parallel computing applications – it represents the broad range of scientific computing my research targets, ranging from physics simulations to artificial intelligence algorithms.

The version that I would want to use is the transpilation into an OpenCilk implementation by Tao B. Schardl.

7.2 Average Memory Bandwidth Given p Processors

I came up with a preliminary idea for computing the average memory bandwidth of a parallel program given a limited number of p processors. This idea is not fully flushed out yet, and takes on a different approach from the other memory bandwidth algorithms – notably, an implementation of this algorithm would not exhibit the simple parallel memory bandwidth calculation pattern described in Section 6.1.

Similar to the calculation of average memory bandwidth with infinite processors, we want to aggregate the total data movement and the total time in number of LLVM pseudo instructions. Total data movement is same as the infinite processors case, where we just sum the data movements collected for all data nodes; however, the calculation for time is more complex.

To paraphrase, we want to calculate T_p, the runtime of the program using p processors. Since our computation model follows the fork-join model [24, 10], we can allocate processors with the work-stealing scheduling strategy [7].

To do so, we can simply model the work-stealing scheduling strategy – maintain a queue of work items for each processor, where any parallel spawned tasks are placed on the queue of the same worker. Idle workers then actively try to steal work items from the queues of busy workers. This way, the computation order is maintained, while runtime is minimized heuristically.

Specifically, when a worker is working on a serial node, the children of the serial node are placed on the queue as one work item, maintaining the defined order of execution, since they must be executed in series. On the other hand, when a worker is working on a parallel node, the children of the parallel node are placed on the queue as separate work items at the same time, which other idle workers are free to steal. It is noted that the computation of that parallel node is not finished until the work on all child nodes has completed and returned. If a parallel node is a child of a serial node, then all children of the parallel node must be completed before the siblings that come after the parallel node maybe be computed.

Whether or not this algorithm works, or if it computes useful metrics, is untested.

Bibliography

[1] Walid Abu-Sufah, David Kuck, and Duncan Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *Computers, IEEE Transactions on*, C-30:341 – 356, 06 1981.

[2] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 298–302, 1998.

[3] George Almási, Cundefinedlin Caşcaval, and David A. Padua. Calculating stack distances efficiently. *SIGPLAN Not.*, 38(2 supplement):37–43, jun 2002.

[4] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011.

[5] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360. Citeseer, 2001.

[6] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, page 44–54, USA, 2009. IEEE Computer Society.

[9] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, 2010.

[10] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), page 139–146, New York, NY, USA, 1963. Association for Computing Machinery.

[11] NIST Big Data Public Working Group Definitions and Taxonomies Subgroup. Big data engineering (frameworks). In *NIST Big Data Interoperability Framework*, volume 1. National Institute of Standards and Technology, 2019.

[12] Mingdong Feng and Charles E Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[13] Jeremy T Fineman. *Provably good race detection that runs in parallel*. PhD thesis, Massachusetts Institute of Technology, 2005.

[14] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing*, 73(7):1011–1027, 2013. Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012.

[15] Tim Kaler. Sptree. `https://github.com/timkaler/SPTree`, 2021.

[16] Tim Kaler, William Kuszmaul, Tao B Schardl, and Daniele Vettorel. Cilkmem: Algorithms for analyzing the memory high-water mark of fork-join parallel programs. In *Symposium on Algorithmic Principles of Computer Systems*, pages 162–176. SIAM, 2020.

[17] Tim Kaler, Tao B Schardl, Brian Xie, Charles E Leiserson, Jie Chen, Aldo Pareja, and Georgios Kollias. Parad: A work-efficient parallel algorithm for reverse-mode automatic differentiation. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 144–158. SIAM, 2021.

[18] Tim Tim FS Kaler. *Programming technologies for engineering quality multicore software*. PhD thesis, Massachusetts Institute of Technology, 2020.

[19] Tushara C Karunaratna. *Nondeterminator-3: A provably good data-race detector that runs in parallel*. PhD thesis, Massachusetts Institute of Technology, 2005.

[20] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[21] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[22] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, jul 1996.

[23] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[24] Linus Nyman and Mikael Laakso. Notes on the history of fork and join. *IEEE Annals of the History of Computing*, 38(3):84–87, 2016.

[25] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

[26] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. The CSI framework for compiler-inserted program instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(2), dec 2017.

[27] Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. Brief announcement: Open cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 351–353, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 249–265, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Tao Benjamin Schardl. *Performance engineering of multicore software: Developing a science of fast code for the post-Moore era.* PhD thesis, Massachusetts Institute of Technology, 2016.

[30] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[31] Wesley Smith, Aidan Goldfarb, and Chen Ding. Beyond time complexity: Data movement complexity analysis for matrix multiplication. *arXiv preprint arXiv:2203.02536*, 2022.

[32] William Stallings. Computer memory system overview. In *Computer organization and architecture: designing for performance (8th ed.)*, Upper Saddle River, NJ, 2010. Pearson Prentice Hall.

[33] Rabin Andrew Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer architecture designs.* PhD thesis, University of Michigan, 1993.

[34] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[35] Wikipedia contributors. Hardware performance counter — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Hardware_performance_counter&oldid=1018317779`, 2021. [Online; accessed 24-November-2021].

[36] Wikipedia contributors. Memory bandwidth — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Memory_bandwidth&oldid=1018429919`, 2021. [Online; accessed 5-May-2022].

[37] Wikipedia contributors. Series–parallel graph — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Series%E2%80%93parallel_graph&oldid=1037819119`, 2021. [Online; accessed 13-May-2022].

[38] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, et al. The suif compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University Technical Report No. CSL-TR-94-620, 1994.

[39] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. A relational theory of locality. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–26, 2019.

[40] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 129–140, 2014.