Parallel computing in Framsticks

Maciej Komosinski      Szymon Ulatowski


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 multi-threaded 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 multi-threading. 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 2
   1.1 Multi-threaded and distributed architectures in Framsticks 3

2 Two experiment definitions: prime-mt and standard-mt 5
   2.1 prime-mt experiment definition 6
   2.2 standard-mt experiment definition 8

3 Summary 12
1 Introduction

The Framsticks simulator [24], since its initial releases in 1996, has been used as a computing engine in a number of diverse applications [23], some of which are mentioned below:

- comparison of genetic encodings in artificial life and evolutionary design [20, 22, 7],
- estimating symmetry of evolved and designed agents [5],
- employing similarity measure to organize evolved constructs [9, 8, 19, 10],
- bio-inspired visual-motor coordination [6] and real-time coordination [18],
- optimization of fuzzy controllers that can be understood by a human [3],
- modeling of plastic neural nets and their evolution [4],
- modeling robots [31, 27],
- user-driven (interactive, aesthetic) evolution [23, 5],
- synthetic neuroethology [25, 26],
- analyses of brain activity evoked by perception of novel biological motion [28, 29],
- evaluation of abductive hypotheses [15, 14],
- modeling perception of time in humans [11, 12, 13],
- modeling foraminiferal genetics, morphology, simulation, and evolution [16, 17],
- modeling communication, emergence, flocking, evolution of restraint, predator–prey coevolution, semiosis, speciation, and other biological phenomena [23, 1, 2].

Many of these applications require considerable amounts of computing power, and is often the case that the more 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 multi-threading in the Framsticks processing engine allows to exploit more computing power on a single machine in a single experiment. These are the major motivations for introducing multi-threading support in Framsticks.

Other options that allow one to exploit multi-core machines and distributed machine clusters without using multi-threading include running independent, separate experiments in parallel (which lets one gather a number of independent results), and using Framsticks server [21] to both parallelize experiments on one machine and distribute experiments among different machines [30].
1.1 Multi-threaded 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 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 multi-threaded 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 the “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.

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

The basic multi-threaded 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 multi-threaded 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 [21] and manages remote servers – an example is Framsticks Java Framework [30].

Given the common interface, each or some of the Servers could run the multi-threaded version of the experiment or could be replaced by another Network Coordinator; increasing the parallelization even further (Fig. 3). In this case, the same multi-threaded 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 multi-threading and are included in the official Framsticks distribution.

The first one, prime-mt, is the basic functional example of the Framsticks multi-threaded 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 multi-threaded 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:

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

prop:
id:threads
type:d
name:Number of threads
group:Parallelized
```
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 that the number of threads will be the number of CPU cores minus 1, 2, 3, ...

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

```c
// Calculate the effective user-selected number of threads (also handles tricks with zero and negative values).
// Include standard.props_threads.inc to create the required ExpParams.threads field.

function getExpParamsThreads()
{
    if (ExpParams.threads>0)
        return ExpParams.threads;
    return Math.max(1, Simulator.cpus+ExpParams.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:

```
function testPrime(N) // extremely inefficient, just for illustration
{
    var i;
    for (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: ExpParams.from_number, ExpParams.to_number, ExpState.current_number, and ExpState.result. By convention, parameters (ExpParams) 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().

```
function onStep()
{
    if (ExpState.current_number > ExpParams.to_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

```plaintext
function onStart()
{
    Simulator.slaves.stopAll();
    Simulator.slaves.size=getExpParamsThreads();
    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 ExpParams.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

```plaintext
function scheduleChunkOfWork(slave)
{
    if (ExpState.current_number <= ExpParams.to_number)
    {
        if (slave.running)
        {
            Simulator.print("Slave simulator is still running in scheduleChunkOfWork(); This should return; // or stop() and continue scheduling, but something went wrong anyway
        }
        slave.expdef="prime";
        slave.expparams.from_number=ExpState.current_number;
        slave.expparams.to_number=Math.min(ExpState.current_number+ExpParams.chunk-1,ExpParams.to_number);
        slave.init();
        //Simulator.print("slave #"+slave.identity+" scheduled "+slave.expparams.from_number+"..."+slave.expparams.to_number);
        ExpState.current_number=slave.expparams.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 "); 
  var res=StopEvent.slave.expstate.result; 
  ExpState.packet_counter++; 
  for (var value in res) 
    ExpState.result.add(value); 
  if (Simulator.running) 
    { 
      scheduleChunkOfWork(StopEvent.slave); 
      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 () 
{ 
  g.starttime=Math.time; 
  Simulator.slaves.stopAll (); 
}
```
Simulator.slaves.cancelAllEvents();
Simulator.slaves.size=getExpParamsThreads();
var t0=Math.time;
var exported_settings=Simulator.export(null,4+16+32,-1,-1);// -1 = export all groups
//Simulator.print("exported="+exported_settings);
var i;
for(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();
ExpState.migration_time+=Math.time-t0;
Simulator.slaves.startAll();
g_slavestart=Math.time; g_slavetimeing=Simulator.slaves.size;

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 closure
  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 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); // import into a slave
    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 multi-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 (`ExpParams.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 (`ExpParams.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()
{
    if (CheckpointEvent.data >= (ExpParams.capacity * ExpParams.mix_period / 100)) // desired number
    {
        ...
    }
}
```

`standard-mt.expdef` supports one of two modes, selected by the `ExpParams.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`

function onSlaveCheckpoint()
{
    ...
    CheckpointEvent.slave.stop();
}
```
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`

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

```javascript
function onSlaveCheckpoint()
{
    ...
    if (Simulator.running && (Simulator.slaves.running==0))
    {
        Simulator.print("migrating slave simulations...");
        var t0=Math.time;
        loadFromSlaves();
        sendToSlaves();
        GenePools[0].mergeInstances();
        ExpState.migration_time+=Math.time-t0;
        Simulator.slaves.startAll();
        g_slavestart=Math.time; g_slavetimeing=Simulator.slaves.size;
    }
    ...
}
```
function loadFromSlaves() {
    Simulator.print("loading 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);
        ExpState.totaltestedcr+=s.expstate.totaltestedcr;
        ExpState.totaltests+=s.expstate.totaltests;
    }
    ExpState.migrations++;
    Simulator.checkpointData(ExpState.migrations);
}

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 shown. 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


