PARASOL: A Simulator for Distributed and/or Parallel Systems

SCS-TR-192

John E. Neilson
School of Computer Science
Carleton University
Ottawa, Canada
KlS 5B6

May 28, 1991

Abstract

This paper describes the kernel of a distributed/parallel system simulator/emulator which provides the user with "fine grained" control over system components including the hardware, the system software, and the application software architectures. The PARASOL system is truly an umbrella system in that it is equally suitable for performance simulation studies, with various degrees of model abstraction, for implementing and testing distributed/parallel algorithms, and for software prototyping. The kernel provides features for defining: heterogeneous network topologies of arbitrary complexity; dynamic task structures with a variety of management tools including task migration; interprocess communication via Mach-like primitives using communication ports; and spinlocks for low-level task synchronization. For those who are interested in performance simulations, PARASOL also provides the customary simulation tools for random variate generation and for statistics collection. Finally. PARASOL includes a monitor/debugging facility for monitoring and/or debugging user applications. PARASOL may be viewed as both "process-based" and "object-based". Its programming paradigm requires users to define PARASOL tasks, typically written in C. These together with built-in PARASOL entities are used to construct system models. The PARASOL kernel is highly portable and will run on virtually any system supporting simple, single-threaded applications written in C.

1 Introduction

Distributed and/or parallel computer systems have become increasingly important with the advent of low-cost microprocessors. Unfortunately, the design of such systems is difficult since performance issues are often of paramount importance and the analysis of concurrent systems from even a functional point of view is often non-trivial. Thus, there is a strong need for tools for both simulation (to answer performance-related questions) [1] and emulation (to answer functional questions) [2]. PARASOL attempts to address both of these requirements within a single tool. It is at once both a simulator capable of handling large-scale systems needing fast simulator performance and an emulator capable of emulating, at the code level of detail, the operation of concurrent systems.

Traditional event-driven, discrete simulation systems have proven useful for computer performance studies. However, these systems offer support for rather general system simulation and often leave much to be desired when modelling distributed/parallel computing systems. For example, general purpose simulators do not directly support the notions of the elements such as multiprocessors, communication links, etc. that make up computer networks. Furthermore, few such systems permit detailed or fine-grained emulation of the behaviour of distributed software including the often important communication and synchronization primitives, etc. On the other hand, interpretive simulation systems, often built on the principles of object-oriented programming, are capable of fine-grained emulation but often fall short on the performance side of the ledger. The advantages of a system that can support both flavours of use well are clear. Inevitably, after the dust settles on a design from the functional point of view, the next question posed relates to performance, e.g., "Given that the system meets its functional requirements, will performance be acceptable?" An integrated tool that can answer both questions has a huge advantage over two tools that are not well integrated and conceivably require two experts to use. This is particularly true when design iterations are required, where multiple strategies are to be explored, and when detailed customization is required.

The design goals for the PARASOL kernel were as follows:

1. Efficiency, i.e., simulation speed, was particularly important since it was anticipated that large-scale systems would be studied. Furthermore, the high computational demands of simulations requiring tight confidence intervals are well known.

2. Portability, meaning the ability to use on a wide variety of compute platforms, was deemed, important. While large-scale, high confidence simulation studies would require fast, number-crunching compute engines, small scale studies or functional validation studies may get by with comparatively "lightweight" engines.

3. "Hardware" configurability was required so that a wide variety of system configurations or topologies could be handled. Systems of both tightly and loosely coupled networks of processors were envisioned as probable candidates for study.

4. An easy-to-use user interface involving a small set of O/S-like primitives would be sought. These would be chosen to maximize the probability that they would satisfy basic user needs while, at the same time providing a foundation on which users could build more exotic interfaces if required or desirable.

At this point in its development, only the PARASOL kernel exists. However, this kernel is sufficient to serve as the foundation on which customized simulators and emulators can be built. In fact, PARASOL may be viewed rather as an operating system since it provides many of the traditional operating system services. i.e., resource management, scheduling, IPC, etc. In fact, it is much more since it also includes "constructor" primitives with which to build the simulated "hardware", a monitor/debugger with which a user can closely monitor the functionality of a model, as well as facilities for statistics gathering and random variate generation which are useful to simulation modellers. Although, in time it is anticipated that PARASOL will grow to include libraries to facilitate model development, debugging tools. etc., this first step, i.e., the development of the kernel is a necessary prerequisite. This paper outlines, in the next section, the realization of that goal by describing the design philosophy of the kernel and its user interfaces. Following this is a section that describes some of the details of the PARASOL user interface and provides many of the technical implementation decisions and details. A discussion of performance related matters and the degree to which the aforementioned design goals were achieved follows in the fourth section of the paper.

2 The PARASOL Design Philosophy

Experience has taught that there is a heavy performance cost to interpretive simulation systems. Furthermore, the ability of compiled programming languages to describe software efficiently is well recognized.

Thus a process-based simulation system seemed appropriate. The first design decision was therefore to employ compiled code for efficiency and to represent modelled software as PARASOL tasks. These entities would be coded in a compiled programming language. The language of implementation chosen was "plain vanilla" C for its widespread availability and its efficiency in writing "system oriented" code. Some may argue that C++ would have been a better choice but its 'object-oriented' features, although tempting, were judged to carry a significant performance cost that outweighed their benefits for kernel design where code reusability was not an important factor.

To model thee activities of software in execution across an arbitrary network of processors, a choice had to be made between a uni-processor PARASOL and a multi-processor PARASOL. Portability considerations as well as the inherent simplicity of a centralized management scheme suggest that the uni-processor solution is the most viable. Critics may argue that a parallel simulator running on a multiprocessor would be faster but there is little doubt that overheads would be larger. i.e., efficiency would be compromised. The quest for speed should be tempered by asking the question – Why? If indeed the answer is to supply computationally costly tight, confidence intervals for performance oriented simulations, the alternative of running slower replications efficiently in parallel on the available processor is clearly superior.

In short, the second design decision - to go with a simple, single-threaded simulator - was taken for reasons of efficiency, portability, and ease of implementation, It did beg the question however, viz., "How can a single processor simulate a potentially large-scale processor network?." The answer is clearly to use simulated "hardware". Three "pseudo hardware" elements were selected processing nodes; point-to-point, one-way communication links and multi-way communication buses. Nodes would be either uni-processor or homogeneous multi-processor entities having a common ready-to-run queue and queueing discipline. Basic links would be one-way to facilitate modelling topologies with one-way communication, e.g., rings. Pairs of links would permit two-way communication if desired. On the other hand, buses could be used to model networks based multi-way communications on a shared channel, e.g., Ethernet. In selecting these elements, choices were made between a more comprehensive "toolbox" and a simple or more fundamental one. For many PARASOL users, the simple, three element toolbox is believed adequate. It can also serve as a basis for the construction of a more comprehensive, non-kernel toolbox, Several such "second-level" PARASOL layers are likely to evolve in time.

A myriad of fine-grained design decisions resulted from this pseudo hardware approach. We will deal with but a few of these. For example, the scheduling ready-to-run tasks is a burdensome chore for most users. ideally, the PARASOL simulator should provide built-in scheduling to suit most user needs. Three basic queueing disciplines are supported by PARASOL: FCFS (first-come-first-served) - a non-priority discipline; HOL (head-of-the-Line) - a non-preemptive priority discipline; and PR preemptive-resume) - a preemptive priority discipline. Two variants of back of these exist depending whether or not a RR (round-robin) policy is implemented using a fixed quantum or time slice. Thus six different ready queue scheduling mechanisms are built-in. In fact PARASOL goes one step further by allowing tasks to be pegged to particular processor within a multi-processor node. This allows modellers the flexibility to mix static and dynamic task-processor assignments within a given node. Note that purely static task-processor assignment can be achieved by pegging tasks to processor within a multi-processor node as well as by using a multiple uni-processor node to model a single multi-processor. User should note that from the point of view of efficiency the latter approach is prefered since it splits the ready queue. Finally, another degree of flexibility is possible with respect to cpu speed. Each node carries with it a user defined "speed factor" so that networks can be constructed with elements of different processing speeds.

With respect to communication facilities, two issues arise: packetization and arbitration. For realism it was apparent that these issues had to be addressed. Again, however, the goals of efficiency and ease of use dictated choices. Presently, PARASOL supports packetized messaging in a simple manner. When defining a link or a bus, users are asked to specify the smallest unit of communication traffic. In this way, message traffic is coerced into units of the specified size. Thus packetizing is achieved at least in terms of the basic units of information exchanged. No attempt is made, however, to handle the transmission of these units independently, i.e., all of the packets comprising a message are simply transmitted contiguously. Simplicity is again the watchword with respect to arbitration. For link communications simple FCFS queueing is enforced, whereas, with buses, users can choose between FCFS and random arbitration. The latter is important since it is a much closer approximation to many CSMA/CD protocols. As with processing nodes, communication links and buses require users to specify device speeds so that heterogeneous networks can be constructed.

Let us turn now to the issue of modelling the software of distributed systems. Given the need for efficiency, compiled code was chosen as the vehicle for software modelling. That is to say, users are expected to describe their software model elements in terms of PARASOL tasks or programs that "execute" on the defined pseudo hardware. These tasks are written in C and make use of PARASOL "system calls" for many of the services traditionally suppled by the operating system. This "O/S approach" permits users to specify interactions between software elements in a natural and familiar manner.

The essential system calls included in the PARASOL kernel include those for task management (including dynamic creation, destruction, inter-node migration, suspension, etc.), inter-task communication, and task synchronization. Unlike many real multiprocessor systems, PARASOL tasks are single-threaded. This approach simplifies, through a more uniform view, the global management of tasks across a PARASOL network. It does, however, require somewhat more of the user to model multi-threaded applications since each thread maps to a PARASOL task. Through its efficient dynamic task creation system call, PARASOL eases this burden considerably. One should note that efficient dynamic task management also permits the accurate modelling of systems that make heavy use of "throw-away" or short-lived tasks. The implications of potentially rapidly changing task sets are that particular care must be exercised in the design of the kennel to keep modelling overheads to a minimum. PARASOL implements all user tasks as "super-lightweight" processes with a minimum amount of information in "non-local" dynamic structures. Perhaps the most unusual task management system call included in the kernel is the task migration call. This call simply recognizes the increasingly important role that dynamic load balancing plays in distrubuted systems and provides a suitable low-level mechanism for the modeller.

The matter of interprocess communication (IPC) or, more correctly in the case of PARASOL, intertask communication raised a number of design issues. One of the more basic questions here is the matter of communication style, i.e., synchronous vs asynchronous. In the end, it was decided to adopt the latter despite its problems with potentially unbounded message queues and inherently lower efficiency. One of the criticisms of the synchronous approach is that it is often quite awkward to break synchronism when it is desirable to do so. On the other and, synchronism is readily established with asynchronous communication through the use of acknowledgments or replies if blocking receive mechanisms are available. Furthermore. the inefficiencies in double copying in real asynchronous messaging systems are largely eliminated when simulations are involved as the actual passed data is usually minimal, e.g., a message type code as opposed to a real message. Besides the synchronous vs asynchronous issue, there are other important IPC considerations. In PARASOL we chose a "port-based" system styled after that employed in the Mach [] operating system. This means that messages are directed towards entities called ports rather than towards specific target tasks. Tasks may receive messages directed to ports which it "owns" and ownership is transferable to facilitate flexibility in message handling, e.g., for fault tolerance, and transparent load balancing. Message sending operations are non-blocking (thereby providing asynchronous messaging) whereas receiving operations are blocking with timeout, providing with zero timeout a kind of non-blocking receive. Three forms of port send operations are provided: a simple "non-delayed" send which imposes no load on communication facilities and indeed, does not even require them; a single-hop bus send which utilizes a specified bus and thus imposes a delay; and a similar single-hop link send. Although the PARASOL kernel does not support multi-hop communication directly, users needing such a facility can easily construct a second-level PARASOL system to do so. Besides simple single port sends, three forms of multicasting are provided:

global broadcasting to all existing tasks; local broadcasting to all tasks co-resident on the senders node; and multicasting to all descendents of the sender. PARASOL messaging supports both the notion of simulated messages (consisting essentially of just a type code and size specification) and, if required for fine-grained emulations, real messages of arbitrary content and size.

Besides the synchronization afforded by message passing (even the asynchronous variety), it was decided to provide other low-level primitives that would be particularly appropriate for modelling shared memory multiprocessor systems. Consequently, PARASOL supports a set of pseudo spinlocks that may be acquired in a mutually exclusive fashion through system calls. These allow users to efficiency model the sort of "busy waiting" that real spinlock systems provide. Again these are simple low-level kernel devices in keeping with the PARASOL design philosophy. If needed, user can construct second-level PARASOL synchronization mechanisms. Since simulations typically involve measurements and collecting statistics, the PARASOL kernel provides two rudimentary mechanisms. One of these permits the user to determine sample means of a population, e.g., response time, message length, etc., while the other allows one to find the time-average of some time function, e.g., processor utilization, queue size, etc. As is usual for PARASOL, the emphasis is on simplicity and efficiency. Consequently, no attempt is made to provide estimates of variance or distribution histograms. The former inevitably require either costly auto-correlation estimation techniques or blocked/replicated experimental design strategies which are best left to the user to devise. Kernel primitives are available to allow mean statistics to be captured and reset by the user to facilitate blocked experiments. Since the production of A distribution histograms also is usually best handled with post processing of a generated data trail, no kernel facilities were provided for this purpose. The stochastic nature of most simulation models requires the use of random number generators. PARASOL really does not provide anything unusual here. In fact it merely provides a few generators of common variates which are based on the ubiquitous uniform generator, drand48, found on most Unix systems.

In summary, the overall design philosophy of the PARASOL kernel is simplicity while emphasizing efficiency. The kernel is well under 4000 lines of C code in length which lends credence to this claim. While small, the PARASOL kernel offers the user considerable flexibility in modelling. As well it offers great scope for embellishment and extension through the development of secondary layers based on the kernel.

3 Details of PARASOL and Its User Interface

The previous section has laid the groundwork for a more detailed description of PARASOL. In this section we will emphasize some of its detailed design but concentrate mostly on the user interface. At this stage reference is made to the *PARASOL Users' Manual* which is available on-line together with the PARASOL libraries, demonstrations, etc. which is intended for serious PARASOL users. This paper is, on the other hand, intended only to provide an overview.

Mention has been made throughout of the necessity for efficiency in implementing a PARASOL kernel. One of the greatest challenges in this respect was to make PARASOL efficient with respect to scheduling, in particular, minimizing the number and costs of context switches between user tasks and the simulation driver itself. In many respects, this is the same problem that faces an operating system designer. In this case however, problems of handling physical I/O are nonexistent. However, they are replaced with the difficulties of dealing with networks of pseudo hardware and large-scale, global scheduling.

The characterization of global system state within PARASOL is worthy of some discussion. Clearly a PARASOL system involves a set of processing nodes. Optionally included are sets of links and buses. In addition other elemental entities exist, e.g., tasks, messages, ports, locks, and statistics. Furthermore, sophisticated users may and certainly the PARASOL driver will have to deal with events and the event calendar. These are the "objects" on which PARASOL is based. Some of these objects are relatively static in a given application, e. g., nodes, while others are very dynamic, e.g., events and messages. Still others lie somewhere in between, e.g., tasks and ports.

The management of such objects inevitably involves their creation and often their destruction. Since data structures are involved in their representation it is natural to think of acquiring memory for same dynamically. To do so using conventional O/S memory allocation and deallocation calls can be computationally expensive, however. For this reason PARASOL manages its own free lists of most of the building blocks of the required data structures. It also employs tables for many entities, e.g., tasks, so that they can be accessed by a more user friendly index rather than a pointer. The penalty for this approach, at least in the present form of PARASOL, is that it supplies fixed quantities of these things and users may run into limits. The work around is trivial requiring a redefinition of whatever table or pool sizes are troublesome. Automatic sizing is likely for future releases.

Event handling and task scheduling are very much interconnected activities of the kernel and its driver. While the driver (and its scheduler) can be viewed as just another task under PARASOL, it is important to realize that this task is special in that it is the only one not user written. Since PARASOL runs under virtually any single-threaded environment, the kernel creates a new multi-node, multi-task environment while it remains inherently singlethreaded. The PARASOL bootstrap process is a relatively simple one. First the driver begins execution under the host environment. After initializing itself, the driver creates a special node on which the first user task executes. In fact, this special node can be used for other purposes although it is frequently reserved for genesis, the first user task to execute. The driver then starts genesis and then becomes an event handler. Typically, genesis is responsible for creating the nodes and communication links of the simulated network. It also creates an initial set of user defined tasks to run on the network. Because it is user written, genesis can create whatever scenario the user wishes given the tools of the PARASOL toolbox. Once this is done, genesis steps aside and the simulation unfolds.

The foregoing description of the driver could give the impression that PARASOL is little more than a light weight process manager but, in fact, it must do more than simply schedule task execution. First, it must clearly execute tasks within the framework of the defined network. This, in turn, means that it must manage the pseudo hardware resources and most importantly handle simulated time. Time management is performed using the conventional event driven technique which sees future events posted in a “calendar” and time advanced only when all activity associated with previous events is complete. Events associated with PARASOL include those dealing with timeouts of various sorts and, perhaps more importantly, with the end of consumption of pseudo resources such as processors, buses, etc. To understand this point, consider the fact that at most one user task can actually be executing code at any given instant in real time while many tasks may consume processor tics simultaneously in simulated time. This illusion is created by time multiplexing between user tasks for real execution while holding simulated time constant. Subsequently, simulated time is advanced through the posting and subsequent handling of appropriate processor execution events. Although PARASOL is designed specifically to permit the asynchronous consumption of resources, synchronizing primitives involving spin locks or message passing can be used to synchronize resource consumption.

Scheduling of execution of tasks is perhaps the single most arduous chore of the kernel. Much thought went into the implementation. To be sure, efficient context switching is an important element in arriving at an efficient scheduler implementation but it is by no means the only consideration. A scheduler embedded within the driver seemed at first a sensible approach. Then it was realized that often there was no advancement of simulated time when the driver regained control and that this approach demanded two context switches as opposed to a single context switch if execution could be passed directly from one user task to another. Furthermore, decentralizing the scheduler logic meant that scheduler searches for the elusive “next thing to do” were usually much shorter.

To gain some understanding of the problems involved in scheduling is instructive to consider the so-called “executing” states of a task. Unlike a real system which has but a single state for tasks executing, the PARASOL environment had several. We will present a somewhat simplified picture with only four of these states discussed here. Clearly we must differentiate between a task which is really executing and those which would do so in a multithreaded simulation, i.e., those which are more than "ready" since they have already acquired a pseudo processor on which to run. These states are referred to as "HOT" and "BLOCKED" respectively. Additionally, when a task wishes to consume processor tics to simulate processor resource consumption, enters another state called the "COMPUTE" state. Typically this state ends when a specified amount of time has passed. Another execution state is the "SPINNING" state which is entered after a spinlock is requested and held until it is acquired. in all of these execution states, a processor is assigned to the task in question unlike, for example, a "READY" task which is otherwise keen to execute but lacks a processor. The goal of the distributed PARASOL scheduler is a simple one. Whenever the system state changes as a result of a system call or event happening, all tasks that can possibly move to an execute state are so moved. On realizing that a task is no longer HOT, the scheduler checks the pool of BLOCKED tasks and performs a direct context switch, if possible. Notice that this may be possible even to a task on a remote node! As a last resort, a context switch is performed to the driver which extracts the next event from the calendar, advances simulated time, and handles the event. The latter activity may invoke the scheduler and cause a context switch to a user task.

This concludes the description of the kernel implementation of PARASOL. As indicated previously, the kernel is but the foundation of the full PARASOL system which should include specialized libraries for particular applications; user friendly interfaces, perhaps of the GUI variety, to simply the matter of model definition; and post-processing systems for analysis of simulation U results. The next section discusses performance related matters and reviews the extent to which the PARASOL kernel met its design objectives.

4 PARASOL Performance

Since system performance was held as a primary design goal, it is appropriate to discuss at some length the performance levels achieved with PARASOL. Although experience to date with PARASOL has been limited due to its newness, highly satisfactory levels of performance have been achieved. It is clear that without a defined context, absolute measures of simulator performance have little practical value. Simulator users are usually more interested in how fast any given simulator is solving their application. Lacking appropriate simulator benchmarks, we will describe the performance levels achieved with two quite different applications.

First, a fairly complex distributed data base application was realistically modelled using more than 200 PARASOL tasks. The system involved an Ethernet-like network of two types of processing nodes including a transaction concentrator and a multithreaded data base manager using a spin-lock protected data base cache. Besides modelling the software involved, PARASOL tasks were used to model customers and the disk devices. Although not an emulation, this model would clearly qualify as a "fineto-medium-grained" simulation with simulated execution "slices" as small as 1 millisecond. On a SUN IPC SPARCstation, this model ran roughly ten times faster than simulated time.

A second application was a generalized stochastic rendezvous network (SRN) simulator. Although this model is highly abstract in that its parameters define the stochastic as opposed to the deterministic behaviour of software, it involves inter-task communication of a purely synchronous variety, i.e., blocking send operations are involved. This style of IPC forces more context switching overheads while minimizing message copying - a consideration of little importance for simulations when pseudo messages are involved. On the other hand, since the PARASOL kernel does not directly support synchronous message passing, a pair of send/receive system calls was needed to simulate each blocking send. Comparing PARASOL SRN performance with that of an earlier "communicating-state-machine" based simulator[], revealed a speedup factor of approximately 4 on a SPARC IPC and over 10 on a Sun 3.

Let us now tum our attention to more detailed performance considerations. First, it should be noted that the kernel was written in C in its entirety for maximum portability and ease of maintenance. A natural question is - "How much perfomrance is lost with this approach?" Examinations of the execution profiles suggest that there is comparatively little to be gained through fine-grained optimization. Running on the SUN SPARCstation, typical execution profiles showed that a large proportion of time ($> 40\%$) was consumed for Unix services, most of these apparently servicing a trap required in performing a context switch. Approximately 30% of the execution time was devoted to kernel code including event handling and scheduling. Finally, about 25% of execution was associated with user code including PARASOL system calls, random number generation, and statistics gathering. Although the application sample size was comparatively small, most PARASOL applications can be expected to behave similarly.

Perhaps a useful absolute performance measure is the overall task context switching rate since this seems to be relatively insensitive provided the application has a reasonable degree of inherent concurrency. Furthermore it can provide at least an estimate of the performance of the simulator for a given application. A context switching rate of approximately 7000 switches per second was observed on a 15 Mip SPARCstation. Performance as seen on an older CISC based SUN 3 machine was slower at about 2000 switches per second. It is noteworthy that the proportion of Unix service on the CISC machine was much reduced (less than 5%) indicating that context switching overheads are proportionately much lower for the non-RISC architecture. Given that approximately half the SPARCstation execution is associated with context switching overheads, this area clearly offers the most potential for performance enhancements. At best, however, a factor of 2 in simulator speed is achievable from this one source and, thus far, we have not succeeded in obviating the expensive O/S trap. Indeed, if the SUN 3 experience is indicative of CISC machines, a more fruitful area to mine for performance improvements might be the simulation driver scheduler area. However, even this source could not yield performance gains of more than 50%. To be sure, the use of double precision floating point arithmetic for timekeeping as well as statistics collection is computationally expensive. However, the alterative of scaling time and employing long integer arithmetic are unattractive from the user point of view and would not offer significant performance gains. Execution profiles indicate that typically less than 10associated with statistics collection. The coarsegrained design decision to make statistics collection userdirected was apparently useful in achieving this relatively low overhead. Finally, it should be noted that a widespread and relatively even distribution of computational effort was seen in execution profiles. In summary, future kernel performance gains of more than a factor of 2 are unlikely. From the user's perspective, performance is largely determined by the amount of context switching involved in the PARASOL model. Care should therefore be exercised to minimize the use of roundrobin and/or preemptive queueing disciplines if simulator speed is important.

Turning our attention now to discuss how well the "other" PARASOL goals were achieved. By implementing PARASOL purely in the C programming language and using a standard C library, the portability of PARASOL is assured. At worst, some adjustments may be required to define smaller default table sizes when memory is scarce and some fine tuning may be appropriate with respect to the debugger interface, random number generation and/or context switching. All of these, matters are well localized in the code making portability easy. With respect to "hardware" configurability, the pseudo-hardware approach ensures that a wide variety of system topologies can be modelled with relatively little effort. In addition, the evolution of "secondary" PARASOL libraries will facilitate detailed O/S emulation. Finally, the kernel user interface is comparatively small making it both easier to learn and to use. Fewer than 40 different system calls are involved in the initial version of the kernel. Given the fact that these calls are used not only to provide "standard" O/S-like services but also to configure the pseudo hardware and collect and report statistics, the "learning threshold" is low for PARASOL users, particularly for those who are familiar with writing software and interfacing to an O/S.

5 Future Developments

This discussion will concentrate on the future developments of the PARASOL kernel although we note that development is desirable and possible in intermediate-level, O/S-type layers to model more closely the functional interfaces and behaviours of real distributed/parallel operating systems as well as in specialized application areas. Another possibility is the integration of PARASOL with an existing software design tool such as TimeBench [3]. It is hoped that these developments can be accomplished without revision of the kernel although the development of additional libraries is probable. One candidate O/S layer would be one designed to give PARASOL a more Mach-like [4] interface. Other promising distributed/parallel O/S's such as Amoeba [5], Sprite [6], or Harmony [7] are likely to follow which would provide valuable tools for comparing systems. Any such work will require the development of PARASOL tools for modelling users and workloads, and hardware devices such as disk drives and networks.

Several specialized application developments are anticipated. The first of these is a tool for we performance studies of generalized stochastic rendezvous networks that is currently under development [8]. Pedogogical tools based on PARASOL are also likely which may do little more than provide "training wheels" to the full PARASOL package or which may adapt PARASOL to some specific application area, e.g., massively parallel processing.

With respect to the kernel, it is hoped that its present services will suffice or at least be augmented in only minor ways. No significant changes are contemplated at this time. On the other hand, it is clear that more sophisticated user interfaces, perhaps GUI, would be useful for model development and monitoring performance. Earlier work [9], proved the viability of a GUI based, front-end code generator to a C based simulator for specifying distributed systems of hardware and software. While not strictly a kernel development, such a front-end code generator for PARASOL would be useful even if it generated code only for genesis, the user specified PARASOL task that defines the hardware configuration, the initial system of tasks, and can exercise control over simulation experiments. The performance monitoring and debugging functions of the PARASOL kernel are primitive at present. A more sophisticated window-based interface for monitoring/debugging would greatly enhance PARASOL. These user functions can be very difficult at the best of times and are particularly so when dealing with highly concurrent software systems with complex system states. While it is true that a general purpose debugging tool such as dbxtool [10] can be useful within the PARASOL context, the fact that it is not tailored specifically to the multitasking PARASOL environment means that even it falls well short of what is desirable.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Telecommunications Research Institute of Ontario (TRIO) whose support is gratefully acknowledged.

References

[1] H. Davis, S. R. Goldschmidt, and J. Hennessy. TANGO: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Computer Systems Laboratory, Stanford University, July 1990.

[2] G. M. Karam and J. T. Wilson. MLOG: a language for prototyping concurrent systems. In *Proceedings of the 1990 Canadian Conference on Electrical and Computer Engineering*, pages 42.2.1–43.2.6, Ottawa, Canada, September 1990.

[3] R. J. A. Buhr, G. M. Karam, C. M. Woodside, R. Casselman, G. Franks, H. J. Scott, and D. Bailey. TimeBench: A CAD tool for real-time system design. Technical report, Department of Systems and Computer Engineering, Carleton University, September 1990.

[4] R. F. Rashid. From RIG to Accent to Mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society 1986 Joint Computer Conference*, November 1986.

[5] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 90's. *IEEE Computer*, 23(5), May 1990.

[6] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, pages 23–36, February 1988.

[7] W. M. Gentleman, S. A. MacKay, D. A. Stewart, and M. Wein. Using the Harmony operating system: Release 3.0. Technical Report NRCC 30081, National Research Council Canada, February 1989.

[8] C.M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar. The rendezvous network model for performance of synchronous multi-tasking distributed software. Technical Report SCE-89-8, Department of Systems and Computer Engineering, Carleton University, March 1989.

[9] H.G. Brauen and J.E. Neilson. Graphically defining simulation models of concurrent systems. Technical Report SCS TR 141, School of Computer Science, Carleton University, September 1988.

[10] Sun Microsystems, Mountain View, CA. *Debugging Tools*, 800-1775-10 edition, May 1988.