UNIVERSITY OF CHICAGO

Persistent Data Models for Automatic Parallelization of Nontrivial Communication Patterns

by

Sean Laguna

A thesis submitted in partial fulfillment for the
degree of Master of Computer Science

in the

Ridgway Scott
Department of Computer Science

September 2016

UNIVERSITY OF CHICAGO

Abstract

Ridgway Scott
Department of Computer Science

Doctor of Philosophy

by Sean Laguna

Acknowledgements

Acknowledgements go here

Chapter 1

Introduction

Sometimes, data and their corresponding algorithms have a "trivially parallelizable" characteristic (e.g. simple "map" operations on arrays of values), where processors have, and write to, disjoint sets of data. But in some seemingly trivial cases, processors may have to read data produced by or shared with other processors and use that data as input to their computations; or, various operations may require different, confounding sets of data, requiring either data sharing across sets or a reconfiguration of those sets. For example, matrix multiplications require a row/column ordering of processors, but this does not minimize interprocessor boundaries as in an ideal domain decomposition. Furthermore, matching these sets to other overlaid structures for computation, such as meshes, presents further difficulties. Even providing a balanced decomposition of data onto processors can itself prove elusive and challenging, and exchange of data across the resulting boundaries may require intricate and complex handling.

1.1 Problem Description

Most software in the molecular/quantum mechanics domain, including molecular dynamics as well as implementations of spanning methods such as Hartree-Fock, semi-empirical, density functional theory, continuum models like force field models, and Monte Carlo, rely on legacy libraries to implement shared and distributed memory parallelism [19] [20] [21]. These libraries almost exclusively use OpenMP and MPI for these respective tasks, sometimes even resorting to fine-grained and manual shared resource management such as locks. In addition, almost all such software uses BLAS/LAPACK and is written in FORTRAN, C, or C++ [1] [9]. Recently, implementations using Python's SciPy package have surfaced, often in the form of high-level and expressive wrappers that organize calls to low-level, optimized code [10]. GPU implementations of crucial SIMD operations have been done manually and almost exclusively in CUDA for Nvidia GPUs and OpenCL for AMD GPUs [5]. More current technologies for distributed memory management in the context of concurrency and parallelism have not integrated into the most popular software, for historical but practical reasons: issues of compatibility,

maintainability, scalability, reliability, complexity, efficiency, and more have prevented an elegant transition [6]. In specialized settings, implementations of such algorithms have emerged [22] [13]. But, this work often lacks comparisons to traditional programs in terms of performance metrics and remains specialized and disconnected from practice and end users. Furthermore, these implementations often do not take advantage of the touted feature-sets of their languages or environments, such as algorithmic immutability (realized as persistency in Clojure), lazy evaluation, a functional programming perspective of state, and abstraction. Books, manuals, and tutorials will recommend using an exposed low-level interface to achieve higher performance, defeating the purpose of using a higher-level language or environment. Haskell suggests writing code in its internal representation to avoid situations where its compiler cannot appropriately optimize code [12].

In order to mitigate these problems, to bridge the gap between performance and ease of programming, I strive to develop a model of concurrency control and parallelism that can

1. Allow programmers to implement algorithms in a way that faithfully describes the task at hand, without the model distracting from the clarity of that algorithm or requiring the programmer to redesign the algorithms to focus on implementation details;

2. Allow programmers to leverage parallelism without requiring invasive changes to the algorithms that they parallelize, even when starting from optimized serial implementations that already exist;

3. Operate efficiently and compete with the current state-of-the-art.

When new theoretical improvements emerge in high performance computing algorithms, programmers should not feel limited by the tools they have in implementing, testing, benchmarking, and analyzing those algorithms in real-world situations. Scientists should have tools that let them express their ideas clearly and readily while also taking advantage of modern hardware and software.

As a field, computer science still has a long way to go before the development and existence of a "holy grail" solution where any program can immediately be run in parallel without significant overhead. But, some itemized problems can be addressed. Data races have emerged as a problem for both shared and distributed memory parallelism in settings where multiple cores write and read the same data, necessitating synchronization. By leveraging ideas from parallel programming, namely persistent and transient data structures, data races can be avoided a priori, removing the requirement of manually synchronizing the aforementioned reads and writes. I will describe and sketch the implementation of a parallel container (that can be realized as a vector, hash map, b-tree, or any other such data structure) in C++ with the following characteristics:

1. Automoatic synchronization of reads and writes in shared memory without data races, and a model for how the same can be done in a distributed memory system;
2. Usable as a drop-in replacement for a C++ standard library vector for a restricted set of member functions and inner types and operations on those types;
3. Asymptotically identical runtimes for the operations when used in serial and the corresponding expected improvements when used in parallel;
4. An interface for arbitrarily partitioning the vector across processors;
5. An interface for extending the set of inner types that can be used;
6. A model for generalizing the set of inner types and operations on those types used.

I will also provide the implementation of the following actual parallel algorithms using an implementation of this container to provide examples of how it will work, and I will show how they correspond to current implementations that use manual concurrency control and parallelism techniques:

1. A "foreach" operation that takes the vector and applies one operation to all items in the vector (either a unary operation or a binary operation with another supplied input), resulting in an output vector of the same size;
2. A "reduce" operation that takes the vector and applies one operation to all items in the vector, resulting in a final output value;
3. A finite differences scheme that uses the vector to store the values of the heat equation on a 2D mesh.

Then, I will discuss the method's effectiveness given different inputs and conditions. Finally, I will discuss future directions for this method of automating parallelism and its viability in incrementially solving the problem of automated parallelism in the truly general case.

Chapter 2

Background

2.1 Motivating Automated Parallelism Given the State of Modern Computational Tools

Moore's law dictates that the number of computational components per unit space (such as an integrated circuit) doubles per unit time. These components cannot necessarily be used to accelerate the rate of sequential computation, and instead, the decrease in component size has been utilized by developing multicore hardware. Further spatial limitations prevent shared memory arrangements for large numbers of processors greater than around about 64 for standard CPUs and about 1000 for new "manycore" processors [3]. This results in the need for distributed memory arrangements that rely on high-latency communication between memory that cannot be written to directly by any one processor. These distributed memory units are connected through a network that uses one or more of many protocols like Ethernet or Infiniband. These protocols vary in their attention to different factors, such as reliability, speed, ease of use, and level of abstraction.

Distributed memory hardware configurations, referred to as clusters, involve the arrangement of many computers, referred to as nodes, that have their memory interconnected by some communication protocol. They may contain over 1 million nodes [25]. Due to the plethora of nodes, computation time within individual nodes decreases relative to the communication time required to send data between nodes. Because of this new bottleneck, developing communication-mitigating algorithms has become a popular area of interest. Some work involves modifying parallel algorithms such that the resulting algorithm requires less communication [2]. Other work involves "pipelining" algorithms such that communication and computation can overlap in time - the program dedicates some threads within a node to performing communication, while other threads carry out any computation that does not rely on unreceived data [7]. More generally, methods that reduce the need for synchronization between nodes can improve parallel CPU utilization.

Using algebraic and combinatorial code analysis, optimization, and generation techniques, methods like polyhedral optimization and Decoupled Software Pipelining (DSWP) have achieved automation of pipelining (and more generally the parallelization of loops with data dependencies between iterations) [8] [11]. The development and use of these methods stems from an implicit yet nonessential sequential ordering in the implementation of the serial version of the algorithms upon which these methods operate. These nonessential orderings derive from the procedural characteristics of the languages used to implement the algorithms in question.

2.2 Programming Models and Implicit Execution Order

Programming languages emerged from abstract models of computation like lambda calculus and Turing machines. Different ways of executing the computations prescribed by these models gives way to different programming models and thus, different ways of thinking about algorithm design. Naturally, these models have different implicit assumptions, different flexibilities, and different implications with regard to execution order. The fallout presents different conditions and avenues for parallelism between two primary programming paradigms, procedural and functional.

2.2.1 Procedural

Procedural programming has the propensity to overdetermine the behavior of a program in such a way that renders it hard to parallelize. Namely, by guaranteeing that each line of a program executes after the previous one has completed, lines which don't explicitly depend on each other become unnecessarily ordered. This guarantee does have upsides though, including the fact that a programmer can easily and naturally encode execution dependencies: lines which do in fact depend on previous lines will execute correctly, and lines that depend on lines that appear afterward will fail, usually through a compile-time or run-time error. But, in a parallel setting, a compiler, interpreter, or scheduler will struggle to identify the full set of lines which can execute in parallel with respect to each other. Depending on the language, different difficulties arise; in languages with pointers or references, these often come in identifying side effects of impure functions. In C++, for example, pointer aliasing creates lots of havoc during any sort of code analysis, because it allows the mutation of a single piece of data from many contexts.

2.2.2 Functional

In functional programming, a different implicit execution ordering exists. Execution ordering derives from the parameterization and structure of function calls, which imply the set of dependencies that each function has. Namely, a fucution depends on any values that are passed to it. Any expressions passed as arguments must be evaluated

before the function accepting those arguments is evaluated, or else the function cannot use those values during its own computation. In truth, the function may not use a value that it takes as an argument, so this style of implicit execution ordering also relies on the programmer to properly specify the parameters to a given function. If parameters are overspecified or specified in some inefficient way, then the compiler, interpreter, or scheduler will likely assume that more dependencies exist than in reality; such an over-specification hides opportunities for parallelism. On the other hand, underspecification takes on a simpler form in functional programming than in procedural programming, and manifests simply as undeclared or unaccessible variables in the current scope. Effectively, functional programming has a more rigid notion of dependency and scope; this inflexibility, while arguably semantically limiting in some ways, results in a more predictable and potentially less error-prone model.

2.2.3 Comparing Optimization

The process of optimizing code in both procedural and functional languages takes very different forms due to the differences above. Both languages optimize by trying to find more computationally efficient yet equivalent forms of expressions (or statements) present in the code, and by removing redundant or unnecessary computations. But, procedural languages, due to their imprecise dependency specifications, are sometimes optimized by reordering the evaluation of certain expressions. Such optimizations make no sense in a functional world, since no explicit ordering exists except for dependency-based orderings, which cannot be changed without changing the meaning and possibly the outcome of the code. Optimizing functional programming languages often takes the form of deciding exactly when to evaluate blocks of code that have no dependence relation. For example, consider a function $f(a, b)$ – f takes a and b as parameters, so depends on both; but, at the point in time where f is called, a does not depend on b nor vice-versa. In order to avoid performing unnecessary computation, many functional languages will try to determine where within f (if at all) both a and b are used in order to determine which should be computed first to maximize efficiency. Functional languages that have lazy evaluation, such as Haskell, will actually by default store these expressions as "thunks" that abide by "call-by-need" behavior: at runtime, thunks will evaluate only when needed, or directly used in some computation. But, even Haskell's compiler, ghc, will try to look ahead to see if it can guarantee that it will actually use the thunk's output. Eager languages often perfrom the opposite optimization, looking into functions to see if they actually don't need or will unlikely need the value, or if storing an expression as a thunk will increase performance due to future partial function applications or reuses of the function's value in different contexts. In pure functional languages without mutability, functions always evaluate to the same value so long as they have identical arguments, making common subexpression elimination easier in certain cases and aiding in the elimination of redundant computation.

2.3 A History of Concurrency Control Constructs

Preparing code for parallel execution shares similarities with optimizing code for serial execution and critically hinges on the determination of execution ordering and data dependencies. Filling in these blanks, due to the rationale provided above, differs dramatically between procedural and functional languages.

2.3.1 Proceural Concurrency Control

Procedural languages focus on managing data, or more generally, state: no two processors should collide while reading or writing a piece of data. Semaphores emerged as a fundamental means of providing this control [4], behaving as blocking counters that allow only so many threads to move beyond "lift" calls simultaneously before other threads that have already done so move beyond the symmetric "lower" calls on that semaphore. This incredibly flexible solution proved complicated: the lift $P(S)$ for a sempahore S calls could occur anywhere, and so could the lower $V(S)$ calls. With an arbitary number of threads in a language such as C executing arbitrary code and accessing arbitrary data, numerous complicated situations can arise, all various forms of resource mismanagement and sources of errors or inefficiency: deadlock, livelock, starvation, priority inversion, busy waiting, and others. Semaphores, originally intended to provide a mechanism for controlling the scheduling of computation assigned to different threads relative to each other, and serving as a means of mutual exclusion, required very careful use to provide this. Race conditions arise whenever a thread writes a piece of data while another thread simultaneously tries to either read or write it; semaphores contain no indication of a thread's behavior nor precisely which data they govern. The correctness and efficiency, and thus overall effectiveness, of their use depends entirely upon the programmer.

Advancements came in the form of specializing control mechanisms for different tasks. Mutexes better embodied the concept of mutual exclusion, requiring the thread that locks (the corresponding "lift" operation) a mutex to also unlock ("lower") it. While more restrictive than semaphores, mutexes address a use case where, truly, only one thread should be accessing some data at any given point. Mutexes specialized further, gaining read-locks and read-write locks, which allow for data to be read simultaneously by any number of processors but written by only one (and ensuring that, while writing, no other processors can read). But, mutexes fall prey to many of the same issues that semaphores do, including the slew of deadlock variants.

Further parallelism control techniques came. One general category involves atomic operations, which stemmed originally from hardware architectures and their corresponding instruction sets. Read-modify-write instructions such as test-and-set and compare-and-swap allow for robust implementations of the above control mechanisms, as well as more complex and fine-tuned non-blocking (lock-free and wait-free) algorithms, usually implemented in the general case of some data structure like a queue. These algorithms

have only recently begun to have practical performance characteristics when compared to blocking algorithms [14]. Data parallelism stems from identifying data structures that can be operated on in an inherently mutually exclusive way, and while they achieve a nearly ideal level of parallelism, they have restrictive use cases.

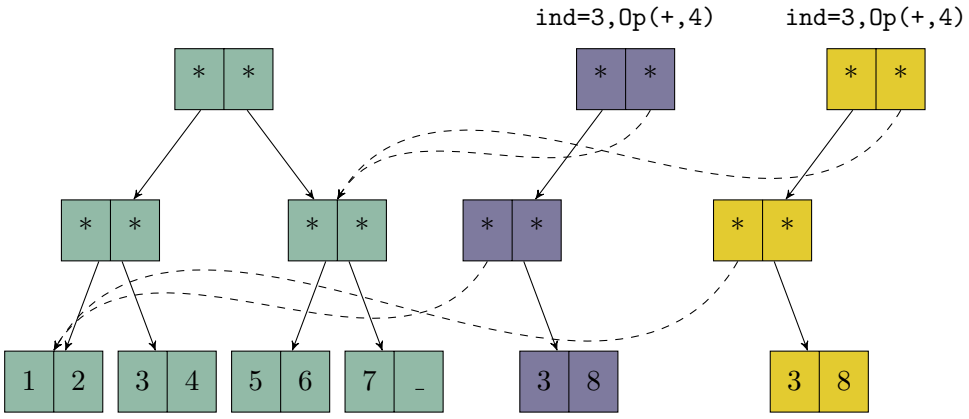
2.3.2 Functional Concurrency Control

In pure functional languages, any function calls with no relative dependencies can safely execute in parallel, because they may not access the same data. Functional programming languages use computation, not state, as a focal point: because they lack mutable state, they also lack the ability to write a value while simultaneously reading or writing it from another process. Lacking mutable state yields correctness at the cost of efficiency. The programmer must now think critically about how to avoid overspecifying function parameters such that two functions that could run theoretically at the same time actually may. Furthermore, runtime information can complicate this process: two threads that want to modify elements of an array may not actually contend for the same data, but the programmer, compiler, interpreter, or scheduler may not have the capacity to identify this or react to this at runtime in a way that results better performance. Ulimately, a purely functional language must either have a very conservative but very safe means of parallelizing a program, or a more aggresive but less safe means that may result in either confusing results or retroactive inefficiencies.

Transactional memory serves as an example of the latter case: blocks of code are marked as "atomic", and code executes in parallel with no explicit concurrency control. If code within an atomic block interleaves with other code in a harmful way, then the block (called a "transaction" in this context) restarts from the beginning, discarding all modifications made since. Clojure has an advanced STM system that allows the user to specify certain equivalences, allowing the STM system to complete even if the prescribed ordering does not match the execution ordering exactly. A user can specify that a function application commutes in time with itself – for example, a user can specify that an increment operation on a counter commutes, so that any number of increments, in any order, performed in transactions on that particular value will not cause transactions to restart.. Further flexibility comes in the form of an `ensure` operation that Clojure uses to require that a read value has not been mutated by that point. Clojure uses the `alter` and `ref-set` operations as its primary read-write and write operations within transactions. Finally, Clojure uses "agents" to perform asynchronous operations on values without guaranteeing when other threads will see those writes. Agents have their own set of functions that provide flexibility: `await` blocks waiting for an agent, `set-validator` associates some kind of check with the mutation of agents, `add-watch` registers observers to trigger when an agent mutates, and `deref` and `send` allow for read and read/write operations to interact properly with those possibilities.

Clojure has a smorgasbord of other concurrency control mechanisms, ranging from "reducers" (which both amount to generalized, parallel SIMD operations) to "transducers" to "futures" and more.

2.4 Leveraging Immutability for Parallelism



Clojure serves as a good example for how immutability can assist in automated parallelism. It provides persistent data structures that appear immutable from any reference, but can also accrue updates from those references, updates that eventually propagate to those references. It does this without making entire copies of those data structures, and instead by using pointers to catalog to portions of the data structure that have not changed from any reference. An excellent informal description of how persistent vectors work can be found at Niklas D'orange's website [17], and a detailed description of an optimized persistent vector can be found in the ICPF 2015 conference proceedings [23]. Clojure's persistent vector uses a b-trie for its internal representation. The values in the vector are stored in leaves. If some reference modifies the vector, it identifies the leaf of the trie that should contain the modified value, creates a new version of that leaf with the updated value, and creates a new root (and hence, a new trie) with the structure that results in the modified vector without having to duplicate anything except the other values in the modified leaf. Sometimes new leaves (and the corresponding internal nodes and branches to that leaf) must be created as well, and sometimes empty leaves and internal nodes and branches must be pruned. Clojure uses 32 children per node, and achieves $O(\log_{32} n)$ (what they consider effectively $O(1)$) time for `append`, `update`, `lookup`, and `subvec`, as a mutable vector does. Even with this generous reading ($\log_{32} n \leq c$ for some constant upper-bound c), the persistent version does incur a constant overhead involved with creating this representation of the vector and with performing the maintenance of this trie. Clojure uses numerous tricks to optimize this data structure. First, it uses bit partitioning, where the trie branches based on 5 possible bit values (generating 32 branches per node). By doing this, Clojure provides a means

for looking up elements in a vector tied to its internal representation. Looking at 5 bits at a time, the algorithm for lookup walks through the internal nodes of the trie until it reaches a leaf, where the value is stored. It furthermore uses a tail node, pointed to directly by the root, to optimize modifications near the end of the vector.

2.4.1 Transient Vectors: Relaxing Immutability When Helpful

Transient versions of persistent vectors modify values directly. Persistent vectors can be copy-constructed to transient vectors, and transient vectors that have been modified are invalidated such that using them after modification results in undefined behavior. Nodes and leaves are given IDs that correspond to the reference that created them, and the transient vector can safely mutate these. Finally, transient vectors can be copy-constructed back to persistent vectors, where that reference behaves as any other persistent vector would, invalidating the transient from which it was copied. By doing this, a stream of unobserved mutations can take place without sacrificing the guise of immutability. D'orange's masters thesis discusses optimizations of Relaxed Radix Balanced Trees (RRB-Trees) through transience [16].

2.4.2 Persistency to Automate Concurrency Control without Race Conditions

In a parallel application, processors can read and write to shared data. Before performing a read of some data, a processor may want an updated version of that data, or a version that corresponds to a certain point in a global ordering of events, even if it stores that data in a persistent data structure. It may hypothesize or expect that the data has changed and specifically want an updated value. When this happens, it has to verify that the other process has completed its operations. Consider that each process converts an initial persistent vector into a transient vector, where each process's vector gets its own ID. Then, that process's transient vector sees its own modifications transiently, in real time, but sees a persistent, immutable copy of the values other processes may have modified. With the knowledge of both the values a process does not own and the values it needs to read, any given process can safely obtain the most updated version of a value: by reading the values it owns as it wishes, and by waiting for other processes only when necessary. If multiple processes write to the same locations in the vector, the situation becomes more complicated, and the values must somehow be reconciled.

Chapter 3

Theoretical Groundwork for Automated Parallel Methods

3.1 Motivation: Parallelism Necessitates Concurrency Control

Consider that in the general case of a program intended for parallel execution, the potential arises for two processors to perform computations that conflict with each other. Specifically, when two processors access the same memory location and at least one of them performs a write at that location, a conflict arises, because both the code does not prescribe a global ordering of those accesses and the order changes the outcome. Hence, this conflict results in a program state that could potentially produce an incorrect result. Conflicts occur when parallel programs cannot be mapped to unique sequential programs due to their asynchronous properties – at least one of the orders in which the instructions could be executed, in combination with the program state, yields an invalid program state at a later point in time.

In the most general case, it may prove challenging or impossible to prevent or resolve these conflicts without relying on synchronization mechanisms that prevent full utilization of parallel execution. But, if certain constraints are applied to

1. the operations that the processors will execute, and
2. the read/write access patterns upon the data with which those operations will interact,

the task becomes feasible. This in hand with a clever organization of memory will yield automated parallelism methods.

3.2 Groundwork for Minimal Concurency Control Mechanisms

The following will assist in the description of concurrency control schemes that come afterward.

3.2.1 Terms, Definitions, and Assumptions

We shall specify constraints for operations that processors may execute in parallel in such a way that conflicts can be detected and eventually resolved. These constraints give way to relaxations of the execution ordering of those operations.

Each relaxation specifies a set of operations $\mathbf{Op}_i$ that contains all functions that it allows. We assume that the operations have the form $\mathbf{Op} : \mathbb{R}^n \mapsto \mathbb{R}$. If an operation effectively produces many values, we model it as a series of operations that take the same inputs. The constraints will apply to the set $\mathbf{Op}$. The constraints may involve restricting the domain to a subset of the general case above. In a practical setting, the function's range will be the values allowed by a given numeric data type, such as `uintXX`, `intXX`, or `floatXX`, where `XX` $\in 8, 16, 32, 64$. We will refer to the inputs and output of a function as *values*, and in code, we will give them the general type `val`.

Then, we will provide a function C_i which will identify a conflict, and a function R_i that will provide a way to resolve conflicts that arise.

Finally, we will define the set $St_{i,t}$ as the relevant subset of the program state after t operations. Here, relevant means that this program state works in isolation with regard to all other operations taking place within the program: no outside operations are reading or writing values in St; furthermore, operations happening within St do not read or write values outside of St either. Sometimes the program state is undefined for a certain t, but we can place limitations on when this is acceptable [15]. Unless an operation produces no output, it modifies the program state via its output.

3.2.2 Caveats of parallelism

For a task to incur speedup through parallelization, no overhead acquired through managing parallelism must exceed the space or time complexity of the serial version of the program. Call $\eta(\vec{x})$ the overhead of a given parallel execution $f_p(\vec{x})$, with corresponding serial execution $f_s(\vec{x})$. We then want

$$\eta(\vec{x}) \in o(f_s(\vec{x}))$$

in space and time. For example, a parallel algorithm for summing a set of values should take $\Theta(\log(n))$ time assuming no overhead. An optimal serial algorithm takes $\Theta(n)$

time – no faster algorithm can exist. The overhead to the parallel algorithm should take asymptotically $o(n)$ time or the parallelization amounts to no asymptotic improvement.

Consider that any variable in a program has a set of dependencies. A dependency is another variable used to compute the first's value at some point during the computation. Adjacent dependencies appear immediately in these computations, and those may have their own dependencies as well. Construct a dependency graph G for a variable in a subroutine. The graph begins as a single node, just the variable itself; call it v_0. Iterate through the execution paths of the program until the variable receives an assignment of some value. If the value is constant and does not come from the evaluation of an expression, do not modify the graph. If the value comes from the result of an expression, take each variable inside the expression and add them to the graph, drawing a directed edge from each of them to v_0. Continue doing this until v_0 shares an edge with every variable contained in an expression assigned to v_0. Then, repeat the process for each of these newfound dependencies. If the assignment to v_0 results from a function call, then this process must be repeated with the return value of the function in question, which itself serves as the sole direct dependency of v_0 from the call. Note that the graph may have cycles. In fact, very simple subroutines have cycles. For example, incrementing a counter has one node and an edge connected to itself – a loop. Now, take any paths for which each node in the path has only a single incoming and outgoing edge and compress them into a single node. These paths result from inherently serial computations. The number of intermediate variables used to represent these computations is incidental and nonessential to the graph's meaning. Such subgraphs cannot be parallelized, as each operation depends exactly on the preceding one. As an example, computing $\sin(\log(x))$ requires performing $\log(x)$ and then computing the sin of the result. The logarithm must be computed first. Having two processors at hand cannot speed up this computation.

The longest path that doesn't repeat a node serves as a lower bound for the runtime of the subroutine as each edge has at least $O(1)$ time complexity. A tighter lower bound would come from determining the complexity of the number of times each cycle in the graph would be performed and to identify the longest path with repeated edges, bounding the number of times a node is incorported to the unique cycles that contain it. Then, each of those cycles would contribute its complexity to the lower bound of the complexity of the runtime of the overall subroutine.

Consider a partition of the edges of the compressed graph by its connected subgraphs. These subgraphs can be executed independently, yielding parallelism. Now consider removing an arbitrary node n from the graph. If the number of subgraphs increases due to the removal of n, then the newly-generated subgraphs correspond to sections of the subroutine that are computationally independent. Without loss of generality, say there are 2 such newly-generated subgraphs P and Q. There are three cases for how these subgraphs P and Q interact:

1. All edges $e \in E(P)$ incident to n are **incoming** to n, and all edges $e \in E(Q)$ indicent to n are **outgoing** from n: n connects P to Q, and Q can only be computed once P has been.

2. All edges $e \in E(P)$ incident to n are **incoming** to n, and all edges $e \in E(Q)$ indicent to n are **incoming** to n: P and Q can be computed in parallel, but the value of n can only be determineded once both P and Q have been computed.

3. All edges $e \in E(P)$ incident to n are **outgoing** from n, and all edges $e \in E(Q)$ indicent to n are **outgoing** from n: P and Q can be computed in parallel, but only once the value of n has been determined.

In the latter two cases, opportunities for parallelism exist, and n identifies a point where concurrency control must take place to ensure the correctness of the program. In (2), n identifies the point at which synchronization between the two parallel regions P and Q must occur: n must wait for both to finish before issuing the expression to which it corresponds, because it depends on the result of those independent computations. In (3), n identifies the point at which parallelism can begin.

Ultimately, this partition results from cuts of the graph along these edges. In (1), two such cuts are required to separate the subgraphs from n. The cut indicates that n has a read dependency on P – it must read the value that subgraph produces at the node opposite the cut edge in order to perform its own computation. Furthermore, Q has a read dependency on n, and cannot begin its execution until the value of n has been determined. In (2) and (3), only one cut is required, as there are no dependencies between P and Q themselves, and the cut can separate both P and Q into the same region apart from n.

At some point, execution of each subgraph must take place. It can either take place eagerly, as soon as the program begins running, lazily, exactly when some node depends on it and only it, or somewhere in-between, such as at some position in an arbitrary or reasoned execution ordering given by the programmer via language semantics, compiler, interpreter, or runtime. If a task computes eagerly and before necessary, it may take up processor time better used elsewhere, causing other, more critical procesesses to wait for it. If a task computes lazily and inopportunely late, other tasks that may depend on it may have to wait as well. Schedulers cannot always account for these dependencies, resulting in idle processors and synchronization overhead.

3.2.3 Commutative and Associative Operations Allow Dependency Relaxation

These graphs may prove too rigid when trying to achieve maximal levels of parallelism in a program. Some apparent dependencies in the graph, as constructed above, may not actutally hold with regard to program correctness. Consider, for example, a case where a processor performs multiple additions on some value, and that all other addends of those

additions are determined by mutually independent subgraphs. The graph produced above may look like a stick with one additional incoming edge at each node; each node in this subgraph then has two incoming edges and one outgoing edge, because the additional incoming edge at each node connects other independent subgraphs. But, the order in which these additions take place does not affect the final outcome. The following explains a more general method for identifying these situations and allowing more parallelism to take place in light of them.

We can limit our set of operations to commutative, associative reductions. This is such that:

$$\mathbf{Op}_0 = \{f\} \tag{3.1}$$
$$f : \mathbb{R}^2 \mapsto \mathbb{R} \tag{3.2}$$
$$f(x, y) = f(y, x) \tag{3.3}$$
$$f(x, f(y, z)) = f(f(x, y), z) \tag{3.4}$$

The result is that if you have a set of values V and would like to apply f onto V repeatedly, you can do so by selecting arbitrary $x \in V$, and performing $V = \{f(x, y) \cup V \setminus \{x, y\}\}$. This implies that our desired start and end program states are:

$$St_{i,0} = \{V\}$$
$$St_{i,T} = \{v_{\text{ans}}\}$$

where v_{ans} is the single value resulting from performing the reduction of V as described above.

In pseudocode, each task might appear as follows:

```
/* preconditions:
    - reduce is only called if size(V) > 1
    - remove() and add() are atomic upon the set */
void reduce(set<val> &V) {
    val x = V.remove();
    val y = V.remove();
    ans = f(x, y);
    V.add(ans);
}
```

Notice that the operations on V must be atomic because the function mutates it. Thus, `reduce` has side effects, and the operations upon it must preserve atomicity lest two processors, e.g., try to remove the same element. Similarly, we want to avoid adding an element to a set in such a way that disrupts other processors' actions upon that set.

The implementation of this reduction on a shared or distributed memory computer remains unspecified; a shared memory implementation will be provided in section 4.1.5. One can envision more or less efficient versions with different pros and cons. For example, using other concurrency control techniques, contention can be reduced. A natural direction, described in more detail in 4.1.5, involves giving processors ownership over

mutually exclusive subsets of values in V, having them perform reduce operations on only the values they own, and then using the above strategy to arrive at the final value only when each processor has exactly one value left. Essentially, delegation of values does not have to happen through the atomic removal of arbitrary values from the set, though doing it this way allows us to describe an arbitrarily ordered method for performing such reductions without any extra machinery or complexity. Conveniently, it also has the side effect of load-balancing the parallel computation, because each processor stays busy until `size(V) == 1`, or one step away from the end of the subroutine.

One might ask, why not design the reduction as follows:

```
/* preconditions:
    - reduce is only called if !isempty(V)
    - remove() is atomic upon the set */
void reduce(set<val> &V, val &ans) {
    val x = V.remove();
    ans = f(ans, x);
}
```

The answer is simple: the entire second line of the function must be atomic, because if another processor modifies `ans` after it is accessed and before it is assigned, the intermittent update to `ans` has been erased. Because the entire second line must be atomic, no parallelism occurs. Implementing the code in this manner results in no parallel speedup. The executions of f must be able to occur on multiple processors simultaneously: this is the definition of parallelism, after all. But, leveraging persistent and transient structures will allow us to achieve code along these lines without needing such atomicity or synchronization while still yielding the correct result.

3.2.4 Persistence and Mitigating Synchronization

A persistent data structure is a data structure that appears immutable from all references. That means that any data accessed through a particular variable stays the same, no matter when you access it and no matter what other operations you supposedly perform on that data structure, i.e., any functions you call that involve that data structure. Object-oriented languages would frame the data structure as a class with member functions; others would frame the data structure as a composite type with corresponding functions that accept it as an argument and mutate it. Data strutures in purely functional languages would operate like persistent data structures but with additional strictness: the data structure does not only appear immutable from all references, it *is* immutable from all references, and the proxy for mutation involves producing a new data structure that corresponds to how the original one would look after the desired mutation.

Define a transient data structure as a data structure that behaves as the above one in all cases except through two documented functions: `resolve` and `deregister`. The first, `resolve`, updates the data structure to correspond to the most recent version of it,

incorporating changes across all processes. This will take some amount of time. Next, `eregister` tells the underlying data structure that the elements referred to need not stay accessible any longer. It allows the data structure to clean up old values that no longer have any use for the program. In a persistent setting, the same mechanism takes place when the variable falls out of scope. But, sometimes, optimizations can be made that would be cumbersome using only this construct. Highly-nested code blocks that delineate the lifetime of variables should not serve as the only means of writing efficient programs using persistent data structures, and furthermore, an efficient program may require the resolution of a transient data structure after versions of it have already fallen out of scope, require the extension of the lifetime of those other versions.

In this work, we develop CT (Contended Transient) data structures that add reconcilability to simultaneous modification. When two processes modify a variable at the same logical time, then necessarily, neither process can see the desired final value of that variable without more work. In a lock-based setting, the extra work would involve two locks and two unlocks. In a persistent setting, the extra work would involve the creation of two extra variables and one extra addition. Those variable creations can be pre-allocated in settings where conflicts are likely, resulting in an overhead of just one extra addition. But in the case of a very small number of additions, the benefits of parallelization don't manifest – one processor can handle such sequences of operations best, negating any synchronization costs.

Then, if other distributed processors need the data, an immediate or queued message can be sent to them by some prioritization. But, if many additions are performed, or if complex sets of operations are performed, a method for parallelizing the tasks and reconciling the conflicts that arise, one optimized for minimal overhead, becomes beneficial. Precisely speaking, parallelism becomes beneficial once

$$\eta + t_{\text{comm}} \leq t_{\text{ser}},$$

where as before, η describes the runtime overhead of synchronization.

3.3 Contention Resolution with CT Data Structures

3.3.1 Tools for Shared Memory

Shared memory models allow many processors access to the same memory locations. The latency of memory accesses are approximately the same among these processors; simultaneous reads can be performed harmlessly. But, simultaneous writes, or interleaved reads and writes, can lead to situations where different processors perceive a conceptual value differently. In the case of the aforementioned relaxation, problems arise when the list that contains the values loses or gains elements, and certain processors tasked with performing the reduce operation on certain values access the wrong values because they

intend to remove certain values from the list but instead remove others. This case arises when other processors have already removed those elements. Contention only arises when processors use information affected by other processors: If there are N values to reduce and P processors, then as long as $P \leq N/2$, then each processor can perform at least one operation for which no contention exists. Each processor has two values that it can reduce independent of all other processors. Otherwise, any processor must use data that another processor has produced, or may try to use itself, and barring algorithmic strategies designed to handle contention (i.e. synchronization), processors will interfere with each other. Even under such strategies, processors may need to wait idly for the algorithm to allow them to continue safely or may end up doing unnecessary work to arrive at the correct answer.

From the perspective of organizing all necessary reduce operations, different strategies exist, and the traditional notion of contention that arises in this laissez-faire understanding of shared-memory parallelism accepts many categorically different strategies.

3.3.2 Preemptive Contention Resolution

Preemption can resolve the contention issues above. For example, consider that each processor has an ID and can compute the set of indices in the original list that it can be said to *own*, such that any computation involving values that it owns. In the case where $2P \leq N$, then we can say each processor owns values e_i for $P \leq i < P + \lceil \frac{N}{P} \rceil$, which allows each processor to own a contiguous block of values that it can perform the reduce operation independent of the other processors. When $2P > N$, then we can say that for two processors p_i and p_j where $i < j$, if a reduction must be done upon a pair of values owned, respectively, by each processor, p_i will perform the reduction. At this point, each processor owns at most one value, so this strategy incurs no wasted cycles due to idle processors. Note that processors with larger IDs may now spend their time on other tasks, such as other computations without interprocessor dependencies, or in a distributed memory setting, the asynchronous communication of already-computed values on which other processors depend.

Preemption incurs an overhead: each processor must know its ID, and as soon as $2P > N/2$, a comparison must execute to determine which processor will handle the following reductions. Ultimately, the overhead here is constant in the number of values to reduce and logarithmic in the number of processors ($O(\log_2 P)$ extra comparisons). Furthermore, each processor in this final stage must receive a signal that it can safely access the corresponding value it did not itself compute: it must either block until the computing processor releases the final value (using, for example, a mutex) or receive an explicit message (such as through message passing, an actor, or a future) alerting it that it can use the value. Essentially, an indeterminate amount of time might elapse before this happens, approximately upper-bounded by the serial runtime of the program including the overhead incurred through the synchronization infrastructure in place (unless bugs produce deadlock, which would cause the program to hang indefinitely).

3.3.3 Postemptive Contention Resolution

Postemption accounts for and corrects contention issues after they occur. Postemption requires a formal understanding of the difference between expected and observed results. A particular reduction corresponds to a detection function that returns a boolean identifying whether contention resulted in an incorrect interleaving of parallel operations. Then, a resolution function can attempt to produce the correct value. Some resolution functions can themselves yield answers identified as incorrect by the detection function, and some could deterministically produce the correct result.

Say we have two processors p and q. Processor p has a dependency on value d_q to be computed by q for its subgraph G_p, and processor q has a dependency d_p to be computed by p for its subgraph G_q.

In a postemptive scheme, if q has always computed d_q before p accesses it, then there could be potentially no concurrency control overhead. This precludes any verification of the shared values – it does not provide a mechanism for determining if the value p receives has, in fact, derived from the expected computation by q. This leaves the desire for a scheme that performs this verification with minimal overhead. Let's call this overhead ν_{post}. In the case where this value may depend on the uncertainty inherent in parallel computation, we can discuss the expected value of this term, $\mathbf{E}[\nu_{\text{post}}]$.

Say that the values d_p, d_q are stored in a CT vector v at indices i_p and i_q respectively. Treat the vector transiently when updating values that do not serve as interprocessor dependencies (i.e., not d_p and d_q), but otherwise, perform persistent updates to its values. When it performs a persistent update, it runs a routine that addresses the possible synchronization problems. Consider the case of p updating d_p and that d_q depends on d_p. When d_p is updated, its routine performs the following tasks:

1. Update q's reference to v;
2. Run C on d_q;
3. If C detects a conflict, run R on d_q.

At the point where C is run, d_q may have been updated using f, or not: p cannot know because no synchronization has taken place. C will determine if this is the case. C has access to: (1) the version of v that q used when it updated d_q, and hence that it used to access d_p, and (2) the fresh version of d_p. It looks to see if $v[i_p]$ agrees with the fresh d_p. If it does, it knows that the resolution is unnecessary. If it does not agree, then it performs R on d_q, using the defined resolution function R for the function q ran to update d_q. This resolution function R is knowable because not only do we know the possibly erroneous, stale value $v[i_p]$ and the correct, fresh value d_p, we also know the function that q used to update d_q; by design, this function must have a resolvable nature. If it is associative and commutative, then R could be automatically generated from f itself.

q must inform the version of v it uses when updating d_q. If q uses the fresh version of v such that $v[i_p] == d_p$, q will not have informed q by the time that q checks dependent values for conflicts No conflict exists, though, because by assumption, $v[i_p] == d_p$. But, there exists the possibility that it informs p of this before p updates q's reference to v but after d_p checks to see if any of its dependencies have used q. Due to this case, if p has not seen C fail to detect a conflict, it cannot consider the dependent d_q resolved, and must wait until q informs it of an update for which C fails to detect a conflict. q informs p of updates by adding a pointer to its copy of v and the indices of v it used to a queue that records the computations that took place on processors other than p that depend on d_p. p pulls these records off its queue in the order they are added, until C fails to detect a conflict. Then the queue can be cleared, and the computations that depend on this particular update to d_p have all been resolved. Just because the queue is empty does not mean that all such conflicts have been resolved.

In a more complicated case, q may have multiple values in G_q that depend on d_p. In this case, p performs the above sequence on each value, using the queue described above. They can be labeled $d_q; i$ for $i \in 0, \ldots, n$. As soon as $C(d_q; i)$ fails to detect a conflict, $C(d_q; j)\ \forall\ j > i$ will also detect no conflict; each $d_q; i$ uses the same v, so once v is fresh, it stays fresh for all subsequent uses. Due to this, all subsequent resolutions are unnecessary, too: the update to d_p took place before each update to $d_q; j$, and thus, any following updates in q used the fresh value of d_p.

Another complication arises when d_q depends on multiple values in p, say $d_{p,i}\ \forall\ i \in 0, \ldots, n$. Then, each resolution will run sequentially and independently on d_q by p, in the order that the values d_p, i are updated by p. The programmer should have p compute the values in the same order that q uses them to minimize the expected number of necessary resolutions, though this is not necessary for a correct program.

A suspiciously complicated case arises when three processors exist, and d_q depends on values computed by two processors, r and s. Both processors may attempt to resolve d_q at the same time. If one updates and then the other does, the updates may fall prey to the same contention problems normally observed sans synchronization. But, recall that performing the resolution is itself an update and thus will be be treated persistently. Therefore, each processor will create its own version of the CT vector it must resolve; inform the other about its value; carry out conflict detection; and, at least one will detect a conflict, namely, the one whose reference was freshened before it ran its C. Then, at least the one processor that detected a conflict will arrive at the correct value. If both values freshen before both conflict detections take place, then both will detect a conflict, and both will produce a new vector. The processor that happens to freshen last will have its version of v used when procuring d_q, but both will have the correct values, making it unnecessary to ensure the use of one or the other.

In fact, in general, there may be n processors p_i for $1 \leq i \leq n$ that all update the same value d_q. We can prove that the correct value will be produced once all resolutions have taken place using induction on the number of processors that d_q depends on, n. If $n = 0$,

there are no dependencies. If $n = 1$ or $n = 2$, the resulting scenarios are as described above. We now assume that the scenario such that $n = k$ produces the correct result and check the case $n = k + 1$. There will always be one processor p_j that freshens the other processors' references first. This processor will not have to resolve its update: it performs the first update. Then, it will run its routine on all other processors p_i such that $i \neq j$. In the worst case, each of these processors did not receive the freshened value, and they will receive an update on their new references. Each of these new references now have $n = 2$ values to potentially resolve, the resolution performed by p_i and the update they themselves perform, corresponding to one of the base cases. These base cases resolve, yielding $n = k$ more updates on the final value, corresponding to the inductive step, which we have assumed to function properly. Note that while this case always results in the correct final value, it derives from poorly-structured parallel code. If we are performing updates to a single value on multiple processors, those updates must happen sequentially at some point or another, which is ultimately what will happen given the resolutions. Restructuring the code to have less contention on this one value could allow for more effective parallelism. But, ultimately, one update will occur correctly, and resolution of just that update occurs independently on each processor's unique updated version of the persistent vector, preventing an increase in the number of updates that must happen.

This system requires no concurrency control outside of triggering subroutines when values are updated. But, it does require supplying or generating a resolution function, which depends on the operation performed. Ideally, a symbolic math library could determine this resolution function automatically if supplied with the symbolic expression involving the interprocessor dependencies. Some trivial but inefficient versions of postemptive contention resolution exist. For example, consider the conflict detection function for reducting a set of values defined as summing the values in serial and comparing the parallel and serial answers. If the parallel answer differs from the serial answer, then take the serial answer instead. This method defeats the purpose of parallelizing the reduce operation and has no value. More fine-grained detection and resolution functions are required for a useful method.

Imagine a slightly different model for performing a reduce than in the preemptive case, where the model works the same until $2P > N$. Imagine that preallocated vectors of size $2P$, P, $P/2$, $P/4$, ..., 1 exist and are initialized to some non-numeric value $\varnothing$ (such that $\varnothing \cdot x = \varnothing$ for any operator $\cdot$ and any value x) at compile-time. Then, when a processor computes its reduce, it puts its answer into the vector half the size the values it produced, in the index of the smaller of the two values it reduced. When a reduce happens such that $R(a, b) == \varnothing$, then we know that it produced an incorrect answer: one value was not ready when the processor computed the reduce, and that value must be that produced by the processor not performing this current computation. This serves as the detection function. The resolution function takes advantage of the fact that the reduce operation has both transitivity and associativity: processors with IDs greater than $P/2$ can reduce any values that were computed too late, and a final reduce can occur between the final computed value and the resolved values. This algorithm

results in a space time tradeoff: it uses $\log_2(P)$ space, and if the larger-ID processors finish before the smaller-ID processors, no time penalty occurs (unless those larger-ID processors could have otherwise been scheduled for other tasks – these don't affect the runtime of the reduce operation, but do affect the runtime of other tasks that the overall program might want to perform).

3.3.4 Programmatically determining when to parallelize

In a simple case, say we have n numbers to operate on. In serial, this requires $n-1$ operations. In parallel, if we have p processors, each processor performs $n/p-1$ operations. Then, $p/2$ processors must send their result to the other $p/2$, resulting in $p/2$ communications. then, $p/2$ more operatioins must be performed, to sum the $p/2$ numbers. Then, $p/4$ more communication must be done, with $p/4$ more operations, etc. until $p/i \leq 1$. So, we have

$$t_{\text{oper}}\left(\frac{n}{p-1}\right) + (t_{\text{comm}} + t_{\text{oper}}) \log p \leq t_{\text{oper}}\,(n-1)$$

The analysis holds for any commutative and associative operation. Technically, more than $\log p$ communications and additions are done ($n-1$ to be exact), but many are done simultaneously, and there are $\log p$ total sequential steps. Still, this analysis is cursory. Namely there are some statistical properties of these times that are very relevant; not all processors finish their work at the same time, and in delineated blocks of operations, the slowest process or process group (depending on the algorithm) serves as the limiting factor.

let's say $t_{\text{comm}} = t_{\text{oper}}$. then we have

$$\begin{aligned}
\frac{n}{p} - 1 + 2\log p &\leq n-1 \\
n + 2p\log p &\leq pn \\
n(1-p) + 2p\log p &\leq 0, \text{or} \\
n(p-1) &\geq 2p\log p \\
n &\geq \left\lfloor \left(\frac{2p}{p-1}\right)\log p \right\rfloor.
\end{aligned}$$

Hence we know the cutoff for how many numbers we must sum in order to benefit from parallelization using a given number of processors. In general, TODO

$$\begin{aligned}
an + (a+b)\,p\log p &\leq apn \\
an(1-p) + (a+b)p\log p &\leq 0, \text{or} \\
an(p-1) &\geq (a+b)p\log p \\
n &\geq \left\lfloor \left(1+\frac{b}{a}\right)\left(\frac{p}{p-1}\right)\log p \right\rfloor.
\end{aligned}$$

In general, the minimum acceptable n grows logarithmically with p but linearly with b/a.

3.4 Parallelizing a Finite Difference Method

Let's consider a domain decomposition of a numerically solved partial differential equation (PDE). Let's assume that each processor has $c = O(1)$ terms to communicate between processors. This may be a common case: in a 1D spatial domain, simple decomposition methods do in fact yield $c = O(1)$) terms to communicate. Let's say we have a 4-point stencil where value $u_x^{t+1} = a(u_{x+1}^t + u_{x-1}^t) + b(u_x^t)$. This produces a backward difference method in time. When $a = r$ and $b = (1 - 2r)$ for $r = \kappa/h^2$, this corresponds to the simplest finite difference approximation for the Heat equation. When performing this update on a 1D domain, values can be updated for time $t + 1$ independently, except for the dependencies that occur at processor boundaries. In these cases, the communication of a single value must occur before updates take place.

Let's assume that we split our domain evenly along the spatial axis. Say we have a domain such that

$$x \in 0, X - 1$$

and that we have P processors in shared memory, and each processor is labeled

$$p \in 0, P - 1$$

Processor p has a chunk of of the domain with values

$$\begin{aligned} x_p &= \{\alpha * p, \alpha * (p + 1) - 1\}, \text{ where} \\ \alpha &= X/P \end{aligned}$$

Given already computed values for $u_x^t \ \forall \ x \in X$, the values for $u_x^{t+1} \ \forall \ x \in X$ can be computed in a data parallel manner. But what dependencies do individual computations at time $t + 1$ have on values at time t? The formula provided above makes this quite apparent:

$$u_x^{t+1} \text{ depends on } \{u_{x-1}^t, u_x^t, u_{x+1}^t\}.$$

If $x - 1$, x, and $x + 1$ are contained within x_p, then these values can be accessed directly from local memory. $x \in x_p$ by definition. But, if $x = \alpha * p$ or $x = \alpha * (p + 1)$, then communication across processor boundaries must take place, requiring communication. Communication implies synchronization, and in order to ensure that the value has been computed, either processor $p - 1$ or processor $p + 1$ must tell processor p that it has

performed that computation. If processors exist in separate memory spaces, the value must actually be sent from one memory space to another. Without loss of generality, let's say processor $p-1$ lags behind in performing its computation indefinitely. What can processor p do while it waits? It can compute values for

$$\begin{aligned} u_{x_{t1}}^{t+1} \text{ for } x_{t1} &\in \{\alpha * p + 1, \dots \alpha * (p+1)\}, \\ u_{x_{t2}}^{t+2} \text{ for } x_{t2} &\in \{\alpha * p + 2, \dots \alpha * (p+1)\}, \\ &\dots \\ u_{x_{t\alpha}}^{t+p} \text{ for } x_{t\alpha} &\in \emptyset. \end{aligned}$$

After α steps through time, processor p must wait for processor $p-1$ to finish computing its data. Likewise, waiting indefinitely for both processor $p-1$ and $p+1$ to complete only allows for $\alpha/2$ steps through time to be completed. Thus, we can say that

$$\begin{aligned} u_{\alpha*p+i-1}^{t+i} \text{ depends on } p-1, \text{ and} \\ u_{\alpha*(p+1)-1-(i-1)}^{t+i} \text{ depends on } p+1, \\ \text{for } i \in \{1, \alpha\}. \end{aligned}$$

Each processor has α values to compute per timestep, yielding α^2 values to compute in α timesteps. In this range, Processor p has $\alpha^2/4$ values that do not have dependencies on other processors, $\alpha^2/2$ values that depend on one adjacent processor, and $p/4$ values that depend on both adjacent processors. Every time a processor receives a value from another processor, up to α new values can be computed by p (in the case that only one adjacent processor's values are needed, and $\alpha/2$ in the case that both are).

Thus it is in the programmer's best interest to perform communication at points in time such that each processor always has work to do: either its own work sans dependencies, or work that relies on values from other processors. Preemptively waiting for other processors to finish their computation results in more values it can perform on its own, but of course results in itself being idle. But, only asking for those values when it absolutely needs them results in a series of coordination problems...and without the other processor realizing that it should prioritize communicating the values it depends on, it may have to wait anyway. Finally, constantly polling to see if adjacent processors have finished their computation wastes time in itself.

At $t = 0$, given an IVP, each processor has the full set of values $u_x^0 \ \forall \ x \in X$. Unsurprisingly, to minimize idle processor time, computing the values at processor boundaries serves as the best method. Interleaving from either side, each processor can compute values at spatial points $\alpha * p$, $\alpha * (p+1) - 1$, $\alpha * p + 1$, $\alpha * (p+1) - 2$, et cetera, for time $t = 1$. At what point in time should it attempt to compute values at time $t = 2$? If it waits until all values at time $t = 1$ have been computed, it has performed computations that do not depend on other processors. If later it has to wait for values its future computations depend on, it has exhausted its own independent computations on which

it could fall back. And, it fails to supply values that other processors need as early as it can. But, the sooner we try to compute values at $t = 2$, the more likely that the values we need from adjacent processors won't have been computed yet.

Stepping back from optimizing execution ordering, consider the following strategies for dealing with synchronization.

3.4.1 Preemptive Concurrency Control Scheme

Commonly, when we synchronize values in shared memory, we perform some check to see if the value we plan to use has been computed. We may use a spinlock, where we check if the computing processor has relinquished its lock, and when it has, we expect the updated value to be in place. Let's call the expected value for the number of times we must perform this test $\mathbf{E}_{\text{pre}}$. Thus, to complete a simulation with T timesteps, we incur a runtime overhead of $O(\mathbf{E}_{\text{pre}}T)$, because we have 2 communications per processor per timestep; note that $\mathbf{E}_{pre}$ may depend on N and P and thus cannot be treated as a constant.

In a preemptive scheme, $\mathbf{E} \geq 1$. Each processor must check to see if its adjacent processors have computed their values the value preliminarily, and if the test succeeds, we can use the value straightaway. If the test fails, we may have to do some unbounded number of checks, say, if processor an adjacent processor enters an infinite loop.

3.4.2 Postemptive Concurrency Control Scheme

In a scheme without preemptive concurrency control, if the adjacent processors to some processor p have always computed their values before p accesses them, then there could be potentially no overhead with regard to performing tests. This precludes any verification of the shared values – it does not provide a mechanism for determining if the values p receives from adjacent processors have, in fact, derived from the expected computation. This leaves the desire for a scheme that performs this verification with minimal overhead. Let's call this overhead $\mathbf{E}_{\text{post}}$.

Consider two processors p and q that are adjacent, and processor p receives value u_{x-1}^t from q while computing values for timestep $t + 1$. If values fail to be resolved before multiple timesteps have taken place on p, all three values might need to be resolved. In the worst case, p completes all of its computation before q has done anything. Then, q will essentially, by resolving all values in p, perform the entire computation of values in p dependent on q in serial via its resolution functions. In the best case, q doesn't have to perform a single resolution, achieving nearly maximal parallelism. To optimize this process, p should perform computations that are not dependent on q while q runs its resolution functions; otherwise, it risks generating more values in need of resolution. Once all resolution has been performed on a particular value, the reference containing

the old version of that value can be deregistered, which results in it being freed from memory, as it no longer serves a purpose.

If the resolutions take less time, asymptotically, than the computations themselves, the algorithm avoids the time wasted by using synchronization schemes. The difference arises from the fact that as soon as a process computes a value, it can either operate upon values that that process also has, or values that another process has. If we limit computation, preemptively, to values local to a process, the algorithm avoids latency. The algorithm should avoid communication until absolutely necessary.

Chapter 4

Implementation

4.1 Overview of Interface

The fundamental structure that affords parallelism without data races is the CT vector, a front-facing interface for a bit-partitioned trie. This vector can be created and used like a C++ standard library vector (with some limitations) and can be modified at will in parallel environments. If used in this way, each process will continuously generate versions of the underlying trie that appear immutable to other processes, and thus each process will have its own view of the vector, independent of the others. Any given process cannot see the operations performed by the other processes, affording us consistency with respect to reading the contents of these vectors, but not yet with writing.

4.1.1 Splinters, Detaching, and Reattaching

In order to get consistency with respect to writing, the concepts of `detach` and `reattach` come into play. A CT vector can be *detached* off another CT vector, operated upon, and later *reattached* onto an output vector. These operations take place in a single process, and the *splinter* that was detached allows for non-blocking, asynchronous, parallel operations. The series of vectors slated for detaching and reattaching determines the global ordering of the operations. A `detach` that comes before a `reattach` sees the data structure as it was before the corresponding operations that conclude via that join. A `detach` that comes after a `reattach` sees the data structure as it was after the corresponding operations that conclude via that join. Orderings where there are two paired detaches and reattaches where both `detach` calls come before both `reattach` calls imply that the two operations are commutative and associative with each other, and the ordering doesn't matter. Both operations see each other, and the final result will be consistent if the commutativity and associativity of those operations hold. If they do not, different results will arise from different arbitrary orderings of those two operations with respect to each other; this is semantically correct behavior and could very well be intentional.

Here the semantics state that the logical ordering of the two operations is simultaneous, that they occur at the same point in time relative to a global logical clock.

4.1.2 Limitations of Splinters

When a user operates upon a splinter, it can only do so via methods available for that vector. It cannot arbitrarily write values into locations of the vector; while there are ways to address add and remove operations on CT vectors, for now we will focus on vectors whose size does not change. These methods allow pre-specified operations to be performed on some subset of values in the vector. The user can create these operations by instantiating instances of an `operator` and then tag the instances as being associative and/or commutative. In addition, pairs and sets of operators can have operations defined relative to each other, so that if multiple operators are used among different splinters with the same parent (a CT vector), that resolution can be optimized to allow the most lenient orderings that do not invalidate the result. Each splinter has a fixed operator it uses for its lifetime. This `compute` method takes a value to use with that operator and an index that determines the value in the splinter upon which the operator will be applied.

4.1.3 Resolving

Finally, there is a `resolve` method that forces blocking, synchronous resolution from one CT vector onto another, its *dependent*. This allows the user to perform intermittent i/o with consistent results, and for the user to create an ending terminus for a computation subgraph that supports resolution. If values from a CT vector are to be used in a non-CT context, the values must be fully resolved in order to to obtain consistent results from that point forward.

So, if the inputs to `compute` for use with the passed operator are values in CT vectors, then the resulting vector will be treated as a *dependent* of those vectors. Otherwise, the CT vector that performs the `compute` will be treated as the starting terminus of any resolutions in its computation subgraph.

4.1.4 Aggregate Operations

The library also contains a `stencil` function that takes an operator, another CT vector, and a set of relative indices; it then performs a *stencil* operation, where it uses the values at those indices relative to any/every index for the passed CT vector as the arguments for the operator provided, placing the results in a new CT vector returned and ignoring out-of-bounds indices. The function still takes iterators into the input vector to determine the range that the stencil is applied. (Note that this equates to performing a diagonal matrix/vector multiplication in the case where the iterators delineate the entire input

vector.) This also allows the resulting vector to receive updates if and when resolutions take place on the dependent vectors used to produce it. Dependencies on singular values can be encoded by passing a relative index vector of size 1 and iterators that only select one value in the vector, and in this case a vector of size 1 is returned. Likewise, a proxy for a reduce operation involves an index vector indicating every other value in the array and iterators that only select one value in the vector.

There are two other aggregate functions, `foreach` and `reduce`, that are special cases of `stencil`. The former takes two iterators (instead of a single index) that determine the start and end points in the vector where the operator will be applied and either a value to use with the operator on that range, or another CT vector whose values will be used along that range. The latter takes two iterators, and reduces the values in the calling vector onto the 0-index spot of a returned vector that has size 1.

4.1.5 Partitioning and Indexmaps

When an aggregate function is called on a CT vector, the appropriate `compute` operations are slated for execution, and each available process receives a partitioned portion of the task. The details for the `foreach` and `reduce` functions are described below. The details are given here because they generalize to operations that are data-parallel with an identity *index mapping* or `indexmap`, and with an all-to-one `indexmap`, respectively. Any other aggregate function boils down into some combination of these forms of data-parallelism. In general, the `indexmap` has a domain and a range. The domain consists of addresses of values that comprise those used for the aggregate function, and the range consists of addresses of values in the resulting, dependent CT vector. The `indexmap` itself relates which input addresses go into the production of a given output address. When a separate `indexmap` is provided for each input which is also a CT vector, the addresses simply become integral indices into those vectors. Recall the internal structure of the CT vector: nodes with either k branches (internal nodes) or k values (leaves). This structure allows us to minimize the amount of work necessary to partition the individual `compute` operations in a data-parallel way that avoids conflicts (and thus possibly costly resolutions) down the line. But, in general, future work could be done to identify and minimize the number of potential conflicts which must be tested for detection and subsequent resolutions which must be then performed for all detected conflicts.

The `foreach` function is split among processes in a data parallel manner. The CT vector is split into k^d nodes at each depth d of the underlying trie. If there are P processes and n values in the vector with depth $d = \lceil \log n \rceil$, The tree is ascended to depth d' where $k^{d'} \geq P$ but $k - 1^{d'} < P$, and each internal node $i \in \{0, \ldots k^{d'}\}$ are partitioned in chunks of k with the final process receiving a smaller chunk if necessary. Using an iterator, each process can move to the first value in the vector it owns and perform the operation at up to k nodes forward at that depth, stopping if it reaches the end of the vector.

The `reduce` function is split into a P chunks of values to be reduced. Each process reduces its chunk of values in serial, and then one processor reduces the resulting P values. If $P > k$, the values will be reduced in chunks of k until there are just k values left, and then one process P will reduce those resulting k values, finally writing the final value as the output, or to the resulting place in another CT vector, tagged with the operation given for the reduction. if $P \leq k$, a single process will reduce the values.

This constitutes a fairly simple strategy for scheduling work in a balanced manner among processes. Alternative and potentially more robust solutions are discussed as future directions.

4.2 Dependency Tracking, Conflict Detection, and Resolution

When a detach/reattach group of splinters begins, the first course of action is to create the output CT vector and label it as being dependent on the calling vector. When this happens, some other bookkeeping takes place to in order to manage the ensuing contention of data among parallel processes.

4.2.1 Freezing

When CT vectors are used as inputs for operations, they must be *frozen* prior to those operations taking place. To `freeze` a vector, one must supply the complete set of dependee CT vectors and have created the dependent vector where the operation's result will exist, in addition to the operators and index mappings that relate those dependents to the resulting vector. Each operation has a primary input and auxiliary inputs. The primary input is the vector that the user thinks of operating upon, and the auxiliary inputs can be seen as helpers, such as a list of coefficients by which to multiply each value in an input vector. Selecting a primary input does not produce uniquely correct programs, but can make the program clearer to the user and others who read the code.

4.2.2 Tracking

The registration process stores the `operator` and `indexmap` in a struct called a *tracker*. It also stores the frozen snapshot of the input vector, which it will use to perform its computations. It needs this because in order to correctly perform conflict detection and resolution, it must have a persistent, immutable reference to the input at the time it was used. Otherwise, subsequent updates to the input may render the reference inconsistent between its use and the conflict/detection and resolution processes. The tracker obtains the snapshot by atomically copy-constructing the CT vector into the tracker (which only really does a `shared_ptr` copy of the root node of the CT vector) and giving the input

CT vector a new id. Now that the input has a new id, any value that it updates will construct new nodes throughout the entire path to the modified value, and the copy in the tracker will remain untouched. This snapshot of the input can thus be seen as frozen at the point in time that this preparation takes place.

4.2.3 Reattach Latches

Finally, the freezing process also creates a *latch* with a count for each splinter that will work on this compute. This count has a default that can be set at compile-time. When the splinters are finished, they reattach their computed values to the output vector and decrement the count. When the count reaches 0, the computation is complete, save for resolution that may or may not need to take place. The Boost C++ library provides an implementation of a latch with this precise interface.

4.2.4 Differences in the Treatment of Primary and Auxiliary Input

The final step of latch creation happens for the primary input as expected, but for auxiliary inputs, it uses a value of 0 to avoid redundancy. Instead, the freezing process for non-primary inputs takes the primary input as an argument as well, and adds it to a list of auxiliary frozen inputs for that particular computation. When resolution happens from the primary input onto the output, it resolves the auxiliary inputs as well, once it has finished performing its own resolution. In this way it will always only resolve once the relevant splinters have reattached, and the user will only have to call resolve once per detach/reattach group.

4.2.5 Detach and Reattach, in Context

The user then calls a method called `detach` from any of the threads it created to perform asynchronous, parallel computation. It is returned with a splinter that it can operate on using the single-index `compute` method described above. If the user intended to operate upon the input during this computation, the splinters have the same contents as the frozen snapshot of the input, which is achieved by copying the snapshot and incrementing the ID of the splinter's copy; now, splinters will have to construct new nodes to the modified value as well, but only for the values that they modify. The user must either be careful to compute either with non-CT values, or to compute with values in a CT vector it specifically froze for the task, using the relevant `indexmap`, and in both cases, the relevant operator (though, the correct operator will be used via the `compute` method of the splinters). Failing to do so will result in an inconsistent resolution process, whether or not that leads to inconsistent values themselves.

Once the user is done, they call `reattach` to connect each splinter to the output vector. The `reattach` method will do its best to efficiently assign the values of the output vector

as necessary. If the splinters performed data-parallel operations with index mappings that have contiguous output values of groups of at least the branch size of the CT nodes, then `reattach` will be able to leverage the internal trie structure of CT vectors such that this operation takes place in as little as $\mathcal{O}(1)$ time.

Detaching and reattaching must be paired for each splinter, sandwiched by calls to `compute`, and exactly P_s calls to both must take place, where P_s is the number of splinters specified when freezing. Within `reattach`, the latch is decremented; if the latch decrements below 0, the code will terminate with an error. The aggregate operations above can be used to avoid having to manually detach and reattach splinters.

4.2.6 Resolution

At any point, the user may choose to never resolve the CT vectors if desired. If the user decides that they do not actually care to see the finalized values in those vectors, they may choose to leave them unresolved, incurring no runtime penalty for synchronization. Otherwise, resolution takes place on the chain of CT vectors in the order produced; resolving CT vectors that used another CT vector as a dependent without first resolving the dependent will not work, because the vector which performs the resolution onto the dependee must, logically, be finalized in order for resolution to take place. During resolution, the finalized input vector compares its values to the frozen version of it that the splinters used. If the values differ, it uses the inverse of the operator to determine what the relevant difference is, and the operator itself to update the value in the output vector. This, of course, relies on the invertibility of the operator. Non-invertible operators cannot be used with CT vectors.

The `indexmap` provided for the computation provides information about which values are prone to conflicts. If the user partitions the addresses that the splinters modify (the range of the `indexmap`), then the only values that may need resolution are ones for whom within a given splinter's range of the `indexmap`, value(s) in their domain lie(s) outside that range. This is so because within splinters, operations on CT vectors happen sequentially, so splinters always see finalized values at indices that lie within their range of the `indexmap`. It can be seen that for purely data-parallel operations (such as `foreach` operations), there are no dependents, and that, say, for a stencil that uses adjacent values in the CT vector, only $2(P-1)$ values must be checked, at each internal boundary between the splinters. If partition functions and index mappings are provided in code as `constexpr` functions and the number of splinters is provided in code as a `const` value, a partition of the range and the corresponding indices with possible conflicts can be determined at compile-time.

Chapter 5

Results

Results are shown from running benchmarks on a dual-core Intel IvyBridge i7-3520M processor with 12GB RAM running Debian Jessie/Stretch using g++ 6.1.1. The implementation, including the code for the benchmarks from which the following results derive, can be found at http://github.com/seanlaguna/contentious. The code also comes with an build process that uses autotools. The code depends on the Boost thread and system libraries, and on the concurrent producer/consumer queue and LifoSem data structures in Facebook's open source Folly library. A version of the code without the dependency on the data structures in Facebook's Folly library exists, but using these data strutures marginally improved performance consistency in the following benchmarks. A minimal subset of the Folly library is provided to avoid the need to install the entire library as a dependency.

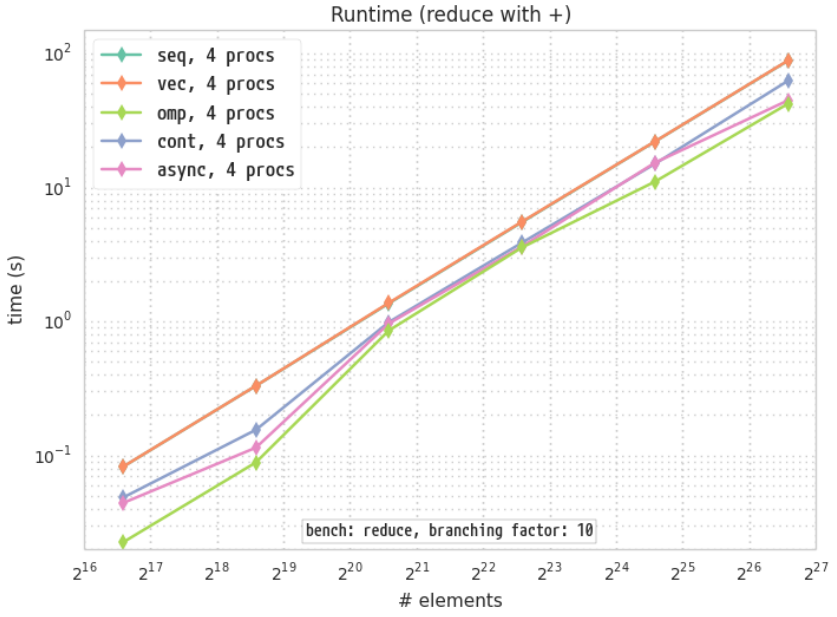
Tests are performed on each machine up to twice the number of physical threads; the maximum number of threads tested are thus 4. These utilize hyperthreading to run two simultaneous threads per core, but hyperthreads will not necessarily provide the same performance as running each thread on its own physical core [24].

5.1 Reduce

Results shown compare various implementations of a reduce operation of a random vector of doubles:

- "seq" is a sequential implementaiton;
- "vec" is a sequential implementation, but with values always stored in vectors (e.g., results of "reduce" operations are stored in size-1 vectors). This is provided because in order for this method to operate in parallel on vectors, the result operations must be itself stored in a ctvector, for comparison's sake, tests are performed with the same restriction with a std::vector;

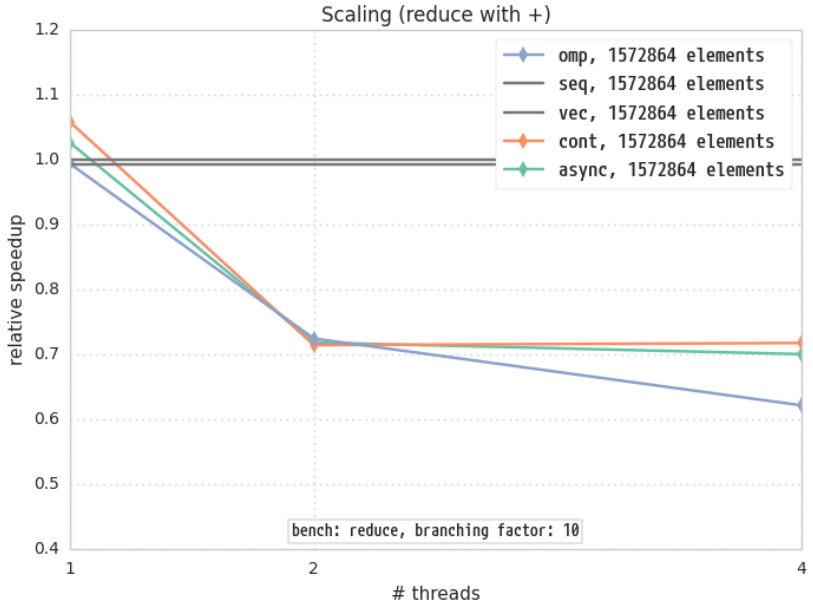
- "omp" is an OpenMP implementation. It uses the maximum number of cores available in all cases, so it is also not sensitive to the number of threads/cores that are manually selected;
- "avx" uses a 256-bit AVX instruction to perform the addition of 4 doubles simultaneously until only 4 values remain, and then uses more avx instructions to sum the resulting values. Thus, the speedup of this implementation can be compared directly to other implementations when using 4 concurrent threads;
- "async" is a C++11-style parallel implementation using the C++ std::async feature to launch threads, each of which are given a fraction of the list equal to the inverse of the total number of threads.

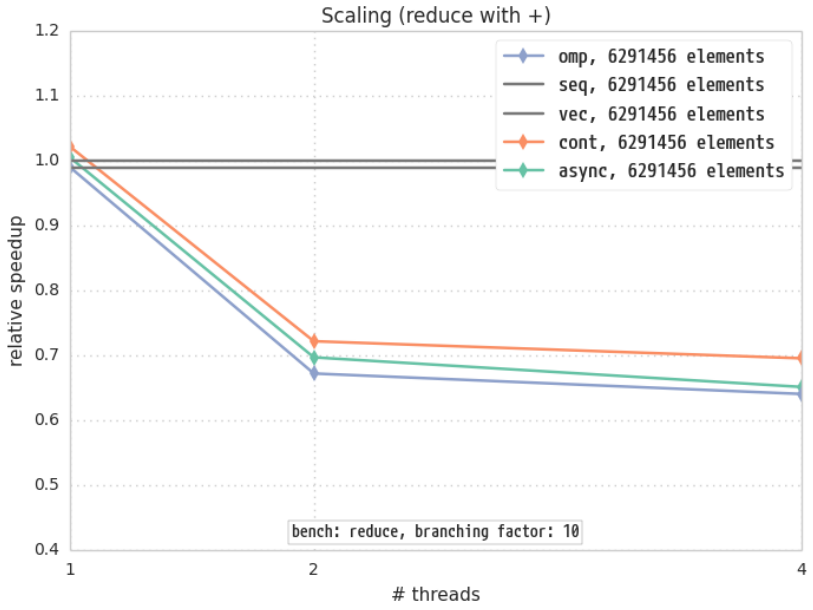


The results indicate that while incurring some overhead, the cont method scales in a similar way to the async and omp methods. The async benchmark better represents a low-overhead but manual parallelization scheme: like the cont benchmark, it does not include strategies to account for hardware details like the memory layout or cache coherence. Because OpenMP provides a more opaque, high-level interface for parallelizing a reduce operation, it stands to reason that the omp benchmark has better performance because of extra tuning provided by the library itself. Although, other software has further improved the cache coherence of OpenMP benchmarks, though similar ideas may have been incorporated into the current OpenMP implementation since then [18].

5.2 Foreach

Results show scaling for the contentious library performing a foreach operation on a random vector of doubles. For comparison, the results show the runtime of a serial

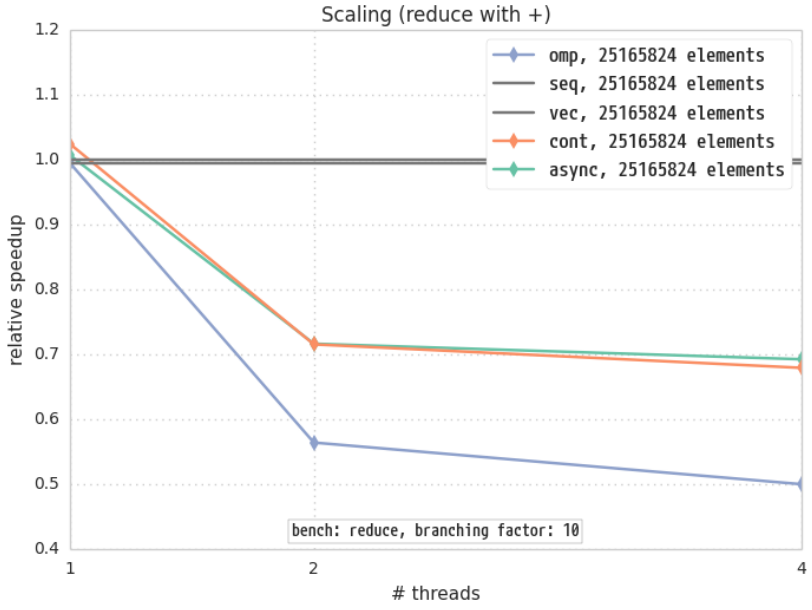


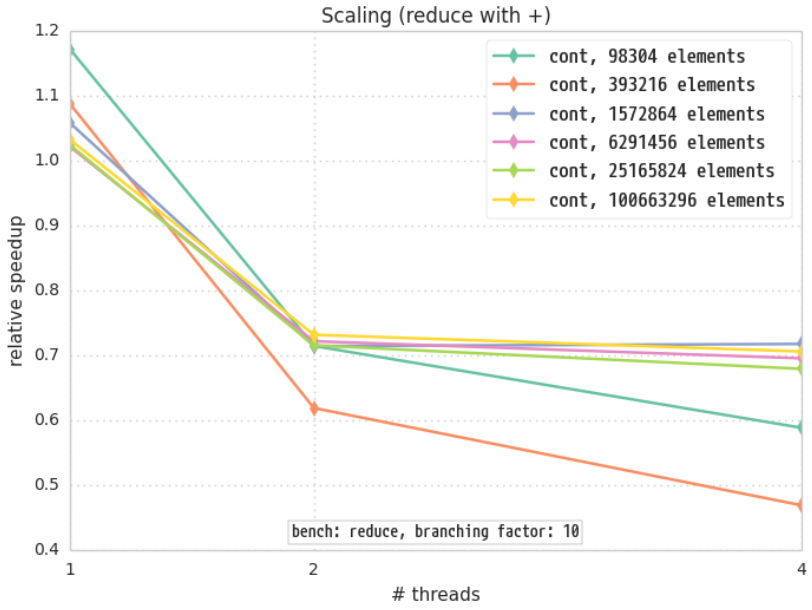


implementation as well.

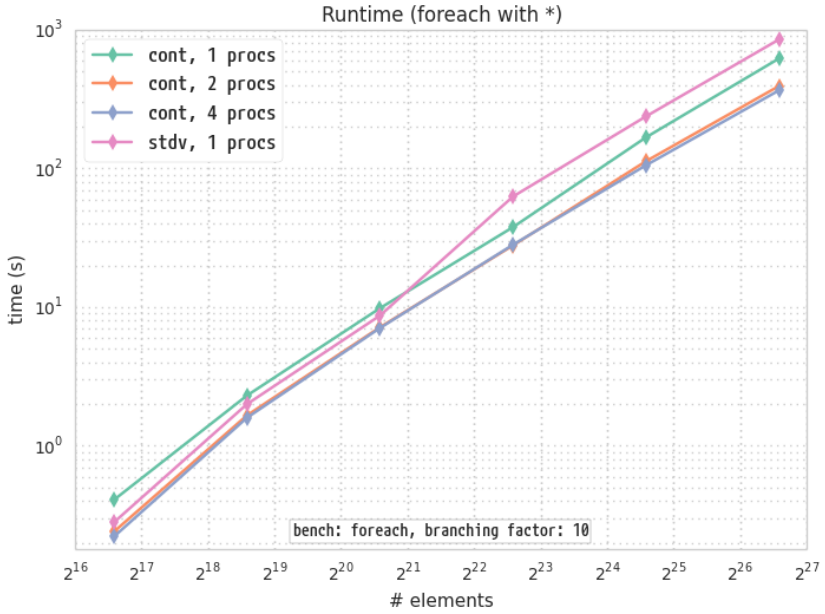
The results follow a similar pattern to those for the reduce benchmarks, except the baseline single-processor comparison shows that using the `ctvector` provides better performance than using `std::vector` at larger sizes. Because the `ctvector` stores data in a non-contiguous manner, the memory allocator and cache may work more efficiently when trying to allocate or compute upon data much larger than 4 MB (the size of the L3 cache on the tested machine).

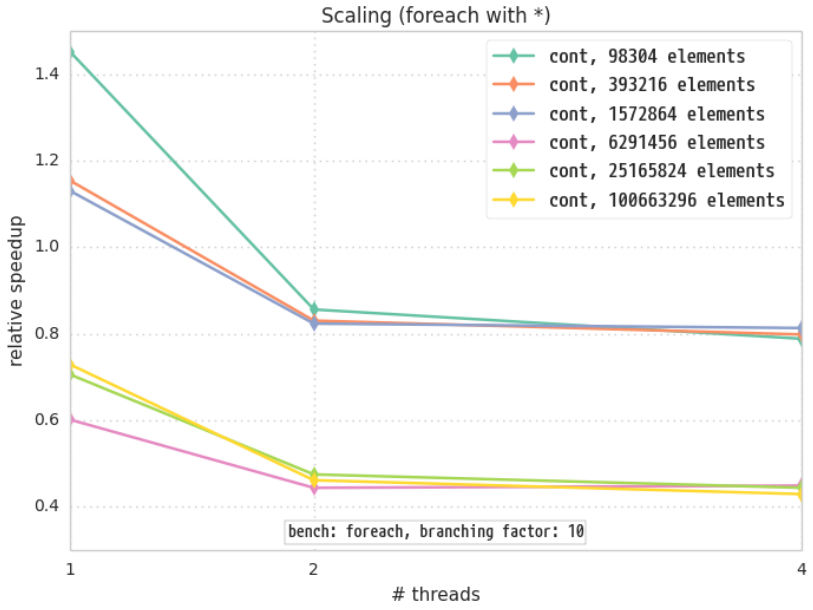
Another test demonstrates the effect on the branching factor of the bit-partitioned vector used to implement the `ctvector`. The results show that a minimum branching factor





of 2^{10} or 1024 provides approximately the best performance for the foreach benchmark; choosing the minimum branching factor allows for minimal copying when performing partial operations on the vector and otherwise reduces contention between processors. Results did not vary much depending on the particular benchmark or number of threads used. Other uses of this data structure, such as by D'orange, indicated that a much lower branching factor would provide near-optimal performance, such as 2^5 or 2^6. But, the use cases tested often involved random, out-of-order mutations, insertions, or deletions; our use cases involve almost entirely patterned mutations that can take better advantage of large, contiguous blocks of values.





5.3 Heat

Results show scaling for the contentious library using a 1st-order scheme to compute the heat equation framed as a Boundary-Value Problem.

Chapter 6

Discussion

Postemptive concurrency control strategies naturally flourish in settings with low contention and aid the programmer especially when that contention doesn't always have a predetermined structure. As demonstrated, such strategies allow for levels of parallelism comparable to current methods and adapt well to the fact that different parallel runs incidentally produce different degrees of parallelism. But, the lack of explicit contention management serves as perhaps the most attractive aspect of postemptive strategies. When using preemptive strategies, the correctness of the program depends on the ability of the programmer to explicitly manage concurrent reads and writes of shared data. Historically it has proven a time-consuming and complex task. Postemptive concurrency control schemes avoid the need for this and allow for reads and writes to shared data under common and practical use cases. Their downsides involve potential decreases in the runtime and space efficiency of the program and the need to supply extra information about the behavior of the program. But, in practical settings, it remains to be seen whether postemptive methods such as the one described competes with optimized preemptive methods.

6.1 Optimization of Implementation

There are a number of existing optimizations to CT vectors that are not present in the current implementation.

6.1.1 Tail Optimization

The CT vector can benefit from a tail optimization where the pointer to the final leaf is stored in the root. This optimization primarily benefits inserting at the end of the tree, and since the CT vector was primarily leveraged for mutations as opposed to inserting and removing elements, this optimization was not prioritized.

6.1.2 Branching Factor

The CT vector's nodes stores a constant number of branches or values where that constant is some power of 2, in order to provide an efficient method for lookup of values through the tree. When the constant is 32 or greater, the runtime of traveling to a leaf is at most $\log_{32}(n)$ where n is the size of the vector. It is unfeasible to store more than 256 billion bytes in memory (this corresponds to a processor that has access to 256 gigabytes of shared memory), and it is unrealistic to store objects smaller than one byte. $\log_{32}(256,000,000,000) \approx 7.58$, which means that the depth of the bit-partitioned trie of nodes and leaves will never exceed 8. With a branching factor of 64 and a vector with at most 1 billion values, the depth will never exceed 5. This growth is so slow that it can, for all intents and purposes, be treated as a constant, which is why a large branching factor is used. But, the larger the branching factor, the larger the copy when a CT vector sets a value in a node it has not yet touched, as it must copy entire nodes. Picking a contextually good branching factor, or even dynamically determining it or using heterogeneous branching factors for different parts of the trie, could improve performance.

In the setting of automated parallelism, often the pattern by which splinters will access and mutate the input vector is known in advance. When this is the case, it may be possible to choose a branching factor for points in the vector that corresponds to those access and mutation patterns. It is much more efficient for splinters to mutate whole leaves, or better yet, whole subtries, and the shallower the root of the subtrie in the CT vector, the more efficient the process. Tailoring the branching structure of the trie to confer these ideal conditions would result in an optimal reattaching and resolution process. Furthermore, assuming a sparse access and mutation pattern, the trie could even change its branching factor at runtime to minimize copying and to avoid unnecessary checking during reattachment and resolution, by keeping the size of newly-created leaves as small as possible. It remains to be seen whether the extra logic necessary to manage a bit-partitioned trie with nodes that do not always have the same branching factor would outweigh the optimizations made possible by that flexibility, or vice-versa.

6.1.3 Computation Ordering to Minimize Contention

When the range of the indexmap is partitioned among splinters, the domain does not necessarily partition in the same way. Any values in the domain that lie outside a given splinter's range are candidates for conflicts. These are the values checked in the resolution process. By having splinters compute these values first, the chance that conflicts will occur is minimized. This optimization has not been implemented, and it remains to be seen if the extra logic necessary to compute the values in a non-consecutive order can be implemented without the overhead outweighing the benefits. Furthermore, performing operations on non-contiguous data can cause cache misses, and so the unpreditable behavior of hardware might limit the performance of this strategy. It is likely that there

are cases, such as multi-dimensional finite difference stencils, where the sparsity and pattern of values with potential conflicts are straightfoward enough to easily prioritize the slated operations.

6.1.4 Optimization of operators and indexmaps

Operators and indexmaps are currently stored in structs with `std::function` objects inside them. While this provides an elegant interface for dealing with functions logically in C++, this standard library function type does not allow for the same level of optimization as raw C-style function pointers, or better yet, truly inline operations that map to an optimized set of assembly instructions. For operations with relatively few (or even exactly 1) assembly instructions, the overhead of working with `std::function` objects adds too much overhead to the execution of the functions contained within. The inability of the C++ compiler to inline functions stored as these types exacts the majority of the overhead. A more efficient interface for managing functions as variables and working with them symbolically is necessary for this vector to be truly competitive with standard implementations. Likely this will involve some combination of C-style function pointers and templated functions that have template parameters that receive those function pointers.

6.1.5 Fully lock-free Reattaching and Resolution

Currently the CT vector implementation uses two locks: one to lock the map where trackers are stored when they are emplaced into the map, and another to lock the dependent vector's bit-partitioned trie when reattaching of reduce-style partitions and when resolution happens. The first lock is necessary and does not affect performance, because the emplacement of values into this map and lookups in the map accounts for very little actual work compared to the amount of work splinters would do in actual uses cases for parallelisim. The second lock has two uses. In the first, really a single processor should perfrom this reattachment, because each processor holding a lock to perform the final reduce of P or fewer values incurs substantial overhead compared to a simple serial reduce. In the second, locking prevents the need to resolve again with the same dependee-dependent pair after resolution takes place if an output vector depends on multiple input vectors. The code has not yet been configured to rerun resolution in the correct sequence to avoid this lock, although as described it is possible to achieve resolution in a totally lock-free manner.

The only other synchronization primitive used is a `std::atomic<int32_t>` for assigning IDs to CT vectors (and raw transient vectors, if used).

6.2 Improvement of Interface

6.2.1 The Copying Problem

CT vectors do not have consisent semantics for being copied by value. Currently when the user calls the copy constructor on a CT vector, it returns a copy of the underlying bit-partitioned vector with a new ID and an otherwise empty set of members. Passing a CT vector by value to a method and expecting it to behave like a genuine copy of the original vector does not work. It will not be able to serve as a candidate for detaching or reattaching splinters as if it were the original vector, and additionally will not be able to serve as a base for resolution of vectors that were produced by using the original vector as input. This is, of course, the correct behavior, but accidentally calling the copy constructor in various situations may confuse users.

Developing a better interface and writing a semantically correct move constructor would solve this problem.

6.2.2 The Intermediate Dependencies Problem

One major concern involves writing subroutines that perform multiple stepwise computations (where each step involves a set of splinter detach and reattach calls) from a local function scope, where the function returns the final output vector. In order to preserve the asynchronous properties of the CT vector, the intermediate functions must be resolved outside the local function scope, or else the resolution will be premature and force synchronization when the function returns. But, there is no good way of moving those intermediate CT vectors outside of the local function. Currently, the destructor of the CT vector waits for all detached splinters to reattach, as otherwise, segfaults will take place while the splinters do try to reattach. The destructor does not perform forward resolution or wait for itself to be resolved; doing so would prevent the desired asynchronous properties. Resolving onto a vector that has been destroyed (by being `delete`d if allocated on the heap, or by going out of scope) is currently undefined behavior in the same way that using any deleted object is undefined behavior. This is why the interface requires that the vectors being resolved onto are passed manually as parameters: it forces users to have undeleted and in-scope references to them in order to perform resolution on them.

Bibliography

[1] Edward Anderson et al. "LAPACK: A portable linear algebra library for high-performance computers". In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press. 1990, pp. 2–11.

[2] Grey Ballard et al. "Communication-optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '12. Pittsburgh, Pennsylvania, USA: ACM, 2012, pp. 193–204. ISBN: 978-1-4503-1213-4. DOI: 10.1145/2312005.2312044. URL: http://doi.acm.org/10.1145/2312005.2312044.

[3] Shekhar Borkar. "Thousand core chips: a technology perspective". In: *Proceedings of the 44th annual Design Automation Conference*. ACM. 2007, pp. 746–749.

[4] Edsger W. Dijkstra. "Over seinpalen". circulated privately. n.d. URL: http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF.

[5] Peng Du et al. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[6] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[7] Pieter Ghysels et al. "Hiding global communication latency in the GMRES algorithm on massively parallel machines". In: *SIAM Journal on Scientific Computing* 35.1 (2013), pp. C48–C71.

[8] Tobias Grosser et al. "Polly-Polyhedral optimization in LLVM". In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Vol. 2011. 2011.

[9] Gaël Guennebaud, Benoit Jacob, et al. "Eigen". In: *URl: http://eigen.tuxfamily.org* (2010).

[10] Ian Henriksen. "Circumventing The Linker: Using SciPy's BLAS and LAPACK Within Cython". In: (2015).

[11] Jialu Huang et al. “Decoupled software pipelining creates parallelization opportunities”. In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM. 2010, pp. 121–130.

[12] Simon L Peyton Jones and André L M Santos. “A transformation-based optimiser for Haskell”. In: *Science of computer programming* 32.1 (1998), pp. 3–47.

[13] M. Kalin and D. Miller. “Clojure for Number Crunching on Multicore Machines”. In: *Computing in Science Engineering* 14.6 (Nov. 2012), pp. 12–23. ISSN: 1521-9615. DOI: 10.1109/MCSE.2012.69.

[14] Alex Kogan and Erez Petrank. “Wait-free Queues with Multiple Enqueuers and Dequeuers”. In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 223–234. ISSN: 0362-1340. DOI: 10.1145/2038037.1941585. URL: http://doi.acm.org/10.1145/2038037.1941585.

[15] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[16] Jean Niklas L’orange. “Improving RRB-Tree Performance through Transience”. MA thesis. Norwegian University of Science and Technology, June 2014.

[17] Jean Niklas L’orange. *Understanding Clojure’s Persistent Vectors, pt. 1*. 2013. URL: http://hypirion.com/musings/understanding-persistent-vector-pt-1 (visited on 12/01/2015).

[18] Jaydeep Marathe, Anita Nagarajan, and Frank Mueller. “Detailed cache coherence characterization for openmp benchmarks”. In: *Proceedings of the 18th annual international conference on Supercomputing*. ACM. 2004, pp. 287–297.

[19] Frank Neese. “The ORCA program system”. In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 2.1 (2012), pp. 73–78.

[20] Steve Plimpton. “Fast Parallel Algorithms for Short-Range Molecular Dynamics”. In: *Journal of Computational Physics* 117.1 (1995), pp. 1–19. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1006/jcph.1995.1039. URL: http://www.sciencedirect.com/science/article/pii/S002199918571039X.

[21] Steve Plimpton et al. “Crossing the mesoscale no-man’s land via parallel kinetic Monte Carlo”. In: *Sandia Report SAND2009-6226* (2009).

[22] Alceste Scalas, Giovanni Casu, and Piero Pili. “High-performance Technical Computing with Erlang”. In: *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG ’08. Victoria, BC, Canada: ACM, 2008, pp. 49–60. ISBN: 978-1-60558-065-4. DOI: 10.1145/1411273.1411281. URL: http://doi.acm.org/10.1145/1411273.1411281.

[23] Nicolas Stucki et al. “RRB Vector: A Practical General Purpose Immutable Sequence”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 342–354. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784739. URL: http://doi.acm.org/10.1145/2784731.2784739.

[24] Xinmin Tian et al. "Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance." In: *Intel Technology Journal* 6.1 (2002).

[25] TOP500.org. *TOP 10 Sites for June 2015*. 2015. URL: http://www.top500.org/lists/2015/06/ (visited on 09/30/2015).