Static Parallel Task Scheduling for Deterministic Discrete-Event Simulations in Lingua Franca

Yang Huang, John Fang, Shaokai Lin

May 2022

1 Introduction

Discrete-event simulation plays a major role in modeling the behavior of discrete-event systems, which covers many applications and domains such as cyber-physical systems (CPS) and Conway's Game of Life. In the past semester, though many parallel languages and frameworks are introduced, the interactions between threads/actors, however, if not constrained, admit nondeterminism. In this project, we aim to bring more parallelism to a framework called Lingua Franca (LF) [13], which is suitable for discrete-event simulation, and its underlying *reactors* model of computation, a recently introduced framework for building computation models that are deterministic and concurrent by construction [10]. We aim to build a framework which has exploitable parallelism, executing on multi-threads without compromising determinacy. Based on this, we bring up a **quasi-static scheduler** which implements a static schedule of parallel tasks, as static schedules can be inferred from the program topology and be partitioned among worker threads for Lingua Franca. This static parallel task scheduler possesses these advantages on parallelism according to our theoretical analysis, some of which have been verified by our experiments:

- Parallelism: the scheduler can generate very efficient parallel schedules at compile time.
- Less synchronization overhead: a worker thread could potentially stick to its static schedule at a logical instant without talking to other threads.
- Less data movement: the static schedule can try to promote spatial and temporal locality.
- Analyzability: the order in which reactions are processed is determined at compile-time, making the real-time behavior more analyzable and verifiable.

2 Background

2.1 Reactors

The *reactors* model [12] is a deterministic, concurrent, reactive, and timing-aware model of computation. The model has roots in classical models including actors [7], dataflow [6, 3], process networks [2], and synchronous languages [5, 4].

In the reactors model, stateful objects are represented by individual *reactors*, each of which contains its own state variables and message handlers. A reactor has *ports*, with input ports receiving data from the outside and output ports sending data to objects outside of the reactor. A reactor "reacts" to external inputs by invoking its *reactions* which are sensitive to specific triggers, including ports and actions. When two reactions in the same reactor are triggered simultaneously, the order of invocation is determined by their priorities. The earlier an reaction is declared, the higher priority the reaction has. An action is a trigger scheduled by a reaction, and it is used to invoke multiple reactions inside the same reactor (potentially at different time instants). An action can be *logical*, which is scheduled by a reaction, or *physical*, which is scheduled asynchronously by the external environment. There are also special triggers, *startup* and *shutdown*, provided by the runtime environment to be present at the beginning of the system and at the end of the execution. A reactor a can connect to another reactor b by drawing a connection between an output port of a and an input port of b. A connection by default transmits signals from one end to the other logically instantaneously, unless logical delays are specified. A reactor can also contain nested reactors.

Lingua Franca (LF) [13] is a polyglot coordination language that implements the reactors model. In LF, the body of a reaction is written in a target language, such as C/C++, Python, Typescript, or Rust. As a running example, we use an LF implementation of the classic Sleeping Barber Problem [1] (shown in Fig. 1) to demonstrate the above concepts.

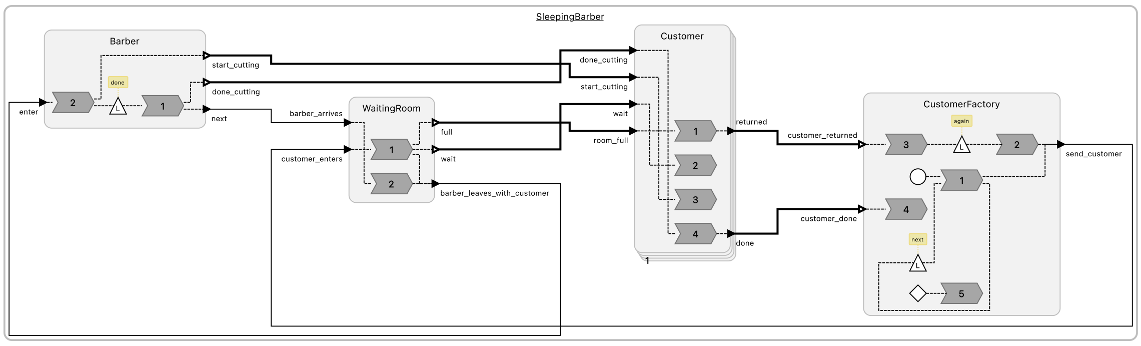


Figure 1: An LF implementation of the Sleeping Barber Problem.

In this program, there are four types of reactors - `Barber`, `WaitingRoom`, `Customer`, and `CustomerFactory` - each rendered as a gray box with rounded corners. The reactions are rendered as dark gray chevron shapes with numbers denoting their priorities. Ports are rendered as black solid triangles. Logical actions are rendered as white triangles with the letter "L" inside. The white circle denotes the startup trigger, and the white diamond denotes the shutdown trigger.

When the program begins to execute, the startup trigger (denoted as white circle) becomes present and triggers reaction 1 of `CustomerFactory`. Reaction 1 of `CustomerFactory` can further schedule a logical action, which triggers itself again at a future time step, as well as send an output through the output port, which 1triggers reaction 2 of `Barber` at the current logical instant. The program executes until there are no more events in the system.

2.2 Related Work

There is a rich literature behind scheduling for concurrent models of computation from generating static schedules in Synchronous Dataflow [3] to hard real-time schedulers based on directed acyclic graph (DAG) [8]. The current reactor runtime environment for LF automatically exploits parallelism exposed by the (lack of) dependencies between reactions in the dependency graph. The runtime scheduling mechanism for Lingua Franca has been extensively described in [11]. To realize the discrete-event semantics, LF uses an *event queue* to store all events with time tags greater than the current logical time and a *reaction queue* to store events occurred at the current logical time. LF features thread-level parallelism, and it is achieved by work stealing (shown in Fig. 2) - having multiple *worker threads* contend with each other for tasks on the reaction queue and using locks to ensure only one worker thread can modify the reaction queue at a time. These existing approaches, however, tend to require centralized control, pose challenges to modular system design. Since critical data structures, such as *event queue* and *reaction queue*, are shared across multiple worker threads in the current implementation, it is inevitable that lock contentions and synchronization between threads contribute to the majority of the computation overhead and prevent the system from scaling efficiently.

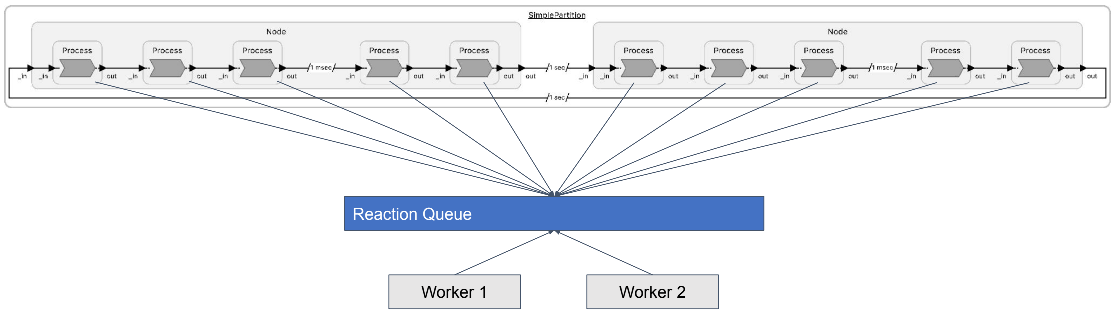


Figure 2: (Current impl.) Two worker threads compete for work from the same reaction queue.

3 Quasi-Static Scheduling

In this section, we introduce our proposed quasi-static scheduling approach. The proposed method contains two parts, a compile-time algorithm and a runtime algorithm. We will now describe these two algorithms in detail.

3.1 Compile-Time Algorithm

The compile-time algorithm analyzes the structure of the LF program and generates a static schedule. To achieve this, the compiler goes through several steps.

Step 1. Obtain a DAG-based dependency graph. The determinism guarantee of the reactors model comes from a well-defined order of the reaction executions, which is captured by a dependency graph, represented in a directed acyclic graph (DAG) shown in Fig. 3. Each node represents a reaction, with the letter prefix denoting the reactor this reaction belongs to and the number postfix denoting the reaction priority. Each edge represents a dependency relation introduced by a connection or the order of reaction priorities. The dependency graph contains two pieces of information: a. the maximum number of reactions to be executed at an instant when they are all triggered simultaneously; b. the order in which these reactions should be executed. In this example, reaction `CF1` should be executed before `WR1`, and `WR1` can only execute after both `CF1` and `CF2` complete.

```
                        +------------------------------------------------+
                        |                                                |
                        |        +----------------------------------+    |
                        |        |                                  |    |
                        |        v                                  |    |
                       CF1 ---> WR1 ---> C1 ---> CF3 <------------ CF2 <-+
                               | |       |       |
                               | |       |       +-----------------------+
                               | |       v                               |
                               | +-----> C2 ------+                      |
                               | |                |                      |
                               | |                v                      |
                               | +-----> B2 ---> C3 -------------+       |
                               |        ^ ^                      |       |
                               |        | |                      |       |
                               |        | +-------------+        |       |
                               |        |               |        v       v
                               |        |               B1 ---> C4 ---> CF4
                               |        |               |
+--------------------+         |        |               |
|B   = Barber        |         |        |               +-----> WR2
|WR = WaitingRoom    |         |        |                        | ^
|C  = Customer       |         |        +------------------------+ |
|CF = CustomerFactory|         |                                   |
+--------------------+         +-----------------------------------+
```

Figure 3: The dependency graph of `SleepingBarber.lf`

Step 2. Partition the graph using an SMT solver. To achieve maximum parallelism, we seek to partition the dependency graph among the number of available worker threads. To do this, we encode the graph partitioning problem into an SMT problem. We first define the following types shown in Fig. 4.

$$\mathit{task} \in \{\epsilon, \mathit{rxn}_0, ..., \mathit{rxn}_n\} \qquad \mathit{schedule} := \{\mathit{task}_0, ..., \mathit{task}_n\} \qquad \mathit{workers} := \{\mathit{schedule}_0, ..., \mathit{schedule}_w\}$$

Figure 4: Type definitions.

Then we define a set of axioms that capture the constraints specified in the dependency graph, which the solution obtained by the SMT solver is required to fulfill. First, the generated schedule cannot be empty.

$$\exists\, \mathit{schedule} \in \mathit{workers}.\ \exists\, i \in \{0, 1, ..., n\}.\ \mathit{schedule}(i) \neq \epsilon \tag{NotEmpty}$$

Each reaction in a time step can only appear once.

$$\begin{aligned} &\forall \mathit{schedule}_1, \mathit{schedule}_2 \in \mathit{workers}.\ \forall i, j \in \{0, 1, ..., n\}.\ \mathit{schedule}_1(i) \neq \epsilon \\ &\quad \implies [\mathit{schedule}_1 = \mathit{schedule}_2 \wedge i = j \iff \mathit{schedule}_1(i) = \mathit{schedule}_2(j)] \end{aligned} \tag{OnlyOnce}$$

In addition, each reaction must happen at least once.

$$\forall \mathit{task} \in \{\mathit{rxn}_0, ..., \mathit{rxn}_n\}.\ \exists \mathit{schedule} \in \mathit{workers}.\ \exists i \in \{0, 1, ..., n\}.\ \mathit{schedule}(i) = \mathit{task} \tag{AtLeastOnce}$$

We further define a binary relation $\prec$ and say that $\mathit{task}_a \prec \mathit{task}_b$ iff there exists an edge in the dependency graph that goes from task_a to task_b. Next, we define an axiom that states the dependency requirement.

$$\forall \mathit{schedule}_1, \mathit{schedule}_2 \in \mathit{workers}.\ \forall i, j \in \{0, 1, ..., n\}.\ \mathit{schedule}_1(i) \prec \mathit{schedule}_2(j) \implies i < j \tag{Dependency}$$

Our compiler extension analyzes the LF program and encodes the above types and contraints into a UCLID5 [9] program. UCLID5, a formal verification engine, compiles the given file into an SMT file. Our compiler extension then further inserts the optimization objective, which requires the solver to find the most parallel schedule, and invokes the Z3 solver. After the Z3 solver returns an optimal partition, represented in a Z3 `Model`, the compiler further generates an executable static schedule for each worker using a custom instruction set.

Step 3. Generate an executable static schedule for each worker. The custom instruction set only contains four instructions shown in Table. 3.1: `execute`, `wait`, `notify`, and `stop`. The `execute` instruction instructs the worker to process a triggered reaction. The `wait` instruction instructs the worker to wait until a semaphore with a specific ID is released. This is used in the case that an upstream reaction, which the current reaction depends on, has not yet been finished by another worker. The `notify` instruction instructs the worker to release a semaphore, so that some downstream reaction owned by another worker can safely proceed. The `stop` instruction instructs the worker to conclude the current time step and wait for new tasks in the next time step.

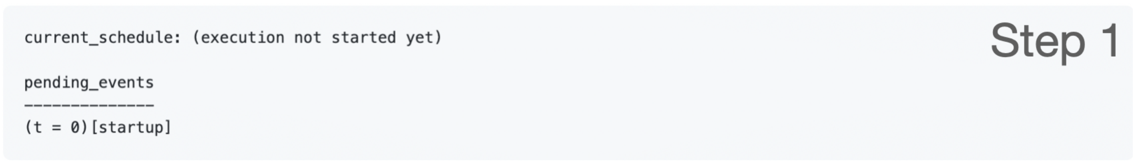
Instruction	Format	Explanation
`execute`	`e [reaction id]`	Execute a reaction.
`wait`	`w [semaphore id]`	Wait for a semaphore.
`notify`	`n [semaphore id]`	Release a semaphore.
`stop`	`s [reaction id]`	Conclude the current time step.

The generated schedule is then stored in a `schedule.h` file along with other C files generated by LF. The data structures in `schedule.h` are declared using the `static` keyword, so that they are placed in the `.text` region of the compiled binary. We are now ready to proceed to the runtime algorithm.

3.2 Runtime Algorithm

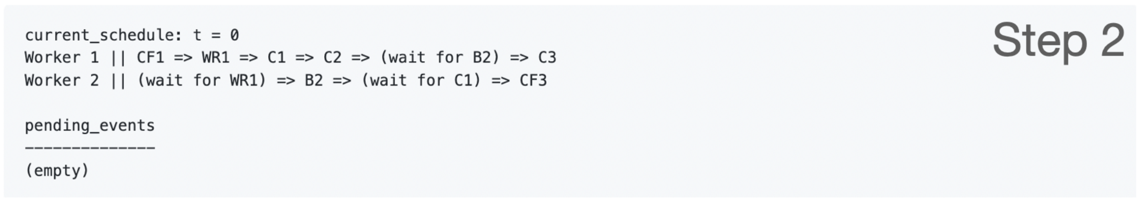
The runtime algorithm makes use of the generated executable schedules and executes them repeatedly until there are no more events to be processed. More specifically, our runtime algorithm performs the following steps.

Step 1. Create a struct variable called *current_schedule* and a *priority_queue* called the *pending_events*, with each element being a list of events to be present at some future time t. For each thread, `current_table` contains a sequence of instructions (defined in the previous section) for the worker thread to execute. The priority queue is used to hold all future events so that closer events will be nearer to the head.



```
current_schedule: (execution not started yet)

pending_events
--------------
(t = 0)[startup]
```



```
current_schedule: t = 0
Worker 1 || CF1 => WR1 => C1 => C2 => (wait for B2) => C3
Worker 2 || (wait for WR1) => B2 => (wait for C1) => CF3

pending_events
--------------
(empty)
```

Figure 5: Executing Step 1 & 2 of the runtime algorithm on `SleepingBarber.lf`.

Step 2. Remove all events present at time t from *pending_events*. All events contains information that indicate what reactions will be triggered, and we mark all triggered reactions runnable in our schedule. From Fig. 5, we pop the list of signals for t = 0 off pending_events, observe that it only contains startup, look for the static schedule corresponding to startup being present, and set current_schedule to the static schedule.

Step 3. Run the entire schedule sequence when the worker thread is requesting a new reaction to execute. The engine will go down the sequence, perform "wait" and "notify" instructions, and return the reaction if we encounter an "execute" instructions and the reaction is marked runnable. Any output generate by the reaction will be converted to events in the system as defined in the LF model. Events in the same time stamp will cause the downstream reaction to be marked runnable immediately before the execution engine resume, and events in the future will be appended to the priority queue *pending_events*. We proceed to the next time step once all threads reach their stopping point. According to Fig. 6, we proceed with executing the static schedule for t = 0. Assume that CF1 has just finished and CustomerFactory.next is scheduled 1 second later. The state of the data structures at this moment looks like chart Step 3.1, Assume that C1 does not produce an output that triggers CF3. Shown in Step 3.2, while no trigger, CF3 in the static schedule will be marked as "to be skipped." Assume that, at this point, B2 finishes and schedules Barber.done at t = 500 msec. The state of the data structure becomes what is shown in Step 3.3.

Step 4. Repeat Step 2 until there are no more events to process.

4 Structural Parallelism in LF

The execution of an LF program strictly follows the semantics of the reactors model. More specifically, for an execution to be valid, it needs to satisfy the constraints that a) events are timestamped, b) reactors process events in timestamp order, and c) reactors process reactions in the order specified by the dependency graph. For this reason, the maximum degree of parallelism is largely dictated by the structure of the program, as we later verified in our experiments.

There are two fundamental patterns that are of special interests. When the reactions form a single chain (i.e. cascade composition) without logical delays, since a downstream reaction needs to wait for its immediate upstream reaction to finish, it is impossible to parallelize the execution regardless how many workers there are (Fig. 7). However, if we introduce logical delays on the connections, the LF program becomes a *pipeline* and is thus parallelizable (Fig. 8). On the other hand, the *scatter-gather* pattern shows a subset of the reactions under a parallel composition. In this case, the execution can be easily parallelizable by mapping each reaction in parallel to a worker (Fig. 9).

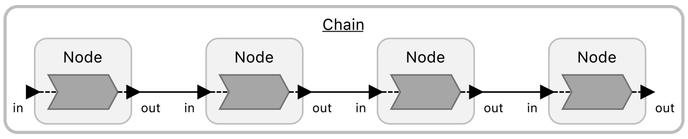
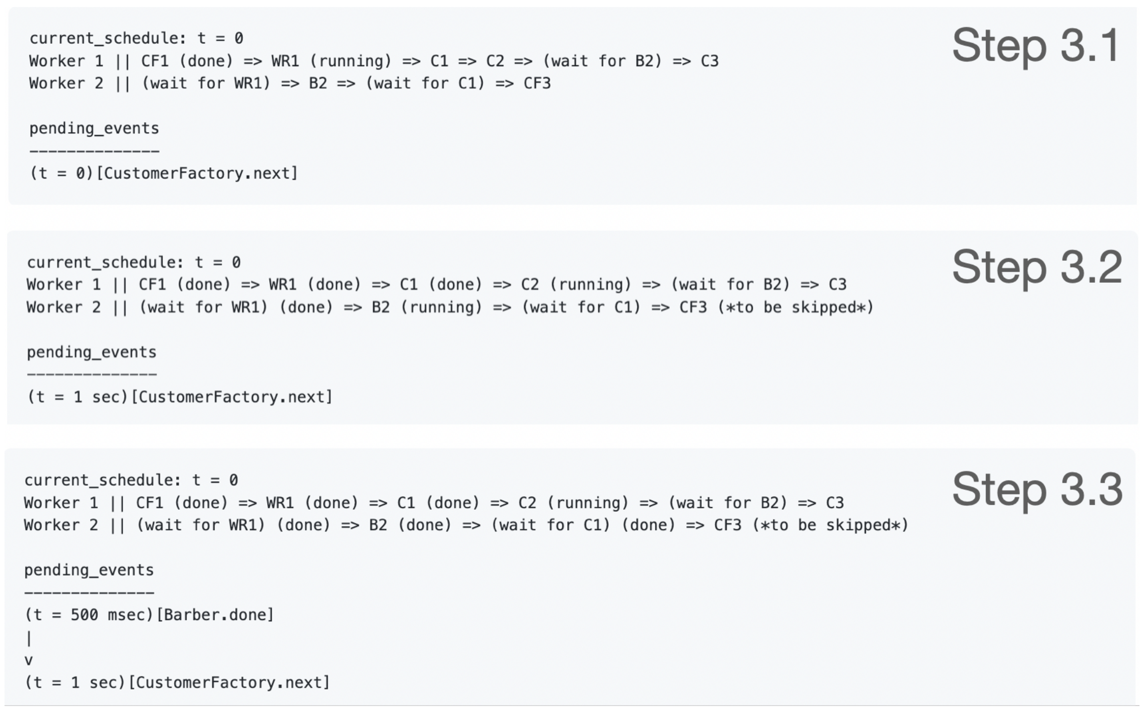


Figure 7: Reactions in cascade composition without logical delays.



```
current_schedule: t = 0
Worker 1 || CF1 (done) => WR1 (running) => C1 => C2 => (wait for B2) => C3
Worker 2 || (wait for WR1) => B2 => (wait for C1) => CF3

pending_events
--------------
(t = 0)[CustomerFactory.next]
```

```
current_schedule: t = 0
Worker 1 || CF1 (done) => WR1 (done) => C1 (done) => C2 (running) => (wait for B2) => C3
Worker 2 || (wait for WR1) (done) => B2 (running) => (wait for C1) => CF3 (*to be skipped*)

pending_events
--------------
(t = 1 sec)[CustomerFactory.next]
```

```
current_schedule: t = 0
Worker 1 || CF1 (done) => WR1 (done) => C1 (done) => C2 (running) => (wait for B2) => C3
Worker 2 || (wait for WR1) (done) => B2 (done) => (wait for C1) (done) => CF3 (*to be skipped*)

pending_events
--------------
(t = 500 msec)[Barber.done]
|
v
(t = 1 sec)[CustomerFactory.next]
```

Figure 6: Executing Step 3 of the runtime algorithm on `SleepingBarber.lf`.

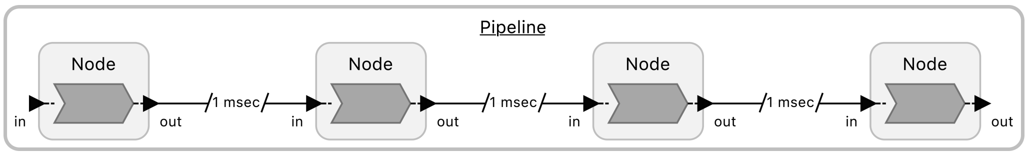


Figure 8: Reactions in cascade composition with logical delays.

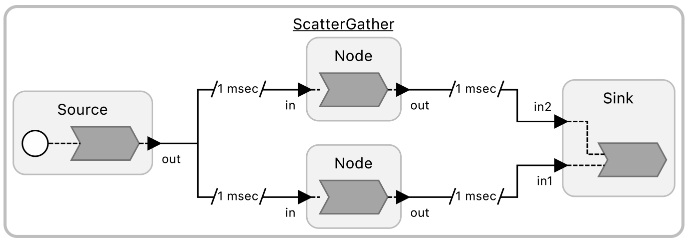


Figure 9: Two reactions in parallel composition.

We algorithmically generate a maximally parallel schedule by inserting an optimization objective in the SMT file generated by our Uclid5 encoding. For each set of the worker schedules, we set up a *countP* function that keeps track of the number of parallel tasks at a particular step in the schedule. We then define a *parallel_metric* variable as follows

$$parallel\_metric = countP(1)^2 + countP(2)^2 + ... + countP(n)^2.$$

In the SMT encoding, we set up an optimization objective, `(maximize parallel_metric)`, which instructs the Z3 solver to find an optimal schedule assignment that maximizes *parallel_metric* variable.

5 Evaluation

We evaluate our quasi-static (QS) scheduler on multiple programs under the `test` folder in LF's Github repo, including `ThreadedThreaded.lf`, `TimeLimit.lf`, and `ActionDelay.lf`. All tests are done on NERSC's Cori supercomputer with **one node** only unless noted. The observed performance is in line with the reasoning presented in Sec. 4: for programs that inherently do not exhibit structures amenable to parallelization, the performance does not seem to improve with an increasing number of workers (in the case of `ActionDelay.lf`). On the other hand, programs such as `ThreadedThreaded.lf` naturally present opportunities for parallelization. We will focus the evaluation section on the `ThreadedThreaded.lf` program (shown in Fig. 10) as it is an crucial pattern for exploiting the structural parallelism in LF. The program exhibits a scatter-gather pattern, with the `Source` reactor sending signals driven by a timer with a period of 200 milliseconds. A bank of reactors named `TakeTime` receives the signals, processes them in parallel, and forwards the outputs to the `Destination`.

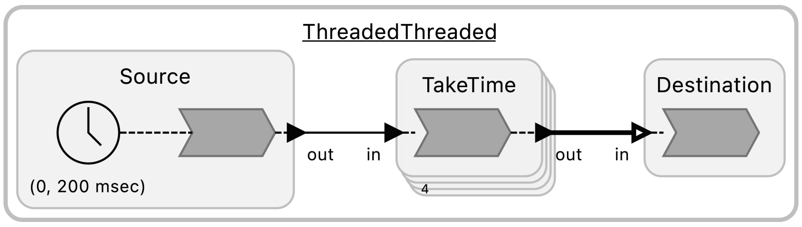


Figure 10: The diagram of `ThreadedThreaded.lf`

In this section, we will also compare our quasi-static (QS) scheduler against the existing non-preemptive (NP) scheduler.

5.1 Strong & Weak Scaling

We test the strong scaling and weak scaling performance on `ThreadedThreaded.lf` program. In strong scaling, we keep the problem size in this case, constant but increase the size of threads. In weak scaling , we increase the problem size, which is the bank size, proportionally to the number of processors so the work/processor stays the same. The reason why we can replace problem size with bank size is that based on tracing observation of `ThreadedThreaded.lf`, `TakeTime` accounts for 99.59% of total process time, which is also the reason that we use `ThreadedThreaded.lf` to evaluate our method. From the Fig. 11, not only taking longer time to finish this task, NP scheduler also lacks a good strong scaling performance. From the line of QS scheduler, we can find that from two workers to four workers, there is an obvious speedup on our our performance. Since the banksize of our `TakeTime` reaction is four, even we add more workers, those extra workers would still be idle. According to Fig. 12, we can see compared to NP scheduler, our QS scheduler has a almost linear speedup with a more balanced and efficient work assignment.

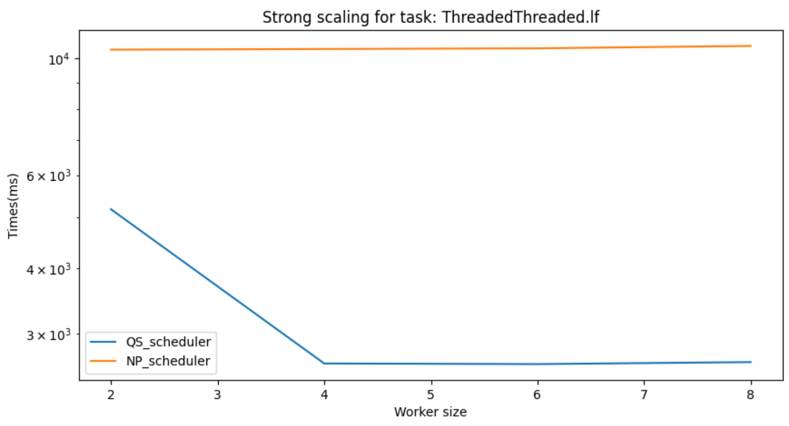


Figure 11: The strong scaling diagram of `ThreadedThreaded.lf`

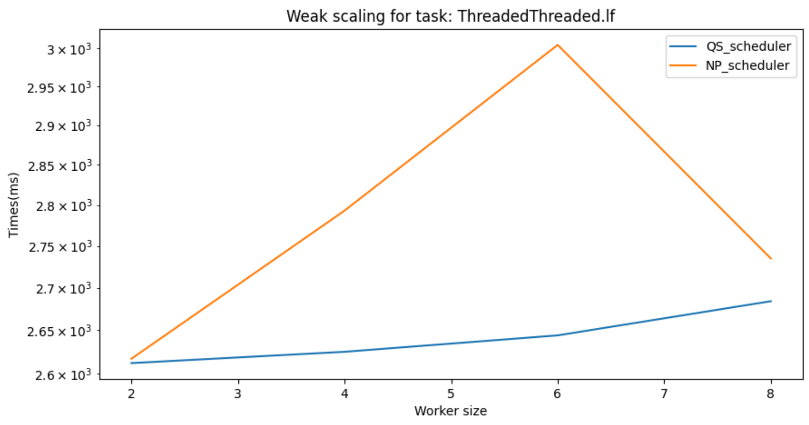


Figure 12: The weak scaling diagram of `ThreadedThreaded.lf`

5.2 Reduced Data Movement

`ThreadedThreaded.lf` seems to support our hypothesis that the quasi-static scheduler can reduce the amount of data movement during execution. When the bank of `TakeTime` reactors contain four reactor instances, under the NP scheduler Fig. 13, a reaction (denoted by a colored rectangle) can be randomly scattered across any of the four horizontal timelines, each of which represents a worker. Under the QS scheduler, however, the same reaction will always be executed by the same worker since the assignment follows a fixed static schedule. In Fig. 14, the colored rectangles are perfectly lined up based on their colors. We obtained the following results on a macOS with 2.3 GHz 8-Core Intel Core i9.

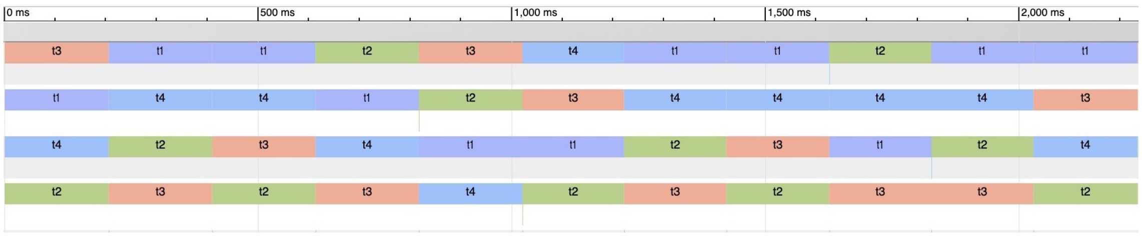


Figure 13: The tracing data of the non-preemptive (NP) scheduler when executing `ThreadedThreaded.lf`.

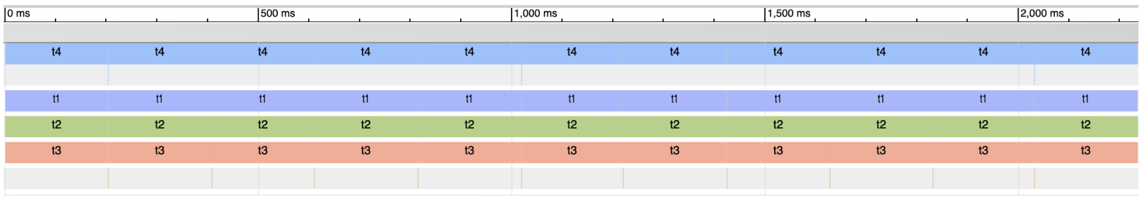


Figure 14: The tracing data of the quasi-static (QS) scheduler when executing `ThreadedThreaded.lf`.

We conjecture that when reactions of the same reactor are executed by the same workers, as in the case of using our QS scheduler, the state variables can preserve in the machine's cache across multiple reaction invocations. This style of execution avoids a type of context switching cost, as in the cost of using the existing NP scheduler, in which the machine needs to flush its cache to load state variables from another reactor. The cost of such data movements could be significant when the computation is data-intensive.

5.3 Synchronization Cost

Below is the total time spent on synchronization. We collect the data by enabling JSON tracing file generation, which is a part of LF's profiling utility. From Fig. 15, we can see our wait time percentage is much smaller than NP scheduler, which means our method saves cost on synchronization, leading to a shorter wait time.

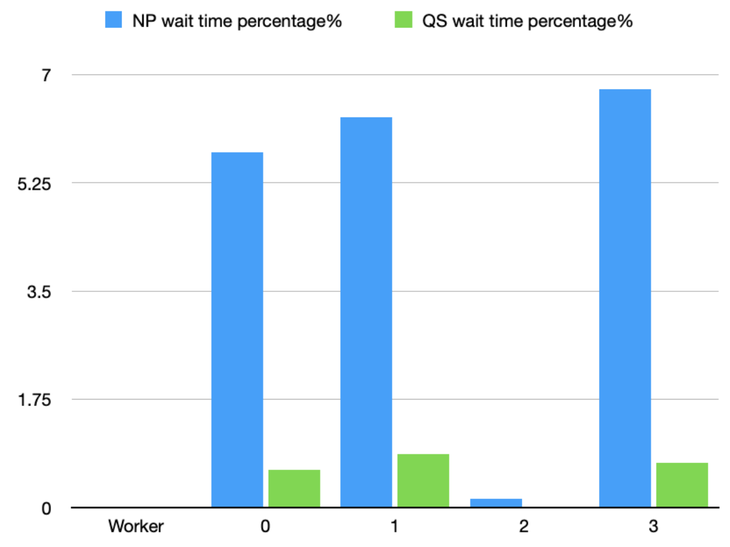


Figure 15: the wait time / total time fraction between two schedulers

6 Limitations and Future Work

It should be noted that our implementation are limited in scope due to the amount of time we have. Currently, our schedule generation algorithm can only detect branches in the reaction dependence graph and treat them as the basic block for worker thread. This algorithm do not consider the possible run time of the reactions, and it is likely to cause load unbalance as idle worker can no longer steal works from other thread. Reducing number of threads in the current version is impossible, but if we can have somehow precise runtime estimation, we can merge the basic blocks and improve thread utilization. Also, this approach may produce very long basic block sequence, and we can break them down for fine-grained parallelization in the future once we have runtime estimation.

We do not have time to implement a distributed version of this algorithm despite LF support federated mode. Our result shows that we can at least do as good as centralized scheduler on the same machine, where communication time is negligible. But, this algorithm can really show its benefit in a distributed environment and large numbers of reactions. Our algorithm removes the serial bottleneck in the centralized scheduler and make every node progress on themselves without communicating with other threads unless explicitly specified to do so (through the notify and wait instructions). Notify and wait instructions are currently implemented on semaphores, and a future distributed version will change them to remote message based.

Also, since we are using SMT solver to generate schedule, it takes a very long time to generate a large schedule. SMT is an NP-complete problem, so the runtime will grow exponentially in proportion to the problem size. We can explore other algorithm to partition the dependence graph in the future to reduce the compilation time.

7 Conclusion

Lingua Franca is a great framework for discrete-time simulation with reasonable performance with its existing scheduler. However, the current scheduler design doesn't work well on distributed environment due to serial bottleneck and constant communication. The quasi-static scheduler is an efficient way to avoid communication and remove bottleneck in theory by providing a "script" for workers to follows without consulting the coordinator. Though, it is difficult to implement it in practice due to various issues, and we are considering continuing our work on this project after the end of class so that we can eventually create a practical scheduler for LF on large-scale distributed system.

References

[1] Edsger Wybe Dijkstra. *Cooperating sequential processes, technical report ewd-123.* 1965.

[2] KAHN Gilles. “The semantics of a simple language for parallel programming”. In: *Information processing* 74 (1974), pp. 471–475.

[3] Edward A Lee and David G Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[4] Nicholas Halbwachs et al. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[5] Gérard Berry and Georges Gonthier. “The Esterel synchronous programming language: Design, semantics, implementation”. In: *Science of computer programming* 19.2 (1992), pp. 87–152.

[6] Edward A Lee and Thomas M Parks. “Dataflow process networks”. In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.

[7] Carl Hewitt. “Actor model of computation: scalable robust information systems”. In: *arXiv preprint arXiv:1008.1459* (2010).

[8] Zheng Dong and Cong Liu. “New analysis techniques for supporting hard real-time sporadic DAG task systems on multiprocessors”. In: *arXiv preprint arXiv:1808.00017* (2018).

[9] Sanjit A Seshia and Pramod Subramanyan. “UCLID5: Integrating modeling, verification, synthesis and learning”. In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–10.

[10] Marten Lohstroh et al. “Reactors: A Deterministic Model for Composable Reactive Systems”. In: *Cyber Physical Systems. Model-Based Design: 9th International Workshop, CyPhy 2019, and 15th International Workshop, WESE 2019, New York City, NY, USA, October 17-18, 2019, Revised Selected Papers.* New York City, NY, USA: Springer-Verlag, 2019, pp. 59–85. ISBN: 978-3-030-41130-5. DOI: `10.1007/978-3-030-41131-2_4`. URL: `https://doi.org/10.1007/978-3-030-41131-2_4`.

[11] Marten Lohstroh. “Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems”. PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html`.

[12] Marten Lohstroh et al. “Reactors: A Deterministic Model for Composable Reactive Systems”. In: *Cyber Physical Systems. Model-Based Design.* Ed. by Roger Chamberlain, Martin Edin Grimheden, and Walid Taha. Cham: Springer International Publishing, 2020, pp. 59–85. ISBN: 978-3-030-41131-2.

[13] Marten Lohstroh et al. “Toward a Lingua Franca for deterministic concurrent systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.4 (2021), pp. 1–27.