Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

COMPARATIVE ANALYSES OF PARALLEL SIMULATION PROTOCOLS

Paul F. Reynolds, Jr,
Christopher F. Weight and J. Robert Fidler II
Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903
804/924-1039

ABSTRACT

Currently there is not a significant body of comparative, experimental performance results for parallel simulation protocols. Nor is there a body of significant analytic studies. The SPECTRUM Testbed [ReDi89] has been designed to support the empirical study of parallel simulation protocols and applications, with the expectation that experience with the testbed will provide insights into the efficacy of various protocols and their interplay with classes of applications. We discuss our experience with the SPECTRUM Testbed, focusing primarily on an observed, unexpected degree of dependency between protocols and applications, and on an unexpected, large set of application design options. The latter gives rise to the definition of a set of application design variables which describes a large design space. We discuss its impact on the testbed design and we discuss a limited set of performance results that we have for selected sets of protocols and applications.

1. INTRODUCTION

We have advocated (See [Reyn88]) the need for recognizing the existence of a spectrum of options for parallel simulation protocol designs. In particular, there is more to be considered than a simple optimistic vs conservative view of protocols: there is an infinity of practical protocol designs. To complicate matters further, it is very unlikely that one protocol design is universally optimal. Thus, in order to make useful statements about the relative performance of various simulation protocols, we must develop analytic and empirical tools that allow us to make comparisons *in a common framework*. The work discussed here concerns the empirical comparative analysis of protocols when applied to a set of applications in a common environment.

To date, other simulation studies have been fairly narrowly focused and inconclusive. Analytic results are beginning to appear (e.g. [Nico89]) but they do not consider important application characteristics. Our hypothesis, shared by many, is that the effectiveness of parallel simulation protocols will be highly dependent on the applications using them. Thus, an open question concerns the determination of the suitability of *classes* of protocols for *classes* of applications. The best way to determine this, short of analytic results that have so far eluded the community, is to study a mix of protocols and applications in a common test environment. SPECTRUM [ReDi89] has been designed to support this sort of activity. In this paper we report on the experiments we have conducted on the SPECTRUM Testbed and the conclusions we have drawn about simulation protocols and their suitability for selected applications.

Our experimentation goals are quite ambitious, and not yet fulfilled. The primary goal of our testbed activities has been to study protocols and applications in an environment that facilitated experimentation. We have achieved that, but not in the manner we originally expected. We have found that the dependence between protocols and applications is greater than expected, forcing us to redesign testbed interfaces and support routines more than once. This also means an application designer must be cognizant of the protocols to be employed. This latter observation is a disappointment because we had hoped to separate the application and protocol design processes more than appears to be possible. We discuss these results later in this paper.

We wish to identify classes of protocols and classes of applications for the purpose of our experiments. That will require insights into applications and protocols, as we explore below, and it will require feedback from individual experiments conducted on individual applications. So far we have applied two protocols to three applications. The protocols represent different degrees of aggressiveness and the applications represent a spectrum of application types. It is the latter, application types, where we have learned much about the interdependence between protocols and applications. For example, the null message protocol [ChMi79] must be modified considerably, depending on whether messages can be consumed, are queued, can be preempted, etc. We discuss these observations in more detail, also.

We give a brief overview of the SPECTRUM Testbed next. Details can be found in [ReDi89]. We follow with a discussion on designing experiments, where we explore the process of making a testbed such as SPECTRUM provide the information we seek. Much to our surprise the contents of this chapter have turned out to represent the most important lessons we have learned so far. In contrast, our small set of experimental results have provided far fewer insights. That will change as we construct and perform experiments based on the information we present in this section. Following that we discuss our experience with filters. Filters were a novel way to construct a testbed for parallel simulations; our experience with them has been, for the most part, positive. Next we present a description of the experiments we have performed. This section is relatively brief, primarily because of significant process-oriented learning and adapting that took place early on, as noted above. Finally, we close with remarks about where we think our approach is going to lead.

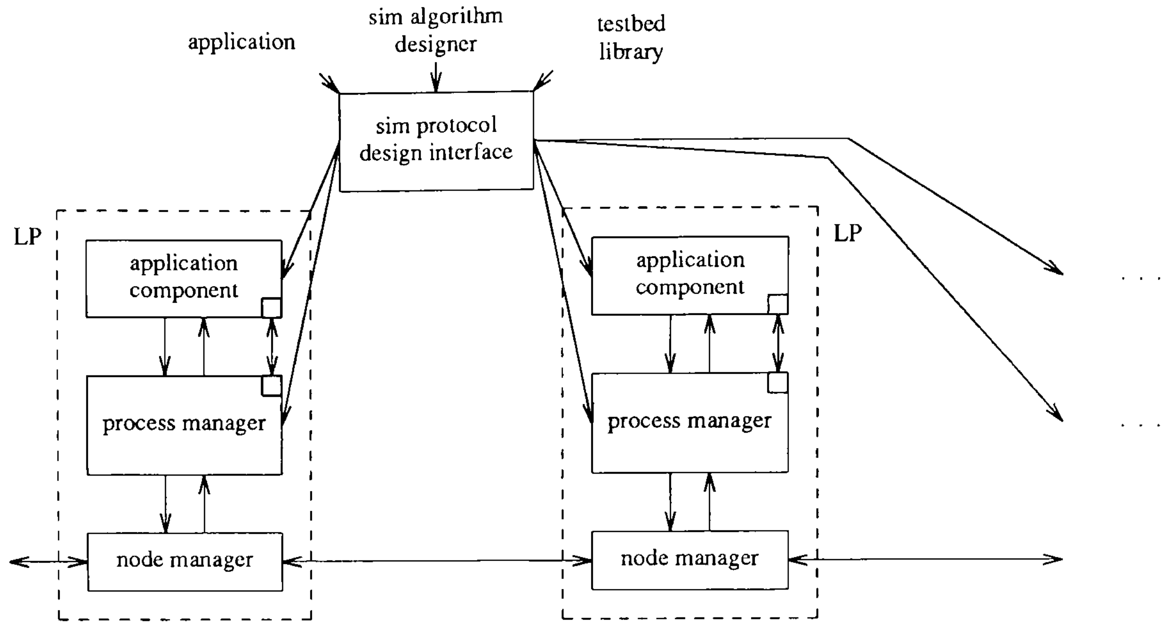

Figure 1. Block Diagram of SPECTRUM Testbed.

2. OVERVIEW: SPECTRUM TESTBED

As noted, SPECTRUM is a testbed for designing and evaluating parallel simulation protocols. It supports experimentation on different simulation protocols in a *common* environment. We have implemented versions on a BBN GP-1000 and an INTEL iPSC/2. The experiments discussed here have all been conducted on the BBN GP-1000.

As shown in Figure 1, a simulation protocol designer approaches the testbed possibly with an *application* brought from the outside. A *protocol design interface* provides a library of applications as well as a library of 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) in typical parallel simulation nomenclature. Node managers are provided by the testbed to support communication among logical processes. This communication can be either a message-based or shared memory paradigm; the node manager is optimized for the underlying hardware.

The simulation designer can implement a simulation protocol by associating *filters* with the routine operations performed in the process manager. For example, with a specification such as:

$$\textit{On post-event apply}(P_1, P_2);$$

the designer can indicate a desire to have procedures P_1 and P_2 called each time an application component calls post-event in the process manager. 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. Customization may be required depending on how much knowledge of the application is required.

In the same vein a designer can mix library-provided filters and procedures with customized functions and procedures. Even in the case where an entirely new protocol is to be tested, we expect many library routines will be useful.

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 τ.
- time-advance functions - advance the current simulation time to time τ.

These operations provide opportunities for a protocol designer to affect event ordering, non-event support, etc., for the application component through the process manager. Also, they provide a useful form of separation between an application component and the activities associated with

customizing parallel simulation protocols. In cases where application data objects must be accessed we require they appear in an externally referenceable data area.

A fifth operation is supplied by the process manager, this time to the node manager:

> post-message - post a message from another LP onto the local LP's event queue.

This operation supports inter-LP communication and, as noted above, can have an underlying implementation that is optimized for the host machine.

The P_i in the filters can be arbitrary procedures. Information for filter procedures 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 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 provides a disciplined approach to protocol design. As noted above we have found that while filters work, they do not provide a simple separation between applications and protocols. We have observed more dependencies (of a performance nature) than we had originally expected. Nonetheless, filters work. We discuss this more later.

We describe a set of filters for implementing the SRADS protocol [Reyn82] as an example of how filters work. The SRADS protocol assumes that receivers of messages can always predict the times when messages *may* arrive from other LP's. SRADS is very well suited for applications where such predictions can be made, or where the inaccuracies introduced do not affect critical metrics adversely.

Synchronization among LP's is enforced by limiting access to shared facilities (SF's), where an SF is a buffer that exists between an LP that can write to it and an LP that can read from it. A reading LP uses a special event called a *poll* in order to attempt access to an SF. At given intervals, determined by the protocol designer, (and dictated by the application) a reader sends a poll to a writer, requesting information about the writer's state. The writer will respond, when its state allows it, either with a message with a timestamp equal to the reader's polling time or with an indication that no events are forthcoming at that time. Before sending a poll of this nature, the reader first checks the SF to determine if the writer has already generated a message for the poll time. If so, the poll is unnecessary and not performed.

When performing a poll a reader has advanced its simulation time to that poll time, even though it may not yet have information from the writer about the potential arrival of a message from the writer at that time. The importance of this observation is that the reader will respond to polls sent to it as though it has advanced its time to that simulation time. As long as LP's impose a non-zero delay on all events (i.e. an arriving message representing an event at time, τ, will depart the LP at a time greater than τ), SRADS can be an accurate protocol (all events are simulated at the correct simulation time). As noted, this accuracy is dependent on the correct selection of poll times as well.

If an LP can receive a message from another at a time other than one of its poll times, or if messages can pass through an LP with no delay, then it is possible for LP's to receive messages in their 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.

In addition to initializing SF's, the *initialization* routine needs a filter to schedule the set of next polls for all of the LP's that can write to the initializing LP. Once initially scheduled, polls are maintained by get-next-event filters.

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 return 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 three things: 1) if an event is coming from an SF (i.e. another LP), 2) if an event is a local poll and 3) if an event is a poll from another LP. If the event is a value from an SF, the filter should pass it on to the local LP and update local information about the sending LP's time. If the event is removed from a full SF, the filter sends a *proceed* message to the blocked LP.

If the event is a locally generated poll, the filter should determine if the poll is necessary. It is if there are no future scheduled events from the LP needing to be polled. In that case the filter initiates sending a message to the appropriate LP and blocks the local LP until a response is received.

If the event is a request for a poll from another LP, a response is sent, thus allowing that LP to unblock. The next event in the local LP is then processed.

The *post-message* routine needs a filter to detect polls. Upon receiving a poll, 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. The post-message filter also processes *proceed* signals coming from other LP's. When a *proceed* is received the local LP is unblocked and processing resumes.

protocol type		application type
null messages SRADS appointments locally aggressive time warp moving time window	X	Stochastic w/ w/o balance, uniformity Deterministic w/ w/o balance, uniformity

Figure 2. An Initial (Naive) Set of Experiments.

We have designed and implemented filters for other protocols as well, including null messages [ChMi79] and SRADS with local rollback [DiRe89]. As we have indicated above, we have had to modify filters as we introduced new applications. It is too early to tell if filters will stabilize or if they will have to be continually updated as new applications are brought into the testbed. We will know better after we gain more experience with the testbed.

3. DESIGNING EXPERIMENTS

Early on we envisioned a set of experiments to be conducted on the testbed. Figure 2 shows the initial set of experiments we considered. The protocols are a representative set, but certainly not complete. The application types were meant to be fairly complete: stochastic representing queueing systems, pool balls, hockey pucks, ant farms, battlefield simulators and the like, and deterministic representing logic networks and stochastic systems that could be adequately represented by deterministic ones. The uniformity and balance factors were meant to capture process computational requirements and uniformity of distribution of process activations, resp.

As we began implementing protocols (null messages and SRADS initially) we brought up three applications to test them: a queueing network, a logic network and a battlefield simulator. It became apparent very quickly that our view of the variables in applications (e.g. stochastic vs. deterministic) was too narrow. We had overlooked the importance of whether or not messages are queued, consumed or delayed among other things. It turns out these variables have significant impact on the performance of various protocols (including whether or not they work at all). At this point we have identified the following as application variables that affect key decisions in protocol implementations:

- determinism
- queueing
- processing delays
- causality
- state change characteristics
- balance
- activity level
- connectivity

Determinism measures the degree to which an application exhibits deterministic behavior. Deterministic simulations are generally regarded as those which, when run on a uni-processor, always produce the same results for the same inputs.

An application that has *queueing* has an "arrivals and departures" paradigm where the time the processing of an event at an LP takes is dependent on the presence of other events. Queueing systems have queueing; logic network, hockey puck, pool ball and ant farm simulations do not.

A *processing delay* occurs if an event emitted by an LP has a simulation time greater than the arriving event that caused it to occur.

Causality is present in an application if every event arriving at an LP leads to an identifiable subsequent event leaving the LP. Variants include *consumption* and *production*: production occurs when messages can be created by an LP and consumption occurs when an LP takes in a message without, as a result of that action, subsequently sending one on to another LP.

State change characteristics measure how objects change state in a simulation: how many objects change and with what frequency. This variable is critical in determining an optimal state-saving strategy in an aggressive protocol.

Balance is a measure of uniformity of processing requirements. Balance depends on LP computational requirements and LP activation frequencies. Good balance is a function of how well an application can be partitioned for parallel processing.

The *activity level* in a simulation is a weighted measure of the number of LP's that are busy at any instant of time. We may speak of mean and variance of activity level as well as minimum and maximum activity levels for individual LP's. Activity level subsumes the notion of uniformity (uniformity of LP processing requirements) as defined above and used in Figure 2.

The *connectivity* of an application is a measure of how much events in one LP can directly affect events in another. Connectivity is multi-faceted: direction of information flow and topology are two components.

Most of these factors are not new. We have studied and reported the effects of the last two in previous papers (e.g. [ReKu86]) as have others. The significance rests in their impact on protocol design and simulation performance. As we developed filters for protocols we discovered that as we added new applications the importance of these factors came to light. We discuss some of these revelations next.

Initially we developed the null messages protocol and a simple queueing network application to test it. It is well known that the null messages protocol can lead to a

proliferation of null messages if some annihilation algorithm is not applied. After careful consideration we decided on an algorithm where any message (event or null) arriving at an LP would annihilate any null messages queued up from the same LP. This worked fine for queueing networks but failed miserably when we tried our battlefield simulation where messages could be consumed (lack of message causality). Messages that consume other messages and then die quickly lead to deadlock in the null messages protocol. We tried running the battlefield simulation with no annihilation scheme but the system became so flooded with null messages that it slowed to an unacceptable pace. Finally, we implemented a scheme where only null messages can annihilate other null messages. That scheme works, but it is clearly less than optimal for a system with causality and queueing. This was the first of many discoveries relating performance and application-dependent protocol design.

The null messages protocol, by definition, assumes a processing delay (minimum processing time). Our implementation of null messages worked fine for our queueing and logic network applications. It failed with the battle simulator because that application sends messages with zero delays. We note that the failure of null messages was due to our attempt to apply it in an environment which violated one of its basic assumptions: all messages incur a non-zero delay. The effect was that we had to modify our battle simulator (all messages incur minimum delays when passing through an LP) in order to make it work with null messages. The general effect was that an application had to be modified in a protocol-dependent way. This, of course, affects the testbed goal of low-effort mixing of protocols and applications.

As noted in the previous section, the SRADS protocol is designed for applications where potential message arrival times can be predicted or where the inaccuracies produced by inaccuracies in the predictions do not adversely affect simulation metrics. SRADS is well suited for deterministic systems, as our preliminary performance statistics given in a later section indicate. The filter descriptions given in the previous section work for a logic simulation. They also work for a simulation of a queueing system but not as well. In any stochastic system, where potential message arrival times cannot be predicted precisely, there is a modification to SRADS that makes it much more likely to process messages at the correct time. This modification is to require a reading LP to wait for all potential writers to have a simulation time that equals or exceeds its own before continuing to process internal events. However, as with null messages above, this is an application-dependent decision affecting performance.

Recently we began implementing a variant of SRADS in which we allowed LP's to do locally aggressive processing - meaning they could act on conditional knowledge [ChMi87] but not pass messages to other LP's if the content of those messages was dependent on conditional knowledge. We have dubbed this approach "SRADS with Local Rollback: SRADS/LR". As with Time Warp [JeSo82], [Jeff85] and other aggressive protocols, SRADS/LR requires a repair mechanism in the event the aggressive processing turns out to have assumed conditions that were not true. Rollback in turn requires a state-saving mechanism. State saving can be a very expensive proposition, as has been discussed in numerous Time Warp papers. We have found that it can also be application dependent if performance is a primary consideration. For example, some computations can be very compute intensive between significant state changes and others not. As a result there are cases where delta state-saving - saving only those objects which have changed state - could be significantly more cost effective than the saving of the entire state of a computation. The latter, of course, is the most general approach, but we can see cases where it would not be the most efficient, whether implemented in hardware or software. Without sophisticated hardware, delta state-saving must be done in an application-dependent manner. Furthermore, in those cases where the saving of an entire state is deemed most efficient, the optimal frequency with which entire states are saved can be application-dependent.

These examples illustrate application dependence in protocol design if performance (including deadlock freedom!) is a primary consideration. What it means for the testbed is filters must be designed in a fashion that allows inclusion or exclusion of features so that a protocol can be customized for a given application. What is not yet clear is how well this can be done. A better understanding will come with more experience with the testbed.

In Figure 3 we present the basis for a revised set of application types. In place of the three-variable design space we presented in Figure 2, we now have an eight-variable design space. The three variables, state-change, activity-level and connectivity are each multidimensional. For example, state-change can have frequently changing states with many state objects changing, or frequently changing states with few state objects changing, etc. The variable, causality, can have more bindings than we've shown (as can most of the other variables). For example, an application may have combinations of the three choices listed.

The combination of the options for the eight design variables shown in Figure 3 describes a set of 9216 unique application variations. Factoring in the variations one can describe just for the protocols listed in Figure 2, as well as questions regarding scale (performance dependence on number of processors) and statistical significance it is easy to see that one could easily construct on the order of millions of experiments. Even so we have not considered the full impact of variations in protocol design.

In [Reyn88] we showed nine design variables that were applicable to parallel simulation protocols. In essence, they formed a basis set for protocol construction in the same way that the variables shown in figure 3 form a basis for application construction. In the case of protocols, many of the design variables had essentially an infinite variety of possible bindings. Even, if we assume for the moment that each protocol design variable has only three bindings, that still produces nearly 20,000 different variations on protocols.

Without the common framework provided by a testbed such as SPECTRUM, we would have to consider all of the factors related to inter-processor communication times, processor interconnection topology, variations in event list maintenance routines and others. If those factors are eventually deemed critical for comparing various protocols, SPECTRUM will support experimentation with them. In the meantime, we continue our analysis assuming these factors are constant.

```
DETERMINISM
  - deterministic event processing.
  - stochastic event processing.
QUEUEING
  - first-come-first-serve queueing
  - pre-emptive queueing
  - no queueing
PROCESSING DELAYS
  - events can be processed in zero simulation time
  - all events require non-zero simulation time
CAUSALITY
  - one-to-one correspondence between
      messages entering/exiting LP's
  - messages can be consumed
  - messages can be spontaneously created
STATE CHANGE CHARACTERISTICS
  - state changes occur often          } X { many state objects change
  - state changes occur infrequently   } X { few objects change
BALANCE
  - LP's activated with ~equal frequency
  - LP's activated with unequal frequencies
ACTIVITY LEVEL
  - LP's have ~equal mean processing   } X { high variance
      requirements                     }
  - LP's have unequal mean processing  } X { low variance
      requirements                     }
CONNECTIVITY
  - (nearly) strongly connected        }
  - weakly connected                   } X {  unidirectional message flow
  - presence of cycles                 } X {  bidirectional message flow
  - presence of forks and joins        }
```

Figure 3. Basis for Application Types

We have mentioned protocol design options, application design options, testing for scalability and statistical significance as design variables, assuming that all other variables are held constant. The combination of these factors suggests, assuming ten tests for scalability and ten tests for statistical significance, that there could be on the order of ten billion experiments to be performed. This observation is what led us to suggest in our introductory remarks that experiments with *classes* of protocols and *classes* of applications was a reasonable (nay, necessary) goal. Without grouping into classes, the number of experiments to be performed is unmanageable. The challenge that remains is the identification of those classes.

Classification is possible. We have observed for example that if one works with a deterministic application, the other variables shown in Figure 3 are essentially irrelevant. It turns out a deterministic application partitioned across some number of processors will proceed at no more than the rate of the processor that has the most work to do. The other variables become important only if bindings slow down the simulation any more. Our initial experiments show that one protocol may dominate another in most if not all situations. As we learn more facts of this nature, we will again be able to scale down the extraordinarily large number of experiments we appear to face.

We discuss our experimental results in Section 5. Before that we discuss our experience using filters.

4. EXPERIENCE WITH FILTERS

We demonstrated earlier that a key to making the SPECTRUM Testbed easy to use is the concept of implementing protocols using filters on a selected set of simulation operations (initialize, get-next-event, post-event, time-advance and post-message). These operations are described in detail in Section 2. Having gained experience implementing three protocols, we discuss the practicality of this approach.

On the negative side we have observed that the design of protocols using the SPECTRUM Testbed can be conceptually difficult due to the limited access points the five operations provide to the protocol implementer. This is compounded somewhat by the layered nature of the testbed. There are times when certain aspects of protocol implementation would be more straight-forward if more access points were provided. Our initial observations were that debugging appeared to be more difficult and overall program flow appeared to be more difficult to follow. One additional undesirable result was that this organization presented a steep learning curve to new testbed users.

design vars	Queue Network	Logic Network	Battlefield Sim
	applications		
DETERMINISM	stochastic	deterministic	deterministic
QUEUEING	FCFS queueing	no queueing	no queueing
PROC DELAYS	all non-zero	all non-zero	some zero
CAUSALITY	one-to-one	one-to-one	production and consumption
STATE CHANGE	frequent/few	frequent/many	frequent/many
BALANCE	balanced	balanced	balanced
ACTIVITY LEVEL	~equal/low var	~equal/low var	~equal/low var
CONNECTIVITY	weak/uni-dir	weak/uni-dir	strong/bi-dir

Figure 4. Implemented Applications and Their Bindings

It may be that we were expecting too much from the testbed. It is natural for experienced programmers to want to implement something they understand well in the most direct manner. The selected operations force a discipline that, in the long run, may produce the most maintainable code. We have observed that implementation of the third protocol was greatly facilitated by the existence of the filters for the other two protocols. There was a great deal of borrowing of concepts and code, which suggests that the SPECTRUM library may be as useful as we had hoped. It is significant that one programmer has implemented three protocols in two months.

Also significant is the fact that filters applied to the five operations that support filters have been sufficient to implement the three protocols we have implemented. While the use of filters may have occasionally run counter to the instincts of a knowledgeable programmer, they have not stood as a significant impediment, or worse, been inadequate.

Not to be overlooked are the benefits gained from machine independence and the existence of common simulation functions. The testbed provides a common virtual simulation engine that hides a significant amount of detail without introducing perceptible performance degradation.

In short, we continue to be convinced that filters are the best way to implement protocols for the testbed. They are constraining in that they limit a programmer's implementation options, but in return they, in the context of the testbed, provide abstraction, access to a reliable simulation engine, and, as the collection of filters grows, a reusable repository of tested protocol functions.

5. A SMALL SET OF EXPERIMENTS

Currently, we have three applications and two protocols running on the SPECTRUM testbed. We have a third protocol, SRADS/LR, implemented but not fully operational. We emphasize that operational has a significant meaning here: a protocol is operational when it can work with any application in the testbed library. SRADS/LR currently works with the logic and queueing applications but not the battlefield simulator. The other two protocols work with all three applications. We expect to have additional protocols and applications in the near future and expect to report on them in a future paper.

The three applications we have implemented are: 1) a queueing network, 2) a logic network and 3) a battlefield simulator. Each of these represents a selected set of bindings for the design variables given in Figure 3. We show these bindings in Figure 4. As can be seen from Figure 4, there is no pair of applications that differs in less than three of the design variable bindings. We note this because it suggests there are many applications that lie on points "between" these more popular applications in the application design space we have defined here. The significance of these missing points remains to be determined.

The protocols we have implemented and made operational are null messages [ChMi79] (See also, [PeWo79].) and SRADS [Reyn82]. Both protocols work with the three applications described above. Given our experience with incorporating new protocols and applications into the testbed, we believe the filters implementing these protocols are rapidly approaching the generality we need for them to work with any application. (That is, generality without sacrificing efficiency. The filters have been designed to allow incorporation of functions as needed to satisfy the requirements of particular types of applications.) As we develop an appreciation for application requirements we expect to be able to do a significantly better job of specifying protocols for all possible applications *before* we implement the protocols. Our experience with implementing SRADS/LR certainly suggests that to us.

Due to difficulties encountered on our GP-1000 the experiments we have been able to perform have been narrow in scope. We have encountered problems with time-switching LP's on individual processors. As a result we have had difficulty performing typical parallel simulation experiments where constant-size problems are run on varying numbers of processors. We have had to settle for a mix of running varying-size problems on varying numbers of processors and running some experiments where we were able to coalesce the functions performed in different LP's, thus producing the constant-size problems on varying numbers of processors that we desired. In the cases where we had to vary the size of an application, we made certain we kept the same bindings (as shown in Figure 4) for the various sizes of individual applications. This was easily done in the case where we kept the problems a constant size.

Experiments	Trend with Processor Increase
variable problem sizes	[1.50, 0.96] --> [0.81, 0.81]
fixed problem sizes	[1.28, 1.09] --> [0.78, 0.08]

Figure 5. Trends in SRADS / Null Messages Relative Performance.

In Figure 5 we show the results of a set of experiments in which we compared the performance of SRADS with that of null messages. As just discussed, we performed one set of experiments with an application that increased in size as we added processors (a queueing network). The second set of experiments consisted of a fixed size application (logic network) performed on a varying number of processors. In each of the queueing network experiments we varied the number of jobs in the network. In the logic simulations we varied the number of inputs presented to the network.

The results shown in Figure 5 show a trend in which SRADS appears to perform better than null messages as the number of processors increases. The values in brackets show the range of relative performance times, SRADS to null messages, for varying numbers of jobs (inputs). For example, in the variable problem size experiments we saw a range of performances where, on a small number of processors (three), the SRADS finishing times ranged from 1.5 that of null messages to 0.96 that of null messages. In the same set of experiments the performance of SRADS on a larger number of processors (18) was essentially a constant 0.81 that of null messages.

We stress the preliminary nature of any conclusions we may draw from these results. We feel we have enough data to consider exploring a trend: as the number of processors increases, SRADS performs as well as or better than null messages. However, as we showed in section 3, there is a large number of application types, any one of which could negate the trend we see initially. Before declaring any trend conclusively, we intend to explore analytic support for it. We believe that is the most productive way, and the only reasonable way, to use the testbed.

6. CONCLUSIONS

We set out to determine experimentally how well various protocols performed with various applications. A key message in this paper is: we have learned far more from the process than we have from the outcomes of the experiments themselves. We had strongly suspected a dependence between applications and protocols. What we had failed to foresee was the number of factors - we've called them application design variables here - that had to be considered in applications. The importance of the comment that we need to identify *classes* of protocols and *classes* of applications has been made very clear. Without this coalescing, the task of studying protocols and applications is too huge. We are not prepared to conduct the billions of experiments we identified in section 3.

There is hope. Our early experiments have shown that SRADS tends to perform increasingly better than null messages as we increase the the number of processors and exceedingly better in some cases (logic networks). This suggests a trend that should be explored analytically. We expect other such trends to show up as we continue adding and testing new protocols and applications.

There are some simple observations that can be made, as well, that simplify the task. For example, we know that deterministic applications can only proceed as fast as the rate of the processor with the most work to do (assuming no load balancing). This observation dominates all other considerations; if an application is deterministic, bindings for the other seven design variables identified in section 3 are effectively irrelevant.

Are there other such simplifying observations? We expect there are and we expect that as we proceed with our experimentation they will come to light. In other cases, where we identify trends, we will have guidelines for analytic models. Thus, we see the testbed as a vehicle for supporting our original goal: understanding protocols and applications, albeit not by the means we had originally envisioned. Such is the way of experimental techniques.

ACKNOWLEDGMENTS

This research was supported in part by JPL Contract #957721. Thanks to Phil Dickens for all of his early development work and later support and to Eric Manning and the University of Victoria for providing an excellent opportunity to think and write.

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.

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

[DiRe89] Dickens, P.M. and Reynolds, P.F., "SRADS with Local Rollback", submitted to *SCS Multiconference*, Wash. D.C., 1990.

[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.

[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.

[Nico89] Nicol, D.M., "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations", unpublished manuscript, June, 1989.

[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.

[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.

[ReKu86] Reynolds, P.F. and Kuhn, C.S., "Three Variations on the SRADS Simulation Protocol," *Proc of SCS Eastern Multi-conference*, Orlando, April, 1986.

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

[ReDi89] Reynolds, P.F. and Dickens, P.M. "SPECTRUM: A Parallel Simulation Testbed", *Proc of 4th Annual Hypercube Conference*, Monterey, CA, march, 1989.

BIOGRAPHIES

PAUL F. REYNOLDS, JR., Ph.D., University of Texas at Austin, '79, is currently on sabbatical from his position as an Associate Professor of Computer Science and Director of the Institute for Parallel Computation at the University of Virginia. He has been a member of the faculty at UVa since 1980. He has published widely in the area of parallel computation, specifically in parallel simulation, and parallel language and algorithm design. He has served on a number of national committees and advisory groups including an oversight group for the National Testbed. He has been a consultant to numerous corporations and government agencies in the systems and simulation areas, and he has been a Research Associate at NASA, Langley, in Hampton Virginia since 1985. During the summer of 1989 he was a visiting member of the faculty at the University of Victoria, Victoria, B.C., and during the 1989/1990 academic year he will be visiting NASA Langley and working quietly at home.

Institute for Parallel Computation
Thornton Hall
The University of Virginia
Charlottesville, VA 22901
(804) 924-1039
pfr@virginia.edu

CHRISTOPHER F. WEIGHT received his Bachelor's degree from the University of Virginia in 1986. He is currently doing full-time research with the Institute for Parallel Computation at the University of Virginia. In the fall of 1989, he will begin graduate studies at the University of Massachusetts at Amherst to pursue a Ph.D. in Computer Science.

J. ROBERT FIDLER II received his Bachelor's degree from Randolph-Macon College in 1988. He is currently a graduate student in Computer Science at the University of Virginia. His research interests include parallel simulation and distributed operating systems.

Institute for Parallel Computation
Thornton Hall
The University of Virginia
Charlottesville, VA 22901
(804) 924-3180
jrf4d@virginia.edu