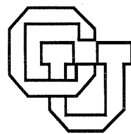
Threaded Runtime Support for the Execution of Fine Grain Code on Coarse Grain Multiprocessors

Richard Neves
Robert B. Schnabel

CU-CS-794-95



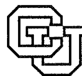
University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Threaded Runtime Support for the Execution of Fine Grain Code on Coarse Grain Multiprocessors

Richard Neves and Robert B. Schnabel
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{neves,bobby}@cs.colorado.edu)

CU-CS-794-95 September 1995



University of Colorado at Boulder

Technical Report CU-CS-794-95
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Copyright © 1996 by
Richard Neves and Robert B. Schnabel
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{neves,bobby}@cs.colorado.edu)

Threaded Runtime Support for the Execution of Fine Grain Code on Coarse Grain Multiprocessors

Richard Neves and Robert B. Schnabel
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:{neves,bobby}@cs.colorado.edu)

September 1995

Abstract

The goal of this research is to provide systems support that allows fine grain, data parallel code to execute efficiently on much coarser grain multiprocessors. The task of writing parallel applications is simplified by allowing the programmer to assume a number of processors convenient to the algorithm being implemented. This paper describes and evaluates a runtime approach that efficiently manages thousands of virtual processors per actual processor. Tight integration and specialization of scheduling, communication, and context switching is used to significantly reduce the overhead of running the fine grain parallel code. A prototype of this runtime approach is evaluated by comparing implementations of three problems, a smoothing kernel of a thin-layer Navier Stokes code, a five point stencil problem, and a block bordered system of linear equations on an Intel Paragon multiprocessor and on a network of DEC Alpha workstations. The additional cost relative to an efficient manually contracted code can be as low as 15% for granularities of 50 floating point operations per virtual processor and is typically 5-20% for granularities of about 100 floating point operations per virtual processor. The overhead is analyzed in detail to show the costs of scheduling, communication, context switching, reduced memory performance, and insuring data consistency. The implementation and analysis indicate that fine grain code can be efficiently executed on a coarse grain multiprocessor using very lightweight, specialized threads.

1 Introduction

1.1 Goals

The goal of this research is to efficiently execute fine grain, massively parallel single-processor multiple-data (SPMD) code on a much smaller number of actual processors. The ability to assume a number of virtual processors that is much greater than that actually available can greatly simplify the task of writing data parallel programs. Accomplishing this efficiently requires that the execution overhead and memory overhead for the virtual processors be as small as possible. The threaded runtime system evaluated in this paper aims to allow virtual machine sizes in the thousands of processors to be executed on actual machine

sizes in the tens of processors, with execution overhead of as small as 10% to 20% on kernels that execute as little as fifty floating point operations between communication points. The threaded runtime system also aims to limit the memory overhead to around 500 kilobytes for every thousand virtual processors (or approximately 512 bytes per virtual processor). The runtime system uses scheduling, communication, and context switching techniques that are tightly integrated, and are specialized to the task of managing virtual processors, to attempt to achieve these goals. This paper examines the limits of the threaded runtime approach, demonstrating that the above goals are achievable.

1.2 Assumptions

1.2.1 Communication hardware and software

A distributed memory, MIMD multiprocessor or network of workstations is assumed in this research. It is also assumed that the underlying communication library has a non-blocking and non-buffering receive interface that allows message completion to be tested. Such a receive interface is used exclusively in the runtime approach described in this paper. This assumption is met by most vendor supplied communication interfaces.

The communication interfaces supplied in the NX/2 [15] and MPI [7] communication libraries are compatible with our design assumptions. Both NX and MPI are used in the research described in this paper, in the implementations on an Intel Paragon and cluster of DEC Alphas, respectively. Both NX and MPI support the ability to check message completion asynchronously. Checking message completion requires periodic checking of individual message handles. Checking for message completion in the presence of scheduling is discussed in detail in Section 3.2.1.

1.2.2 Programming model

The research assumes a data parallel (SPMD) programming model that uses send/receive semantics and features similar to those found in Intel's NX/2 or in the MPI standard. It is assumed that the model is extended to allow the mapping of data to virtual processors and virtual processors to physical processors, and that the mapping of virtual processors to actual processor is fixed. Lastly, it is assumed that limited definition/use information for communicated data is available (see Section 3.2.2).

1.3 Motivation

Allowing the programmer to assume a "virtual" machine size simplifies the task of writing data parallel send/receive programs. The programmer is no longer responsible for how many actual processors are available. Thus, the programmer can focus on implementing the algorithm on a convenient number of virtual processors. In many scientific applications, the natural number of virtual processors is very large.

For example, the virtual processors may correspond to the points in a three dimensional grid which can easily be in the thousands or millions. Mapping a problem to a fixed number of processors requires more logic than assuming a number of virtual processors convenient to the problem. The extra logic includes the use of loops to iterate over parts of the data structure owned by the processor. Abstracting the number of processors also allows the same source code to be compiled and executed on different sized multiprocessors.

1.4 Organization of paper

The remainder of this paper is organized as follows. Section 2 describes previous work related to the runtime approach evaluated in this paper. Section 3 describes interesting new aspects of our runtime approach that are necessary to efficiently manage large numbers of virtual processors. Section 4 discusses the experiments used in evaluating our new runtime approach. Section 5 analyzes the results of these experiments and breaks down the runtime overheads. Section 6 summarizes the effectiveness of using a runtime approach to execute fine grain SPMD send/receive code and describes future work in improving this approach.

2 Related work

To our knowledge, the execution of fine grain parallel send/receive code on coarser grain multiprocessors has not received much study. The problem has been studied for pure SIMD paradigms [11]. For the general send/receive paradigm, the closest work is in the area of message passing thread libraries. These packages are not appropriate for our objectives, however, because they do not attempt to retain send / receive semantics at the thread level. That is, they do not support efficient same-processor thread communication, and they are not intended to manage thousands of communicating threads. Sections 2.1–2.2 discuss user level thread libraries, distinguishing between single address space and separate address space thread libraries, and contrasts them to our needs.

Related work is also found in two other research areas: interprocess communication (IPC) and remote procedure call (RPC) mechanisms. Kernel IPC must maintain consistency of data when message passing in a shared memory address space. This is addressed in [17] by keeping the sender from writing to the sent data, which resembles our approach. Similar work is found in the scheduling mechanism used for lightweight RPC in [3]. This scheduling mechanism was derived from shared memory IPC in [5]. Utilizing threads, this work uses a communication style scheduling similar to our own. Sections 2.3 and 2.4 briefly discuss this work.

2.1 Single address space, user-level thread libraries

Single address space, user-level thread libraries such as C threads [4], POSIX threads (Pthreads) [12], Presto [2], or Awesime [8] offer flexible interfaces to managing threads in a single address space. Except for

Cthreads, these thread packages use a separate scheduler thread that requires an additional context switch for every scheduling decision. Except for Cthreads, these thread packages use a separate scheduler thread that requires an additional context switch for every scheduling decision. This scheduler design supports the provision of alternative scheduling schemes. All of these packages allow runtime specification of each thread's stack. The flexibility of allowing scheduling schemes, stack sizes, and other options to be configured at runtime by the application results in runtime overhead.

Our approach requires flexibility in scheduling and stack management while avoiding extra runtime overhead. The scheduling and stack policies used in our approach require access and modification of information normally maintained within the thread library itself, such as thread state and position in scheduling data structures. As we will discuss in Section 3, our system requires single context switch scheduling decisions (i.e. no scheduler thread) and the ability to install two-tiered scheduling that we refer to as "communication-based" (see section 3.1.2). Our system also requires that the stack for any given thread be changed "on the fly" in order to reduce stack usage (see section 3.3). Off-processor communication between threads necessitates scheduler involvement that cannot be described to "configurable" thread libraries. The above user-level libraries cannot satisfy these requirements without adding a runtime layer and incurring a great deal of overhead. Thus, we have implemented our own thread kernel specialized to the needs of efficiently managing thousands of communicating virtual processors.

2.2 Separate address space, user-level thread libraries

Separate address space, user-level thread packages such as NewThreads [6] and Chant [10] are similar to this research in that they allow threads of execution to communicate across memory spaces. Both packages allow computation to continue when a thread blocks on a receive by scheduling another thread. The system described here is more general in that it allows threads (or virtual processors) to communicate on the same or different processors. This is handled transparently by the runtime system. In general, both NewThreads and Chant's threads are not intended to handle thousands of threads communicating both within and across processors.

2.2.1 NewThreads

NewThreads allows threads to communicate through "ports" on the same or different processors. Supporting communication between threads in the same address space using ports requires extra buffering for communication between threads on the same or different processors. This is because ports specify a temporary memory location for the received message instead of the memory location that will the message data will ultimately reside in during computation. The NewThreads package also attempts to receive messages at every scheduling decision. This is accomplished in the Paragon implementation of NewThreads using the "probe" interface to check if a particular message can be received. In the PVM implementation of

NewThreads, the non-blocking receive interface is used. In both cases, an extra copy is necessary to move the data from the communication layer into the location where the application expects the message to reside. For the purposes of tolerating latency for very few, coarse grain threads, this design works well. The fine granularity of this research requires much more efficient non-blocking communication in order to avoid unnecessary buffering, and very infrequent polling of the communication layer.

2.2.2 Chant

Chant allows threads to communicate directly with each other, provided the threads are on different processors. Chant specifies a communication interface that extends the POSIX thread standard and utilizes the POSIX thread library. In utilizing a user-level thread library (Pthreads), Chant inherits the limitations described in Section 2.1. Unlike NewThreads, however, Chant uses "immediate receives", thus removing unnecessary memory copies when receiving off-processor messages.

The polling strategy used in Chant differs from our own in that it is better suited for a handful of coarse grain threads. The Chant authors have experimented with three different polling techniques: thread polls, scheduler polls with waiting queue, and scheduler polls with partial switch. The first technique increases the number of context switches because threads blocked on receives are still considered runable by the scheduler. This allows threads blocked on receives to poll. The second technique requires enumerating through a linked list of all posted messages at every scheduling decision. This technique adds significant scheduling overhead. The third technique is the most efficient of the three, but still incurs scheduler overhead since the scheduler must linearly search for a ready thread at every scheduling decision. The third technique also requires polling the communication layer every time a blocked thread is encountered during the search for a ready thread. Scheduler overhead and additional context switches aside, each technique requires frequent polling, in some cases at every context switch. Frequent polling is not an issue in coarse grain applications, but is very significant in fine grain threaded applications where efficiency is achieved by polling only when every thread is blocked (see Section 3.2.1).

2.3 Shared memory message passing

To allow inexpensive same processor thread communication, our system supports message passing in a shared address space without copying messages. Essential to the approach described in Section 3.2.2 is the notion that the sender of a message should not be capable of changing it until the receiver is finished. An operating system called MUSS [17] utilized a technique similar to this in managing IPC. In MUSS, the sender cannot access the sent data again, eliminating any potential inconsistencies. In our system the sender cannot access the sent data for a bounded period of time. This period is bounded by how long the receiver uses the data. This is determined using limited definition/use information.

2.4 Communication-based scheduling

Scheduling threads based on communication partners is useful in the context of lightweight RPC and shared memory IPC. In [5], a technique called "fast path" is used to directly schedule a thread when the receiving thread is already waiting on it. The author shows how scheduling overhead is reduced by removing queueing operations and scheduling latency. This technique is specialized for lightweight RPC in [3]. The specialization includes always guaranteeing a server thread is available to schedule. Our use of communication-based scheduling specializes and extends the above techniques. Our runtime system bases scheduling decisions not solely on send partners, but on receive partners as well. We have found that this extension in combination with FIFO scheduling in the context of fine grain multithreading also improves cache locality in addition to reducing scheduling overhead. This is discussed in more detail in Section 3.1.

3 A fine grain runtime system

This section describes the key aspects of our threaded runtime system that allows it to manage large numbers of virtual processors with minimal overhead. A fine grain runtime system must assume several responsibilities that are potential causes of unacceptable overhead. The system must schedule the virtual processors on each actual processor, support general send/receive communication among virtual processors within and across address spaces, insure that communicated data is consistent among virtual processors within the same address space, and switch between virtual processor contexts. Managing these responsibilities with very little overhead becomes increasingly challenging as the granularity becomes finer. This section describes our approach to these issues, as well as the overall system architecture. Our approach is based upon the use of very carefully managed user level threads, one per virtual processor.

3.1 Scheduling virtual processors

The scheduler is responsible for scheduling a fixed set of virtual processors (VP) on a given actual processor. Efficiently scheduling a very large number of fine grain VPs requires aggressive scheduling performance. This section first describes the key requirements for efficient scheduling in the presence of frequent VP communication and blocking. These requirements cannot all be met by relying purely on FIFO scheduling. The techniques we have found successful, communication-based scheduling and tight integration with the communication layer, are then described.

3.1.1 Scheduling requirements

In order for scheduling to contribute very little overhead to the execution time of the program, the following requirements must be met:

1. The scheduling of a VP must require only one context switch. Because the system will have large numbers of threads (VPs) that context switch frequently, an additional context switch for each scheduling decision must be avoided.

2. The scheduling algorithm must favor VPs that compute on data neighboring or shared with the blocking VP. This is important because it often approximates a hand-scheduled order, thus avoiding scheduling orders that result in frequent blocking of VPs and increased context switches. It is also important because it improves data locality.

3. The scheduling algorithm must avoid linear searches for VPs that are ready due to newly arrived messages. This implies that the scheduler must interact directly with the underlying communication library.

3.1.2 Communication-based scheduling

The first two requirements given above, one context switch per scheduling decision and scheduling that favors communication partners, are achieved by utilizing a combination of communication-based and traditional "first in, first out" (FIFO) scheduling policies. This approach avoids the use of a separate thread for scheduling. Heuristics that schedule based on communication patterns were introduced by Neves and Schnabel in [13], and were shown to reduce the total number of context switches and improve application data locality.

Briefly, our scheduling mechanism works by first attempting to switch to the last VP data was received from. If this VP is blocked, the scheduler next attempts to schedule the VP data was last sent to. In a majority of cases, the next VP to resume execution can be chosen using this communication-based method.

It is often the case, however, that no communication partner is in a ready state when a particular VP becomes blocked. This is also considered to be the case if the communication partners of a blocked VP are not on the same processor. In these cases, the scheduler chooses the next VP from a ready list of VPs that are scheduled in a FIFO order. In the worst case, the scheduler must perform two conditionals and a ready list manipulation to schedule the next VP. An additional context switch is never necessary, however, since the scheduling routine uses the stack of the blocked thread instead of a separate thread dedicated for scheduling. Thus, this combination of communication-based and ready FIFO scheduling policies works well in meeting the first two requirements of Section 3.1.1.

3.1.3 Scheduling and communication

The third requirement given in Section 3.1.1 is to avoid frequent and linear searches for VPs that are ready due to newly arrived off-processor messages. This requirement is achieved by infrequently testing for the completion of off-processor receives attempted by VPs. This is referred to as polling. When a VP attempts

to receive an off-processor message, a handle is associated with this receive message request. This request handle is stored in a FIFO that is used solely for receive message request handles. This FIFO is referred to as the polling FIFO since it is used to poll for off-processor messages. The polling FIFO is only searched for newly arrived off-processors messages when all other VPs are blocked, resulting in infrequent polling. When a blocked VP corresponding to the newly arrived message is located, it is immediately scheduled. This method meets the third requirement by avoiding linear searches of the complete VP list for ready VPs.

3.2 Communication between virtual processors

In our experience, communication between virtual processors accounts for the majority of the overhead in a runtime approach to supporting fine grain parallelism. The key techniques that we use to reduce this overhead are efficiently polling for off-processor messages using "immediate receives", and avoiding expensive buffering when communicating between VPs on the same processor. Failing to do either of these results in a notably inefficient fine grain runtime system. The next two subsections describe these aspects of our runtime system.

3.2.1 Polling for off processor messages

To make off-processor communication efficient, our runtime system uses a non-blocking receive mechanism that returns immediately, effectively posting a receive request by specifying the memory location to be used when a message actually arrives. When this form of communication is used, the completion of the receive request must be checked periodically. This is referred to as polling. This approach has two important advantages: it eliminates unnecessary buffering and it allows the receipt of data to occur asynchronously to other computation.

In the context of MPI and NX/2, polling is accomplished using the "immediate receive" primitives available in these libraries. When an immediate receive is called, it returns a handle for the message request. Note that the receive returns immediately regardless if the message has actually arrived or not. The arrival of the message must be tested by calling another routine provided by the communication library using the message request handle as an argument. Thus, polling is the process of asking the communication library if the message associated with a message request handle has arrived.

The key for this approach to be efficient is that polling be infrequent. Our runtime system reduces the cost of polling in several ways. Polling only occurs when all VPs are blocked. In such cases, message requests are immediately located and tested by traversing a list of outstanding message requests in a FIFO ordering. A FIFO ordering was observed to have superior performance because the oldest outstanding receive request tends to be the most likely request to be satisfied. When it is determined that a message request has completed, the VP associated with the request is immediately switched to without testing other message requests or incurring additional scheduling overhead. Each of these polling techniques is necessary

to obtain the required efficiency. Polling for any or all requests at every context switch is much too costly in a runtime system that must support a large number of threads. It is also too costly to search a table of all thread descriptors for outstanding message requests.

3.2.2 Same-processor communication

An important feature of the assumed programming model is that VPs sharing the same address space can exchange messages. For this to be efficient, these communications must be implemented differently than off-processor communications. For example, copying messages into separately allocated buffers between VPs is too inefficient in memory usage and computation. The VP runtime system avoids these problems by treating intra-processor communications as synchronization between virtual processors. Each VP thread has two data structures that aid in synchronizing the execution of VPs based on communication: the message table and the data table. These data structures are embedded in the thread descriptor associated with each VP. They are used in the bookkeeping necessary to manage communication between VPs on the same processor (in the case of the message table) and the consistency of communicated data (in the case of the data table). This approach is similar to I-Structures [1].

Treating communication between VPs in the same address space purely as synchronizations does present a consistency problem that we now address. Consistency of communicated data among VPs on the same processor is violated when a VP sends data and then immediately modifies that data. The receiving VP's view of that data is no longer correct. This problem is overcome by requiring the sending and receiving VPs to provide minimal definition/use information to the runtime system. The sending thread indicates when it first modifies "sent" data. The receiving thread indicates when it last uses "received" data. If necessary, the sending thread blocks at the modification point until all receivers of the data being sent have indicated that they have used or read the data for the last time. While this approach could cause deadlock in contrived examples, this is not the case in scientific applications, or any application where data is used in the same order as it is sent.

Scheduling VPs using this definition/use information involves adding and removing VPs from the ready list based on usage counters associated with the communicated data. These usage counters are semaphores. Surprisingly, this aspect of the runtime system never accounts for more than 13% of the total overhead (see Section 5) in any example. The definition/use information required by this approach can be inferred at compile time without burdening the programmer. Nonetheless, we have found that having the programmer provide such information to the runtime system is a relatively simple task that is easily made up for in the ability to program a virtual machine. We are considering modifying the runtime system to allow the programmer to incrementally add dependence information and incrementally achieve performance improvement. At one extreme the programmer could provide no dependence information. The runtime system could create duplicate copies of sent data and perform perform garbage collection. This would have a significant runtime and memory cost.

3.3 Context Switch Optimization

User-level threads have inexpensive context switch cost relative to kernel threads or processes. The dominant cost in switching context between two user-level threads is the cost of saving and restoring register state. This section briefly summarizes a whole program, link time optimization approach utilized in this research that reduces context switch cost by recognizing that not all registers must be saved and restored at context switches. This optimization is described in more detail in [9].

Registers that are not live at a potential context switch need not be saved and later restored. Using information available at link time, the analysis necessary to identify live registers at every potential context switch can be carried out. The steps that we use to accomplish this include determining intraprocedural register liveness, identifying potential context switches, finding indirect function calls, building a call tree, patching indirect calls, determining interprocedural register liveness based on saved registers of the called procedure, determining all call chains that include a potential context switch, and finally generating the minimal register saving and restoring code for each potential context switch. Using the above process, the saved register state is reduced from 14 general purpose registers to 7 general purpose registers for each of the three test applications described in Section 4.

The benefit of this optimization is particularly significant in applications with a large number of frequently context switching threads, such as the application domain described in this paper. Other domains with this characteristic include operating system kernels or threaded databases. As an example, in the virtual processor version of the smoothing kernel of a thin-layer Navier-Stokes algorithm, context switch cost is reduced fifty percent by this optimization with overall overhead reduced up to twenty percent. All DEC Alpha results in this paper have the benefit of this optimization. It is equally applicable to the Intel Paragon implementation, but was not performed because the tools necessary to modify a program at link time were not available.

3.4 Architecture of the runtime system

This section gives a higher level view of various components of the runtime system architecture. Scheduling, communication, and enforcing data consistency require several mutually dependent data structures. Most information is kept in each VP's thread descriptor. This includes detailed information to manage communication between same-processor threads, to facilitate communication-based scheduling, and to maintain data consistency. All thread descriptors are indexed by thread identifiers in the thread descriptor table. Entries in the polling FIFO and ready FIFO point to entries in the thread descriptor table. Each scheduling decision is triggered using information from one of these three sources (thread descriptor table, ready FIFO, and polling FIFO).

Figure 1 illustrates how scheduling decisions are made using these data structures. Scheduling is communication-based. After a thread blocks, the scheduler first attempts to switch to a communication partner stored in its thread descriptor. This occurs for a majority of scheduling decisions. When communication-

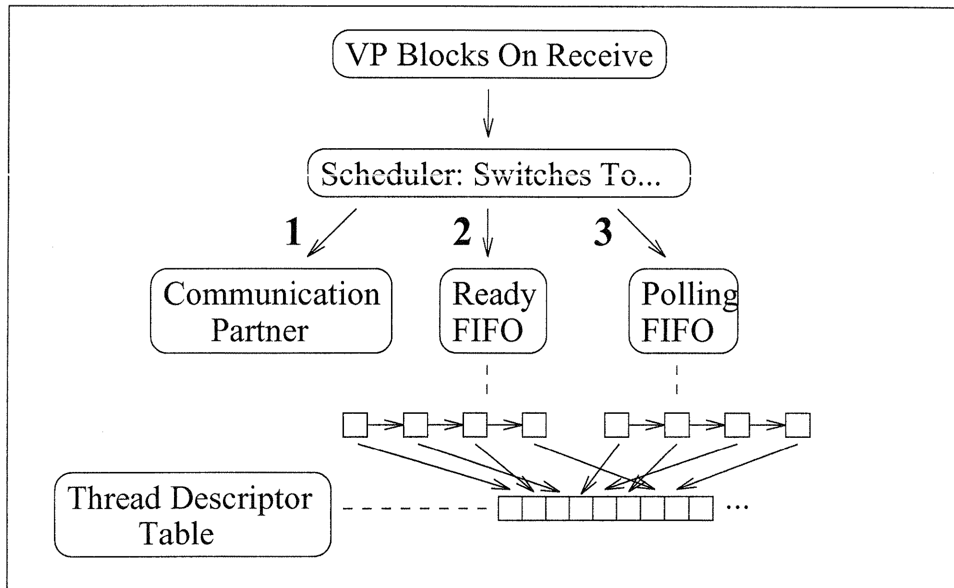


Figure 1: Scheduling flow chart and data structures for virtual processor threads and .

based scheduling fails, the ready FIFO is used to schedule a ready thread. Polling occurs only as a last resort. When polling is necessary, the scheduler searches the polling FIFO until a thread is found that is no longer blocked due to a newly arrived message. This thread is then scheduled. Thus, scheduling relies on the ready FIFO, the polling FIFO, and information in the thread descriptor table.

Figure 1 also illustrates the interdependence of these data structures. Threads blocked on off-processor communication are found in the polling FIFO. Threads ready to execute again after being blocked can be found in the ready FIFO. Both of these FIFO lists point to the thread descriptors in the thread descriptor

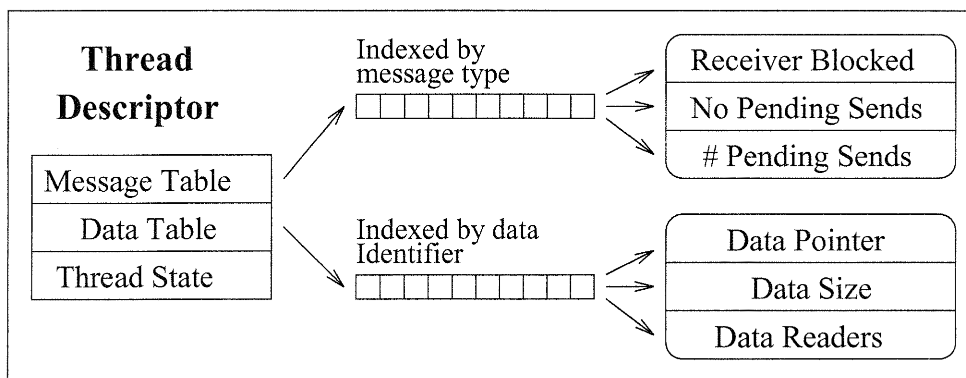


Figure 2: An entry in the thread descriptor table. Each entry stores information necessary for intraprocessor communication, scheduling and maintaining data consistency.

table and do not actually contain the thread descriptors. Each thread descriptor contains the remaining information about each thread.

Figure 2 shows the data stored in each thread descriptor. This consists of a message table, data table, and thread state. The message table is used in managing communication between same-processor threads and in performing communication-based scheduling. For example, the message table is used to determine if the receiver of a message recently sent has already blocked trying to receive this message. If so, the thread should be set to a ready state. In managing communication-based scheduling, the scheduler determines the communication partner for a blocked thread by referencing the message table of the thread descriptor for the blocked thread. Enforcing data consistency also relies on information in the thread descriptor table. Specifically, the data table is used to block threads that may violate data consistency were they to continue execution. This is accomplished by assigning a data identifier to each piece of communicated data. This data identifier is used to index the owner thread descriptor's data table and determine if there are still reader threads referencing the data.

In summary, the mechanisms described in this section work together to promote communication-based scheduling, tight integration between message polling and scheduling, only one context switch per scheduling decision, infrequent polling, consistent data, no memory copying for internal communication, very little memory per VP, and efficient use of data dependence information.

4 Experiments

Three applications have been used so far in evaluating the runtime system. They are a smoothing algorithm on a three dimensional grid, a Jacobi-like algorithm on a two dimensional grid, and a linear equations solver for block bordered matrices.

The first application implements the smoothing kernel of a thin-layer Navier-Stokes algorithm using a parallel algorithm from [14] which itself is an adaptation of the sequential TLNS3D Navier-Stokes code developed by V. Vatsa and B. Wedan at NASA Langley Research Center [18]. The algorithm performs forward and backward tridiagonal solves in three spatial directions (i,j,k). The virtual processors correspond to a 2-D grid in the (j,k) dimensions. Each virtual processor computes over all the i values for one (j,k) value. Virtual processors are mapped to actual processors by blocks of columns, i.e. all values of j (and i) for one or more values of k are mapped to an actual processor. For each stage, computation in the i and j directions can be done naturally in parallel, with no communication, while computation in the k direction involves extensive communication and must use pipelining to achieve parallelism. The implementation of the algorithm allows the granularity of computation between communication points in each VP to be adjusted by changing the grid size in the i direction.

The second application is a Jacobi-like algorithm that iterates over a 2-D grid using a five point stencil that computes the average of its neighbors. Each grid point is mapped to a virtual processor. Virtual processors

$$\begin{pmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_Q & B_Q \\ C_1 & C_2 & \cdots & C_Q & P \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_Q \\ x_{Q+1} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_Q \\ f_{Q+1} \end{pmatrix}$$

Figure 3: Block bordered linear system of equations.

are mapped to actual processors in blocks of rows and columns. Each virtual processor repeatedly receives data from the neighboring virtual processors, computes the average of its neighbors, and sends the result to each of its neighbors. Thus there is easy, full parallelism available but extensive communication. As in the first application, the amount of data at each grid point can be adjusted, thereby increasing or decreasing the granularity of computation between communication points in each VP.

The third application solves a linear block bordered system of equations of the form shown in Figure 3 utilizing the structure of the system. The approach used is similar to that derived in [19]. Each A_i, B_i, C_i, x_i, and f_i, $i = 1, ..., Q$, is assigned to one of the first Q virtual processors. The $(Q + 1)$st processor is assigned P, K_{Q+1}, and f_{Q+1}. Each of the first Q virtual processors solves the sets of linear systems $A_i v_i = f_i$ and $A_i W_i = B_i$ and computes terms $C_i v_i$ and $C_i W_i$. The $(Q + 1)$st processor receives the summations of the terms $C_i v_i$ and $C_i W_i$, solves the linear system $(P + \sum_{i=1}^{Q} C_i W_i) x_{Q+1} = f_{Q+1} - \sum_{i=1}^{Q} C_i v_i$, and broadcasts the result to the first Q virtual processors. The first Q virtual processors then compute $x_i = v_i - W_i x_{Q+1}$. The amount of computation performed by each virtual processor can be adjusted by changing the size of the blocks in the matrix, i.e. M and N where each A_i is $N \times N$, B_i is $N \times M$ and C_i is $M \times N$.

In addition to a virtual processor version of each application that is implemented as described above, a version that aggregates virtual processors manually and uses one thread of execution per actual processor also was implemented. This "manually contracted" implementation represents parallel code that would be produced by a send/receive parallel programmer. The manually contracted version was then compared with a virtual processor version that utilizes our runtime system. The results are presented and discussed briefly here, and analyzed carefully in Section 5.

Tables 1–3 compare the performance of the virtual processor and manually contracted versions of all three applications on an Intel Paragon distributed memory MIMD multiprocessor and one or more DEC Alpha SMP workstations. The DEC Alpha results in Table 1 are from a single DEC Alpha SMP consisting of two processors. The DEC Alpha results in tables 2 and 3 are from two two-processor DEC Alpha SMPs networked together. The latter configuration was necessary for evaluating the block bordered and five point stencil applications, both of which require at least four processors. The latter configuration was also useful for demonstrating VP runtime system behavior in the presence communication performance slower than that on the Intel Paragon. Each comparison between virtual processor and manually contracted

Table 1: Manually contracted and virtual processor results for the Navier-Stokes application. Execution times of virtual processor and manually contracted versions of Navier-Stokes smoothing kernel for varying granularities. The DEC Alpha results utilize a 2 processor DEC Alpha workstation. The Intel Paragon results utilize 16 Paragon processors.

Platform	Threads Per Processor	Flops Between Communication Points	Manual Running Time	VP Running Time	Overhead
Intel Paragon	784	50	5.63	6.45	14.6%
		100	9.26	9.41	1.6%
		150	12.85	12.38	-3.7%
	1296	50	7.89	10.06	27.5%
		100	13.88	14.86	7.1%
		150	19.40	19.61	1.1%
	1936	50	10.82	14.49	33.9%
		100	19.61	21.60	10.1%
		150	28.82	28.56	-0.9%
Network of DEC Alphas	800	50	3.74	4.97	33%
		100	5.94	6.58	11%
		150	8.05	8.17	1%
	900	50	3.64	4.76	31%
		100	5.74	6.27	9%
		150	7.67	8.03	5%
	1250	50	5.40	7.15	32%
		100	8.59	10.13	18%
		150	13.29	14.23	7%

versions of the applications is made for varying granularities of computation between communication points in each VP. The granularity of computation is adjusted as follows: by varying the first (i) dimension of the three dimensional grid for the Navier-Stokes problem, by varying the number of unknowns per grid point for the five point stencil problem, and by varying the size of the blocks for the block bordered solver. Note that typically, the granularity indicates the number of the flops that the VP performs between context switches. It is possible, however, that additional context switches are required if additional context switches are necessary to maintain data consistency as discussed in Section 3.4. It is also possible, although very rare, that no context switch would be required at a communication boundary. This would occur if the communication boundary consisted of a receive and the VP executing the receive had already been sent the requested data from the sending VPs.

In the Navier-Stokes application, the finer the granularity of computation between communication points in each VP, the larger the relative overhead of our runtime approach. This is observed in both the Intel Paragon and DEC Alpha results. For example, with 1296 threads per processor, the overhead on an Intel Paragon is as little as 1.1% for 150 flops per VP, or as much as 27.5% for 50 flops per VP. The DEC Alpha overhead is relatively similar to the Paragon overhead except in very coarse granularities. For these coarser granularities, the VP version is slightly faster than the manual version on the Paragon in some cases. The

Table 2: Manually contracted and virtual processor results for the five point stencil algorithm. Execution times of virtual processor and manually contracted versions of the five point stencil application for varying granularities. The DEC Alpha results utilize two networked DEC Alpha workstation. Each DEC Alpha workstation has two processors.

Platform	Threads Per Processor	Flops Between Communication Points	Manual Running Time	VP Running Time	Overhead
Intel Paragon	625	60	4.43	5.68	28.2%
		125	6.23	7.54	21.0%
		250	9.60	10.93	13.9%
		375	13.10	14.42	10.1%
	1521	60	7.82	12.69	62.2%
		125	11.78	16.97	44.1%
		250	19.39	24.73	27.5%
		375	27.69	32.43	17.1%
	2500	60	11.05	20.00	81.0%
		125	17.77	26.79	45.1%
		250	30.28	39.48	30.3%
		375	43.21	51.74	19.7%
Network of DEC Alphas	3600	125	10.93	11.03	1%
		250	13.52	13.19	-2%
		375	16.61	15.68	-6%
	4900	125	12.96	13.65	5%
		250	16.60	16.31	-2%
		375	20.38	19.38	-5%
	6400	125	15.29	16.61	9%
		250	20.25	20.23	0%
		375	24.81	24.97	1%

Paragon, unlike a network of DEC Alphas, has a separate CPU dedicated to processing communication. By exclusively using asynchronous communication, the VP version is able to reduce communication cost relative to the manually contracted version on the Paragon and achieve slightly better performance.

Table 2 shows that as in the Navier-Stokes application, coarser granularity results in smaller runtime overhead in the five point stencil application on the Paragon. However, the overhead is much higher for comparable numbers of VPs per processor and flops between communication points than for Navier-Stokes.

The higher overhead of the five point stencil application is due to the fact that the VP version of the five point stencil application context switches more frequently to enforce data consistency, performs more same-processor communication per VP, and has a more significant gap in memory performance between the VP and manual versions than in the Navier-Stokes application. That is, reduced cache locality is more significant in the five point stencil application when using the VP approach than in the Navier-Stokes application (see Section 5.2). For example, on the Intel Paragon, the Navier-Stokes application has an overhead of only 1.1% for a granularity of 150 flops and 1296 threads per processor. The five point stencil

has an overhead of 44.1% for a granularity of 125 flops and 1521 threads. This large difference illustrates that the limits of a VP approach will be different for various applications.

Table 2 also shows that in the five point stencil application and the block bordered application, the VP versions yielded slightly faster performance than the manual versions on the DEC Alpha platform in most cases. This will also be seen to be true for the block bordered application on the DEC Alphas. For the five point stencil application, this performance gain is due to the high communication cost on the network of DEC Alphas compared to the Paragon. This high communication cost allows our VP approach to exploit the VP-level of parallelism when the manual version would be blocked on a receive (see Section 5.4). In the block bordered case, the faster VP times are due to locality and communication library issues that are described in more detail later in this section as well as in Section 5.5.

Another trend to note from the Navier-Stokes results in Table 1 and the five point stencil results in Table 2 is that, as the number of threads per actual processor grows, the overhead increases noticeably. This trend exposes the limitation of a polling mechanism that is layered on top of a communication library. Specifically, as the polling FIFO becomes larger (and more threads are used), more computation is necessary to find a newly arrived message corresponding to a blocked thread. Thus, the polling FIFO does not scale. A thread aware communication library would solve this scaling problem at the cost of portability. A thread aware communication library would be integrated with threads such that the arrival of messages immediately causes "communication library threads" to be scheduled. Thus, the communication library threads would be scheduled directly by the communication library when messages arrive. This would be much more efficient than having a separate thread library layered on top of the communication library, checking for message arrival by polling periodically (as in our approach). To our knowledge, such a library does not exist at the moment and would need to be developed and incorporated into the VP runtime system. In addition, such a communication library would require that the VP runtime system take explicit advantage of the communication library threads, making it difficult to port the VP runtime system to platforms without a thread aware communication library.

The block bordered application is coarse grained in comparison to the other two applications. The number of flops between communication points is roughly $O(N^3 + M^2N)$, i.e. at least 1000 for the cases considered. The block bordered case also utilizes very little virtual processor communication. For these reasons, one would expect the block bordered application to show very little overhead in the virtual processor case. The results in Table 3 show this to be true. In fact, the virtual processor versions are faster than the manual versions. On the Intel Paragon, the virtual processor version is at least 30% faster for all granularities. On the DEC Alpha, the virtual processor version is between 1% and 4% faster in most cases.

Two factors contribute to the superior performance of the virtual processor versions of the block bordered application. First, locally allocating data owned by each VP yields significantly better memory performance than globally allocating data as in the manual case. This factor is far more significant for the Paragon which has no level two or level three cache, than for the DEC Alphas which have a 32k level

Table 3: Manually contracted and virtual processor results for the block bordered algorithm. Execution times of virtual processor and manually contracted versions of the block bordered application for varying granularities. The DEC Alpha results utilize two networked DEC Alpha workstation. Each DEC Alpha workstation has two processors.

Platform	Threads Per Processor	Q, M, N	Manual Running Time	VP Running Time	Overhead
Intel Paragon	750	11250, 10, 8	1.79	0.89	-50%
		11250, 20, 8	3.88	2.06	-47%
		11250, 30, 8	6.23	3.75	-40%
		11250, 40, 8	11.09	5.99	-46%
	1000	15000, 10, 8	2.37	1.18	-50%
		15000, 20, 8	5.07	2.72	-46%
		15000, 30, 8	8.16	4.98	-39%
		15000, 40, 8	12.48	7.94	-36%
	1250	18750, 10, 8	3.31	1.46	-56%
		18750, 20, 8	6.25	3.38	-46%
		18750, 30, 8	10.27	6.18	-40%
		18750, 40, 8	15.31	9.88	-35%
Network of DEC Alphas	1000	3000, 25, 10	1.03	1.04	1%
		3000, 50, 10	3.02	2.95	-2%
		3000, 75, 10	5.93	5.74	-3%
		3000, 100, 10	9.59	9.44	-2%
	1500	4500, 25, 10	1.50	1.55	3%
		4500, 50, 10	4.43	4.36	-2%
		4500, 75, 10	8.67	8.42	-3%
		4500, 100, 10	14.45	14.06	-3%
	2000	6000, 25, 10	2.13	2.08	-2%
		6000, 50, 10	5.81	5.62	-3%
		6000, 75, 10	11.58	11.12	-4%
		6000, 100, 10	19.05	18.54	-3%

two cache and a very large level three cache. This difference accounts for the far larger improvement of the VP version in the Paragon than on the DEC Alphas. Second, the virtual processor runtime system uses a communication interface that is faster than that used in the manual version for the block bordered application. Instead of specifying the receiver (as is necessary in the manual version), the runtime system combines the receiver's identity with the message type. These two phenomena were identified using cache simulations and communication timings.

In summary, the results in the section show that it is possible to execute fine grain parallel codes with over 1000 VPs per actual processor and as few as 50 flops per VP with very acceptable overhead in comparison to manually contracted code. They also indicate that some performance advantages may arise from a VP implementation, due to the different styles of VP parallel programming and manually contracted parallel programming. While the above experiments used a fixed number of actual processors, it was observed on the Intel Paragon that varying the number of actual processors did not effect the magnitude of VP overhead.

5 Performance analysis

While the previous section gives the overall overhead when executing fine grain parallel code using VPs, the reasons for this overhead have not been fully explored. It is important to analyze and understand the overhead in our runtime approach so that the approach can be improved. To this end, this section analyzes the overhead in executing fine grain virtual processor versions of these applications.

This section is organized as follows. The methods used in obtaining performance metrics are discussed in Section 5.1. Section 5.2 analyzes the single processor overhead in detail, providing the relative overhead costs of thread communication, data consistency, context switching, scheduling, and reduced memory performance for each of the three applications. Section 5.3 then gives the total overhead of a single processor, fine grain VP version of each application and compares it to the parallel fine grain overhead determined in Section 4. There are significant differences between single processor and multiprocessor results, indicating that measurement of the time spent waiting on communication is an important factor. Using such communication measurements, the overlap of communication and computation is characterized in Section 5.4. Section 5.5 describes the effects of data layout and communication library behavior in the block bordered case. A key conclusion of the analysis in this section is that the overhead in our runtime approach is greatly reduced by overlapping communication and computation.

5.1 Tools and techniques

Execution times for the three test applications on both the Intel Paragon multiprocessor and on a network of DEC Alpha workstations were measured in single user mode. Network congestion was minimized by running test applications at off-peak hours. Variances in runs were insignificant when this step was taken.

An instrumentation tool called ATOM [16] was used on a single DEC Alpha to measure the overhead costs due to thread communication, maintaining data consistency, context switching, and scheduling. Memory performance also was evaluated using an ATOM-based cache simulator. This cache tool provided simulated miss rates and miss counts for the Alpha hardware configuration used. The cycles per instruction (CPI) and instruction count for each function of the program also were obtained using ATOM.

The time spent waiting on communication in the manually contracted case was obtained by measuring the length of time for receives to return control to the program after being called. In the virtual processor case, the time spent waiting on communication was measured by timing the computation necessary to poll the communication layer and the asynchronous receive call.

5.2 Analyzing single processor overhead

Virtual processor and manually contracted versions of all three applications were run on a single Paragon node and a single DEC Alpha workstation. That is, the entire problem was executed on one processor, with grid sizes comparable to the per processor grid sizes on the multiprocessor versions. These runs were used both to categorize individual overheads as discussed in this section, and to measure total single processor overheads as discussed in Section 5.3. Because the single processor versions do not communicate off-processor, the overhead due to managing the request FIFO associated with polling is not accounted for in these runs. This overhead is insignificant compared to the other sources of overhead mentioned previously, but is accounted for in the time spent communicating in the multiprocessor cases.

In order to understand the overhead of our approach in detail, we divide the overhead into the following categories:

1. Thread communication cost: This cost represents the cycles needed to maintain information about basic communication dependencies between VPs in the same address space. For example, a VP blocked on a receive must be added to the ready list when the corresponding send has been called. The bulk of this cost is due to manipulating data structures representing communicated data and message types.

2. Data consistency cost: The cycles needed to manage information about the number of readers and writers of data communicated between VPs on the same processor.

3. Context switch cost: The cycles needed to save and restore register state when switching between user level threads representing VPs.

4. Scheduling cost: The cycles needed to manage a ready list and implement communication-based scheduling (see Section 3.1.3) to determine which VP runs next.

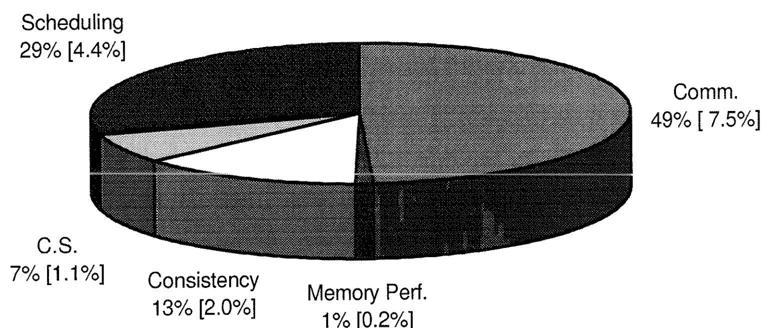


Figure 4: Overhead for the Navier-Stokes application on a DEC Alpha workstation shown as percentage of total overhead [percentage of sequential execution time]. Overhead represents 15% of execution time.

5. Reduced memory system performance: The cycles due to cache misses resulting from additional references by the runtime system, additional references due to changing stacks, and different data references patterns due to VP scheduling order.

Figures 4–6 show the categorization of the total overhead described above for each of the three test cases on a single DEC Alpha processor. The Navier-Stokes application was run with 100 flops per VP and 900 threads. The five point stencil was run with 250 flops per VP and 3600 threads. The block bordered application was measured such that $M = 75$ and $N = 10$ using 1500 threads per processor. The percentage breakdown of the overhead was arrived at utilizing overhead based on the number of cycles for each category of overhead.

The first four costs listed above were measured by using the ATOM instrumentation tool to count the number of cycles used by the functions that make up each of these four categories. The contribution of reduced memory system performance to the virtual processor overhead described above was arrived at using a cache simulator. We are interested in additional cycles due to cache misses in the virtual processor cases. The number of data and instruction references in the manual and virtual processor cases were counted. It was then determined how many of these references result in additional misses in the virtual processor case. The extra cycles needed to execute memory references were then calculated based on a simple model where the miss penalty is six cycles. The additional cycles that are required for memory references in the virtual processor cases of the test applications were then determined and combined with the other overhead results.

Figures 4–6 show that the overhead in our VP approach is very dependent on application behavior. Scheduling and memory performance are particularly dependent on application behavior. Scheduling accounts for 29% of the overhead in the Navier-Stokes (4.4% of sequential execution time) and only 5% in the five point stencil case (2.1% of sequential execution time). Due to the irregular, pipelined nature of the Navier-Stokes application, VPs in the Navier-Stokes application utilize the ready FIFO much more frequently than in the five point stencil case resulting in a higher scheduling cost due to more conditionals per scheduling decision.

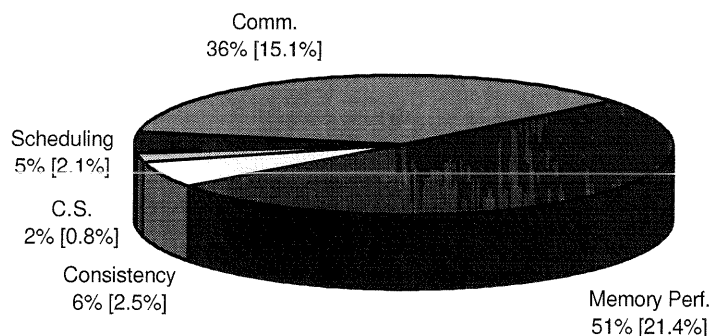


Figure 5: Overhead for the five point stencil application on a DEC Alpha workstation shown as percentage of total overhead [percentage of sequential execution time]. Overhead represents 42% of execution time.

Memory performance is also dependent on the application. In the five point stencil application, 51% of the overhead is due to memory performance overhead while in the Navier-Stokes application only 1% of the overhead is associated with reduced memory performance. These two applications exhibit entirely different memory performance characteristics between their respective VP and manual cases. The memory performance of the five point stencil application is much more sensitive to threads because the manual version achieves excellent locality using loops in optimal coordinate direction to traverse the grid points, while the asynchronous VP scheduling in the VP case leads to worse data locality. In contrast, the multiple direction sweeps required in the Navier-Stokes application results in less optimal memory locality in the manual version so that the asynchronous VP scheduling in the VP case doesn't lead to an appreciable extra cost.

From the figures, context switch overhead appears to be more significant in Navier-Stokes than the five point stencil case where memory performance accounts for a large share (51%) of the overhead. When comparing percentage of execution time, however, the context switch overhead in these two applications is very similar. Context switching accounts for 7% of the overhead in the Navier-Stokes (1.1% of sequential execution time) and 2% of the overhead in the five point stencil case (0.8% of sequential execution time).

Similarly, data consistency appears to be more significant in the Navier-Stokes application (13%) than in the five point stencil (6%). When comparing the percentage of execution time, however, the overhead due to enforcing data consistency accounts for 2.5% of the sequential execution time in the five point stencil case and 2% in the Navier-Stokes applications. As previously described, data consistency overhead is due to executing the vp_modify and vp_receive functions which update the "Number Readers" field in the data table of the VP that owns the communicated data. It is not surprising that this overhead accounts for slightly more of the execution time in the five point stencil case where blocking due to consistency is more frequent than in the Navier-Stokes case. In the Navier-Stokes application, blocking to enforce consistency never occurs.

The difference in communication overhead in the five point stencil and Navier-Stokes applications is significant. While communication overhead accounts for a much larger share of the total overhead in the Navier-Stokes application (50% compared to 36%), the percentage of execution time devoted to managing communication between VPs is much higher in the five point stencil application (15% instead of 7.5%). Each VP in the five point stencil application communicates with four neighbors at communication boundaries. In Navier-Stokes, only one neighbor is communicated with at communication boundaries. Therefore, a VP in the five point stencil application must perform four times the work necessary to update the message table than a VP in Navier-Stokes. This results in more time spent in managing VP communication in the five point stencil application.

The block bordered overhead is much different than the other two applications. Data consistency overhead is absent since asynchronous intraprocessor communication is not necessary. Memory performance is actually better in the virtual processor version than the manual version. Thus, memory performance overhead is absent. What is potentially misleading about the overhead categorization for the blocked bordered application is that the total overhead is very small compared to the other to applications. This was verified by removing the data layout advantages inherent in the VP block bordered case as described in Section 5.5. Using the same data layout in the VP and manual cases, the sequential VP block bordered application runs 1% slower than the manual version. This is much lower than the VP runtime overhead of 30% to 50% observed in the other two applications. Thus, the overhead described in Figure 6 for the block bordered application makes up a much smaller portion of the execution time than the overhead of the other two applications.

The overhead categorization shown in Figures 4–6 shows that context switch overhead (even without whole program analysis) is a relatively small cost of multithreading. Saving and restoring register state is actually a very small cost in a threaded runtime that is specialized for a particular domain. The categories of memory performance and communication between VPs, not context switching, dominate the overhead of our runtime system. The overhead due to communication between VPs represents a significant portion of the total overhead, from 24% to 51% in all three applications.

Finally, it should be noted that the context switching and memory performance overheads shown in Figures 4–6 would be higher if we did not perform two important optimizations: communication-based scheduling and analysis to reduce register state at context switches. Additional measurements show that the scheduling approach improves execution performance 15% in the five point stencil case and reduces the data cache miss rate from 40% to 3.5% relative to a random scheduling approach. Additional measurements also show that context switch overhead would double if whole program analysis was not used to identify minimal register state. This is discussed in detail in the next chapter.

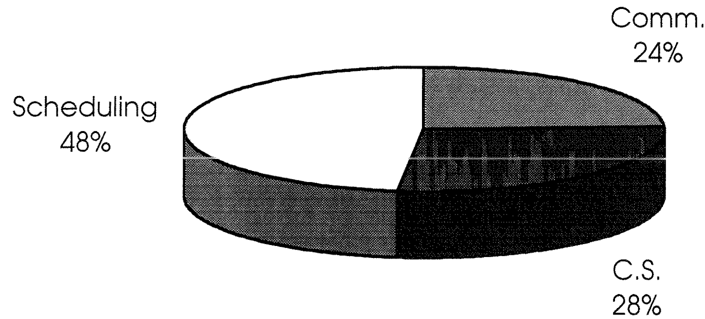


Figure 6: Overhead for the block bordered application on a DEC Alpha workstation shown as percentage of total overhead [percentage of sequential execution time]. Overhead represents less than 1% of execution time.

5.3 Single processor versus multiprocessor overhead

The total overheads of the three applications were measured on a single paragon node and a single DEC Alpha workstation, for a subset of the cases tested in Section 4. They are given in Tables 4 and 5. They show that the single processor overhead of the VP implementation is significantly greater than the multiprocessor overhead in almost all cases. For example, on the Intel Paragon, the single processor overhead for the 784 thread Navier-Stokes case is 21.4% while the multiprocessor overhead is only 1.1%. Similarly, the 1521 thread five point stencil case on the Intel Paragon has 38% single processor overhead and only 24.7% multiprocessor overhead. The DEC Alpha single processor and multiprocessor Navier-Stokes results have similar discrepancies. For example, the 900 thread Navier-Stokes case has single processor overhead of 16% while the multiprocessor overhead is only 9%. The 3600 thread five point stencil application on the DEC Alpha has single processor overhead of 77% while the multiprocessor overhead shows the virtual processor version is 2% faster.

Unlike the other two applications, the block bordered results show improved performance relative to the manual results on both platforms. For example in the 750 thread case, the virtual processor version of the block bordered application is 20% faster than the manual version on a single Intel Paragon processor. On a multiprocessor, however, the virtual processor version is 40% faster. The performance behavior of the block bordered application is discussed in more detail in Section 5.5.

In all cases, the single processor VP overheads would lead one to expect slower multiprocessor VP performance than actually is achieved. The discrepancy is described next.

5.4 "Communication overlap" effect on overhead

The results in Section 4 and 5.3 show lower overhead due to the VP runtime system in the multiprocessor cases than in the single processor cases. This section explains this phenomenon.

Table 4: **Single processor overhead on the Intel Paragon.** Execution times for manual and VP implementations of each test application on a single Intel Paragon Node. The Navier-Stokes application was run with a third dimension of size 100. The five point stencil was run with a granularity of 250 flops. For the block bordered case, M and N were fixed at 30 and 8. In some cases, the number of iterations executed is increased relative to the multiprocessor results to achieve accurate timings.

Application	Threads	Manual Running Time	VP Running Time	Overhead
Navier-Stokes	784	5.41	6.57	21.4%
	1296	9.01	11.06	22.8%
	1936	13.86	16.75	20.9%
Five Point Stencil	625	5.51	7.58	38%
	1521	13.60	18.72	38%
	2500	22.35	31.06	39%
Blocked Border	750	4.5	3.6	-20%
	1000	5.9	4.8	-19%
	1250	7.4	6.0	-19%

Table 5: **Single processor overhead on a DEC Alpha.** Execution times of manual and VP implementations of each test application on one DEC Alpha workstation. The Navier-Stokes application was run with a third dimension of size 100. The five point stencil was run with a granularity of 50 flops. For the block bordered case, M and N were fixed at 75 and 10. In some cases, the number of iterations executed is increased relative to the multiprocessor results to achieve accurate timings.

Application	Threads	Manual Running Time	VP Running Time	Overhead
Navier-Stokes	800	2.35	2.68	14%
	900	2.61	3.04	16%
	1250	3.70	4.30	16%
Five Point Stencil	625	0.89	1.33	49%
	900	1.28	1.89	48%
	3600	5.08	8.97	77%
	4900	6.88	13.20	92%
	6400	9.80	18.87	93%
Blocked Border	1000	5.58	5.38	-4%
	1500	8.24	8.08	-2%
	2000	11.09	10.73	-3%

Table 6: **Time spent waiting on communication on the Intel Paragon.** Time spent waiting on receive primitives in the underlying communication library while executing the Navier-Stokes and the five point stencil applications on the Intel Paragon. The Navier-Stokes application was run with a granularity of 100 flops. The five point stencil application was run with a granularity of 250 flops.

Application	Threads	Manual Communication Time	VP Communication Time	Percent Reduction
Navier-Stokes	784	3.62	2.48	31%
Five Point	625	4.00	2.55	36%

In a multiprocessor algorithm with multiple VPs per actual processor, some of the runtime overhead for the VPs can be masked when the processor would otherwise be waiting on communication. That is, when the manually contracted version must block on a receive, the VP version may be able to schedule and run another VP. The overhead of the VP version is partially masked since the VP version does not have to wait for the receive to complete. Thus, the time spent waiting on communication overlaps with useful computation. This factor turns out to account reasonably well for the discrepancy between the multiprocessor overhead and the single processor overhead in the Navier-Stokes and five point stencil applications. (The discrepancies in the block bordered results are described in Section 5.5.)

The time spent waiting on receives in the virtual processor and manually contracted versions of two applications is compared in Tables 6 and 7 for one five point stencil case and one Navier-Stokes case on each of the two platforms. The time spent waiting on receives in the manually contracted case was obtained by measuring the length of time for receives to return control to the program after being called. In the virtual processor case, the time spent waiting on communication was measured by timing the computation necessary to poll the communication layer and the asynchronous receive call. Thus, all functionality in the VP application having to do with receiving messages is included in the time spent waiting on receives in Tables 6 and 7.

Tables 6 and 7 show that the VP versions of the applications spend less time waiting on communication than the manual versions of the applications. As stated above, the difference arises because a threaded runtime system such as ours is able to utilize some of the time that would otherwise be spent waiting on communication, by running other VPs.

Using the data in tables 1 or 2, 4, and 6, it can be verified that the lower overhead of the parallel VP implementation on the Intel Paragon relative to the single processor VP implementation is accounted for by the overlapping of communication with computation in the parallel VP case. Specifically, the sum of the single processor execution time and the time spent waiting to receive data in the VP case closely matches the parallel execution time in both cases. This data also shows that the reduction in parallel overhead in the VP case closely corresponds to the reduction in communication cost. For example, on the Paragon, the manual version of Navier-Stokes application requires 5.41 seconds to execute on a single processor. The

Table 7: Time spent waiting on communication on the DEC Alpha. Time spent waiting on receive primitives in the underlying communication library while executing the test cases on the DEC Alpha platform. The Navier-Stokes application was run with a third dimension of size 100. The five point stencil application was run with a granularity of 250 flops.

Application	Threads	Manual Communication Time	VP Communication Time	Percent Reduction
Navier-Stokes	900	1.42	1.24	13%
Five Point	3600	9.68	7.38	24%

same application spends 3.62 seconds waiting on communication when run in parallel. Thus, we expect the parallel Navier-Stokes case to take 9.03 seconds to execute. This is very close to the actual 9.26 seconds actually taken. Similarly, the single processor VP version of the Navier-Stokes application executes in 6.57 seconds and spends 2.48 seconds waiting on communication in the parallel case. Again, we would expect the parallel case to execute in 9.05 seconds which is very close to the 9.41 seconds actually taken. If the manual and VP multiprocessor cases are compared, as well as the single processor data, we see that there are 1.14 fewer seconds spent waiting on communication in the parallel VP case than the manually contracted parallel case, almost the same as the 1.16 seconds of single processor overhead due to our VP approach. Thus we expect the manual and VP multiprocessor execution times to be nearly identical and indeed they are (9.26 and 9.41 seconds).

The above analysis can be repeated for five point stencil application on the Paragon with similar results. Unfortunately, the MPI implementation on the DEC Alpha does not allow this same verification. This is due to the fact that the MPI UNIX process that manages communication causes additional CPU load in the parallel case, the effect of which cannot be quantified in our estimates of time spent waiting on communication without significant kernel-level instrumentation. Nonetheless, the DEC Alpha communication times (see Table 7) also show that the VP versions spend less time communicating than the manual versions, resulting in lower parallel than single processor VP overhead.

5.5 Data layout and communication in block bordered application

The block bordered application is unique among the three applications in that the VP version has inherently better cache behavior compared to the manual version. While the other two applications have a single data structure that is mapped to each of the virtual processors, the block bordered application has several matrices and vectors that are individually owned by each virtual processor. It is natural for the VP programmer to allocate this data local to the virtual processor memory area. In contrast, the manual version globally allocates arrays of matrices and vectors. While each approach is the natural way to allocate data for that parallel programming model, the virtual processor approach yields dramatically better locality.

The block bordered application is also unique in that the runtime system manages communication in an inherently superior way compared to the manual version. When receiving data, the manual version must distinguish the sender. This isn't necessary in the other two applications. To distinguish the sender, the manual version uses a special receive primitive that allows this. The virtual processor version uses the normal receive primitive because the VP identifier, which is embedded in the message type, provides the required sender information. The latter approach is much faster on the Intel Paragon and slightly faster using a MPI communication library on the DEC Alpha platform.

To verify the above advantages, the virtual processor version was modified to use the same data layout as the manual version (which is unnatural in the VP case). The runtime system was also modified to use the same communication mechanism as the manual case. These changes caused the virtual processor version to be 6% slower instead of 40% faster. This result is consistent with additional cache simulation and communication timings.

6 Conclusion and future work

This research shows that a virtual machine abstraction based on specialized multithreading can be efficient and effective for implementing fine-grain parallel send/receive code on coarse grain multiprocessors. This efficiency is achieved through specialized scheduling, communication, and context switching. An important observation is that the combination of multithreading and asynchronous communication offsets the overhead of the VP runtime system by overlapping communication with computation. Lastly, the VP approach can have inherent performance advantages over manually written code.

Future work will focus on automating the whole program optimization described in section Section 3.3 to reduce context switch cost. Future work will also focus on thread aware communication library extensions to improving polling performance. Lastly, it would be interesting and useful research to examine the applicability of the thread specialization techniques used in this research to other application domains where polling, scheduling, and context switching are equally applicable. Examples of such domains include microkernel threads, threaded databases, distributed shared memory systems, and virtualized parallel thread file services.

References

[1] Arvind, Rishi Yur S. Nikhil, and Keshav K. Pingali. I-Structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, Oct 1989.

[2] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software Practice and Experience*, 18(8):713–732, Aug 1988.

[3] Virgil Bourassa and John Zahorjan. Implementing lightweight remote procedure calls in the mach 3 operating system. Technical Report 95-02-01, Department of Computer Science and Engineering University of Washington, February 1995.

[4] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Jun 1988.

[5] Richard P. Draves. A revised ipc interface. In *Proceedings of First Mach USENIX Workship*, pages 101–121, Oct 1990.

[6] Edward W. Felton and Dylan McNamee. Improving the performance of message-passing applications by multithreading. In *Scalable High Performance Computing Conference*, pages 84–89, Apr 1992.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. Available on netlib.

[8] Dirk Grunwald. A user's guide to Awesime: An object oriented parallel programming and simulation system. Technical Report CU-CS-522-91, Universit of Colorado, 1991.

[9] Dirk Grunwald, Richard Neves, and Robert B. Schnabel. Reducing user-level thread overhead using whole program analysis. *[In preparation for journal submission]*, 1995.

[10] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. Technical Report ICASE Report 94-25, Institute for Computer Applications in Science and Enginerring, NASA Langley Research Center, Hampton, VA 23681, Apr 1994.

[11] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[12] IEEE. Threads extension for portable operating systems. Technical Report Draft 7, POSIX, Feb 1992.

[13] Richard Neves and Robert B. Schnabel. Efficient compile-time/run-time contraction of fine grain data parallel codes. In *Lecture Notes in Computer Science, Languages and Compilers for Parallel Computer*, pages 430–448. Springer-Verlag, Aug 1993.

[14] D. Olander and Robert B. Schnabel. Preliminary experience in developing a parallel thin-layer navier stokes code and implications for parallel language design. In *Proceedings of Scalable High Performance Computing Conference*, pages 276–283. IEEE Press, 1992.

[15] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, 1988.

[16] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[17] Colin Theaker. The design of the muss operating system. *Software, Practices, and Experience*, 9(8):599–620, 1979.

[18] V. Vatsa and B. Wedan. Development of an efficient multigrid code for 3-d navier-stokes equations. In *AIAA 20th Fluid Dynamics, Plasma Dynamics and Laser Conference*, Jun 1989.

[19] Xiaodong Zhang, Richard H. Byrd, and Robert B. Schnabel. Parallel methods for solving nonlinear block bordered systems of equations. *Siam J. Sci. Stat. Comput.*, 13(4):841–859, Jul 1992.