# SPEDE: Simple Programming Environment for Distributed Execution

James Gochee
*Dartmouth College*

# SPEDE – A Simple Programming Environment for Distributed Execution

JAMES GOCHEE

Senior Thesis
Department of Math and Computer Science
Dartmouth College, 1992

One of the main goals for people who use computer systems, particularly computational scientists, is speed. In the quest for ways to make applications run faster, engineers have developed parallel computers, which use more than one CPU to solve a task. However, many institutions already posses significant computational power in networks of workstations. Through software, it is possible to glue together clusters of machines to simulate a parallel environment. SPEDE is one such system, designed to place the potential of local machines at the fingertips of the programmer. Through a simple interface, users design computational objects that can be linked and run in parallel. The goal of the project is to have a small portable environment that allows various types of computer systems to interact. SPEDE requires no altering of the kernel and does not require system privileges to use. Using SPEDE, programmers can get significant speedup for computationally intensive problems. As an example, a Mandelbrot image generator was implemented, that attained a five-fold speedup with eight processors.

## Introduction

With the advent of parallel computing, users have been able to achieve significant speedup of computationally intensive applications. Parallel machines can be quite expensive, so researchers have looked to clusters of workstations as an alternative. Through software, machines on a network can be linked together, forming a virtual parallel computer. SPEDE (Simple Programming Environment for Distributed Execution) is a tool for UNIX programmers who want to write parallel applications that run on clusters of workstations. Because of the diversity of machines found at many sites, including Dartmouth College, SPEDE is designed to allow a heterogeneous group of machines to work together. SPEDE is a user level system, meaning no system privileges are needed for installation or use. This is admittedly redundant, requiring each user to run the entire system separately, but has many advantages. SPEDE uses only UNIX facilities that are universally available to the average user. For instance, SPEDE requires every participating machine to run a special daemon. This daemon is executed via the UNIX remote execution facility (rsh). This allows the daemon to be quickly started and stopped without modifying the kernel or using privileged facilities. In this paper, the term *session* is used to describe an interaction with SPEDE that lasts as long as any remote entity is running. When all processes associated with SPEDE have been terminated, the *session* is finished.

## Overview

The main data type in SPEDE is the *object*. Objects can be thought of as black boxes that communicate with each other by passing messages. No global variables are shared between objects. During a session, objects can be run in a heterogeneous environment, as long as each object was pre-compiled on each type of machine. To UNIX, an object is a fully executable program that runs as a normal process. Programmers connect their objects together through well-defined callback routines that interface to a SPEDE library. Objects can pass simple messages, or they can pass messages that wait for a return message, either synchronously or asynchronously. Return messages are the result of a remote *invocation*. Messages are delivered in the order they are sent, which is guaranteed by the underlying data transport mechanism (UNIX stream sockets).

What SPEDE is not is an *interactive* environment. It only runs when a user's application is running. This avoids leaving processes associated with SPEDE running in the background, consuming resources on other users' workstations. SPEDE is a user level environment and should have as little impact on other people's machines as possible.

## Related Work

There are several varieties of distributed systems, ranging from process-oriented distribution [3], to object-oriented distribution with a custom operating system [1], [2]. Most systems are somewhere between the two. These "mixed systems" offer finer-grained distribution than a user's *process*, but they do not usually work with the kernel itself. Instead, they sit on top of the kernel and use a run time model other than a UNIX *process*, such as an *object*. Systems of this type include Mentat[4], Linda[5], and SPEDE. Remote procedure calls [6] can also be considered in this category, although they are only part of an integrated environment (for example, they do not support remote process execution).

Linda [5] is a unique distributed environment because it uses a much different paradigm than the *object* model. Instead of one machine telling another what to do, remote machines that are idle join a pool of free processors. The system can then assign a machine any work that needs to be done. Work is defined as a *tuple*, composed of a name and data. These tuples are managed in a *tuple space*. Machines can take tuples out and put tuples in. The idea is that a tuple is taken out to be processed, and put back in when a result is found. This is clearly different from SPEDE, which is based on objects sending messages to other objects. While the tuple-space model is powerful because of its simplicity and abstractness, performance may be limited due to the complexities of managing the tuple space.

Mentat [4], on the other hand, uses an object-oriented paradigm based on the C++ programming language. Language extensions to C++ allow the system to handle remote communication and synchronization. For instance, after a remote invocation, the caller does not block until the return value from the invocation is needed for some computation. Mentat manages this by creating a dependency graph at compile time that is used at run time to determine when variables are referenced. This is convenient, freeing programmers from worrying about synchronization and message passing. Mentat was the original inspiration for SPEDE because of its *object*-based model. While SPEDE is not as complete as Mentat, it does allows programmers to have more control over message asynchronicity. Also, SPEDE does not require C++, does not require read privileges on the kernel, and does not require a new compiler.

Remote procedure calls, while not in the same class as Linda, Mentat, or SPEDE, allow distributed processing and therefore should be mentioned. Normally, the RPC mechanism allows a function to be executed on a remote machine with the same syntax of a local procedure call. The local machine blocks until the remote function is finished executing. While it is sometimes possible to perform asynchronous RPC calls, the interface is difficult to work with. As a result, many systems use RPC as an underlying mechanism, providing other paradigms, such as *objects*, as the programmer's interface. Mentat in fact uses RPC to accomplish message passing between processors.

SPEDE's method of communication between remote entities is message passing. Message passing is conceptually simple and allows asynchronous activity. One drawback with messages is that the C interface for programmers can be awkward. To help with this, SPEDE supports both synchronous (RPC-like) message passing and asynchronous message passing (by sending a message and specifying a callback function to be used when a return message comes back).

## The SPEDE Architecture

SPEDE has three major components. The first is a name service (**REG**), which keeps track of resources (processors, objects) being used by SPEDE. The second is a remote daemon (**DEAN**) that must run on every participating machine. The third is the set of **objects** which the user creates and runs on clusters of machines. The entire system can be started and stopped relatively quickly, and it is designed to be running only when the user's objects are running. This is different from many systems which must be installed at system boot time and remain in the background for the life of the machine.

The backbone of a SPEDE session is the name service called **REG**. REG is responsible for keeping track of the set of remote machines that are able to participate in a SPEDE session, and the DEANs and objects that are currently running. Ideally, REG would run on a well-known UNIX port number, on a stable machine. This is so that REG would be easy to locate. However, due to the user-level nature of this project, I decided not to require that REG have a fixed port number (which requires a system administrator to install). As a result, objects need another way to find where REG is running. DEANs that are started by REG are given this information as a command-line argument. The difficulty is an object started by the user, at the command line. If the object is on the same file system as the REG, it can read a special file in the user's home directory, written by REG when it starts. Otherwise, the object has to be specifically told the name of the machine and the port where REG can be found.

Machines that participate in a session must run a daemon process called the **DEAN**. The DEAN sits on a UNIX stream socket waiting for objects to connect. Objects that connect can ask the DEAN to create or delete objects on that machine. All messages which go to objects on a machine pass through the DEAN, which allows DEANs to trace all message traffic. In the future, this indirection could be used to migrate objects and to support fault tolerance. DEANs are started automatically by the name service (REG) in response to a request from an object that needs to communicate with remote machine. Also, DEANs terminate after 10 minutes of being idle. This prevents them from lingering in the background after a user has finished running an application.

**Objects** in SPEDE are programs constructed by a programmer and linked into a SPEDE library. The library handles tasks such as inter-object communication and the creation and deletion of objects. The object's program must be compiled on every type of machine that is going to participate in a SPEDE session. Compiled objects are placed in a well-known directory so that they can found by the DEAN, which performs a *fork* and *exec* to run the object. Once an object is started, it connects with the local DEAN to inquire about why it is running (The DEAN is found by asking REG). Objects are started either when other objects create them or when the user runs an object from the command line.

SPEDE relies on UNIX stream sockets for all communication. The decision to use stream sockets rather than datagram sockets was primarily made in light of the time constraints placed on the project. Stream sockets are easy to use and are excellent for transporting large amounts of data from one point to another, but they have problems when used in an environment like SPEDE. The biggest problem is that stream sockets are implemented as file descriptors and every object that connects must have a separate socket. Most machines only allow a process to have 50 or 60 sockets open at a time, which limits the number of objects that can be use. Another drawback with stream sockets is the high overhead cost for sending small amounts of information. Datagram sockets may be better for SPEDE, supporting an unlimited number of "connections" and better handling small messages. Then again, datagram sockets do not guarantee packet delivery, so an error checking mechanism would have to be added.

One interesting implementation aspect of SPEDE is the way DEANs and objects decipher messages. Every message has a sender reference number and a receiver reference number. This reference number is used to distinguish the objects that are communicating, much like a unique ID number. However, these reference numbers are actually pointers into the DEAN's memory where an object structure is stored. When a message is received, the reference number of the receiver is used as a pointer to directly find the information for the object being addressed. If the reference number is not a valid pointer, then a segmentation fault occurs when the false address is referenced. The alternatives, a hash table or list to map valid ID numbers into object information, is more secure but incurs overhead to perform the look-up. Given that SPEDE can only handle 50-60 objects at most (due to the limits of stream sockets), a simple indexed array would have sufficed. However, SPEDE was originally designed to handle many more objects than the UNIX limitation imposes..

## Results

To test the system, I wrote a program which generates Mandelbrot images. I then recorded average run times for a sequential version and a parallel version of the program. The parallel version has a single master process and P worker processes running on other machines. The parallel version was run using the same hardware type (DECstation 5000) at a time of day when the machines were under light load. Nevertheless, all data that is presented is an average time and is under the influence of slight variations in the load of machines and the congestion of the network.

Shown in Figure 1 is a graph of execution times for a Mandelbrot image generator. The image generator was implemented both as a stand-alone sequential version, and a parallel SPEDE version. Actually, the SPEDE version was not completely parallel because the collector object that displayed the final image segments, collected from computational objects, was sequential. The lines on the graph show the run times of the two versions of the program with the time for the parallel version shown with and without startup time included. The horizontal line represents the sequential execution time, where the image was generated without using the SPEDE system. The "Startup Included" line shows the parallel execution time including all start-up costs. The "Steady State" line shows the parallel execution time if start-up costs were ignored.

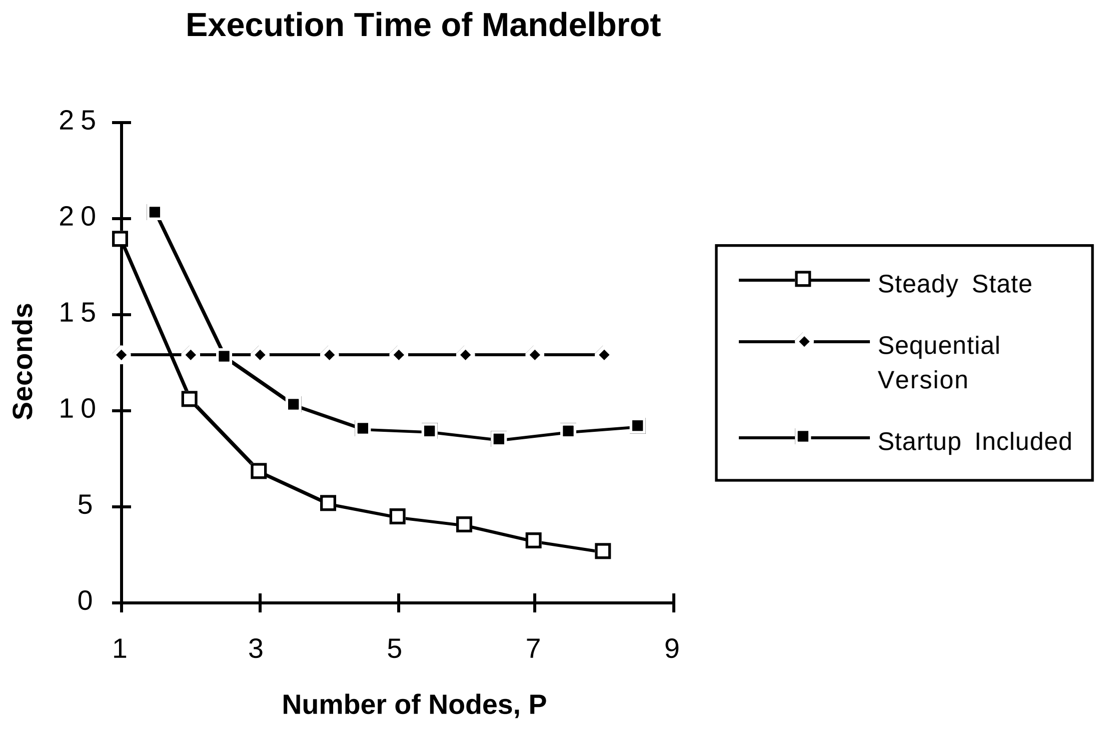

Figure 1

There are a couple of things that are apparent from Figure 1. The first is that execution times decrease with P, if the initial start-up costs are ignored (i.e., timing the system in the steady state). On the other hand, if start-up costs are included the run time bottoms out at 6 nodes and begins to increase, although it is better than sequential. This is clearly a problem with the SPEDE system that could be improved by creating objects in parallel instead of sequentially, as in the current implementation.

Figure 2 shows the speedup achieved using the same data as in Figure 1. Linear (perfect) speedup is represented by the straight line, while the other two lines show speedup of the SPEDE version with and without startup costs. Once again, it is clear that the start-up costs are expensive, since speedup starts to decrease after 6 nodes. On a positive note, speedup in the steady state is reasonable.

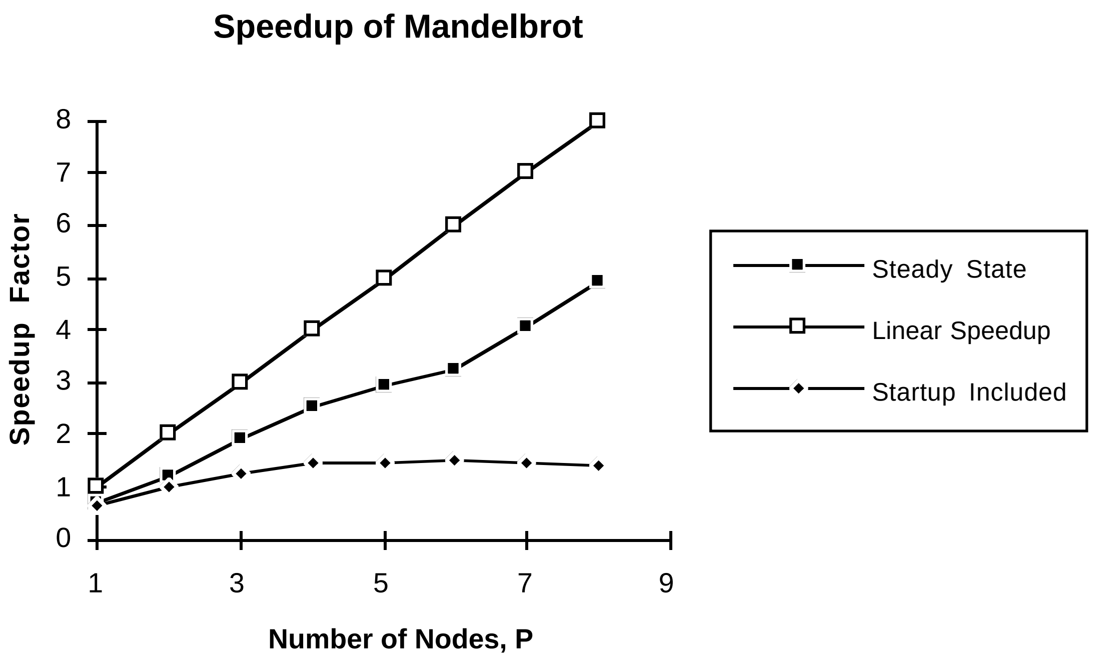

Figure 2

## Future Work

There were two possibilities that did not get implemented. One is fault tolerance and the other is process migration, which, incidentally, go hand in hand. Although these were not included in SPEDE, the structure of SPEDE was set up with both in mind. SPEDE contains a facility for an object to link to any other object and send it messages to be recorded. The messages could be stored and used to restart an object at a later time. The DEANs would also play a role by forwarding all incoming messages to be recorded. This would make the object oblivious to whether it was being check-pointed. The other necessity for fault tolerance to work is that the object must be able to save and restore its state. Since SPEDE does not have its own compiler, this burden would have to be placed on the programmer of the object.

An area for improvement in the current version of SPEDE is the time for remote object creation. From my results, I found that the start-up costs for a session were expensive. Security is another area that needs work. Currently, no user authentication is performed by DEANs or REG. They simply accept remote commands and process them. This hole would allow a mal-intentioned user to connect to another user's DEAN and starting running their objects.

Finally, if there were more time, I would have wrote several more test programs. The Mandelbrot demo tests the system using a simple master/slave object model, however objects can communicate among themselves. One program that would have been usefull in testing this is an n-body simulator.

## Conclusion

Although SPEDE does not have the bells and whistles found in many distributed environments, it can be used to attain significant speedup. An increase in speed by a factor of five (on eight processors) was gained with the Mandelbrot demo while running in a steady state. One of the interesting aspects of this project was to see how much of a system could be built in a such short time period. After only 8 weeks of programming, the bulk of SPEDE was completed. Although there are things to work out in SPEDE, such as reducing start-up costs, the system has proven itself to be worth while. It is clear that distributed applications can work, especially for programmers who do not have access to parallel machines. Distributed computing is a low cost (or no cost) solution to high performance computing and should be considered by anyone needing more CPU power.

## REFERENCES

1. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. ACM Transactions on Computer Systems, 6(1):109-133, February 1988.

2. S. Mullender, G. Rossum, A. Tanenbaum, R. Renesse, and H. van Staveren. Amoeba: A Distributed Operating System for the 1990s. IEEE Computer, 23(5):44-53, May 1990.

3. H. Clark and B. McMillin. DAWGS: A Distributed Compute Server Utilizing Idle Workstations. Journal of Parallel and Distributed Computing, 14:175-186, February 1992.

4. A. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. UVA Computer Science Report No. TR-91-07. April 4, 1991.

5. M. Arango, D. Berndt, N. Carriero, D. Gelernter, and D. Gilmore. Adventures with Network Linda. Supercomputing Review, October 1990.

6. A. Birrell and B. Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems. 2(1):39-59. February 1984.