Parallel computing in Framsticks

Maciej Komosinski Szymon Ulatowski

Technical Report RA–18/2013

Institute of Computing Science
Poznan University of Technology
December 2013

Abstract

This report demonstrates how parallel computation can be implemented in the Framsticks environment. A number of possible multithreaded and distributed architectures and configurations is shown. The main part of this report discusses and explains two experiment definitions (`prime-mt` and `standard-mt`) that exploit multithreading. These experiment definitions are included in the official Framsticks distribution. The first one serves as a minimal example of how parallelization can be implemented in Framsticks. The second one is more complex: it shows how to deal with Slave experiments that do not have an internal stop condition, how to migrate the evolved genotypes between Slaves, and how to use Slave checkpoint events to monitor the progress of evolution.

Contents

1 Introduction

The Framsticks simulator [25], since its initial releases in 1996, has been used as a computing engine in a number of diverse applications [24], some of which are mentioned below:

- comparison of genetic encodings in artificial life and evolutionary design [21, 23, 8],
- estimating symmetry of evolved and designed agents [6],
- employing similarity measure to organize evolved constructs [10, 9, 20, 11],
- bio-inspired visual-motor coordination [7] and real-time coordination [19],
- optimization of fuzzy controllers that can be understood by a human [4],
- modeling of plastic neural nets and their evolution [5],
- modeling robots [33, 29],
- user-driven (interactive, aesthetic) evolution [24, 6],
- synthetic neuroethology [27, 28],
- analyses of brain activity evoked by perception of novel biological motion [30, 31],
- evaluation of logical abductive hypotheses [16, 15] and solving abductive problems [3],
- modeling perception of time in humans [12, 13, 14],
- modeling foraminiferal genetics, morphology, simulation, and evolution [17, 18],
- modeling communication, emergence, flocking, evolution of restraint, predator–prey coevolution, semiosis, speciation, and other biological phenomena [24, 1, 2].

Many of these applications require considerable amounts of computing power, and it is often the case that the more the computing resources are available, the more meaningful the experiments and their results are. With modern computers equipped with many processors and cores and a clear direction of hardware development in the near future, using multithreading in the Framsticks processing engine allows to exploit more computing power on a single machine in a single experiment [26]. These are the major motivations for introducing multithreading support in Framsticks.

Other options that allow one to exploit multi-core machines and distributed machine clusters *without using multithreading* include running independent, separate experiments in parallel (which lets one gather a number of independent results), and using Framsticks server [22] to both parallelize experiments on one machine and distribute experiments among different machines [32].

1.1 Multithreaded and distributed architectures in Framsticks

Many aspects of the Framsticks environment can be extended by editing scripts responsible for certain tasks. This is usually done by Framsticks developers and advanced users, while normal users can use these scripts without learning about Framsticks technical details. One of the most important types of scripts is *Experiment Definition* (short: *expdef*), each one describing a

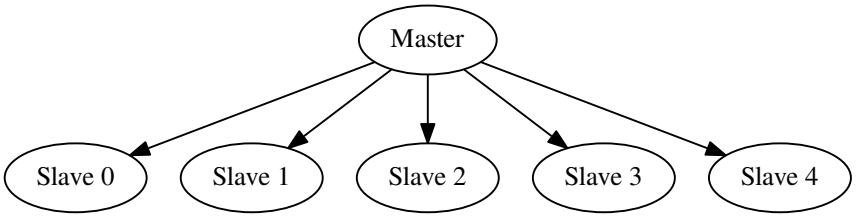

Figure 1: Basic multithreading architecture.

class of experiments, defining its input (including the user-adjustable parameters), output, and processing rules.

In order to benefit from parallelization, the *Experiment Definition* script has to handle multithreading explicitly, which means it is responsible for assigning work to individual threads and integrating the results. Each Slave thread runs its own *expdef* script inside its own Simulator object, independent from the Master thread's Simulator, as illustrated in Fig. 1. The single-threaded script can often be used as a building block when creating the multithreaded version of the experiment.

The *Master* can create, delete and control Slaves by calling any method of the Slave's Simulator object. Most of the Slave's functions and fields can only be accessed by the Master while the Slave is not running. The exceptions are: `Simulator.start()`, `Simulator.stop()`, `Simulator.running`, and `Simulator.simspeed`.

In addition to the usual *expdef* events, the Master script receives two additional slave-related events:

- `onSlaveStop()`: called when any of the Slave Simulators stops. A Slave can stop by itself or can be stopped by the Master.
- `onSlaveCheckpoint()`: called when a Slave announces a checkpoint. Checkpoints allow for asynchronous Slave-to-Master notifications and can pass any data as an argument.

All slave-triggered events are asynchronous and subject to delays.

Slaves are not aware of the Master's existence – they work just like single-threaded standalone Simulators. While this architecture allows the Master to control Slaves on a low level, it may be beneficial to restrict the Master-Slave interaction to a few well-defined high level operations:

- Load experiment state,
- Save experiment state,
- Start simulation,
- Stop simulation.

This way, the Slave tasks become self-contained (one sends a complete experiment state each time) which helps avoid unexpected behaviors. Using only Load/Save for data exchange promotes encapsulation, because these operations are handled by the Slave script which can enforce internal consistency. Manipulating "live" Slave objects by the Master thread may be thought of as an equivalent of accessing private members in some programming languages – it is useful for simple tasks and debugging, but it is easy to create a Master script that makes the correctly written Slave script to fail.

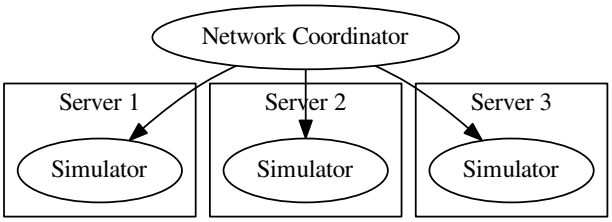

Figure 2: Running the multithreaded experiment using network servers instead of Slave threads.

If possible, keeping the experiment state file format interchangeable between Master and Slaves is recommended. This way experiment data can be loaded into a multithreaded or a single-threaded environment.

The basic multithreaded experiment may be implemented as follows:

1. Make sure the Slave (single-threaded) experiment supports Load and Save operations.
2. Create a new experiment definition by appending `-mt` suffix to the single-threaded experiment name.
3. Implement `function onStart()`: divide the task into subtasks, send them to Slaves using `Simulator.slaves[...].load` or `import`, and start the Slave Simulators (`Simulator.slaves.startAll(`
4. If the Slave experiment has no stop condition, your Master will be responsible for calling `Simulator.slaves.stopAll()` or `Simulator.slaves[...].stop()` at some point during the experiment. Slave checkpoint events can be used to react to Slave progress, as demonstrated in the `standard-mt.expdef` described below.
5. Implement `function onSlaveStop()`: retrieve slave data using `StopEvent.slave.save()` and integrate with the current Master state.
6. Implement `function onStep()`: Typically, the Master does not perform any continuous work and only reacts to events, yet its `onStep()` function is executed periodically once the experiment is started, because the Master is still a normal Framsticks Simulator. To avoid wasting CPU power and hampering performance of Slave threads, call `Simulator.sleep(...)`, which causes the Master thread to enter the sleep state for a number of milliseconds and to stop using the processor.

Once the multithreaded experiment is designed and implemented, it may become a distributed experiment by running Slave Simulators on multiple Framsticks Servers (Fig. 2), as the basic Slave Simulator and Framsticks Server capabilities are essentially identical. This requires a *Network Coordinator*, which is an external client application that uses the Framsticks server protocol [22] and manages remote servers – an example is Framsticks Java Framework [32].

Given the common interface, each or some of the Servers could run the multithreaded version of the experiment or could be replaced by another *Network Coordinator*, increasing the parallelization even further (Fig. 3). In this case, the same multithreaded script is used in all *Network Coordinator* and *Master* nodes.

2 Two experiment definitions: `prime-mt` and `standard-mt`

This section discusses two experiment definitions that exploit multithreading and are included in the official Framsticks distribution.

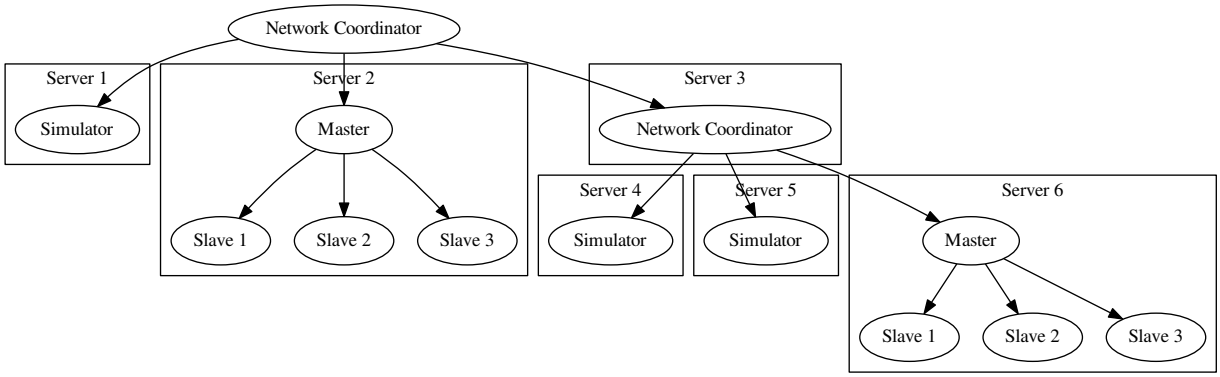

Figure 3: Using multithreaded servers in a distributed experiment.

The first one, `prime-mt`, is the basic functional example of the Framsticks multithreaded experiment, demonstrating how the Master thread creates and controls its Slave threads. Following the usual convention, `prime-mt`'s single-threaded counterpart is `prime`, the experiment that finds the list of prime numbers in a given range. By the nature of the problem, there is always a solution, so the `prime` experiment always stops itself when the solution is ready and `prime-mt` relies on this behavior.

The other multithreaded experiment, `standard-mt`, apart from being an useful extension of the most common Framsticks evolutionary optimization experiment `standard`, shows how to deal with Slave experiments that do not have an internal stop condition. `standard-mt`'s Master manages the parallelization by periodically stopping the Slaves and migrating the evolved genotypes between them. Slaves' checkpoint events are used for monitoring the progress of evolution.

All multithreaded experiments are recommended to include two script pieces that facilitate thread-awareness. The first piece is a property that allows users to adjust the number of threads used by the experiment:

standard_props_threads.inc

```
# use standard_threads.inc to calculate the effective number of threads.

property:
id:threads
type:d
name:Number of threads
group:Parallelized
help:~
Use this setting to set the number of threads:
- positive values (1,2,3,...) are interpreted literally as the number of threads,
- zero (0) means the number of threads equal to the number of CPU cores,
- negative values (-1,-2,-3,...) mean the number of CPU cores less 1,2,3,...
~
```

The second piece one should include when writing their own multithreaded experiment definition is a little function that computes the effective number of threads:

standard_threads.inc

```
// Calculate the effective user-selected number of threads
// (also handles tricks with zero and negative values).
// Include standard_props_threads.inc to create the ExpProperties.threads field.
```

```
function getExpPropertiesThreads()
{
  if (ExpProperties.threads > 0)
    return ExpProperties.threads;
  return Math.max(1, Simulator.cpus + ExpProperties.threads);
}
```

2.1 `prime-mt` experiment definition

Let us start with an illustrative experiment definition that uses an extremely simple case of a computationally-intensive task: finding prime numbers. The core function below is used as an example of a computationally costly procedure:

Listing 1: A simple computationally-intensive function used in `prime.expdef`

```
function testPrime(N) //extremely inefficient, just for illustration
{
  for(var i = 2; i < N; i++)
    if ((N / i)*i == N) return 0;
  return 1;
}
```

`prime.expdef` and `prime-mt.expdef` introduce the same three domain-specific experiment parameters and a state: `ExpProperties.from_number`, `ExpProperties.to_number`, `ExpState.current_number` and `ExpState.result`. By convention, parameters (`ExpProperties`) mean something constant that is defined before starting the experiment, while the state (`ExpState`) is variable.

`prime.expdef` was meant to mimic a regular Framsticks experiment, where simulation steps reflect passing simulation time, so instead of just looping through the input range in one step, it tests primality of only one value per step. As a side effect of this approach, such an experiment can be easily interrupted, and the current computation state can be saved in the experiment state file. The `onStep()` function is listed below; note the stop condition – calling `Simulator.stop()`.

Listing 2: The single step of `prime.expdef`

```
function onStep()
{
  if (ExpState.current_number > ExpProperties.to_number)
  {
    Simulator.stop();
    return;
  } else
  {
    if (testPrime(ExpState.current_number))
    {
      ExpState.result.add(ExpState.current_number);

      // Optional data associated with a checkpoint:
      // Simulator.checkpointData(ExpState.current_number);
    }
    ExpState.current_number++;
  }
}
```

`prime-mt.expdef` is slightly more complex. It starts by setting up its Slave Simulators (`Simulator.slaves.size=...`) and sending the initial chunk of work (that is, a subrange of the whole input range).

Listing 3: Starting `prime-mt.expdef`

```
function onStart()
{
  Simulator.slaves.stopAll();
  Simulator.slaves.size = getExpPropertiesThreads();
  Simulator.print("Using " + Simulator.slaves.size + " threads");
  g_scheduled = 0;
  for(var i = 0; i < Simulator.slaves.size; i++)
  {
    scheduleChunkOfWork(Simulator.slaves[i]);
  }
}
```

Our `scheduleChunkOfWork()` function works by accessing the Slave Simulator object directly and setting its experiment parameters and state variables. A chunk of work is simply the next range of numbers (of the length defined by `ExpProperties.chunk`), after the last scheduled one. Then, a Slave Simulator is started. The `g_scheduled` variable keeps the current number of the "work in progress" slaves.

Listing 4: Scheduling a single piece of work to a Slave Simulator in `prime-mt.expdef`

```
function scheduleChunkOfWork(slave)
{
  if (ExpState.current_number <= ExpProperties.to_number)
  {
    if (slave.running)
    {
      Simulator.print("Slave simulator still running in scheduleChunkOfWork();!");
      return; //or stop() and continue scheduling, but something went wrong
    }
    slave.expdef = "prime";
    slave.expproperties.from_number = ExpState.current_number;
    slave.expproperties.to_number = Math.min(ExpState.current_number +
      ExpProperties.chunk - 1, ExpProperties.to_number);
    slave.init();
    ExpState.current_number = slave.expproperties.to_number + 1;
    slave.start();
    g_scheduled++;
  }
}
```

`prime.expdef` stops automatically after finishing its work and this automatically sends the `onSlaveStop()` event, where our `prime-mt.expdef` receives the partial result and schedules the next chunk (unless the whole task is already completed).

In `scheduleChunkOfWork()` function, we again access the Slave Simulator variables directly (through the StopEvent object).

The last part shows the purpose of the `g_scheduled` variable – it tells if we are still waiting for some more Slaves. It is important to note that checking the current running state of Slaves (instead of `g_scheduled`) would be incorrect here, because `slave.running==0` is also possible after some slave finished the work, but its `onSlaveStop` event was not yet processed. The difference is that `slave.running` only checks the slave part of the transaction, while `g_scheduled` knows if the Master script has finished processing the Slave's return data.

Listing 5: Actions performed by `prime-mt.expdef` after the chunk of work is completed

```
function onSlaveStop()
{
  g_scheduled--;
  //Simulator.print("slave #"+StopEvent.index+" stopped");

  // For many experiments, saving the experiment state could be a more appropriate
  // way for retrieving the result:
  // var s=StopEvent.slave.save(null); //null=save to string instead of named file
  // Simulator.print("saved by #"+StopEvent.index+" :"+s);

  // But here, we access the ExpState.result field directly:
  var res = StopEvent.slave.expstate.result;
  //Simulator.print("slave #"+StopEvent.index+" result:"+res);
  ExpState.packet_counter++;
  for(var i = 0; i < res.size; i++)
    ExpState.result.add(res[i]);

  if (Simulator.running)
  {
    scheduleChunkOfWork(StopEvent.slave);

    // It may be tempting to use Simulator.slaves.running to see if the computation
    // is still advancing, but the right way is g_scheduled:
    // g_scheduled==0 means that there is no scheduled work because we are done.
    // Simulator.slaves.running==0 could mean the same, but it could also be 0
    // because all the simulations have finished their current jobs and are waiting
    // for the next assignments.

    if (g_scheduled == 0)
      Simulator.stop();
  }
}
```

2.2 `standard-mt` experiment definition

Like in the previous example, `standard-mt.expdef` splits the work among a number of Slave Simulators that run the single-threaded version of the experiment – in this case, `standard.expdef`.

Compared to `prime-mt.expdef`, the script demonstrates two new techniques:

- using file import/export for transferring bigger amounts of data between Simulators,
- using checkpoint events for monitoring the Slave Simulator progress.

Here is how `onStart()` transfers the Master Simulator settings into the Slave: exporting to `null` filename returns the file contents as a string instead of writing to the actual file, and the exported data is imported by passing the entire contents preceded by a "string://" pseudo-URL as the filename argument in the `import()` function.

Listing 6: Starting the multithreaded `standard-mt` experiment

```
function onStart()
{
  SHOULD_STOP = 0;
  Simulator.slaves.stopAll();
  Simulator.slaves.cancelAllEvents();
  Simulator.slaves.size = getExpPropertiesThreads();
```

```
  var exported_settings = Simulator.export(null, 4 + 16 + 32, -1, -1);
  // -1 means export all groups
  //Simulator.print("exported="+exported_settings);

  for (var i = 0; i < Simulator.slaves.size; i++)
  {
    var s = Simulator.slaves[i];
    s.expdef = "standard";
    s.import("string://" + exported_settings, 4 + 8 + 16);
  }

  sendToSlaves();
  GenePools[0].mergeInstances();
  Simulator.slaves.startAll();
}
```

The same method is used for transferring genotypes between Simulators except that for the Master-to-Slave export, the string is created by individually saving `Genotype` objects to `File`.

Listing 7: Saving the selected genotypes to a string

```
function exportSelectedGenotypes(selection)
{
  var f = File.new(); //File.new() creates a new memory file,
  // its content is then returned as a text string upon closing

  for (var i = 0; i < selection.size; i++)
    f.writeNameObject("org", GenePools[0][selection[i]]);
  return f.close();
}
```

Then, we use the already known "string://" URL to import the string into a different Simulator:

Listing 8: Key operations of the `sendToSlaves()` function in `standard-mt.expdef`

```
function sendToSlaves()
{
  ...
  GenePools[0].splitInstances();
  var mixed = []; //mixed[i] = genotypes exported for slave #i
  ...

  var a = randomAllocation(GenePools[0].size, Simulator.slaves.size);
  ...
  var a=randomAllocation(GenePools[0].size, Simulator.slaves.size);
  for (var i = 0; i < a.size; i++)
    mixed[i] = exportSelectedGenotypes(a[i]);
  ...
  for (var i = 0; i < Simulator.slaves.size; i++)
  {
    var s = Simulator.slaves[i];
    s.genepools[0].clear();
    s.import("string://" + mixed[i], 2 + 128); // import into slave
    Simulator.print("slave #" + i + " has " + s.genepools[0].size + " genotypes");
    s.genepools[0].mergeInstances();
    ...
  }
```

```
  ...
}
```

Note that `sendToSlaves()` calls two rarely used functions: `splitInstances()` before preparing the tasks for Slaves and `mergeInstances()` after each Slave has imported its file. This is because assigning work to Slaves works on a Genotype level, which in `standard.expdef` can represent a number of identical genotypes (`Genotype.instances` field). `GenePool.splitInstances()` turns all multiple-instance Genotypes into single-instance Genotypes by cloning, and `GenePool.mergeInstances` restores the normal state where one genotype can have many instances.

Unlike `prime.expdef`, the standard Framsticks experiment does not stop itself. `standard-mt.expdef` uses checkpoint events to watch the progress of its Slave Simulators and stops the Slaves after the desired number of evaluations. The Master Simulator is able to do it because its Slave script `standard.expdef` sends the checkpoint event each time the next creature has been evaluated. `ExpState.totaltestedcr` is the current number of evaluations in the given Slave, the following code makes it available on the Master side in the `CheckpointEvent.data` field upon receiving the *SlaveCheckpoint* event.

Listing 9: Sending the checkpoint event in `standard.expdef`

```
function onDied(cr)
{
  ...
  Simulator.checkpointData(ExpState.totaltestedcr);
  ...
}
```

The migration period (`ExpProperties.mix_period`) parameter introduced by our `standard-mt.expdef` controls the migration frequency. A migration occurs after reaching the desired number of evaluations, expressed as a percentage of the gene pool capacity (`ExpProperties.capacity`): there are capacity×mix_period/100 evaluations between migrations. For the default mix_period=1000, the number of evaluations between migrations is 10× the capacity of the gene pool.

Listing 10: Checking the number of Slave's evaluations in `onSlaveCheckpoint()` function, `standard-mt.expdef`

```
function onSlaveCheckpoint()
{
  //check if desired number of evaluations done?
  if (CheckpointEvent.data>=(ExpProperties.capacity*ExpProperties.mix_period/100))
  {
    ...
  }
}
```

`standard-mt.expdef` supports one of two modes, selected by the `ExpProperties.keep_threads_running` parameter:

- When `keep_threads_running==0`, each Slave is stopped by the Master after reaching the desired number of evaluations. All gene pools will get an equal number of evaluations, regardless of the relative Slave performance.

  Listing 11: Handling the keep_threads_running==0 mode in the `onSlaveCheckpoint()` function, `standard-mt.expdef`

  ```
  {
  ```

```
  ...
  Simulator.print("Slave #" + CheckpointEvent.index + " " +
    CheckpointEvent.data + " evaluations, stopping");
  CheckpointEvent.slave.stop();
  Simulator.slaves.cancelEventsFromSlave(CheckpointEvent.slave);
  ...
}
```

- When `keep_threads_running==1`, the Master waits until all Slaves reach the desired number of evaluations and only then they are all stopped. This means that gene pools from the "faster" Slaves will get more evaluations than those from the "slower" ones.

  In this mode, the information about reaching the goal is stored in the `g_goal_reached` array. All 1's mean that all Slaves can be stopped. The array is not used in the first mode, where we simply check the Slaves' `running` state (they are being stopped one by one after reaching the goal, so if they are all stopped, we are done).

  Listing 12: Handling the keep_threads_running==1 mode in the `onSlaveCheckpoint()` function, `standard-mt.expdef`

```
{
  ...
  if (g_goal_reached[CheckpointEvent.index])
    return; //already handled

  Simulator.print("Slave #" + CheckpointEvent.index + " " +
    CheckpointEvent.data + " evaluations");
  g_goal_reached[CheckpointEvent.index] = 1;
  if (testAllGoalsReached())
  {
    Simulator.slaves.stopAll();
    Simulator.slaves.cancelAllEvents();
  }
  ...
}
```

The purpose of `cancelAllEvents()` and `cancelEventsFromSlave()` is to protect against the case where a Slave managed to emit more events before the Master could handle the first of them (because events are asynchronous). Not cancelling the events might make the Master receive more events even after a Slave has just been stopped.

If all Slaves have been stopped, the migration is executed. It is basically a retrieval of all genotypes from Slaves into the Master Simulator (`loadFromSlaves()`), followed by a restart of the work cycle, like in the onStart() function.

Listing 13: Migration-related source, `standard-mt.expdef`

```
function onSlaveCheckpoint()
{
  ...
  if (Simulator.running && (Simulator.slaves.running == 0))
  {
    Simulator.print("migrating slave simulations...");
    loadFromSlaves(); // may set SHOULD_STOP
    if (SHOULD_STOP)
    {
      Simulator.message("Done %d migrations, stopping." % ExpState.migrations, 0);
```

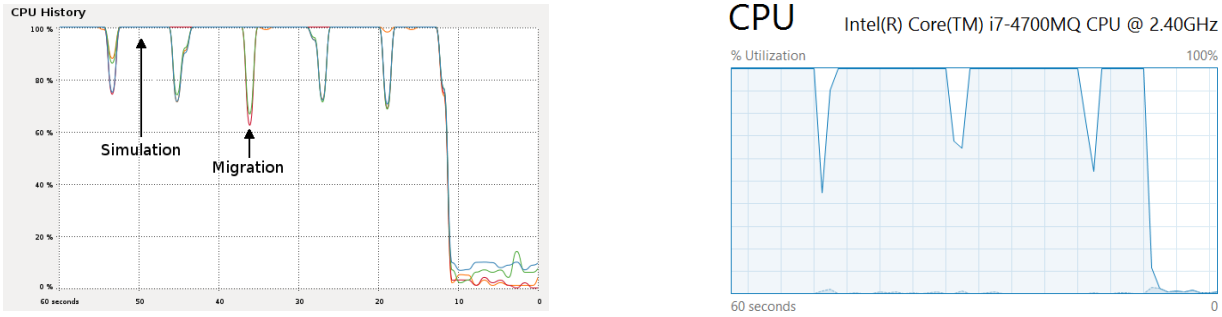

Figure 4: Example CPU utilization charts. Left: Linux MATE System Monitor. Right: Windows Task Manager.

```
      Simulator.stop();
      GenePools[0].mergeInstances();
    }
    else
    {
      sendToSlaves();
      GenePools[0].mergeInstances();
      Simulator.slaves.startAll();
    }
  }
  ...
}

function loadFromSlaves()
{
  Simulator.print("loading genotypes from slaves... (migrations=" +
    ExpState.migrations + ")");
  GenePools[0].clear();
  for (var i = 0; i < Simulator.slaves.size; i++)
  {
    var s = Simulator.slaves[i];
    var fromslave = s.export(null, 2, 0, 0);
    Simulator.import("string://" + fromslave, 2 + 128);
    ExpState.totaltestedcr += s.expstate.totaltestedcr;
    ExpState.totaltests += s.expstate.totaltests;
  }
  ExpState.migrations++;
  Simulator.checkpointData(ExpState.migrations);
  if (ExpProperties.max_migrations > 0 &&
    ExpState.migrations >= ExpProperties.max_migrations)
    SHOULD_STOP = 1;
}
```

In the `standard-mt.expdef` experiment definition, the migration stage temporarily stops all Slave threads. This is demonstrated in the CPU utilization charts shown in Fig. 4; higher gene pool capacities mean longer migration periods. Apart from these periods, threads are highly independent and there is nearly no additional locking and synchronization, so an almost linear speedup is possible when the number of threads is increased [26].

As described in Sect. 1.1, in this experiment Master does not perform any intensive work and only reacts to events. However it still calls `onStep()`, so we use `Simulator.sleep(delay)` to avoid using the processor unnecessarily. Computational experiments showed that the delay

does not affect performance in a noticeable way. The following delays were tested: 1ms, 10ms, 100ms, 1000ms. There was no significant correlation between the delay and the performance. The only pattern found was an increase in the amount of time spent inside system calls when the delay is decreased – it could be because of the delay function being called more times. The conclusion may be different on other platforms, if, for example, some waiting periods cause additional performance penalty due to a low timer resolution (e.g. if short waiting periods were implemented as busy waiting).

3 Summary

This report demonstrated the numerous ways parallel computation can be implemented in the Framsticks environment. It started by enumerating various fields of science and various kinds of experiments where Framsticks has been used. Then, multithreaded and distributed architectures and configurations have been presented. Sect. 2 discussed two experiment definitions (`prime-mt` and `standard-mt`) that exploit multithreading and that are included in the official Framsticks distribution. The first one serves as a minimal example of how parallelization can be implemented in Framsticks. The second one is more complex: it shows how to deal with Slave experiments that do not have an internal stop condition, how to migrate the evolved genotypes between Slaves, and how to use Slave checkpoint events to monitor the progress of evolution.

References

[1] Walter de Back. Eco-evolutionary experiments with situated agents. Master's thesis, 2006. URL: `http://www.framsticks.com/files/common/MSc_deBack_EcologyEvolution.pdf`.

[2] Walter de Back, M. Wiering, and E. de Jong. Red Queen dynamics in a predator-prey ecosystem. *Proceedings of the 8th annual conference on genetic and evolutionary computation*, pages 381–382, 2006. URL: `http://igitur-archive.library.uu.nl/vet/2007-0302-210407/wiering_06_red.pdf`.

[3] Andrzej Gajda, Adam Kups, and Mariusz Urbański. A connectionist approach to abductive problems: employing a learning algorithm. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 353–362. ACSIS, 2016. URL: `http://dx.doi.org/10.15439/2016F484`, `doi:10.15439/2016F484`.

[4] Maciej Hapke and Maciej Komosinski. Evolutionary design of interpretable fuzzy controllers. *Foundations of Computing and Decision Sciences*, 33(4):351–367, 2008. URL: `http://www.framsticks.com/files/common/Komosinski_EvolveInterpretableFuzzy.pdf`.

[5] Matej Hoffmann. Structural coupling with environment and its modelling on neural driven agents. Master's thesis, 2006. URL: `http://www.framsticks.com/files/common/MSc_Hoffmann_StructuralCoupling.pdf`.

[6] Wojciech Jaskowski and Maciej Komosinski. The numerical measure of symmetry for 3D stick creatures. *Artificial Life Journal*, 14(4):425–443, Fall 2008. URL: `http://dx.doi.org/10.1162/artl.2008.14.4.14402`, `doi:10.1162/artl.2008.14.4.14402`.

[7] Jacek Jelonek and Maciej Komosinski. Biologically-inspired visual-motor coordination model in a navigation problem. In Bogdan Gabrys, Robert Howlett,

and Lakhmi Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 4253 of *Lecture Notes in Computer Science*, pages 341–348. Springer, Berlin/Heidelberg, 2006. URL: http://www.framsticks.com/files/common/BiologicallyInspiredVisualMotorCoordinationModel.pdf, doi:10.1007/11893011_44.

[8] Maciej Komosinski. Evolutionary design of tall structures. Research report RA–06/12, Poznan University of Technology, Institute of Computing Science, 2012.

[9] Maciej Komosinski, Grzegorz Koczyk, and Marek Kubiak. On estimating similarity of artificial and real organisms. *Theory in Biosciences*, 120(3-4):271–286, December 2001. URL: http://dx.doi.org/10.1007/s12064-001-0023-y, doi:10.1007/s12064-001-0023-y.

[10] Maciej Komosinski and Marek Kubiak. Taxonomy in Alife. Measures of similarity for complex artificial organisms. In Jozef Kelemen and Petr Sosík, editors, *Advances in Artificial Life*, volume 2159 of *Lecture Notes in Computer Science*, pages 685–694. Springer, Berlin/Heidelberg, 2001. URL: http://www.framsticks.com/files/common/Komosinski_TaxonomyAlife_ECAL2001.pdf, doi:10.1007/3-540-44811-X_79.

[11] Maciej Komosinski and Marek Kubiak. Quantitative measure of structural and geometric similarity of 3D morphologies. *Complexity*, 16(6):40–52, 2011. URL: http://dx.doi.org/10.1002/cplx.20367, doi:10.1002/cplx.20367.

[12] Maciej Komosinski and Adam Kups. Models and implementations of timing processes using Artificial Life techniques. Research report RA–05/09, Poznan University of Technology, Institute of Computing Science, 2009.

[13] Maciej Komosinski and Adam Kups. Implementation and simulation of the Scalar Timing Model. *Bio-Algorithms and Med-Systems*, 7(4):41–52, 2011.

[14] Maciej Komosinski and Adam Kups. Time-order error and scalar variance in a computational model of human timing: simulations and predictions. *Computational Cognitive Science*, 1(1):1–24, 2015. URL: http://dx.doi.org/10.1186/s40469-015-0002-0, doi:10.1186/s40469-015-0002-0.

[15] Maciej Komosinski, Adam Kups, Dorota Leszczyńska-Jasion, and Mariusz Urbański. Identifying efficient abductive hypotheses using multi-criteria dominance relation. *ACM Transactions on Computational Logic*, 15(4):28:1–28:20, 2014. URL: http://doi.acm.org/10.1145/2629669, doi:10.1145/2629669.

[16] Maciej Komosinski, Adam Kups, and Mariusz Urbanski. Multi-criteria evaluation of abductive hypotheses: towards efficient optimization in proof theory. In *Proceedings of the 18th International Conference on Soft Computing*, pages 320–325, Brno, Czech Republic, 2012.

[17] Maciej Komosinski, Agnieszka Mensfelt, Paweł Topa, and Jarosław Tyszka. Application of a morphological similarity measure to the analysis of shell morphogenesis in Foraminifera. In Aleksandra Gruca, Agnieszka Brachman, Stanisaw Kozielski, and Tadeusz Czachórski, editors, *ManMachine Interactions 4*, volume 391 of *Advances in Intelligent Systems and Computing*, pages 215–224. Springer, 2016. URL: http://dx.doi.org/10.1007/978-3-319-23437-3_18, doi:10.1007/978-3-319-23437-3_18.

[18] Maciej Komosinski, Agnieszka Mensfelt, Paweł Topa, Jarosław Tyszka, and Szymon Ulatowski. Foraminifera: genetics, morphology, simulation, evolution. http://www.framsticks.com/foraminifera, 2014.

[19] Maciej Komosinski and Jan Polak. Evolving free-form stick ski jumpers and their neural control systems. In *Proceedings of the National Conference on Evolutionary Computation and Global Optimization*, pages 103–110, Poland, 2009. URL: http://www.framsticks.com/files/common/Komosinski_Polak_EvolvedSkiJumping.pdf.

[20] Maciej Komosinski and Krzysztof Rosinski. Estimating similarity of neural network dynamics. Research report RA–10/10, Poznan University of Technology, Institute of Computing Science, 2010.

[21] Maciej Komosinski and Adam Rotaru-Varga. Comparison of different genotype encodings for simulated 3D agents. *Artificial Life Journal*, 7(4):395–418, Fall 2001. URL: http://dx.doi.org/10.1162/106454601317297022, doi:10.1162/106454601317297022.

[22] Maciej Komosinski and Szymon Ulatowski. Framsticks Network Server. http://www.framsticks.com/common/server.html, 2000.

[23] Maciej Komosinski and Szymon Ulatowski. Genetic mappings in artificial genomes. *Theory in Biosciences*, 123(2):125–137, September 2004. URL: http://dx.doi.org/10.1016/j.thbio.2004.04.002, doi:10.1016/j.thbio.2004.04.002.

[24] Maciej Komosinski and Szymon Ulatowski. Framsticks: Creating and understanding complexity of life. In Maciej Komosinski and Andrew Adamatzky, editors, *Artificial Life Models in Software*, chapter 5, pages 107–148. Springer, London, 2nd edition edition, 2009.

[25] Maciej Komosinski and Szymon Ulatowski. Framsticks web site, 2016. http://www.framsticks.com.

[26] Maciej Komosinski and Szymon Ulatowski. Multithreaded computing in evolutionary design and in artificial life simulations. *The Journal of Supercomputing*, pages 1–15, 2016. URL: http://www.framsticks.com/files/common/MultithreadedEvolutionaryDesign.pdf, doi:10.1007/s11227-016-1923-4.

[27] Pete Mandik. Synthetic neuroethology. *Metaphilosophy*, 33(1&2):11–29, 2002. URL: http://www.petemandik.com/philosophy/papers/synthneur.pdf.

[28] Pete Mandik. Varieties of representation in evolved and embodied neural networks. *Biology and Philosophy*, 18(1):95–130, 2003. URL: http://www.framsticks.com/files/common/Mandik_RepresentationsInNeuralNetworks.pdf.

[29] Raja Mohamed and P. Raviraj. Biologically inspired design framework for robot in dynamic environments using Framsticks. *International Journal on Bioinformatics & Biosciences*, 1(1):27–35, 2011.

[30] J.A. Pyles, J.O. Garcia, D.D. Hoffman, and E.D. Grossman. Visual perception and neural correlates of novel 'biological motion'. *Vision Research*, 47(21):2786–2797, 2007. URL: http://dx.doi.org/10.1016/j.visres.2007.07.017, doi:10.1016/j.visres.2007.07.017.

[31] J.A. Pyles and E.D. Grossman. Neural adaptation for novel objects during dynamic articulation. *Neuropsychologia*, 47(5):1261–1268, 2009.

[32] Piotr Sniegowski. Development of the environment for distributed computing in the Framsticks system. Master's thesis, Institute of Computing Science, Poznan University of Technology, 2013. `http://www.framsticks.com/files/common/MSc_Sniegowski_DistributedFramsticks.pdf`.

[33] Matthew Templeton. On the origin of robotic species. Master's thesis, 2011. URL: `http://www.framsticks.com/files/common/MSc_Templeton_RoboticSpecies.pdf`.