Project Report on,

Parallel Continuation Passing Style

as a part of

Concurrent Programming Course

By,
S Pradeep Kumar (CS09B021)
Mahesh Gupta (CS12M018)

Guided by,
Dr Shankar Balachandran

1. Problem Specification

1.1 Area

Languages and Compilers

1.2 Problem Statement

Given a Program in Continuation Passing Style (CPS), to convert it into Parallel Continuation Passing Style (pCPS) form, consisting of set of tasks, and given their dependency specified in terms of a directed graph and a scheduler, which executes the tasks to achieve in parallel if they are independent of each other and hence achieves parallelism if multi-core processor is used.

Hence, given a code executable in single core processor, to convert it into correct program executable in multi-core processor.

1.3 Why it is interesting ?

The Intermediate Representation for most compilers is either Static Single Assignment (SSA) or Continuation-Passing Style (CPS). Neither have any default support for parallelism.

By automatically converting CPS code into pCPS, we can get faster executable programs without manually parallelizing the code.

2. Parallel CPS Codes

2.1 Opportunity of Parallelism:

Consider a code written in Continuation-passing Style as

```
int fib(int k) {
      if( k <= 2)
         return 1;
      else
         return fib(k-1) + fib(k-2);
}
```

The above code can be written as

```
int fib(int k) {
      if( k <= 2)
         return 1;
      else
         int f1 = fib(k-1);
         int f2 = fib(k-2);
         return f1 + f2;
}
```

Here, we can see that the code has two function calls, whose computation are independent of each other. Hence, instead of second function call waiting for first to finish, we can execute both the functions in parallel and once their results are obtained we can execute the return statement.

2.2 Concurrency constructs:

As per our analysis, we have developed certain constructs which can help in achieving concurrency and to define the dependency among tasks. These constructs are as follows.

2.2.1 Task:

Consider, each piece of code which is to be executed in parallel as a task to be executed. For example, in the code above, each method call can be considered as a task to be executed.

Each task can be represented as a vertex in the graph.

2.2.2 Activation:

Activation construct acts as a wrapper for the task and helps to define what to execute next after the task has been finished. It also helps to define the dependency among the tasks. For example, if summation is also a task then summation task must wait for the previous two tasks to be completed before it starts its execution. The dependency among tasks is also define in terms of separate constructs.

2.2.3 Happens Before Edge:

As, task have dependency among themselves, it is represented as happens before edge. Consider two tasks left and right and a relation,

left → right

which means that right can execute only after left has been executed. Even though certain part of right might be executed immediately but that would result in right waiting for the left to finish and hence unnecessarily waiting. Thus, right should start executing only after left is done with execution.

It is represented as an edge in the directed graph edge in the given graph.

2.3 Generated Code Looks

So, after applying all the above constructs our generated code would look something like this,

```
int fib (int k){
    if(k <=2 )
        now.result = 1;
    else {
        Activation left = schedule(fib(k-1));
        Activation right = schedule(fib(k-2));
        Activation sum = schedule (
          { now.left = left.result + right.result;
           });
        left → sum;
        right → sum;
   }
}
```

The code conversion from cps to parallel cps can be automated. As

far as the naïve dependency is concerned, we assume that all the function calls happen sequentially, I.e., one function call happens-before the other. At compile-time, we optimize the happens-before schedule using information which would be given by static analysis.

3. Work done during Project

During the work of project, we tried several approaches to see which approach leads to less over head in terms of resource usage and leads to optimum code in terms of memory usage. We spent considerable amount of time doing experiments with various other resources.

3.1 Tasks with Blocking Queue for Data Passing feature:

Initially we started with data passing feature where all the tasks are started simultaneously and wherever the data dependency exists task 2 waits for task1 to send the data. For implementing this feature we were using LinkedBlockingQueue class which blocks the request of consumer until producer has put the data. Each set of tasks with dependency have a queue among themselves.

For smaller set of data, code was performing well, but as the data size increases, we observed that there were so many threads scheduled and were waiting for other thread to finish their working. Which leads to poor performance in terms of memory usage. Also too many threads for a sufficiently large input, causes runtime error to be thrown. For fibonacci series as input size becomes > 1000, the code starts to perform badly.

Hence, we drooped this idea and started to consider the option where a task is scheduled only when there is no other task on which it is dependent on.

3.2 Experiments with ThreadExecutor and doing the Performance Evaluation:

Later, while implementing the thread scheduler, one of the key task was implementing the thread executor. When a task becomes ready for execution, instead of immediately executing it, its execution is deferred by putting into ThreadExecutor. Java provides two inbuilt ThreadExecutors i.e. FixedThreadPoolExecutor and CachedThreadPoolExecutor.

CachedThreadPoolExecutor though in some cases gives good performance but as the input size increases it tends to produce too many threads which are difficult to handle by the system and sometimes causes system to crash as no new threads can be scheduled. Also over a long input

the performance is badly hampered.

3.3 Experiments with schedule optimization:

As the task and their dependency needs to be considered and since the graph is generated at compile time and might have certain redundancies in it, care must be taken to optimize the schedule. The redundant edges must be removed from the graph. Various experiments were done to optimize the schedule including doing Breadth first traversal of the graph, finding single source shortest path, finding longest paths and other things to see which one takes care of redundant edges.

4. Implementation

4.1 Language & Library

We have done all the implementations in java.

- JgraphT for representing the graph structure.
  - This library is not threadSafe, so we have externally tried to put the synchronization while performing the operations.
- Junit for writing test cases.
  - We have written test cases to check the behavior of various components of the system such as scheduler.
  - We also have written the test cases to check the behavior of various test programs and their output.
- Ant build script for building the code

5. Parallel CPS

5.1 Brief Overview of Parallel CPS

The basic idea behind Parallel CPS is that we execute as many method calls as we can in parallel, respecting any happensBefore constraints between them.

Here, we use Continuations to wrap the method call in a context along with other variables it might need so that it can be run in a separate thread.

5.2 Implementation of the Parallel CPS Scheduler

5.2.1 The Schedule

First of all, each method in pCPS comes with a schedule formed by the happensBefore constraints specified at the end of the method. At the time of execution, we represent this schedule as a Graph of Activations (which are basically runtime representations of tasks).

For this purpose, we have used the excellent JGraphT which comes with umpteen graph classes and assorted helper methods.

Note, however, that the graphs provided by JGraphT aren't synchronized. They will throw ConcurrentModificationException as soon as another thread even iterates through the vertices, etc. So, we have to manually synchronize access to the graph when adding or removing tasks and edges.

5.2.2 Ready Tasks

Ready tasks are basically tasks that don't have any happensBefore constraints. They are ready to be scheduled. In the directed graph of constraints (i. e., our schedule), ready nodes are those with in-degree = 0

5.2.3 Listenable Directed Graph

To make it faster for us to getReadyNode() and avoid unnecessary synchronization, we use a Concurrent Set (ConcurrentSkipListMap) to keep track of all the ready nodes available at the moment. The set of ready nodes will be updated whenever an edge is removed from the graph (and the node's in-degree becomes 0).

To this end, we use a ListenableDirectedGraph so that we can add a Listener (our ReadyNodeListener) to look out for fresh ready nodes. This way our ready nodes set is always up-to-date.

5.2.4 Running Tasks

The Scheduler waits till ready nodes are available and then passes one on to a thread pool (ExecutorService here) to be run in a different thread. It will remove that node from the ready nodes set, but it won't remove it from the graph of activations. Otherwise, nodes that must happen AFTER the current node would now seem ready to the Scheduler which might schedule them, thereby breaking the happensBefore constraint.

It then again loops and tries to schedule other ready nodes. Else, it waits on a condition variable for the availability of ready nodes. This way, we avoid spin-waiting.

5.2.5 Thread Pool

We use ExecutorService to get convenient access to a thread pool without worrying about the gory details of number of threads, or spawning threads, or re-using old ones. It takes care of all that for us.

5.2.6 Finished Tasks

When a task finishes, it removes itself from the graph. The Scheduler will automatically get to know of this through the ready nodes set which will now contain any nodes which have become ready as a result of this task completion.

5.2.7 Last Task

As a special case, the Last Activation to run will tell the Scheduler that there are no more tasks to be scheduled and it can shut down.

5.3 Other Aspects of the Project

We have made use of the JUnit testing framework for Java to ensure that our incremental changes to the Scheduler don't break the functionality. This assures us that the final output of the parallel execution is still the right one.

Whenever we add new pCPS programs, we add corresponding unit

tests for them to ensure correctness.

Also, we have added asserts for preconditions and postconditions of important functions to catch subtle errors due to concurrent execution.

6. Test Cases

Our test was focused mainly towards writing test cases and see how programs are performing for them. We have written test cases which cover all possible test cases i. e. partial parallelism, total parallelism and no parallelism.

No Parallelism: These are cases where the tasks generated have dependcy between themselves so that second task cannot be executed until first task has been executed. In such case, the parallel execution is not better than sequential code.

For this case, we executed Prefix Sum.

Partial Parallelism: Usually tasks have partial dependency between them and hence they need to wait for other one to finish. Hence executing such task becomes interesting as by optimizing the scheduling, such tasks can be executed more efficiently than others.

For this case, we executed Quick Sort program.

Total Parallelism: The tasks generated, if have no dependency among themselves then each task can be executed in parallel so that full utilization of multi-core system can be done. For such system, as no of processors increases the parallelism also increases.

For this case, we executed used Map function.

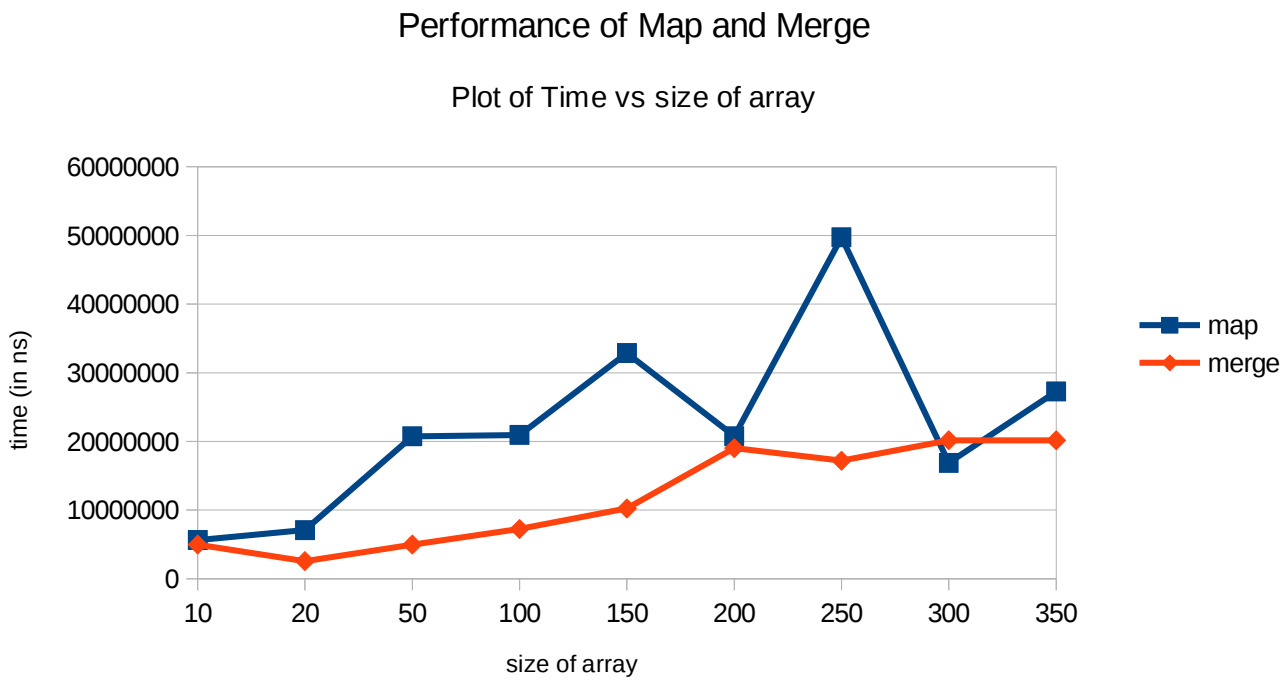
8. Test Results

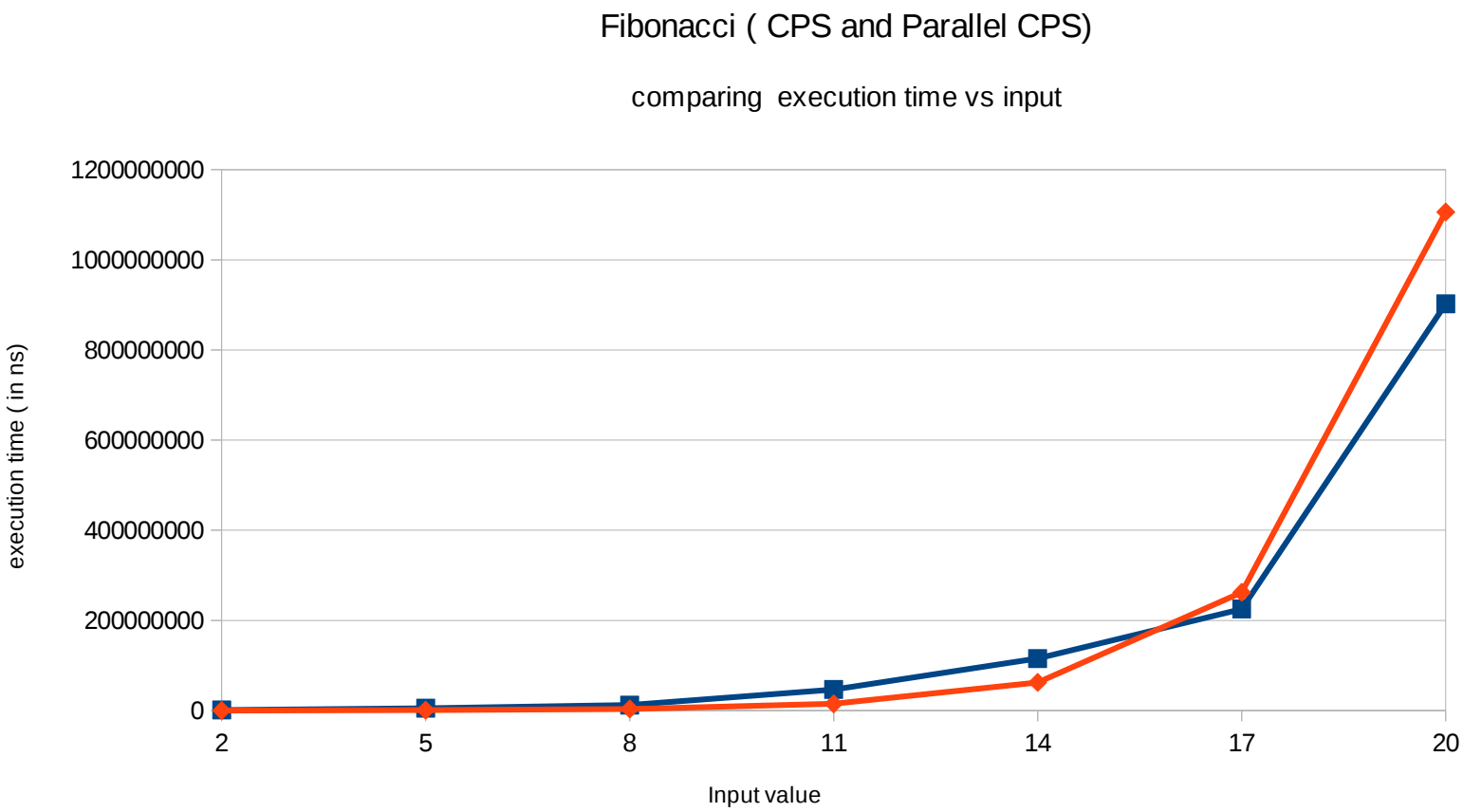
8.1 Test Results Obtained.

For the test cases mentioned above we executed the code to see how the p-CPS code performs against the CPS code from which it was generated. For the test cases we tried to see the performance regarding various aspects. Various aspects considered are:

- Input size vs Execution time
- Comparing Execution time of linear vs Non linear execution schedule.
- Comparison of following
  - Sequential code
  - CPS Code
  - pCPS Code

Results are follows





8.2 Explanation for Negative Test Results

Despite implementing the basic ideas in Parallel CPS paper faithfully, our pCPS programs currently take orders of magnitude more time than the equivalent sequential versions.

Here are a few of our experiments regarding the above and our inferences

8.2.1 Runtime Scheduler

In order to tease out the reason behind the slow code, we removed all the concurrency constructs in the Scheduler code till we were basically running the given code in a sequential manner, only via the Scheduler.

Our results were that the runtime had hardly changed despite removing all the **synchronized** blocks and Concurrent Sets, which we had suspected of being the limiting factor.

The foremost reason for the slowdown, in our opinion, is that all the code in the given pCPS program is being run through the Scheduler, which incurs heavy penalties. Ideally, the Scheduler would be a highly optimized part of the JVM, but because we are dealing with objects and library code at the user level, the resulting code is slowed down no matter what.

In fact, this is the intention of the original authors. They planned to bake the Scheduler into the JVM and so our Scheduler could be considered a proof of concept implementation.

8.2.2 Concurrency Constructs

Another reason for the slowness could also be constructs like **synchronized**, condition variables, concurrent set, etc.

Neither the sequential code nor the CPS code have any of this overhead and so they run much faster.

8. Conclusion

During the course of project we used the concept of parallel continuation passing style. Various experiments performed to enhance the concurrency in the system. We realized that concurrency is improved only when a part of code whose execution takes considerable amount of time is executed in parallel. Thus, we focused mainly towards doing loop execution and method invocation in parallel. Even these constructs would

be better if the new task created does significant amount of work. Otherwise creating new task does not improve the performance.

Various constructs which are not available in Java such as representing task execution as an entity rather creating a full fledged object would have improved the performance. Moreover, Scheduler should have be implemented at JVM level rather than as an application code, this would have given huge improvement for runtime code optimization.

Also since number of processors in the system are limited, creating number of threads beyond certain number of point will not give any better improvement. Hence the parallelism should be limited by keeping tracks of how many threads have been created and after certain threshold each task should have do sequential execution. This would also mean that each task is doing significant amount of work. So, parallelism can be improved by optimizing over number of threads issue.