*SAMS*, 2002, Vol. 42, pp. 1575–1599

Taylor & Francis
Taylor & Francis Group

# RUNTIME SUPPORT FOR COLLABORATIVE AIR POLLUTION MODELS

EMMANUEL VAVALIS*

*Mathematics Department, University of Crete, 714 09 Heraklion, Greece and CSI/FORTH, 711 10 Heraklion Greece*

*(Received 6 July 2001)*

We present the design of a computational methodology and a prototype software environment and infrastructure that promotes collaboration in air pollution simulations in a highly interactive way during the development and use of the simulation engine over scalable distributed systems of heterogeneous computational components. In particular we focus on the software runtime system to support such a distributed simulation engine.

*Keywords:* Air pollution; Distributed high performance computing; Multiagent systems

## 1. INTRODUCTION

During the past two decades the modeling of atmospheric pollution has proven to be a very challenging and a very important problem [13]. The main difficulties are due to: (1) the complexity of the physical and chemical processes involved; (2) the fact that the associated mathematical formulation is not well established yet; (3) the increase need of significant amounts of various input data of certain "quality" from different sources; and (4) to the usually high CPU requirements.

We have recently proposed a new distributed methodology for efficient and accurate simulation of air pollution phenomena [37,45]. This multidisciplinary effort is still underway, and is based on

*E-mail: mav@ics.forth.gr

collaborative operations of two kinds: cooperative operation between distinct but neighboring numerical solvers (simulation engines), and also collaboration between users/modellers who interact with neighboring solvers during model development, experimentation and production mode operation. This methodology leads to scalable, collaborative operations on heterogeneous networked machines and requires a runtime support system.

In this paper we present the design of an efficient and portable runtime support system which is based on modern techniques and practices from the software systems area. Although the novel software system proposed here is targeted to a distributed air pollution model it can be easily adapted to a broad class of applications that both involve multi-physics phenomena on multi-domain geometries and teraflop-level computations.

The rest of this paper is organized as follows. Next we briefly describe the main characteristics of a truly distributed long range air pollution model and the rest of the paper is devoted on the specification of a runtime support system needed for the effectiveness of such a model. In Section 3 we present the general architecture of the proposed runtime system together with details on the various issues related to our model. In the appendix we provide basic information about the various software toolkits we mention in this paper.

## 2. COLLABORATIVE AIR POLLUTION MODELS

We now know that the simulation of highly complex physical phenomena can be done by viewing the system in terms of sub-problems or groups of sub-problems. Initially, independent solutions are obtained for each sub-problem, and estimates are made of values of concentration functions at interfaces. By exchanging function values across sub-problem boundaries, a global model is had. For each sub-problem, the analyst can make change or add to the kinetics or the chemistry, without modifying the rest of the model or the interface mechanisms. Naturally, different numerical approaches may be used on distinct sub-problems.

The main difficulty in air pollution modeling is the complexity of the physical and chemical processes involved. In simplification, one

partitions the domain into sub-domains, thereby splitting the model into distinct (simpler) submodels [37,45]. Assuming that one can solve exactly any single partial differential equation (PDE) that governs the air pollution model on any simple domain (or, more realistically, given such a PDE problem one can select a highly accurate solver for it from one's library), the *interface relaxation method* iterates through the following steps:

1. Guess solution values (and derivatives if needed) on all interfaces.
2. Solve all single PDEs exactly and independently with these values as boundary conditions.
3. Compare and improve the values on all interfaces using a relaxer.
4. Return to Step 2 until satisfactory accuracy is achieved.

Fortunately, the splitting of an air pollution problem can be done in an easy and natural way since the transition zones between sub-areas are known a priori. Coasts, the limits of urban areas and strong variations in the landscape are examples of such zones that can be easily identified. Most of the chemistry sub-models defined this way can be significantly simpler then the universal model but need to be coupled together by imposing certain conditions on the interface boundaries [37,45]. Although the various schemes that are already in use [36] (which are solely based on the continuity or discontinuity of the concentration function and its flux) can be used for interfaces, we strongly believe that more complicated conditions that naturally reflect the particular atmospheric and meteorological characteristics will be powerful tools in expert hands. The formulation of such advanced relaxers is a challenging and open problem.

Interface relaxation is naturally suited for distributed high performance computing.[1] The method defines a mathematical network with a single PDE solver at each node (representing a domain) and relaxers connecting the nodes. One distributes the single (and usually different) PDE *or other* solvers to high performance machines (since many subproblems have to be solved) and let the relaxers "control" the computation. However, even given the existence of independent solvers, such experimentation is not possible without efficient runtime system support that provides the appropriate functionality: scalability,

[1]A crude form of interface relaxation is already in fairly widespread use – a "trade" of current values across interfaces without any relaxation.

balanced-loads, co-operative mechanisms, multiway communicaton, and realtime protocols for collaborative interactions at the analyst's level.

## 3. RUNTIME SUPPORT

Our objective is to develop a computational methodology and software environment that supports collaborative air pollution models. We focus on: (1) runtime support for collaboration between models; and (2) runtime support for user-level collaboration between analysts (simulation engine developers and/or users) who interact with models on neighboring subdomains. At the simulation-engine level, collaborative support implies efficient mechanisms for managing distributed threads and synchronization (API, functionality and efficiency issues) and scalability (threads, and communication). At the user-level, collaborative support implies efficient communication protocols (multiprotocol support, realtime, multiway, low-latency and high-throughput communication) that provides all the requisite efficiency and functionality for user-level collaborative interactions.

### 3.1. The Proposed Architecture

We propose the architecture shown Fig. 1. A major portion of the research in this paper deals with issues at the *lowest layer*: here software systems like ARIADNE and ARACHNE (see Appendix) will offer threads functionality, while others like CLAM will offer highly efficient (as will be seen shortly) communication functionality.

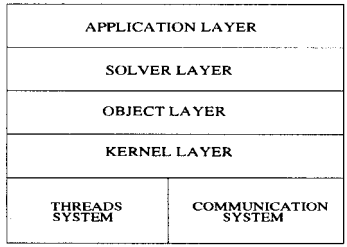


FIGURE 1 Architecture for collaborative solver.

The *kernel layer* exports ARIADNE or ARACHNE (see Appendix) threads to the *solver layer*, to new or legacy solvers like ELLPACK, and provides support for threads scheduling and process synchronization via speculative executions [26,28]. That is, we have to provide a methodology in which *application layer* code can be written without use of host ids, explicit send/receive or processor synchronization primitives, with appropriate help from the *solver layer*, *object layer* and *kernel layer*. Using migrant-threads and object proxies, we have had considerable success in implementing optimistic and speculative executions in the PARASOL system. We propose to borrow a page from the PARASOL book to synchronize processors. All this functionality will reside within the *kernel layer*.

The *object layer* is an *application-independent layer*, but requires object definitions for every interface (boundary) in the problem space. Because these definitions depend on the data (problem) and not the solver, they should be defined by class libraries which represent different boundary geometries. With this, the *object layer* is made completely independent of the data structures used by legacy systems, and also provides for object definitions in new multithreaded solvers. We initially plan to offer users a bind_subdomain(A, hostid) primitive that, at the start of execution, statically binds subdomain A to a solver running on processor hostid. Since the *object layer* will be equipped with an object location mechanism, users are relieved of the responsibility of programming in terms of processor ids. Once bind_subdomain(A, hostid) runs to establish the object-processor map, any invocation of object A by a thread automatically and transparently migrates the thread to processor hostid which hosts A. This system have been very effectively used in both ARIADNE [30,31] and PARASOL [27,28].

Because a collaborative solver must iterate and communicate with neighboring solvers in a chaotic way, communication is highly irregular. To enable such unpredictable communication patterns and yet enhance communication performance (i.e., by eliminating high polling costs which involve several layers of a communication protocol like TCP/IP), and to make speculative executions possible, all (interface) objects in the object layer must provide *save* and *restore* methods [28], so that they can be transparently checkpointed at appropriate times that are indexed by iteration indices. These checkpointed states

will be (transparently) used to restore objects, wherever necessary, during speculative computations [27,28].

The *solver layer* provides for either legacy or new solvers. Using envelopes, modular legacy solvers like ELLPACK can be implemented as three fat threads: (1) an "input" thread that a user can interact with for dynamic parameter modification; (2) an "output" thread which offers dynamic display [29,32] of simulation results; and (3) a "solver" thread that iterates over its subdomain, given boundary values. New solvers can be fully designed in terms of an arbitrary number of threads, to maximize the potential benefits of design simplicity, concurrency on multiprocessors, and performance. For example, even with as few as three threads for legacy solvers like ELLPACK, ARIADNE's time-slicing feature enables separate actions to be performed concurrently: execution, output visualization and run-time modification of parameters. Depending of the legacy code, further concurrency and object-definition support for checkpointing and restoration (to offer more flexibility at the application-level) may be also be possible.

Finally, application-code is developed at the *application layer*, using primitives provided in the *object layer*. As explained below, different types of threads may be created at the application-level, with support from below. ARIADNE's thread-migration capability is exploited in the kernel level, based on a method that is successfully used in the PARASOL system [26,28]. The *kernel layer* offers applications access to migratable threads as follows. Distinct processors are given object proxies for all objects that they do not host, but which are resident on other processors. When a thread invokes a method on a proxy, the thread invokes code provided in the *object layer* of the architecture. This code transparently packs the thread, migrates it to the processor on which the real object resides, unpacks the thread, and finally makes it available to the threads scheduler at the target host. Each thread is scheduled for execution based on its priority. Thread migration offers significant benefits: program simplicity, locality of reference, and one-time transmission. If a thread repeatedly accesses large remote objects, it makes sense to migrate the thread to the remote host. Data migration will be poor if data size is large or replicated write-shared data is used. Thread migration will be poor if a thread's state is large relative to the average amount of

work it does between consecutive migrations. This is equivalent to sending large messages frequently. Migrating a thread is as efficient as passing a message which contains the information on the thread's stack. It has been found this to be an often misunderstood feature of thread migration vs. message passing. Details on migration can be found in [26,28,30].

### 3.2. Issues in Synchronization

We have introduced above the notion of Solver threads (S-threads), Interface threads (I-threads) and Control threads (C-threads). Because S-threads operate on subdomains while I-threads operate on boundaries, a single iteration of an S-thread over its subdomain takes an order of magnitude longer than a single iteration of an I-thread over its interface. Processors hosting S-threads communicate asynchronously with processors hosting I-threads. Indeed both types of threads may reside on the same processor/multiprocessor. Using functionality from CLAM, each I-thread (which computes interface values) may either post an active-message [22] containing relevant boundary (interface) data to each neighboring S-thread (solver), or this data may be relayed by threads which repeatedly migrate between I-thread hosts and neighboring S-thread hosts. In either case, an S-thread begins its next iteration step with new boundary data and – as explained in the previous subsection – explicit send/receive primitives, synchronization actions or processor ids need never be used by model developers at the application-level. Whenever an S-thread completes an iteration step, it uses the reverse process (an active message or a relay-thread which migrates back to each neighboring I-thread host) to offer each neighboring I-thread new boundary data based on the interaction step it just completed.

Execution is speculative in the following sense. At the end of an iteration step, an S-thread sends its boundary data to neighboring I-threads. It may choose to begin its next iteration step without receiving interface data from all neighboring I-threads. Such data may not have been received due to a variety of reasons (e.g., slow network, some S-thread is more computationally taxed because of its subdomain, or because of competition for the CPU). However, whenever

an I-thread receives interface values from S-threads, it is free to checkpoint the state of its interface objects (e.g., an array of boundary values), indexed by iteration number. With kernel and object layer support, checkpointing is completely transparent to the application-level. An I-thread may use any one of a number of methods to select a "best" interface object (i.e., the best boundary values) based on an error-norm, and migrate this object to neighboring S-threads. In this way, S-threads continue computing – whenever they are unsure if new interface data will ever arrive – but may later be forced to take "better" routes when and if such data eventually arrives from neighboring I-threads. A second level of speculative execution can be had by checkpointing object states for data structures within the solver, so that recomputation is avoided in those problems which may require more complicated evaluations on subdomain interiors. This is hard to provide in legacy systems, but is highly recommended with new solvers because recomputation can be avoided whenever a solver's object states are checkpointed. In complex physical simulations, this technique can yield significant savings [26–28].

C-threads are defined at the application-level to enable user-defined policies for convergence. These threads invoke methods in the object class that define appropriate control policies, which in turn decide the frequency and type of checkpointing done. Like the S-threads and I-threads, C-threads run in time-slicing mode and share a host processor's CPU time-slice with other threads, based on thread priority. We plan to enhance ARIADNE's timers to support highly accurate internal timing by exploiting context-switching intervals to examine delta-queues and expired timers.

### 3.3. Issues in Integrating MultiProtocols and User-Space Threads

To achieve better efficiency and functionality we have to eliminate the traditional layering that separates communication from computation to provide *an integrated system of threads which support both communication and computation*. Experience [22] indicates that the net result is increased CPU efficiency, low latency, high throughput, increased scalability, and a capacity for multiprotocol support. The underlying research question is: how is this to be done?

How can user-level threads be used in the development of efficient, scalable multiprotocols with multiway communication support? And, how can methodology from our software subsystems be integrated to provide transparent (speculative) synchronization, migration, checkpointing and restoration. In employing user-level threads, the main idea is to perform efficient scheduling of multiple compute- and communicate-functions that are integrated within a single process, so as to minimize the effects of OS-level context switches. The resulting protocol actions will be efficient only if thread operations are efficient, and if threads are scheduled in a manner that minimizes packet-delays. Tailored thread schedules [21] will help reduce packet-loss (and thus retransmission delays) at a receiver's transport layer, simultaneously increasing client processing ability. We believe this approach is valuable because these factors are critical in realtime or low latency message delivery (to support user-level collaborative interactions) and efficient multiway transport.

### *Problems with Single-Threaded Communication*

Consider the standard communication model, e.g., for workstation networks, shown in Fig. 2(a). Processes $A$ and $B$, on machines $X$ and $Y$, respectively, open sockets and obtain receive-buffers. The OS kernel routes a process's incoming packets from its network interface to its receive-buffer. Let labels $W$, $S$ and $R$ represent a process's *work*, *send* and *receive* functionalities, respectively. Ordinarily, a single-threaded process must "complete" an invoked function before invoking another. Though non-blocking calls are permitted, these can be expensive (e.g., probing a buffer for packets). If process $A$ is in $W$ when process $B$ sends it a string of packets, $A$'s packet-retrieval cost involves either a high polling overhead, or a high latency overhead (since it must complete $W$ before a read operation). In the former case (a) computation is unnecessarily taxed, and (b) polling constructs complicate applications. In the latter case, packets are dropped when the buffer is full. Buffers sizes are limited, and our experience [21] with Ethernets is that even large buffers routinely overflow (besides consuming valuable space); faster networks imply potentially higher losses. Because a process's receive-buffer (or buffers,

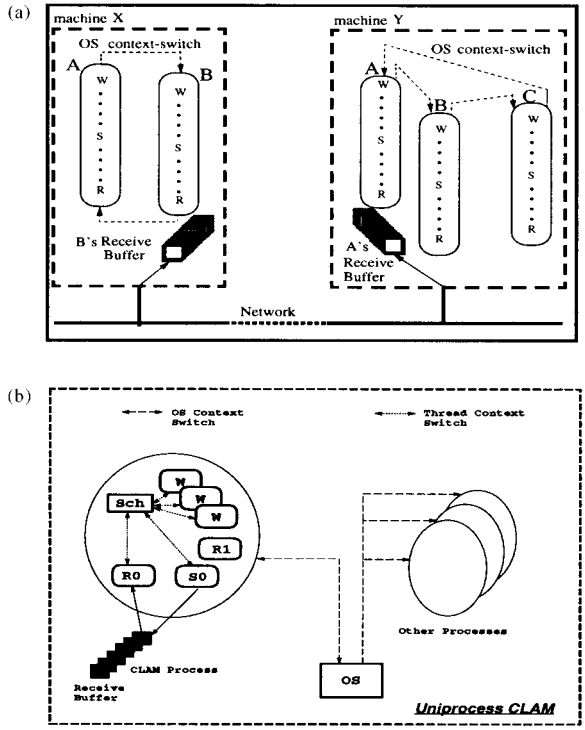


FIGURE 2 (a) Single-threaded operation in an OS. (b) Integrating threads and protocols.

in the case of multiprotocols) may be shared by packets from many senders, flow- and congestion-control are not workable; these point-to-point solution mechanisms cannot address the additive effects of multiple senders.

In extant message-passing systems (e.g., PVM, P4/Parmacs, LAM/MPI) clients typically implement *work*, message *sending* and message *receiving* tasks in a single thread of control. A client thus computes until it is necessary to send or to receive. Functions for sending, receiving, or both are invoked as and when necessary. If either *sends* or *receives* involve blocking calls to the OS, an OS context switch ensues. Once switched, a process regains control at some later time, but only after the condition for the block has been satisfied. Though it may appear pointless for a CPU to stay with a process

once a blocking call is made, the forced context switch actually penalizes the process. It prevents such a process from continuing with other useful work that is unrelated to the block, were the process to have access to such work.

Some systems (e.g., PVM) exploit multiple processes – a daemon and a compute process – to overlap communication and computation. One or more OS context switches must take place while daemons talk to local compute processes. This hampers the ability to both a daemon and a compute process to attend to their respective receive buffers. Moreover, communication between compute processes is hindered by the presence of intermediate daemons. Finally, since such systems are generally TCP/IP based, integrating services such as scalability, realtime delivery, and multipoint communication is difficult. P4 suffers from similar drawbacks and for similar reasons, though daemons are not used in communication.

To support scalability, it is best to use a single UDP-kernel buffer to support input from various end-points. As shown in [22], this scheme is highly effective in reducing the cost of message receipt from multiple sources. It may appear that packet-loss can be prevented by point-to-point flow control or congestion control algorithms. But although the flow of packets in each session may be controlled at a given time, the aggregate packet flow of all the protocol sessions cannot be controlled. Such control is particularly difficult when the number of sessions is large. Given that point-to-point flow control cannot be used to regulate packet-loss in our environment, we propose smart scheduling mechanisms as an alternate path to this goal. It may appear appropriate to increase UDP-buffer size, depending on the number of protocol sessions in effect. While this may help in alleviating packet-loss at a receiver, in our experience [21] it does little to reduce latency. Further, because it is impossible to predict the number of active sessions a protocol will simultaneously attend to, and because kernel-shape memory is a precious and limited resource, overestimation in receive-buffer size may sharply curtail the number of processes or protocols that can be within a host. Also, if instead of using UDP we are able to directly access a network interface as our best-effort delivery subsystem, buffer size will be limited not by the configuration of the system, but by the on-board physical memory included in the interface.

### *Proposed Integration of MultiProtocols and Threads*

Consider a multithread version of the standard model, as depicted in Fig. 2(b). Threads enable distinct functionalities to operate within a single address space, with low context-switching costs, compared to the relatively high context-switching costs offered by kernel-level processes. To simplify discussion, because we emphasize the efficient scheduling of a *receive* thread to minimize incoming packet-loss and latency, we assume that the *send* and *work* threads are lumped together in *W*.

An OS-level context-switch may occur during execution of any thread, placing CPU attention on other processes in the system. When control is returned to the process in question, execution resumes within the thread preempted by the OS. In general, actions taken by the OS are outside our (user) control; what is in our control is deciding just *how* our process's CPU time-slice is to be divided between its *receive* thread and other threads, given that the goal is to keep packet-loss and packet-latency low, and work-efficiency high.

A reasonable scheduling scheme entails sharing CPU attention between *send*, *receive*, and *work* threads is some equitable manner. Details of a variety of scheduling schemes can be found in [21]. For example, one simple scheme (but not the best) is to give each thread a fixed quantum of CPU attention, using a *fixed* time-slice. A running thread is preempted by the threads system scheduler when a signal is generated on time-slice expiry. This fixed-slice, preemptive approach is taken by the Conch message-passing library [43], which uses a signal-generated multiplexing scheme.

When the threads system scheduler selects a thread that is ready to run, it has no way of knowing if the thread will have work to do; a thread's work may depend on the thread's data. Thus, such a thread begins to run even though to may have work to do. If it finds it has no work to do, it will typically yield control to the threads scheduler. If this action is frequent, many unnecessary thread context-switches are generated. With too small a time-slice, a frequently scheduled *receive* thread may tend to find its receive-buffer empty. Even if it finds its buffer occupied every so often, a small time-slice will cause the *receive* thread to be preempted even while it works on removing packets from the buffer. Both factors – frequent context-switching

and small *receive* thread run times – can cause protocol performance degradation.

### *Examples: Systems Without Integrated Threads and Protocols*

Popular examples of single-threaded systems include PVM [40], P4 [4], Express [16], Zipcode [39], PARMACS [5], and Conch [43]. These support a variety of hardware platforms, ranging from HeWNs (heterogeneous workstation networks) to hardware DMMPs (distributed memory multiprocessors), which use XDR [34] to provide data homogeneity in heterogeneous environments. Although they all provide similar services, their interfaces are different. This hinders portability between implementations. All these systems contributed important ideas towards a message-passing interface (MPI) standard [33]. PVM contributed support for dynamic process creation. P4 is known for its support of shared-memory multiprocessors (SMMPs) and its monitor primitives for coordinating access to shared data. Express built high-level functions upon messaging primitives to facilitate parallel computing; it also tackled problems of parallel I/O and dynamic load balancing. Zipcode introduced the idea of communication contexts, to enable the safe implementation of parallel libraries. PARMACS evolved from P4, and one of its main contributions was the concept of virtual process topologies. Here, computational nodes are arranged in a structured manner and referenced by their location in the structure. For example, if the structure is a two-dimensional grid, processes can be referenced by their grid coordinates. Conch and Zipcode also contributed to the field of virtual topologies. PVM and P4 are perhaps the most successful of these messaging systems. P4 is currently used as the communication layer of choice for a portable version of MPI (MPICH [3]). PVM is still widely used and has served as a low level communication layer for MPI and Zipcode implementations.

Most message-passing systems for HeWNs, including MPI libraries, rely on TCP/IP. As a result, *scalability* is seriously limited. Some implementations attempted to provide partial solutions. In P4, a direct connection is established between two nodes only when one needs to transfer a message to the other. When a process runs out of connections, however, there is no connection recycling, and the

application simply fails. Connection recycling would help, but brings in the added expense of connection management: closing old unused TCP/IP connections, and opening and maintaining new ones. The number of simultaneous connections remains relatively small.

PVM and LAM–MPI use communication daemons to enhance scalability. Though both systems support a fully connected model using TCP/IP, they also support a more scalable model running on UDP-based user-level protocols. But the increase in scalability comes at a high cost: a reduction of at least 50% in throughput and a sharp increase in latency [22]. Throughput falls mainly because of process scheduling and store-and-forward overheads at daemons that lie on message paths. Further, these user-level protocols are usually incomplete (e.g., PVM–UDP). For example, they may not provide flow- and congestion-control mechanisms; the consequences can be disastrous. The lack of adequate control mechanisms makes such systems function poorly in WAN settings.

Because of a well-recognized need for improved communicability in irregular applications [18,22] systems proposing *application-level threads services* have begun to appear (e.g., TPVM [15], Nexus [18], LPVM [47], ARIADNE [30,31], PARASOL [28], Chant [23], Nexus [17], and UPVM [25]). There are many benefits to multithreading, including support for unpredictable data-access patterns, efficient and transparent masking of communication and I/O latencies, dynamic computation schedules, asynchronous operations, and fine-grained load-balancing. *It should be emphasized that the above-mentioned efforts are all geared towards concurrency enhancements at the application-level*. But, concurrent threads are forced to rely on the old communications framework (e.g., TCP/IP, or PVMs reliable-UDP) for message delivery. This leaves intact the problems of layering and poor protocol/application interactions [41] discussed earlier. Further, these proposals do not offer solution mechanisms for "collaborative" operations, i.e., multiprotocol support, multiway communication, efficient integration of realtime communication and computation etc. Although these systems provide asynchronous communication primitives to enable overlap of communication and computation, *the application programmer is forced to explicitly poll for incoming messages. This results in increased computational overheads and increased programming complexity*. Further, these systems *cannot*

offer applications the many potential benefits of multithreading that our design offers: support for unpredictable data-access patterns, efficient and transparent masking of communication and I/O latencies, dynamic computation schedules, asynchronous operations, and fine-grained load-balancing.

### 3.4. Issues in Threads Scheduling

The basic equation is: how should communication threads be scheduled? There are two possible types of solutions: schemes based on I/O signal or network interrupts, and schemes based on polling. In interrupt-based algorithms the receive action is initiated from below, i.e., from the network. In polling algorithms, the receive action is instantiated periodically from above, i.e., from the upper layers. Further, polling algorithms can be made to work either adaptively or non-adaptively. Adaptive algorithms may repeatedly adjust their polling frequency to match network input, while non-adaptive algorithms poll the network at fixed intervals, independently of network activity.

There are several ways to implement interrupt-based polling. One way is to process network input directly, within an interrupt handler. Because most threads systems are not designed to support interrupt handlers that invoke thread primitives, this may not work if input processing requires thread primitives; the complex race conditions that result may be difficult or impossible to detect and solve. Another way to process network input is to exploit I/O interrupts for scheduling a receive thread when messages arrive. With this, input can be processed using thread primitives.

Polling may be implemented in one of three ways. Polling calls may be embedded by a compiler or a preprocessor at compile-time. This usually results in inefficient code because compilers or preprocessors have little or no knowledge of communication patterns that can arise when the application runs. Similarly, explicit polling calls may be embedded by the application programmer during code development. This also results in inefficiencies, and for the same reason, in addition to the extra development work for the application programmer. We believe that it is best to implement an implicit polling

mechanism within the communication library because this relieves the application programmer of unnecessary work, and enables run-time optimizations via adaptive polling strategies.

There are several mechanisms that can potentially support an integration of threads and protocols. These include OS processes, user-level threads [30], kernel-level threads [14], scheduler-activations [1], Filaments [19], Upcalls [7], and user-level processes. A simple way to integrate user-level protocols with applications is to place protocols and applications in distinct kernel-level processes which can communicate via shared-memory or any from of IPC. Having distinct protection domains facilitates development and debugging. But because of the high cost of OS process context-switching, this is only useful for support of coarse-grained parallelism; context-switching time between kernel-level processes is typically higher than small-message round-trip latency in modern LANs.

User-level threads offer a highly efficient alternative for integrating communication and computation. Distinct protocols may run as threads within a single address-space. Naturally, this yields a big reduction in application-to-protocol interface costs; these now turn into function calls and global-memory or heap management issues. Context-switching time for user-level threads is generally two orders of magnitude smaller than context-switching times for OS processes (e.g., ARIADNE threads context-switch in 10–15 μs, while OS processes may require 500–1000 μs on typical workstations), and only two to four times the cost of a function call. Scheduling user-level threads is significantly cheaper than process scheduling, since it does not require kernel intervention or remapping virtual memory to physical addresses. User-level threads are portable and do not require modifications to the kernel.

There are, however, some limitations to user-level threads, stemming from the loose integration of processes, the OS scheduler, and the virtual memory system. In general, the kernel is oblivious to the existence of user threads and so may preempt a process hosting a high priority thread. Control may be given to a process hosting a low priority thread, resulting in undesirable behavior. Without kernel-level threads support, a process blocks whenever one of its threads blocks on I/O. For multithreaded applications, this is not a desirable feature. The use of non-blocking calls is a portable and

effective way of circumventing the latter problem, at the expense of some increase in programming complexity. Handling OS context switches on page-faults remains a problem, although [22] not a significant one. We propose solutions based on additional support from kernel-level threads. An alternative is to use Scheduler Activations [1]. Through a separate interface, the kernel is able to notify processes of events like page faults, blocked I/O, and preemption. When a process is so notified, it can immediately schedule another runnable thread, thus avoiding an OS context-switch. Also, the kernel is able to utilize user-space thread priority information in scheduling decisions. A serious disadvantage is that scheduler-activations are not widely supported in commercial operating systems, and thus conflict with our portability requirements.

One might decide to use the ARIADNE threads system because it supports a crucial feature: *preemptive time-slicing*. With this, all runnable threads receive CPU attention in a finite time interval. Preemption aids in implementation of time-sensitive asynchronous protocol action which cannot tolerate indefinite delay. Spinlocks are avoided whenever possible; they are known to perform poorly in the presence of preemptive time-slicing [2]. Finally we could further enhance the threads system with an efficient *timer-subsystem* for realtime control, using appropriate kernel-threads support.

### 3.5. Issues in Protocol Structuring and Efficiency

A protocol's performance is determined by how its implementation is modularized and integrated with a host OS [22]. Despite TCP/IP's success, its in-kernel, stream-oriented design has hampered the performance and expandability of distributed computing systems that have come to rely on it. For example, hard limits on resources within the kernel renders TCP ineffective[2] in the support of large distributed computations. In Section 3.3, we described a number of systems which follow the general trend of layering threads systems and services upon existing protocols and messaging libraries. This is a top-down

[2]Reconfiguring the OS kernel to redefine limits requires management intervention and is not a viable alternative because it effects normal machine usage, and hampers system flexibility and application portability.

use of threads for enhancing applications. In contrast, we propose a bottom-up view: *eliminate the protocol layering that separates computing threads from communicating threads, and implement protocols entirely in terms of threads*. Freed from layering constraints, the communication and computation subsystems can be made to interact optimally. If all threads are in user-space, the controllable part of the communication framework is in user-space. Apart from satisfying the important portability and flexibility requirements [12], this setup offers the potential for shared-memory multiprocessing of protocol actions [20].

A basic question is: how should protocols be structured if we would like to have multiprotocol and realtime support? There are three possibilities. One is to leave all protocols in the kernel, another is to place them all in user-space, and the last is to leave some in the kernel and put others in user-space. There are many advantages to user-space implementations [42]. New protocol design principles like Application Level Framing and Integrated Layer Processing [8] cannot be implemented efficiently in systems where protocols are not highly sensitive to application needs. For example, regular in-kernel implementations of TCP/IP do not understand boundaries imposed by the application on its data, and a TCP layer cannot deliver data received out-of-order [22].

Protocols implemented in user-space have been shown to perform at least as efficiently as in-kernal implementations; given adequate low-level hardware support, user-space implementations have even outperformed kernal-space implementations [12]. In-kernal protocols are generally unable to offer an integrated set of data manipulation functions, such as functions for rapid data presentation and data movement to and from an application's address space. Such features can be critical to the performance of applications like realtime collaboration which are highly sensitive to rapid data transfer. The availability of a common threads framework at both the application and the protocol level enables the efficient integration and optimal scheduling of communication and computation functions within a single OS-level process. User-space protocols may implemented without resorting to distinct OS-level process for computation and communication, i.e., no communication daemons are necessary. Finally, this approach provides for scalability in two senses. First,

multiple protocols transporting, for example, message-data multimedia, may be optimally scheduled – without OS context switches – within a single OS process. Each protocol is handled on a dedicated socket. Second, each socket may connect to an arbitrary number of remote sockets with minimal or no kernel involvement.[3]

### *Issues in Multiway Communication*

Our interest multicast streams directly from its use in implementing efficient collective operations (e.g., scatter, gather, broadcast), fault-tolerance algorithms, and support for "collaborative" interactions, both between solvers and between users at neighboring subdomains. We seek ways to provide efficient, portable and reliable user-level multicast as an option in CLMAM's suite of protocols.

The large scale air pollution models we target at require *efficient* service that scale to a large number of nodes (in the order of a 1000 nodes). There are basically two types of reliable multicast protocols [35], but choosing an appropriate one for an application is not simple. Sender-initiated multicasts suffer from scalability problems like ACK implosion. Receiver-initiated multicasts conserve ACK bandwidth by allowing receivers to initiate packet-retransmission based on NACKs; this can, however, lead to memory shortage at senders that do not receive timely feedback on which packets can be discarded. A combination of the two techniques appears to be a convenient alternative, but some experimentation is needed to confirm this.

Problems of congestion and flow-control have not been previously addressed for reliable multicast [6]. Experimental work is crucial in handling such problems. For example, with widely distributed receivers, timely feedback will enable a sender to determine a near-optimal sending-rate. Also, if receivers are all treated in the same manner, the sender is forced to tailor its rate to suit the slowest receiver, or a receiver with smallest buffering capability. Thus, the main issue involve congestion-and flow-control, scalability and a lack of standardization; the latter is readily apparent from the number of different proposals

[3]The degree of kernel involvement depends on how much of the protocol is implemented in use-space.

for multicast protocols, including those described in [46]. Most systems now support IP multicast [9]. But, providing reliable and scalable service is nontrivial, and IP traffic may be restricted at gateways that filter traffic or are not configured for multicast routing. Note that one might use the idea of "grouping" to split receivers in a multicast group. In this way, reliable transmission to fast receivers will not be delayed because of slower receivers.

### *Acknowledgement*

Part of this paper was performed while the author was visiting the Computer Science Department at Purdue University. He wishes to thank Prof. Vernon Rego for helping him throughout in this paper.

### *References*

[1] T. Anderson, B. Bershad, E. Lazowska and H. Leavy (2002, 1992). Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelsim. *ACM Transaction on Computer Systems*, **10**(1), 53–79, February.
[2] M. Bjorkman and P. Gunningberg (2002, 1993). Locking Effects in Multiprocessor Implementation of Protocols. In: *ACM SIGGOMM' 93*.
[3] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk and A. Skjellum (2002, 1995). *Users' Guide to MPICH, a Portable Implementation of MPI*, October.
[4] R. Butler and E. Lusk (2002, 1994). Monitors, messages and clusters: the p4 parallel programming system. *Parallel Computing*, **20**(4), 547–564, April.
[5] R. Calkin, R. Hempel, H. Hoppe and P. Wypior (2002, 1994). Portable programming with PARMACS message-passing library. *Parallel Computing*, **20**(2), 463–480, April.
[6] S. Cheung and M. Ammar (2002, 1995). Using destination set grouping to improve the performance of window-controlled multipoint connections. ( *preprint*), pp. 388–395.
[7] D. Clark (2002, 1985). The structuring of systems using upcalls. In: *10th ACM Symposium in Operating System Principles*, pp. 171–180, December.
[8] D. Clark and D. Tennenhouse (2002, 1990). Architectural considerations for a new generation of protocols. In: *ACM SIGGOMM'90*.
[9] S. Deering (2002, 1989). Host extension for IP multicasting. RFC-112, August.
[10] Bozhidar Dimitrov and Vernon Rego (2002, 1997). Arachne: a portable threads library supporting migrant threads on heterogeneous network farms. In: Dhabaleswar Panda and Craig Stunkel (Eds.), *Communication and Architectural Support for Network-Based Parallel Computing*, Volume 1199 of *Lecture Notes in Computer Science*, pp. 102–114. Springer-Verlag, February.
[11] T. Drashansky, J.R. Rice, E. Houstis, E. Vavalis, S. Weerawarana and P. Tsompanopoulou (2002, 1997). Collaborating problem solving agents for multi-physics problems. In: *Proceeding of 15th IMACS World Congress*, Vol. 4, pp. 541–546.

[12] A. Edwards and S. Muir (2002, 1995). Experiences implementing a high performance TCP in user space. In: *ACM SIGGOMM*.
[13] L. Peters *et al*. (2002, 1995). The current state and future direction of Eulerian models in simulating the tropospheric chemistry and transport of trace species: a review. *Atmospheric Environment*, **29**, 189–222.
[14] M. Powell *et al*. (2002, 1991). Sun OS multithreaded architecture. In *Proceedings of the Winter USENIX Conference*.
[15] A. Ferrari and V. Sunderam (2002, 1995). Multiparadigm distributed computing with TPVM. Technical Report CSTR-951201. Department of Math and Computer Science, Emory University.
[16] J. Flower and A. Kolawa (2002, 1994). Express is not just a message passing system: current and future directions in express. *Parallel Computing*, **20**(4): 463–480, April.
[17] I. Foster, C. Kesselman and S. Tuecke (2002, 1995). Nexus: runtime support for task-parallel programming languages. Technical Report MCS-TM-205. Argonne National Laboratory, February.
[18] I. Foster, C. Kesselman and S. Tuecke (2002, 1996). The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, **37**, 70–82, October.
[19] V.W. Freeh, D.K. Lowenthal and G.R. Andrews (2002, 1994). Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. Technical Report TR 94-11a. Department of Computer Science, The University of Arizona.
[20] J. Gomez, V. Rego and E. Mascarenas (2002, 1998). The CLAM approach to multithreaded communication on shared-memory multiprocessors: design and experiments. *IEEE Transactions in Parallel and Distributed Systems*, **9**(1), January.
[21] J. Gomez, V. Rego and V. Sunderam (2002, 1997). On tailoring thread schedules in protocol design: experimental results. Technical Report. Purdue University, West Lafayetter, IN 47907.
[22] J.-C. Gomez, V. Rego and V. Sunderam (2002, 1997). Efficient multithreaded user-space transport for network computing: design and test of the TRAP protocol. *Journal of Parallel and Distributed Computing*, **40**(1), 103–117.
[23] M. Haines, P. Mehrotra and D. Cronk (2002, 1995). Ropes: Support for collective operations among distributed threads. Technical Report 95-36. ICASE, April.
[24] E.N. Houstis, J.R. Rice, S. Weerawarana, A.C. Catlin, P. Papachiou, K.Y. Wang and M. Gaitatzes (2002, 2000). Parallel ELLPACK: a problem solving environment of PDE based applications on multicomputer platforms. *ACM Trans. Math. Software*, **24**(1), 30–73.
[25] R. Konuru, S. Otto and J. Walpole (2002, 1997). A migratable user-level process package for PVM. *Journal of Parallel and Distributed Computing*, **40**(1), 81–102.
[26] E. Mascarenhas (2002, 1996). A system for multithreaded parallel simulation and computation with migrant threads and objects. PhD Thesis. Department of Computer Sciences, Purdue University, August.
[27] E. Mascarenhas, F. Knop, R. Pasquini and V. Rego (2002, 1997). Adaptive state saving in parasol. In: *Proceedings of the 1997 Winter Simulation Conference*.
[28] E. Mascarenhas, F. Knop and V. Rego (2002, 1995). PARASOL: a multi-threaded system for parallel simulation based on mobile threads. In: *Proceedings of the Winter Simulation Conference*, pp. 690–697.
[29] E. Mascarenhas and V. Rego (2002, 1995). An architecture for visualization and user interaction in parallel environments. *Computer and Graphics*, **19**(5), 739–753.
[30] E. Mascarenhas and V. Rego (2002, 1996). ARIADNE: architecture of a portable threads system supporting thread migration. *Software-Practice and Experience*, **26**(3), 327–357, March.
[31] E. Mascarenhas and V. Rego (2002, 1998). Migrant threads on workstation farms: parallel programming with ARIADNE. *Concurrency-Practice and Experience*, **10**(9), 673–698.

[32] E. Mascarenhas, V. Rao and J. Sang (2002, 1995). Display: a system for visual interaction in distributed environments. In: *Proceedings of the Winter Simulation Conference*, pp. 698–705.
[33] Message Passing Interface Forum. *MPI*: *A Message-Passing Interface Standard*, June 1995.
[34] SUN Microsystems. XDR: External Data Representation Standard. RFC-1014, June 1995.
[35] S. Pingali, D. Twosley and J. Kurose (2002, 1994). A comparison of sneder-initiated and receiver-initiated reliable multicast protocols. In: *ACM Sigmetrics on Measurement and Modeling of Computer Systems*, pp. 221–230.
[36] J.R. Rice, P. Tsompanopoulou and E. Vavalis (2002, 1999). Interface relaxation methods for elliptic differential equations. *Applied Numerical Method*, **32**, 219–245.
[37] J.R. Rice and E. Vavalis (2002, 1998). Collaborative agents for modeling air pollution. *Systems Analysis Modeling Simulation*, **32**, 93–101.
[38] J.R. Rice, E. Vavalis and D. Yang (2002, 1998). Analysis of a non-overlapping domain decomposition method for elliptic PDEs. *J. Comput. Appl. Math.*, **87**, 11–19.
[39] A. Skjellum, S. Smith, N. Doss, A. Leung and M. Morari (2002, 1994). The design and evolution of zipcode. *Parallel Computing*, **20**(4), 565–596, April.
[40] V. Sunderam, G. Geist, J. Dongarra and R. Manchek (2002, 1994). The PVM concuurent computing system: evolution, experiences, and trends. *Parallel Computing*, **20**(4), 531–545, April.
[41] D. Tennenhouse (2002, 1996). Layered multiplexing considered harmful. In: *1st international Workshop on High-Speed Networks*, pp. 143–148.
[42] C. Thekkath, T. Nguyen, E. Moy and E. Lazoswka (2002, 1993). implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, **1**(15), October.
[43] B. Topol (2002, 1993). The CONCH system. Technical Report. Emory University, Atlanta, GA. 30322.
[44] P. Tsompanopoulou (2002, 2000). Collaborative PDEs: theory and practice. PhD Thesis. Mathematics Department, University of Crete, Greece.
[45] E. Vavalis (2002, 1999). A collaborating framework for air pollution simulations. In: *NATO-ASI series*, pp. 349–358.
[46] B. Whetten, T. Montgomery and S. Kaplan. A high performance totally ordered multicast protocol. *Theory and Practice in Distributed Systems*. Springer-Verlag.
[47] H. Zhou and Al Geist (2002, 1995). LPVM: a step towards multithread PVM. Technical report. Oak Ridge National Laboratory.

## APPENDIX

Next we include a brief description (with the basic references) of the set of software toolkits mentioned so far in this paper. Some of these tools are already in daily use, others are not fully developed yet. They all provide the necessary basic blocks for building an effective distributed air pollution model.

ELLPACK [24] is a problem solving and development environment for PDE based applications. It is well over a million lines of C and Fortran code and it was implemented using the ELLPACK language

and sequential solver libraries as a foundation, and then introducing finite element methods, foreign system solvers, parallel execution, and a graphical user interface for problem specification and solution. The ELLPACK system employs several parallel reuse methodologies based on the decomposition of discrete geometric data to map sparse PDE computations to parallel machines. An instance of the system is available as a Web server for public use at http://pellpack.cs.purdue.edu.

SCIAGENTS [44] is an agent-based approach to building Multidisciplinary Problem Solving Environments which is natural parallel and highly scalable; and is especially suited for a distributed high performance computing environments. It is based on the interface relaxation methodology [11,36,38]. A SCIAGENTS prototype implementation has been created which uses the ELLPACK system as its solver agents. Mediator agents have been created for a class of PDE problems as well. The system uses the KQML querying language, makes extensive use of multithreading, and at present runs on Sun/Solaris machines. A newer version has replaced the command line oriented input of the earlier version with a GUI which has been developed in Tcl/tk and the KQML support with an integrated Agent system.

ARIADNE [30] is a threads system that offers scheduler customizability, highly efficient timers, time-slicing, and thread migration. It is known [30] to be as efficient as other threads systems (including P-Threads), in addition to offering more functionality (e.g., migration, accurate internal timers, scheduler customizability). ARACHNE [10] is a sister threads system that supports migration between heterogeneous computing platforms. It is been shown to be more efficient that P-threads. The Ariadne system [30] consists of three layers: the bottom layer contains the kernel, the middle layer provides threads support, and the top layer provides customization support. The kernel layer facilitates thread creation, initialization, destruction, and context-switching; it uses an internal priority-queue based scheduler. The support layer enables applications to use threads via supervised kernel access and offers customization support, i.e., for shared-memory access, thread migration, distributed computations and specialized schedulers. The customization layer provides independent modules that aid in customization, e.g., schedulers for sequential and

parallel simulation [28]. The ARACHNE [10] threads system is a sister threads system that, in addition to ARIADNE's functionality, also supports efficient thread migration between *heterogeneous* platforms. As shown in [30], ARIADNE is as efficient as other threads systems, including POSIX threads, and offers significantly more functionality (e.g., migration, accurate internal timers, scheduler customizability), and ARACHNE is even more efficient than POSIX threads [10].

CLAM is a connectionless, light weight and multiway communication architecture [20–22] designed to address the requirements listed above and, in particular, to support scalable, high-performance and collaborative applications that manage multimodal data. Experimental design of CLAM components, and their relationship to the other systems mentioned here will form the *major part* of the proposed work. Instead of a heavy cumbersome functionality, CLAM offers a plug-and-play methodology with its protocol suite [20]. It is layered on top of the ARIADNE threads library [30] and the UDP protocol, but it can be ported to any "best-effort" communication system. It consists of a software layer that offers runtime support for global process management, and three native protocol modules: an unreliable module which provides efficient unreliable uni-and multicast, a module which provides reliable multicast, and a reliable module which is described below. Each protocol module is implemented with a specific set of communicating threads, depending on the functionality required. For example, the reliable module requires three threads (i.e., *a receive, a send* and *a timer* thread); the unreliable module requires only a *receive* thread.

PARASOl [26–28] is a parallel simulation system based on the process-interaction (i.e., migrant threads) paradigm, and uses optimistic and speculative [26,27] synchronization protocols. Instead of using timestamped message for communication and synchronization between "discrete-event simulation solvers" as done in other systems, PARASOL exploits a highly effective *transparent* thread migration facility to obtain a more powerful programming interface at the application level. Indeed, this transparency leads to greatly simplified model development for many simulation problems, and was an important design consideration [26]. PARASOL presents a programming environment that offers migratable threads – dynamic, computational units with some private data – and a set of *global objects*. Both transactions and objects are distributed among the physical

processors hosting a simulation. Threads, which are dynamically created and destroyed, usually spend their time either performing local computations or accessing objects. To access an object located at a remote process, a transaction *migrates* to the process where the object is located, thus enhancing locality.