SOURCE-TO-SOURCE TRANSFORMATIONS FOR PARALLEL OPTIMIZATION IN STAPL

An Undergraduate Research Scholars Thesis

by

BRIAN KELLEY

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Lawrence Rauchwerger

May 2019

Major: Computer Science
Applied Math

TABLE OF CONTENTS

ABSTRACT

Source-to-Source Transformations for Parallel Optimization in STAPL

Brian Kelley
Department of Computer Science & Engineering
Texas A&M University

Research Advisor: Dr. Lawrence Rauchwerger
Department of Computer Science & Engineering
Texas A&M University

Programs that use the STAPL C++ parallel programming library express their control and data flow explicitly through the use of skeletons. Skeletons can be simple parallel operations like *map* and *reduce*, or the result of composing several skeletons. Composition is implemented by tracking the dependencies among individual data elements in the STAPL runtime system. However, the operations and dependencies within a *compose* skeleton can be determined at compile time from the C++ abstract syntax tree. This enables the use of source-to-source transformations to fuse the composed skeletons. Transformations can also be used to replace skeletons entirely with equivalent code. Both transformations greatly reduce STAPL runtime overhead, and zip fusion also allows a compiler to optimize the work functions as a single unit. We present a Clang compiler plugin and wrapper that automatically perform these transformations, and demonstrate its ability to improve performance.

CHAPTER I

INTRODUCTION

STAPL is a powerful system for both distributed and shared-memory parallel programming. It provides a library of distributed containers inspired by the C++ standard template library (vector, graph, matrix, etc.) and high-level parallel execution patterns (map, scan, reduce, etc.). The execution patterns are called skeletons. Each skeleton can call user-defined work functions on each element, and accesses container data through “views”. It also provides a runtime system for scheduling these parallel operations according to their data dependencies [13]. STAPL can coarsen skeletons, which means that operations on individual elements are be grouped together to reduce the overhead. The goals of STAPL are similar to the popular Cilk system [2, 7], a C language extension for shared-memory parallel programming. Cilk provides lower-level parallel constructs like “spawn thread” and “synchronize”, but also higher-level patterns like parallel for loops (the “map” skeleton in STAPL) [7]. Through this lower-level approach, Cilk’s runtime builds a “spawn tree” of tasks, and launches these tasks as soon as its dependency tasks have been completed [2]. STAPL uses a similar but higher-level approach: each skeleton depends on some views for input, and may modify some views as output. This data dependency graph is called a “PARAGRAPH” [13]. PARAGRAPHs may be arbitrarily complex, since skeletons can be nested and composed (the output of one skeleton becomes the input of another). The STAPL runtime is also able to send work through MPI to idle processors (remote method invocation) [13].

Generally, STAPL handles skeleton nesting, composition and coarsening at runtime. For example, if two map skeletons (with work functions f and g) are composed and operate on the same vector, the PARAGRAPH will contain two separate nodes with a dependency between the two. However, it is possible to combine these two operations into one: h(x) = g(f(x)). In this case, running a single map skeleton with work function h has the same effect as running the original maps f and g in sequence [13]. Combining the operations has several benefits: it simplifies the PARAGRAPH, it lets the compiler eliminate redundancies between the two operations, and it allows data elements to be used twice while they are in the cache.

Coarsening is a general skeleton transformation that STAPL implements with template metaprogramming. It replaces a skeleton with a two-level nested version. The outer skeleton is coarse-grained and operates on chunks of input data. The inner skeleton runs the work function within chunks. The granularity (chunk size) can be tuned for specific skeletons, but chunks respect data locality - the elements in a chunk are all stored in the same location (an MPI process, or a thread within a process) [13]. Coarsening requires that the input views be transformed to nested views at runtime to match the structure of the nested skeleton. The inner view type supports zero-overhead iteration, since no communication or global-to-local index translation is needed. Source-to-source transformations benefit performance by replacing the coarsened skeleton with equivalent statements that create the same nested view, but implement the skeleton's algorithm directly. This avoids the overhead of PARAGRAPH creation and STAPL's runtime scheduler.

We use the Clang C++ compiler infrastructure to transform STAPL programs. Clang is a C/C++ compiler, but it also exists as a library intended for writing syntax tree analysis and transformation tools. This can be done either in a standalone program or in a Clang compiler plugin, which runs as a pass after the normal C++ frontend [5]. Duffy et al. used LibTooling (the library for standalone AST transformation programs) to analyze the control flow complexity of

C++ programs [3]. The ROSE compiler infrastructure project is another example of a syntax tree analysis framework. ROSE is particularly suited for analyzing parallel programs that use OpenMP and MPI [8, 10]. However, STAPL programs are represented using abstract skeletons that are insulated from the details of threads, communication and synchronization, so a Clang plugin is more than sufficient for our purposes. A plugin was used instead of a standalone tool because plugins do not make persistent changes to source files and do not interact directly with the many compiler flags needed for building STAPL programs.

CHAPTER II

METHODS

Plugin and Compiler Wrapper

The source-to-source transformations performed in this paper are done in a single Clang plugin. The plugin is a shared library that is linked to various LLVM and Clang libraries, providing AST traversal and source rewriting. The Clang plugin can easily be added as a compilation step between parsing and code generation. It traverses the AST, detects opportunities to perform transformations, and performs changes using a Rewriter. However, the AST in memory is an immutable data structure - it is not possible to reparse modified source code and continue the compilation process [5]. The Rewriter simply stores a list of textual changes and can output the full modified source (either to a new file, or by overwriting the original). Because of these constraints, the original goal of optimizing STAPL programs with no user intervention is achieved using a compiler wrapper.

The compiler wrapper is a simple script that replaces the normal compiler (e.g. mpicxx) in the user's Makefile. This only requires setting the variable CC_USER. The wrapper takes exactly the same set of arguments as the normal compiler. It determines from its arguments the set of input source files. For each input file, it runs Clang with the plugin set to output a temporary transformed file. The -emit-ast flag is added to stop the compiler after the frontend. After each input file has been transformed, the original compiler command is run with substituted input files. Running the compiler frontend an extra time for each file does increase build times, but otherwise the transformation process is invisible to the user.

Each transformation in the plugin has the same structure. A PluginASTAction is registered in Clang's global list of frontend actions to run. The "action" in this case is to traverse the AST using a custom subclass of both ASTConsumer and ASTVisitor. ASTConsumer's HandleTranslationUnit method is overridden to traverse the AST and then output the modified file. ASTVisitor's VisitStmt method is called on every statement and expression in the program. VisitStmt is overridden to test whether the statement is a STAPL construct that can safely be transformed, and if so does the transformation using the Rewriter.

Zip Fusion

The first and simplest transformation to be implemented was zip fusion within compose skeletons. The zip skeleton runs a work function with an element from each input view to produce an element of an output view [13]. When two or more zip skeletons are composed, their work functions can be fused to construct a single equivalent zip. This can be expected to reduce STAPL runtime overhead when the skeleton is executed. Compose skeletons can be identified in the AST by their canonical type name "stapl::skeletons::skeletons_impl::compose". The compose type's template parameters contain the types of the internal skeletons. Knowing which are inputs and outputs of each skeleton depends on the *flow* type of the compose (also determined from template parameters). The flow types are *inline* and *piped*. When an inline compose is constructed, special placeholder objects are used to identify the inputs and outputs of each skeleton by index. STAPL already supports the zip fusion transformation for inline flows: stapl::skeletons::transform<tags::zip_fusion>(…) produces a new skeleton where consecutive zips have been fused. The plugin detects calls to the inline compose constructor, and adds this transformation automatically if there is more than one zip skeleton.

If the compose uses a piped flow, the order of skeletons passed to compose's constructor determines the dataflow. The output of one skeleton becomes the input(s) of the next. For example, if zips *A*, *B*, and *C* are fused, the new skeleton has the same inputs as *A* and the same outputs as *C*. The new work function calls the work functions of *A*, *B*, and *C* in sequence, passing the return value of one as an argument for the next. The plugin generates the definition of the new work function: its call operator returns *C(B(A(...)))*. *A* may accept different sets of parameters, so one version of the call operator is generated for every overload and template instantiation of *A*'s call operator. std::result_of is used to determine the return type of each version. With the input and output types identical, the new work function is guaranteed to be compatible with every valid usage of the original compose.

Coarse Zip Elimination

Skeleton coarsening is a commonly used feature of STAPL that reduces runtime overhead. Instead of managing the data dependencies for each view element individually, a coarsened skeleton processes elements in chunks. The default coarsener forms chunks by grouping elements by locality. For an array_view with the default distribution, each chunk is the contiguous group of elements stored on a given thread. If all input views have the same data distribution, zip skeletons require no communication or synchronization. In this case, the coarsened zip execution is equivalent to a for-loop over the local elements that calls the work function. This means that all skeleton overhead can be completely eliminated. Because the view distribution is not known at compile time, the replacement code must first check that all views have the same distribution. If not, the original skeleton is executed. Constructing the nested views (where the inner level is local) is a relatively cheap operation, so little performance is lost in the fallback case. In the fast case, each thread simply iterates over the inner views' values and calls the work function. No synchronization is needed, because a thread can’t participate in a later collective operation until it has finished processing its chunk.

Coarse Reduce Elimination

This transformation idea can also be applied to skeletons that do require communication. A reduce skeleton is equivalent to a thread-local reduction, then a reduction within the process, and finally a global reduction. The local reduction can be replaced by a loop over the local chunk. The process reduction also happens within shared memory, but is done by a single thread. The global reduction can be done by a direct call to MPI_Reduce. The intra-process reduction uses a shared list of reductions from each thread, and local thread 0 computes the reduction of those values. To do local synchronization between the steps, a boost::thread::barrier is used. The barrier is created as a shared object within the STAPL process. If the program is running in shared memory, then thread 0 has the final result. Otherwise, thread 0 participates in an MPI reduction on behalf of the process.

There are some criteria for when a distributed reduction is possible - if they are not met at runtime, the original skeleton is used as a fallback. It is only done if the original STAPL skeleton was run across all processes in the system (MPI_COMM_WORLD). This is because creating a subcommunicator is an expensive operation and it would negate any speedup from STAPL overhead reduction. Another limitation is that the element type must be trivially copyable. Although STAPL supports serialization of all user-defined datatypes, MPI requires types to be of a fixed size. C++ primitive types are built in (MPI_INT, MPI_DOUBLE, etc.). Other types are defined as a fixed-size byte array using MPI_Datatype_Contiguous. Next, the reduce operation is converted to an MPI_Op. All associative arithmetic functions in STAPL are also built-in MPI operations: stapl::plus becomes MPI_SUM, stapl::bit_xor becomes MPI_BXOR, etc. More complex work functions are wrapped in a lambda that conforms to the MPI_User_function interface, and then this lambda is passed to MPI_Op_create.

Coarse Scan Elimination

The final skeleton that can be rewritten by the plugin is scan. The scan skeleton generalizes a prefix sum. Each element of the output view is the reduction over all previous elements of the input view. An inclusive scan includes element k of the input when computing element k of the output, but an exclusive scan does not. Scan is similar to reduce in that they both use an associative binary operator. Scan is also a built-in algorithm in MPI. Scan skeletons can be replaced using a two-pass algorithm, as shown in Figure 1.

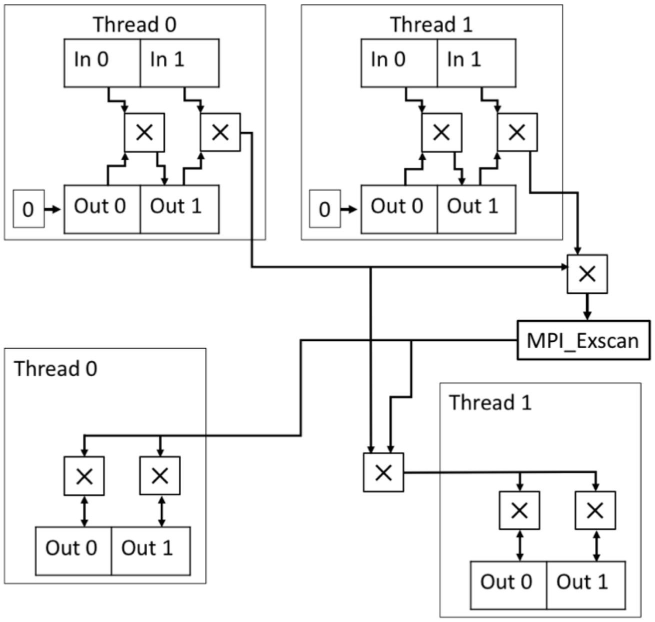

Figure 1: Two-pass distributed exclusive scan.

In the first pass, each thread does a local scan and places the final reduction in a shared array. Thread 0 from each process does a scan over the thread reductions, and passes its reduction to MPI_Exscan. The output of MPI_Exscan is the reduction over all previous processes. In the second pass, each thread combines the MPI scan output with the process scan output, and finally combines that with each local element. An inclusive scan is identical except the outputs are shifted to the right by one element in the local scan. The identity element (“0”) is implicitly an input to the first binary operation, and the local reduction is just the last element of “Out”. The logic to get the MPI_Datatype and MPI_Op is the same as with the reduce transformation. This algorithm does not require the operation to be commutative - at each application of the operator, it is known which argument precedes the other.

CHAPTER III

RESULTS

Shared Memory and Parallel Library Comparison

To measure the impact of skeleton elimination, a test program was created to run each skeleton on 10 million element array_views and test for strong scaling. The array size was chosen to be large enough for scaling to be possible, but small enough that milliseconds of overhead have a significant impact on runtime. All tests were run on Texas A&M University's "Ada" cluster. Each node has two sockets and uses 10-core Intel Xeon E5-2670 v2 processors. The interconnect is FDR Infiniband [12]. GCC 6.3.0 with OpenMPI 2.0.2 was used as the compiler since Ada does not support Clang with MPI. The transformations were done on another machine and the modified source files were copied onto Ada.

To frame the impact made by the transformations, each skeleton (map, reduce and scan) was also benchmarked in two other parallel libraries, Kokkos [4] and Intel TBB [6]. Unlike STAPL, both work only in shared memory. The two projects have similar goals: ease of use compared to raw threads, portability and scalability [4, 6]. For a fair comparison, each library was run within a single node with between 1 and 20 threads. STAPL supports both Pthreads and OpenMP as threading backends, so each was measured separately. Also, a naïve serial version was measured as a baseline. All times are averages over 50 trials.

The map test simply scaled a vector of 10 million doubles. Figure 2 shows that the transformations improved scalability - running time never increased from adding more threads and the transformed OpenMP version went on to be the fastest. Without the transformations, there was a performance cliff above 8 threads. In Pthread STAPL, the transformation gave 6%

and 13% speedups at 4 and 8 threads respectively; for other cases at 1-8 threads there was no significant difference. TBB appears to enable better compiler vectorization than all other libraries; however, TBB's runtime decided not to use more than one thread so it shows no scaling.

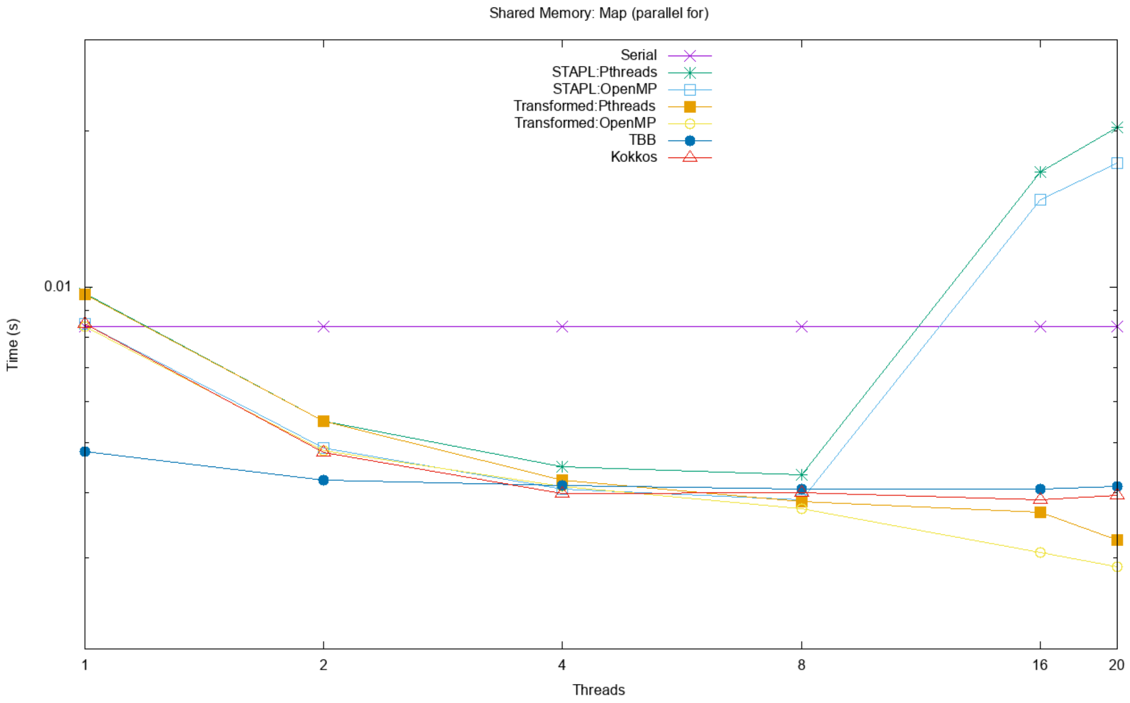

Figure 2: Shared memory map performance

Figure 3 shows the running times of sum-reduction on the same 10 million elements. The transformation has very little effect (~1%) on OpenMP but provides a good speedup (up to 41% at 8 threads) on Pthreads. The transformed OpenMP code stays competitive with Kokkos at all thread counts.

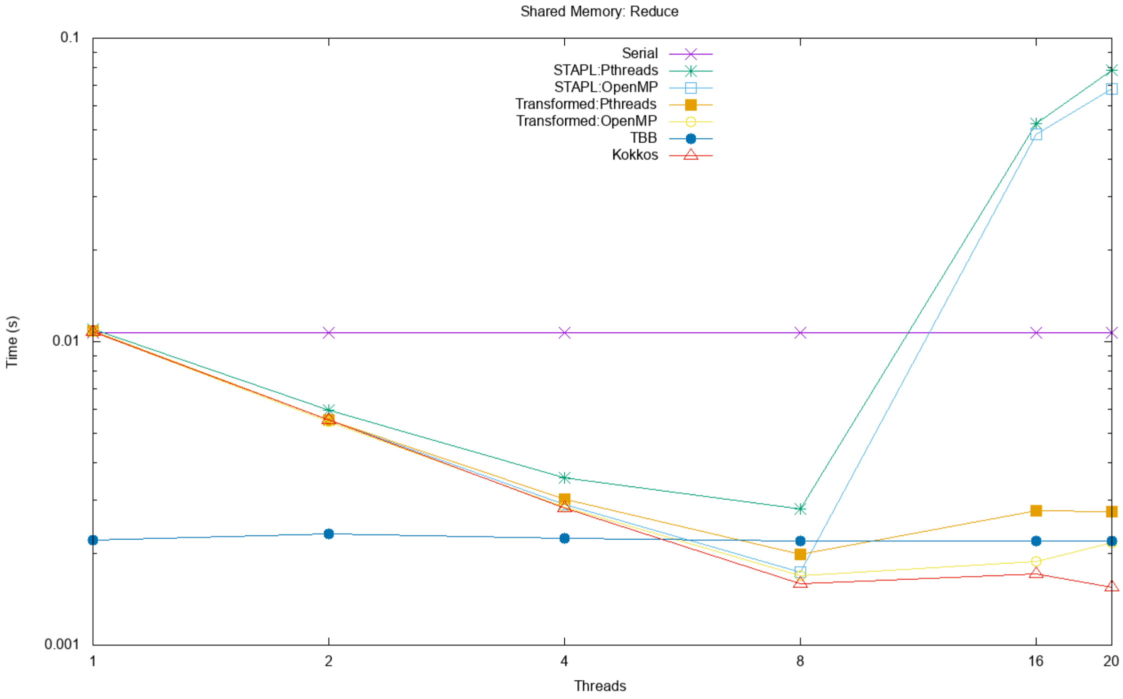

Figure 3: Shared memory reduce performance

Figures 4 and 5 are for exclusive and inclusive prefix sums, respectively. Exclusive scan is the only skeleton in which the transformed code is sometimes slower than the original - at 8 threads the transformed OpenMP code is 29% slower than the original. However, it is also 42% faster at 2 threads. The transformed code now experiences the performance cliff at 16 and 20 threads. STAPL is the only library that uses a different algorithm for inclusive and exclusive scans by default - a Blelloch scan is used for inclusive and a binomial scan for exclusive [1, 13]. In Kokkos and TBB, the distinction between inclusive and exclusive is instead made in the user-defined “join” function.

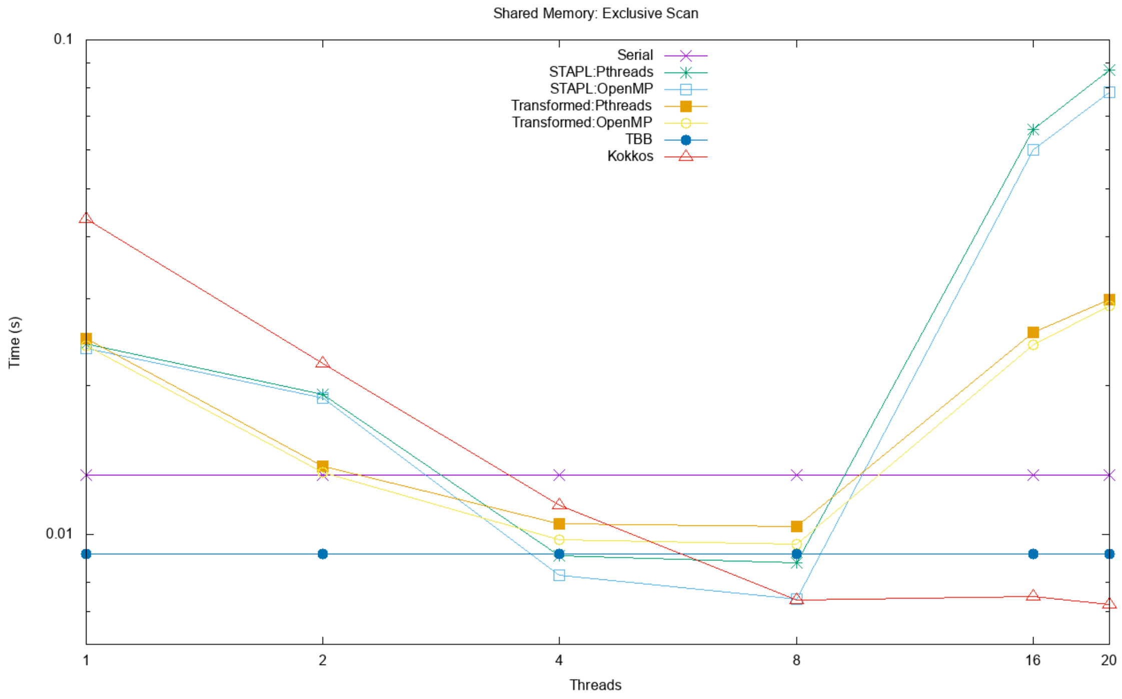

Figure 4: Shared memory exclusive scan performance

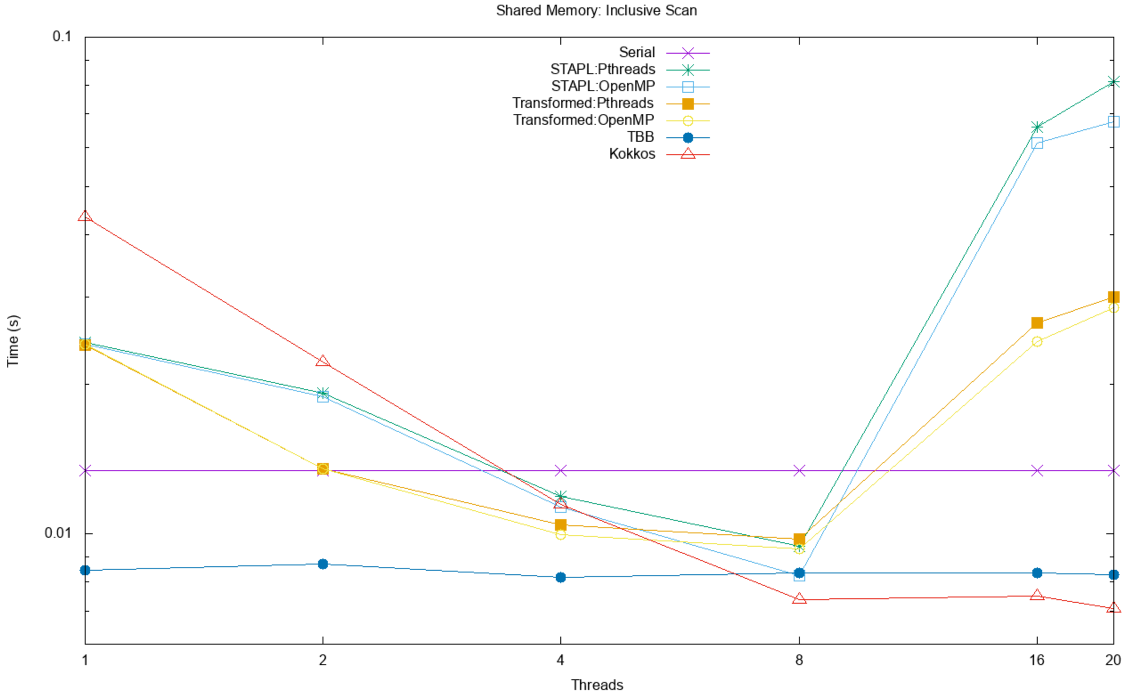

Figure 5: Shared memory inclusive scan performance

Distributed Memory

Because Kokkos and TBB are exclusively shared-memory, the only comparison that can be made is between STAPL and the transformed programs. The original and transformed algorithms did similarly well at 4 threads, so a constant 4 threads per node was used to exercise hybrid parallelism without an unfair comparison. OpenMPI was configured to give each process an entire node using “-npernode 1”, so all interprocess communication is over the interconnect [12].

Figures 6 and 7 show that transforming both map and reduce skeletons leads to dramatic speedups. Using OpenMP, the average map speedup is 3.3 and the average reduce speedup is 4.6. The transformed programs also scale better, with consistent speedups after each doubling of the node count.

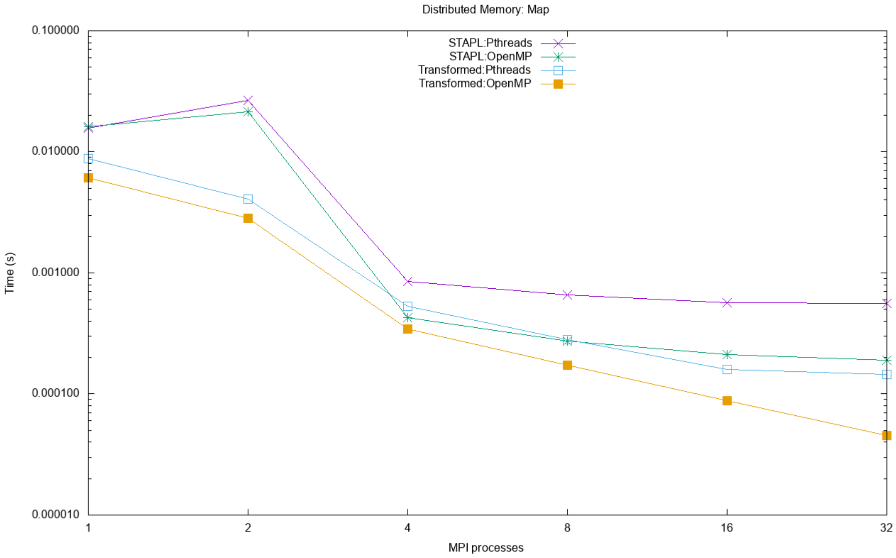

Figure 6: Distributed map performance

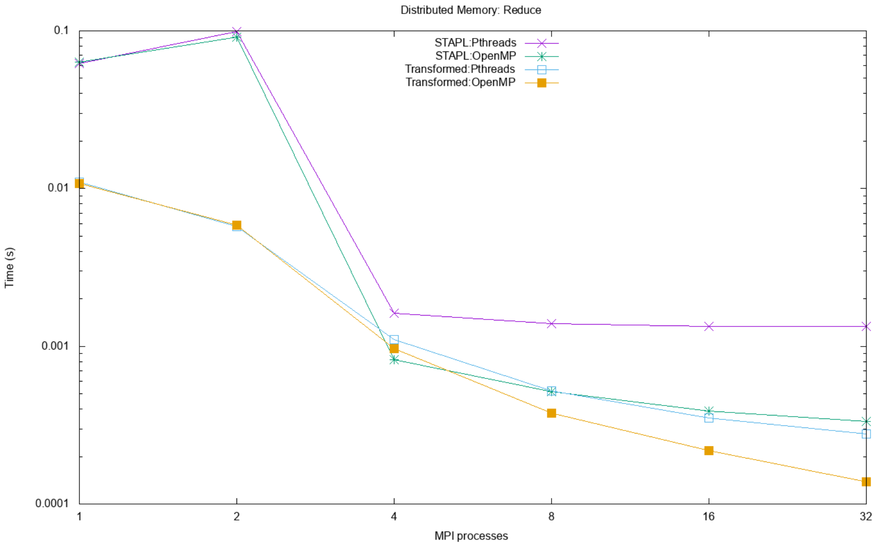

Figure 7: Distributed reduce performance

Transformations on exclusive (Figure 8) scans were not successful in the distributed case. At 32 nodes the transformed code was 73% slower. By contrast, transformation on inclusive scans (Figure 9) gave a consistent speedup, averaging 85%.

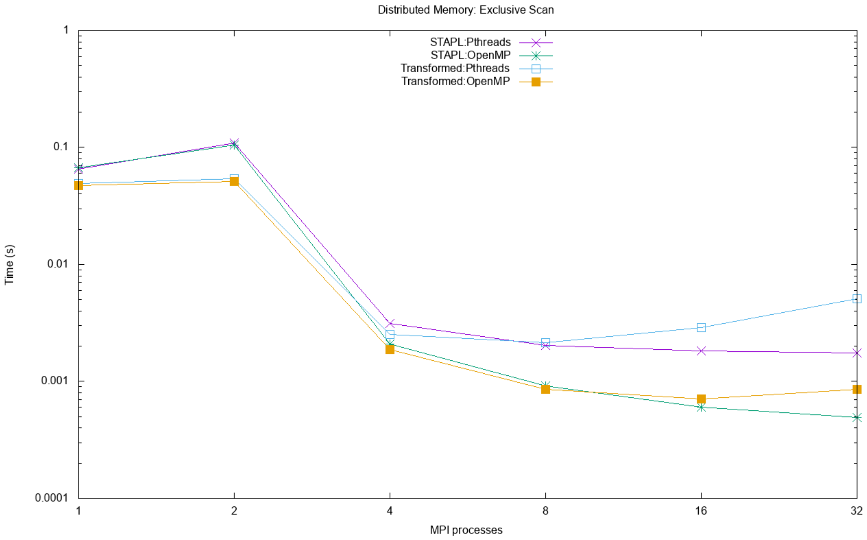

Figure 8: Distributed exclusive scan performance

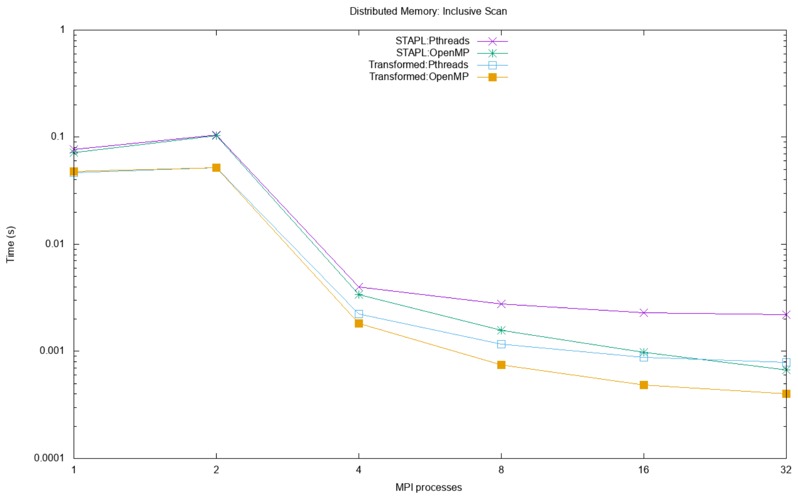

Figure 9: Distributed inclusive scan performance

Piped Flow Zip Fusion

Zip fusion on piped compose skeletons was demonstrated by composing four zip skeletons. Each represents a piece of a procedure to generate a Mandelbrot set image. The input to the compose skeleton is a simple counting sequence $0 \ldots width * height$, and the output is the image as a flat RGB array. This test is only to demonstrate the decrease in PARAGRAPH overhead, so the compose skeleton is not coarsened. The image is 960x540 pixels, so each skeleton is executed 518,400 times. Figure 10 shows the effect of fusing the four zip skeletons into one. A consistent speedup of 76% is achieved.

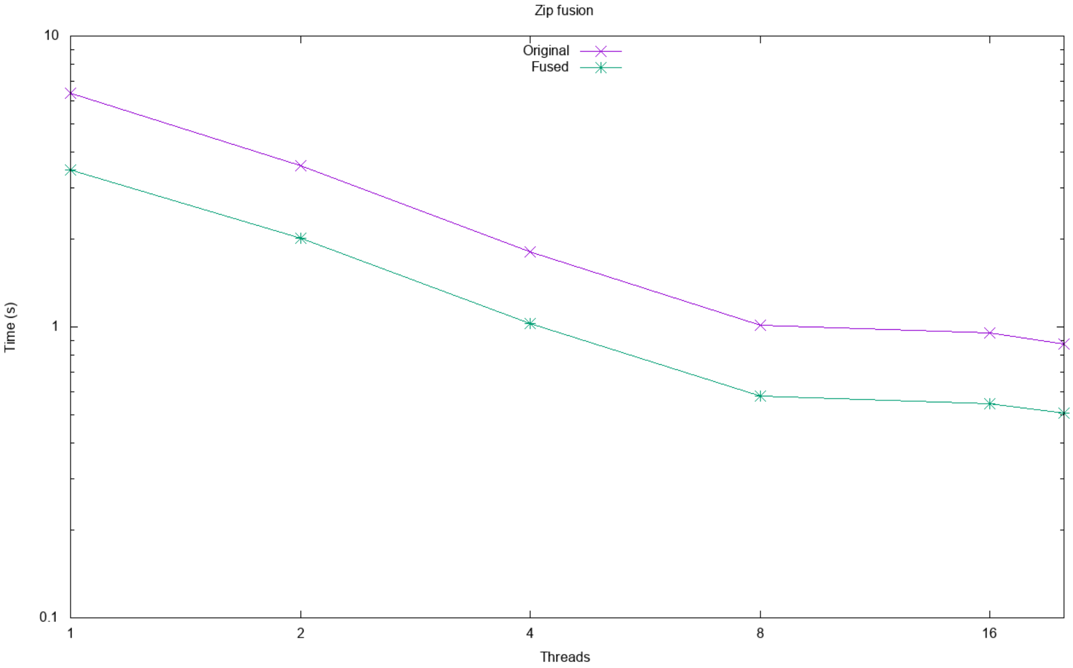

Figure 10: Zip fusion performance (not coarsened)

CHAPTER IV

CONCLUSION

We have demonstrated that source-to-source transformations can automatically improve the performance of STAPL programs. For three of the most fundamental algorithmic skeletons (zip, reduce and scan), a Clang frontend plugin is able to find STAPL calls that are candidates for transformation and then replace them with direct implementations of the algorithms [5, 13]. The replacement code has minimal interaction with STAPL's runtime system, reducing overhead and communication. The transformation process is also robust. It can handle arbitrary user-defined data types and work functions with few limitations.

STAPL fully supports hybrid parallelism, using Pthreads or OpenMP for threads and MPI for communication. The skeleton transformations support both - they generate explicit MPI calls where communication is necessary. Although STAPL's intended use case is scaling efficiently on thousands of nodes [13], we demonstrate that after transformations it compares favorably with two leading shared-memory parallelism frameworks (Intel TBB [6] and Kokkos [4]).

Future Work

This project's handling of exclusive scan skeletons is neutral or detrimental to performance. STAPL's binomial scan implementation is very efficient [13], and it might not be possible for this optimization strategy to be improve upon it. The project could also be continued by adding transformations for other STAPL skeletons. The challenge is to design the replacement code in a way that preserves parallelism. For skeletons with more complicated data access patterns like stencil and wavefront this would be difficult. Another task that might be automated using transformation is allowing skeletons to run on GPUs using CUDA. Managing memory transfers between host and device efficiently would be more sophisticated than simply iterating over chunks of elements as was done for the simple skeletons.

References

[1] Blelloch, Guy E. 1990. *Prefix Sums and their Applications.* Department of Computer Science, Carnegie Mellon.

[2] Blumofe, Robert D., Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. "Cilk: An Efficient Multithreaded Runtime System." *Journal of Parallel and Distributed Computing* 37 (1): 55-69.

[3] Duffy, Edward B., Brian A. Malloy, and Stephen Schaub. 2014. "Exploiting the Clang AST for Analysis of C++ Applications." *ACM Southeast Conference.*

[4] Edwards, Carter H., Christian R. Trott, and Daniel Sunderland. 2014. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." *Journal of Parallel and Distributed Computing* 74 (12): 3202-3216.

[5] Finkel, Hal, and Gábor Horváth. 2017. *Code transformation and analysis using Clang and LLVM.* Paris, June 12.

[6] Intel Corporation. *Intel® Threading Building Blocks.* https://software.intel.com/en-us/intel-tbb.

[7] Leiserson, Charles E. 2010. "The Cilk++ concurrency platform." *The Journal of Supercomputing* 51 (3): 244-257.

[8] Liao, Chunhua, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski. 2010. "A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries." *Lecture Notes in Computer Science* (Springer, Berlin, Heidelberg) 6132: 15-28.

[9] Negara, Stas, Kuo-Chuan Pan, Gengbin Zheng, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kalé, and Paul M. Ricker. 2010. "Automatic MPI to AMPI Program Transformation." *Euro-Par.* Berlin: Springer-Verlag. 531-539.

[10] Preissl, Robert, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. 2010. "Transforming MPI source code based on communication patterns." *Future Generation Computer Systems* (Elsevier) 147-154.

[11] Quinlan, Daniel J., Markus Schordan, Bobby Philip, and Markus Kowarschik. 2001. "The Specification of Source-To-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications." *LCPC.* Cumberland Falls, KY: Springer-Verlag.

[12] Texas A&M HPRC. 2019. *Ada Hardware Intro.* 2 28. Accessed 4 6, 2019. https://hprc.tamu.edu/wiki/Ada:Intro.

[13] Zandifar, Mani, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. 2014. "The STAPL Skeleton Framework." *Workshop on Languages and Compiler for Parallel Computing.* Hillsboro, OR: LCPC.