CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2001; **00**:1–99 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

A comparison of concurrent programming and cooperative multithreading

Aaron W. Keen, Takashi Ishihara, Justin T. Maris, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson*,†

Department of Computer Science, University of California, Davis, CA 95616 USA

Notes to Editor and Reviewers:

- *Please send correspondence regarding this paper to* Olsson.
- *A preliminary version of this paper ([1]) appeared in Euro-Par 2000, held in Munich, Germany, August 2000. The present paper has been significantly revised and updated. It now contains a more thorough discussion of several important issues — e.g., implementation issues, tradeoffs, and experiments with Java (new) — and much more comprehensive performance results and explanations.*

SUMMARY

This paper presents a comparison of the cooperative multithreading model with the general concurrent programming model. It focuses on the execution time performance of a range of standard concurrent programming applications. The overall results are mixed. In some cases, programs written in the cooperative multithreading model outperform those written in the general concurrent programming model. The contributions of this paper are twofold. First, it presents a thorough analysis of the performances of applications in the different models, i.e., to explain the criteria that determine when a program in one model will outperform an equivalent program in the other. Second, it examines the tradeoffs in writing programs in the different programming styles. In some cases, better performance comes at the cost of more complicated code. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: cooperative multithreading; concurrent programming; parallel and distributed programming languages; synchronization mechanisms; synchronization optimization

*Correspondence to: Prof. Ronald A. Olsson, Department of Computer Science, University of California, Davis, CA 95616 USA
†E-mail: {keen,ishihara,maris,liti,fodor,olsson}@cs.ucdavis.edu
Contract/grant sponsor: Z-World, Inc. and the University of California under the MICRO program.

Received Sometime 2001

1. INTRODUCTION

The general concurrent programming execution model (CP) typically provides independent processes as its key abstraction. Processes execute nondeterministically. That is, processes run in some unknown order, which can vary from execution to execution, and context switches can occur arbitrarily. Multiple processes within a given program may execute at the same time on multiple processors, e.g., on a shared-memory multiprocessor or in a network of workstations. This model of execution is found in many concurrent programming languages — e.g., Ada [2], CSP [3], Java [4], Orca [5], and SR [6, 7]. These languages provide various synchronization mechanisms (e.g., semaphores, monitors, or rendezvous) to coordinate the execution of processes. Some of these languages require process execution to be fair, so that a language implementation might need to force context switches to prevent starvation. A language implementation might represent these conceptual, language-level processes as system-level processes or as threads within one or more system-level processes.

The cooperative multithreading execution model (CM) is a more specialized model of execution. Threads execute one at a time. A thread executes until it chooses to yield the processor or to wait for some event to become true. The kinds of events for which a thread can wait include a shared variable meeting a particular condition, a device completing some operation, or a timeout occurring. This model of execution is especially well-suited for writing programs for real-world programmable controllers for embedded systems [8], such as those found in irrigation control systems and railroad crossing control systems. One language for writing these controllers is Z-World's Dynamic C [9].

The CM model as defined above allows only one thread to be active at any given time. A natural generalization of CM (called PCM, for Parallel CM) is to allow multiple threads to be active simultaneously, so that a CM program can run on multiple processors. However, to preserve some of the advantages of CM (described later), some restrictions need to be placed on which particular threads can be run simultaneously. For example, only one thread per module, as in Lynx [10, 11], or one thread per group of threads with possibly interfering variable usages may be active at any time.

Two significant advantages of CM have been pointed out in the key work [10, 11] and specifically for controllers in [8]: CM is a simpler conceptual model and threads often do not need to synchronize explicitly because threads yield the processor at fixed places in the code. Two other tradeoffs [12] involve how the effect of I/O can vary in the different models and the relationship of execution fairness to program determinacy.

In this paper, we present a comparison of the cooperative multithreading model (CM or PCM) with the general concurrent programming model (CP). We focus on the execution time performance of a range of standard concurrent programming applications. The overall results are mixed. In some cases, programs written in the cooperative multithreading model outperform those written in the general concurrent programming model. The contributions of this paper are twofold. First, it presents a thorough analysis of the performances of applications in the different models, i.e., to explain the criteria that determine when a program in one model will outperform an equivalent program in the other. Second, it examines the tradeoffs in writing programs in the different programming styles. In some cases, better performance comes at the cost of more complicated code. (This paper extends our preliminary comparisons of CM and CP [12] and PCM and CP [1]).)

The programs used in our experiments — CP-style, CM-style, and PCM-style — are written in SR. For specifying concurrent or multithreaded execution, all three styles use SR's basic process notation. For synchronization, the CP-style programs use SR's semaphores, the CM-style programs use shared variables, and the PCM-style programs use both. (Further description of how SR is used for all three styles appears later.) Although the programs used in our experiments are written in SR, the general performance results should apply to some extent to other languages and systems. The specific performance results will vary depending on relative costs of synchronization and context switches, specific implementation details, etc. Toward testing the general applicability, we also performed some experiments with CP-style and CM-style Java programs.

The rest of this paper is organized as follows. Section 2 briefly compares language features typical in the three models. Section 3 discusses implementation issues that are key in understanding the experimental results and how they affect the basic execution time costs in implementations of the three models. Section 4 presents execution time performance comparisons for SR programs written in the CM- or PCM-style with their counterparts written in the CP-style for several standard CP applications. Section 5 reports on some similar performance comparisons repeated for Java programs. Section 6 addresses additional issues raised by our work and discusses related work. Finally, Section 7 concludes the paper.

2. LANGUAGE FEATURES

We assume most readers are familiar with CP languages (such as those languages mentioned in Section 1) but are less familiar with CM or PCM languages. We therefore briefly present the essential ideas of two such languages — Dynamic C [9] and Lynx [10, 11].

Dynamic C extends the C language with various features to support CM. Its **costate** statement defines a block of statements, which executes as a separate thread with its own hidden statement counter. A thread executes until it chooses to yield the processor or to wait for some event to become true. Yielding the processor is accomplished via explicit statements: **yield** and **waitfor**. **yield** context switches to another ready thread, if any, or resumes the current thread if no other thread is ready. **waitfor** evaluates the condition. If true, the thread continues; otherwise, the thread yields and will, therefore, reevaluate the condition when it runs again. (Some notations use **await** rather than **waitfor**.)

To illustrate Dynamic C, Figure 1 shows code for a piston controller [8]. The pneumatic piston moves between two ends of a cylinder. It stays 1.3 seconds on one end, moves to the other end, stays 4 seconds on that end, and then repeats. The controller also maintains a two line LCD that displays the current time updated each second and the current status of the piston.

Lynx is conceptually similar to Dynamic C. A Lynx program consists of a single module, called a *process*. Each process may contain multiple threads. As in Dynamic C, only one thread may be active at a time and threads execute until they block. An important feature of Lynx is that Lynx programs (processes) can communicate via messages using a *link* mechanism; the

```
main() {
   while(1) {
      UpdateTime(&MsCount); /* set MsCount to current MilliSecond counter */
      costate TimeLCD {
         while(1) {
            get current clock time and display it on LCD line 1
            ClockMsCount += 1000;
            waitfor(MsCount >= ClockMsCount); /* wait for another second */
         }
      }
      costate PistonThread {
         while(1) {
            display "moving to END1" on LCD line 2
            turn off valve1 and then turn on valve2
            waitfor(piston to reach END1);
            display "arrived at END1" on LCD line 2
            PistonMsCount += 1300;
            waitfor(MsCount >= PistonMsCount); /* wait for 1.3 seconds */
            display "moving to END2" on LCD line 2
            turn off valve2 and then turn on valve1
            waitfor(piston to reach END2);
            display "arrived at END2" on LCD line 2
            PistonMsCount += 4000;
            waitfor(MsCount >= PistonMsCount); /* wait for 4 seconds */
         }
      }
   }
}
```

Figure 1. Dynamic C pseudo-code for a piston controller

receipt of a message creates a new thread to run code to handle the message. Lynx falls under the PCM model because multiple threads — but at most one in each process within a group of processes — may execute at a time.

To illustrate the differences between the CP and CM models, Figure 2 shows how the classic dining philosophers problem can be solved in CP and CM. The CP code uses standard SR features; for synchronization, it uses the shared array of semaphores `fork`. The CM code use only CM-like features from SR; for synchronization, it uses the shared array of integers `fork` and a simulated **await** statement. This simulation uses SR's `nap` function to explicitly yield. (SR programs can be executed in CM-style; see Section 3.1.)

The key difference in these program fragments is how the philosopher checks the status of its two neighboring philosophers to decide whether it can eat. In CP, synchronization is required to avoid race conditions. By contrast, in CM, a context switch can occur only explicitly. Thus, no context switch can occur within evaluation of the conditions of the (simulated) **await** or between that test, if true, and the subsequent setting of the two elements of `fork`. (Note that the CM code here is not valid under PCM; see Section 4.2.)

```
do true ->
   # think
     ...
   # get forks
   # (use semaphore operations
   # on shared sem array fork;
   # left and right are indices
   # of neighboring philosophers)
   P(fork[left]);P(fork[right])
   # eat
     ...
   # release forks
   V(fork[left]);V(fork[right])
od
```

(a) CP-style

```
do true ->
   # think
     ...
   # get forks, by simulating:
   #   await fork[left]=1 & fork[right]=1
   do not(fork[left]=1 & fork[right]=1) ->
     nap(0) # i.e., yield
   od
   fork[left] := 0; fork[right] := 0
   # eat
     ...
   # release forks
   fork[left] := 1; fork[right] := 1
od
```

(b) CM-style

Figure 2. Code for a Philosopher in Dining Philosophers

3. KEY IMPLEMENTATION ISSUES AND BASIC COSTS

As noted in Section 1, a language implementation might represent conceptual, language-level processes as system-level processes or as threads within one or more system-level processes. Tradeoffs between these approaches are discussed in detail in [13, 14]. In the SR implementations used in this paper, language-level processes are represented as threads within a single system-level (UNIX) process. We use the term "process" for both a language-level process and its implementation-level representation as a thread.

Also as noted in Section 1, context switches from one process to another in a CP program can occur arbitrarily, whereas context switches from one thread to another in a CM or PCM program occur only when a thread blocks or explicitly yields. This section discusses key implementation issues and how they affect the basic execution time costs in implementations of CP, CM, and PCM programs.

3.1. Key Implementation Issues

One key implementation issue is how an implementation of a CP language realizes implicit context switching. The two main approaches are:

time-slicing: Allow a process to execute for some quantum of time. When that quantum expires, switch to another process. This approach is nearly identical to how operating systems time-slice user processes, but a language implementation typically performs time-slicing within a single user process. This approach requires the language's run-time system (RTS) to interact with the underlying operating system's timing facilities.

iteration-counting: Allow a process to execute for some number of loop iterations. This approach is based on the realization that if a process is running and it does not block, then it must be looping. This approach requires the language's compiler to generate additional iteration-counting code as part of each loop. When this code detects that the iteration limit has been reached, it invokes the run-time system to switch to another process.

The standard SR implementation [6, 7] uses the iteration-counting approach; this paper will refer to this implementation as SR_{IC}. An experimental SR implementation uses the time-slicing approach; it uses UNIX signals. We developed this implementation as part of this work; this paper will refer to this implementation as SR_{TS}.

Another key implementation issue is to ensure that internal RTS data structures are accessed with appropriate exclusion. In SR_{IC}, that is simple because the RTS is entered only via calls from the generated code. So, the single processor version of SR's RTS requires no extra code, but the multiprocessor version, known as MultiSR [15], requires code to lock key data structures in case two processes enter the RTS at about the same time. (Typically, the implementation uses spin locks.) In SR_{TS}, ensuring appropriate exclusion is more complicated because a process might be executing within the RTS when the signal indicating end of time-slice occurs. Our initial implementation of SR_{TS} disabled signals whenever an RTS procedure is entered. Although that approach is correct, it requires frequent interaction with the operating system (i.e., making system calls is expensive), which degrades performance. Our current implementation instead uses a flag variable to indicate that the RTS is locked, thereby reducing the overhead considerably. (Exclusive access to the flag variable is ensured in the multiprocessor version by using locks.)

SR programs can be written in the CM-style (as seen in Figure 2(b)) and executed in the CM-style. The standard SR implementation provides a compile-time option that prevents the implicit context switches described above. When invoked with this option, the SR compiler simply does not generate the iteration-counting code. This paper will refer to this use of SR as SR_{CM}.

The above discussion introduces three kinds of SR implementations: SR_{IC}, SR_{TS}, and SR_{CM}. Each of those has a single processor version and a multiprocessor version (based on MultiSR). However, we will use the same term (e.g., "SR_{IC}") for both versions and will distinguish between them when necessary (e.g., "the multiple processor version of SR_{IC}").

3.2. Basic Implementation Costs

The key factors in understanding the relative performances of the three kinds of SR implementations — SR_{IC}, SR_{TS}, and SR_{CM} — are context switching costs and synchronization costs. Below, we discuss each of these in detail and present data to illustrate. The data are from tests run on a 550 MHZ Pentium III processor PC running RedHat Linux 6.2 with the 2.2.14-5.0smp kernel. This system is a dual processor, but the single processor versions of SR run SR processes (threads) on only a single processor at any time, so it is equivalent to running on a single processor for those tests.

To compare relative performance of CM-style programs to CP-style programs, we use the ratio of execution times multiplied by 100%, i.e.,

$$T_{SR_{CM}}/min(T_{SR_{IC}}, T_{SR_{TS}}) * 100\%$$

Table I. SR simple loop results on a single processor

SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	CM/CP ratio (%)
5.41	1.46	1.46	100.0

For brevity, we shall refer to this ratio as CM/CP or PCM/CP. Note that this ratio uses the time of the best (fastest running) CP implementation for each test.

3.2.1. Context Switching Costs

As described above, to effect implicit context switching, SR_{IC} generates additional code as part of each loop, whereas SR_{TS} uses a timer. SR_{CM} does not effect implicit context switching. By default, SR_{IC} uses a *switch count* of 10,000 iterations (i.e., the generated code requests a context switch every 10,000 iterations) and SR_{TS} uses a quantum of ten millisecond. Ten milliseconds is the smallest quantum possible on the test system. Section 4.3 discusses how different sized quanta and switch counts can affect performance.

Table I presents the execution times for the three kinds of implementations of a simple micro-benchmark program designed to measure costs of the context switching code. The program consists of four processes, each of which just executes a loop of 10,000,000 iterations. As can be seen, the SR_{IC} performance is considerably worse than the performances of the other two implementations, which are identical. The reason is that in such a simple program the iteration-counting code is the dominant cost. Further, given the default switch count of 10,000, the SR_{IC} context switches 4,000 times. In contrast, the SR_{TS} context switches only 146 times and SR_{CM} does not context switch at all.

Table II presents the execution times for the above micro-benchmark when run on the multiprocessor versions of the three implementations. Given that the test system has two processors, one would hope for a speedup of 2.0 with respect to the data in Table I. As shown in the table, SR_{CM} obtains the ideal speedup[†] and SR_{TS} obtains very close to the ideal speedup. The difference for the latter is due to extra initialization costs and context switch costs. SR_{IC}, however, obtains a speedup of only 1.55. The difference is due to the extra locking it must now perform within the RTS, which occurs, in this program, when the RTS is entered for a context switch. As noted above, the single processor version of SR_{IC} did not need any locking.

[†]Because this test has no shared variables, it does not require processor affinity. Section 4.2 further discusses this issue.

Table II. SR simple loop results on a dual processor (cf. Table I)

SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	PCM/CP ratio (%)
3.50	.75	.73	97.33

Table III. SR cost of synchronization *per operation* on a single processor

SR_{IC} P and V (μsec)	SR_{TS} P and V (μsec)	SR_{CM} yield (μsec)	CM/CP ratio (%)
.45	3.82	.37	82.20

3.2.2. Synchronization Costs

As noted in Section 2, CP-style SR programs use semaphores, whereas CM-style SR programs use shared variables and yield. Table III shows the relative basic costs of these operations. These data were obtained from two micro-benchmark programs. Each program consists of two processes, with control alternating between them on each iteration. The data for the CP program shows the combined cost of a P operation that blocks a process and a V that unblocks a process. The data for the CM program shows the cost of a yield (i.e., `nap(0)`); the yield in this program is unconditional, whereas often the yield is conditional based on the result of testing shared variables. Unlike the other tables in this paper, this table (and Table IV) shows the execution time per indicated operation, not the execution time for the entire micro-benchmark program; thus, the costs of loop overhead, etc. are *not* included. As can be seen from the table, SR_{IC}'s performance is considerably better than SR_{TS}; the difference is due to the cost of extra locking within the RTS. SR_{IC}'s performance is slightly worse than SR_{CM}; the difference is due to the extra testing and queueing and dequeuing involved in implementing semaphore operations compared to implementing a simple yield.

Table IV presents the execution times for the above micro-benchmark when run on the multiprocessor versions of the three implementations. As shown, all implementations require more time than their single processor counterparts (Table III) due to the extra locking of the RTS that is required. SR_{CM} has the smallest absolute increase because its implementation of yield requires fewer entries into the RTS than the others' implementations of semaphores. Although SR_{TS} has the largest absolute increase and the worst absolute performance, it has the smallest relative increase because its single processor implementation already required some locking.

Table IV. SR cost of synchronization *per operation* on a dual processor (cf. Table III)

SR_{IC} P and V (μsec)	SR_{TS} P and V (μsec)	SR_{CM} yield (μsec)	PCM/CP ratio (%)
1.54	5.19	1.14	74.02

Table V. Standard CP applications used in the experiments

JI	Jacobi iteration (approximate the solution to a partial differential equation)
PC	Producer/Consumer
DP	Dining Philosophers
RW	Readers and Writers

4. EXPERIMENTAL RESULTS FOR SR PROGRAMS

We wrote in the CM and PCM programming styles the standard CP applications listed in Table V.[‡] We programmed the applications in the SR language, using standard CP features or CM-like features, as we did for the DP code in Figure 2. The PC, RW, and JI programs are fairly straightforward (the CP versions are taken from [7], which are also available online as part of the SR distribution [15]).

Below, we compare CP with CM (both running on a single processor[§]) and CP with PCM (both running on a multiprocessor) by looking at the execution times for the applications mentioned above. We ran each application on many different problem sizes. For example, for JI we tested with different size matrices, convergence values, and initial values; for RW, we tested with different numbers of reader processes and writer processes. For DP, PC, and RW, we tested with different amounts of time spent inside and outside of critical sections. The results we report below are representative of the observed results.

We ran the programs using the three kinds of SR implementations — SR_{IC}, SR_{TS}, and SR_{CM} — described in Section 3.1. Specifically, we ran the CM-style programs using SR_{CM}

[‡]The PC problem consists of a single slot buffer accessed, in general, by one or more producers and one or more consumers. The PC solutions in this paper use a single producer and a single consumer. The more general Bounded Buffer problem includes a multiple slot buffer; some results of our experiments for that problem appear in [1].
[§]CM applications are typically run on a single processor; some CP applications (e.g., servers) are run on a single processor.

and the CP-style programs using both SR_{IC} and SR_{TS}. The data presented in this section are from the tests run on the same test system as used in Section 3.2: a 550 MHZ single or dual processor PC. To compare the relative performance of CM-style programs to CP-style programs, we use the CM/CP or PCM/CP ratio defined in Section 3.2. To understand better where execution time was being spent, we modified the SR run-time system to report, upon program termination, the number of context switches, the total number of semaphore P operations performed, and the number of P operations that block. Finally, we used *gprof* to profile the code.

4.1. CP versus CM (single processor)

This section presents and explains the results on the applications listed in Table V. The CM programs outperformed CP programs in all cases, although just slightly in about half the cases. (However, Section 4.3 presents additional cases, in which the CP programs outperform the CM programs.) In some cases, the SR_{TS} significantly outperforms SR_{IC}; in other cases, the situation is the opposite. The key factors in understanding the relative performances are the basic implementation costs (for context switching and for synchronization) described in Section 3.2 and application effects:

work: "Work" indicates how much non-critical activity a process performs. For example, in PC, it represents what a producer must do to produce a new item. In some applications, such as PC, the work is "real", i.e., some computation is performed. However, in other applications, such as DP, the work representing the philosopher's eating or thinking is simply a busy-waiting loop. Note that whether or not work includes real computation can have a significant effect on the overall results. In applications run under SR_{IC} with just busy-waiting work, the loop overhead can dominate, as discussed in Section 3.2. (See Section 6 for further discussion.) We expect that most practical applications would fall within the lower work categories, which incur fewer context switches and synchronization points.

patterns of synchronization: The different applications have different patterns of synchronization. For example, in PC, the producer and consumer alternate in their access to the buffer, whereas in RW, multiple readers can concurrently access the database.

Table VI presents the data for DP. For small work values, CM outperforms CP. However, for large work values SR_{CM} and SR_{TS} are very close because a given philosopher can run to completion without ever having to delay for the forks it needs. That happens here in the CM program due to the nature of CM model and in the SR_{TS} program because the work that needs to be done can be finished within a single quantum. The difference between SR_{CM} and SR_{TS} for small amounts of work is due to overhead in the SR_{TS}. In contrast, SR_{IC} performs poorly for large work values because it context switches more frequently. For example, when one philosopher is eating, SR_{IC} might context switch to that philosopher's neighbor, which might also want to eat. However, because the neighbor is not able to eat, it will need to block.

Table VII presents the data for PC. The execution times for the three programs are close. The slight differences are due to the different costs between `P/V` and `nap(0)` and due to the

Table VI. SR DP results on a single processor

work	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	CM/CP ratio (%)
100	0.42	2.00	0.13	30.95
1000	3.69	3.18	1.19	37.42
10000	33.41	13.73	11.73	85.43
100000	318.39	119.41	117.13	98.09
1000000	3166.19	1175.76	1171.12	99.60

Table VII. SR PC results on a single processor

work	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	CM/CP ratio (%)
100	2.32	2.27	2.19	96.47
1000	22.95	21.72	21.61	99.49
10000	229.09	216.24	215.76	99.77
100000	2296.20	2161.61	2157.51	99.81

overhead in the CP implementations. In this application, the work includes real computation, which dominates the overall execution.

Table VIII presents the data for RW. CM outperforms both CP implementations. For small work values, SR_{TS} outperforms SR_{IC} because no context switches occur due to having reached the switch count or the end of the quantum, so SR_{IC}'s loop overhead dominates. However, for larger work values, SR_{IC} outperforms SR_{TS} because this application requires a fair amount of synchronization. As seen in Table III, `P/V` synchronization in SR_{IC} is less costly than in SR_{TS}. Further, the amount of synchronization in this application takes sufficient time so as to force some additional context switches.

Table IX presents the results for JI. Each "test" entry indicates the size of the (square) matrix and the number of processes used in the test. (The initial values and convergence criteria also differed, although those are not shown in the "test" entry.) The performances are very close because this application is mostly computational, which is the dominant cost. However, the small differences between execution times can be attributed to context switching overhead. The difference increases for larger problems due to increased context switching.

Table VIII. SR RW results on a single processor

work	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	CM/CP ratio (%)
100	0.85	0.49	0.44	89.79
1000	0.93	0.87	0.49	56.32
10000	1.75	2.19	1.02	58.28
100000	10.09	14.61	6.37	63.13
1000000	93.01	141.20	59.81	64.30

Table IX. SR JI results on a single processor

test	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	CM/CP ratio (%)
128 8	3.72	3.81	3.71	99.73
256 2	19.25	18.80	18.47	98.24
512 8	77.59	77.17	73.11	94.73

4.2. CP versus PCM (multiprocessor)

We ran the applications again using the multiprocessor versions of the three kinds of SR implementations — SR_{IC}, SR_{TS}, and SR_{CM} — described in Section 3.1. The data presented in this section are from the tests run on the same test system as used in the previous tests (Sections 3.2 and 4.2): a 550 MHZ dual processor PC.

For the DP problem, we used the same CP version as before (Figure 2(a)). The PCM version, though, is new. The basic approach, illustrated in Figure 3, splits philosophers into two regions: East and West. Each of these regions is assigned to a processor. Synchronization within a region uses shared variables, represented by dashed lines in Figure 3. Synchronization between the two regions, however, uses a semaphore, represented by solid lines in Figure 3. The code is, therefore, a hybrid of the code in the two parts of Figure 2 with three kinds of philosophers: interior philosophers 2-7 and 10-15, each of which uses shared variables to get both of its forks; border philosophers 1 and 9, each of which uses a semaphore to get its right fork but a shared variable to get its left fork; and border philosophers 8 and 16, each of which uses a semaphore to get its left fork but a shared variable to get its right fork. (Thus, interior philosophers can be viewed as being written in CM-style and border philosophers as in a combination of CP-style and CM-style.)

The MultiSR implementation, unfortunately, does not support processor affinity. (The same holds true for the underlying Linux Threads [16] on which MultiSR is built.) So, different

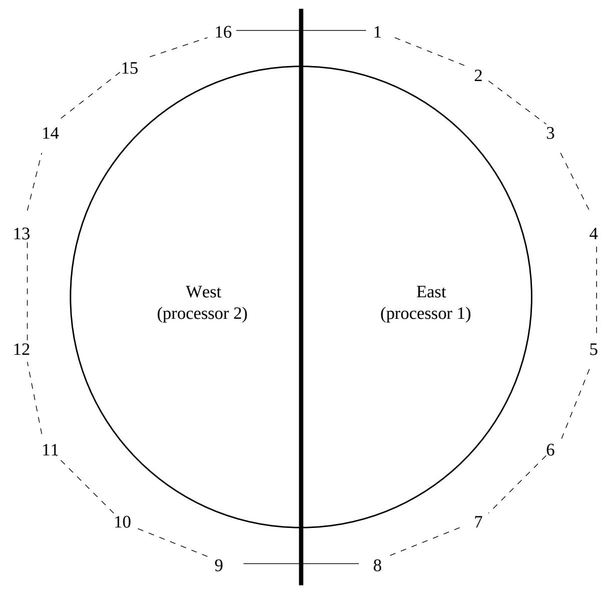

Figure 3. Layout of PCM Dining Philosophers for 16 philosophers and 2 processors

processes in the same region (East or West) could run at the same time, which violates the PCM assumption and could lead to a race condition on the shared variables.

In our simulation, therefore, we tested two versions — DP_1 and DP_2 — of PCM DP. Both include extra (semaphore) synchronization to protect the shared variables ("protection synchronization"). DP_2 includes additional synchronization to ensure that only one process in a region runs at the same time ("region synchronization"). DP_1, on the other hand, allows more than one from the same region to run at the same time. DP_1 is a reasonable, conservative approximation to how PCM DP would perform. Given the characteristics of the tests (e.g., number of philosophers), it is likely that multiple philosophers from each region can run at the same time; so the overall performance is not likely to be improved by running two philosophers from the same region at the same time. The extra, protection synchronization means the measured costs are (most likely) higher than they would be for a pure PCM DP solution.

Table X presents the results for DP. DP_1 outperforms the CP versions of DP for all tested workloads, even though DP_1 has extra, protection synchronization. DP_2 is slightly slower due to its use of extra, region synchronization, but it too outperforms the CP versions of DP. The reason that the PCM programs outperform the CP program is essentially the same as that for the sequential version of the program (see Section 4.1): a philosopher can run to completion. However, given that the PCM programs have two regions of philosophers, some competition between philosophers can occur. When that happens, one philosopher will block and another in that region can run; such blocking and later unblocking, however, does not significantly

Table X. SR DP results on a dual processor

work	SR_{IC} (sec)	SR_{TS} (sec)	DP1 SR_{CM} (sec)	DP1 PCM/CP ratio (%)	DP2 SR_{CM} (sec)	DP2 PCM/CP ratio (%)
100	0.93	3.49	0.56	60.21	0.59	63.44
1000	3.72	4.59	0.81	21.77	0.88	23.65
10000	31.60	12.69	6.09	47.99	6.26	49.33
100000	299.82	63.56	59.17	93.09	62.41	98.19
1000000	2977.32	600.64	590.49	98.31	595.61	99.16

Table XI. SR PC results on a dual processor

work	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	PCM/CP ratio (%)
100	1.43	1.87	1.25	87.41
1000	12.04	11.88	10.91	91.83
10000	123.40	110.59	108.85	98.42
100000	1163.44	1094.92	1080.14	98.65

Table XII. SR RW results on a dual processor

work	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	PCM/CP ratio (%)
100	1.35	1.39	0.60	44.44
1000	1.49	1.52	0.65	43.62
10000	2.81	3.30	1.33	47.33
100000	16.04	19.95	7.92	49.38
1000000	155.80	186.20	73.86	47.41

contribute to the overall execution costs. The effect of DP_2's extra, region synchronization is small, which reflects that it does not cause many processes to block. As for the sequential version of the program (see Section 4.1), the differences in the execution times between the SR_{IC} and the SR_{TS} programs are due to extra context switching.

Table XI presents the results for PC. The PCM version of the PC program places the producer on one processor and the consumer on the other. The code uses protection

Table XIII. SR JI results on a dual processor

test	SR_{IC} (sec)	SR_{TS} (sec)	SR_{CM} (sec)	PCM/CP ratio (%)
128 8	4.30	4.21	2.66	63.18
256 2	13.79	14.13	13.68	99.20
512 8	48.18	47.95	43.84	91.42

synchronization (semaphores) to protect the shared buffer. In fact, the semaphore structure is identical to that in the CP version. The performance of the PCM program is better than the CP versions because it does not incur the context switching overhead present in the CP implementations.

Table XII presents the results for RW. As for the PC program, the RW program uses the same code for the CP versions and the PCM version. The overall results are similar too: PCM outperforms CP. The difference is more pronounced here, though, because, as noted in Section 4.1, the PC program does more real computational work than does the RW program.

Initially, we wrote another PCM version of the RW program, perhaps more in the PCM style. To allow concurrent readers, we split readers between the two processors, $P1$ and $P2$. We placed all writers on one of the processors, say $P1$. A reader on processor $P1$ can simply check a shared variable to see whether it can begin reading. A reader on $P2$ needs to access that shared variable with additional semaphore protection. A writer can access the number of readers on $P1$ using a shared variable, but it must also access the variable representing the number of readers on $P2$ with additional semaphore protection. As for the DP problem, extra, protection synchronization is also required here to protect shared variables. However, the PCM/CP ratio ranged from 109%-127%. The performance is worse for this program than the other PCM RW program because they differ in how they synchronize. The first program uses a mutual exclusion semaphore and separate semaphores on which to block readers and writers. The second program uses a single mutual exclusion semaphore around its tests of the shared variables. Hence, the second program incurs extra costs for awakening blocked writers when readers are still accessing the database; such writers just retest the shared variables and block again. In contrast, the first program awakens either a waiting writer or all waiting readers, depending on the system state.

Table XIII presents the results for JI. The CP and PCM versions of JI each use a barrier. The difference, though, is in how the barrier is coded. In the CP version, the barrier uses semaphores shared by all processes. On the other hand, the PCM version is similar in spirit to the PCM version of DP (DP_1). The N processes are divided into two groups. Each group of $N/2$ processes uses shared variables to implement the barrier among the group. The first $N/2 - 1$ workers in each group to arrive at the group's barrier use region-specific shared variables to synchronize. The last worker in each group to arrive at the group's barrier signals a semaphore to indicate that all processes in this region have reached the barrier. It then waits for the last process in the other region to signal it on another semaphore that all processes in the other region have reached their barrier too. (The actual code uses extra, protection synchronization, as in DP_1, to prevent race conditions.) The PCM version performs better, despite the extra, protection synchronization, than the CP version. The reason is its use of the simple shared variable barrier versus the more expensive semaphore barrier.

The overall speedups obtained between single processor and multiprocessor executions varied considerably among the applications and problem sizes. For the dual processor on which we ran the tests, one would hope for speedups close to 2.0. Such speedups were obtained for the PC problem (Tables VII and XI). Speedups between roughly 1.1 and 1.7 were obtained for the JI problem (Tables IX and XIII). Both the PC and JI problems are dominated by real computation, although the cost of the barrier synchronization in JI is also significant (more so in the multiprocessor versions due to its higher cost) and explains the lower speedup. For smaller work values, the multiprocessor versions of DP performed worse than the single processor versions (Tables VI and X) due to the increased implementation overhead (see Section 3.2). For larger work values, the SR_{TS} and SR_{CM} multiprocessor versions of DP achieved speedups of close to 2.0 (Tables VI and X). However, the SR_{IC} version achieved speedups of only about 1.1 (Tables VI and X) due to the dominant cost of loop overhead (see Section 3.2). For all work values, the multiprocessor versions of RW performed worse than the single processor versions — i.e., speedups less than 1.0 — (Tables VIII and XII). The reason is that the RW program does very little computation and much synchronization, the cost of which is considerably higher in the multiprocessor versions of SR (see Section 3.2).

4.3. Effect of Implementation Optimization and Tuning

The quantitative results presented in Sections 4.1 and 4.2 depend on the quality of the implementations. We attempted to improve and tune the implementations in several ways.

As noted in Section 3.1, we reduced the use of (expensive) signals in the time-slicing implementation. In addition, we attempted to improve the code the compiler generates for iteration-counting. This code is executed on each iteration of each loop and is thus critical in the overall performance of the iteration-counting SR implementation. However, this modification did not give significantly better results (about 1% improvement).

We also experimented by varying the switch count in SR_{IC} and by varying the quantum size used in SR_{TS}. Recall from Section 3.2 that the defaults are a switch count of 10,000 and a quantum of 10 milliseconds.

The results for PC and JI were nearly independent of the switch count or quantum (about 1% difference) because real computation dominates these applications. Although larger switch counts or quanta can reduce the amount of context switching, that amount is not significant. Moreover, such context switches do not make a significant difference in the pattern of synchronization in these applications. That is, unless the quantum is very small, a process will block on synchronization (e.g., when the buffer becomes full in the PC problem) before it reaches the end of its quantum.

However, the results for DP and RW on various switch counts and quanta showed significant differences. The results for DP appear in Tables XIV, XVI, XVIII, and XX. Varying the

Table XIV. SR_{IC} DP results on a single processor (cf. Table VI)

work	SR_{IC} with switch count of			
	1,000 (sec)	10,000 (sec)	100,000 (sec)	1,000,000 (sec)
100	0.43	0.42	0.41	0.41
1000	3.52	3.68	3.68	3.67
10000	33.03	33.41	36.33	36.33
100000	328.26	318.31	341.32	356.61
1000000	3271.07	3165.20	3203.71	3385.04

switch counts or quanta can affect the overall synchronization patterns.[¶] For the single processor versions, the SR_{IC} performances vary by about 11% (Table XIV), whereas the SR_{TS} performances vary significantly for small work values, but not much for large work values (Table XVIII) where the larger quanta does not change the overall synchronization pattern. A similar pattern in the data occurs for the multiprocessor versions (Tables XVIII and XX).

The results for RW appear in Tables XV, XVII, XIX, and XXI. The SR_{IC} performances vary little for the single processor and multiple processor versions (Tables XV and XVII). As noted in Section 4.1, the dominant cost for RW under SR_{IC} is the iteration counting. However, the SR_{TS} performances vary significantly for the single processor and multiple processor versions (Tables XIX and XXI). As noted in Section 4.1, the dominant cost for RW under SR_{TS} is synchronization, signal handling, and context switching. With larger quanta, fewer such costs are incurred. In fact, the single processor version of RW under SR_{TS} even outperforms its CM counterpart (Table VIII) due to "needless awakening". As noted in Section 4.2, the CM version needlessly awakens waiting writers when they have no chance of progressing, whereas the CP version awakens a waiting writer only when it is able to proceed.

5. EXPERIMENTAL RESULTS FOR Java PROGRAMS

We repeated the same experiments from Section 4.1 for programs written in Java, both CP-style and CM-style. The Java CP-style programs use the standard Java monitor-like synchronization mechanisms wait and notify. The Java CM-style programs use shared variables and yield. We ran these experiments on a Sun Sparc 20 workstation running SunOS 5.7, i.e., Solaris. (We did not repeat the experiments from Section 4.2 that involve PCM because we

[¶]A quantum of ∞ means that no timer interrupts occur. However, the implementation still uses the same locking as it does for finite quanta.

Table XV. SR_{IC} RW results on a single processor (cf. Table VIII)

work	SR_{IC} with switch count of 1,000 (sec)	10,000 (sec)	100,000 (sec)	1,000,000 (sec)
100	0.86	0.84	0.84	0.84
1000	0.94	0.93	0.94	0.92
10000	1.78	1.75	1.75	1.75
100000	10.16	10.01	10.00	10.00
1000000	94.01	92.66	92.53	92.51

Table XVI. SR_{IC} DP results on a dual processor (cf. Table X)

work	SR_{IC} with switch count of 1,000 (sec)	10,000 (sec)	100,000 (sec)	1,000,000 (sec)
100	0.89	0.93	0.93	0.94
1000	4.06	3.68	3.62	3.56
10000	33.55	31.47	31.70	31.35
100000	328.75	297.81	305.47	304.60
1000000	3263.59	2969.86	2941.91	2962.02

Table XVII. SR_{IC} RW results on a dual processor (cf. Table XII)

work	SR_{IC} with switch count of 1,000 (sec)	10,000 (sec)	100,000 (sec)	1,000,000 (sec)
100	1.40	1.35	1.33	1.34
1000	1.54	1.48	1.46	1.47
10000	2.89	2.81	2.79	2.78
100000	16.33	16.04	16.00	15.99
1000000	150.76	148.41	148.08	148.03

Table XVIII. SR_{TS} DP results on a single processor (cf. Table VI)

work	SR_{TS} with quantum of			
	10 msec (sec)	20 msec (sec)	1,000 msec (sec)	∞ msec (sec)
100	2.00	1.56	0.73	0.75
1000	3.18	3.13	1.79	1.80
10000	13.73	13.76	12.33	12.34
100000	119.41	119.19	119.14	117.78
1000000	1175.76	1173.44	1173.38	1171.79

Table XIX. SR_{TS} RW results on a single processor (cf. Table VIII)

work	SR_{TS} with quantum of			
	10 msec (sec)	20 msec (sec)	1,000 msec (sec)	∞ msec (sec)
100	0.49	0.50	0.31	0.31
1000	0.87	0.79	0.33	0.33
10000	2.19	2.17	0.60	0.60
100000	14.61	14.63	3.31	3.30
1000000	141.20	140.87	136.29	30.30

Table XX. SR_{TS} DP results on a dual processor (cf. Table X)

work	SR_{TS} with quantum of			
	10 msec (sec)	20 msec (sec)	1,000 msec (sec)	∞ msec (sec)
100	3.49	3.51	3.53	2.13
1000	4.59	4.56	4.56	3.18
10000	12.69	12.81	12.84	12.90
100000	63.56	63.06	63.13	62.72
1000000	600.64	597.48	597.46	596.74

Table XXI. SR_{TS} RW results on a dual processor (cf. Table XII)

work	SR_{TS} with quantum of 10 msec (sec)	20 msec (sec)	1,000 msec (sec)	∞ msec (sec)
100	1.39	1.37	1.38	0.69
1000	1.52	1.55	1.73	0.76
10000	3.30	3.29	3.18	1.54
100000	19.95	20.07	20.01	9.36
1000000	186.20	186.59	187.10	87.74

do not have access to a multiprocessor running Solaris.) We chose this platform because the "Green Threads" Java implementation (see below) runs only on Solaris.

The Java Language Specification [17] neither requires nor prohibits that Java threads execute fairly. Each Java thread has a priority, but the specification does not require that higher-priority threads be given preference over lower-priority threads (Section 17.12 of [17]). If all threads in a Java program have equal priority, then an implementation of Java can execute threads CP-style or CM-style. Most Java implementations (e.g., IBM's Version 1.3.0 of Java 2 SDK and Sun's Version 1.3.0 of Java 2 SDK) use the "native threads" package of the underlying operating system, which provides CP-style execution. However, older Solaris implementations (such as JDK 1.2.1-a) provide a "Green Threads" package that implements threads within the JVM (Java Virtual Machine). Such an implementation provides CM-style execution. However, periodic execution of the high-priority garbage-collection thread (which we will designate TGC) can affect thread scheduling. In particular, under Green Threads, it can make scheduling among equal, normal priority threads appear to be round-robin. For example, suppose threads $T1$ and $T2$ have equal, normal priority, and each thread contains a long loop with no yields. Then, the threads execute in the order $T1$, $T2$, $T1$, $T2$, etc. Our observations indicate that after TGC executes, the scheduler picks the next thread to execute rather than pick the thread that TGC preempted. On the other hand, if threads $T1$ and $T2$ have equal, high priority, then only $T1$ gets to execute. Our observations indicate that TGC does not get to execute, presumably because it has a lower priority. (Or, if the TGC does execute, the scheduler picks the preempted thread to run again afterwards.)

We ran Java versions of the DP, PC, RW, and JI applications (Section 4) on a single processor. We ran CM-style versions and CP-style versions of these applications on two different Java implementations — one using Green Threads and one using native threads — as shown in Table XXII. The table indicates that the two "natural" combinations are allowed. It also indicates that "CPonGreen" — CP-style program executed on CM-style implementation — is allowed: the applications do yield via their explicit synchronization (e.g., in the PC problem,

Table XXII. Java applications run on which Java implementations

	CM-style program	CP-style program
Green Threads (CM-style implementation)	yes “natural”	yes “CPonGreen”
native threads (CP-style implementation)	no	yes “CPnatural”

Table XXIII. Java DP results on a single processor

work	CPnatural (sec)	CPonGreen (sec)	CM (sec)	CM/CP ratio (%)
100	1891.8	2300.6	965.6	51.0
1000	9226.6	9575.8	8235.4	89.3
10000	82059.8	82344.6	80972.0	98.7
100000	810625.6	810543.4	808722.8	99.7

a producer will yield if it finds that the buffer is full).[||] Finally, the table indicates that one possibility — CM-style program executed on CP-style implementation — is disallowed: that combination could have race conditions.

Tables XXIII, XXIV, XXV, and XXVI[**] present the performance results for the applications. As in Section 4, the CM/CP ratio in these tables shows the ratio between the CM performance and the best of the two CP performances, i.e.,

$$T_{CM}/min(T_{CPnatural}, T_{CPonGreen}) * 100\%$$

The results show that on smaller amounts of work for DP, PC, and JI, CM outperforms CP, but the situation is reversed for RW. The results also show that CPonGreen outperforms CM or CPnatural in some cases, but it never outperforms both.

Despite these trends observed in the performance data, however, one should be cautious in drawing conclusions. The implementations of native threads and Green Threads are quite

[||]As noted earlier, this kind of execution is allowed by the Java Language Specification even if the application does not yield. However, for programs written in SR and some other languages, such an execution would not be allowed because process execution is required to be fair.

[**]Due to the significantly slower processor used for these tests, we used a slightly different mix of tests for JI in Java than we did for JI in SR (Tables IX and XIII).

Table XXIV. Java PC results on a single processor

work	CPnatural (sec)	CPonGreen (sec)	CM (sec)	CM/CP ratio (%)
100	27027.6	20244.6	19035.0	74.9
1000	194425.8	186983.8	185012.6	96.2
10000	1851453.0	1853873.8	1845413.8	99.6
100000	18444763.0	18440204.0	18438873.0	99.9

Table XXV. Java RW results on a single processor

work	CPnatural (sec)	CPonGreen (sec)	CM (sec)	CM/CP ratio (%)
100	6504.2	7034.0	8753.4	134.6
1000	7301.6	7665.6	9123.8	125.0
10000	14436.4	13807.2	12787.6	95.6
100000	133882.3	90370.4	49440.8	67.5

Table XXVI. Java JI results on a single processor

test	CPnatural (sec)	CPonGreen (sec)	CM (sec)	CM/CP ratio (%)
8 2	0.59	0.49	0.38	77.6
128 8	10413.45	9784.68	9645.64	98.6
256 2	147045.93	146885.12	146132.26	99.5

different. First, as noted above, native threads use OS (or kernel-level) threads whereas Green Threads use threads within the JVM (or user-level) threads. These threads have quite different performance characteristics ([13, 14]). For example, switching between two threads is more expensive using kernel-level threads because it requires a system call. Second, one thread package might provide additional functionality not provided by the other, the implementation of that additional functionality can have an impact on the performance of even its basic functionality. (In contrast, the only difference in the implementations of SR used in Section 4.1 was whether the RTS provided CM-style or CP-style execution; the thread package was identical in either case.) Nonetheless, the data gives some idea of performance of Java CM and CP.

6. DISCUSSION

We focused on the DP, PC, RW, and JI applications. We also experimented with SR versions of Matrix Multiplication (MM) and Traveling Salesman Problem (TSP). For reasonable problem sizes, MM is computationally intensive with almost no synchronization and TSP is computationally intensive with little synchronization. Thus, their performances were roughly similar to those seen for JI, i.e., very small differences between the models.

Applications such as JI, MM, and TSP run on a uni-processor system will (almost always) be faster if they are written as sequential rather than concurrent programs. However, applications such as PC, DP, and RW represent servers. Those applications and others such as HTTP or file servers lend themselves to multiple logical threads. For example, an HTTP server multiplexes multiple connections and might have one thread for each request being serviced. These threads might be expressed as processes within a CP program or as threads within a CM program (e.g., as in the Boa server [18]). Having logical threads can also be important with respect to I/O [12].

Earlier tests on other single processor platforms confirmed the general trends seen in Section 4.1. Those tests were run on several workstations including a Sun Sparc 5 workstation running SunOS 5.5, various Intel-based PCs running various versions of Linux, a DEC 5000/260 running ULTRIX 4.3, and a DEC Alpha running OSF1 V3.2. Our earlier paper ([1]) gives details. However, for the current paper, we have made changes to some of the test programs, so direct comparisons between the data in the two papers are not always possible. An example of such a change, and also an interesting general point about the benchmarks, is what exactly comprises "work" (Section 4.1). If we modify the work in our experiments so that it involves context switching — e.g., if DP uses a `nap(10)` to represent a philosopher eating for 10 milliseconds — then the resultant costs can dominate the program's execution time and the extra switching can also affect the program's synchronization pattern. Such changes can affect the results presented in Section 4. For example, the region synchronization in DP_2 can now often cause processes to block; the extra costs can make PCM program perform worse than the CP program, as indicated in [1].

We observed that, in some CM programs, processes were sometimes busy waiting more than necessary due to the order in which processes execute. For example, consider the code for the Bounded Buffer problem with two producers, two consumers, and one slot. If a producer that has just deposited an item yields to the other producer, then that producer will be awakened, perform any "work" it might need to do, see that it cannot proceed to deposit its item, and in turn will yield. Some context switching can be eliminated if the producer instead yields to a consumer, who can then fetch the item. We call this ability to select the process to which to yield a *named yield* and the usual yield an *unnamed yield*. (A named yield is like a coroutine resume.) We simulated named yield for the above problem and generally saw improvements of about 10%, especially for small amounts of work (i.e., where context switching is more dominant than work); see [1] for details. The simulated named yields relied on known attributes of the particular implementation's underlying scheduler. We are considering how named yields might be represented within the language or how that information might be made available to the scheduler, either statically or dynamically. The latter approach might be able to incorporate some of the dynamic feedback ideas presented in [19].

The style in which programs are written differs between the models. The difference between CP and CM can be seen clearly in Figure 2. The difference between CP and PCM was described, for example, for the DP problem in Section 4.2. There, the code is a hybrid of styles and the argument made regarding simplicity in [10, 11] is not as solid — the programmer needs to understand both models of execution to program in PCM. As one specific example, our original version of the PCM code for DP had a slight mistake (an "off by one" error) in coordinating the border philosophers. We did not discover the mistake until after the publication deadline for [1]. As a result, the DP data in Table 2 of that paper is incorrect, although the general trend in that table still holds for that specific benchmark.

The PCM DP example also raises the issue of load balancing. As presented, the processes were split statically into two groups (regions), under the implicit assumption that, overall, each group of processes was doing about the same amount of work. If that assumption is not correct, then philosophers could be moved. Specifically, a border philosopher could be moved from one group to the other; such a change would directly affect three philosophers and their roles as border or interior. The actual code to effect such a move would be complicated, especially when the philosophers are in the midst of synchronizing. In the CP model, load balancing can happen implicitly without any change to how the processes synchronize.

The results we gave for SR programs are based on measurements of both CP-style programs and CM- and PCM-style programs. The results might be skewed a bit in favor of the SR_{IC} programs because SR's underlying run-time system was designed for SR_{IC}. An implementation targeted specifically for SR_{CM} or SR_{TS} might be structured differently and perform better for some programs. However, having all applications written in the same language was useful: the implementation of other language features (e.g., code generated for accessing array elements) is consistent between the different versions of programs and therefore does not unfairly influence the results as it might when comparing programs written in different languages.

Some thread packages also provide a CP model of execution. For example, Linux Threads [16] uses threads within the operating system kernel to provide preemptible threads. Other thread packages provide a CM model of execution. For example, GNU's Portable Threads (Pth) [20] provide threads that are not preemptible. Yet other thread packages provide a PCM model of execution. For example, the Ultra-lightweight Thread (uThread) package [21] provides "run-to-completion" threads to avoid context switching. By allowing only one kernel-level process to run at each processor, it can eliminate the need for mutual-exclusion primitives for threads running within that kernel-level process.

A potential advantage of PCM-style programs is that they might benefit from cache affinity [22]. That is, processes that use the same variables will be placed on the same processor, for example, as in the PCM DP example.

Our work is related generally to other work that attempts to eliminate synchronization or replace synchronization by less expensive forms. Examples: eliminating barrier synchronization from parallel programs [23] and replacing more costly forms of message passing with less costly ones [24]. Both of those approaches employ compiler analysis, whereas the approach in this paper is aimed at the higher, language level.

We have also investigated how to transform CP programs into PCM or CM programs. Even if some programmers prefer to express their code within the CP model, that code can be transformed to PCM or CM and run more efficiently by eliminating synchronization. For example, a P/V pair of semaphore operations used for mutual exclusion can simply be eliminated under certain conditions. Note that some implementations of CP languages essentially map a CP into a PCM program anyway, but they generally need to assume the worst case of when context switches will occur. It is also desirable to automate these transformations. Unfortunately, we have not been successful in devising general transformations that would work for many programs. It seems that application-specific semantic information (which is not readily apparent from simple program structure) is required, thus defeating our goal of (relatively) simple and general automatic transformations.

7. CONCLUSION

We have presented a comparison of the cooperative multithreading models (CM and PCM) with the general concurrent programming model (CP). We examined execution time performance of a range of standard concurrent programming applications. The overall results are mixed. In some cases, programs written in the cooperative multithreading model outperform those written in the general concurrent programming model. The key factors are that CM and PCM programs can avoid context switching overhead and can use less costly synchronization or even eliminate some synchronization. We also examined the tradeoffs in writing programs in the different programming styles, and observed that, in some cases, better performance comes at the cost of more complicated code. Our experience indicates that the cooperative multithreading models (CM or PCM) deserve further exploration as viable alternatives to the general concurrent programming model (CP).

ACKNOWLEDGEMENTS

Joel Baumert made valuable technical suggestions on this work. Greg Benson provided helpful technical information.

REFERENCES

1. T. Ishihara, T. Li, E. F. Fodor, and R. A. Olsson. A comparison of concurrent programming and cooperative multithreading. In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Notes in Computure Science, pages 729–738. Springer-Verlag, August 2000.
2. Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995. `ftp://sw-eng.falls-church.va.us/public/Ada- IC/standards/95lrm_rat`.
3. C.A.R. Hoare. Communicating Sequential Processes. *Communications ACM*, 21(8):666–677, August 1978.
4. G. Cornell and C. S. Horstmann. *Core Java*. Sun Microsystems, Inc., Mountain View, CA, 1996.
5. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
6. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
7. G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.
8. Tak Auyeung. Cooperative multithreading. *Embedded Systems Programming*, pages 72–77, December 1995.
9. Z-World, Inc. *Dynamic C 5.x Integrated C Development System Application Frameworks (Rev.1)*, 1998. Dynamic C 5.x.
10. M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.
11. M. L. Scott. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages*, 16(3/4):209–233, 1991.
12. E.F. Fodor and R.A. Olsson. Cooperative multithreading: Experience with applications. In *The 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, pages 1953–1957, July 1999.
13. U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.
14. G. Benson and R. Olsson. A framework for specializing threads in concurrent run-time systems. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 139–152. Springer-Verlag, Pittsburgh, PA, 1998.
15. The SR programming language, version 2.3.2, August 1999. `http://www.cs.arizona.edu/sr/`.
16. S. Walton. *LinuxThreads*, 1997. `http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/`.
17. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, second edition, 2000. `http://java.sun.com/docs/books/jls/`.
18. `http://www.boa.org`, 1999.
19. P. C. Diniz. Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. *ACM Transactions on Computer Systems*, 17(2):89–132, May 1999.
20. GNU Pth — the GNU portable threads, March 2001. `http://www.gnu.org/software/pth/`.
21. Wei Shu. Runtime support for user-level ultra lightweight threads on massively parallel distributed memory machines. In *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 448–445, 1995.
22. R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 26–40, December 1991.
23. H. Han, C.-W. Tseng, and P. Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International Journal of Parallel Programming*, 25(5):591–612, October 1998.
24. C. M. McNamee. Transformations for optimizing interprocess communication and synchronization mechanisms. *International Journal of Parallel Programming*, 19(5):357–387, October 1990.