CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2002; **00**:1–99 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

A Comparison of Concurrent Programming and Cooperative Multithreading under Load Balancing Applications

Justin T. Maris, Aaron W. Keen, Takashi Ishihara, and Ronald A. Olsson*,†

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

SUMMARY

Two models of thread execution are the general concurrent programming execution model (CP) and the cooperative multithreading execution model (CM). CP provides nondeterministic thread execution where context switches occur arbitrarily. CM provides threads that execute one at a time until they explicitly choose to yield the processor. This paper focuses on a classic application to reveal the advantages and disadvantages of load balancing during thread execution under CP and CM styles; results from a second classic application were similar. These applications are programmed in two different languages (SR and Dynamic C) on different hardware (standard PCs and embedded system controllers). An SR-like run-time system, DesCaRTeS, was developed to provide inter-process communication for the Dynamic C implementations. This paper compares load balancing and non-load balancing implementations; it also compares CP and CM style implementations. The results show that: in cases of very high or very low workloads, load balancing slightly hindered performance; and in cases of moderate workload, both SR and Dynamic C implementations of load balancing generally performed well. Further, for these applications, CM style programs outperform CP style programs in some cases, but the opposite occurs in some other cases. This paper also discusses qualitative tradeoffs between CM style programming and CP style programming for these applications.

KEY WORDS: cooperative multithreading, concurrent programming, load balancing, parallel and distributed programming languages, synchronization mechanisms

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

Received Sometime 2002

1. INTRODUCTION

Two popular models of thread execution are the general concurrent programming execution model (CP) and the cooperative multithreading execution model (CM). CP provides independent threads as its key abstraction. Threads execute nondeterministically and context switches can occur arbitrarily. This model of execution is found in many concurrent programming languages, e.g., Ada [8], CSP [7], Java [3], Orca [2], and SR [1]. These languages provide various synchronization mechanisms to coordinate the execution of threads (e.g., semaphores, monitors, or rendezvous). CM provides threads that execute one at a time until they choose to yield the processor. Typically, threads yield the processor by executing an explicit yield statement. A more general version of CM, called parallel CM (PCM), allows multiple threads to be active simultaneously so that a CM program can run on multiple processors [9]. PCM programs can be run on a multiprocessor machine using shared memory or on a distributed network of processors. Reference [9] discusses performance tradeoffs between CM style programs and CP style programs.

This paper examines load balancing under dynamic circumstances to reveal the advantages and disadvantages of load balancing during thread execution under both CP and CM styles. Load balancing is defined as spreading workload across all processors in a system with the goal of reducing overall program execution time. Many load balancing algorithms attempt to load balance as new work enters the system [14, 10]. Over time, however, the load on the system may become unbalanced due to uneven processing speeds and thus needs to be redistributed. Studies have shown that this kind of dynamic load balancing can produce favorable results [6]. Similarly, the research presented in this paper examines load balancing that is done periodically during thread execution. As the state of the system becomes unbalanced, threads are moved from one processor to another.

Our research explores load balancing by performing experiments with applications in two environments: a standard network of Pentium class PCs and a network of embedded system controllers, which, compared to the PCs, are very slow and limited in memory. The specific embedded system controllers used for this research are Z World's Rabbit 2000 TCP/IP development boards. The Rabbit processor has an 18 MHz clock speed. Each Rabbit provides 256K of Flash EPROM for program and data, 256K for file storage, and 128K of SRAM. We use two classic applications: Dining Philosophers and Readers/Writers. (Because the overall results for these applications are similar, this paper discusses in depth only the first application; reference [11] discusses the second.) These application are representative of server applications that would be appropriate for both environments. For example, a web server can run in either environment; e.g., a web server for a weather station can run on a network of embedded systems [12]. Although computationally intensive applications can be run in either environment, they are not typically run on a network of embedded systems because of the very low processing power. In such environments, any intensive computation would typically be performed on a separate, more powerful system (e.g., a PC) connected to the network.

The two applications are programmed in the SR and Dynamic C [4] programming languages. SR supports both CP and CM styles. Dynamic C supports only CM style. The SR implementations are run on a network of PCs. The Dynamic C implementations are run on a network of embedded system controllers. Programs for these stand-alone microprocessors

Prepared using cpeauth.cls

are developed using a standard PC running the Dynamic C debugger and then downloaded to the microprocessors. As part of our work, we developed an SR-like run-time system, DesCaRTeS (Dynamic C Run-Time System), to provide inter-process communication for the Dynamic C implementations. Both SR and Dynamic C implementations are run with and without load balancing to test the overall effectiveness of load balancing. Furthermore, the SR implementations are written using both CM and CP styles so that performance comparisons can be made. This paper focuses on the distributed form of PCM style programs on a network of single processor machines.

The experimental performance results presented in this paper show that, for these load balancing applications, CM style programs outperform CP style programs in some cases but CP style programs outperform CM style programs in some other cases. This paper provides insight into these tradeoffs between CM and CP, and also into algorithms for practical load balancing.

The rest of this paper is organized as follows. Section 2 discusses background material. Section 3 describes the Dynamic Dining Philosopher application developed using various load balancing models. Section 4 presents the experimental performance results. Section 5 discusses these results. Section 6 presents ideas for future work and the conclusions reached through this research. Further details of this research can be found in Reference [11].

2. BACKGROUND

2.1. SR Overview

The SR programming language [1] provides support for both CP and CM programming styles. Furthermore, it provides intuitive methods to dynamically create threads and virtual machines. SR uses resources as its key component. A resource is essentially a template from which resource instances can be dynamically created. Each resource may also spawn more threads and/or create additional resources. In addition, threads can communicate through synchronous and asynchronous message passing.

The standard SR implementation uses an iteration-counting approach to provide support for the implicit context-switching found in CP. This approach allows a thread to execute for some number of loop iterations, based on the observation that if a thread is running and it does not block, then it must be looping. The SR compiler generates code to effect these implicit context switches. The SR implementation provides a compile-time option that inhibits this code generation, eliminating these implicit context switches. This allows the execution of CM style SR programs. A thread can use SR's **nap** function to effect the equivalent of an explicit yield statement (i.e., to force a context switch) in a CM style SR program.

The classic Dining Philosophers problem involves a number of philosophers gathered around a table. Each philosopher spends time either eating or thinking. In order to eat, a philosopher must grab both its left fork and its right fork. These forks, however, are shared with a philosopher's left and right neighbors. These forks must be obtained in a way that guarantees exclusive access (e.g., a semaphore). As a result, no two neighboring philosophers may eat at the same time. Figure 1 illustrates a philosopher from Dining Philosophers written in both

```
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 1. Code for a philosopher in Dining Philosophers

CP and CM styles. This version of the Dining Philosophers problem presents a decentralized solution. In the CP version, philosophers acquire forks through semaphores. In the CM version, philosophers acquire forks using shared variables. Each philosopher, except for one, grabs its left fork first and its right fork second. To prevent deadlock due to a circular wait condition, an asymmetric philosopher is introduced. This philosopher grabs its right fork first and its left fork second.

2.2. Dynamic C Overview

Dynamic C [4] 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 instruction 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.

cofunc functions are useful when multiple **costate** statements wish to execute the same code. These are the only kind of functions that can use **yield** or **waitfor** statements. However, to allow multiple calls to a **cofunc** function, multiple instances must be declared. When the function is called, the caller must designate which instance it is using. Calling an instance of a **cofunc** function that is already in use will cause the earlier call to be terminated. Instances of **cofunc** function must be called within an **wfd** (wait for done) statement. The **wfd** statement may have multiple **cofunc** functions as parameters. This statement yields if any of the **cofunc**

```
#define T 1000      // number of "eat think sessions"
#define N    5      // number of philosophers
int fork[N];
cofunc void philosopher[N](int right, int left){
        int i;
        for(i = 1; i <= T; i++){
                // grab forks
                while(fork[left] == 0){ yield; }
                fork[left] = 0;
                while(fork[right] == 0){ yield; }
                fork[right] = 0;
                // eat
                // release forks
                fork[left] = 1;
                fork[right] = 1;
                //think
        }
}
main(){
        int i;
        CoData philosopher0, philosopher1, philosopher2,
               philosopher3, philosopher4;
        // start the philosophers
        CoBegin(philosopher0);
        CoBegin(philosopher1);
        CoBegin(philosopher2);
        CoBegin(philosopher3);
        CoBegin(philosopher4);
        // initialize forks
        for(i = 0; i < N; i++){ fork[i] = 1 }
        // execute all of the threads
        for(;;){
                costate philosopher0{
                        wfd philosopher[0](0, 1);
                }
                costate philosopher1{
                        wfd philosopher[1](1, 2);
                }
                costate philosopher2{
                        wfd philosopher[2](2, 3);
                }
                costate philosopher3{
                        wfd philosopher[3](3, 4);
                }
                // the asymmetric philosopher
                costate philosopher4{
                        wfd philosopher[4](5, 4);
                }
        }
}
```

Figure 2. Dining Philosophers in Dynamic C

functions do not complete execution as a result of explicit yields. When the thread runs again, the **cofunc** functions resumes execution of all uncompleted **cofunc** functions.

Figure 2 illustrates a solution to the Dining Philosophers problem written in Dynamic C. It uses many features of Dynamic C, including those mentioned in this section.

3. DYNAMIC DINING PHILOSOPHERS

This paper uses an expanded version of the classic Dining Philosophers problem (see Section 2.1) that allows philosophers to periodically enter or leave the system. This version of the dining philosophers is typically referred to as the Dynamic Dining Philosophers (DDP)[5].

When a philosopher attempts to enter the system, it first will designate a new fork as its left fork. Next, it is informed of its new left and right neighbors. It then must introduce itself by informing both neighbors of its arrival. The left neighbor receives additional information about its new right fork, the fork that the incoming philosophers brings to the table. The new philosopher may enter the system only after these changes have been made.

When a philosopher attempts to leave the system, it must first inform each of its neighbors of its departure. The left neighbor receives additional information about its new right fork, which is the old right fork of the departing philosopher. Then, the departing philosopher disposes of its left fork and leaves the system.

To guarantee that the system never reaches an incorrect state, simultaneous arriving and departing philosophers should be serviced sequentially. This is to protect against an incoming philosopher attempting to communicate with a recently departed philosopher. Such a situation may cause the state of the system to become erroneous or even deadlock [5, 13].

To run under the PCM model, the set of philosophers is divided into N groups. Each group lies in its own region, which represents either a real or virtual processor. Since each region runs CM style code, forks that are on the interior of the region are obtained through shared variables. Forks that lie between regions, however, are obtained through distributed semaphores. For simplicity, those philosophers that lie on the border of a region and use semaphores to grab forks are considered immortal and are not allowed to leave the system. This simplification eliminates the code needed to deal with special cases that have one or zero philosophers present in a region. Such cases can cause the region boundaries to become unclear. Figure 3 illustrates the general setup for DDP using two regions.

3.1. General Synchronization Issues

As mentioned, philosophers are allowed to leave and enter the system, which requires synchronization to meet the following requirements:

Mutual Exclusion A philosopher brings or takes a fork as it arrives or leaves from the table. Thus, a philosopher's two neighboring philosophers must be notified before it attempts to enter or leave the system. This prevents the grabbing of the same fork simultaneously or grabbing an incorrect fork. To simplify matters, such notifications are only serviced during the neighbors' non-eating phases, i.e., obtaining forks is non-preemptive.

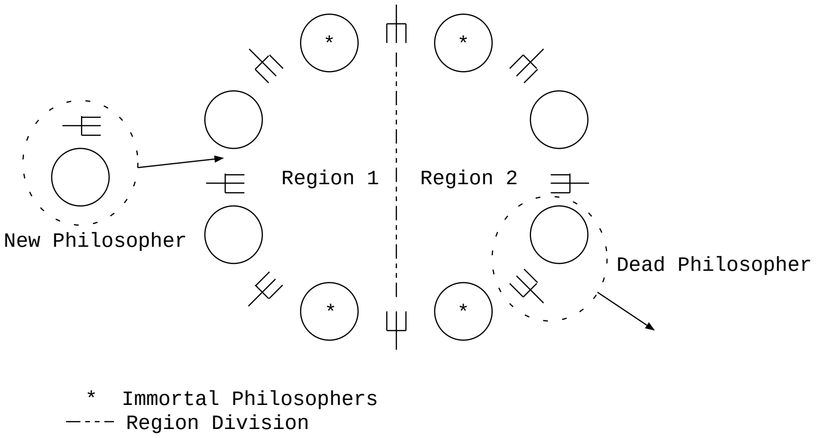

Figure 3. The Dynamic Dining Philosophers Model used to implement load balancing

Deadlock In the static model of Dining Philosophers, deadlock is avoided by having one asymmetric philosopher, which grabs its forks in the reverse order from other philosophers. Thus, a deadlock might occur if the asymmetric philosopher leaves the system. All remaining philosophers could grab their forks in the same order, which may lead to a circular wait. Currently, such a scenario is avoided by not allowing the asymmetric philosopher (one of the immortal ones) to leave. A less restrictive solution would be to reassign the asymmetric philosopher. To accomplish this, additional code would be needed to change fork grabbing methods of affected philosophers. This alternative, however, was not used since immortal philosophers already existed and designating one of these philosophers as asymmetric simplified the problem.

Starvation When a philosopher attempts to enter or leave the system, its two neighboring philosophers must pause in order to synchronize. If philosophers repeatedly enter and leave the system in the same location, then the neighboring philosophers might continually be trying to synchronize and never eat. Thus, a dirty bit is introduced to force an affected philosopher to eat and think at least once before attempting to synchronize.

3.2. Load Balancing Models

To introduce load balancing, an efficient algorithm was designed that will reduce overhead yet provide the necessary synchronization. Initially, moving philosophers from one region to the next was restricted to those philosophers that lie on the border. Thus, load balancing only considered whether moving a philosopher on the border of one region to the border of the other region would balance the workload. A more flexible approach allowed groups of consecutive philosophers to move across this border. But, unfortunately for both cases, the overhead of performing such a move proved too costly. Thus, we use the DDP model as the basis for implementing load balancing, utilizing its death and creation abilities to move a philosopher from one region to another. This model also allows the load balancer to move any mortal philosopher, adding to the algorithm's flexibility. Furthermore, synchronization for load balancing was provided in large part by DDP. One process on a separate processor handles all deaths, creations, and load balancing. The philosophers are run on two separate virtual machines (one for each region), each running on a separate processor.

A balanced system is determined by whether or not workloads on each region can be redistributed better by moving a philosopher. This information is gathered by having the balancing process send load request messages to each processor in the system. These messages are serviced by threads running on each processor. Once the balancer process has gathered system load information from each threador, load balancing potential is evaluated. Other research has shown this load information gathering scheme to be an effective method to achieve load balancing [6].

Two different load balancing models for DDP have been implemented. The first model balances by moving the philosopher that has the smallest amount of work remaining. This only occurs if moving this philosopher will yield a more balanced system. This model will be referred to as **DPLBsmall**. The second spends more time analyzing the status of each processor and determines the philosopher that has the largest amount of work remaining that could be moved to yield the most balanced system. This approach requires more time because all possible moves and their resulting workload distribution must be evaluated before making a decision. This load balancing model will be referred to as **DPLBlarge**. The non-load balancing model will be referred to as **DPNLB**.

3.2.1. Synchronization Issues for Load Balancing

To ensure that the system never reaches a bad state due to load balancing, some synchronization issues must be examined.

Mutual Exclusion A fork cannot be grabbed by two philosophers simultaneously due to load balancing since the algorithm uses DDP's death and creation procedures.

Deadlock Currently, the four philosophers aligned along the border never die or move. Since one of these is the asymmetric philosopher, this avoids the potential deadlock situation as described in Section 3.1. It also means that how a given philosopher grabs its fork is fixed (always by semaphores or always by using shared variables). Specifically, border philosophers grab their interior forks using shared variables and their forks shared with philosophers in other regions using semaphores. Interior philosophers always use shared variables to acquire forks.

Starvation A philosopher could be starved if it is moved from one region to the next repeatedly such that it never gets to eat. To avoid this, we take advantage of the dirty bit so that a philosopher cannot change regions until it cycles through at least one iteration of eating and thinking.

4. EXPERIMENTAL PERFORMANCE RESULTS

4.1. Experimental Setup

This section describes the general setup for the experiments including a description of the parameters that define each test. The parameters discussed provide the same functionality in both SR and Dynamic C versions. As described later in this section, the experiments were run using SR on a network of workstations running Linux and using Dynamic C on a network of stand-alone embedded Rabbit systems (see Section 1). Because SR runs only on Linux and Dynamic C runs only on Windows systems and Rabbits, we cannot run experiments to evaluate Dynamic C on a network of Linux systems or SR (directly) on a network of Rabbits. Our work on DesCaRTeS (described in Section 4.3) is an attempt to provide SR-like functionality on a network of Rabbits.

Threads in all experiments are executed on three different processors. One processor handles thread creation, thread synchronization, and load balancing. The other two processors represent regions as described in Section 3. Our experiments are limited to three procesors, with two regions, because we had only three Rabbit processors. (See Section 5 for further discussion.)

The experiments used eight parameters as defined in Table I. Figure 4 illustrates the use of NON_CS_WORK, CS_WORK, and WORK with code fragments written in SR. As described in Section 3.2, the load balancer uses "remaining work" in deciding when to move a thread. In the code, that quantity corresponds to how many total iterations of all loops remain for the given thread. A philosopher leaves the system when it has completed all its work. The loop bounded by WORK is nested within the two separate loops bounded by NON_CS_WORK and CS_WORK. Under PCM, an explicit context switch is placed immediately after this inner loop finishes. This context switch makes PCM more comparable to CP. Without it, PCM would simply execute all work loops until completion. CP, however, would perform implicit context switches for WORK values greater than or equal to 10000. This is the value for which the SR implementation performs implicit context switches (see Section 2.1). The data presented for the experiments represent averages of multiple executions.

Two sets of experiments using different parameters were conducted for DDP. Table II shows the values of the parameters used for each experiment.

4.2. SR Implementations

Three Pentium class machines were used for the SR tests. Initial tests revealed that the programs running on relatively equivalent machines remained balanced during the duration of their executions. Thus, to provide an environment that required load balancing, both regions for each experiment were placed on machines with significantly different performance capabilities. Region 1 was run on a 200 MHz Pentium Pro. Region 2 was run on a 1400 MHz Pentium 4. The main resource, which included the load balancer, was run on an 866 MHz Pentium 3. These machines run Red Hat Linux 7.0 communicating on a 100 Mb ethernet network.

Table I. Parameter definitions for SR experiments

Parameter	Description
N	Limit on the number of threads allowed in a region.
TEST_NUM	Number of threads created during a test. Test completes when all of these threads terminate.
ITER	Upper bound on the number of iterations a thread executes during its lifetime. Actual value is randomly selected between 1 and this value.
NON_CS_WORK	Number of iterations the loop in the non-critical section of a thread's code executes.
CS_WORK	Number of iterations the loop in the critical section of a thread's code executes.
WORK	Upper bound of inner loop. Under CM, context switches appear after this loop.
CREATE_TIME	The upper bound on the number of seconds between thread creations. Actual time is randomly selected between 1 and this value.
BALANCE_TIME	Time, in seconds, between load balancing attempts.

Table II. Parameters for SR experiments

Parameter	Experiment 1	Experiment 2
N	20	20
TEST_NUM	50	50
ITER	100	1000
NON_CS_WORK	100	10
CS_WORK	100	10
CREATE_TIME	10	20
BALANCE_TIME	2.5	5

```
# iter is a randomly generated integer
# between 1 and ITER
do iter > 0 ->
        ...
        # think
        fa i := 1 to NON_CS_WORK ->
                fa j := 1 to WORK -> af
                nap(0) # PCM version only
        af
        ...
        # grab forks
        # eat
        fa i := 1 to CS_WORK ->
                fa j := 1 to WORK -> af
                nap(0) # PCM version only
        af
        # release forks

        iter--;
od
```

Figure 4. Code fragments in SR illustrating use of test parameters for DDP

4.2.1. Load Balancing

This section presents the results of DDP when comparing load balancing and non-load balancing models. Tables III, IV, V, and VI present data for DDP. Figures 5, 6, 7, and 8 show these data graphically. All axes use a log scale. **DPLBsmall** and **DPLBlarge** both perform better than **DPNLB** for larger workloads (see Section 3.2 for definitions of models). That is, in these tests, load balancing improves performance despite its extra overhead. However, there is only a negligible difference in performance between the two load balancing models. Tables VII and VIII show the performance ratios of load balancing models over the non-load balancing model for both experiments. This ratio is defined by the following equation:

$$\frac{\text{time for load balancing model}}{\text{time for non-load balancing model}}$$

The trends illustrated by the data can be explained as follows. Let C be the average time in milliseconds between philosopher creations. Now let $T = C * N$, where N is defined as in Table I. If the average lifetime of a philosopher is much less than T, then the system generally has few, if any, philosophers present at any time. In this state, illustrated by workloads of 100 and 1000, load balancing is ineffective since there is not enough workload to cause the system to become uneven. If the average lifetime is about T, then the system will have a moderate amount of philosophers present. This state occurs for workloads between 10000 and 100000. Moderate loads benefit from load balancing since a large enough workload is present to benefit from redistribution, but the number of philosophers alive is not so great that the

Prepared using cpeauth.cls

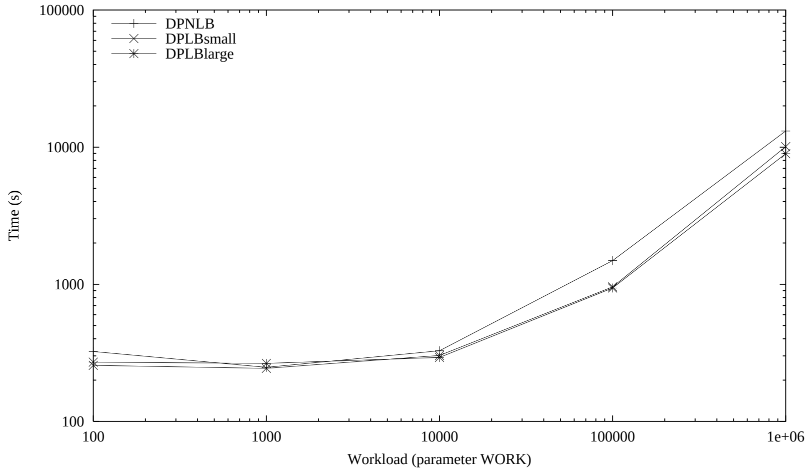

Figure 5. SR DDP under PCM (Experiment 1, axes are in log scale)

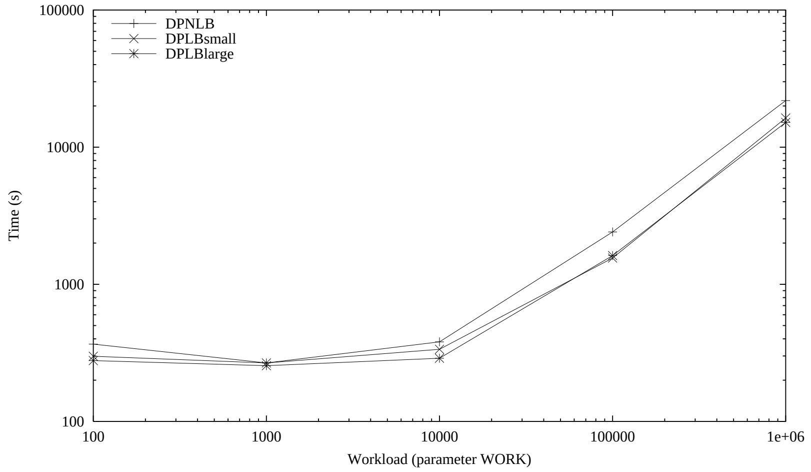

Figure 6. SR DDP under CP (Experiment 1, axes are in log scale)

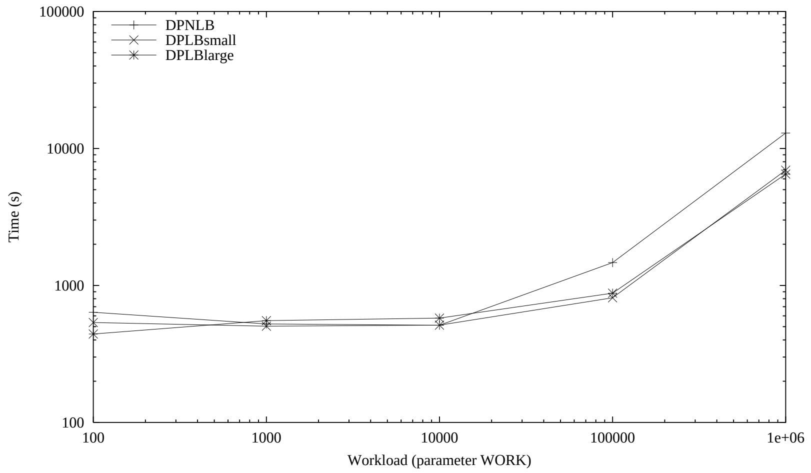

Figure 7. SR DDP under PCM (Experiment 2, axes are in log scale)

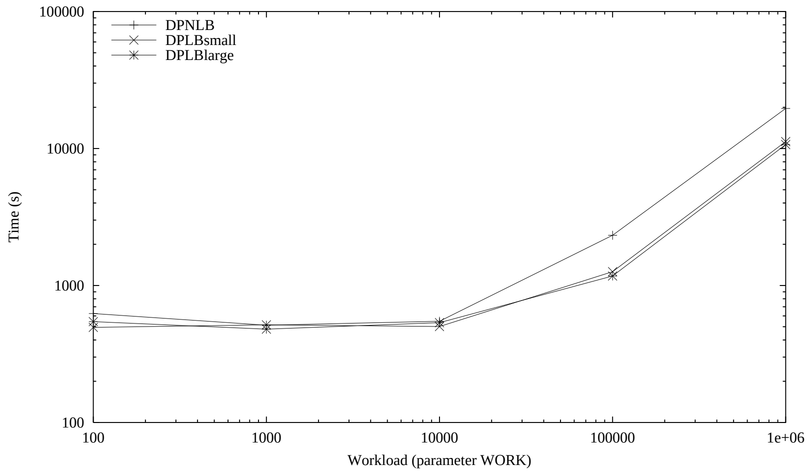

Figure 8. SR DDP under CP (Experiment 2, axes are in log scale)

Table III. SR DDP under PCM (Experiment 1; time in secs.)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	323.5	256.3	270.5
1000	247.9	243.5	264.7
10000	327.5	302.5	292.6
100000	1487.1	957.2	939.4
1000000	13101.0	10079.0	8951.1

Table IV. SR DDP under CP (Experiment 1; time in secs.)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	366.3	298.8	277.8
1000	267.1	267.1	254.9
10000	380.4	335.8	289.0
100000	2410.6	1555.1	1621.1
1000000	21878.9	16335.4	15152.1

Table V. SR DDP under PCM (Experiment 2; time in secs.)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	637.3	536.3	442.0
1000	523.7	504.0	553.1
10000	511.9	512.7	578.0
100000	1467.5	813.4	878.8
1000000	12953.4	6938.4	6504.4

regions become full. For average lifetimes much greater than T, the system becomes heavily impacted (i.e., each region typically has N philosophers present at any given time), leaving no room for the load balancer to move philosophers across regions. In this state, shown by WORK values between 100000 and 1000000, load balancing loses its effectiveness. Gains for these values are achieved only during the beginning and ending stages of the experiment, when the number of philosophers is just starting to grow or diminish, respectively.

Successful load balancing is defined as the load balancer attempting and accomplishing a move of one thread from one region to another. More specifically, the load balancer first evaluates whether or not moving a thread would yield a more balanced system. Then, the load

Table VI. SR DDP under CP (Experiment 2; time in secs.)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	624.7	494.6	544.5
1000	512.8	516.4	480.2
10000	548.2	502.2	535.6
100000	2327.1	1262.4	1174.1
1000000	19616.0	11241.5	10684.3

Table VII. Performance ratios for DDP using load balancing (Experiment 1)

WORK	DPLBsmall/ DPNLB (PCM)	DPLBlarge/ DPNLB (PCM)	DPLBsmall/ DPNLB (CP)	DPLBlarge/ DPNLB (CP)
100	0.7924	0.8362	0.8156	0.7584
1000	0.9822	1.0678	1.0001	0.9545
10000	0.9238	0.8934	0.8827	0.7597
100000	0.6437	0.6317	0.6451	0.6725
1000000	0.7693	0.6832	0.7466	0.6925

Table VIII. Performance ratios for DDP using load balancing (Experiment 2)

WORK	DPLBsmall/ DPNLB (PCM)	DPLBlarge/ DPNLB (PCM)	DPLBsmall/ DPNLB (CP)	DPLBlarge/ DPNLB (CP)
100	0.8415	0.6935	0.7918	0.8716
1000	0.9624	1.0561	1.0069	0.9364
10000	1.0015	1.1291	0.9160	0.9771
100000	0.5543	0.5989	0.5425	0.5045
1000000	0.5355	0.5021	0.5731	0.5447

Table IX. SR DDP load balancing success percentages (Experiment 1)

WORK	DPLBsmall (PCM)	DPLBlarge (PCM)	DPLBsmall (CP)	DPLBlarge (CP)
100	9.33	8.64	0.95	0.35
1000	10.95	12.71	1.59	2.04
10000	24.69	36.40	23.63	38.19
100000	26.27	31.58	14.66	22.83
1000000	4.84	5.46	2.09	3.25

balancer checks to make sure that the destination region of a thread migration has enough available space to hold another thread. When both of these conditions are met, the balancer proceeds to move a thread. Once started, a thread migration never fails since a slot in the destination region is reserved before any other threads are created or destroyed. Load balancing failure occurs when the destination region contains the maximum number of threads (defined by **N**) or the system is already balanced. In this case, the load balancer aborts and retries again later. Tables IX and X show the percentages of successful load balancing on all load balancing DDP tests. These percentages are calculated as the the number of actual thread migrations over the total number of load balancer attempts and multiplied by 100. These percentages illustrate the success of load balancing under different workloads. Generally, load balancing is more successful for larger workloads. The sudden drop in these percentages for WORK values of 1000000, however, can be explained in two ways. First, region populations are restricted to 20 philosophers. When the WORK increases to large enough values, the system becomes full. Thus, the load balancer still attempts to execute, but fails due to unavailable room to complete a move. Second, only a limited number of load balancing moves can be made during the execution. In our experiments Region 1 runs more slowly than Region 2. Thus, most philosophers are moved from Region 1 to Region 2. In the worst case (in which all philosophers are created on Region 1), the balancer would succeed at most 49 times. But since the workload is higher, the experiment will take longer to run. This causes the load balancer to execute more times. Thus, the percentage of successful load balancing will drop. Of course, a few philosophers are moved from Region 2 to Region 1. Thus, a realistic estimate for the number of load balancing successes could exceed 50.

4.2.2. PCM versus CP

Tables XI and XII illustrate the performance ratios of PCM versus CP. For smaller WORK values, CP and PCM have similar performance. For larger WORK values, PCM outperformed CP. More specifically, this gain in performance begins at some WORK value greater than 10000. This can be explained by the frequency of implicit context switches in SR, which occur every 10000 iterations of a loop (see Section 2.1). Thus, programs written in PCM will make

Table X. SR DDP load balancing success percentages (Experiment 2)

WORK	DPLBsmall (PCM)	DPLBlarge (PCM)	DPLBsmall (CP)	DPLBlarge (CP)
100	14.20	21.24	18.49	17.61
1000	12.04	10.80	11.76	17.02
10000	23.18	20.18	27.80	23.96
100000	26.42	43.02	22.63	39.82
1000000	5.21	11.02	2.56	6.32

Table XI. PCM vs. CP ratios for DDP (Experiment 1)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	0.8830	0.8578	0.9735
1000	0.9282	0.9115	1.0384
10000	0.8609	0.9010	1.0124
100000	0.6169	0.6155	0.5795
1000000	0.5988	0.6170	0.5908

Table XII. PCM vs. CP ratios for DDP (Experiment 2)

WORK	DPNLB	DPLBsmall	DPLBlarge
100	1.0203	1.0843	0.8117
1000	1.0212	0.9761	1.1518
10000	0.9338	1.0209	1.0791
100000	0.6306	0.6443	0.7485
1000000	0.6604	0.6170	0.6088

fewer context switches than their CP counterpart for WORK values exceeding 10000. That results in less total CPU time used to perform context switches and more CPU time used to perform real work.

4.3. Dynamic C Implementations

Three Rabbit 2000 TCP/IP Development Boards (see Section 1) were used for the experiment. Each processor has an 18 MHz clock speed and communicates on a 10 Mb ethernet network.

Table XIII. Parameters for Dynamic C DDP experiments

Parameter	Experiment 1	Experiment 2
N	10	10
TEST_NUM	30	30
ITER	10	100
NON_CS_WORK	100	10
CS_WORK	100	10
CREATE_TIME	10	20
BALANCE_TIME	2.5	5

The Dynamic C programs used in these experiments were written using the DesCaRTeS library. This library provides SR-like features for programs written in Dynamic C. The library code itself is written in Dynamic C (using CM style programming) and is targeted for programs running on Rabbits. The SR-like features that DesCaRTeS provides include inter-process communication via synchronous and asynchronous message passing, pseudo-dynamic thread creation, and virtual machine creation. The use of DesCaRTeS simplifies writing the programs for these experiments. For example, DesCaRTeS hides the details of communicating across the network from the programmer by providing higher-level communication functions; the code within these functions uses the TCP/IP library available with the Dynamic C distribution to provide such communication. Details about DesCaRTeS can be found in Reference [11].

Two sets of experiments using different parameters were conducted. Table XIII shows the parameter values used for each DDP experiment. Section 4.1 gives a description of each of these parameters. The values used for these experiments are smaller than those for SR (see Section 4.2) because of the differences in processor speeds and the limited amounts of memory. Both sets of experiments are run stand-alone.

Tables XIV and XV present data for DDP. Figures 9 and 10 show these data graphically. The x-axes representing WORK use a log scale. Tables XVIII and XIX show the performance ratios of load balancing versus non-load balancing models. In some cases, load balancing has only a small amount of benefit for smaller amounts of work. But as workload increases and the system begins to reach maximum capacity, load balancing gradually becomes less effective and begins to hinder performance due to overhead. In other cases, as in Experiment 1, increased work per iteration causes synchronization between philosophers and the load balancer to take longer amounts of time. This is due to the longer periods of time between possible times of synchronization, which occur only once per iteration. Thus, load balancing is never effective, and has a larger negative impact on performance for larger WORK values.

Tables XVI and XVII show the percentages of successful load balancing on all load balancing DDP tests. These percentages peak somewhere between WORK values of 10 and 50, and then begin to drop off. The reason for this drop off is the same as for the SR experiments and is explained in Section 4.2.1.

Table XIV. Dynamic C DDP (Experiment 1; time in secs.)

WORK	DPNLB	DPLBsmall	DPLBlarge
1	206.0	212.7	214.3
10	840.3	1211.0	958.7
50	3742.7	5349.3	4053.0
100	7167.0	9839.7	9316.3

Table XV. Dynamic C DDP (Experiment 2; time in secs.)

WORK	DPNLB	DPLBsmall	DPLBlarge
1	316.0	278.3	311.3
10	480.7	473.0	412.3
50	1183.7	1429.0	1139.3
100	2038.7	2446.3	2630.3

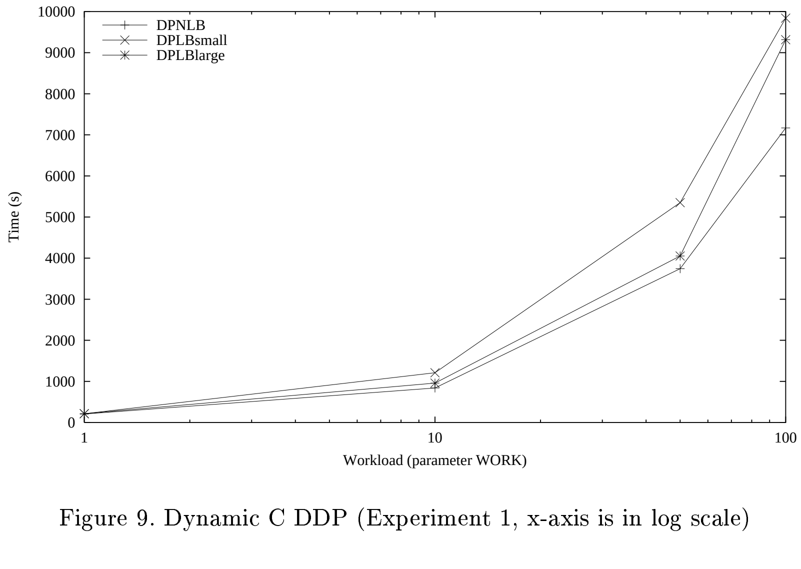

Figure 9. Dynamic C DDP (Experiment 1, x-axis is in log scale)

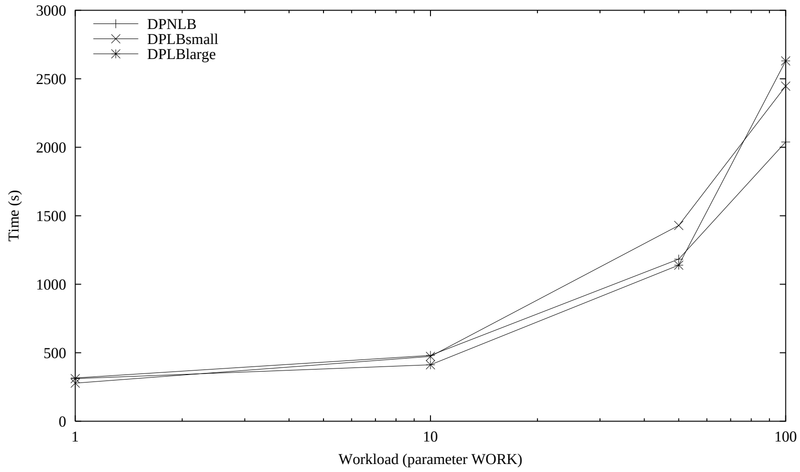

Figure 10. Dynamic C DDP (Experiment 2, x-axis is in log scale)

Table XVI. Dynamic C DDP load balancing success percentages (Experiment 1)

WORK	$DPLBsmall$	$DPLBlarge$
1	12.86	0.00
10	16.67	3.24
50	10.16	7.56
100	8.10	1.51

Table XVII. Dynamic C DDP load balancing success percentages (Experiment 2)

WORK	$DPLBsmall$	$DPLBlarge$
1	13.67	1.14
10	28.86	11.22
50	9.94	13.14
100	7.60	6.15

Table XVIII. Dynamic C performance ratios for DDP using load balancing (Experiment 1)

WORK	DPLBsmall/ DPNLB	DPLBlarge/ DPNLB
1	1.0324	1.0405
10	1.4411	1.1408
50	1.4293	1.0829
100	1.3729	1.2999

Table XIX. Dynamic C performance ratios for DDP using load balancing (Experiment 2)

WORK	DPLBsmall/ DPNLB	DPLBlarge/ DPNLB
1	0.8808	0.9852
10	0.9840	0.8578
50	1.2073	0.9625
100	1.2000	1.2902

5. DISCUSSION

This section discusses the results presented in Section 4. It also comments on differences between CM and CP style programming and differences between SR and Dynamic C and their respective run-time systems.

In addition to DDP, data for another application, Dynamic Readers/Writers (DRW), was also collected. Like DDP, DRW involved two regions, each containing at most N threads. Each thread represents a reader or writer and enters the system for a specified number of iterations. During its lifetime, a thread accesses a centralized database. A few load balancing models were applied to DRW. However, data for all models yielded results similar to those for DDP and are not explicitly given in this paper. This data can be found in reference [11].

5.1. Load Balancing

Load balancing is beneficial on moderately loaded systems where the workload per processor becomes uneven. Load balancing has negligible effectiveness on systems with light workloads, since little if any balancing can be done. On heavily loaded systems with limits on regional load carrying capacities, load balancing is ineffective and can even slow the system down. In such a case, no room exists to move threads from one region to another, but additional CPU time, messages, and network traffic are introduced by the load balancing algorithm. These effects were discussed for the SR experiments in Section 4.2.1 and for the Dynamic C experiments in Section 4.3.

The frequency of load balancing attempts should also be considered when developing a load balancing model. Too many attempts will incur excessive overhead with minimal gains resulting from any additional thread migrations. Too few attempts will not be effective since the system will spend more time in unbalanced states. Choosing an optimal time interval to run load balancing is system dependent and requires extensive study of the workload status of each processor during program execution. Moreover, the length of intervals between load balancing attempts could be determined by dynamic feedback. This involves calculating an appropriate interval length determined by statistics from previous load balancing attempts. This scheme, however, would also require extensive study to determine how to correctly calculate the dynamic interval lengths that would yield desirable performance results.

For load balancing to be effective in the applications examined, threads must perform CPU intensive tasks. Early tests revealed that when threads spend a majority of their lives idle, load balancing had negligible effects. When threads are told to sleep for a given amount of time, that time passes at the same rate on all processors. Thus, threads that spend the majority of their lives idle have execution times that are dominated by the time spent sleeping. This causes the processors, regardless of speed, to appear to execute threads at the same rate. Threads need also to be present in the system for long enough so that load balancing can redistribute the workload. If threads have short lives with high CPU intensive work, and the system on average has a large number of these threads present at any given time, then the load balancer cannot gather accurate information about the state of the system. The load balancer will try and move threads that have terminated (or are about to). This will significantly reduce the performance of the system since being moved prolongs the lives of threads.

Load balancing seems to be equally effective under both CM and CP. That is, as seen in Section 4.2.2, speed-up percentages due to load balancing for both experiments are nearly equivalent. Load balancing does not perform any work that would take advantage of CM style programming. Load balancing code executes until completion unless it is waiting to synchronize with another thread. Both styles of programming execute this code in the same manner, thus producing similar results.

Load balancing effectiveness over different applications, however, does vary. In the SR experiments, DDP benefitted more from load balancing than DRW did due to the differences between these two problems. In DDP, no philosopher can stall the system every time it eats*. DRW, however, is different. One writer, regardless of which region it lies in, can prevent all threads in the system from entering the critical section. While this is happening, all threads could potentially sit idle while waiting for their turns. This bottleneck occurs every time a writer enters the critical section and can cause a faster processor to execute at the same rate as the slower processor. A similar comparison in Dynamic C, however, cannot be made

*Technically, the only philosopher that can stall the entire system is the asymmetric philosopher mentioned in Section 3.1. This state is reached only when the asymmetric philosopher is eating and all other philosophers are involved in a near circular wait condition that is broken when the asymmetric philosopher puts down its forks. This state, however, is rarely reached.

due to the differences in the setup of the experiments. The DDP experiments allowed more threads per region and tested for a larger number of total threads created than did the DRW experiments. This would make load balancing performance in Dynamic C implementations of DDP and DRW incomparable.

Load balancing performance can also vary between different implementations. As discussed in Section 5.3, differences in Dynamic C, SR, and their respective run-time systems can cause algorithms to perform differently.

The overhead of performing load balancing appears to be low. In our tests, work dominates execution and idle time is near zero. Under light loads, the performance data are quite close. The basic overhead incurred by the load balancer periodically checking load status is very small and moves occur infrequently. Under moderate loads, moves occur more frequently and the overhead increases, mainly due to needing to synchronize philosophers. However, we believe these overall costs are relatively low too, in part because the load balancing is beneficial and the load balancer runs at reasonable intervals, not continually. Also, our observations have shown that moves do not result in "thrashing" (i.e., a philosopher repeatedly moving back and forth between regions without accomplishing any real work), in part because our load balancing algorithm guarantees that a philosopher will execute at least one iteration before being moved.

As noted in Section 4.1, our experiments were limited to two regions and three processors. The effect of having more regions is not clear and would require additional tests to determine with certainty. However, we would expect that the current load balancing algorithm would continue to be beneficial up to some threshold number of regions. Beyond that threshold, the amount of computation or communication required by the load balancing algorithm would outweigh the benefits and could result in worse overall performance. (Of course, the algorithm could be adjusted to perform load balancing less often, thus effectively increasing the threshold.)

5.2. CM vs. CP

This section compares CM to CP. It examines the general performance of CM compared to CP and discusses the advantages and disadvantages of program development using CM as an alternative to CP.

5.2.1. Performance

In general, CM outperforms CP when programs contain loops that execute for long periods of time. Under CM, longer loops may have fewer context switches (if any) than under CP. This allows threads to perform more work before giving up the processor. The end result yields fewer context switches during the execution of a program.

Fairness, however, must be considered when comparing CM to CP. Although CM performs the same amount of work as CP in less time, individual threads under CM spend more time waiting for the CPU than in CP. Of course, allowing threads to execute to completion or point of synchronization would yield the best results. But having threads spend long periods of time idle is undesirable if not unacceptable in some scenarios. Multitasking systems that service multiple users simultaneously, for example, must divide up the CPU time appropriately among the various users.

5.2.2. Program Development

Developing programs using CM has some advantages and some disadvantages when compared to CP. Although CM generally needs less overhead than CP to ensure properties such as mutual exclusion, a programmer must be careful to guarantee correct program execution under CM.

When developing large programs or libraries with multiple threads under CP, programmers often must use some method to guarantee exclusive access to variables stored in shared memory, e.g., semaphores, locks, or monitors. Programs written in CM, however, do not need such protection. Code segments that require exclusive access simply execute, without context switches, until they leave a critical section. As a result, libraries like DesCaRTeS (see Section 4.3) do not require additional code to provide exclusive access to shared memory. For example, the invocation queues, blocking queues, operation table, and global variables within DesCaRTeS could all be accessed by multiple threads simultaneously. All operations on these resources, however, do not yield until after completion of critical sections and are essentially atomic in nature. This was an advantage during the development of the library, making the algorithms much simpler to implement.

Programming under CM, however, does have its disadvantages. It may be difficult, for example, to know exactly when and where to include a yield. Failure to place these yields in certain areas may result in starvation. More specifically, a thread may enter a particular state and loop indefinitely while waiting for a certain condition, set by another thread, to become true. If a yield is not present inside the loop, then no other thread will execute and the program will fail to make progress. Yields may also be poorly placed, causing the program to enter a faulty state. Introducing a yield inside a critical section that relies on CM for mutual exclusion instead of condition testing would eliminate any guarantee of mutual exclusion. Thus, a programmer using CM has the responsibility of placing yields in appropriate locations that prevent starvation or deadlock, but guarantee mutual exclusion for critical sections.

5.3. SR vs. Dynamic C

As the results revealed, data between SR implementations and Dynamic C implementations was not always consistent. This can be explained through the differences between the two languages and the implementation of SR's run-time system and Dynamic C's DesCaRTeS. In addition, the algorithms for DDP and DRW on SR and Dynamic C had some small, and possibly significant, differences. Moreover, the hardware used for SR and Dynamic C are considerably different. This section discusses these differences and how they might affect the results.

Surprisingly, SR is more dynamic than Dynamic C. Specifically, SR provides dynamic creation of operations, resources, remote procedure calls, and threads. SR has this capability since it is able to dynamically allocate and deallocate memory. In Dynamic C programs, on the other hand, the maximum number of threads, variables, and other memory dependent components of a program must be known at compile-time. This is due to the Dynamic

C compiler's inability to provide memory management*. Furthermore, access to memory allocated dynamically (via the DesCaRTeS library) takes longer for DesCaRTeS: all allocated memory must be read or written using explicit function calls. This can occur frequently, since many of the run-time systems data structures require memory that is allocated dynamically.

Current implementations of DDP and DRW could be optimized to improve performance. In both DDP and DRW load balancing implementations, extra information is passed but not used during the periodic load information gathering. Thus, removing this extra information may lessen the amount of message passing and favor load balancing.

The SR implementations and Dynamic C implementations also used different hardware. Dynamic C's Rabbit processors (see Section 4.3) are much slower than the processors used by SR (see Section 4.2). Furthermore, SR implementations communicated on an ethernet that transmitted messages at a rate ten times faster than the ethernet used for Dynamic C implementations. Thus, Dynamic C tests were able to perform far less work than the SR tests were.

6. CONCLUSION

This research presented a comparison between various load balancing and non-load balancing applications. The applications developed involved variations of two classic problems: Dining Philosophers and Readers/Writers. These applications were implemented in both CM and CP styles. Furthermore, two different programming languages, SR and Dynamic C, were used to implement the CM versions. Tests conducted on these implementations allowed us to compare load balancing performance using different programming styles, languages, and platforms. Data collected from tests not presented in this paper revealed that in cases of very high or very low workloads, load balancing slightly hindered performance. In cases of moderate workload (as shown in this paper), both SR and Dynamic C implementations of load balancing generally performed well. Dining Philosophers implementations benefitted more from load balancing than Readers/Writers implementations. The exception to these conclusions is when threads have fewer opportunities to synchronize with the load balancer due to increased amounts of work per iteration. Such situations cause load balancing to yield unfavorable results. For larger workloads, CM outperformed CP. For small to moderate workloads, however, performance comparisons favored CP. This research also revealed that developing programs under CM can have both advantages and disadvantages. Programming under CM requires less code to provide properties such as mutual exclusion, but a higher level of responsibility is placed on the programmer to ensure that these properties are not removed by misplaced yields.

This research only examined a few load balancing applications. These applications could be further studied by varying a number of parameters. Different numbers of threads, values for workloads, and numbers of regions could all produce interesting results. As mentioned earlier, a variety of tests using various different parameters revealed that load balancing is not always

*Even though the DesCaRTeS library does provide memory management routines, these features are not integrated with the compiler and cannot be used to support features such as dynamic thread creation.

favorable. Other classic problems — such as Producer/Consumer, Jacobi Iteration, or Barnes-Hut — could also be studied to reveal possible benefits and drawbacks of load balancing or CM style programming.

ACKNOWLEDGEMENTS

Gene Fodor provided valuable technical support on this work. Matthew Farrens, Matthew Bishop, and the anonymous referees provided detailed, constructive 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. 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.
3. G. Cornell and C. Horstmann. *Core Java.* Sun Microsystems, Inc., Mountain View, CA, 1996.
4. Z World: Product Documentation Dynamic C for the Rabbit 2000, 2002. `http://www.zworld.com/products/dc/docs_dcpremier.html`.
5. G. Findlow and J. Billington. High-level nets for dynamic dining philosophers systems. In *Semantics for Concurrency. Proceedings of the International BCS-FACS Workshop*, pages 185–203, 1990.
6. M. Hamdi and C. Lee. Dynamic load balancing of data parallel applications on a distributed network. In *9th International Conference on Supercomputing*, pages 170–179, July 1995.
7. C. A. R. Hoare. Communicating Sequential Processes. *Communications ACM*, 21(8):666–677, August 1978.
8. Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995.
9. Aaron W. Keen, Takashi Ishihara, Justin T. Maris, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson. A comparison of concurrent programming and cooperative multithreading. *Concurrency and Computation: Practice and Experience*, 15(1):27–53, January 2003.
10. K. Li. Deterministic and randomized algorithms for distributed on-line task assignment and load balancing without load status information. In *ACM symposium on Applied Computing*, pages 613–622, February 1998.
11. Justin T. Maris. A Comparison of Concurrent Programming and Cooperative Multithreading under Load Balancing Applications. Master's thesis, University of California, Davis, Department of Computer Science, June 2002.
12. Rabbit Semiconductor Weather Station Online Demonstration, 2003. `http://demo.zweng.com`.
13. C. Sibertin-Blanc. Cooperative objects: Principles, use and implementation. *Concurrent OOP and PN, LNCS 2001*, pages 216–246, 2001.
14. B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, Inc., Upper Saddle River, New Jersey, 1999.