An execution flow for dynamic concurrent systems: simulation of WSN on a Smalltalk/CUDA environment.

Hritam Dutta, Thibault Failler, Nicolas Melot, Bernard Pottier and Serge Stinckwich

***Abstract*— The progress in wireless sensor networks is the driving force behind the idea of achieving ambient intelligence through pervasive computing and robotics. Sensor networks are fault-prone and must have high energy efficiency. Furthermore, the applications are event-driven distributed algorithms requiring high performance in bursts. Hence, high-level synchronous simulation is a necessary step for programming and analyzing such systems in terms of scalability, reliability, and other design objectives. In this paper, we introduce a simulation framework for dynamic networks in Smalltalk. Its originality lies in using massive parallelism of Graphic Processing Units (GPU) through dynamic linked libraries (DLL), for accelerating the synchronous simulation. Another important contribution is the capability of the framework to handle mobile sensors interacting with a static sensor network. The results of the paper show the feasibility of the GPU acceleration approach by showing the fast simulation time of a case study defining mobile interaction with dynamic network. A result of this work is a tight coupling of a Smalltalk VM with a set of process networks that can bring parallel execution opportunities to the VM.**

I. INTRODUCTION

Wireless sensor networks (WSN) are now being developed on a wide scale and for important applications. An example is the monitoring of parking along streets as investigated on San Francisco harbor. As many as 6000 sensors are deployed that collect and propagate parking status, allowing automated booking and guidance leading to savings on energy, pollution, and time. Many other applications are appearing in domains such as person safety, urban transports, farming, cyber-physical systems, and environemental monitoring.

Investigating solutions to build large networks requires method and tools organized around a design flow. We see this flow organized in two step similar to the one found in hardware development (see Figure 1). It involves

1) a specification and simulation loop covering
   a) network organization represented by a process architecture,
   b) local node behaviors operating in a synchronous communication model,
   c) simulation, and refinement loop back to 1a), or 1b).
2) code synthesis reproducing the specification as actual sensor program.

Hritam Dutta, Thibault Failler, Nicolas Melot and Bernard Pottier are with LabSTICC, UMR CNRS 3192, bernard.pottier@univ-brest.fr hritam.dutta@univ-brest.fr

Serge Stinckwich is with UMMISCO/MSI/IRD, Vietnam Serge.Stinckwich@ird.fr

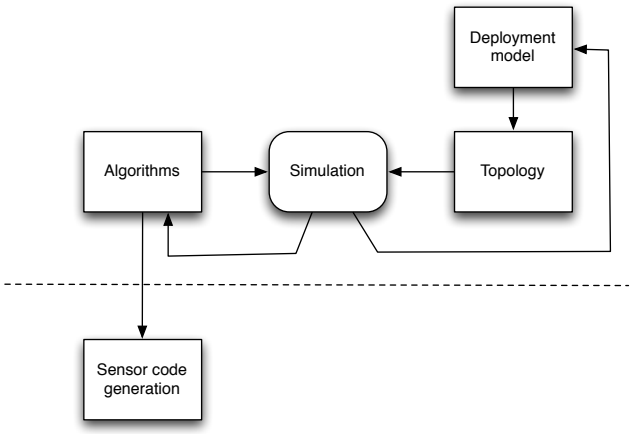

Fig. 1. General flow showing loop back at simulation level. The core model is a simple organization based on graphs, nodes, and links.

Simulation is becoming a challenging computing problem, due to the number of nodes that need to be processed, and the number of hops necessary to obtain distributed computation results. Very few tools address this question as most of the WSN simulators are based on communication medium level. Well-known low-level simulators are ns-2[1], OMNeT++[9].

In this paper, we firstly present simulation technique and tools centered on the upper part of the flow as shown in Figure 1. The NetGen framework based on Smalltalk has an internal model that associates sensors into a network with peer-to-peer links representing wireless connections. Instances of this network model like mesh and random network generators, textual representation, architectures (rings, pipelines) can be produced by the provided libraries. Secondly, synthesis of the architecture and algorithm model include code generators for simulation targeting:

- Occam code [7] whose fine-grained MIMD execution model is apt for simulation on multi-core computers,
- CUDA code whose SIMD execution model with communication in shared memory is suitable for GPU execution [6].

A novelty of the simulation framework is the coupling of a Smalltalk virtual machine (VM) with high performance CUDA execution on GPUs (Graphics Processing Unit accelerators). NetGen allows to generate dynamically:

1) a synchronous communication description in CUDA representing the network model connectivity,
2) a compute program for local node behavior with hun-

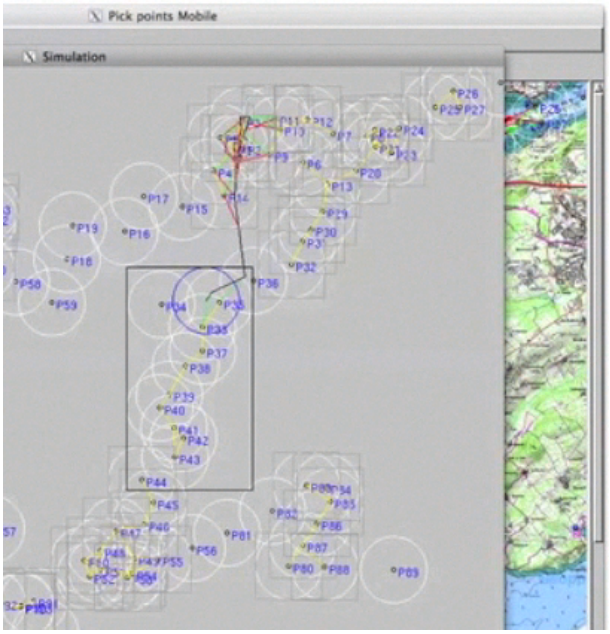

Fig. 2. A sea shore map (below) has been used to program sensor locations and mobile path (boat). The simulation consists of finding the next event between the mobile and a network, updating the network description on the GPU data structure, restarting a simulation to get a bounding box result at its next stop, then displaying this box.

dreds of processors working in real parallelism,
3) flexible binding to the high level simulator in Smalltalk for status observation, introduction of failures,
4) control of mobiles (persons, robots, and vehicles) interacting with dynamic systems.

The proposed solution has been executed on several platforms, from laptop to high-end computers.

The rest of the paper is organized as follows:

- abstract synchronous model for simulation of distributed algorithms with CUDA,
- Smalltalk virtual machine and CUDA execution relationship for dynamic configuration of networks and control of simulation,
- assessment on mobility and collected performances,
- perspective of the coupling as a way to empower Smalltalk virtual machine on high end computing devices.

II. PRINCIPLES

A. WSN semantic: Synchronous Communication model

A preliminary observation refers to the frequent need of WSN to preserve energy. As power needs to be supplied locally, wasting energy means repetitive replacement of batteries, or additional costs for local power generation by photo-voltaic panels.

An obvious way to reduce energy consumption is to switch off as many parts as possible on the sensor node, for as long as possible. Depending on the application, this is feasible due to the slow evolution of the observed phenomena. For example, sampling meteorological values or detecting presence would not require more than a measure each second. If this approach is chosen, there will be a need to awake periodically the sensor to take measurements or control actions, and then to communicate with neighbors. This behavior induces a loose synchrony in the schedule of the communications. Therefore, sensors will need to decide together at which moment to wake up, and what is the order of emissions to avoid communication collisions in the neighborhood.

A practical way to implement communication is to use a clock drift correction algorithm to manage local time and a communication protocol producing a schedule on time slots to emit and listen to the others. In the case of a single wireless communication channel, one can use a Time Division Multiple Access (TDMA) protocol to implement these exchanges. The duration of time slots depends on the delay to broadcast a packet to the neighborhood and the nominal drift of clocks for the sensor technology [8], [3].

We can therefore assume the following hypotheses for sensor programs:

1) loop on sleeping, measurement, and neighborhood communications
2) need for time adjustment protocol, i.e. clock drift correction
3) need for slot assignment protocol in a neighborhood for TDMA communication

WSN that executes these hypotheses also qualifies for the *Synchronous communication* model[5] for which there are lot of known distributed algorithms. Programs following the synchronous model are defined as sequences of initializing node state for each node (C_0), then a loop executing output message production, input/output communications (M_iN_i), update (C_i) of the local state according to the present state and incoming messages. This is explained later in the context of CUDA in Section II-E.

This choice reduces considerably the difficulties in programming dynamic network applications. It allows to target many different kind of parallel architectures for execution of simulation software. By separating the computation and communication into cycles, we also have a good way to setup observation, control, and debugging services.

B. Abstract representation in Smalltalk

In this section, we present an high-leverl representation of synchronous model of execution for distributed algorithms in Smalltalk. A representation of a system can be produced by separating node behaviors into objects representing communication channels and blocks for setting the initial node state and executing transitions.

As a pure software example expressed in Smalltalk, the method **synchronousBehaviour** from class **SynchronousBlock** produces a block (aka lexical closure) that operates the synchronous model. Instance variables are input and output shared queues, **initialBlock** and **transitionBlock** that handles a distributed algorithm as shown in Figure 3. The last variable is **step** that counts a number of static loops for the algorithm.

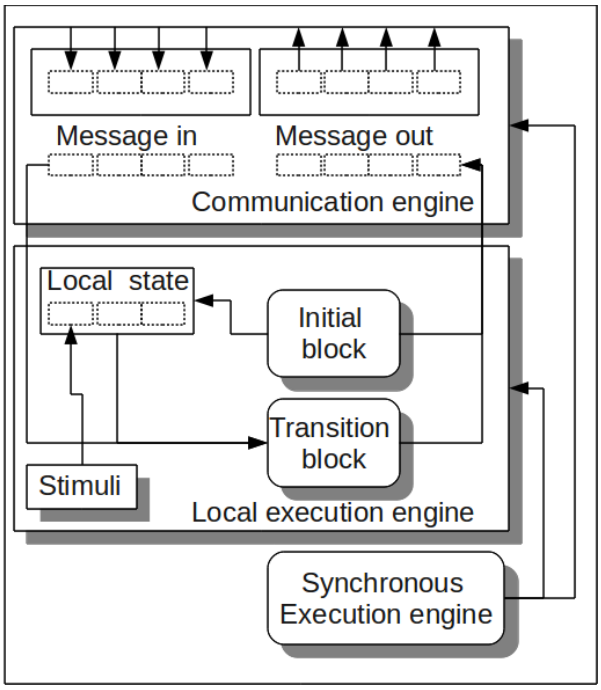

Fig. 3. Software architecture for a node, showing communication mechanisms (abstracted), message buffers (generated), program defined blocks for initializing and state transitions. The synchronous model can thus be implemented in various ways, such as pure software process systems, hardware supported simulators, and physically distributed systems such as WSN.

```
synchronousBehaviour
  | messageOut messageIn |
  messageOut := Array new: self outputs size.
  messageIn := Array new: self inputs size.
  "Return a block to be executed later"
  ^
  [
  " initial state "
  self initialBlock value: self value: messageOut.
  " loop step times"
  self step timesRepeat:
      ["send messages asynchronously"
       [self outputs keysAndValuesDo:
           [:i :output | output nextPut:
                          (messageOut at: i)]] fork.
        "receive messages"
        self inputs keysAndValuesDo:
           [:i :input | messageIn at: i
                                 put: input next].
        "automata transitions and next messages"
      self transitionBlock value: messageIn
                          value: messageOut]]
```

In order to define specific local behavior, we then write methods in class **SynchronousBehaviour**, or subclasses. The **max** example method given below is enough to implement leader election on a ring.

```
max
  | max transition initial id |
  max := 0.
  id := self nextIdea.
  max := id.
  transition :=
      [:in :out |
      in do: [:val | max := max max: val].
      1 to: out size do: [:i | out at: i
                                 put: max]].
  initial := [:sb :out |
              1 to: out size
                do: [:index | out at:  index
                                 put: id]
  ^Array with: initial with: transition
```

Finally, we can bind processes together by program and produce an execution as shown in method **testRing**. One of the aim of NetGen is to be able to produce such architecture for a variety of situations.

```
testRing: n
  | processes block0 linksPrev
    links0 blockNext linksNext
    syncBlock barrier item |
  processes := OrderedCollection new.
  SynchronousBehaviour makeIds: n.
  block0 := SynchronousBehaviour max.
  barrier := SharedQueue new: 1.
  linksPrev := links0 := Array with:
                              (SharedQueue new: 1).
  n - 1 timesRepeat:
      [blockNext := SynchronousBehaviour max.
      linksNext := Array with:
                          (SharedQueue new: 1).
      syncBlock := SynchronousProcess
            block: blockNext
            inputs: linksPrev
            outputs: linksNext
            steps: n
            barrier: barrier.
      processes add: syncBlock.
      linksPrev := linksNext].
  syncBlock := SynchronousProcess
        block: block0
        inputs: linksPrev
        outputs: links0
        steps: n
        barrier: barrier.
  processes add: syncBlock.
  "start!"
  processes do: [:sb | sb resume].
  "wait terminations"
  1 to: n do: [:i | item := barrier next]
```

Figure 4 displays performance numbers for different ring size. It is the idea of this paper to smooth the path between an abstract model for networks and execution to the best hardware now available: multi-cores and, when possible GPUs. For this reason, the next section explores the possibility of accelerating simulation in Smalltalk VM by linking automatically to a generated CUDA DLL enabling execution on GPUs.

C. Time driven simulation and communication channels

Synchronous time-driven simulation is known to be sub-optimal in comparison with event-driven or more aggressive simulation methods. However it is a good choice for WSN due to the periodicity of communications and the need to simulate random behavior in the network. Our project is currently developing two time-driven simulation kernels to obtain the necessary computing efficiency: MIMD/Multi-core approach based on concurrent threads, and SIMD/GPU approach based on inherent data parallelism existing in synchronous execution of the WSN. MIMD has been presented in [4], while this paper discusses specification and simulation mechanism for SIMD execution on GPUs. The synchronous

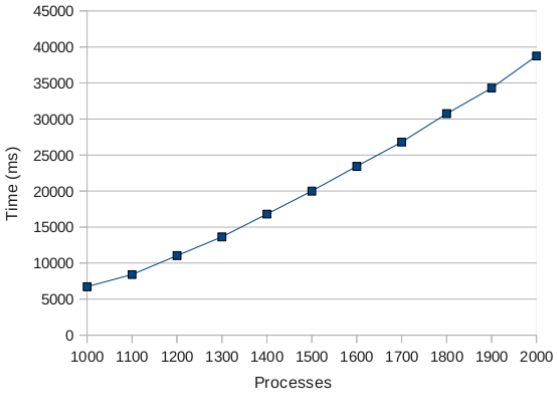

Fig. 4. Leader election on rings, complexity is O(n^2). Synchronous model is executed as communicating processes communicating on shared queues in a 1-thread Smalltalk, Visualworks 7.7, i5 CPU

model can be implemented by associating a process to each node in the network. Isolation of the steps of the synchronous model execution requires the synchronization of processes on finished communication step (M_iN_i section II-A). One can reproduce this behavior in several ways through:

- *blocking channels*, as used in Communicating Sequential Processes (CSP), and Occam [7],
- *synchronization barriers*, as available in the SIMD execution style and current GPUs [6].

We wish to use the power of hundreds of processing elements in a GPU for accelerating the simulation of thousands of sensor nodes. We can use the SIMD nature of GPUs to decide when a synchronization barrier is reached. A key point remains on, how to simulate communications, i.e. data exchanges on wide networks having irregular connections.

D. Channel arrays in memory

The execution architecture can be seen as a wide linear array of processes. Each of these process has local and global variables. We use local variables to implement the process state of sensors, while communication buffers are implemented as global variables. Global variables can be modified from the processing engine, therefore the communication structure and contents can be changed dynamically.

We have two channel arrays for input and output. For each process they are described using:

- output destination channel array says at which node and which position a message data must be written. This is generally the same data updated in local node buffers.
- input destination channel array says at which node and which position a message data must be read, to be stored locally. This is generally the data from local node buffers needed for computing new node state.

Additional description includes number of outgoing and incoming links, plus spare mapping for links representing dynamic communications in the case of mobiles. Since, here the number of channels can vary depending on the position of the mobile. All these data structures are generated automatically by the simulation framework in Smalltalk.

```
typedef struct s_mapped /* communication pointers */
{
  int node; /* a neighbour index */
  int channel; /* an index in the neighbour channels */
} mapped;

typedef struct sChannels
{
  int nbOut;
  int nbIn;
  int nbDyn;
  mapped write[MAX_FANOUT]; /* MAX_FANOUT from NetGen */
  mapped read[MAX_FANOUT];
  mapped writeDyn[DYNAMIC_CHAN];
  mapped readDyn[DYNAMIC_CHAN];
} channels;

channels channels_h[] =
{  /* the generated communication structure*/ };
```

Data buffers are application dependent and include type variants according to the data carried on communication links. An example C type declaration is the **message** described below, while input and output communication buffers are arrays of such messages.

```
typedef union /* type variant for message contents */
{
  int val; /* an integer */
  mesBuff buff[BUFFSIZE]; /* a packet */
  int zone[4];/* a computed rectangle */
  s_closer closer; /* an application type */
} msg_data;

typedef struct
{
  int type, start;
  msg_data data;
} message;

message messageIn[MaxNode], messageOut[MaxNode];
```

E. Synchronous cycle

The synchronous model works on this memory mapped communication system with a generic engine that undertakes message transfers and call application specific procedures for state transition. The program of this generic machine is as follows:

1) C_0
   a) initialize local variables from an available global array defining initial state
   b) compute first output messages
2) **loop**, with three stages and barriers:
   a) M_i: read input buffers by looping on the communication control data structure,
   b) N_i: write output buffers by looping on the communication control data structure,
   c) C_i: read message from input buffers for updating update local variables representing the node state.

This abstract machine can be reproduced in lot of environments, including state-of-the-art GPUs featuring as much as 500 processing elements using CUDA programming model. In the machine cycle, we need to generate data structure of

message for the communication. Currently, the local node behavior reflecting state transition and fixed support for execution of communications (i.e., send and receive) must be handcrafted by the *firmware* developer.

The next section explains how this abstract machine is bound to the NetGen generator, and how it can interact with the high level part of the simulation, including mobiles and behavior representation.

III. NetGen/CUDA interactions

A. Software architecture

The core data structure in NetGen framework is the network description made of nodes and associated links to the neighbours. This structure is intended to be processed and translated for various purposes, including visualization and simulation. In the case of simulation, generation is based on the separation of two concerns.

- topology description
- local behavior description

The local sensor behavior and topology descriptions may be expressed in target languages depending on the simulation platform, for example, Occam, CUDA, Smalltalk, and others. A first technical question is how to assemble program contents to produce a correct source code. When using CUDA for simulation, the target is not a standalone program implementation, but a dynamic library (DLL), that is coupled to the host simulation environment in Smalltalk. The CUDA DLL is generated automatically from Smalltalk simulation framework and contains:

- initial communication control defined as an array of structures, representing send and receive executed for the network being simulated (section II-D)
- *firmware* functions for describing the local node behavior (currently handcrafted)
- glue and API allowing Smalltalk to control and to access variables and communication support on the GPU (library)

The DLL is compiled and bound transparently *and dynamically* for each new network to be processed, hence producing a new dedicated simulation engine. To do this the simulation framework generates, assembles and compiles the CUDA library code, then bind it as a class based DLL interface to the Smalltalk simulation environment. The method from the DLL interface allows to start simulation, to control the execution, to fetch simulation results. Furthermore, at each synchronous step the data structures can be changed for reflecting dynamic and mobile changes of the network structure (see figures 2 and 5). These include:

- adding/disabling nodes
- changing the network connectivity to represent mobile objects.

Therefore, the code and library generation for simulation is tightly coupled with NetGen. Technically, we are using Visualworks (a commercial Smalltalk implementation) platform that can parse C header files and bind to C libraries.

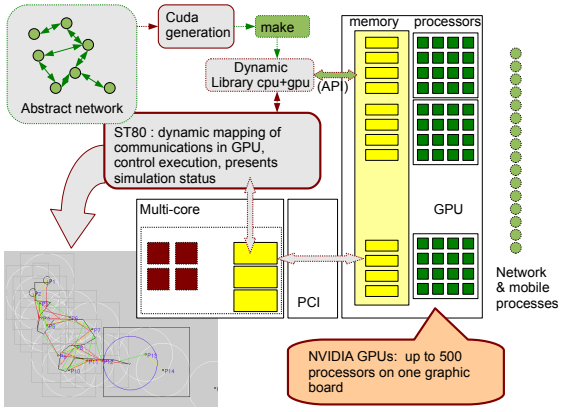

Fig. 5. Binding the accelerator in the compilation and execution flow: the core model is used to produce CUDA communication connectivity, and then to interact with the communication engine during execution.

The CUDA code is hidden behind a C interface layer running on the host computer under the Smalltalk virtual machine.

B. Assessment: a mobility case study

As shown figure 2, the case of mobiles navigating across network of sensors has been simulated. Developing similar simulation can be done at application engineering level, by specifying topologies, and selecting sequence of procedures to produce functional *firmwares* for the sensor nodes. The following steps are undertaken for simulation of mobile interaction with sensor network:

- a distribution of sensors is picked on an image or a map.
- a mobile path is defined as a serie of points.
- a wireless range is selected and a network connectivity is computed based on this range, defining the core model organization.
- CUDA code generator is called to produce a data structure that mimics network connectivity, including geometric characteristics such as sensor positions.
- the dynamic library is regenerated and bound to the Smalltalk virtual machine.
- execution is started and all the networks are simulated for computing diameters, electing leaders, or any other specified local behavior for distributed algorithm. Leaders will serve to label networks.
- on the high level side, simulation loop proceed by following the mobile path carrying connectivity events. Events are points where the connectivity changes. More details and movies can be found in project homepage[1].
  - at each new event, the connectivity on the CUDA is updated to reflect the new network.
  - when a new connection is established, the network label is fetched and a distributed algorithm function is called to compute a *bounding box* for example.

[1] http://wsn.univ-brest.fr/

– when the distributed processing returns, the bounding box is passed to Smalltalk and the graphic representation is updated.

C. Performance and execution characteristics

For observing scalability of execution and application aspects, two kind of measures have been collected.

First we have generated a simulation program for networks having a number of nodes ranging from 500 to 3000 nodes. The distributed behavior is to compute diameter and elect leaders, inserting a point that change network connectivity, and reprocessing initial characteristics. The CUDA programs were compiled with a fixed C program to check performances on platforms going from low end laptop GPU to high end Nvidia GFX480 GPU. Figure 6 shows the relative acceleration obtained on these processors and the efficiency of their architectures as compared to sequential execution. A second

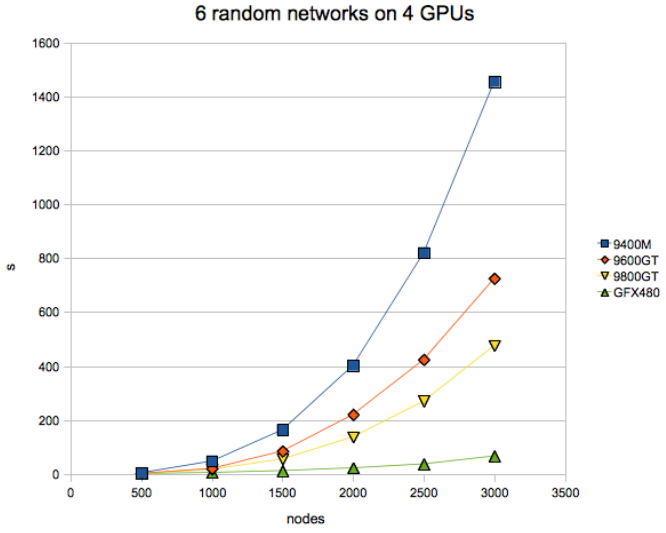

Fig. 6. Execution times in seconds for 4 different GPUs. The algorithm computes network diameters, leaders, then inserts one node, and re-executes previous computations. The 9400M is a Mac Mini hardware, others are current generation Linux i5/i7 boxes. The first stage that computes diameter, given the knowledge of the number of nodes, can be compared to the results shown figure 4.

experiment concerns the mobility aspect, i.e., by moving a mobile on a fixed path on which it can surely discover each sensor. The sensor distribution is fixed, but the wireless range is varying, the lower ranges producing networks with higher connectivity as visible on the upper curve in Figure 7. The system converges to a single network. The diagram also shows collected values for maximum fan-out which is used to define the size of communication buffers (section II-D) and number of channels.

IV. Conclusions and Future Works

As far as we know, it is the first time that a Smalltalk VM is associated to high performance GPUs with dynamic generation of GPU programs and dynamic control of the execution. The result is a very powerful and flexible approach

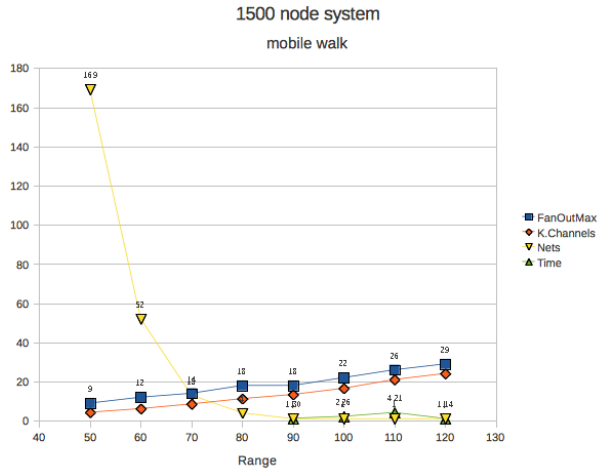

Fig. 7. Several 1500 node networks are randomly produced with an increasing wireless range. A mobile will walk on the same surface, connecting to, and disconnecting from networks. The yellow curve shows a decreasing number of found networks. Other statistics display the node fan-out, and number of channels.

with clear benefits for simulation of a large distributed WSN. The computing efficiency scales with the number of processors and the available memory, with the possibility to follow the progresses of GPUs.

We have demonstrated the possibility to control the integration of a mobile in a set of fixed sensor networks, and the possibility to induce failures in the networks or communications. Besides the capability of high speed simulation, the association of the Smalltalk VM with a GPU also allows accelerated execution of applications such as cellular automata, neural networks, genetic algorithms.

The benefits of this work may exceed the case of WSN. The machine should be considered as an irregular execution architecture associated to a Smalltalk controller. In the future, we expect to map program representations to GPUs to support stream execution, and synthesize node *firmwares* in similar ways as [2].

References

[1] ns2 home page. http://www.isi.edu/nsnam/ns/.

[2] German Fabregat, German Leon, Olivier Le Berre, and Bernard Pottier. Embedded system modeling and synthesis in oo environments. a smart-sensor case study. CASES'99 proc. Washington DC, 1999.

[3] Rui Fan, Indraneel Chakraborty, and Nancy Lynch. Clock synchronization for wireless networks. In *In Proc. 8th International Conference on Principles of Distributed Systems (OPODIS)*, pages 400–414, 2004.

[4] Guillaume Kremer, Jimy Osmont, and Bernard Pottier. A process oriented development flow for wireless sensor networks. IWST, ESUG'09, Brest, Sep 2009.

[5] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, USA, 1996.

[6] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[7] Dick Pountain and David May. *A tutorial introduction to Occam programming*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[8] Jana van Greunen and Jan Rabaey. Lightweight time synchronization for sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 11–19, New York, NY, USA, 2003. ACM.

[9] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Simutools '08*, Brussels, 2008. ICST.