# Parallel Simulation of Embedded S-Net Processes

Thomas C. Henderson and Sriram Karra
University of Utah
Salt Lake City, UT 84112 USA
tch@cs.utah.edu

## Abstract

*We have developed Smart Sensor Networks (*S-Nets*) which are comprised of non-mobile devices which can compute, communicate and sense. We have demonstrated through simulation experiments the feasibility and performance capabilities of this approach, and we are now studying them via parallel simulation of the devices. In particular, we are looking at distributed algorithms for leadership protocols, coordinate frame calculation and level set based shortest path computation. These algorithms are being studied on an SGI Origin 2K, and include a Unix process approach, a threads implementation, as well as an MPI implementation.*

## 1. Introduction

Our major area of study is the robustness of embedded systems, including relevant organizational principles (e.g., spatial structure, redundancy, modularity, diversification, and hierarchy), operational issues (e.g., noise, stability, resistance, resilience, and recovery) and consequences for system fitness, adaptability and evolvability. As a particular effort, we study *Smart Sensor Networks (S-Nets)*; these are embedded systems comprised of algorithms, computer architectures, communications capability, and sensors. The *S-Net* forms a rich information field to be exploited by various mobile autonomous agents.

Current computing systems are plagued with the inability to adapt or to perform outside a very narrow range of specified parameters. The failure of a single element or module can cause the whole system to fail. We study how robustness relates to system structure, modules and their redundancy. This includes communication methodology, power allocation, multi-sensor integration, computational reliability, and element diversification (e.g., analog vs. digital and qualitative vs quantitative).

The utilization of nonmobile, distributed sensor and communication devices by a team of mobile robots may offer performance advantages in terms of speed, energy, robustness and communication requirements. Models of mobile robots with on-board sensors, a communication protocol and the *S-Net* system have been established. Algorithms are defined for the *S-Net* which perform cooperative computation and provide information about the environment. Behaviors include robots going to or surrounding a temperature source. We have run simulation experiments that show that the *S-Net* performs well, and is particularly robust with respect to noise in the environment. System cost versus performance has been studied, and guidelines are formulated for which the *S-Net* system out-performs the *non-S-Net* system.

At one extreme, mobile robots can be provided with a wealth of on-board sensing, communication and computational resources [1, 2]; at the other extreme, robots with fewer on-board resources can perform their tasks in the context of a large number of stationary devices distributed throughout the task environment [3]. We call the latter approach the *Smart Sensor Network*, or the *S-Net*. We have performed simulation experiments on sequential machines using software (C and Matlab), and the performance of robot tasks with and without the presence of an *S-Net* (i.e., a set of distributed sensor devices) has been evaluated in terms of various measures. See [4, 5] for a more detailed account.

Our ultimate goal is to develop the *S-Net* as an embedded system. In particular, this requires:

- Develop algorithms amenable to embedded systems use.
- Develop prototype computer architectures for *S-Nets*.
- Model and study the communications aspect of *S-Nets*.
- Model and study one or more sensor modalities for *S-Net* exploitation.

In order to facilitate the study of these different aspects of the system, we are developing a distributed simulation capability. This will allow us to study: (1) distributed algorithms (2) communication models and experimentally determined parameter values, (3) the *S-Net* architecture (includes computation, sensing and communication), and (4) sensor models and experimentally determined parameter values. Local and global frames are defined and created. A method for the production of global patterns using reaction-diffusion equations is described and its relation to multi-robot cooperation has been demonstrated. In addition, we have shown how to compute shortest paths in the *S-Net* using level set techniques [7]. Heretofore, however, all these have been developed in simulation as sequential codes.

The results of our sequential simulation experiments helped us better understand the benefits and drawbacks of the *S-Net*. We have shown that for behaviors of one mobile robot going to a temperature source, and multiple mobile robots surrounding a temperature source, in the ideal situation (which means no noise), the *S-Net* takes more time and distance. When noise is added in, which is more realistic, the *S-Net* system is more robust than the non-*S-Net* system. For the task of multiple mobile robots going back and forth to a temperature source, there are thresholds above which the *S-Net* system out-performs the non-*S-Net* system.

Here we will describe distributed frameworks for the simulation of the *S-Net*. The set of distributed algorithms studied includes: (1) *S-Cluster* formation with leader, (2) coordinate frame calculation, (3) gradient calculation, (4) reaction-diffusion computation, and (5) shortest paths using level sets. The distributed simulation frameworks include:

- Unix processes,
- SGI Origin 2K threads, and
- SGI Origin 2K MPI.

We still use the Unix process framework, as well as the MPI implementation. However, the threads version is no longer in use and is described in Appendix A. (Details of algorithms are given there, too.)

## 2. Basic Code Layout and Unix Process Simulation

The code must handle a set of independent S-elements, and each S-element has associated state (known to itself):

- unique ID: *UID*
- leader flag: $leader \in True, False$
- broadcast range: *range*
- sensor value; e.g., temperature

and state necessary for the simulation (unknown to the S-element); e.g.,

- location: $(x, y, z)$
- working or not

The *UID* is assigned by the user, the locations are either randomly picked or assigned by the user.

For example, the following layout has been used:

```
3           8
|           |
|           |
2---4---5---7---9
|           |
|           |
1           6
```

with nodes located at:

| *Node* | *X-Location* | *X-Location* |
|---|---:|---:|
| 1 | 0 | 0 |
| 2 | 0 | 1 |
| 3 | 0 | 2 |
| 4 | 1 | 1 |
| 5 | 2 | 1 |
| 6 | 3 | 0 |
| 7 | 3 | 1 |
| 8 | 3 | 2 |
| 9 | 4 | 1 |

We assume a broadcast range of 2.1 units.

The S-elements communicate through simulated RF *broadcast* and *receive* functions, which can be implemented in various ways. Here the RF communication is handled through files.

In the overall simulation, the first step is to determine clusters of S-elements which act together to provide redundancy and robustness of sensing, as well as to provide a local coordinate frame to communicate direction to autonomous agents which use the *S-Net*. A cluster consists of a leader S-element and set of S-elements within the broadcast range of the leader. The leader is selected as that S-element having the lowest *UID*[8].

To determine the clusters and leaders, each S-element must do the following things:

- determine its neighbors, and
- determine whether it is a leader or not.

The leadership algorithm is a 3-phase algorithm:

1. **Phase 1**: Each S-element broadcasts its *UID*.
2. **Phase 2**: Each S-element receives the broadcasted *UID*'s and this sets its neighbors; it then follows a protocol to determine the distance between it and its neighbors.

3. **Phase 3**: Each S-element determines if it is a leader; if so, this is communicated to its neighbors, who determine that they are not leaders, and broadcast the known cluster to their neighbors. This allows the formation of any further clusters.

For more details on the algorithm and its correctness, see [8].

Once leadership is determined, each leader calculates a local coordinate frame. This is done by finding 3 S-elements in the cluster such that they are not collinear. An agent can then use this coordinate frame after determining whether it is right- or left-handed.

The Unix process simulation is a relatively convenient framework for developing algorithms on a small number of S-elements. Once too many processes are created, the system will not perform very well.

## 3. SGI Origin 2K MPI Simulation

The major simulation in use now is an MPI implementation; MPI stands for Message-Passing Interface, and is a specification for a set of C, Fortran and C++ routines that provide message passing capability[9]. The basic Unix program is easily modified to run in the MPI environment, where each S-element is assigned a unique processor. In fact, MPI allows overloading of logical processes on the physical processors.

We have already described the algorithms and problem type, and here we simply give results of running the leadership protocol and calculation of local coordinate frames.

We use the 9-node problem described in Section 2. The neighbors found for the 9-node system are described in the execution trace as:

```
Neighbors of Device 5: 2 4 6 7 8 9
Neighbors of Device 1: 2 3 4
Neighbors of Device 4: 1 2 3 5 7
Neighbors of Device 7: 4 5 6 8 9
Neighbors of Device 3: 1 2 4
Neighbors of Device 6: 5 7 8 9
Neighbors of Device 8: 5 6 7 9
Neighbors of Device 2: 1 3 4 5
Neighbors of Device 9: 5 6 7 8
```

The resulting leaders and clusters are described as:

```
Device 1 leader:  Yes
Device 3 leader:  No
Device 4 leader:  No
Device 2 leader:  No
Device 5 leader:  Yes
Device 8 leader:  No
Device 9 leader:  No
Device 7 leader:  No
Device 6 leader:  No

Cluster 0:  1  2  3  4
Cluster 1:  2  4  6  7  8  9
```

The coordinate frames calculated are:

```
Device 1 leader:  Yes
Frame nodes: (1,2,4)
Frame(1) dist: (1.000000,1.414214,1.000000)
Device Frame (1)
F[0,:]: (0.000000,0.000000,0.000000)
F[1,:]: (1.000000,0.000000,0.000000)
F[2,:]: (1.000000,1.000000,0.000000)

Device 5 leader:  Yes
Frame nodes: (5,2,6)
Frame(5) dist: (2.000000,1.414214,3.162278)
Device Frame (5)
F[0,:]: (0.000000,0.000000,0.000000)
F[1,:]: (2.000000,0.000000,0.000000)
F[2,:]: (-1.000000,1.732051,0.000000)
```

The code is organized so that one process serves as the simulation master and knows the S-element locations, etc., and calculates the distances between S-elements in order to control the broadcast and receive semantics. Once it performs an MPI broadcast, the other processes can proceed. All the other processes handle a unique S-element.

### 3.1. Scaling Results

A sequence of experiments were run using from 1 to 9 S-elements laid out as shown in Section 2. The times for these 9 runs are:

| *S-elements/Processors* | *Execution Time* |
|---|---|
| 1 | 3.90 |
| 2 | 4.52 |
| 3 | 6.94 |
| 4 | 9.51 |
| 5 | 13.63 |
| 6 | 18.51 |
| 7 | 17.29 |
| 8 | 26.54 |
| 9 | 41.99 |

There were from 2 to 10 processors used, respectively, as there is one process per S-element, and one master simulation process.

Figure 1 shows the times taken to run from 1 to 9 S-element problems, as well as a least squares linear fit to the data. Figure 2 shows the per process time across the same set of experiments.

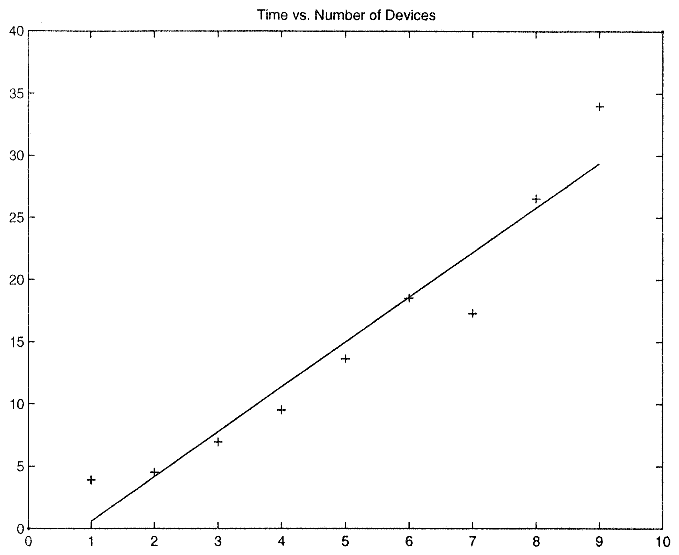

Figure 1. Times for 1 to 9 S-elements on 1 to 9 processors)

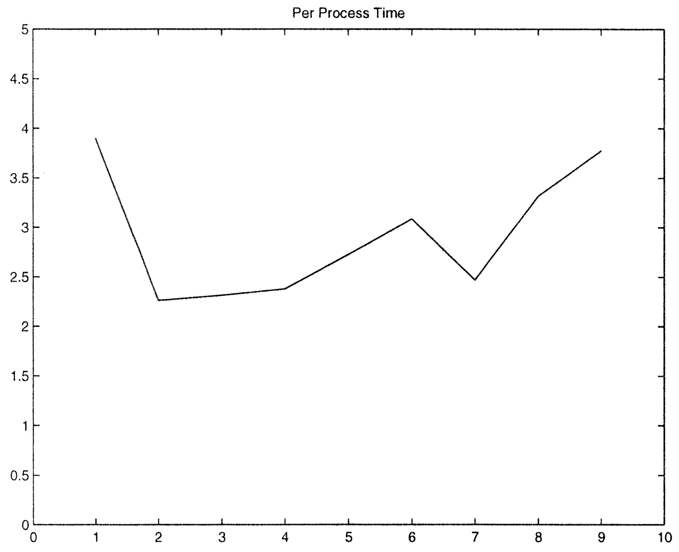

Figure 2. Times for 1 to 9 S-elements on 1 to 9 processors)

## 4. Summary

The results so far are very encouraging, and the MPI implementation demonstrates good scaling. Our future MPI work includes:

- run with large number of S-elements
- study distributed reaction-diffusion algorithms
- study distributed level set algorithms
- study communication issues.

## Appendix A - SGI Origin 2K Threads Simulation

The simulation code was written using the POSIX threads (pthreads) library on an SGI Origin 2000. The *S-Net* was modeled as a set of threads all executing the same code, each thread corresponding to an S-element. The communication between threads was done using mutexes and condition variables provided by the pthreads library. The goal was to abstract an S-element into a thread, and to spawn multiple threads this way, so as to model the independent device nature of the S-elements.

### Cluster Formation

To simulate the cluster formation phase, a number of assumptions are made:

- Each thread has a unique ID that is assigned at time of spawning.
- There is a global queue which is shared by all the threads for communicating with other threads.
- Each device is aware of its position in some global frame. This point is worthy of further elucidation. It is expected that real S-element will not have this luxury. This information is provided to S-elements to decide which of the others are within range. In the physical world, this can be trivially decided - if the device can receive RF signals from any other device, then they are neighbors, otherwise they are not.

Each thread runs the following algorithm:

- Construct a message with its ID and its positional information and enqueue it in the shared queue.
- Go to sleep for fixnum seconds (2, say)
- Go through the entire queue, and check to see if any of the other S-elements are within range. If so, add them to the list of neighbors.
- Remaining neighbors = list of neighbors (copy the list)
- while (!done) do
  - If own ID is smallest in remaining neighbors
    - self is a leader
    - Form a new cluster comprised of all neighbors in cluster
    - Broadcast this information
    - done = true
  - else

```
        * sleep until neighbor with lowest ID makes up its mind
        * list = Obtain the list it broadcasts
        * if self exists in that list
            1. self is not leader
            2. broadcast the obtained list
            3. done = true
        * else
            1. foreach item in list do
            (a) remove item from remaining neighbors
            2. endfor
        * endif
    – endif
• endwhile
```

One important thing to note is that a single S-element can be a constituent of more than one cluster. This is not just an artifact, it is absolutely essential for inter-cluster navigation if the clusters each calculate their own local coordinate frames. It would not be possible to transform one coordinate system to another unless we know the coordinates of at least two points in both coordinate frames. This is possible only if there are some S-elements in both frames under consideration.

**Coordinate Frame Determination**

Now, we are faced with the following problem: *Given a cluster of S-elements and the distances between pairs of them, form a 2-D coordinate frame and compute the positions of all the S-elements in the cluster.*

**Local Frame Formation**

Let us consider three S-elements $S_1$, $S_2$, and $S_3$, with distances between them being $d_{12}$, $d_{13}$ and $d_{23}$. Now, care should be taken such that the three S-elements are not collinear. This can be done by using the Law of Cosines; if:

$$| (d_{23} * d_{23} - d_{12} * d_{12} - d_{13} * d_{13})/(2.0 * d_{12} * d_{13}) |$$

is close to 1, then the points are collinear.

We can choose $S_1$ to be $(0,0)$, the origin in the local frame, and the line $\overline{S_1 S_2}$ to be the X-axis of the frame. Thus the coordinates of $S_2$ are $(d_{12}, 0)$.

Now we can compute the coordinates, $(x_3, y_3)$, of the $S_3$ by solving:

$$x_3 = \frac{d_{12}^2 + d_{13}^2 - d_{23}^2}{2d_{12}}$$

$$y_3 = \sqrt{d_{13}^2 - \frac{d_{12}^2 + d_{13}^2 - d_{23}^2}{2d_{12}}}$$

**Coordinates for other places in the frame**

Let $(x_i, y_i)$ be the location of $S_i$. Then $x_i$ and $y_i$ can be computed in the coordinate frame, provided the distances to $S_1$, $S_2$ and $S_3$ are known.

Let $d_{i1}$, $d_{i2}$, and $d_{i3}$ be the distances of $S_i$ to $S_1$, $S_2$ and $S_3$, respectively. Then:

$$x_i = \frac{(y_3 - y_1)C_1 + (y_1 - y_2)C2}{y_1(x_3 - x_2) + y_2(x_1 - x_3) + y_3(x_2 - x_1)}$$

$$y_i = \frac{(x_1 - x_3)C_1 + (x_2 - x_1)C_1}{y_1(x_3 - x_2) + y_2(x_1 - x_3) + y_3(x_2 - x_1)}$$

where,

$$C_1 = \frac{1}{2}(d_{i1}^2 - d_{i2}^2 - x_1^2 + x_2^2 - y_1^2 + y_2^2)$$

and

$$C_2 = \frac{1}{2}(d_{i1}^2 - d_{i3}^2 - x_1^2 + x_3^2 - y_1^2 + y_3^2)$$

**System Details**

The system has been built on a 64-node SGI Origin 2K. The call line is:

```
./leader [-t <int>] [-n <int>]

-t <n> : Set the gloabl trace level to n.

-n <n> : Simulate an S-Net with n S-elements.
         The default is 50.
```

Given a number of S-elements to simulate, the driver routine generates global positions at random for that many S-elements so that they are within a square grid (whose size can be controlled as mentioned below) and spawns that many threads.

There are more parameters that can be set, but there is no command line interface to them. The two most interesting parameters that can be tweaked in the source are BROADCAST_RANGE and NUMBERR. These are #defined macros in the file driver.c. The former is a number (int or float), that limits the range of the communication between S-elements. The latter is an integer that is half the size of a square that acts as a bounds for the randomly generated coordinates. In other words, if NUMBERR, is 3, then the S-elements' coordinates are generated at random and adjusted such that they all lie in +3 to -3 in both x and y directions.

A sample output is given below:

```
=========Info for Thread #<37>===========
Thread id             = 65574
x_pos of S-Element = 1.970
y_pos of S-Element = -0.130
```

```
neighbors           =  0 1 4 5 6 8 9 12 15
                      16 20 22 24 26 27 29
                      31 34 35 39 41 42 45
                      48 49
leader_p?           = 0
leader->index       = 0
Local Coordinate Information:
        In Cluster #1, x_pos = 1.646,
                       y_pos = -1.524
        In Cluster #3, x_pos = 1.354,
                       y_pos = 0.000

cluster #           = 1
========================================
```

### Results and Analysis

The implementation makes poor use of threads, and is difficult to run. In maintaining a very explicit communication model between the S-elements, the shared memory paradigm has caused trouble. Although effort was made to abstract the communications code into modular functions (for the most part), there are many instances when making it explicit is cumbersome.

## Acknowledgment

Partial support for this effort was provided by NSF grant ASC-89-20219.

## References

[1] Bares J E, Wettergreen D S 1999 Dante II: Technical description, results, and lessons learned. *Int J Rob Res.* 18(7):621-649 July

[2] Smith R, Frost A, Probert 1999 P A Sensor System for the navigation of an underwater vehicle. *Int J Rob Res.* 18(7):697-710 July

[3] Henderson T C, Dekhil M, Morris S, Chen Y, Thompson W B 1998 Smart Sensor Snow. *IEEE Conf IROS.* Oct, pp 1377-1382

[4] Chen Y 2000 S-Nets: Smart Sensor Networks. MS Thesis, University of Utah

[5] Henderson T C, Chen Y, 2000 Smart Sensor Networks. *IEEE Conf ISER.* Dec, pp 85-94

[6] Lynch N 1996 *Distributed Algorithms.* Morgan Kaufman Pub, San Francisco

[7] Sethian J A 1999 *Level Set Methods and Fast Marching Methods.* Cambridge University Press, Cambridge UK

[8] Henderson T C 2001 Leadership Protocol for S-Nets. *IEEE Conf MFI* Aug, to appear

[9] Gropp W, Lusk E., Skjellum A. 1999 *Using MPI.* MIT Press, Cambridge MA