Auto-Parallelization with Execution Runtime

ECE1754 Project Report

Lichen Liu

liuli15, 1001721498

issac.liu@mail.utoronto.ca

1.0 Abstract

This project implemented a compiler and execution runtime infrastructure to automatically parallelize the loops inside the target program. The compiler pass, written using the Rose compiler infrastructure, performs loop analysis to identify parallelizable loops, converts each iteration of the loop into execution runtime tasks and lastly inserts the execution runtime into the target code. On the execution runtime side, the project implemented various types of thread-pools that use different strategies to schedule how tasks are executed. The benchmark experiment shows after automatic parallelization, the program with an unbalanced workload pattern now yields a 5.68x of speedup using the work-stealing pool with 8 worker threads.

2.0 Introduction

In the case of a loop that must perform parallelism inside the inner layer of a nested loop structure, using the fork-join model incurs a ton of overhead due to the cost of pthread construction and destruction. A previous project [1] on optimizing an N-body simulation program demonstrated such an issue. The outer loop, which represents the system evolution in time, cannot be optimized via loop transformation, whereas the inner loops, which are doing the real computation such as calculating acceleration in the gravitational field and updating the velocity and position of the bodies after a timestamp, can be parallelized. In certain situations, it is not hard to see that a simple and trivial fork-join parallelism model may become very inefficient due to the non-negligible overheads incurred on pthreads itself, as it constructs threads, then runs tasks, and joins and destroys these threads every time parallelism is needed. If the workload under such parallelism runtime is not computationally heavy, then parallelizing the

loop may even reduce the performance, which was observed in the previous project where the performance was significantly slower than the one-threaded version. To address the overhead issue associated with trivial fork-join model implementation, an execution runtime that reuses the worker threads is needed.

The problem becomes more challenging when the algorithm of the acceleration calculation of the bodies in the gravitational field exhibits a triangular nested-loop pattern. If the iterations of the parallelizable loop are statically and uniformly distributed to the worker threads, then the unbalanced workload can cause a few worker threads to waste computational resources idling and waiting, while other worker threads are still busy computing. To address the issue, one simple solution is to use a thread-pool that supports dynamic load balancing to be the execution runtime for achieving the parallelism. The dynamic load balancing approach is both generic in balancing all kinds of workloads and without requiring knowledge of the target program.

Lastly, if the program has multiple loops that can be parallelized, the manual work of modifying the program to utilize the parallelization execution runtime can be tedious and disrupting. Thus, a framework that automatically identifies the parallelizable loops and applies the execution runtime which adapts to various types of workloads, enables the developers to focus on high-level algorithms while enjoying the performance gain from the modern multi-core architectures.

In this project, a compiler and runtime framework to automatically parallelize and execute loops is implemented. On the compiler side, the project utilizes the Rose compiler infrastructure to create a compiler pass to insert the execution runtime into the target program. Firstly, it analyzes the loops in the target program to determine where the parallelism can be performed. Then it turns the body of the parallelizable loops into tasks for thread-pool execution.

Lastly, it modifies the source code of the target program to insert the execution runtime. On the runtime side, the project provides various types of thread-pool for executing parallel tasks with a simple task execution interface. The execution runtime is a header-only library, thus allowing the compiler pass to easily integrate into the target program.

3.0 Methods

The project mainly consists of two parts, the execution runtime consists of several thread-pools implementations, and the compiler side involves analyzing loops that are parallelizable and later transforming the target code to parallelize the loops by inserting the execution runtime code. The project is divided into four stages.

The first stage is the execution runtime or the thread-pools. The thread-pool has a simple interface that executes a list of tasks and waits for all tasks to finish before returning. Three different thread-pools are implemented: a dynamic pool that supports dynamic load balancing by work-stealing, a static pool that assigns tasks to worker threads uniformly at launch time, and a single-worker pool that executes the tasks sequentially. The dynamic work-stealing pool is a modification of the receiver-initiated private deque implementation from the work of Acar et al. [2]. All pools implement the same invocation interface and are distributed as a header-only library.

The second stage is loop analysis for determining parallelizable loops. This stage involves using the Rose compiler infrastructure to analyze the abstract syntax tree (AST) of the target program. It firstly identifies dependency candidates that may prevent parallelization, and then gradually eliminates the dependency candidates. If no dependency remains at the end of the analysis, then the loop can be safely parallelized. As loop analysis is a complicated process, the

project borrows the code from autoParalleliztion (by Liao et al. [3]), which is a tool that automatically parallelizes the target code by adding OpenMP directives to parallelizable loops. Adaptations are made to the borrowed loop analysis to accommodate the different dependency rules that OpenMP and the execution runtime of this project support.

The third stage is to chunk up the parallelizable loops and convert each chunk into a task that can be executed by the execution runtime. In this project, the chunking is done at the iteration granularity, such that every single iteration is converted into a separate task. The conversion is done using a simple but powerful C++ feature, lambda expression, which captures both the statements and referenced variables into a closure object in place. The transformation here effectively delays the execution of the loop work, by capturing the work and sending it to the execution runtime. As a result, the execution runtime now has full control over how the loop work is executed. Using the C++ lambda expression feature requires minimal modifications to the original target code, while also ensuring the converted code is a fully valid C++ program by itself.

The last stage is code generation using the Rose compiler infrastructure. After the loop analysis has identified parallelizable loops, those loops need to be chunked up and converted into a list of tasks by capturing each loop iteration into C++ lambda expression. The list of tasks is then launched by the execution runtime. Code regarding the initialization of the execution runtime and corresponding header files also need to be inserted into the generated code. Due to the simplicity of C++ lambda expression syntax and the execution runtime interface, a text-based code generation approach is sufficient for the scope of this project. This avoids a lot of complexity in dealing with the AST manipulation if going through that path.

4.0 Implementation

4.1 Execution Runtime

Execution runtime, or the thread-pool, is responsible for launching the provided tasks and running them to completion. For blocking execution mode, the runtime is also responsible for ensuring that all tasks are completed before the execution runtime task invocation function returns to the caller. The project implemented a few different types of thread-pools, namely a dynamic pool that supports dynamic load balancing by work-stealing, a static pool that assigns tasks to worker threads uniformly at launch time, and a single-worker pool that executes the tasks sequentially. These different thread-pools share the same interface, which allows the thread-pool user to easily switch between different thread-pools.

A task is the fundamental carrier of a piece of work to be done by the thread-pool. It is defined to be std::function<void()>, which takes void as input and returns void as output (see Figure 1). Since it does not accept any inputs, it is recommended to use a lambda expression to represent a task, as the lambda expression can capture variables that identify the current task into the lambda expression body. Nevertheless, any function pointers or objects that have their function call operator (i.e., operator()) defined, can be used as a task as long as its function signature matches.

```
using RAW_TASK = std::function<void()>;
```

Figure 1. A task is any function that has no inputs and outputs

The thread-pool interface is an abstract class that declares the execute method to be purely virtual (see Figure 2). The different thread-pools are all required to implement this

method, which takes in a vector of tasks and runs them to completion before returning the control to the caller. The interface also declares a few other optional methods, such as start and terminate, which allows the caller to properly initialize and start the thread-pool, and properly terminate it when all tasks are done.

```
class POOL
{
public:
    explicit POOL(size_t num_workers) : num_workers_(num_workers) {}
    virtual ~POOL() = default;

    virtual void start() {}
    virtual void terminate() {}
    // A single session of execution, blocking until completed
    virtual void execute(const std::vector<RAW_TASK> &tasks) = 0;
    virtual void status() const {}

    size_t num_workers() const { return this->num_workers_; }

private:
    size_t num_workers_;
};
```

Figure 2. Pool interface is implemented by all thread-pools

4.1.1 Work Stealing Private Deque (WSPDR) Pool

The dynamic work-stealing pool, or Work Stealing Private Deque (WSPDR) pool, is the receiver-initiated variant from the work of Acar et al. [2]. It achieves dynamic load balancing by work-stealing, where an idle worker thread steals tasks from a busy worker thread at runtime. Furthermore, this thread-pool achieves work-stealing by passing messages between worker threads. This allows each worker thread to keep their task queue private to themselves, thus avoiding using expensive mutex locks to directly operate on each other's task queues. This

approach greatly improves efficiency due to the relaxing requirement on memory fences on certain weak memory model modern architectures.

The thread-pool consists of a user-specified number of worker threads, which is typically the number of processors of the current system. Each worker follows the receiver-initiated algorithm described in the paper [2], where the task request is initiated by the idle worker and sent to another randomly-chosen worker. At a high level, each worker maintains a private deque for storing tasks: the bottom end of the deque is for adding new tasks and popping tasks for execution, whereas the top end of the deque is for distributing tasks to other workers.

Every worker is in an event loop that constantly checks whether it still has tasks in its queue (see Figure 3). If not, it terminates the event loop if it is requested to do so; otherwise, it attempts to find a target worker to steal work from, by randomly picking a target worker thread that still has tasks in its task queue. It automatically sends its ID to the target worker only if that worker has not been requested by another worker yet. Then only if the request is successfully sent to the target worker, the current worker would wait for a response from the target worker in a blocking fashion, by constantly polling its task receiving queue to see whether it has been replied to, and would then add the received tasks to its task queue. It is interesting to note that the target worker could reply with the request with no tasks, which implies that the target worker no longer has any distributable tasks at the moment it processes the request. In such a case, the current worker simply continues into the next iteration of the event loop.

On the other hand, if the current worker has tasks in its task queue, it first pops a task and executes it. Then it checks for any incoming task request and continues to the next event loop iteration if there is none. In this design, all workers must always respond to all incoming requests regardless of whether it has tasks in the queue. It has to be done as soon as possible, even when

waiting for the response of its own request, to avoid blocking the other worker that sent the request. In the case that the current worker has no more tasks to offer, the response would be a null task; otherwise, the worker pops out some tasks from its queue and sends them to the requesting worker. The project implemented two policies of distributing tasks, namely steal-one and steal-half, where steal-one sends out one task per response whereas steal-half sends out half of the tasks in the queue.

There are a few special designs in this thread-pool that are not presented in the original paper, such as proper worker shutdown and task creations. As workers are in an event loop constantly stealing tasks from other workers and obligated to respond to other workers' requests, special attention is made to ensure the termination sequence would not ignore requests and skip executing tasks. On the other hand, due to the non-locking design, task creation is also prone to race conditions. There are two ways tasks can be added, namely by non-worker threads and by worker threads. In the case of task creation by a non-worker thread, there is a restriction that only a single task can be sent to the entire thread-pool, and the entire thread-pool must have no tasks at the moment. Whereas task creation by worker threads is achieved by slightly modifying the function signature of a task to provide a proxy object for the caller to append new tasks into. These new tasks are later properly inserted into the current worker's task queue.

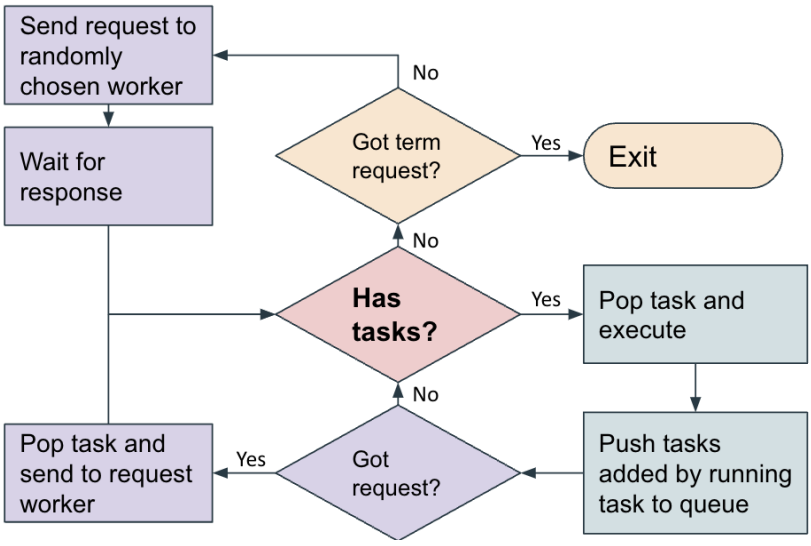


Figure 3. Event-loop of the WSPDR Pool worker

Upon task launching, the thread-pool manager is responsible for properly sending all tasks to the workers, and providing synchronization to wait for all tasks to finish. Every user-provided task is wrapped into another task lambda closure with an atomic counter decrementing after the original task is done. As such, synchronization can be achieved by waiting for this counter to become zero. The thread-pool manager launches the task by creating and sending a master task which inserts all user tasks into the worker whoever executes this task. This is done to satisfy the constraint that only one task can be added to the thread-pool from a non-worker thread.

4.1.2 Statically and Uniformly Assigned Private (SUAP) Pool

The static multithreaded pool, or Statically and Uniformly Assigned Private (SUAP) pool also contains a user-specified number of worker threads for executing tasks, but this time, all tasks are pre-assigned to the worker thread at launch time. In other words, there is no task redistribution at runtime. The thread-pool manager statically and uniformly splits the input list of tasks into many sublists of tasks, then sends the sublist of tasks to each worker, and waits for all

workers to finish (see Figure 4). The worker thread uses a simple concurrent queue with a size of one to receive task from the thread-pool manager.

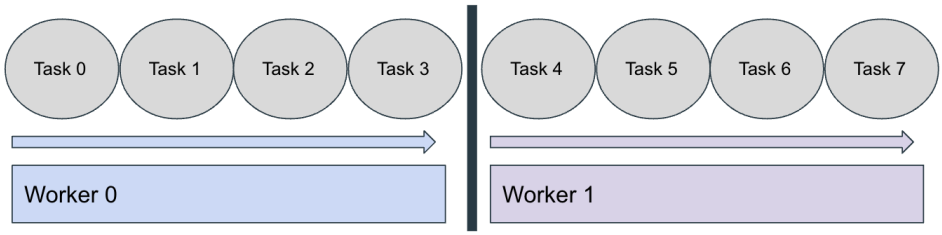


Figure 4. Task execution mode of the SUAP Pool

4.1.3 Serial Pool

A simple serial thread-pool is implemented for measuring the single-threaded performance of the thread-pool interface. Input tasks are simply executed serially in the thread-pool manager thread upon invocation of the thread-pool execute method.

4.2 Loop Analysis

Loop analysis is the key part of the compiler pass to identify parallelizable loops. However, due to the complexity involved in the loop analysis process, code is borrowed from autoParallelization, which is the work of Liao et al. [3]. It is part of Rose compiler's demo project, located at rose/projects/autoParallelization, and is a tool that automatically parallelizes the target code by adding OpenMP directives to parallelizable loops.

The loop analysis uses the following algorithm to determine whether a loop can be parallelized using OpenMP runtime. It firstly does some preprocessing such as loop normalization and constant folding to simplify the loop structure. It also checks and skips semantics that is not supported yet, or function calls with unknown side effects. Then it performs

the dependence analysis and liveness analysis. Based on the liveness analysis result, shared variables are classified into auto-scoped variables such as private, firstprivate, lastprivate, or reduction, which are required to be labelled in the OpenMP directives. After that, dependencies such as the auto-scoped variables, local variables and order-independent write accesses are eliminated. Loops without any dependencies left are indeed parallelizable OpenMP loops, which are annotated with the OpenMP directives. Lastly, it undoes the normalization to minimize changes to the original code. The details of the loop analysis algorithm can be found in the paper [3].

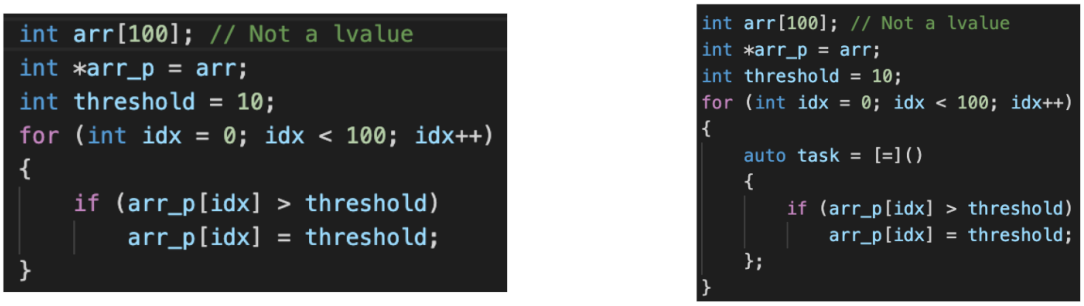
However, due to the difference in the execution runtime the borrowed loop analysis is targeting, a few modifications are made to fit the execution runtime employed in this project. Firstly, OpenMP allows lastprivate and reduction type of variable sharing, whereas the execution runtime in this project does not, due to the lack of supporting write-access for non-array types of variable. Secondly, OpenMP is capable of handling directives of parallel for-loops in a nested fashion, whereas the current execution runtime cannot. Therefore candidate parallelizable loops are filtered to only parallelize the outer loop when both inner and outer loops are found to be parallelizable. Lastly, the OpenMP directives generation stage is replaced by code generation logic to create tasks from parallelizable loops and then pass them to the execution runtime.

4.3 Task Creation

After a loop is determined to be parallelizable by the loop analysis stage, the next step is to convert the loop body into tasks that can be launched onto the execution runtime. To do so, a loop chunking scheme needs to be decided first. There are multiple ways that the loop work can be divided into tasks, such as grouping a number of iterations into fixed-sized tasks or even

creating tasks of different sizes by assigning different numbers of loop iterations to different task-size groups. In this project, the loop chunking is done at the iteration granularity. In other words, each iteration of a loop is created as an individual task, and the total number of tasks created for that loop equals the iteration count of that loop. This scheme is chosen for its simplicity which fits the scope of the project, and also serves as a benchmark reference for future exploitation of different loop chunking schemes.

Transforming the loop iteration code into a task is done by using the C++ lambda expression, where this powerful C++ language feature can capture both the statements within the loop body and also the shared variables that are referenced in the loop body (see Figure 5). By capturing the loop body statements into the lambda expression task, the execution of these statements is postponed to when the task is being invoked, which is the most crucial code transformation that seamlessly moves the execution of the loop of the original target program to the execution runtime. After the transformation, the statements inside the original loop body now reside inside the lambda expression body. As the execution runtime only supports shared variable types of private and firstprivate, the lambda expression can simply capture them by value (i.e., [=]) in the lambda capture list. These types of shared variables are declared and initialized before the loop iteration and may be modified by the loop iteration. Thus they need to be copied into the lambda expression body to avoid side effects. On the other hand, as an exception, array-like variables should be captured into the lambda expression by reference since all dependencies associated have been eliminated by the loop analysis. However, as the project now only supports C-style arrays which use pointer type as the array variable, capture-by-value behaviour is sufficient.



```
int arr[100]; // Not a lvalue
int *arr_p = arr;
int threshold = 10;
for (int idx = 0; idx < 100; idx++)
{
    if (arr_p[idx] > threshold)
        arr_p[idx] = threshold;
}
```

```
int arr[100]; // Not a lvalue
int *arr_p = arr;
int threshold = 10;
for (int idx = 0; idx < 100; idx++)
{
    auto task = [=]()
    {
        if (arr_p[idx] > threshold)
            arr_p[idx] = threshold;
    };
}
```

Figure 5. Example of how each iteration of the loop body is converted into a C++ lambda expression

4.4 Code Generation

The last stage of the project is to use the Rose compiler infrastructure to integrate the execution runtime into the target program. Code generation involves transforming each iteration of the loop into a task, where each loop iteration is converted to a lambda expression with variables captured by value. The original loop now becomes a loop to create tasks and insert the task into a vector. Then the list of tasks is passed into the execution runtime. The execution runtime also needs to be initialized in all functions that use the execution runtime. Lastly, the corresponding header file that defines the execution runtime is included in the target program.

All above code modifications are achieved using a text-based approach in Rose, as opposed to directly modifying the AST of the target program. Although the AST modification approach is more robust and systematic, it does have a few unexpected complexities, especially for creating lambda expressions. Whereas the text-based approach is capable of transforming the code also with good quality by fully leveraging the simple-to-use execution runtime interface and the similar C++ syntax between a loop body scope and the lambda expression body. The

code generation uses the Rose API SageInterface::addTextForUnparse function. Figure 6 shows an example of the generated code.

```
void foo()
{
    int n = 100, m = 100;
    double b[n][m];
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            b[i][j] = b[i - 1][j - 1];
}
```

```
#include "wspdr_pool.hpp"

void foo()
{
    ERT::WSPDR_POOL __apert_ert_pool(4);
    __apert_ert_pool.start();
    {
        int n = 100;
        int m = 100;
        double b[n][m];
        for (int i = 0; i <= n - 1; i += 1)
        {
            {
                std::vector<ERT::RAW_TASK> __apert_ert_tasks;
                /* ================ APERT ================ */
                for (int j = 0; j <= m - 1; j += 1)
                {
                    auto __apert_ert_task = [=]()
                    {
                        b[i][j] = b[i - 1][j - 1];
                    };
                    __apert_ert_tasks.emplace_back(std::move(__apert_ert_task));
                }
                __apert_ert_pool.execute(std::move(__apert_ert_tasks));
            }
        }
    }
}
```

Figure 6. Example of how generated code (right) transforms the parallelizable loop from the source code (left) to use the execution runtime

5.0 Evaluation

5.1 Benchmark Setup

Several benchmark kernels are implemented to measure the performance gain from the execution runtime alone and as well as from the entire automatic parallelization framework. Four benchmark kernels are implemented, namely sorting, matvec_mul, calc_grav_acc and n_body_sim. The first three kernels all share unbalanced workload behaviours inside the loop. Sorting and matvec_mul are loops that use bubble sort and matrix-vector multiplication that operates on input size that depends on the current iteration index to simulate workload; whereas

calc_grav_acc is a kernel that computes the acceleration for bodies in the gravitational field using nested triangular for-loops. On the other hand, n_body_sim is a program that simulates the n-body problem across several timestamps and does not exhibit any unbalanced workload behaviours.

The performance benchmarking is done on the ECF remote server, with two sockets of Intel(R) Xeon(R) Gold 6230 CPU @2.10GHz 20 cores and 40 threads processors, and 128 GB RAM. Programs are compiled using gcc 8.5.0, with -O3 optimization. As the server is a shared environment, in addition to running the benchmarking experiments while the system has almost minimum load, each experiment is also run three times, and the geomean speedup is taken as the result.

5.2 Performance of Different Thread-pools

Thread-pools	sorting	matvec_mul	calc_grav_acc
Static	2.19	1.64	2.20
Dynamic - Steal One	1.29	1.07	1.92
Dynamic - Steal Half	3.61	3.22	3.65

Table 1. Speedup of different thread-pools on the three benchmark kernel programs

The performance of each thread-pool is measured first. The benchmark kernels are manually inserted with various types of thread-pools all configured to use four worker threads, namely the WSPDR pool (or dynamic pool in short), SUAP pool (or static pool) and serial pool. In particular, the dynamic pool is tested with two different policies of work-stealing, steal-one

and steal-half. Speedup is measured against the wallclock time of the execution on the serial pool.

The kernels used in this benchmark experiment all have dynamic workload characteristics and are all created with a number of tasks much larger than the number of worker threads. Table 1 shows the speedup of different thread-pools on the benchmark kernel programs. As expected, the dynamic pool with steal-half gives the best speedup in all three testing kernels. This geomean of 3.49x of speedup is very close to the upper bound of 4x speedup, even under the unbalanced workload scenario. For the static pool, the average speedup is only 2.0x, mainly because the entire execution needs to wait for the slowest worker thread to finish the work, even when the rest of the worker threads have already been idling. The dynamic pool with the steal-one policy gives the worst performance of only 1.38x of speedup, and matvec_mul shows almost no speedup. This huge performance gap with the steal-half policy is mainly due to the task launching mechanism, where all tasks are initially launched to only one worker while the steal-one policy is too slow in transferring tasks to other workers, effectively starving the rest of the thread-pool.

5.3 Performance of Generated Code

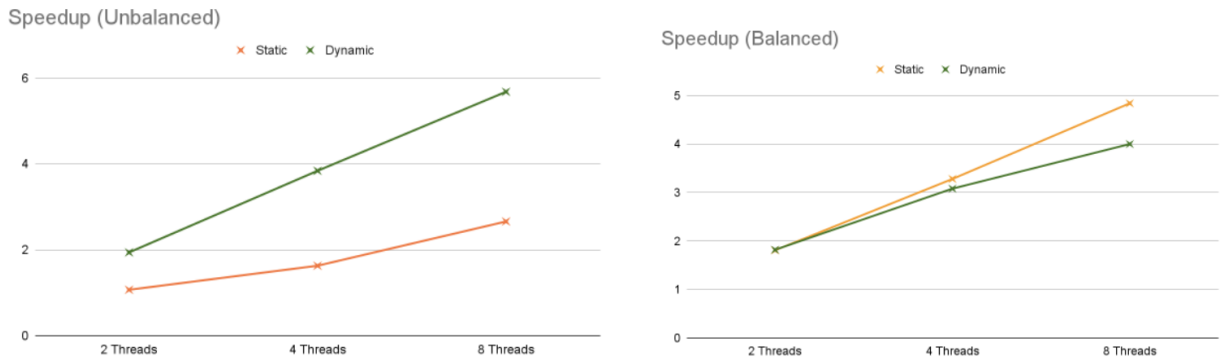


Figure 7. Speedup of unbalanced (left) and balanced (right) kernel programs after automatic parallelization using dynamic and static thread-pools

The performance of the benchmark kernels that have been undergone through the entire automatic parallelization pass are also measured. The compiler pass automatically identifies the parallelizable loops and converts the loop to run with WSPDR pool (or dynamic pool) and SUAP pool (or static pool) with 2, 4 and 8 worker threads. The performance is compared against the result from using the serial pool.

Benchmark kernels are classified into two categories by nature of the workload, balanced and unbalanced. Figure 7 shows the results of the experiment. When under the unbalanced workload, the dynamic pool outperforms the static pool under all worker threads configurations by almost 2x. The performance of the dynamic pool scales reasonably well as the number of worker threads increases and achieves 5.68x speedup with 8 worker threads, while the static pool is only able to speed up by 2.66x. On the other hand, when under the balanced workload, the dynamic pool no longer has the lead. Both pools perform equally well with 2 threads, but the performance gap starts to increase as the number of workers increases, with the static pool

performing the best with 8 workers with a speedup of 4.84x. This result suggests that there is a non-negligible overhead associated with the work-stealing-based pool, and it is only beneficial for the unbalanced workload.

6.0 Discussion

Currently, the automatic parallelization framework only supports a few types of semantics due to the limitations of the loop analysis and the text-based code generation. For example, only C-style arrays are supported for parallelization, but not complex C++ data structures that behave like arrays, such as std::vector. To support it, the type annotation system needs to be properly configured for the loop analysis, and task generation needs to capture the array-like variables into the lambda expression by reference. In addition, loops that involve function calls are also not parallelizable due to the lack of the side-effect annotation system integration.

The execution scheme for the parallel loop is also very preliminary now. The loop is only chunked statically at the per-iteration level, and therefore the overhead of task creation and task scheduling may potentially hurt the performance. Secondly, the thread-pool used as the execution runtime is manually selected by the user. As results from the previous section have shown, there is a performance gap between thread-pools that fit the current type of workload and those that do not fit it. Therefore, the thread-pool selection should incorporate knowledge from the workload analysis by both compile-time and runtime.

In addition to the above limitations, there are some possible future work ideas to further improve the project. The code generation of the project should insert the execution runtime code via AST manipulation, which would open up space for more complicated execution runtime

invocation opportunities. Nested parallelizable loops should also be handled more smartly, as opposed to the current approach of only parallelizing the outermost loop.

7.0 Conclusion

In this project, a compiler and execution runtime infrastructure is implemented to help developers automatically parallelize the loops inside their program. The project implemented a compiler pass that uses the Rose compiler infrastructure. The pass analyzes loops to determine parallelizable ones, then chunks the loop up at iteration granularity to convert the loop body into tasks for running on the execution runtime, and lastly inserts the execution runtime invocation code. On the execution runtime side, the project implemented various types of thread-pools that use different strategies to schedule how tasks are executed, such as a pool that supports dynamic load balancing by work-stealing, and a pool that statically and uniformly assigns tasks to workers at launch time. The benchmark experiment shows that the program generated by the compiler pass based on a program that exhibits unbalanced workload patterns yields 5.68x of speedup using the work-stealing pool with 8 worker threads.

8.0 Project Code

The code of the project can be found on GitHub: https://github.com/lichen-liu/apert.

9.0 References

[1] Ray Huang, Lichen Liu, Sining Qin, Haiqi Xu, Tiny Universe Simulator System (TUSS), (2021), GitHub repository, https://github.com/qsnsidney/tuss

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. SIGPLAN Not. 48, 8 (August 2013), 219–228. DOI:https://doi.org/10.1145/2517327.2442538

[3] Liao, C., Quinlan, D.J., Willcock, J.J. et al. Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions. Int J Parallel Prog 38, 361–378 (2010). https://doi.org/10.1007/s10766-010-0139-0