# SPECTRUM: A PARALLEL SIMULATION TESTBED

Paul F. Reynolds, Jr.
Phillip M. Dickens

IPC-TR-89-012
MARCH 12, 1989

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

This research was supported in part by Jet Propulsion Laboratory Contract #957721.

# SPECTRUM: A Parallel Simulation Testbed†

Paul F. Reynolds, Jr
and
Phillip M. Dickens
Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

**ABSTRACT**

SPECTRUM is a testbed for designing and evaluating parallel simulation protocols. The concept is based on the work reported in [Reyn88] in which it was shown that a broad range of possibilities exists for designing parallel simulation protocols. SPECTRUM is the first known testbed in which experimentation on a full range of protocols is supported in a *common* environment. We introduce the use of *filters* as a means for efficiently specifying protocols. We have implemented prototype versions of our testbed on an INTEL iPSC/2 and a BBN GP-1000.

**INTRODUCTION**

The SPECTRUM testbed is meant to support experimentation with simulation protocol design variables. Our goal is to facilitate experimentation so that a designer may focus on protocols and performance rather than implementation details. We give a brief overview of the state of research into parallel simulation protocols and then discuss the SPECTRUM testbed.

*Parallel simulation*, generally called *distributed simulation* in the literature, is concerned with the parallel execution of discrete event simulations. Beginning with the research of Chandy and Misra [ChMi79] and Peacock et al. [PeWo79], a number of approaches have been described for coordinating cooperating processes so that the outcome of a parallel simulation is the same as would occur in a more conventional sequential simulation. Protocols have been called *conservative* if they satisfy the property that no process receives information from any other process that predates the current simulation time of the receiving process. They have been called *optimistic* if processes can act on incomplete information, thus admitting the case where messages may arrive "in the past." Optimistic methods have typically required that some sort of rollback mechanism exist to allow for repair of incorrectly sequenced events. A survey of parallel simulation appeared in [Misr86].

Examples of additional approaches generally considered conservative include the blocking table protocol [PeMa80], deadlock detection [ChMi81], SRADS [Reyn82], appointments [NiRe84], feed-forward [Kuma86], conditional knowledge [ChMi87] and bounded lag [Luba87]. The optimistic approach has its foundation in the time warp method [JeSo82]. Others have explored variations on the optimistic approach including [Jeff85] and [Soko88]. SRADS and moving time window [Soko88] have features that bridge these characterizations.

Parallel simulation was originally named *distributed simulation* in the early Chandy and Misra paper [ChMi79]. More recently distributed simulation has come to be associated with geographically distributed simulations, for example the National Testbed [Word88] and BBN's SIMNET [PoMi87]. Without attempting to establish formal definitions here, we distinguish distributed simulations from parallel simulations on the basis of inter-process communication times and goals for employing multiple processors. Distributed simulations tend to incur communication delays on the order of seconds, and they tend to be employed for the purpose of bringing physically separated resources together for simulation purposes. Distributed simulations may not be concerned with processor utilization or minimum finishing time, although real-time requirements may bring these into play. Parallel simulation, on the other hand, has been studied primarily for the purpose of maximizing processor utilization and/or minimizing simulation finishing time. Inter-processor communication times have generally been presumed to be relatively small (e.g. on the order of milliseconds). Parallel simulation research has not been concerned with real-time issues, human-in-the-loop or hardware-in-the-loop.

In [Reyn88] we showed that, contrary to prevailing beliefs, there is a spectrum of possible parallel simulation protocols. The labels "conservative" and "optimistic" are simply inadequate. At the heart of our research was the establishment of a set of design variables for parallel simulation protocols. We list these design variables and refer the interested reader to [Reyn88].

**DV.0:** **Partitioning** - Determine clusters of logical processes (LP's) based on distinct sets of design variable bindings.

†This research was supported in part by JPL Contract #957721.

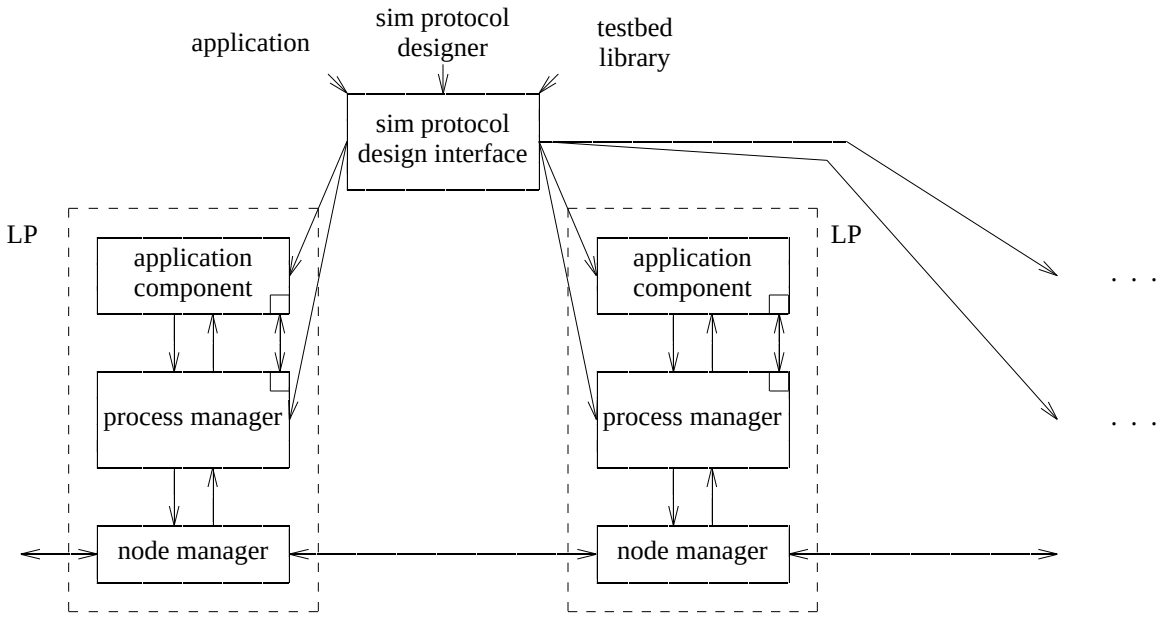


**Figure 1. Block Diagram of SPECTRUM Testbed.**

**DV.1:** **Adaptability** - Changing design variable bindings based on knowledge of selected aspects of the simulation state.

**DV.2:** **Aggressiveness** - Processing messages based on conditional knowledge; that is, relaxing the requirement that messages be processed in a strict monotonic order with respect to message times.

**DV.3:** **Accuracy** - Requiring that events within LP's *ultimately* be processed in a correct (monotonic) sequence.

**DV.4:** **Risk** - Passing messages which have been processed based on aggressive or inaccurate processing assumptions in an LP.

**DV.5:** **Knowledge embedding** - Knowledge about LPs' behavioral attributes is embedded in the simulation.

**DV.6:** **Knowledge dissemination** - LP's initiate the transmission of knowledge to other LP's.

**DV.7:** **Knowledge acquisition** - LP's initiate requests for knowledge from other LP's.

**DV.8:** **Synchrony** - Degree of temporal binding among LP's.

We note that LP's can send two types of messages. *Event messages* carry information that can be related directly to events in the physical system being simulated. *Non-event messages* are all other messages. Messages contain *time-stamps* representing the logical time at which the message was sent. While message passing is a presumed part of our logical model, and fits neatly with the hypercube architecture, message passing may not be required in actual implementation. The SPECTRUM testbed, as implemented on the iPSC/2, does use message passing. However, the ease with which SPECTRUM can be ported to a shared memory environment allows an easy comparison of these different architectures.

The SPECTRUM testbed has been designed to facilitate experimentation with parallel simulation protocols based on bindings for the design variables described above. We discuss it next.

## THE SPECTRUM TESTBED

A protocol designer approaches the testbed with an *application*, possibly brought from the outside. A testbed *simulation protocol design interface* provides libraries of applications and support routines for protocol design experimentation. The designer, with this support, constructs an experiment using the simulation protocol design interface. This leads to the generation of *application components* - pieces of the original application which can be executed concurrently - as well as a customized *process manager* for each application component. A process manager supplies many of the routines common to all simulations such as functions for

managing time and event queues. An application component, a process manager and a node manager constitute a logical process (LP). Node managers are provided by the testbed to support communication among LP's. Figure 1 shows the relationships among these components.

Our view is there are two major activities: applications design and parallel simulation protocol design. Both activities may be done by one experimenter, but they need not be. We provide a clear and simple interface between those activities an applications designer would do, and those a protocol designer would do. This interface is supported by a designer defined global data area in each application component and well defined operations on that data area --many provided by the testbed.

It is the responsibility of either the applications designer or the protocol designer to partition the simulation into application components. There need not be any relationship between the number of application components created and the number of physical processors available to execute them.

The functional behavior of a simulation is provided at the application level. For example, in a logic network simulation, the application level includes the logic for gate level simulation, signal path delays, etc. However, in the testbed, simulation-level functions such as time maintenance, event maintenance and message handling are provided by the process manager.

Within an LP it is assumed that an application component calls upon the process manager for time and event maintenance. Among other things, the application component has access to the following operations in the process manager:

- initialization - each process manager initializes a local clock and an event queue.
- post-event - post an event for future processing. A future simulation time and the event to be simulated must be provided.
- get-next-event - get the next event for processing, given that the current simulation time is $ tau $.
- time-advance functions - advance the current simulation time to time $tau $.

As we shall see, this provides opportunities for the protocol designer to affect event ordering, non-event support, etc., for the application component through the process manager. It also provides a useful form of separation between an application component and the activities associated with customizing parallel simulation protocols. In those cases where simulation-level data objects must be accessed it is assumed that they are made available through an externally referenceable data area.

Simulation protocol design and the overall piecing together of the parallel simulation is done through the *simulation protocol design interface*. We assume the designer has available to him/her:

- The application to be simulated in code form.
- An *understanding* of how that code works, if necessary (this is required for some simulation protocols).
- A library of testbed-provided simulation protocols (A standard set is provided).
- A library of standard functions and procedures for building simulation protocols (for those who wish to build their own or customize existing simulation protocols).
- A library of simulation protocol designer-supplied functions and procedures.
- A set of symbolic names for the logical processes (as represented by the application components) representing the parallel simulation.

The simulation designer can implement a simulation protocol by associating *filters* with the operations provided by the process manager (initialize, post-event, get-next-event, time-advance). For example, with a specification such as:

*On post-event apply(* $P sub 1$, $P sub 2$);

the designer can indicate a desire to have procedures $P sub 1$ and $P sub 2$ called each time an application component calls post-event in the process manager. This can be customized for each LP. If the simulation protocol designer wishes to use a simulation protocol residing in the testbed library then a set of filters and procedures will be supplied automatically. Some customization may be required depending on how much knowledge of the application is required for the simulation protocol. In the same vein a designer can mix library-provided filters and procedures with customized functions and procedures. Even in the extreme case where an entirely new protocol is to be tested, many library routines will be useful.

The primary output of the simulation protocol design process is a specification, for each logical process in the simulation, of the filters and related procedures and functions that will be invoked. We provide a standard set of actions in each process manager, each of which can have a filter associated with it. Thus, in any or all process managers a filter on the post-event routine can be specified. The particular function applied as a result of the filtering process can be dependent on the logical process.

The *process manager* (one per logical process) is a passive entity, providing time and events maintenance to the application level and applying protocol designer-specified filters on a subset of a standard set of operations in the process manager. From the application point of view the process manager manages the local (logical process) simulation clock and the events list. From the protocol design point of view, the process manager implements the functions and procedures necessary to effect the parallel simulation protocol.

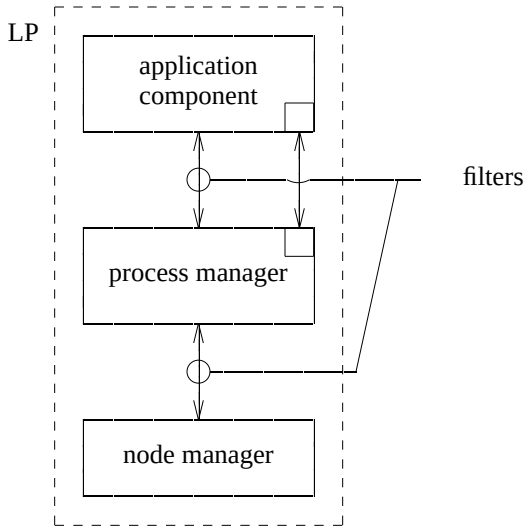


**Figure 2. Applying Filters**

The *node manager* provides low level message handling among logical processes and can perform scheduling functions in the event multiple application components occupy a single processor. There is one node manager per processor. Each node manger can have one or more process managers associated with it, and each of them, in turn, will have one associated logical process.

## FILTERS

In the testbed we rely on filters exclusively to implement parallel simulation protocols. Filters are of the general form:

*On <action> apply($P sub 1$, $P sub 2$, ..., $P sub n$);*

where <action> is one of the five calls: initialize, get-next-event, post-event, advance-time or post-message. The first four of these occur between the application layer and the process manager. The post-message action occurs between the node manager and the process manager. As a result, all filters can be placed between the application component and the process manager or between the node manager and the process manager within an LP, as depicted in figure 2.

As noted above, the $P sub i$ can be arbitrary procedures. Information for filter procedures to act on comes from the following sources: 1) contents of the action requests and responses that occur between the application component and the process manager, 2) data placed in the common data areas of the process manager and the application layer (as depicted by the small boxes in the application component and the process manager in figure 2), and 3) messages that pass between the node manager and the process manager.

Filters, as defined, provide the modularity and organizational power necessary to make a testbed useful. Our experience to date has shown that filter implementation of protocols can be done in a straight-forward manner. In the following section we demonstrate how filters can be used to implement specific protocols.

## FILTERS FOR SPECIFIC PROTOCOLS

*SRADS* - The SRADS protocol [Reyn82] is, in terms of the design variables given earlier, aggressive, potentially inaccurate and potentially with risk. The potential inaccuracy and risk are determined by how accurately the protocol can predict the arrival time of messages at any LP. If the prediction is always correct, there is not potential for inaccuracy or risk. Otherwise, there is. SRADS is a protocol that is very well suited for applications where such predictions can be made, or where the inaccuracies introduced do not affect critical metrics adversely.

In SRADS, synchronization is enforced by limiting access to shared facilities (SF's), where a shared facility is a buffer that exists between an LP that can write to it and an LP that can read from it. The following simple sequencing rule must be met when reading from a shared facility:

> An LP may read from an SF (use a value) and proceed only if the LP writing to the SF has a logical time equal to or greater than the logical time of the reader.

The reader uses a special event called a *poll* in order to attempt access to an SF. At given intervals, determined by the protocol designer, the reader sends a poll to the writer, requesting information about the writer's logical time. Based on this time returned by the writer, the reader determines what actions to take. If the reader finds that its logical time is less than the writer's logical time, then the reader continues simulating. If the reader's time is greater than the writer's, the reader must wait until the writer's logical time becomes equal to or greater than its own.

Since polls are issued by the reader at regularly scheduled intervals, the reading LP can be prevented from simulating far beyond the writer's logical time, yet it can still receive a message that should have been received in its logical past. However, messages will always arrive a bounded amount of time in a reader's logical past at the very worst. We call this potential inaccuracy *time slip*. Since the occurrence of time slip can be controlled by a judicious choice of polling frequency, time slip is not as severe a problem as one might imagine.

An LP attempting to write never blocks unless it encounters a full shared facility. This assumption, coupled with polling is sufficient for a deadlock-free protocol.

Given a partitioning of a simulation, shared facilities would exist on each potential communication path between any pair of LP's. This means there may be many SF's

between a given pair of LP's. SF's would not need to be known to the applications programmer; they can be created by the protocol designer. Assuming these SF's are initialized by applying a filter when each application calls its respective process manager's initialization routine, the following filtering activity is required.

The *post-event* routine needs a filter to determine if an event is for a part of the application residing in another LP. If it is, there is an SF in another LP in which the event should be placed. The filter routine should request that the local node manager place it there. The filter routine should block if there is an indication the target SF is full and terminate otherwise. If it blocks it will unblock when a filter associated with the target LP's get-next-event routine makes the SF not full.

The *get-next-event* routine needs a filter to determine if an event is coming from an SF (rather than internally). If so, the routine needs to enforce the synchronizing requirement given above. If the get-next-event filter finds an empty SF, it emits a (non-event message) request (a poll as defined above) to the LP that writes to the empty SF. This request will always be answered. If the get-next-event filter removes an event from a full SF, it sends a *proceed* message to the blocked LP. If the event is a request to acknowledge a poll, the get-next-event filter sends a *proceed* message to the waiting LP, and then processes the next event.

The *post-message* routine that functions between a node manager and a process manager in an LP needs a filter to detect polls. Upon receiving one, this routine inspects the receiving LP's current simulation time. If it is no less than the polling LP's time then a *proceed* signal is returned to the polling LP. Otherwise, this filter schedules an event for the receiving LP to send a *proceed* message when its simulation time does equal or exceed the requesting LP's time.

*Null Messages* - The null message protocol [ChMi79], a non-aggressive, accurate, non-risk protocol, is simple to characterize: after setting initial conditions among LP's, special non-event messages, called *null messages*, should be sent on all remaining paths of a fork whenever an event message is sent along any path of the fork. A fork is a set of paths between a single LP and the LP's it may send messages to.

In SPECTRUM, this algorithm can be expressed quite easily by placing filters on three routine operations performed by the process manager. These operations are the initialization routine, the get-next-event routine and the post-event routine. We briefly describe each of the filters below.

The *initialization* filter is called once in each LP at the beginning of the simulation run. This routine initializes the data structures used by the post-event and get-next-event filters. In addition this filter sends an initial null message out on each of the LP's output lines.

The *get-next-event* filter enforces the null message requirement that an LP cannot process a message until it has a message on each of its input lines. The filter blocks the LP until this condition is met. Once there is a message on each input line the filter processes the message with the earliest timestamp. If this is an event message the filter passes the message to the applications program and returns. If the processed message is a null message and the application has no other event message to process, the get-next-event filter sends a null message along each of its output arcs. In this case the filter again blocks the LP until it has a message on each of its input arcs. This continues until an event message can be returned to the application.

The *post-event* filter is called whenever the application sends an event message on one of its output lines. In this case the filter sends a null message on each of the remaining output lines.

The *post-message filter* is called whenever the LP receives an incoming message from another LP. The filter then updates some of the data structures used by the get-next-event filter. These data structures are used to keep track of which of the LP's input lines have a message pending and which of the input lines are still empty.

*Other Protocols* - The appointments method [NiRe84] requires the setting of future times at which an event message may be sent. To accomplish this, one needs access to the internals of an application. This is a form of knowledge embedding. SPECTRUM supports protocols using knowledge embedding through the isolation of data structures as discussed above. It is assumed that filters can access those isolated data structures. This kind of protocol requires a more detailed understanding of a simulation than the approaches discussed above.

Time warp [Jeso82], which is an aggressive, accurate, at risk approach to parallel simulation, requires the identification of out-of-sequence messages. This is easily done using a filter on incoming messages. Time warp also requires periodic state saving, which is easily implemented as a filter that is set up once during initialization, and then is self-scheduling on a periodic basis.

## CONCLUSIONS

We see two major reasons for having a testbed such as SPECTRUM: to facilitate proof of concept and to facilitate performance comparisons. We have little doubt that SPECTRUM supports the first goal quite handily. We have demonstrated the ease with which some parallel simulation protocols can be implemented in our testbed. We have every reason to believe that it is just as easy to describe others.

Performance comparisons are another matter. We recognize that, ultimately, there is no substitute for a carefully crafted implementation of an protocol for a given architecture. SPECTRUM strips away some of the opportunity to do this, just as high level languages and virtual memory do for hard-core target-machine coding. However, just as with virtual memory and compilers, we believe that SPECTRUM may

also provide efficiencies that might otherwise be overlooked. For example, the library-provided event maintenance routines use very efficient insertion and look-up methods, as opposed to the linear routines used in most experimental implementations. Message passing in the node managers is customized for the iPSC/2. In short, we have attempted to implement the best alternative in each of the functions provided by SPECTRUM.

More to the heart of the performance question is this: significant performance changes can occur with minor changes in the sequencing of activities, in particular message passing. By implementing a protocol using SPECTRUM and filters, it is possible to develop an implementation that could be faster if it were developed from scratch. We're not inclined to believe that for the following reasons: 1) the functions provided by SPECTRUM are necessary functions - event maintenance, time management and message passing, among others - and they are implemented efficiently. 2) Filters do not really separate the protocol designer from implementation levels that may provide critical efficiency. In fact we think it more likely that we could be faulted for not providing enough abstraction. 3) There are plenty of anomalies in scheduling theory which have been observed in practice as well, where increasing resource demands can reduce total finishing time. It seems to us that a well designed SPECTRUM experiment has as much of a chance of producing optimal or near optimal performance as any other approach.

On the whole, we expect SPECTRUM to provide the opportunity for determining the *relative* performance of various protocols. We find that to be as useful a result as one could hope for. In a world where the underlying hardware technology is changing rapidly, testbeds like SPECTRUM are our best hope for testing performance questions quickly.

## REFERENCES


**[ChMi79]** Chandy, K.M. and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Trans on Software Engineering.*, SE-5,5, May, 1979, 440-452.

**[ChMi81]** Chandy, K.M. and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *CACM*, 24,4, April, 1981, 198-205.

**[ChMi87]** Chandy, K.M. and J. Misra, "Conditional Knowledge as a basis for Distributed Simulation," *CalTech Report*, 5251:TR:87, Sept 1987.

**[JeSo82]** Jefferson, D. and H Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism," *A Rand Note*, N-1906-AF.

**[Jeff85]** Jefferson, D., "Virtual Time," *ACM TOPLAS*, 7,3, July, 1985, 404-425.

**[Kuma86]** Kumar, D. "Simulating Feed-forward Systems Using a Network of Processors," Annual Simulation Symposium, Dec. 1985, 127-144.

**[Luba87]** Lubachevsky, B., "Bounded Lag Distributed Discrete Event Simulation", *Proc., SCS Multiconference,*, San Diego, CA, Feb., 1988.

**[Misr86]** Misra, J., "Distributed Discrete Event Simulation," *ACM Computing Surveys*, 18,1, March, 1986, 39-65.

**[NiRe84]** Nicol, D.M. and P.F. Reynolds, "Problem Oriented Protocol Design," *ACM Winter Simulation Conference*, Dallas, Texas, Nov., 1984, 471-474.

**[PeWo79]** Peacock, J.K., Wong, J.W. and E. Manning, "Distributed Simulation Using a Network of Processors," *Computer Networks*, 3, North Holland Pub., 1979, 44-56.

**[PeMa80]** Peacock, J.K., Manning, E. and J.W. Wong, "Synchronization of Distributed Simulation Using Broadcast Algorithms," *Computer Networks*, North Holland Pub., 1980, 3-10.

**[PoMi87]** Pope, A.R. and D.C. Miller, "The SIMNET Communications Protocol for Distributed Simulation," BBN Technical Report, BBN Laboratories Incorporated, Cambridge, MA, 1987.

**[Reyn82]** Reynolds, P.F. "A Shared Resource Algorithm for Distributed Simulation," *Proc of the Ninth Annual Int'l Comp Arch Conf*, Austin, Texas, April, 1982, 259-266.

**[Reyn88]** Reynolds, P.F. "A Spectrum of Options for Parallel Simulation Algorithms," *ACM Winter Simulation Conference*, San Diego, Dec, 1988.

**[Soko88]** Sokol, L., et al. "MTW: A Strategy for Scheduling Discrete Events for concurrent Execution",*Proc of SCS Multi-Conference*," February, 1988, San Diego, 34-42.

**[Word88]** Worden, J. "National Testbed Program," *Proc of SCS Multi-Conference: Aerospace Simulation III*," February, 1988, San Diego, CA.