Conf-9109276--9

WSRC-MS--91-305

DE92 009405

DISTRIBUTED VISUALIZATION

by

T. R. Arnold

Westinghouse Savannah River Company
Savannah River Site
Aiken, SC 29808

A paper proposed for presentation at the
Cray Users' Group Meeting
Santa Fe, NM
September 23, 1991

and for publication in the proceedings

DISCLAIMER

This paper was prepared in connection with work done under Contract No. DE-AC09- 89SR18035 with the U. S. Department of Energy.

DISTRIBUTED VISUALIZATION

by
T. R. Arnold

NRT/SC
Westinghouse Savannah River Company
Aiken, SC 29808

ABSTRACT

Within the last half decade or so, two technological evolutions have culminated in mature products of potentially great utility to computer simulation. One is the emergence of low-cost workstations with versatile graphics and substantial local CPU power. The other is the adoption of UNIX as a de facto "standard" operating system on at least some machines offered by virtually all vendors. It is now possible to perform transient simulations in which the number-crunching capability of a supercomputer is harnessed to allow both process control and graphical visualization on a workstation. Such a distributed computing system is described as it now exists: a large FORTRAN application on a CRAY communicates with the balance of the simulation on a SUN-3 or SUN-4 via remote procedure call (RPC) protocol. The hooks to the application and the graphics have been made very flexible. Piping of output from the CRAY to the SUN is nonselective, allowing the user to summon data and draw or plot at will. The ensemble of control, application, data handling, and graphics modules is loosely coupled, which further generalizes the utility of the software design.

INTRODUCTION

The Problem

The purpose of this paper is to present one way to achieve parallelism and distributed processing using installed technology and currently developed, validated software. The Westinghouse Savannah River Nuclear Plant Analyzer (NPA) is used as an example to present the technology.

The UNIX operating system provides process control and synchronization support at several levels of abstraction. How can we exploit this with existing software?

A basic problem facing engineering and scientific modelers—and the underlying problem that this paper addresses--is how to reconcile the following three conditions:

1. Most modeling software is large, FORTRAN, batch-oriented, and often validated in some form by some customer (e.g., the Department of Energy). It can be ported to other platforms, but rewriting is much more expensive.

2. The current crop of engineering workstations provides good graphics, reliable networking, and reasonable prices.

3. The wide distribution of UNIX, including UNICOS on the CRAY, allows distributed processing with minimal effort.

One aspect of the NPA provides a solution to exactly this problem. We are able to implement distributed multi-processing using a large FORTRAN code on a network of CRAY and SUN workstations, all running some variant of UNIX. The finished product runs between SUNs and CRAYs, but the engineering prototype has been tested on ULTRIX machines (DEC UNIX) and Macintoshes running A/UX. The control and synchronization programs were unchanged.

This paper concentrates on the UNIX features that enable that design and how they can be applied to a wide range of existing software.

A Solution

The fundamental mechanism that is used to exploit parallelism is the UNIX pipe. Extending the pipe across the network and serializing the output of

multiple processes on multiple machines is the fundamental goal realized by the software described here.

The NPA is an example of an engineering simulator. As such, its control and synchronization requirements are more stringent than typical training simulators. In particular, the NPA can be paused at known points in the processing so that changes can be made to the state of the model in such a manner as to be repeatable. These synchronization requirements are far more stringent than those required to simply display the data in parallel with its generation.

The NPA design demonstrates the ease with which coarse-grain parallelism can be achieved using UNIX. The salient feature of the NPA from the perspective of this paper is that it is a distributed architecture. Several programs run in parallel on at least two computers. The programs fall into the following four categories:

- modified "batch" codes that can be run stand-alone
- data transmission programs that transmit and translate binary data formats
- synchronization and control programs that control the simulation
- an iconic "point-and-click" user interface

Overview

Some common terms are introduced to the reader, followed by a discussion of parallelism and distributed processing. The conceptual view of the NPA is then described. This will provide the backdrop against which UNIX synchronization and control will be presented. The functional view of the NPA is also described, which identifies the programs and how the data travels between them. The final discussion covers the process view of the NPA, where the design objectives lead to daemon processes. This is the point where the UNIX features are introduced.

TERMINOLOGY

process

This refers to a program that is executing on a CPU. The same executable file can lead to several processes, each managed by the operating system (e.g., a set of text editor sessions, each from the same executable program file).

pid (process identification)

A unique number—on a given machine—by which the UNIX operating system refers to a process.

server–client

Describes two processes, one of which waits for messages from the other and sends replies based on requests. The two processes can be on the same or different computers.

daemon

This is a process on a computer that basically waits in the background until a client calls it with some request. Most servers are daemons.

IPC (Inter–Process Communication)

When capitalized, this refers to any generic means by which two executing processes communicate with one another, whether on the same machine or not.

ipc

This is the UNIX term for the UNIX kernel support for IPC between two processes on the same machine.

NFS (Network File System)

This was originally developed by SUN and is currently available on almost all UNIX implementations. It provides a distributed file system (i.e., files that are on remote processors appear as though they are local). The NFS is significant to the NPA only because of RPC.

RPC (remote procedure call)

RPC is the network equivalent of ipc, originally developed by SUN and currently the mechanism by which NFS is implemented. RPC provides a subroutine-call metaphor for IPC and data translation facilities.

PARALLEL AND DISTRIBUTED—BACKGROUND

Parallel and distributed are two words that have several connotations in the literature. In this paper

distributed means that the work to be done is performed by more than one process. The processes can be on one or more processors or CPUs. *Parallel* means that more than one of the processors can actually, or apparently, be executing at the same time. For single CPU systems, the "apparently" proviso means that the operating system shares the CPU without the program being any wiser. For processes on distinct CPUs, more than one can be executing concurrently.

Parallelism can be further subdivided. Coarse-grained parallelism means that parallel execution is at the program or process level. The NPA is parallel at this level. Finer levels include subroutine calls that actually execute asynchronously on other machines, as well as more advanced forms where the operating system decides where to execute segments of code.

> NOTE: RPC calls are exactly like normal subroutine calls to an application program. They block until the processing is done. As such, RPC calls are not inherently parallel, but they can be distributed. RPC is used in our work, but only to transfer data. Once the transfer is complete to the daemon process, the caller resumes execution. It does not wait for a (possibly elaborate) calculation by the callee to be completed.

Multi-threaded refers to another kind of apparent parallelism, whereby the process itself has several threads of control that compete with others for the CPU. Multi-threaded processing is not used in the NPA.

In a perfect world with infinite resources, the best way to take advantage of parallelism and distributed architectures would be to rewrite all of the existing applications. Given the size and complexity of the vast majority of large FORTRAN batch codes, this would be a difficult task. Fortunately, this is not necessary to achieve coarse-grained parallelism for many of the current groups of Scientific and Engineering Modeling codes. [One such code is TRAC (Transient Reactor Analysis Code), written in the Los Alamos National Laboratory.]

Most large FORTAN batch programs share the following characteristics:

1. First, the program reads a user input file that describes the model to be executed.
2. The program then allocates the dynamic arrays needed to run the simulation, based on the user's input.
3. The program goes into a big simulation loop, periodically updating time and recalculating the state of the system being modelled. Data may or may not be buffered, but are periodically written to one or more output files (in binary or ASCII text).
4. Eventually the problem ends and the final output data are written.

The primary design goal of the NPA Simulator is to use TRAC as the simulation engine, capturing the output data as it is produced rather than post-processing it. A secondary goal is to provide means to interact with the run-time model internals of TRAC to perform such tasks as changing variables. Specifically, these would be the parameters that govern trip variables (e.g., the limits) and control variables (e.g., gain, bias, and lag). The second feature requires minor changes to TRAC, but the first does not.

The value of the chosen UNIX-based approach is that it is not machine-specific and that it generalizes to a wide range of large FORTRAN high-fidelity codes.

CONCEPTUAL VIEW—THE DESIGN PROBLEM

Figure 1 shows the conceptual view of the NPA in simulator mode. Conceptually, we place a thin wrapper around TRAC on the CRAY and around MOVIE and TRENDS on the SUN. Then we allow the simulation engine (TRAC) and the display programs to operate in parallel.

As Figure 1 shows, the NPA must be able to interchange information between processes on different vendor computers while reusing as much of the batch-oriented programs (TRAC, TRENDS, and MOVIE) as possible. The main design issues are listed below.

Process Control

The NPA Executive must be able to invoke and control the SUN-resident display programs and at least start up the CRAY-resident Protocol program, which has the analogous task on the CRAY with respect to TRAC.

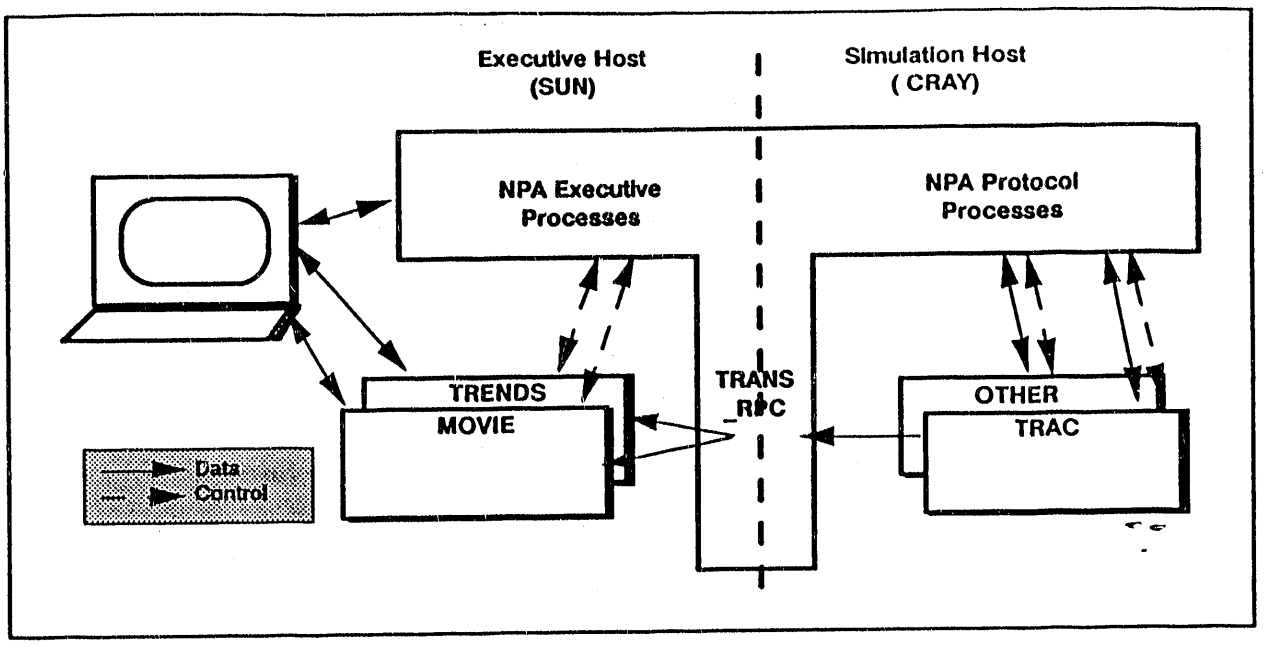

Figure 1. Conceptual View of the NPA Simulator

Synchronization

There are two types of process synchronization involved: (1) data synchronization, where data is processed for display when (as soon as) it is available, and (2) user synchronization, where the user halts the simulation to make changes to the data display or to modify the internal state of the simulation engine.

NOTE: Halting the simulation to modify the parameters that drive the application is a concession to engineering simulation. This is in contrast to some training simulators, which allow input to be randomly asychronous.

Inter-Machine Communications

All of the NPA inter-machine communications are via RPC. The TRANS_RPC component in Figure 1 basically extends the UNIX notion of a pipe across heterogeneous hardware boundaries in an extremely portable way.

There are lower levels of abstraction at which inter-machine communications could be achieved (e.g., sockets), but RPC isolates the application from such hardware dependencies as byte order, etc.

The output processing components of the NPA are TRENDS and MOVIE. TRENDS allows the user to display trend-line plots of any variables in the output of TRAC, while MOVIE provides a similar color-driven "movie" of structured abstract representations of the modeled components of the simulation. In these images, the range of color represents the variation of a selected parameter in each portion of the image displayed.

Two features of this design approach are in Figure 1.

- Programs other than TRAC can be plugged in if their output can be translated to TRAC format.
- Display programs other than MOVIE and TRENDS can be plugged in if they can read translated TRAC output.

FUNCTIONAL VIEW--SOURCES AND DESTINATIONS FOR DATA AND CONTROL

Figure 2 translates the design requirements of Figure 1 into the first-cut processes that must exist for the NPA to execute.

Each of the circled programs has synchronization and control requirements.

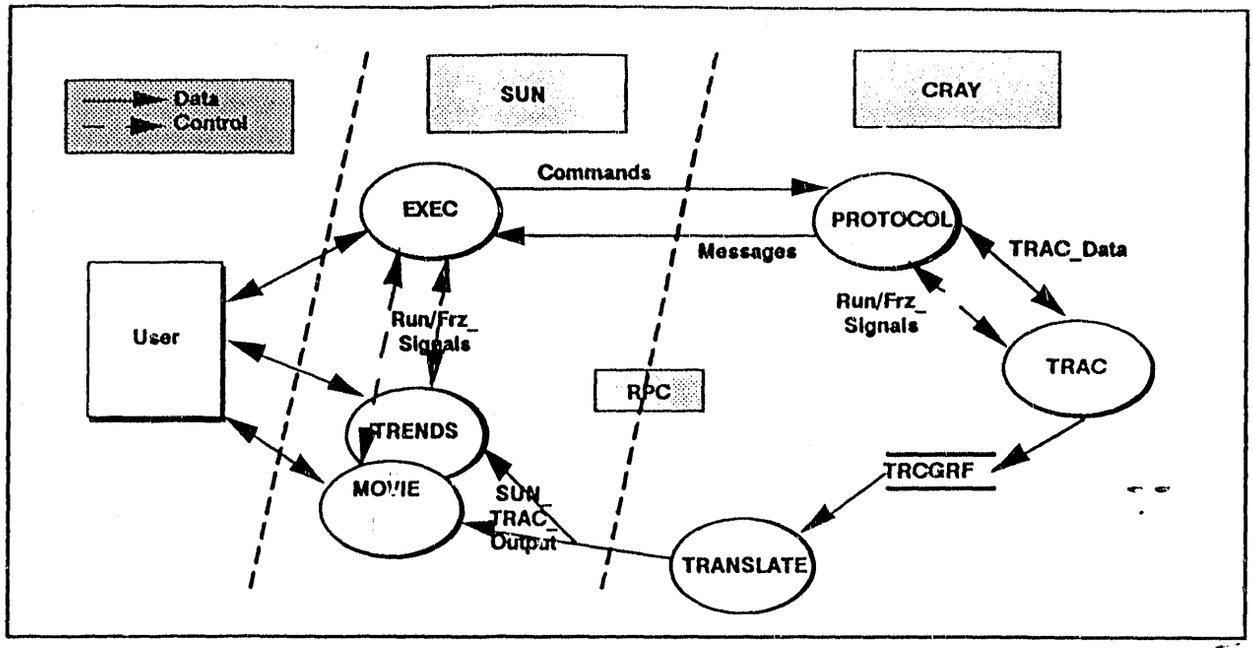

Figure 2. Functional Process View of the NPA Simulator

EXEC

When the user enters simulation commands, send them to PROTOCOL on the CRAY. *When* the user wants to pause the simulation, pause the CRAY-side processes and the SUN-side processes.

TRENDS, MOVIE

When the simulation has been paused, display the window buttons and panels that allow the user to modify the information displayed. *When* the simulation is no longer paused, remove the window buttons and panels. *When* translated data arrives from the simulation engineer, process it.

TRANSLATE

When TRAC data is available, read it, translate it, and send it to the display programs on the SUN.

PROTOCOL

When commands are available, read them and act on them. When errors occur or responses to commands are required, send them to the EXEC.

TRAC

Write data to the TRCGRF pipe. *When* pause commands arrive—at the end of the simulation loop—pause. *When* run commands arrive, resume execution.

As can be seen in Figure 2, simulation data basically flow from the TRAC program, through the TRANSLATE program, to the display programs. Commands and messages flow back and forth between EXEC and PROTOCOL. Data are displayed at the user's terminal. Each of the statements that begin "*When* . . . " reflects a synchronization requirement. As will be seen, much of this is provided by the UNIX operating system without explicit user-written software.

PROCESS VIEW OF THE NPA—HOW CONTROL AND SYNCHRONIZATION IS IMPLEMENTED

Figure 3 contains the processes that exist at run-time for the NPA, together with the means by which they interact.

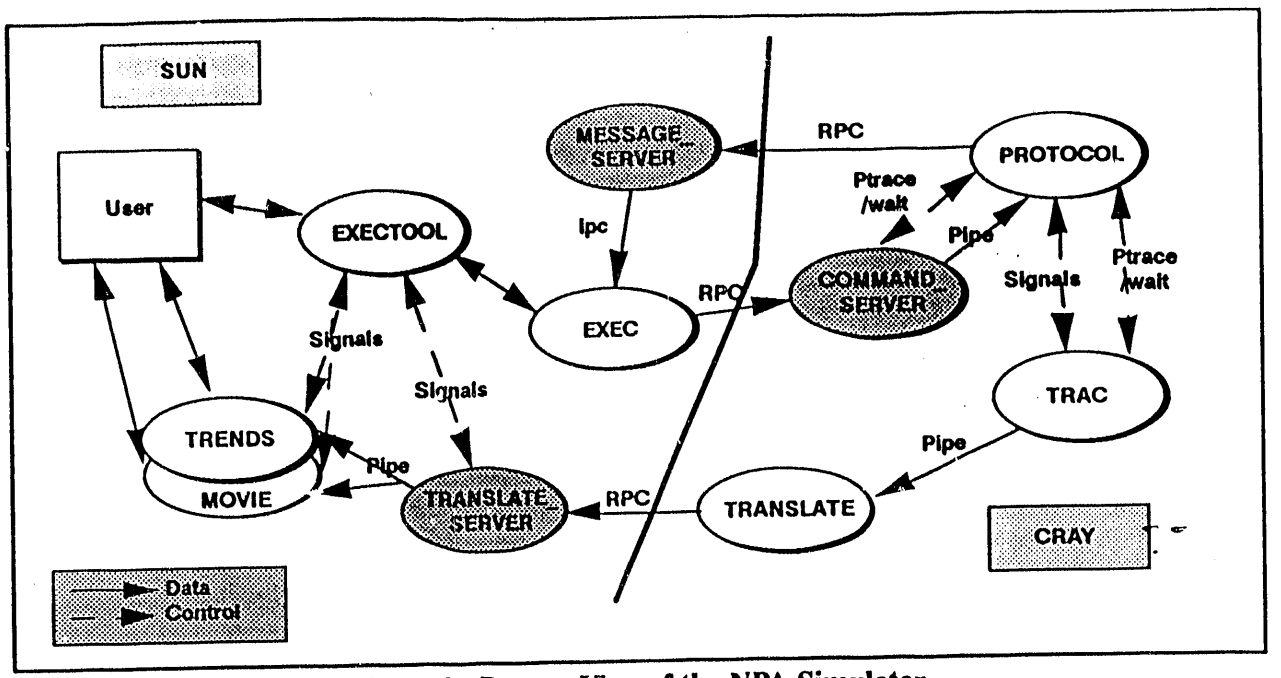

Figure 3. Process View of the NPA Simulator

EXECTOOL is a window-oriented interface to the command-line oriented EXEC. The three shaded processes are RPC daemon processes. They are started in the background and wait until an RPC message arrives that is addressed to them. When it does so, user code is executed, based on the message. UNIX daemons are one way to handle asynchronicity in a multi-machine environment. As can be seen in Figure 3, they serve as interfaces to all inter-machine transactions.

Several UNIX mechanisms are visible in Figure 3. Each will be briefly introduced and the reason for their use in the design will be addressed.

Process Control

The UNIX process model is simple. After the system comes up, the only way to invoke another process is via the *fork* and *exec* system calls. The relationship is hierarchical (i.e., a parent invokes children, who in turn invoke their own children). Most of the parent's environment is inherited across the *fork*; in particular, open file descriptors. *Fork* creates a virtual duplicate process, whereas *exec* replaces the process with another executable one; thus, the normal UNIX sequence to startup process is to *fork*, then have the child branch of the *fork* replace itself via *exec*. The system identifies each process on a given CPU by its pid.

In Figure 3, the EXECTOOL process is the parent of the other processes on the SUN, while PROTOCOL is the parent of the others on the CRAY.

fork

This system call creates a virtual duplicate of the executing process. File descriptors are inherited.

exec

This system call replaces the currently executing program with another, but inherits the open file descriptors.

ptrace

This system call allows the parent process to control the child process. The most important features are the ability to run and freeze the process and to read and write into the child's memory. All reference to the child is via its pid. This is the system call that interactive debuggers use. A side effect of this call is that the parent intercepts all signals destined for the child.

wait

This system call puts the invoking process into a "hard wait" until one of its children either ends or, in the case of the ptraced children, enters "sleep mode". When the wait returns, the return value contains the reason and the pid of the process that provoked the wakeup.

Stevens' book, *UNIX Network Programming*, recommends that users stay away from the *ptrace* system call; however, coupled with the *wait* call, it is exactly the call to use in our design. It allows access to memory of a process for which the user may not necessarily have the source code, as well as allowing the process to wait until any of its children wake it up. Unlike other wakeup mechanisms, the pid of the process that woke up the controller is available as part of the wait-return information.

Synchronization

The most important of the synchronization issues revolves around the readers and writers of data that only read and write when there is data available. This is handled automatically by UNIX when the intermediary through which two processes communicate is a named or unnamed pipe. By replacing the normal output/input filename with a filename that is a pipe, the system does most of the work. In Figure 3, the outputs of both TRAC and the TRANSLATE daemon are through named pipes. The input from the command server to PROTOCOL is through an unnamed pipe.

NOTE: The only problems encountered in applying this approach have been the following:

- FORTRAN records larger than the capacity of the pipe. The system will cope with this provided the reader is prepared to read less than the full amount expected.

- FORTRAN open filenames that are specified as NEW rather than UNKNOWN cause the named pipe to be deleted. To work around this problem is somewhat language dependent. The assign statement under UNICOS is a good place to start.

The second type of synchronization that is evident in Figure 3 revolves around the signals interchanged. Signals are actually sent by the *kill* (2) system call, and they are received by the handler defined in the *signal* (2) system call. The only information conveyed to the receiving process is the signal number. The number of signals varies somewhat from UNIX implementation to implementation, but certain low-level numbers are fixed. A programmer can count on signals USR1 and USR2 being unused by the system.

Signals are quick, low-level, very information-poor ways for processes to synchronize, but they are adequate if some care is taken by the programmer. However, signals cause read and write system calls to be terminated prematurely if they are waiting in the kernel. This can be provided for in the reader of the system call by explicitly checking why the read or write ended in error; however, for language-library routines, this is out of the programmer's hands.

SUMMARY

A great deal of hard work has gone into many old programming packages that perform detailed calculations, yet output their data in text or binary form. This paper has dealt with one means to visualize that output in parallel with the data production while avoiding rewriting the large application.

The mechanism employed in the Westinghouse Savannah River NPA consists of the following features:

- A large FORTRAN batch program is run on the CRAY under UNICOS.

- The output file(s) of the large code are replaced with UNIX pipe files.

- Reader(s)—separate programs—are attached to the output side of the pipe(s), which transmit the data via RPC to applications on engineering workstations that are running UNIX and RPC.

- The workstation applications take advantage of the graphics capacities and software to reformat the data into images in parallel with the data production on the CRAY.

If there are no further requirements on the system, synchronization is free in that the UNIX processes on each side of the pipe(s) and RPC connections can pause the applicable processes until data are available to be read or written.

Note that a further feature to the approach of distribution of processing and graphical display is that the RPC interface, which can be over a TCP/IP link, can connect graphics to programs running on geographically, widely separated systems.

BIBLIOGRAPHY

1. Wyatt, P. W., T. R. Arnold, K. E. Hammer, J. S. Perry, and G. A. McKaskle. "A Distributed UNIX-Based Simulator". *ANSI-ENS International Topical Meeting on Advances in Mathematics, Computations, and Reactor Physics*, Pittsburgh, PA, April 1991.

2. Bach, J. *The Design of the UNIX Operating System*. Prentice-Hall Software Series, 1986.

3. Stevens, W. *UNIX Network Programming*. Prentice-Hall Software Series, 1990.

4. *Remote Procedure Call (RPC) Reference Manual, SR-1089*. CRAY Research, Inc.

5. *Network Programming*. Sun Microsystems.

END

DATE FILMED

4/15/92