Tensor Program Hyperparameter Search Parallelization with Shared Address Space and Message Passing Models

Joshua Hong, Kit Ao

{jjhong, kitao}@andrew.cmu.edu

Based on the work of Wu *et al*

Webpage URL: https://joshua-j-hong.github.io/15618-Parallel-DFS-Hyperparam-Search/

Github Repository: https://github.com/AMKCode/mirage

Summary

We parallelize the DFS-based hyperparameter search algorithm of the Mirage tensor optimizer using a Shared Address Space model (via OpenMP) and a Message Passing model (via MPI) by analyzing the potential resource bottlenecks and workload imbalance inherent in the original system. Testing on the Catalyst Cluster as well as PSC machines, we achieve better performance and parallelization for both models compared to the original implementation. Additionally, we analyze the shortcomings of our implementations and propose additional improvements.

Background

Modern machine learning applications rely on tensor programs that are often optimized through engineered rules and heuristics. However, this method of optimizing tensor programs is time-consuming and may not always maximize optimization opportunities. Instead, recent research has demonstrated automated tensor program optimization as a promising alternative. These approaches search over tensor programs that are equivalent to the original program and assess their relative performance on target GPUs. We focus on Mirage, a tensor optimizer that performs an exhaustive search over the kernel, thread block, and thread levels in order to find configurations (known as μGraphs) that are equivalent to the input tensor program.

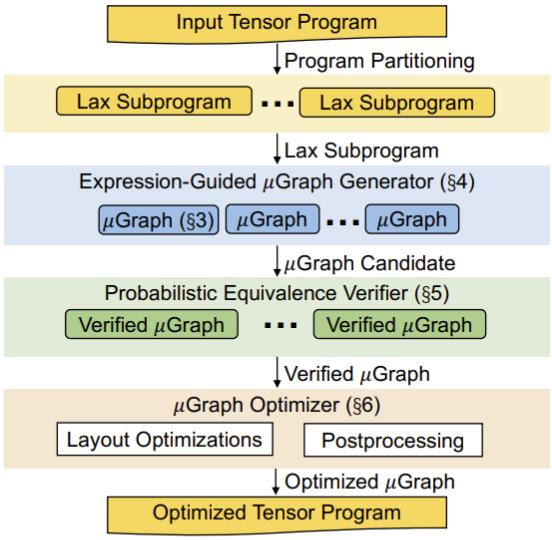

Figure 1. An overview of Mirage

Given a tensor program, Mirage performs the following steps. First the tensor program is split into smaller subprograms called LAX fragments consisting of multi-linear operations (matrix multiplication, convolution), division, as well as exponentiation. This aims to reduce the search space of equivalent tensor programs without obstructing optimization opportunities. For each LAX fragment, Mirage then exhaustively generates μGraphs that are equivalent to the fragment by considering optimizations at the kernel, thread block, and thread levels. μGraphs generation can be thought of as a DFS search, where Mirage traverses a tree to find valid program configurations. Mirage then uses a probabilistic equivalence verifier to ensure that the μGraphs generated in the exhaustive search are functionally equivalent to the original program. For valid μGraphs, Mirage then considers possible data layouts and selects the best μGraphs and layout combinator as the output for the corresponding LAX fragment. The μGraphs from all LAX fragments are then combined into one tensor program and returned. For our project, we focus on the μGraphs generation and verification steps which can benefit substantially from parallelism.

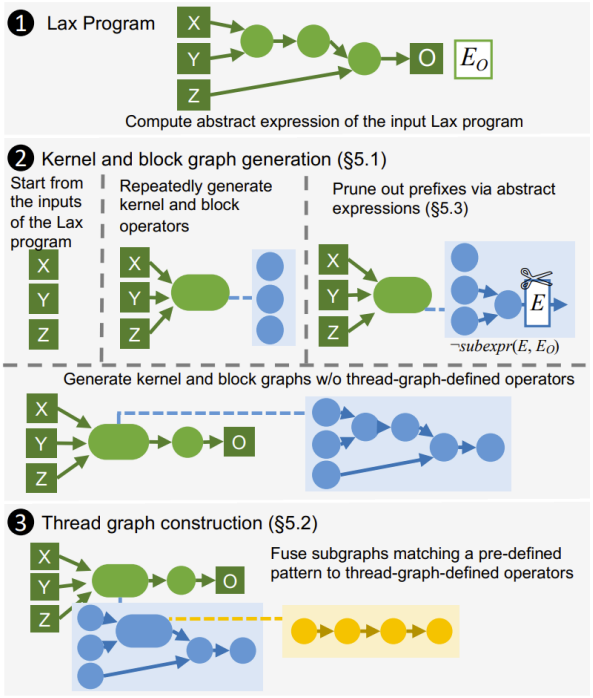

Figure 2. An overview of μGraph exhaustive generation

The exhaustive search that Mirage performs is facilitated by a novel pruning technique based on abstract expressions. This technique first computes the abstract expression, formally a function in first-order logic, of the LAX fragment. It then compares the expression to the prefixes of the generated μGraphs and prunes graphs based on a list of equivalence axioms. This is accomplished using a satisfiability modulo theory solver (SMT), with the specific solver used in Mirage being Z3. Additionally, Mirage caches repetition results to speed up computation, as SMT queries are relatively expensive.

Mirage's probabilistic equivalence verifier evaluates μGraphs and LAX fragments on random inputs in finite fields, which upon repetition provides correctness guarantees. To implement this, Mirage performs the random testing on GPUs to take advantage of optimizations and parallelism.

Approach

The code base we are starting from is the public Mirage repository, available on Github. This code base is primarily written in C++ and Cuda, with the library being built as a python package to be used with Pytorch. For our shared address space and message passing approaches, we utilize the OpenMP framework and the MPI framework respectively. We benchmark and test our parallelization implementations on the catalyst-cluster as well as PSC machines. To test our implementations, we use the repository provided demo tensor programs, which provide example programs of varying complexity.

Baseline Analysis

As we are attempting to improve on an existing system, we analyze the original parallelization approach that Mirage uses. We highlight 2 possible avenues of improvement: workload imbalance and resource contention.

First, we examine possible workload imbalance in Mirage. The original system balances search by first generating the search tree up to a height of 1. These middle states are then distributed to threads with an interleaved static workload assignment strategy, where the rest of the nodes and corresponding μGraphs are generated and verified. However, the pruning techniques used in Mirage to reduce the search space can result in middle states being eliminated early, allowing there to be variable amounts of work for each assigned middle state. We experimentally confirm this by collecting the runtimes of individual threads in Figure 3, where we observe that some threads finish their assigned portion of the exhaustive search substantially earlier than other threads.

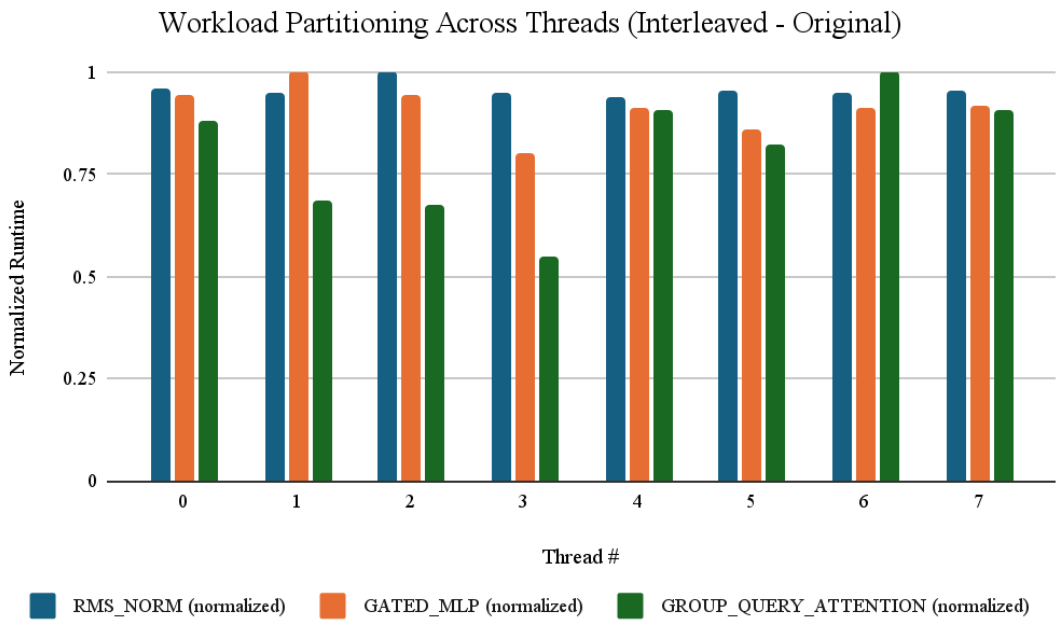

Figure 3. Individual thread runtimes for different demo tensor programs for the original Mirage implementation with 8 threads

OpenMP

To address workload imbalance due to static assignment, we add dynamic task assignment to Mirage using OpenMP. We accomplish this by defining the exhaustive search as a parallel region. Within this region, individual threads pick up tasks similar to the original approach and compute the valid μGraphs in the subtree. Additionally, we add some critical regions to protect shared data structures, such as the Z3 cache of seen abstract expressions and the returned list of μGraphs. We also add a tunable parameter to control the granularity of tasks for our OpenMP approach. This parameter, “max recursive depth”, controls the depth at which a thread should stop creating new tasks for the subtree and instead compute the μGraphs serially. If a task has a depth less than or equal to the recursive depth, the thread it is assigned to generates the children nodes and creates new tasks for them, which can be assigned to any thread. For a max recursive

depth of 1, each task will be equivalent to the original Mirage approach, which first generates the search tree to a height of 1 before performing static assignment. For greater recursive depths, each serially executed task will have a smaller associated subtree, offering greater task granularity. However, the number of tasks also becomes exponentially greater as the max recursive depth increases (as shown in Figure 4), which could potentially incur high overhead due to scheduling.

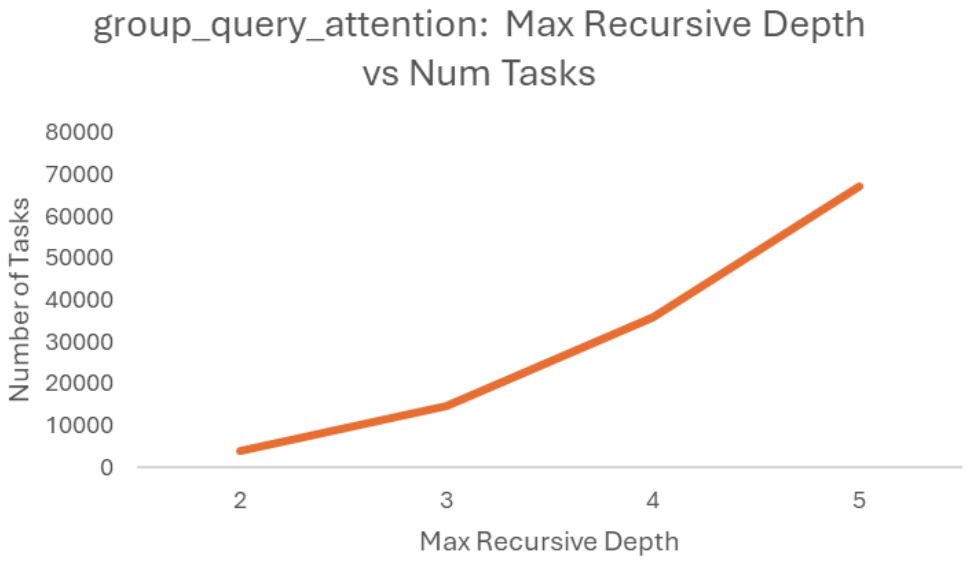

Figure 4. As Max Recursive Depth increases, the number of tasks created increases exponentially.

After implementing our OpenMP approach and benchmarking, we observed a drop in speedup for high thread counts. Upon further investigation, we noticed that there was resource contention between threads, specifically with the Z3 solver which is used to prune nodes during the exhaustive search.

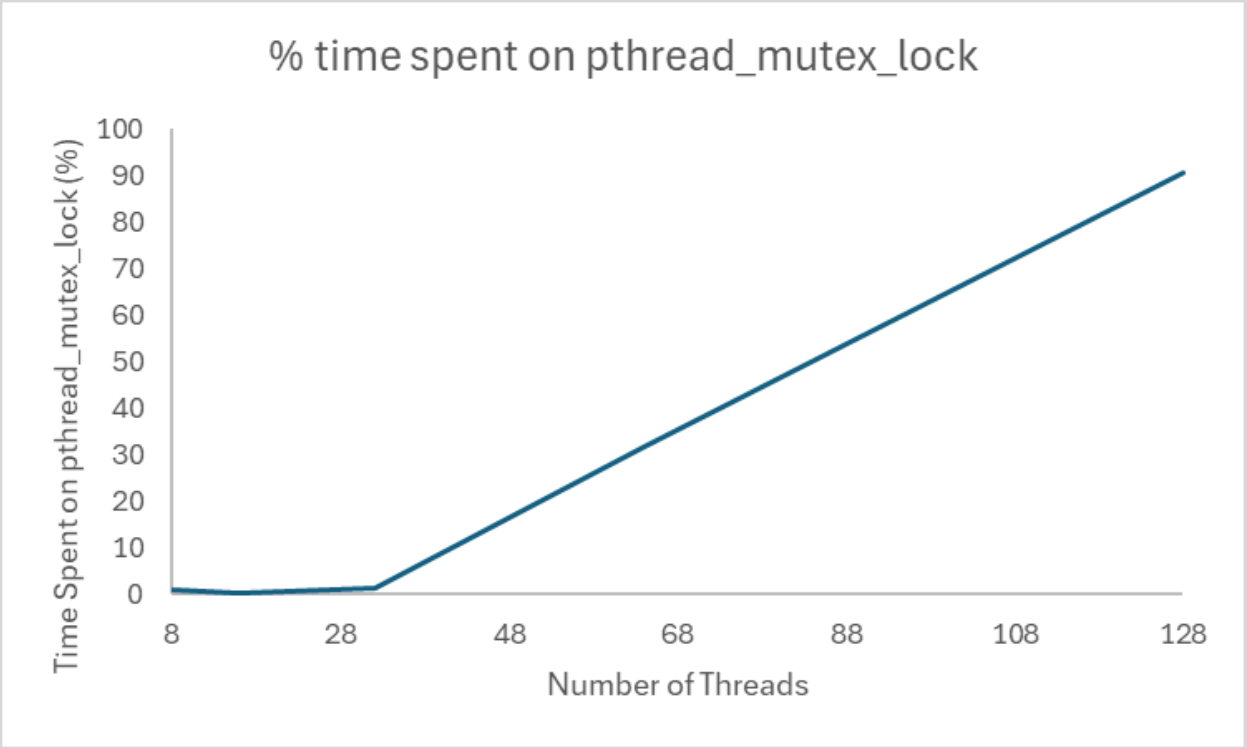

Figure 5. This graph depicts the time spent on pthread_mutex_lock() inside the Z3 solver at different thread counts as a percentage of the total execution time of the search process. This is done for the demo_rms_norm.py demo on the catalyst-cluster. The data is collected using Nvidia Nsight.

MPI

To address the issues surrounding resource contention, we implement a MPI-based message passing model. For our approach, we first generate the search tree up to a height of 1 on all nodes (essentially in the same way that the original implementation performed workload partitioning). Each node then computes a portion of the middle states, which are assigned via an interleaved static workload assignment strategy. As all nodes calculate these middle states, we observe that no communication is needed between nodes. Additionally, we verify experimentally that the overhead associated with this initial generation is negligible. After computing the associated μGraphs, each node then communicates the μGraphs back to the root node, along with corresponding search statistics. While it is possible that this introduces a communication

bottleneck on the root node, the number of valid μGraphs is low enough where we observe that the overhead associated with this step is negligible compared to the total runtime of performing the exhaustive search.

Results

Original vs OpenMP

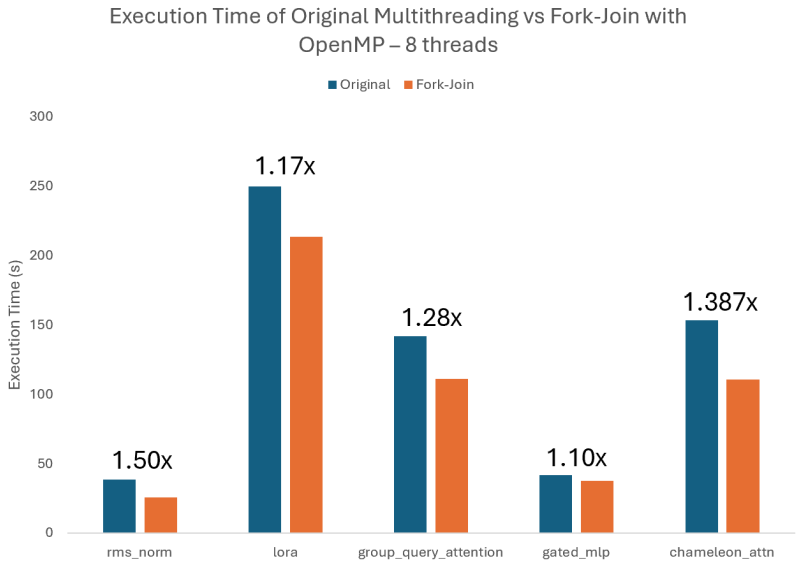

Figure 6. Execution time of the original Mirage parallelization approach compared to OpenMP, evaluated on the catalyst-cluster

The results demonstrate that the OpenMP-based fork-join parallelism approach resulted in a speedup of around 1.1x to 1.5x across the five examples, compared to the original pthread-based static assignment approach. This is because dynamic scheduling ensured good workload balancing across threads. While we couldn't implement timing code to evaluate the execution time of each individual OpenMP thread, program analysis using Nvidia Nsight demonstrates that

for the OpenMP approach, each thread demonstrated similar CPU utilization, which indicates that the computational load is balanced across OpenMP threads.

Original vs MPI

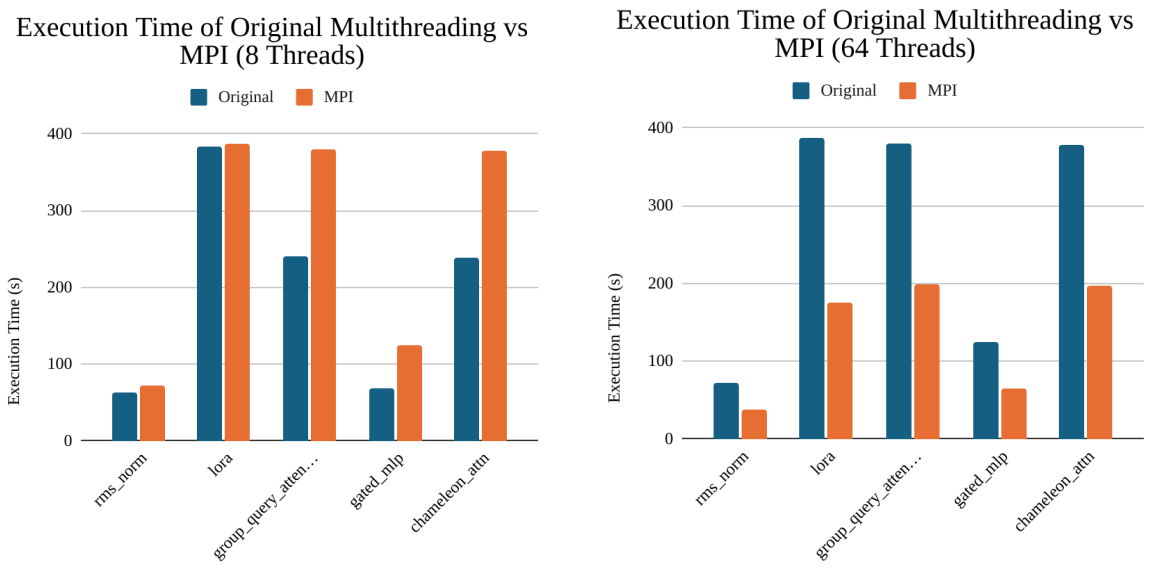

Figure 7: These graphs depict the execution time on PSC machines for the original pthreads-based approach and the MPI-based approach, for 8 threads and 64 threads.

At 8 threads, the MPI implementation performs worse than the original multithreading approach. This is due to both the overhead of creating separate address spaces for each process, as well as the fact that the message-passing model cannot utilize the z3 caching mechanism across threads to reduce the number of calls to the z3 solver. However, at higher thread counts (64 threads in this case), the MPI approaches were able to achieve on average a 2x speedup over the original implementation. This is precisely due to the fact that the MPI-based message passing model reduces the resource contention for the z3 solver by allowing each process to own its unique copy of the z3 solver.

MPI vs OpenMP

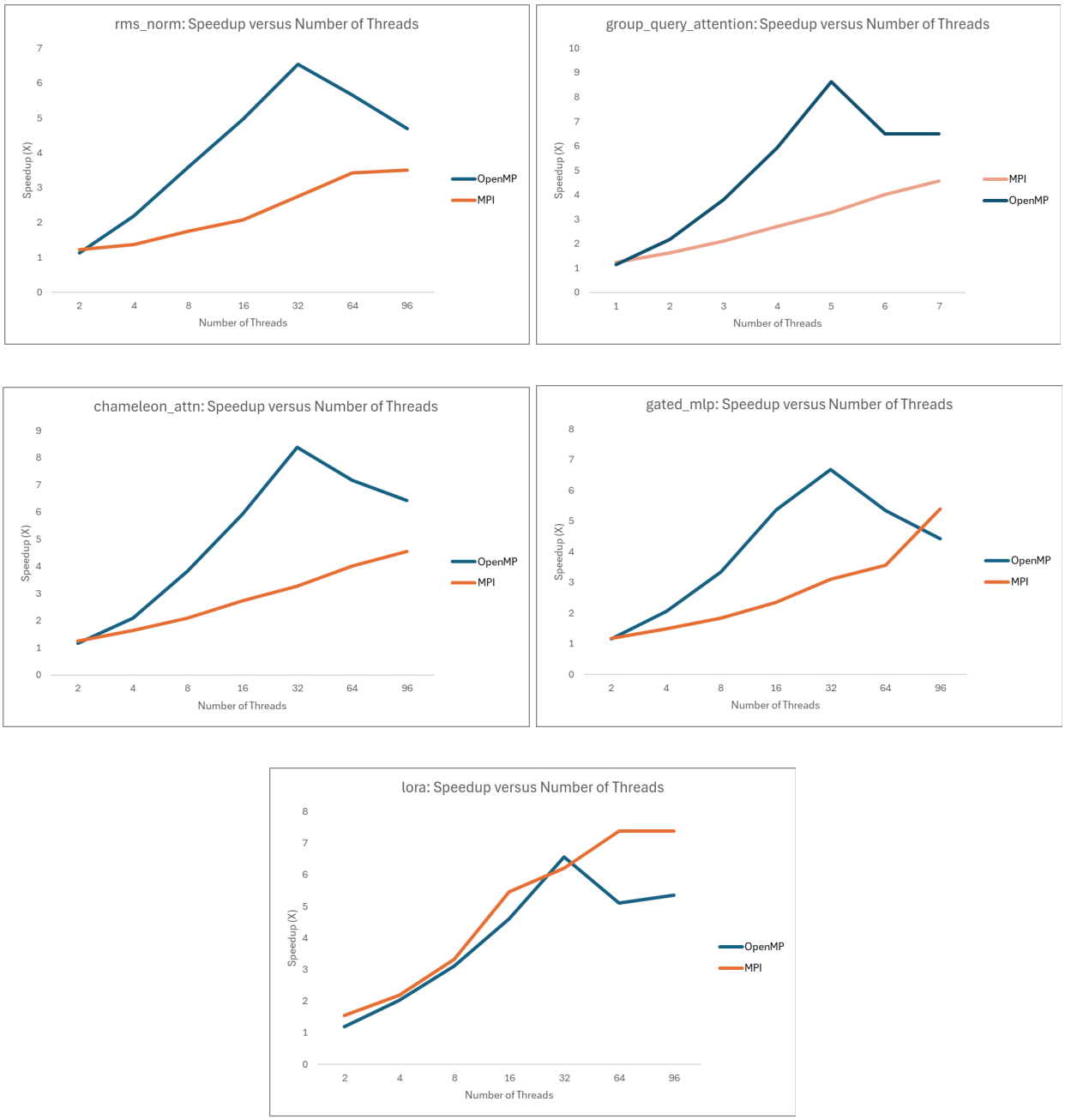

Figure 8. Speedup comparisons between OpenMP and MPI for varying thread counts and different demo tensor programs. All results were run on PSC machines.

For four of the five examples that we tested with, OpenMP was able to achieve greater overall speedup compared to MPI. However, for all five examples, the speedup of OpenMP drops off at

64 and 96 threads. This is because for OpenMP, Z3 solver resource contention drastically impairs the performance of the program, and the issue increases with higher thread counts as more threads contend for the same Z3 solver. However, for MPI, the speedup continues to increase steadily as the thread count increases, indicating that MPI is more scalable and parallelizable. This is precisely because for the MPI-based approach, there is no resource contention for the Z3 solver because every process has their own copy of the Z3 solver.

Interestingly, for the LORA example, which is the example with the longest overall runtime, MPI outperformed OpenMP at 64 and 96 threads, while maintaining comparable performance with OpenMP at lower thread counts. This could indicate that the MPI-based approach benefits from a larger problem size. One explanation for this is that the significant execution time for the LORA example makes the setup time of the message-passing model (which is more than that of the shared address space model because we need to create separate address spaces for each process) negligible. More experiments on problems with longer execution times needs to be performed in order to validate this hypothesis.

MPI Shortcomings

However, additional analysis revealed some shortcomings with our MPI implementation. First of all, the approach we took to reduce communication between nodes causes workload imbalance due to static assignment. We experimentally verify this by collecting data for the runtimes of individual nodes on the PSC machines.

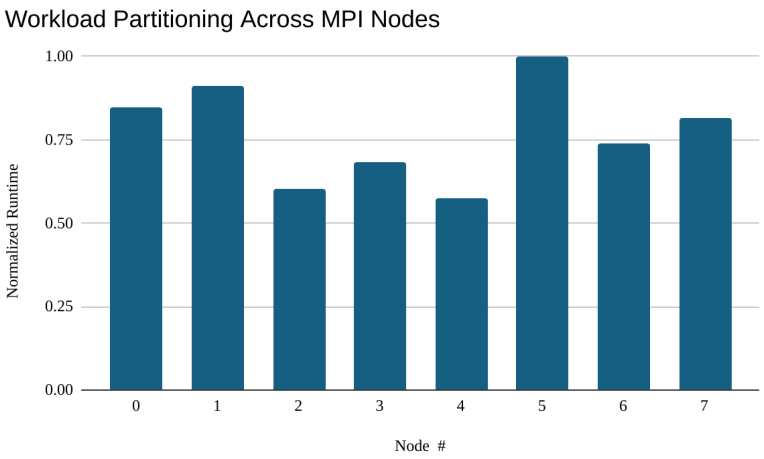

Figure 9. Individual node runtimes for our MPI approach, run on PSC machines with the GroupQueryAttention demo tensor program

Additionally, we further investigate why our MPI approach achieves significantly less speedup for low thread counts compared to our OpenMP approach. As the majority of the runtime for each node is spent on queries to the Z3 solver, we hypothesize that this observation is due to the lack of caching when splitting computation across MPI nodes. While the OpenMP approach has a shared map that prevents repetitive expensive queries to the Z3 solver, each node in MPI maintains its own independent map and doesn't have access to queries that other nodes run. To test this, we collect some additional metrics on PSC regarding the total number of cache misses across nodes and the average number of cache misses per node.

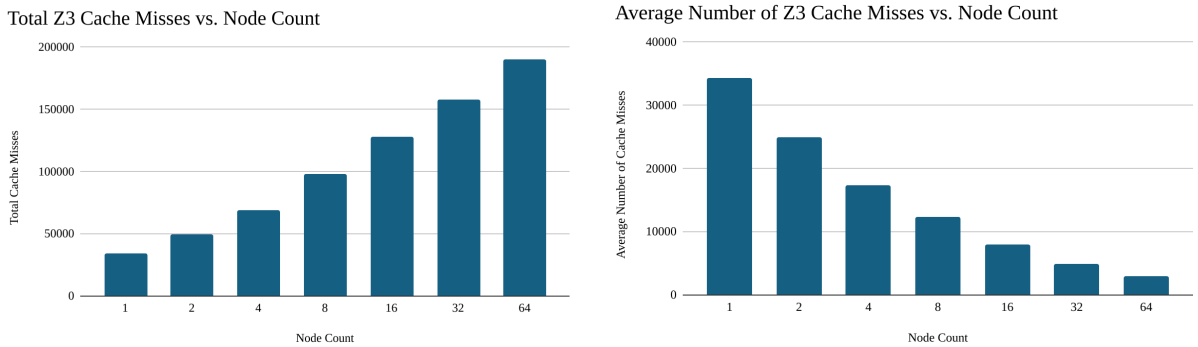

Figure 10. Total and average Z3 cache misses for different node counts, run on PSC machines with the GroupQueryAttention demo tensor program

These additional metrics confirm our suspicions, as we observe that doubling the number of nodes doesn't result in the number of Z3 queries being halved on each node. This results in the overall speedup of increasing the number of nodes to be lower than expected, as the total number of Z3 queries increases as well. Therefore, we see that the lack of Z3 cache sharing among nodes is a significant limit on the potential speedup.

Conclusion

In this project, we examined two major axes of performance bottlenecks for the Mirage tensor superoptimizer's parallel search process, and addressed each of them using different models of parallelization. We first identified that the static assignment method used in Mirage for per-thread workload distribution results in workload imbalance due to divergent execution, leading to threads assigned with more compute-heavy workloads bottlenecking the parallel search process. We addressed this by implementing a shared address space model which uses OpenMP to perform dynamic workload scheduling during the recursive search process. This results in improved workload balancing, which improved execution time by 1.1x and 1.5x across the examples tested.

We then identified major resource contention for the Z3 solver, particularly at higher thread counts. This causes the OpenMP approach to suffer a decrease in speedup at higher thread counts. We addressed this by implementing a message passing model using MPI. This effectively eliminates resource contention for the Z3 solver since every process now owns a unique copy of the z3 solver in their address space. We demonstrated that the MPI-based approach allows for scaling the parallelism to higher thread/process counts while maintaining an increase in speedup.

However, since we had to use the static assignment approach in the MPI-based implementation, the current MPI implementation also suffers from workload imbalance.

Future Directions

One future direction can be explored is how to implement dynamic scheduling inside an MPI-based framework. An example would be to, at each recursive call, send a copy of the current middle states to one of the other processes running concurrently. However, that would require a lot of expensive communications between processes, therefore while this approach might achieve better workload balancing, it also significantly increases the communication cost. Balancing communication cost and workload imbalance would be the major issue to look into. An alternative approach to achieving dynamic workload partitioning would be to first generate the middle states on each process, then the master process would send the index that each process should compute dynamically. While this still incurs more communications than the current static assignment approach (since we have to communicate the indexes), it is very likely that dynamic scheduling will improve the execution time by ensuring good workload balance.

Another future direction that can be explored is if combining OpenMP and MPI can yield better results than using any of them on their own. One approach would be to do MPI between nodes, but OpenMP within a node. MPI would reduce resource contention for the Z3 solver, but there are advantages in allowing a few threads to share one Z3 solver as well - Mirage's code that interfaces with the Z3 solver implements caching for intermediate results such that if a configuration is already calculated with the Z3 solver, the result is reused, bypassing the Z3 solver entirely. By allowing a few threads to use the same Z3 solver, we would reduce the overall

number of calls to the Z3 solver since if two or more threads called the Z3 solver with the same configuration, the Z3 solver would only be called once and the cached result will be used instead.

Additionally, it could also be possible to communicate cached Z3 results between nodes in the MPI approach. While the number of cache hits for Z3 queries quickly scales into the millions for the provided demos, the number of expensive Z3 queries remains relatively low. One possible approach is to organize the MPI nodes into a ring and communicate new cached results to neighboring nodes. By using a ring, we avoid broadcasting to all nodes which can become very expensive as the number of nodes increases. While it is still possible for nodes to compute the same cache result independently due to message latency, this communication scheme could reduce the number of Z3 queries on each node and improve the potential scaling of our MPI approach.

References

Wu, M., Cheng, X., Padon, O., & Jia, Z. (2024). A Multi-Level Superoptimizer for Tensor Programs. arXiv preprint arXiv:2405.05751.

Work Distribution

The credit distribution was roughly 50/50, with a more detailed breakdown of how work was split as follows.

Kit Ao: OpenMP approach, MPI approach, PSC testing, catalyst-cluster testing, Final Report

Joshua Hong: MPI approach, PSC testing, Poster, Final Report