ACL2 for Parallel Systems Software: A Progress Report[*]

Ewing Lusk and William McCune

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

October 15, 2000

1 Introduction

A significant development in high-performance computing has occurred in recent years with the proliferation of "Beowulf" clusters [6]. Beowulf clusters are parallel computers assembled from commodity-priced personal computers and networks. The explosive growth of the personal computer marketplace, together with rapid technological advances in the hardware sold there, has driven the price/performance ratio so low for Beowulf clusters that they have become competitive with traditional tightly integrated (and thus expensive) supercomputers.

The current bottleneck is systems software. Small clusters are already working well in a large variety of situations, as well as large systems devoted to single applications. To truly compete with highly parallel supercomputer systems, however, Beowulfs need scalable parallel libraries, system management tools, job schedulers, process managers, and other systems software to support a mix of users and applications on systems consisting of hundreds of network-connected computers. Traditional systems software either is not scalable or is tied to specific vendor systems.

The Mathematics and Computer Science Division at Argonne National Laboratory conducts research and prototype software development in parallel systems software. The MPICH implementation [2] of the MPI standard for message passing is already widely used in the Beowulf community. A new project is MPD (for Multi-Purpose Daemon), whose purpose is to explore issues in scalable process management for parallel jobs on Beowulf clusters. Process management includes process startup, job control, management of standard I/O (especially for interactive jobs), security, reliable process

[*]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

shutdown, and provision of some set of facilities for use by an application-level parallel library (such as MPICH) to dynamically establish network connections among processes. The MPD system as a whole consists of dynamically changing network of processes. A diagram of one possible state of the system is shown in Figure 1. Details may be found in [1].

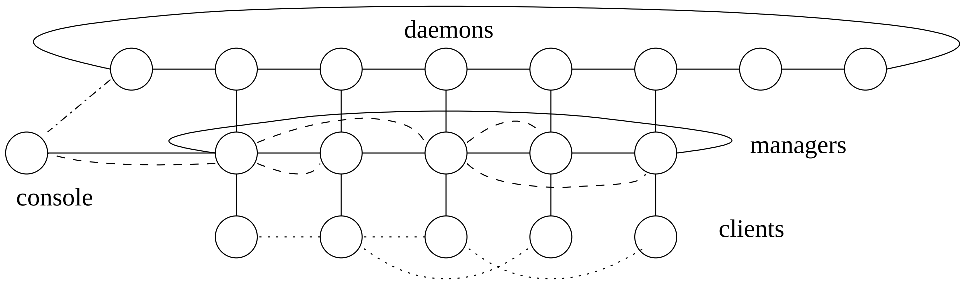

Figure 1: Daemons with console process, managers, and clients

The prototype implementation of MPD in C has not been easy. Informal reasoning about dynamically changing parallel systems is difficult. Compounding the problem is the fact that the software layer on which MPD is built is a low-level-one: it is composed of C programs that make Unix system calls to manage processes (`fork`, `exec` and its variants, `kill`, `rsh`, etc.) to establish TCP connections (`socket`, `bind`, `listen`, `accept`, `connect`, etc.), and to communicate (`read` and `write` on sockets) [4].

We decided to bring higher-level tools to bear on the problem and to employ ACL2 [3] in doing so. This paper describes our experiences so far.

2 Goals of the Project

Our abstract goal was to explore the use of high-level tools in the development of complex software. What attracted us to this particular project was the fact that the system we wished to apply the tools to, the MPD process manager, was

1. complex enough that ACL2-based technology might really hasten its development and improve its robustness,

2. important enough in the context of Beowulf system software to be worth an investment in tools, and yet

3. simple enough that we had hope of concrete success. The various MPD processes themselves have relatively simple structures, in which handlers are attached to various sockets and invoked when messages arrive to process the messages. Although there are many types of messages and thus many handlers, each one is relatively straightforward.

The core of the project is a simulator. We hoped that the exercise of defining the relevant states of (Unix) processes and the state of a system of such processes connected by communication channels with the characteristics of Unix sockets would lead us to a simple data structure that would abstract the significant aspects of the system, enabling us to effectively reason about it, both informally and formally. The `man` pages of the various Unix system calls would be abstracted into the simple semantics of functions that update this data structure.

Our concrete goal was to improve the MPD system as its development continued, by testing it in a way orthogonal to the traditional testing procedures. We expect to modify its code to conform closely to the version expressed in the language used by the simulator to express the individual programs that the MPD processes execute.

In the long run we hope to use this phase of the project as a step toward proving theorems about MPD and similar systems, thus establishing a new level of reliability for parallel systems software.

3 The Multiprocess Model

We are modeling a collection of Unix-style processes communicating via TCP using Unix system calls. That is, our model is a slight, rather than an extreme, abstraction of the C implementation.

3.1 The State of a Multiprocess Computation

A *multiprocess-state* (or *m-state*) is a 4-tuple:

(*process-states connection-states listening-states program-list*).

Each *process-state* is a 5-tuple:

(*process-id program-name program-counter runtime-stack memory*).

The programs that update process states can contain system calls that update other parts of the multiprocess state, for example, creating new connections, sending and receiving messages, and creating new process states.

Each *connection-state* is a 4-tuple:

(*source destination transit-queue inbox-queue*).

A connection-state represents a one-way communication channel between a pair of processes. The source and destination are pairs (*process-id file-descriptor*), because there can be any number of connections between a pair of processes (as in Unix). The transit queue represents messages en route, and the inbox queue represents messages that have been delivered but not yet received by the destination process.

A *listening-state* is a 4-tuple:

(*process-id file-descriptor port-number request-queue*).

Each listening-state represents a process listening for new connections. As in Unix, the file descriptor is known only to the local process, and the port number is known globally. The request queue is a list of (*process-id file-descriptor*) pairs representing processes that are asking for connections.

The *program-list* is simply an alist that maps program names to program code. This is used when starting new processes.

3.2 The System Calls

The programming language has a set of system calls for setting up communication channels with other processes, sending and receiving messages, and creating new processes. The system calls reflect similar Unix system calls, in particular, the use of ports and file descriptors. We have simplified things, however, by omitting error handling.

file-descriptor = **setup-listener(***port***).** This takes the place of the Unix **socket** and **bind** system calls. It modifies a multiprocess state by creating a new listening state.

file-descriptor = **connect(***host, port***).** A request to connect to another process inserts an entry into the request queue of a listening state. The process waits until the connection is accepted by the remote process.

file-descriptor = **accept(***file-descriptor***).** Accepting a connection takes a member of the request queue and creates a new connection state. If the request queue is empty, the process waits until a request arrives.

file-descriptor-list = **select().** Select returns the list of file descriptors that have messages ready to be received, in particular, nonempty inbox queues in connection states and nonempty request queues in listening states.

send(*file-descriptor, message***).** Send inserts a message into the transit queue of a connection state.

message = **receive(***file-descriptor***).** Receive takes a message from the inbox queue of a connection state. If the inbox queue is empty, the process waits until a message has been delivered.

return-code = **fork().** Fork creates a copy of the current process, returning a flag telling whether the process is the parent or the child.

exec(*program, arguments***).** Exec replaces the current process with a new process.

rsh(*host, program, arguments***).** Rsh creates a new process on a given host.

3.3 The Multiprocess Simulator

The individual processes are simulated as ordinary state machines, with an ACL2 function that steps the process by executing one instruction of the program, updating the process state. When a simulator executes a system call, the multiprocess state can be updated as well.

The multiprocess is simulated by executing two types of step: (1) stepping an individual process, and (2) stepping a connection state. Stepping a connection state is simply transferring a message from the transit queue to the inbox queue, that is, delivering a message. An oracle tells the simulator which kind of step to perform and which process to step or which connection state to step.

A deficiency of our multiprocess model is that the `connect`, `accept`, and `rsh` commands are executed is if they had instantaneous effects on other processes. The model could be improved by adding another type of communication channel for system operations; that would allow delays before the operations are carried out, analogous to the transit and inbox queues of connection states.

3.4 Status of the ACL2 Code

We have constructed a prototype multiprocess simulator in ACL2 (see the file **README** of the associated directory for pointers to the books) and run it on two simple parallel programs. The first contains console and daemon programs (see Figure 1) that cause a message to be sent from the console to a daemon, around the ring and back to the console. The second is a set of manager and client programs by which the managers implement a barrier operation for the clients: all clients must call the barrier before any of them can leave it. The general case of such a barrier was difficult to get right in the real MPD system; we wish we had been able to test it first on the simulator, which didn't exist at the time.

The programs are available in the files **trace** and **fence**, respectively, of the associated directory.

4 The Next Steps

Aside from verifying guards, we have not proved anything about the simulator. But we are hopeful.

In [5], J Moore presents a method for proving properties of shared-memory multiprocess programs. The key idea of the method is that a substantial part of the proofs can be done from the points of view of the individual processes. As an individual process steps from state to state, an oracle tells the process how the shared memory is changed by the other processes. The properties of the uniprocessor view are then related to the global multiprocessor view. Perhaps we can use a similar method to prove properties of our (message-passing) multiprocess model by replacing the shared-memory oracle with an oracle that tells how the rest of the multiprocess state changes, in particular, how the connection and listening states change.

Experience with designing and debugging the C version of MPD has shown us many areas where formal verification with ACL2 would be of great value. In addition to the barrier example above, we have had difficulties in ensuring that if any client aborts, the entire job is brought down cleanly. These are cases where even attempting formal proofs would help us get it right and increase our confidence that we had it right.

5 Conclusion

This project is still in its early stages. Nonetheless, we have learned a few things. It is possible to usefully abstract the complex collection of Unix interprocess communication system calls without trivializing the problems inherent in real parallel algorithms. Using ACL2 has a steep but climbable learning curve. Our simulator is slow, but we have hopes of speeding things up by using single-threaded objects.

References

[1] Ralph Butler, Ewing Lusk, and William Gropp. A scalable process-management environment for parallel programs. In Peter Kacsuk, editor, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting, Balatonfured, Hungary*, Lecture Notes in Computer Science. Springer Verlag, 2000. (to appear).

[2] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[3] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods. Kluwer Academic, 2000.

[4] Linux `man` pages.

[5] J Moore. A Mechanically Checked Proof of a Multiprocessor Result via a Uniprocessor View. Tech. report, Department of Computer Sciences, University of Texas, Austin, August 1998.

[6] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *International Conference on Parallel Processing, Vol. 1: Architecture*, pages 11–14, Boca Raton, FL, August 1995. CRC Press.