A Comparison of Concurrent Programming and Cooperative Multithreading *

Takashi Ishihara, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson

Department of Computer Science, University of California, Davis, CA 95616 USA
{ishihara,liti,fodor,olsson}@cs.ucdavis.edu

Abstract. This paper presents a comparison of the cooperative multithreading models with the general concurrent programming model. It focuses on the execution time performance of a range of standard concurrent programming applications. The results show that in many, but not all, cases programs written in the cooperative multithreading model outperform those written in the general concurrent programming model. The contributions of this paper are an analysis of the performances of applications in the different models and an examination of the parallel cooperative multithreading programming style.

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 [9], CSP [8], Java [5], Orca [3], and SR [1]. These languages provide various synchronization mechanisms to coordinate the execution of processes (e.g., semaphores, monitors, or rendezvous).

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 [2], 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 [15].

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

* This work is partially supported by Z-World, Inc. and the University of California under the MICRO program.

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 [12, 13], 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 [12, 13] and specifically for controllers in [2]: 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 [6] 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 models (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 results show that in many, but not all, cases programs written in the CM- or PCM-style outperform those written in the CP-style. The contributions of this paper are an analysis of the performances of applications in the different models and an examination of the PCM programming style. (This paper extends our preliminary performance comparison of CM and CP [6].) Although the programs used in our experiments are written in SR, the general performance results should apply to other languages and systems. The specific performance results will vary depending on relative costs of synchronization and context switches, etc.

The rest of this paper is organized as follows. Section 2 briefly compares language features typical in the three models. Section 3 presents execution time performance comparisons of programs written in the CM- or PCM-style with their counterparts written in the CP-style for several standard CP applications. Section 4 addresses additional issues. Finally, Section 5 concludes the paper.

2 Language Features

We assume most readers are familiar with CP languages (such as Ada, etc. 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 [15] and Lynx [12, 13].

Dynamic C extends the C language with various features to support CM. 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**.)

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 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 1 shows

```
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

Fig. 1. Code for a Philosopher in Dining Philosophers

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. The CM code is compiled with an option that prevents normal, implicit context switches.

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 3.2.)

3 Experimental Results

We wrote in the CM and PCM programming styles several standard CP applications including grid computations such as Jacobi Iteration (JI) (for approximating the solution to a partial differential equation), http servers, Producer/Consumer (PC[1]), Dining Philosophers (DP), Readers and Writers (RW), Matrix Multiplication (MM), and Traveling Salesman Problem (TSP). For the PC problem, we use the notation xPyCzS to mean x producers, y consumers, and a buffer with z slots.

We focus below on the DP, PC, and JI applications. 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 1. The PC and JI programs are fairly straightforward (the CP versions are taken from [1]).

Below, we compare CP with CM (both running on a single processor)[2] and CP with PCM (both running on a multiprocessor) by looking at the execution times for some of the applications mentioned above. We ran on many different problem sizes for each application. For example, for JI we tested with different size matrices, convergence values, and initial values. For PC, we tested with different numbers of consumer processes, producer processes, and number of slots in the buffer. For PC and DP, we tested with different amounts of time spent inside and outside of critical sections. The results we report below are representative of and summarize the observed results; see [10] for complete details.

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. We also ran several micro-benchmarks to determine the basic costs of context switches and costs of P operations that block and of those that do not block. Finally, we used *gprof* to profile the code.

3.1 CP versus CM (single processor)

These 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 5000/240 running ULTRIX 4.3, and a DEC Alpha running OSF1 V3.2. The data presented in this section are from the tests run on the Sparc 5 or the Pentium Pro 200 running Linux 2.2.5-22. The overall patterns of results on all platforms are similar. We also focus on relative performance and so give most results in terms of the ratio of execution times multiplied by 100%, i.e., $T_{cm}/T_{cp} * 100\%$.

Table 1 shows representative results for the DP, PC (1P1C1S), and RW problems. The "work" in the table indicates how much non-critical activity a process

[1] The PC problem with more than one slot is usually called the Bounded Buffer problem. We use PC for both in this paper.

[2] CM applications are typically run on a single processor; some CP applications are run on a single processor.

WORK	DP	PC	RW
100	20.0	92.7	77.8
1000	44.3	92.9	74.8
10000	52.8	93.1	70.8
100000	53.8	93.3	69.8
1000000	52.9	93.1	70.2

Table 1. Execution time ratio CM/CP (%) for three applications

performs. For example, in PC, it indicates how much relative work a producer takes to produce a new item. We expect that most practical applications would fall within the lower work categories, which incur fewer context switches and synchronization points.[3]

For the applications in Table 1, the CM programs perform better for two reasons: they perform no P/V operations (i.e., they synchronize using shared variables and less often) and they make fewer context switches. Of these two factors, the former is more important because it is more costly. For example, on a Sparc-5, it takes 20-30 μs per P operation versus 5 μs per context switch.

These results prompted us to look further into how we could reduce costs. We observed that, in CM programs, processes were sometimes busy waiting more than necessary due to the order in which processes execute. For example, in the code for 2P2C1S, if a producer that has just deposited an item yields to the other producer, then that producer will be awakened, see that it cannot proceed, and in turn will yield. Some context switching could be eliminated if the producer instead yields to a consumer. 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 PC problem. The graph in Figure 2 shows the results. The results are better than those shown for PC in Table 1. The results are also better on some tests for small amounts of work (*not* shown in Table 1), where the execution time ratios (%) using unnamed yields ranged from 90% to 120%.

We also looked at TSP, MM, and JI. For TSP and and MM, the programs written in the CM model obtain execution time ratios (%) between 97-99%, due to the small amount of synchronization in those applications. For JI, the CM model obtains execution time ratios (%) of between 98.5-100%, despite somewhat frequent barrier synchronization. For this application, the CM's barriers — busy waiting on shared variables with yields — turned out to cost about the same as the CP's barriers — performing P/V semaphore operations.

[3] Note that "work" means iterations of some computational loop. It does not necessarily correspond directly to elapsed time because, for example, if a process is waiting a long time for I/O to complete, other processes can execute during that time.

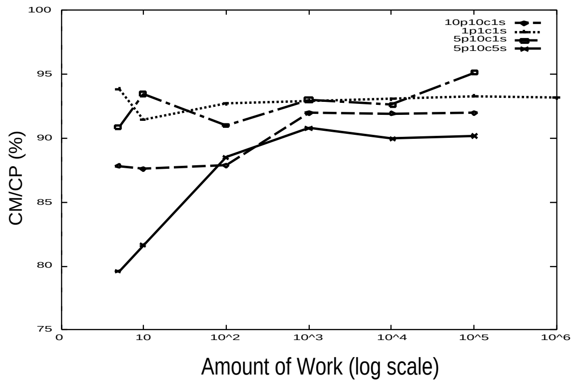

Fig. 2. Execution time ratio CM/CP (%) for the xPyCzS Problem

3.2 CP versus PCM (multiprocessor)

We ran further experiments on a dual 550 MHZ processor PC running Red Hat Linux 6.0. These experiments used the SR implementation (MultiSR) with support for multiprocessors.

We ran the applications mentioned previously. Again, the results for TSP and MM differed little due to the small amount of synchronization in those programs. Note that the PCM versions do need to synchronize between groups of processes (more on that below). The more interesting tests were DP, PC, and JI.

For the DP problem, we used the same CP version as before (Figure 1(a)). The PCM version, though, is new. The basic approach, illustrated in Figure 3,

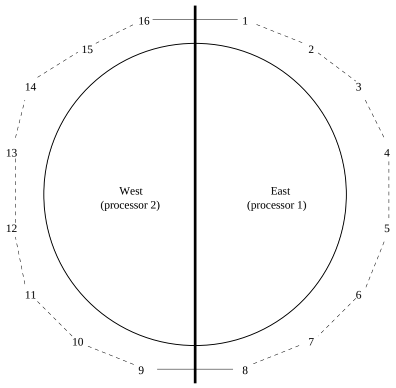

Fig. 3. Layout of PCM Dining Philosophers for 16 philosophers and 2 processors

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 1 with three kinds of philosophers: interior philosophers 2-7 and 10-15, each of which uses shared variables to get both of its forks; borders 1 and 9, each of which uses a semaphore to get its right fork but a shared variable to get its left fork; and borders 8 and 16, each of which uses a semaphore to get its left fork but a shared variable to get its right fork.

The MultiSR implementation, unfortunately, does not support processor affinity. (The same holds true for the underlying LinuxThreads on which MultiSR is built.) So, different 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 2 shows some representative speed-ups for the two PCM programs compared with the CP DP program. DP_1 outperforms the CP version of DP for all tested workloads, even though DP_1 has extra, protection synchronization. DP_2 performs considerably worse for most workloads due to its use of extra, region synchronization.

WORK	DP_1	DP_2
2	45.4	57.9
4	54.6	83.0
10	81.8	126.0
20	86.6	139.1
100	99.7	178.2
200	95.0	171.9
400	98.1	172.2
1000	95.9	167.4

test	PC
1	101.5
2	102.8
3	101.8
4	100.8

test	JI
1	52.6
2	94.8
3	88.4
4	83.8

Table 2. Execution time ratio PCM/CP (%) for three applications

We tested a PCM version of the PC program. The synchronization requirements of the program led us to place all producers on one processor and all consumers on the other because producers (consumers) need to synchronize in accessing the variable that indicates which item was removed (inserted). Unfortunately, the extra, protection synchronization required to protect shared variables results in code whose semaphore structure is identical to that in the CP version *and* has additional synchronization to simulate the **await** statements. Accordingly, its performance was slightly worse than the CP version, as indicated in Table 2.

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 processes are divided into two groups. Each group of processes uses shared variables to implement the barrier among the group. Two semaphores are used to synchronize between the two groups. (The actual code uses the extra, protection synchronization, as in DP_1, to prevent race conditions.)

Table 2 shows some representative speed-ups for JI. As shown, 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 based barrier. Note that the improvement for JI here is significant, whereas the improvement for JI in CM was nearly non-existent (see Section 3.1). The difference is that processes do fewer busy waits since they are now split onto two processors.

4 Discussion

Applications such as MM, TSP, and JI 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 [4]). Having logical threads can also be important with respect to I/O [6].

The results we gave are based on measurements of SR programs, both standard, CP-style programs and CM- and PCM-style programs. The results might be skewed a bit in favor of the CP-style programs because SR's underlying run-time system was designed for CP. 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.

The style in which programs are written differs between the models. The difference between CP and CM can be seen clearly in Figure 1. The difference between CP and PCM was described, for example, for the DP problem in Section 3.2. There, the code is a hybrid of styles and the argument made regarding simplicity in [12, 13] is not as solid — the programmer needs to understand both models of execution to program in PCM.

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.

A potential advantage of PCM-style programs is that they might benefit from cache affinity [14]. 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 [7] and replacing more costly forms of message passing with less costly ones [11]. Both of those approaches employ compiler analysis, whereas the approach in this paper is aimed at the higher, language level.

We are investigating 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 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 automatically transform programs like the CP version of DP into a PCM version. However, it is not clear how to devise general transformations that would work for many programs.

5 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 results showed that in many, but not all, cases programs written in the CM- or PCM-style outperform those written in the CP-style. The key factor is that cooperative multithreading allows less costly synchronization to be used or even some synchronization to be eliminated. We also examined the PCM programming style. Our experience indicates that the cooperative multithreading models (CM or PCM) are viable alternatives to the general concurrent programming model (CP) and are worthy of further exploration.

Acknowledgement

Joel Baumert made valuable technical suggestions on this work. The anonymous reviewers provided detailed comments that helped us to improve this paper.

References

1. G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.
2. Tak Auyeung. Cooperative multithreading. *Embedded Systems Programming*, pages 72–77, December 1995.
3. 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.
4. `http://www.boa.org`, 1999.
5. G. Cornell and C. S. Horstmann. *Core Java.* Sun Microsystems, Inc., Mountain View, CA, 1996.
6. 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.
7. 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.
8. C.A.R. Hoare. Communicating Sequential Processes. *Communications ACM*, 21(8):666–677, August 1978.
9. 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`.
10. Takashi Ishihara, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson. Cooperative multitasking versus general concurrent programming: Measuring the overhead associated with synchronization mechanisms. Unpublished Manuscript, University of California, Davis, September 1999.
11. C. M. McNamee. Transformations for optimizing interprocess communication and synchronization mechanisms. *International Journal of Parallel Programming*, 19(5):357–387, October 1990.
12. M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.
13. M. L. Scott. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages*, 16(3/4):209–233, 1991.
14. 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.
15. Z-World, Inc. *Dynamic C 5.x Integrated C Development System Application Frameworks (Rev.1)*, 1998. Dynamic C 5.x.