Spring 2011

EVALUATION OF CLASSICAL INTER-PROCESS COMMUNICATION PROBLEMS IN PARALLEL PROGRAMMING LANGUAGES

Arunesh Joshi
San Jose State University

EVALUATION OF CLASSICAL INTER-PROCESS COMMUNICATION PROBLEMS IN PARALLEL PROGRAMMING LANGUAGES

A Writing Project
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science

By
Arunesh Joshi
Spring 2011

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

_____ _____

Dr. Robert Chun Date

_____ _____

Dr. Soon Tee Teoh Date

_____ _____

Mr. Snehal Patel Date

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my advisor, Dr. Robert Chun, for providing his constant guidance and support throughout this project. I appreciate my committee members Dr. Dr. Soon Tee Teoh and Mr. Snehal Patel for their time and suggestions.

Special thanks to my dear friends, Priyadarshini Shanmuganathan and Payal Gupta, and to Mom and Dad for their encouragement and support during the completion of the project.

Also, I would like to thank and offer my regards to all of those who supported me in any way for completing my Master's project in Computer Science at San Jose State University.

ABSTRACT

EVALUATION OF CLASSICAL INTER-PROCESS COMMUNICATION PROBLEMS IN PARALLEL PROGRAMMING LANGUAGES

By Arunesh Joshi

It is generally believed for the past several years that parallel programming is the future of computing technology due to its incredible speed and vastly superior performance as compared to classic linear programming. However, how sure are we that this is the case? Despite its aforesaid average superiority, usually parallel-program implementations run in single-processor machines, making the parallelism almost virtual. In this case, does parallel programming still remain superior?

The purpose of this document is to research and analyze the performance, in both storage and speed, of three parallel-programming language libraries: OpenMP, OpenMPI and PThreads, along with a few other hybrids obtained by combining two of these three libraries. These analyses will be applied to three classical multi-process synchronization problems: Dining Philosophers, Producers-Consumers and Sleeping Barbers.

TABLE OF CONTENTS

INDEX OF FIGURES, GRAPHS, LISTING AND TABLES

INTRODUCTION 1.0

As the speed of new processor technologies continues to grow, so do the programming techniques that would obtain the most out of them. Multi-core processors have followed this trend and even work stations with several physical processors promise to deliver higher performance rates than their predecessors.

Recently, multi-core processors have become very popular. Multi-core processing is now a trend in the growing technology industry, as single core processors have reached the physical limits of possible complexity and speed. Programmers can make use of these multi-core processors by developing parallel programs. Multiprocessing is defined as "the coordinated processing of programs by more than one computer processor" [11]. It is a general term that can be used to describe the dynamic assignment of a program to one or more computers working in tandem, or it can involve multiple computers working on the same program in parallel.

Multiprocessing can be either asymmetric or symmetric. These terms refer to how the operating system divides tasks between the processors in the system [15] [20]. Asymmetric multiprocessing designates some processors to perform only system tasks, and others to only run applications. This is a rigid design that results in a loss of performance during the times when the computer needs to run many system tasks and no user tasks, or vice versa. Symmetric multiprocessing, often abbreviated as SMP, allows either system or user tasks to run on any processor, which is more flexible and therefore leads to better performance. Most multiprocessing PC motherboards use SMP nowadays.

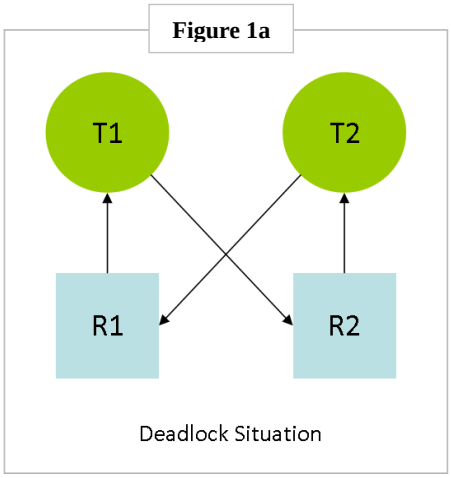
Two or more threads operate simultaneously in a multithreaded program. These threads communicate with each other using synchronization calls. If two or more threads try to access the same memory location without any interfacing synchronization calls, a race condition occurs. Due to the non-deterministic behavior of the multithreaded programs, data races are considered program **errors**, which are most difficult to find and debug. Even if we run the program with the same inputs; data races are difficult to reproduce. Data races do not crash the program immediately, but they corrupt the existing data structures. Data races may even cause system failures in some unrelated codes. Automatic race detection is a high priority research problem for the shared memory multithreaded programs. Multiprocessing and multithreading can be effective if the computer system has a suitable operating system and motherboard support which utilizes a motherboard that is capable of handling multiple processors or a processor with multiple cores or a processor that can handle multiple threads.

1.1 DEADLOCKS

In a multiprogramming environment, several processes may compete for a finite number of resources. If the resources are not available at the time they are requested, the process enters a waiting state. It is sometimes the case that some waiting processes may never change their state because other waiting processes hold the resources they have requested. This situation is referred to as **deadlock**.

A deadlock occurs when two or more tasks permanently block each other by virtue of each task having a lock on a resource, which the other tasks are simultaneously trying to lock. Figure 1a represents a high-level view of a deadlock state where:

1. Task T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).
2. Task T2 has a lock on resource R2 (indicated by the arrow from R2 to T2) and has requested a lock on resource R1 (indicated by the arrow from T2 to R1).
3. Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists.



In order for a deadlock to occur, four conditions must apply:

1. Mutual Exclusion - Each resource is either currently allocated to exactly one process or it is available. (Two processes cannot simultaneously control the same resource or be in their critical sections).
2. Hold and Wait - Processes currently holding resources can request new ones.
3. No Preemption - Once a process holds a resource, another process or the kernel cannot take it away.
4. Circular Wait - Each process is waiting to obtain a resource, which is held by another process.

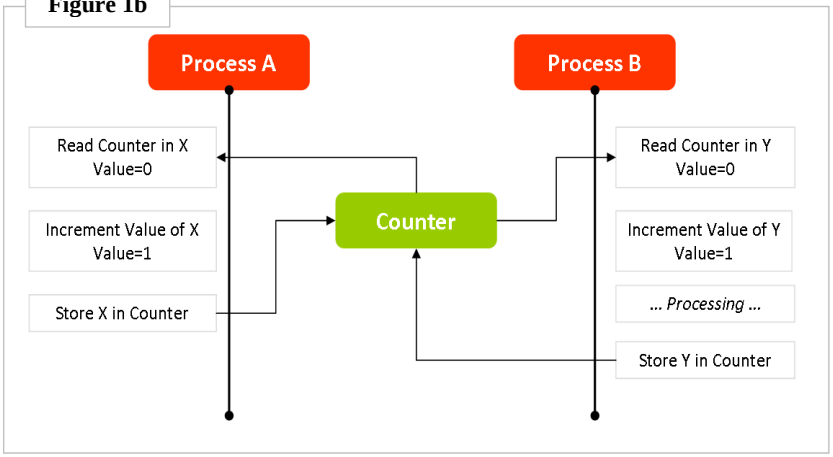
Another method of avoiding deadlocks is to require additional information about how the resources are to be requested. With the knowledge of the complete sequence of requests and releases for each process, we can decide for each one whether or not the process should wait.

The simplest and most effective method requires that each process declares the maximum number of resources of each type it may need. Given prior information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never reach a deadlock state [10].

1.2 RACE CONDITION

A race condition is the state in which one or more processes or threads attempt to concurrently modify a shared resource. This results in a general failure of the application causing the shared resource to have an incorrect or non-synchronized value.

Figure 1b



For example, suppose there are two processes A and B, both processes have their own tasks to perform and share a common resource, "Counter." One of their characteristics is that after some internal computations they have to increment the counter by one.

As shown in the figure 1b, if both processes are without any kind of restricted access to the same resource, and they write to it simultaneously without taking into consideration any previous modifications made by other processes, the result is usually undefined. But as in the example, the counter will end up having the value of "1," whereas it should have the value "2" because there has been one action on each process.

In the following sections I will attempt to define several methods that can be implemented to successfully overcome and prevent any kind of race conditions in parallel programming.

1.2.1 RACE CONDITION DETECTION TECHNIQUES

Analysis of a multithreaded shared memory parallel application is a difficult task. Due to the non-deterministic behavior of parallel application, it is difficult to find and debug errors. Even if the code is modified, it is difficult to make sure that the error is actually corrected and not concealed. Data race detection in parallel program is like an NP-problem. There are two approaches for race detection in multithreaded programs [4] [18].

1.2.1.1 STATIC ANALYSIS

Static analysis employs compile-time analysis on source programs. It finds all the execution paths in a program. The static analysis tool is known for finding low-level programming errors such as null pointer de-references, buffer overflows, use of uninitialized variables, etc [19]. For race detection, the static analysis tool tries to find data races at compile time. Finding data races in parallel programs using static analysis is very difficult. The Static Analysis Techniques are as follows:

Model Checking: The model of a system, like hardware or software systems, is tested automatically, irrespective of whether this model meets the specifications. The latter includes safety requirements so as to avoid deadlocks or race conditions.

Data-flow Analysis: This technique gathers information about possible sets of values calculated at various points in a computer program. A control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. Compilers, when optimizing a program, often use information gathered.

Advantages: In order to detect the errors, static analyzers do not run a program. A static analyzer does not depend on an execution environment.

Disadvantages: It is very difficult to apply static analysis to find race conditions in multithreaded programs. It is difficult to determine an interleaving between threads, so static analysis makes a conservative estimation. Scaling is also difficult to achieve using static analysis and produces false positives.

1.2.1.2 DYNAMIC ANALYSIS

Dynamic race detector analyzes the trace generated by execution of a parallel program. This trace contains memory accesses and synchronization operations done during the executions of a program. Dynamic analysis is totally dependent on the input dataset. So, it can be run more than once with a variety of datasets to assure the correctness of a program [21].

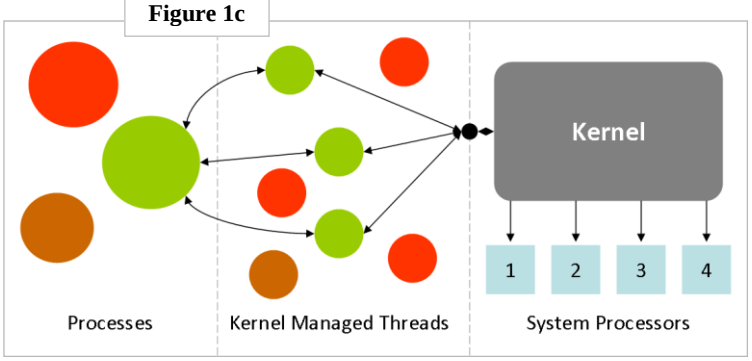
Advantages: Dynamic analysis generates fewer false positives as compared to the static analysis. Dynamic analysis provides the reasonable accurate race detection for multithreaded shared-memory parallel programs.

Disadvantages: Dynamic analysis analyzes traces of executed programs. So, it does not consider all the possible execution paths of a program. Dynamic analysis is not a sound predictor and race conditions can occur after dynamic analysis proves it to be correct. Dynamic analysis is also dependent on the execution of a program, so it has overhead on the execution of a program.

Henceforth it is assumed that the reader has previous knowledge of several topics, including threads, processors, processes, memory management, operating system, posix standards, C/C++ language, Inter-Process Communication (IPC), signals, Multi-Process Synchronization (MPS), mutual exclusion, multithreading, multiprocessing, distributed programming, deadlocks, locks / mutexes, race conditions, spin locks, critical sections, and semaphores.

1.3 PTHREADS

Figure 1c

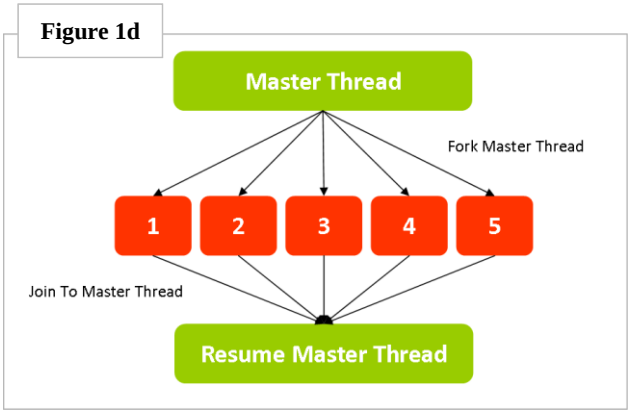


POSIX Threads is a standard library that defines a set of C language functions, constants, macros and types that allow the user to create and manipulate threads in a reliable way. It consists of almost one hundred procedures, which are categorized according to their use; these categories are: thread control, mutual exclusion, conditional variables and synchronization. As shown in figure 1c, a process can have multiple threads, and each thread is connected directly to the kernel, this latter will schedule the threads on different processors when available.

With Pthreads the user will be able to create threads without going through much trouble, and since this library is traditionally implemented at the kernel level (Pthreads is KLT – Kernel Level Threads), Pthreads are subject to the operating system and are able to take advantage of certain capabilities (if present) such as SMP (symmetric multiprocessing).

1.4 OMP

Figure 1d



Open Multi Processing is an application programming interface that supports shared memory multiprocessing in C/C++ and several other languages [9]. Unlike regular libraries in which the user has to invoke several library procedures, OpenMP is implemented as a set of compiler directives supported by languages, such as C/C++ and Fortran. This makes its use a lot easier, simpler and powerful [7]. As it is shown in Figure 1d, the master thread's execution is forked into five sub-tasks

running concurrently in parallel; once the sub-tasks (threads) are finished they join the master thread for continued execution.

OpenMP is an implementation of multithreading, "a method of parallelization whereby the master thread forks a specified number of slave threads and a task is divided among them" [8]. These threads run concurrently in the runtime environment which allocates threads to different processors. The tasks that are commonly forked by the master thread are loop language structures such as blocks: **while**, **do**, and **for** [1]. These forked blocks will run in parallel until all of them are finished; the master thread will then continue its execution.

1.5 MPI

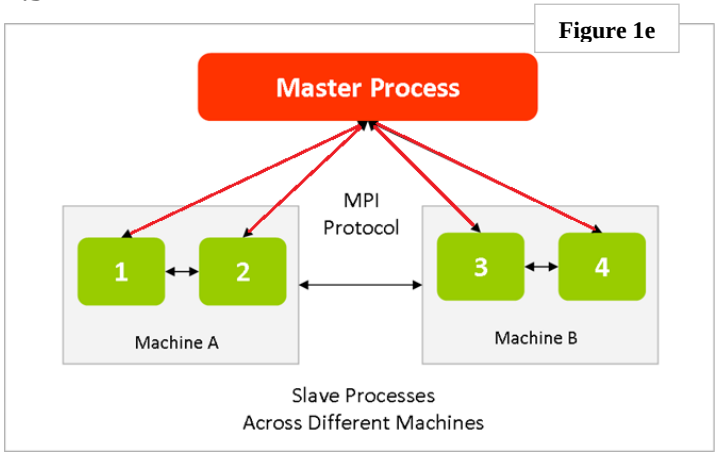


Figure 1e

Open Message Passing Interface or OpenMPI is an open source library project that combines technologies and resources from several other projects, and represents the merging of contributions from FT-MPI (University of Tennessee), LA-MPI (Los Alamos National Laboratory), LAM/MPI (Indiana University) and PACX-MPI (University of Stuttgart). These four institutions comprise of the founding members of the OpenMPI development team (vide www.lam-mpi.org or www.open-mpi.org).

Message passing interface (MPI) is an API specification that allows processes to communicate with one another by sending and receiving messages [6]. It's typically used for parallel programs running on computer clusters, as briefly described in figure 1e. MPI is a language-independent communication protocol, which is able to manage processes remotely and conveniently for distributed multiprocessing purposes.

The OMP, MPI and P-Threads libraries provide a great deal of advantage when parallel programming is the main objective. How each of these libraries was used in the testing subjects (the multi-process synchronization problems) and the test results are described in subsequent sections.

These three libraries let us successfully solve several IPC problems and avoid several error states such as an occurrence of a race condition. The race and deadlock conditions in a shared-

memory parallel program [2] are subtle and harder to find than in a sequential program. These conditions cause non-deterministic and unexpected results from the program and are difficult to find in a multithreading environment. In a sense, it is not easy to debug a multithreaded application, but there are ways to avoid these conditions while programming in multithreading languages, such as the ones mentioned above. Research on this topic reveals that there are many ways to write a race and deadlock free code in a multithreading environment, but each language or MP library has its own way of writing this code [4] [16].

The problems chosen in this study for implementation in these languages are three classical IPC problems: Dining Philosopher, Producer Consumer and Sleeping Barber. These problems have both deadlock and race conditions while they are executed. The goal is to code these problems in such a way that the deadlock and race conditions will not occur, and we can plot the performance matrices easily. Therefore, if a new programmer wants to use parallel programming language, he/she can choose the best one by using the results of performance matrices described in this study.

MPS PROBLEM DETAILS 2.0

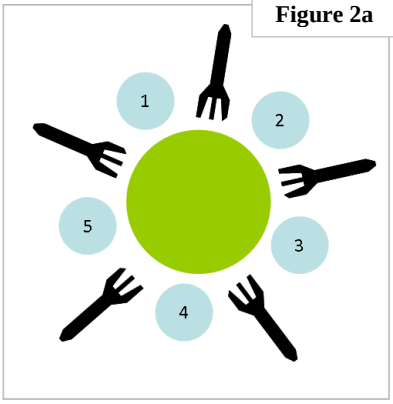
IPC or Inter-Process Communication is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory and remote procedure calls (RPC). The method of IPC used may vary based on the implementation library, bandwidth and latency of communication between the threads and the type of data being communicated. **MPS** or **Multi-Process Synchronization** is a set of techniques where one or more threads or processes communicate with one another using simple IPC methods, in order to accomplish a specified task.

MPS problems are simple statements that implicitly define a great deal of concurrency in which synchronization is a rather complex issue that must be correctly managed in order to fulfill the requirements of the problem. It also accomplishes the goals without errors of any kind. Classic MPS problems are Dining Philosophers, Producer-Consumer and the Sleeping Barber, all of which will be briefly introduced below.

2.1 DINING PHILOSOPHERS

This problem is generalized as N philosophers sitting at a round table doing one of two things only, either thinking or eating, but not both. Usually the problem is exemplified using five philosophers sitting at a circular table with a bowl of spaghetti for each one (shown as the cyan circles in figure 2a); a fork is placed between each philosopher. Each philosopher has one fork at their left, and one on their right. It is assumed that each philosopher will require two forks in order to eat; each subject can use only the forks immediately to his left and right. The problem states that the philosophers cannot talk to each other, imposing a possible deadlock if the problem is solved incorrectly. The solution to the problem is a state when all philosophers can eat and think concurrently, without leaving any philosopher to starve. This problem is solved by using locks for each fork and not entering a deadlock state by checking if both forks are free before taking them.

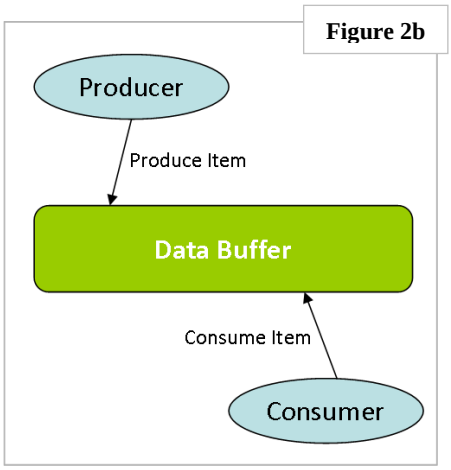
Figure 2a



2.2 PRODUCER AND CONSUMER

This problem, also known as a bounded-buffer problem, is a good example of MPS. It describes two processes, the producer (P) and the consumer (C); they share a common fixed size

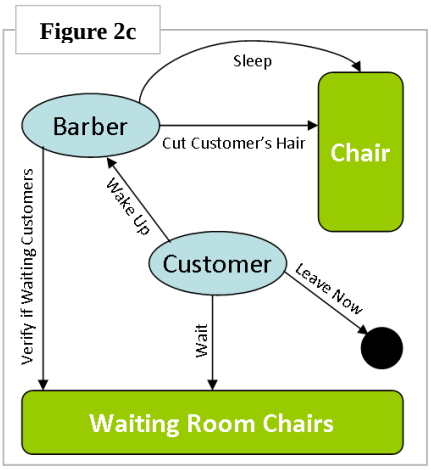
buffer, the producer's job is to generate an item of data and put it into the buffer, this is perpetual.



At the same time, the consumer is consuming the data items (i.e. removing an item from the common buffer) one by one. The problem states that no consumer (C) will attempt to remove data from an empty buffer, and no producer (P) will attempt to produce an item when the buffer is completely full. This example shows a practical way of learning to manage shared resources; in this case the common item is the buffer. The solution is achieved by means of a semaphore implemented as a counter between the range 0 to N, where N is the maximum length of the item buffer.

2.3 SLEEPING BARBER

This problem initially is stated using only one barber, and later on can be escalated to use more than one barber at the same barber shop. However primarily, the problem is stated as having one barber at one barber shop. This barber has one chair for cutting hair and a waiting room for customers with N chairs.



The barber finishes cutting a customer's hair and goes to check if there are any customers in the waiting room; if there are, he brings one of them to the chair, cuts his hair and goes to check for more customers. If there are no more customers waiting, he returns to a sleep state. Now, when another customer arrives, the customer looks at the barber to see what he is doing. If the barber is sleeping, this customer gladly walks to him, wakes him up, sits in the chair and gets his hair cut. If the barber was cutting hair, the new customer simply walks into the waiting room and waits, but if there are no empty chairs in the waiting room, he leaves.

Despite of its simplicity, it's very complicated to synchronize this barber shop to make it work perfectly all the time. The solution is to implement a simple mutual exclusion lock that will ensure that all the participants in one particular moment can change state only one at a time.

PTHREADS 3.0

Programming with PThreads presented no problems; all that is needed is to have the latest version of the PThreads library installed on your system. In the C-files it is required to include the "pthread.h" header file in order to use PThread's procedures and structures, and **remember** to always compile using the "-pthread" linker flag when using gcc, in order to embed the Pthread's library in your final executable program.

3.1 PTHREADS: DINING PHILOSOPHERS

This problem was implemented in PThreads by the use of one-thread for each philosopher on the table; each thread has a target of N-eats, i.e. the number of eats a thread has to perform before finishing its execution.

Given M-philosophers (M is a mandatory odd-number that cannot be even), an array of Forks using the type pthread_mutex_t is created for M elements. This array represents the forks on the table, and is described as mutexes since they are resources that need to be managed. M-threads are also created, these are the philosopher threads, and each of these threads will perform the following.

Code Listing 3a

```
1. INT Id = Current Thread Number
2. INT Num_Eats = 0
3. INT State = "THINKING"
4. INT Left = Id
5. INT Right = (Id + 1) MOD M
6. WHILE Num_Eats < N
    a. IF State = "THINKING" THEN
        i. IF pthread_mutex_trylock( Forks[Left] ) THEN
            1. IF pthread_mutex_trylock( Forks[Right] ) THEN
                a. State = "EATING"
            2. ELSE
                a. Pthread_mutex_unlock( Forks[Left] )
            3. END IF
        ii. END IF
    b. ELIF State = "EATING"
        i. pthread_mutex_unlock( Forks[Left] )
        ii. pthread_mutex_unlock( Forks[Right] )
        iii. Num_Eats = Num_Eats + 1
        iv. State = "THINKING"
    c. END IF
7. END WHILE
```

3.2 PTHREADS: PRODUCER AND CONSUMER

This problem was implemented in PThreads by using N threads for producers and M threads for consumers. T is the total number of items to produce and K is the length of the item buffer. Note that the execution of the program is finished once the number of produced items reaches T.

This Producer/Consumer problem was solved in PThreads by using one mutex (known as bufferLock) to protect a common resource (which in this case is the item buffer), and two index counters known as c_index for the consumer threads and p_index for the producer threads. The consumer threads will get elements from the buffer at c_index offset, and the producers will produce items at the p_index. Both threads increment and wrap-around their indices, when necessary. The pseudo-code implementation for this solution is shown below.

Code Listing 3b

```
INT Num_Items_Produced = 0
1.  THREAD PRODUCER
        a.  INT Id = Current Thread Number
        b.  WHILE Num_Items_Produced < T
                i.  pthread_mutex_lock( bufferLock )
                ii.  IF Num_Items_Produced < T THEN
                        1.  buffer[p_index] = Random Number (Item)
                        2.  p_index = (1 + p_index) MOD K
                        3.  Num_Items_Produced = 1 + Num_Items_Produced
                        4.  Items_Available = 1 + Items_Available
                iii.  END IF
                iv.  pthread_mutex_unlock( bufferLock )
        c.  END WHILE
2.  END THREAD
3.  THREAD CONSUMER
        a.  INT Id = Current Thread Number
        b.  WHILE Num_Items_Consumed < T
                i.  IF Items_Available = 0 THEN CONTINUE
                ii.  pthread_mutex_lock( bufferLock )
                iii.  IF Num_Items_Consumed < T AND Items_Available > 0 THEN
                        1.  Consume Item at buffer[c_index]
                        2.  c_index = (1 + c_index) MOD K
                        3.  Num_Items_Consumed = 1 + Num_Items_Consumed
                        4.  Items_Available = Items_Available - 1
                iv.  END IF
                v.  pthread_mutex_unlock( bufferLock )
        c.  END WHILE
4.  END THREAD
```

3.3 PTHREADS: SLEEPING BARBER

Given there are N chairs in the waiting room, B barbers threads, T haircuts to perform and C customers threads, the sleeping barber problem using PThreads was solved by one barber-state buffer (bsb) with B elements, and three mutex locks m1, m2 and m3. These locks are used as critical section sentinels to make sure certain areas remain single-threaded.

Code Listing 3c

```
1.  THREAD BARBER
2.  INT Id = Current Thread Number
        a.  WHILE Num_Customers_Serviced < T
                i.  IF NOT (bsb[Id] = "SLEEPING") THEN
                        1.  pthread_mutex_lock( m1 )
                        2.  IF NOT (Chairs_Occupied = 0) THEN
                                a.  Num_Customers_Services += 1
                                b.  Chairs_Occupied -= 1
                                c.  State = "READY"
                        3.  ELSE
                                a.  State = "SLEEPING"
                        4.  END IF
                        5.  pthread_mutex_unlock( m1 )
                        6.  bsb[Id] = State
                ii. END IF
        b.  END WHILE
3.  END THREAD
4.  THREAD CLIENT
5.  INT Id = Current Thread Number
        a.  WHILE Num_Customers_Serviced < T
                i.  pthread_mutex_lock( m2 )
                ii. FOR i = 0 TO B – 1
                        1.  IF bsb[Id] = "SLEEPING" THEN
                                a.  Num_Customers_Services += 1
                                b.  bsb[Id] = "READY"
                                c.  pthread_mutex_unlock( m2 )
                                d.  GOTO 2.a
                        2.  END IF
                iii. END FOR
                iv. pthread_mutex_unlock( m2 )
                v.  pthread_mutex_lock( m1 )
                vi. IF Chairs_Occupied < N THEN
                        1.  Chairs_Occupied += 1
                vii. END IF
                viii. pthread_mutex_unlock( m1 )
```

    b. END WHILE
6. END THREAD

OMP presented a few issues, primarily because the entire programming scheme had to be reconsidered in order to match the syntax and execution flow of a regular OMP program. The main feature of OMP is defined by forking the master thread and then re-joining it. In the latest OMP library for the C-language, one should include the "omp.h" header file in one's applications and remember to compile using the **–fopenmp** linker flag (when using gcc) in order to compile OpenMP programs. Another important thing to always remember is that OMP requires a higher level of anti-race condition code. [17].

4.1 OMP: DINING PHILOSOPHERS

This problem was implemented in OMP exactly as in PThreads, with the only exception that each PThread function call was replaced by its respective OMP C-preprocessor directive. This is because OMP works with preprocessor directives rather than function calls. It makes it easier to use and escalate. Refer to the section 3.1 to understand the meaning of certain variable names used in this code listing.

Some of the issues encountered while using OMP are those related to synchronization of global variables. The reason is that OMP provides a realistic parallel execution of each thread. The global variables shared among the threads have to be protected by using the **#omp critical** or **#omp atomic** directives in order to prevent a race condition. The following pseudo code listing explains the overall operation of the philosopher's thread in the main program. The OMP directives are kept as-are to avoid confusion.

Code Listing 4a

```
1. #pragma omp parallel shared(Num_Eats, N, M, Forks) private(Id,State,Left,Right)
2. INT Id = omp_get_thread_num ()
3. INT State = "THINKING"
4. INT Left = Id
5. INT Right = (Id + 1) MOD M
6. WHILE Num_Eats < N
    a. IF State = "THINKING" THEN
        i. IF omp_test_lock ( Forks[Left] ) THEN
            1. IF omp_test_lock ( Forks[Right] ) THEN
                a. State = "EATING"
            2. ELSE
                a. omp_unset_lock ( Forks[Left] )
            3. END IF
        ii. END IF
    b. ELIF State = "EATING"
        i. omp_unset_lock ( Forks[Left] )
```

ii. omp_unset_lock (Forks[Right])
iii. #pragma omp atomic
iv. Num_Eats = Num_Eats + 1
v. State = "THINKING"

c. END IF

7. END WHILE

4.2 OMP: PRODUCER AND CONSUMER

Due to the nature and direct relation of PThreads and OMP (as both libraries provide multithreading mechanisms), this problem was solved in a similar way as in PThreads, with the obvious differences of more efficient anti-race condition codes. Refer to section 3.2 to understand several variables shown in the code listing below.

Code Listing 4b

```
INT Num_Items_Produced = 0
1. FUNCTION MAIN
2. omp_set_num_threads ( N + M )
3. #pragma omp parallel private(Id)
    a. WHILE Num_Items_Produced < T OR Num_Items_Consumed < T
    b. Id = omp_get_thread_num ()
    c. IF Id < N THEN
        i. PRODUCER(Id)
    d. ELSE
        i. CONSUMER(Id)
    e. END IF
    f. END WHILE
4. END FUNCTION

5. FUNCTION PRODUCER ( Id : INT )
6. #pragma omp critical (GCS)
    a. IF Num_Items_Produced < T THEN
        i. buffer[p_index] = Random Number (Item)
        ii. p_index = (1 + p_index) MOD K
        iii. Num_Items_Produced = 1 + Num_Items_Produced
        iv. Items_Available = 1 + Items_Available
    b. END IF
7. END THREAD

8. FUNCTION CONSUMER ( Id : INT )
9. #pragma omp critical (GCS)
    a. IF Items_Available = 0 OR Num_Items_Consumed >= T THEN RETURN
    b. Consume Item at buffer[c_index]
```

```
        c. c_index = (1 + c_index) MOD K
        d. Num_Items_Consumed = 1 + Num_Items_Consumed
        e. Items_Available = Items_Available - 1
10. END THREAD
```

As it can be seen, the PThreads' buffer mutex was replaced in OMP by using a global critical section (GCS). GCS is used to prevent each thread from modifying the global state variables and/or the items buffer, thus preventing a race condition.

4.3 OMP: SLEEPING BARBER

This was solved by using one barber-state buffer (bsb) with B elements, and two critical sections to protect the barber-buffer and the chairs-buffer. The solution is similar to the one created for the PThreads problem. Refer to section 3.3 for more information about several variables used in the following pseudo-code listing.

Code Listing 4c

```
1. FUNCTION MAIN
2. omp_set_num_threads ( B + C )
3. #pragma omp parallel private(Id) shared(Num_Customers_Serviced)
        a. WHILE Num_Customers_Serviced < T
        b. Id = omp_get_thread_num ()
        c. IF Id < B THEN
                i. BARBER(Id)
        d. ELSE
                i. CUSTOMER(Id)
        e. END IF
        f. END WHILE
4. END FUNCTION

5. FUNCTION BARBER ( Id : INT )
6. IF Num_Customers_Serviced < T THEN
        a. #pragma critical ( CBSS )
        b. IF NOT (bsb[Id] = "SLEEPING") THEN
                i. #pragma critical ( CCSS )
                        1. IF NOT (Chairs_Occupied = 0) THEN
                                a. Num_Customers_Serviced += 1
                                b. Chairs_Occupied -= 1
                                c. State = "READY"
                        2. ELSE
                                a. State = "SLEEPING"
                        3. END IF
```

```
                4. bsb[Id] = State
        c. END IF
7. END FUNCTION

8. FUNCTION CUSTOMER ( Id : INT )
9. IF Num_Customers_Serviced < T THEN
    a. #pragma critical ( CBSS )
        i. FOR i = 0 TO B – 1
            1. IF bsb[Id] = “SLEEPING” THEN
                a. Num_Customers_Serviced += 1
                b. bsb[Id] = “READY”
                c. RETURN
            2. END IF
        ii. END FOR
    b. #pragma critical ( CCSS )
        i. IF Chairs_Occupied < N THEN
            1. Chairs_Occupied += 1
        ii. END IF
10. END FUNCTION
```

The CCSS (Check Chair State Section) and CBSS (Check Barber State Section) are critical areas that are used to protect the two most important global variables: the barber state and the chair state.

MPI 5.0

MPI, also known as Message Passing Interface, is the distributed-parallel-programming library used for the next test. One will require the latest MPI library to be installed on the system, if one is using Linux, an original implementation of MPI, such as LAM (www.lam-mpi.org) or OpenMPI (www.open-mpi.org) to prevent any problem. If Windows is being used, the only reliable library is DeinoMPI (mpi.deino.net). Compile using "mpicc –g source.c", the "-g" option. This allows the compiler to attach debugging information, just in case one wants to debug the program. In order to run an MPI program in a single machine the "mpirun" command should be issued indicating the number of processes to run the program binary code, e.g. "mpirun -np 18 a.out", this example specifies to run "a.out" in 18 processes.

MPI imposes several difficulties when it comes down to synchronizing the processes using the provided message interface. Since this communication channel is somewhat slow, it causes the application's response time to be severely affected.

5.1 MPI: DINING PHILOSOPHERS

In the implementations of this problem described so far we had the ability to use a shared-buffer to describe the fork-states. Using this scheme, we were able to synchronize all the philosophers without problems, but in MPI each process has its own memory-space. MPI uses only mutex-tool, controlled using messages generated by the processes.

This problem was solved by implementing one monitor-process and N-philosopher processes. The monitor process is used primarily as a "shared-resource" among all the other processes. Any kind of information regarding the state of the forks is obtained by sending and receiving messages to the monitor-process; the philosopher's finish when a required amount of global eats (M) has been attained. The following pseudo-code listing shows a brief description of the overall operation of the monitor and philosopher processes in this solution.

Code Listing 5a

```
1. PROCESS MONITOR
2. NUM_EATS = 0
3. WHILE NUM_EATS < M
    a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
    b. SWITCH MPI_MESSAGE_TAG
        i. CASE "GRAB_FORKS"
            1. L = MPI_SENDER_PROCESS - 1
            2. R = MPI_SENDER_PROCESS MOD N
            3. IF FORKS[L] = 0 AND FORKS[R] = 0 THEN
                a. FORKS[L] = 1
```

```
                        b. FORKS[R] = 1
                        c. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
                        d. NUM_EATS++
                    4. ELSE
                        a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
                    5. END IF
                ii. CASE “RELEASE_FORKS”
                    1. L = MPI_SENDER_PROCESS - 1
                    2. R = MPI_SENDER_PROCESS MOD N
                    3. FORKS[L] = 0
                    4. FORKS[R] = 0
                    5. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
        c. END SWITCH
4. END WHILE
5. SEND SIGNAL TO TERMINATE PROCESSES 1 TO N - 1
6. END PROCESS

7. PROCESS PHILOSOPHER
8. STATE = “THINKING”
9. WHILE TERMINATION_SIGNAL NOT RECEIVED
    a. IF STATE = “THINKING” THEN
        i. STATE = “HUNGRY”
    b. ELIF STATE = “HUNGRY” THEN
        i. MPI_SEND (“GRAB_FORKS”) TO (0)
        ii. IF MESSAGE_RESPONSE = “OK” THEN
            1. DELAY FOR RANDOM TIME
            2. MPI_SEND (“RELEASE_FORKS”) TO (0)
        iii. END IF
        iv. STATE = “THINKING”
    c. END IF
10. END WHILE
11. END PROCESS
```

5.2 MPI: PRODUCER AND CONSUMER

Using a similar technique as with the Philosophers, the Producer-Consumer was implemented using a monitor-process followed by N-Producer processes and M-consumer processes. The monitor process provides the shared-item buffer for K-elements and a message handler that reads commands from the producer/consumer processes and interprets them. The following pseudo-code listing describes the functionality of this scheme. The simulation ends when the number of desired item productions are completed (T).

Code Listing 5b

```
1.  PROCESS MONITOR
2.  PRODUCED = 0, CONSUMED = 0
3.  P_INDEX = 0, C_INDEX = 0, COUNT = 0
4.  WHILE (PRODUCED < T) AND (CONSUMED < T)
        a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
        b. SWITCH MPI_MESSAGE_TAG
            i. CASE "PRODUCE_ITEM"
                1. IF COUNT >= K THEN
                    a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
                    b. CONTINUE
                2. END IF
                3. BUFFER [ P_INDEX ] = RANDOM ()
                4. COUNT ++, PRODUCED++
                5. P_INDEX = ++P_INDEX MOD K
            ii. CASE "CONSUME_ITEM"
                1. IF COUNT = 0 THEN
                    a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
                    b. CONTINUE
                2. END IF
                3. COUNT--, CONSUMED++
                4. C_INDEX = ++C_INDEX MOD K
                5. MPI_SEND ("OK") TO (MPI_SENDER_PROCESS)
        c. END SWITCH
5.  END WHILE
6.  SEND SIGNAL TO TERMINATE PROCESSES
7.  END PROCESS

8.  PROCESS CONSUMER
9.  WHILE TERMINATION_SIGNAL NOT RECEIVED
        a. MPI_SEND ("CONSUME_ITEM") TO (0)
        b. IF MESSAGE_RESPONSE = "OK" THEN
            i. PRINT "ITEM CONSUMED"
        c. END IF
10. END WHILE
11. END PROCESS

12. PROCESS PRODUCER
13. WHILE TERMINATION_SIGNAL NOT RECEIVED
        a. MPI_SEND ("PRODUCER_ITEM") TO (0)
        b. IF MESSAGE_RESPONSE = "OK" THEN
```

```
            i. PRINT "ITEM PRODUCED"
        c. END IF
14. END WHILE
15. END PROCESS
```

5.3 MPI: SLEEPING BARBER

In the same way as the previous tests, this solution involves using a monitor process that manages the chair and barber state buffers. The process consists of barber processes (N), customer processes (M), number of chairs (C) and number of customers (T) needing service in order to finish the test.

Code Listing 5c

```
1.  PROCESS MONITOR
2.  SERVICED = 0
3.  CHAIRS_USED = 0
4.  WHILE SERVICED < T
        a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
        b. SWITCH MPI_MESSAGE_TAG
            i. CASE "SIT_AND_WAIT"
                1. IF CHAIRS_USED >= C THEN
                    a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
                    b. CONTINUE
                2. END IF
                3. CHAIRS_USED++
                4. MPI_SEND ("YES") TO (MPI_SENDER_PROCESS)
            ii. CASE "WAKE_SOMEBODY_UP"
                1. FOR I = 0 TO N – 1
                    a. IF BARBER_STATE[ I ] = "SLEEPING" THEN
                        i. BARBER_STATE[ I ] = "CUTTING"
                        ii. SERVICED++
                        iii. MPI_SEND ("OK") TO
                            (MPI_SENDER_PROCESS)
                        iv. EXIT SWITCH
                    b. END IF
                2. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
            iii. CASE "SLEEP"
                1. BARBER_STATE [MPI_SENDER_PROCESS] = "SLEEPING"
            iv. CASE "CHECK_WAITING_ROOM"
                1. IF BARBER_STATE[MPI_SENDER_PROCESS] = "SLEEPING"
                THEN
                    a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
                    b. EXIT SWITCH
```

2. END IF
3. IF CHAIRS_USED >= 0 THEN
    a. BARBER_STATE[MPI_SENDER_PROCESS] = "CUTTING"
    b. SERVICED++, CHAIRS_USED—
    c. MPI_SEND ("OK") TO (MPI_SENDER_PROCESS)
4. ELSE
    a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
5. END IF

c. END SWITCH

5. END WHILE
6. SEND SIGNAL TO TERMINATE PROCESSES
7. END PROCESS
8. PROCESS BARBER
9. WHILE TERMINATION_SIGNAL NOT RECEIVED
    a. MPI_SEND ("CHECK_WAITING_ROOM") TO (0)
    b. IF MESSAGE_RESPONSE = "OK" THEN
        i. PRINT "CUSTOMER SERVICED"
    c. ELSE
        i. MPI_SEND ("SLEEPING") TO (0)
    d. END IF
10. END WHILE
11. END PROCESS
12. PROCESS CUSTOMER
13. WHILE TERMINATION_SIGNAL NOT RECEIVED
    e. MPI_SEND ("WAKE_SOMEBODY_UP") TO (0)
    f. IF MESSAGE_RESPONSE = "OK" THEN
        i. PRINT "CUSTOMER SERVICED"
    g. ELSE
        i. MPI_SEND ("SIT_AND_WAIT") TO (0)
    b. END IF
14. END WHILE
15. END PROCESS

A very important and curious fact should be noted: in all of the MPI solutions no mutexes or anti-race-condition measures were taken. This is because, since all messages are coming in linearly through the MPI message queue one by one, the monitor process of each solution is able to respond to one request at a time, thus avoiding the need to implement any kind of mutual exclusion code. When a huge amount of data needs to be processed, MPI seems to be effectively the best library, because of its distributed nature [12].

OMP+MPI 6.0

By creating a hybrid combination of MPI and OMP, one can obtain a slightly more optimized solution that will take advantage of the CPU's resources on each machine. By using MPI, one is able to create a distributed application very quickly. Unfortunately MPI is very slow due to the message transmission [14] [16].

In previous test cases, the MPI was implemented treating each process like an atomic entity (i.e. monitor, producer, consumer, philosopher, etc). Here's where the OMP integration will come in handy: one can delegate more tasks to each process and instead of treating each process like atomic entities; one can treat them like ***blocks*** that hold several atomic entities.

In order to compile this specific breed of code, one needs OpenMPI and OpenMP installed on one's system. For more information refer to the proper sections above. To compile, one will have to use mpicc and the linker option –fopenmp (i.e. *mpicc –fopenmp program.c*).

6.1 OMP + MPI: DINING PHILOSOPHERS

For this problem, the solution is quite simple. One is still tied to the master/slave architecture of any MPI program [13]. However, this time each slave process will be treated like a block of atomic units, each atomic unit in this case is a philosopher thread running with OMP.

The master thread's task will be to balance the number of threads along all the slave processes to maintain a high level of performance. Say one has 351 philosophers (N) and one must run the program using "mpirun –np 10 philos.bin". This will produce 1 master process and 9 slave processes (K). Then the master thread will divide the 351 threads along the 9 slaves, resulting in a balance of 39 OMP threads on each slave process.

Code Listing 6a

```
1. PROCESS MONITOR
2. NUM_EATS = 0
3. THREADS = N / K
4. TOTAL = N
5. FOR I = 1 TO K
    a. IF TOTAL > THREADS THEN
        i. M = THREADS
    b. ELSE
        i. M = TOTAL
    c. END IF
    d. MPI_SEND ( M ) TO ( I )
    e. TOTAL = TOTAL - M
6. NEXT
```

7. WHILE NUM_EATS < M
    a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
    b. SWITCH MPI_MESSAGE_TAG
        i. CASE “GRAB_FORKS”
            1. L = MPI_SENDER_PROCESS - 1
            2. R = MPI_SENDER_PROCESS MOD N
            3. IF FORKS[L] = 0 AND FORKS[R] = 0 THEN
                a. FORKS[L] = 1
                b. FORKS[R] = 1
                c. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
                d. NUM_EATS++
            4. ELSE
                a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
            5. END IF
        ii. CASE “RELEASE_FORKS”
            1. L = MPI_SENDER_PROCESS - 1
            2. R = MPI_SENDER_PROCESS MOD N
            3. FORKS[L] = 0
            4. FORKS[R] = 0
            5. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
    c. END SWITCH
8. END WHILE
9. SEND SIGNAL TO TERMINATE PROCESSES 1 TO N
10. END PROCESS
11. PROCESS PHILOSOPHER_CONTROLLER
12. omp_set_num_threads (MPI_RECEIVE (INT))
13. #pragma omp parallel
    a. PHILOSOPHER ()
14. END PROCESS
15. THREAD PHILOSOPHER
16. STATE = “THINKING”
17. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. IF STATE = “THINKING” THEN
        i. STATE = “HUNGRY”
    b. ELIF STATE = “HUNGRY” THEN
        i. V_MPI_SEND (“GRAB_FORKS”) TO (0)
        ii. IF V_MESSAGE_RESPONSE = “OK” THEN
            1. DELAY FOR RANDOM TIME
            2. V_MPI_SEND (“RELEASE_FORKS”) TO (0)
        iii. END IF

iv. STATE = "THINKING"

c. END IF

18. END WHILE
19. END THREAD

The functions prefixed with "V_" are aliases of the MPI message functions that are created locally in order to add the request to a queue. Since the MPI does not allow a thread to use the message transmission interface, one has to queue all the requests from the threads in to a buffer and then process them from the main thread. Sending messages from a thread will cause the MPI to produce very strange results at the end. This is a limitation of the MPI's thread safety implementation of the MPI library used.

6.2 OMP + MPI: PRODUCER AND CONSUMER

This solution is very similar to the one created for MPI, with the exception that instead of sending messages directly to the MPI library, one has to use the V_ wrappers to send them to the local process queue for later sending. One needs to balance the threads along the processes.

Since the producers-consumers problem has 2 entities (producers and consumers), one will need to specify how many processes will be assigned for each entity, and also the total threads for each entity. Using that information, one can determine how many threads each process will contain, and how many processes each entity will be holding. npProds and npCons tells the number of producer and consumer processes, in the same way, ntProds and ntCons tells the number of producer and consumer threads.

Code Listing 6b

1. PROCESS MONITOR
2. P = 1
3. THREADS = ntProds / npProds
4. TOTAL = ntProds
5. FOR I = 1 TO npProds
    a. IF TOTAL > THREADS THEN
        i. M = THREADS
    b. ELSE
        i. M = TOTAL
    c. END IF
    d. MPI_SEND (M) TO (P)
    e. TOTAL = TOTAL - M
    f. P = P + 1
6. NEXT
7. THREADS = ntCons / npCons

8. TOTAL = ntCons
9. FOR I = 1 TO npCons
    a. IF TOTAL > THREADS THEN
        i. M = THREADS
    b. ELSE
        i. M = TOTAL
    c. END IF
    d. MPI_SEND (M) TO (P)
    e. TOTAL = TOTAL - M
    f. P = P + 1
10. NEXT
11. PRODUCED = 0, CONSUMED = 0
12. P_INDEX = 0, C_INDEX = 0, COUNT = 0
13. WHILE (PRODUCED < T) AND (CONSUMED < T)
    a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
    b. SWITCH MPI_MESSAGE_TAG
        i. CASE “PRODUCE_ITEM”
            1. IF COUNT >= K THEN
                a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
                b. CONTINUE
            2. END IF
            3. BUFFER [P_INDEX] = RANDOM ()
            4. COUNT ++, PRODUCED++
            5. P_INDEX = ++P_INDEX MOD K
        ii. CASE “CONSUME_ITEM”
            1. IF COUNT = 0 THEN
                a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
                b. CONTINUE
            2. END IF
            3. COUNT--, CONSUMED++
            4. C_INDEX = ++C_INDEX MOD K
            5. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
    c. END SWITCH
14. END WHILE
15. SEND SIGNAL TO TERMINATE PROCESSES
16. END PROCESS
17. PROCESS CONSUMER_CONTROLLER
18. omp_set_num_threads (MPI_RECEIVE (INT))
19. #pragma omp parallel
    b. CONSUMER ()

20. END PROCESS
21. PROCESS PRODUCER_CONTROLLER
22. omp_set_num_threads (MPI_RECEIVE (INT))
23. #pragma omp parallel
    c. PRODUCER ()
24. END PROCESS
25. THREAD CONSUMER
26. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. V_MPI_SEND ("CONSUME_ITEM") TO (0)
    b. IF V_MESSAGE_RESPONSE = "OK" THEN
        i. PRINT "ITEM CONSUMED"
    c. END IF
27. END WHILE
28. END PROCESS
29. THREAD PRODUCER
30. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    d. V_MPI_SEND ("PRODUCER_ITEM") TO (0)
    e. IF V_MESSAGE_RESPONSE = "OK" THEN
        i. PRINT "ITEM PRODUCED"
    f. END IF
31. END WHILE
32. END PROCESS

If the number of threads specified is not divisible by the number of processes, the monitor process will assign a few extra threads to the last slave process in order to match the wanted number of threads.

6.3 OMP + MPI: SLEEPING BARBER

Similar to the previous tests, this solution involves using a global monitor process that manages the chair and barber state buffers and acts like an intermediary between the other processes. It consists of barber processes (N), customer processes (M), number of chairs (C) and number of customers (T) needed service in order to finish the test, also the number of threads for the barbers (A) and the number of threads for the customers (B).

Code Listing 6c

1. PROCESS MONITOR
2. P = 1
3. THREADS = A / N
4. TOTAL = A
5. FOR I = 1 TO N
    a. IF TOTAL > THREADS THEN

i. M = THREADS

b. ELSE

i. M = TOTAL

c. END IF

d. MPI_SEND (M) TO (P)

e. TOTAL = TOTAL - M

f. P = P + 1

6. NEXT
7. THREADS = B / M
8. TOTAL = B
9. FOR I = 1 TO M
    a. IF TOTAL > THREADS THEN
        i. M = THREADS
    b. ELSE
        ii. M = TOTAL
    c. END IF
    d. MPI_SEND (M) TO (P)
    e. TOTAL = TOTAL - M
    f. P = P + 1
10. NEXT
11. SERVICED = 0
12. CHAIRS_USED = 0
13. WHILE SERVICED < T
    a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
    b. SWITCH MPI_MESSAGE_TAG
        i. CASE “SIT_AND_WAIT”
            1. IF CHAIRS_USED >= C THEN
                a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
                b. CONTINUE
            2. END IF
            3. CHAIRS_USED++
            4. MPI_SEND (“YES”) TO (MPI_SENDER_PROCESS)
        ii. CASE “WAKE_SOMEBODY_UP”
            1. FOR I = 0 TO N – 1
                a. IF BARBER_STATE[I] = “SLEEPING” THEN
                    i. BARBER_STATE[I] = “CUTTING”
                    ii. SERVICED++
                    iii. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
                    iv. EXIT SWITCH

b. END IF

2. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)

iii. CASE “SLEEP”

1. BARBER_STATE [MPI_SENDER_PROCESS] = “SLEEPING”

iv. CASE “CHECK_WAITING_ROOM”

1. IF BARBER_STATE[MPI_SENDER_PROCESS] = “SLEEPING” THEN
    a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
    b. EXIT SWITCH
2. END IF
3. IF CHAIRS_USED >= 0 THEN
    a. BARBER_STATE[MPI_SENDER_PROCESS] = “CUTTING”
    b. SERVICED++, CHAIRS_USED—
    c. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
4. ELSE
    a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
5. END IF

c. END SWITCH

14. END WHILE
15. SEND SIGNAL TO TERMINATE PROCESSES
16. END PROCESS
17. PROCESS BARBER_CONTROLLER
18. omp_set_num_threads (MPI_RECEIVE (INT))
19. #pragma omp parallel
    a. BARBER ()
20. END PROCESS
21. PROCESS CUSTOMER_CONTROLLER
22. omp_set_num_threads (MPI_RECEIVE (INT))
23. #pragma omp parallel
    a. CUSTOMER ()
24. END PROCESS
25. THREAD BARBER
26. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. V_MPI_SEND (“CHECK_WAITING_ROOM”) TO (0)
    b. IF V_MESSAGE_RESPONSE = “OK” THEN
        i. PRINT “CUSTOMER SERVICED”
    c. ELSE
        i. V_MPI_SEND (“SLEEPING”) TO (0)
    d. END IF

27. END WHILE
28. END THREAD
29. THREAD CUSTOMER
30. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. V_MPI_SEND (“WAKE_SOMEBODY_UP”) TO (0)
    b. IF V_MESSAGE_RESPONSE = “OK” THEN
        i. PRINT “CUSTOMER SERVICED”
    c. ELSE
        ii. V_MPI_SEND (“SIT_AND_WAIT”) TO (0)
    d. END IF
31. END WHILE
32. END THREAD

PTHREADS+MPI 7.0

Similar to the previous test case when OMP was combined with MPI, one is able to make another hybrid resulting from merging PThreads multithreading library with MPI. One will require the same items that were mentioned in the PThreads and MPI sections, respectively. The same principle will apply, that is, using a primary monitor process that will act like a shared resources manager.

In order to compile this type of code, one needs OpenMPI and PThreads installed on one's system, for more information refer to the proper sections. To compile, one will have to use mpicc and the linker option –pthread (i.e. *mpicc –pthread program.c*).

7.1 PTHREADS + MPI: DINING PHILOSOPHERS

Again, the brief description of this solution might be somewhat redundant, as it is the same as the one used for the previous test (when OMP and MPI was combined). Skipping over to the pseudo-code listing for this code, K is the number of philosophers to use, remember that K must be an odd number, or else an error will be issued and N is the number of threads to assign for the purpose of this problem.

Code Listing 7a

```
1. PROCESS MONITOR
2. NUM_EATS = 0
3. THREADS = N / K
4. TOTAL = N
5. FOR I = 1 TO K
    a. IF TOTAL > THREADS THEN
        i. M = THREADS
    b. ELSE
        i. M = TOTAL
    c. END IF
    d. MPI_SEND ( M ) TO ( I )
    e. TOTAL = TOTAL - M
6. NEXT
7. WHILE NUM_EATS < M
    a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
    b. SWITCH MPI_MESSAGE_TAG
        i. CASE “GRAB_FORKS”
            1. L = MPI_SENDER_PROCESS - 1
            2. R = MPI_SENDER_PROCESS MOD N
            3. IF FORKS[L] = 0 AND FORKS[R] = 0 THEN
                a. FORKS[L] = 1
```

b. FORKS[R] = 1
c. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)
d. NUM_EATS++

4. ELSE
   a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
5. END IF

ii. CASE “RELEASE_FORKS”
1. L = MPI_SENDER_PROCESS - 1
2. R = MPI_SENDER_PROCESS MOD N
3. FORKS[L] = 0
4. FORKS[R] = 0
5. MPI_SEND (“OK”) TO (MPI_SENDER_PROCESS)

c. END SWITCH

8. END WHILE
9. SEND SIGNAL TO TERMINATE PROCESSES 1 TO N
10. END PROCESS

11. PROCESS PHILOSOPHER_CONTROLLER
12. INT THREADS = MPI_RECEIVE (INT)
13. FOR I = 1 TO THREADS
    a. PTHREAD_ALLOC_THREAD (&PHILOSOPHER)
14. NEXT
15. END PROCESS

16. FUNCTION PHILOSOPHER
17. STATE = “THINKING”
18. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. IF STATE = “THINKING” THEN
       i. STATE = “HUNGRY”
    b. ELIF STATE = “HUNGRY” THEN
       i. V_MPI_SEND (“GRAB_FORKS”) TO (0)
       ii. IF V_MESSAGE_RESPONSE = “OK” THEN
           1. DELAY FOR RANDOM TIME
           2. V_MPI_SEND (“RELEASE_FORKS”) TO (0)
       iii. END IF
       iv. STATE = “THINKING”
    c. END IF
19. END WHILE
20. END FUNCTION

The function named “PTHREAD_ALLOC_THREAD” encapsulates the thread initialization, i.e. creating a handle for the thread, initializing the handle with the appropriate attributes and then setting the entry point to the provided address of the function.

7.2 PTHREADS + MPI: PRODUCER AND CONSUMER

The overall functionality of this solution is incredibly similar to the OMP+MPI solution. For these problems, the main and only difference is the use of the PThreads library to allocate threads, whereas the other solution used OMP. The variables used here are npProds and npCons. These specify the number of producers and consumers (processes), and ntProds and ntCons which tell the number of threads for each entity.

Code Listing 7b

```
1.  PROCESS MONITOR
2.  P = 1
3.  THREADS = ntProds / npProds
4.  TOTAL = ntProds
5.  FOR I = 1 TO npProds
        a.  IF TOTAL > THREADS THEN
                i.  M = THREADS
        b.  ELSE
                i.  M = TOTAL
        c.  END IF
        d.  MPI_SEND ( M ) TO ( P )
        e.  TOTAL = TOTAL - M
        f.  P = P + 1
6.  NEXT
7.  THREADS = ntCons / npCons
8.  TOTAL = ntCons
9.  FOR I = 1 TO npCons
        a.  IF TOTAL > THREADS THEN
                i.  M = THREADS
        b.  ELSE
                ii.  M = TOTAL
        c.  END IF
        d.  MPI_SEND ( M ) TO ( P )
        e.  TOTAL = TOTAL - M
        f.  P = P + 1
10. NEXT
11. PRODUCED = 0, CONSUMED = 0
12. P_INDEX = 0, C_INDEX = 0, COUNT = 0
13. WHILE (PRODUCED < T) AND (CONSUMED < T)
```

a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
b. SWITCH MPI_MESSAGE_TAG
    i. CASE "PRODUCE_ITEM"
        1. IF COUNT >= K THEN
            a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
            b. CONTINUE
        2. END IF
        3. BUFFER [P_INDEX] = RANDOM ()
        4. COUNT ++, PRODUCED++
        5. P_INDEX = ++P_INDEX MOD K
    ii. CASE "CONSUME_ITEM"
        1. IF COUNT = 0 THEN
            a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
            b. CONTINUE
        2. END IF
        3. COUNT--, CONSUMED++
        4. C_INDEX = ++C_INDEX MOD K
        5. MPI_SEND ("OK") TO (MPI_SENDER_PROCESS)
c. END SWITCH

14. END WHILE
15. SEND SIGNAL TO TERMINATE PROCESSES
16. END PROCESS

17. PROCESS CONSUMER_CONTROLLER
18. INT THREADS = MPI_RECEIVE (INT)
19. FOR I = 1 TO THREADS
    a. PTHREAD_ALLOC_THREAD (&CONSUMER)
20. NEXT
21. END PROCESS

22. PROCESS PRODUCER_CONTROLLER
23. INT THREADS = MPI_RECEIVE (INT)
24. FOR I = 1 TO THREADS
    a. PTHREAD_ALLOC_THREAD (&PRODUCER)
25. END PROCESS

26. FUNCTION CONSUMER
27. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. V_MPI_SEND ("CONSUME_ITEM") TO (0)
    b. IF V_MESSAGE_RESPONSE = "OK" THEN

```
            i. PRINT "ITEM CONSUMED"
        c. END IF
28. END WHILE
29. END FUNCTION

30. FUNCTION PRODUCER
31. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
        a. V_MPI_SEND ("PRODUCER_ITEM") TO (0)
        b. IF V_MESSAGE_RESPONSE = "OK" THEN
            i. PRINT "ITEM PRODUCED"
        c. END IF
32. END WHILE
33. END FUNCTION
```

7.3 PTHREADS + MPI: SLEEPING BARBER

The variables used for these solutions are: N: Number of Barber processes; M: Number of Customer processes; C: Number of chairs in the waiting room and T: Number of customers serviced to reach in order to finish the test; A: Number of threads for barbers and B: Number of threads for customers.

Code Listing 7c

```
1. PROCESS MONITOR
2. P = 1
3. THREADS = A / N
4. TOTAL = A
5. FOR I = 1 TO N
        a. IF TOTAL > THREADS THEN
            iii. M = THREADS
        b. ELSE
            iv. M = TOTAL
        c. END IF
        d. MPI_SEND ( M ) TO ( P )
        e. TOTAL = TOTAL - M
        f. P = P + 1
6. NEXT
7. THREADS = B / M
8. TOTAL = B
9. FOR I = 1 TO M
        a. IF TOTAL > THREADS THEN
            i. M = THREADS
        b. ELSE
            i. M = TOTAL
```

c. END IF
d. MPI_SEND (M) TO (P)
e. TOTAL = TOTAL - M
f. P = P + 1

10. NEXT
11. SERVICED = 0
12. CHAIRS_USED = 0
13. WHILE SERVICED < T
    a. IF MPI_MESSAGE_AVAILABLE = 0 THEN CONTINUE
    b. SWITCH MPI_MESSAGE_TAG
        i. CASE "SIT_AND_WAIT"
            1. IF CHAIRS_USED >= C THEN
                a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
                b. CONTINUE
            2. END IF
            3. CHAIRS_USED++
            4. MPI_SEND ("YES") TO (MPI_SENDER_PROCESS)
        ii. CASE "WAKE_SOMEBODY_UP"
            1. FOR I = 0 TO N – 1
                a. IF BARBER_STATE[I] = "SLEEPING" THEN
                    i. BARBER_STATE[I] = "CUTTING"
                    ii. SERVICED++
                    iii. MPI_SEND ("OK") TO (MPI_SENDER_PROCESS)
                    iv. EXIT SWITCH
                b. END IF
            2. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
        iii. CASE "SLEEP"
            1. BARBER_STATE [MPI_SENDER_PROCESS] = "SLEEPING"
        iv. CASE "CHECK_WAITING_ROOM"
            1. IF BARBER_STATE[MPI_SENDER_PROCESS] = "SLEEPING" THEN
                a. MPI_SEND ("NO") TO (MPI_SENDER_PROCESS)
                b. EXIT SWITCH
            2. END IF
            3. IF CHAIRS_USED >= 0 THEN
                a. BARBER_STATE[MPI_SENDER_PROCESS] = "CUTTING"
                b. SERVICED++, CHAIRS_USED—
                c. MPI_SEND ("OK") TO (MPI_SENDER_PROCESS)

4. ELSE
    a. MPI_SEND (“NO”) TO (MPI_SENDER_PROCESS)
5. END IF

c. END SWITCH

14. END WHILE
15. SEND SIGNAL TO TERMINATE PROCESSES
16. END PROCESS

17. PROCESS BARBER_CONTROLLER
18. INT THREADS = MPI_RECEIVE (INT)
19. FOR I = 1 TO THREADS
    a. PTHREAD_ALLOC_THREAD (&BARBER)
20. NEXT
21. END PROCESS

22. PROCESS CUSTOMER_CONTROLLER
23. INT THREADS = MPI_RECEIVE (INT)
24. FOR I = 1 TO THREADS
    a. PTHREAD_ALLOC_THREAD (&CUSTOMER)
25. NEXT
26. END PROCESS

27. FUNCTION BARBER
28. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. V_MPI_SEND (“CHECK_WAITING_ROOM”) TO (0)
    b. IF V_MESSAGE_RESPONSE = “OK” THEN
        i. PRINT “CUSTOMER SERVICED”
    c. ELSE
        i. V_MPI_SEND (“SLEEPING”) TO (0)
    d. END IF
    e. END IF
29. END WHILE
30. END FUNCTION

31. FUNCTION CUSTOMER
32. WHILE V_TERMINATION_SIGNAL NOT RECEIVED
    a. V_MPI_SEND (“WAKE_SOMEBODY_UP”) TO (0)
    b. IF V_MESSAGE_RESPONSE = “OK” THEN
        i. PRINT “CUSTOMER SERVICED”
    c. ELSE

ii. V_MPI_SEND (“SIT_AND_WAIT”) TO (0)

f. END IF

33. END WHILE

34. END FUNCTION

TEST SCENARIO 8.0

Table 8a

	Single Core	Dual Core	Quad Core	MTL
Hardware	**Intel® Pentium® 4 Processor (2.60 GHz, 512K Cache, 800 MHz FSB)** **2 GB RAM**	**Intel® Core™2 Duo Processor E4700 (2M Cache, 2.60 GHz, 800 MHz FSB)** **2 GB RAM**	**Intel® Core™2 Quad Processor Q6700 (8M Cache, 2.66 GHz, 1066 MHz FSB)** **2 GB RAM**	**Intel 32 Core Processor**
Operating System and Software Packages.	**Linux Ubuntu 10.4** **OpenMPI 1.4.3** **OpenMP 3.0**	**Linux Ubuntu 10.4** **OpenMPI 1.4.3** **OpenMP 3.0**	**Linux Ubuntu 10.4** **OpenMPI 1.4.3** **OpenMP 3.0**	**N/A**

Hardware and Software configuration used for Testing

Table 8b

	Case 1	Case 2	Case 3
Dining Philosopher	**M1 = 3-257** **N1 = 100**	**M1 = 3-257** **N1 = 1000**	**M1 = 3-257** **N1 = 10000**
Producer Consumer	**M2 = 2-256** **N2 = 2-256** **O2 = 2000** **P2 = 2000**	**M2 = 2-256** **N2 = 2-256** **O2 = 20000** **P2 = 2000**	**M2= 2-256** **N2 = 2-256** **O2 = 200000** **P2 = 2000**
Sleeping Barber	**M3 = 150** **N3 = 2-256** **O3 = 200** **P3 = 2-256**	**M3 = 150** **N3 = 2-256** **O3 = 2000** **P3 = 2-256**	**M3 = 150** **N3 = 2-256** **O3 = 20000** **P3 = 2-256**

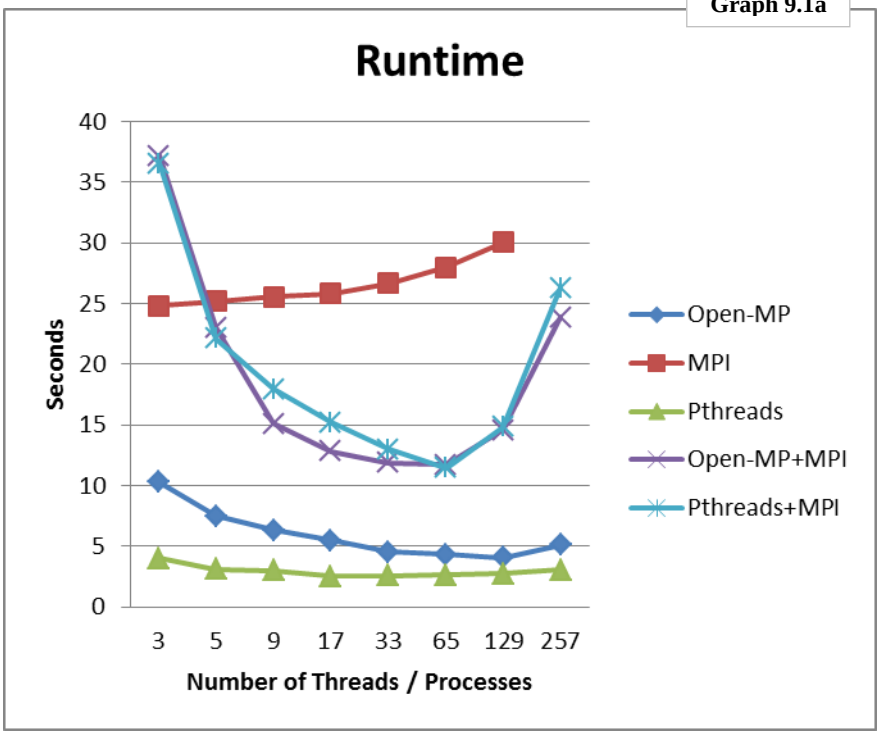
Test Cases for Inter-Process Communication Problems

- M1 - Philosopher Threads / Processes, N1 - Total Eats
- M2 - Producer Threads / Processes, N2 - Consumer Threads / Processes, O2 - Total Packets, P2 - Buffer Size
- M3 - Chairs in Barber Shop, N3 - Barber Threads / Processes, O3 - Total Haircuts, P3 - Client Threads / Processes

RESULTS 9.0

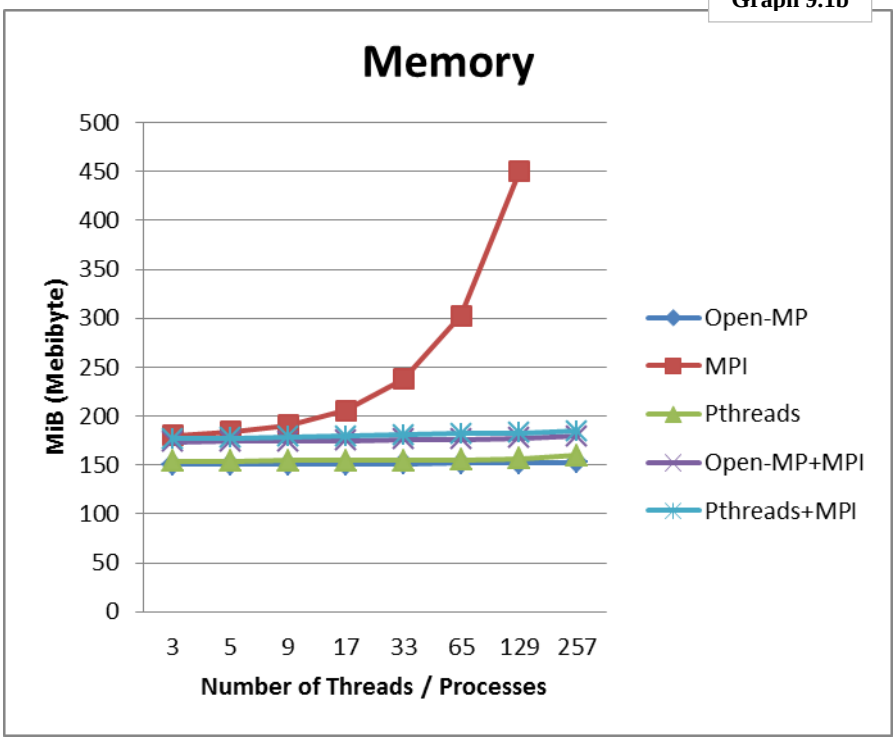
9.1 DINING PHILOSOPHERS

Graph 9.1a



Dining Philosophers execution time in Seconds vs. Threads / Processes (Quad Core – Test Case 2)

Graph 9.1b

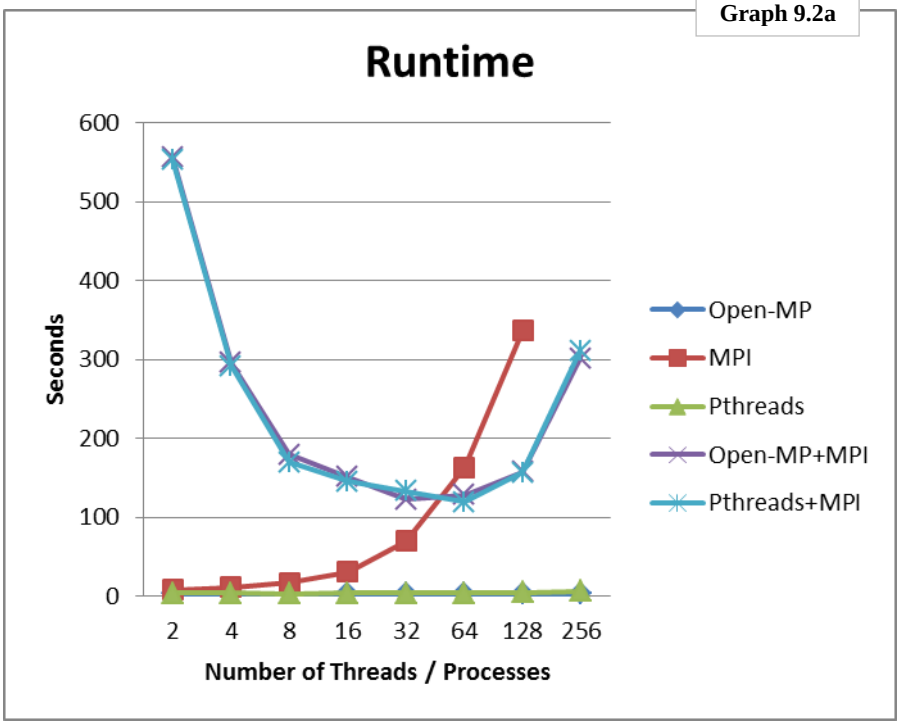


Dining Philosophers memory consumption in MiB vs. Threads / Processes (Quad Core – Test Case 2)

The MiB or **mebibyte is a multiple of the unit byte for quantities of digital information. The binary prefix mebi means 2^{20}, therefore 1 mebibyte is 1048576bytes*.

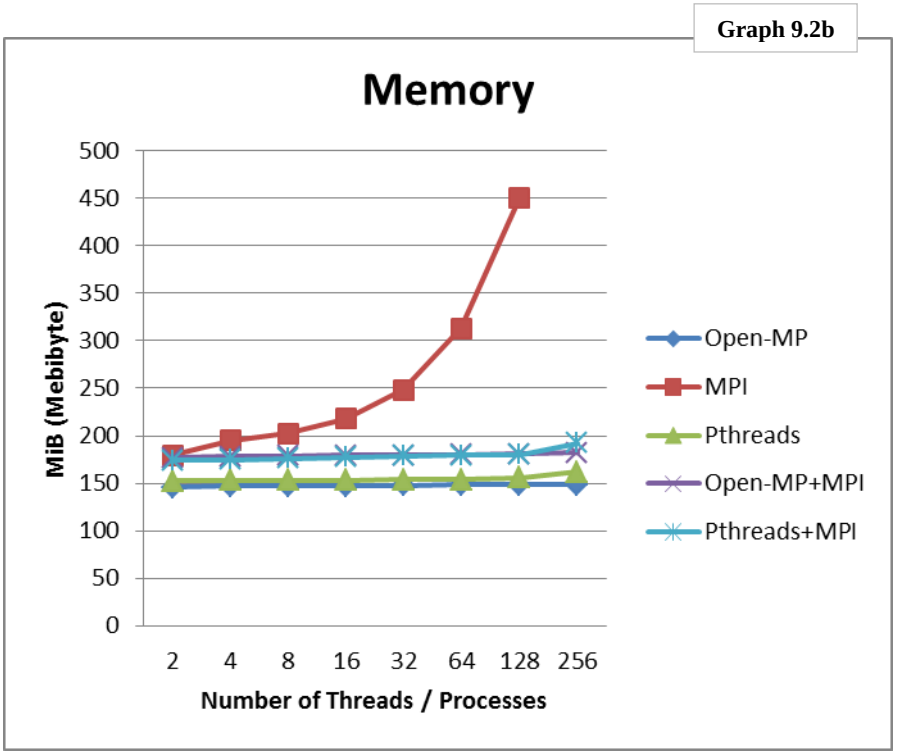
9.2 PRODUCER AND CONSUMER

Graph 9.2a



Producer Consumer execution time in Seconds vs. Threads / Processes (Quad Core – Test Case 2)

Graph 9.2b

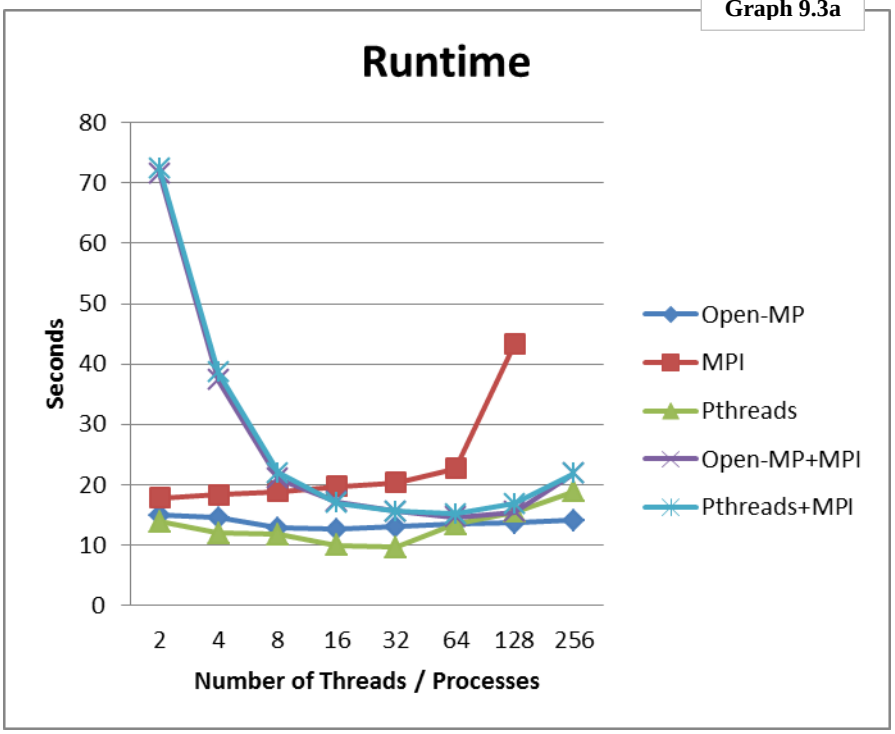


Producer Consumer memory consumption in MiB vs. Threads / Processes (Quad Core – Test Case 2)

The MiB or **mebibyte is a multiple of the unit byte for quantities of digital information. The binary prefix mebi means 2^{20}, therefore 1 mebibyte is 1048576bytes*.

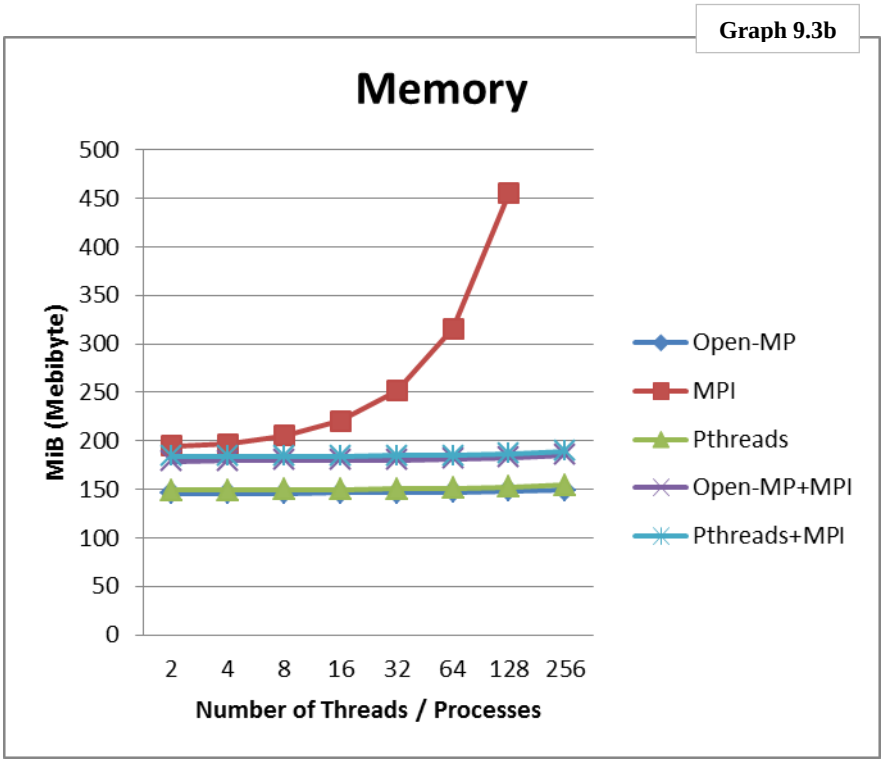
9.3 SLEEPING BARBER

Graph 9.3a



Sleeping Barber execution time in Seconds vs. Threads / Processes (Quad Core – Test Case 2)

Graph 9.3b

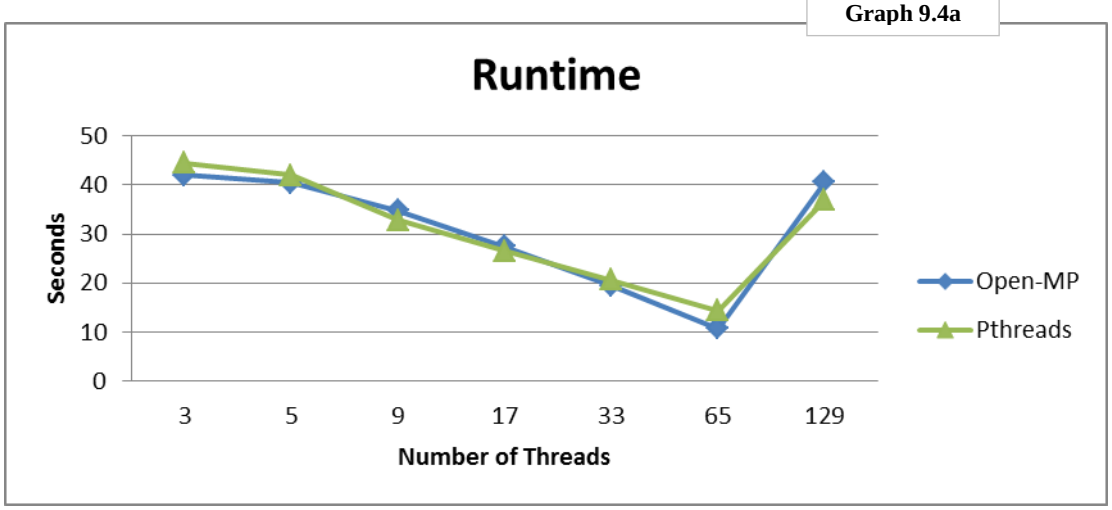


Sleeping Barber memory consumption in MiB vs. Threads / Processes (Quad Core – Test Case 2)

The MiB or **mebibyte is a multiple of the unit byte for quantities of digital information. The binary prefix mebi means 2^{20}, therefore 1 mebibyte is 1048576bytes*.

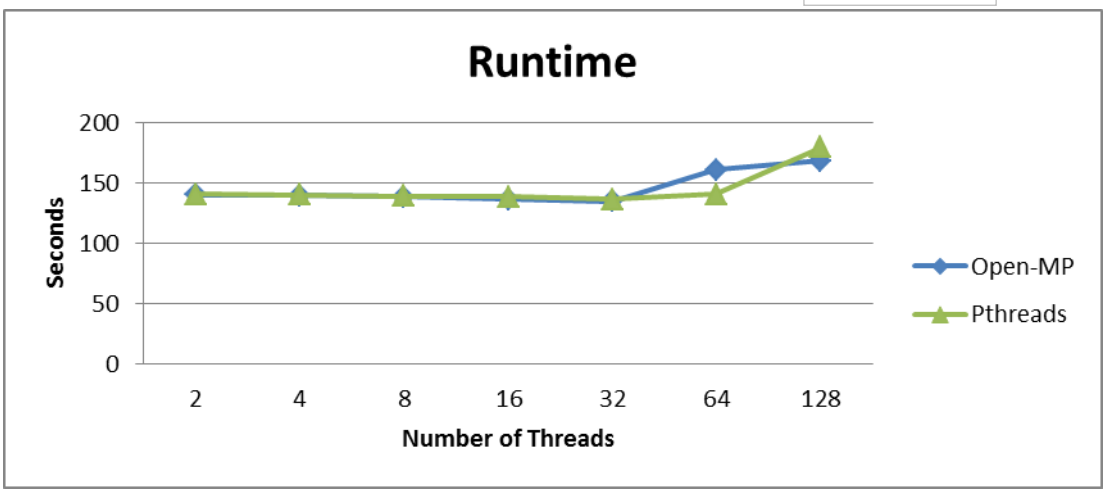
9.4 MTL RESULTS

Graph 9.4a



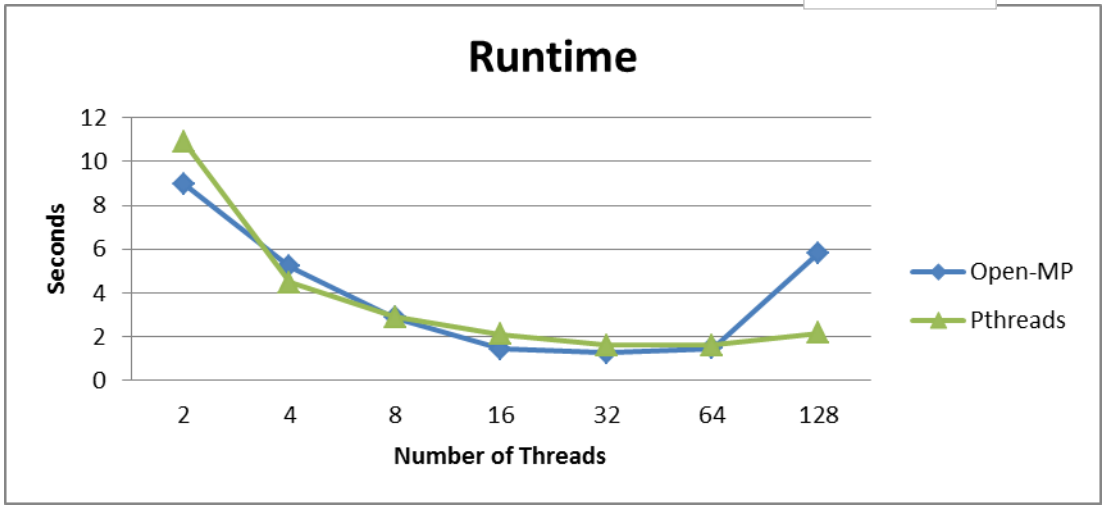
Dining Philosophers execution time in Seconds vs. Threads (32 Core MTL – Test Case 2)

Graph 9.4b



Producer Consumer execution time in Seconds vs. Threads (32 Core MTL – Test Case 2)

Graph 9.4c



Sleeping Barber execution time in Seconds vs. Threads (32 Core MTL – Test Case 2)

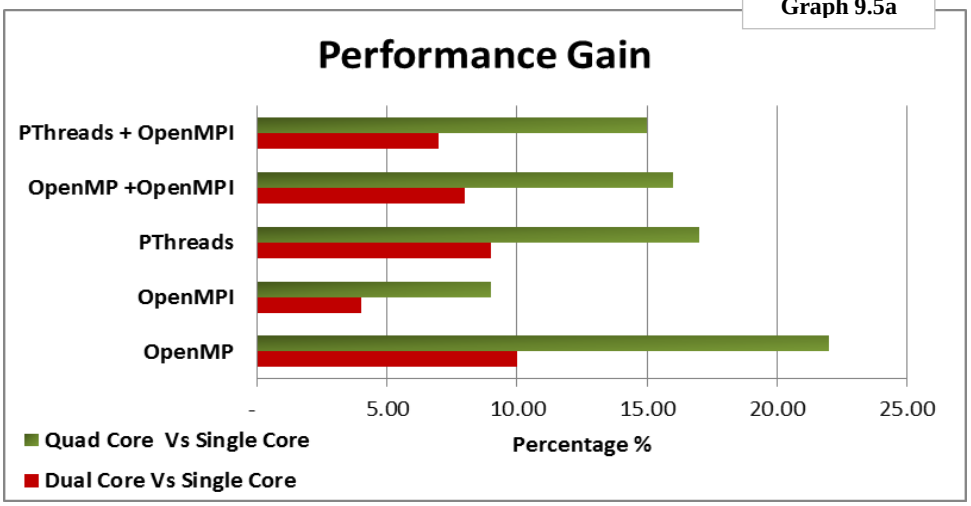
9.5 PERFORMANCE STATISTICS

Table 9a

	OpenMP	OpenMPI	PThreads	OpenMP + OpenMPI	PThreads + OpenMPI
Dual Core Vs. Single Core	10%	4%	9%	8%	7%
Quad Core Vs. Single Core	22%	9%	17%	16%	15%

Performance Gain by decrease in execution time

Graph 9.5a



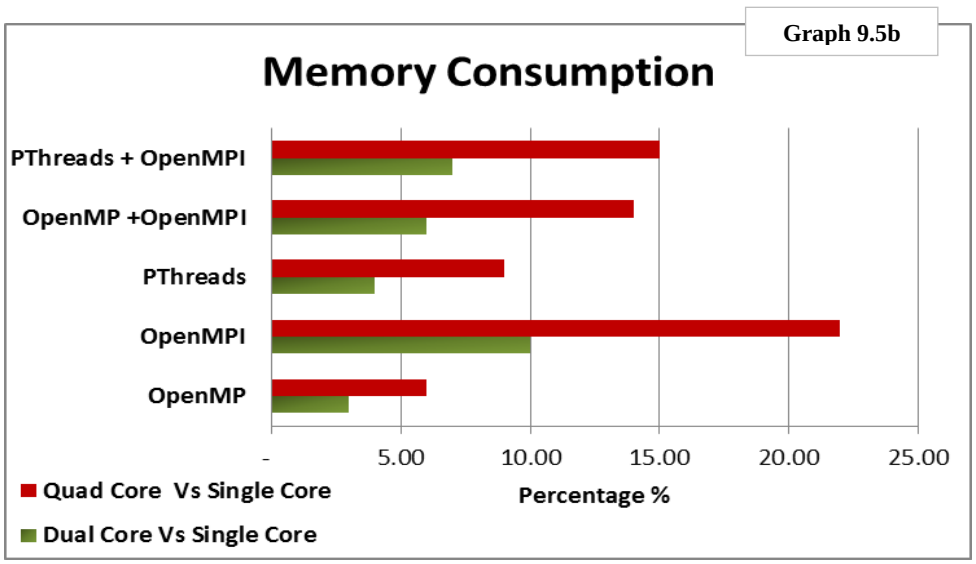
Performance Gain

Table 9b

	OpenMP	OpenMPI	PThreads	OpenMP + OpenMPI	PThreads + OpenMPI
Dual Core Vs. Single Core	3%	10%	4%	6%	7%
Quad Core Vs. Single Core	6%	22%	9%	14%	15%

Increase in memory consumption by decrease in execution time

Graph 9.5b

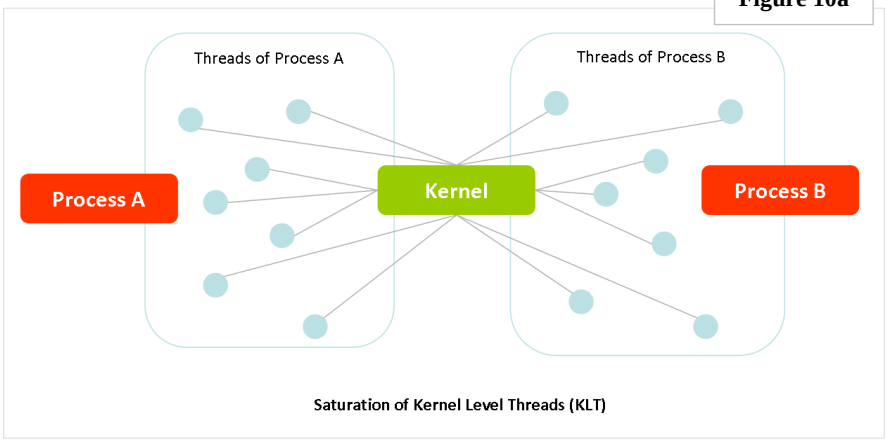


Memory Consumption

ANALYSES 10.0

From the analyses of graphs and performance matrices, it can be concluded that due to context switching and OS scheduling policies, a CPU can run several threads concurrently over a system. But with an increase in the number of threads, the performance can rise up to a certain limit, rather than increasing infinitely. After that, if we increase more threads the performance starts degrading. This is due to the fact that as more threads are created, the kernel management modules become too overloaded to handle such level of threads, resulting in a noticeable degradation of the performance. This saturation of kernel management related to multi-threading can be seen in the following figure.

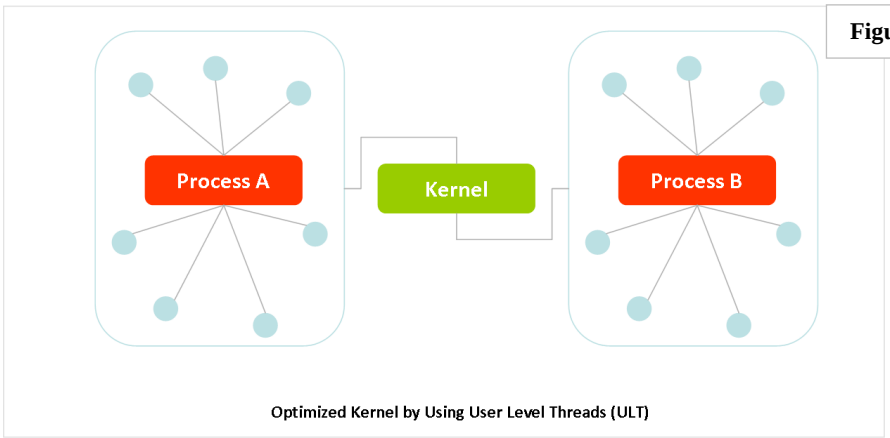
Figure 10a



Saturation of Kernel Level Threads (KLT)

For the presented inter-process communication problems, until now, OpenMP has, undoubtedly, proven to be the best contender in both performance and memory usage. My opinion in this matter is rather down to earth. OMP is the only one of the libraries used that implement its functionality completely hidden from the user. In the rest (MPI and PThreads), the user needs to create and manipulate the threads, causing a certain level of user-library interaction, but OMP hides all the actual management and provides only very easy-to-use #pragma directives to create the threads. The actual management is done internally by OMP, providing a much greater optimization. The kernel is freed from working with so many threads, causing the overall experience of working with OMP to be a lot more efficient, as shown in the following figure.

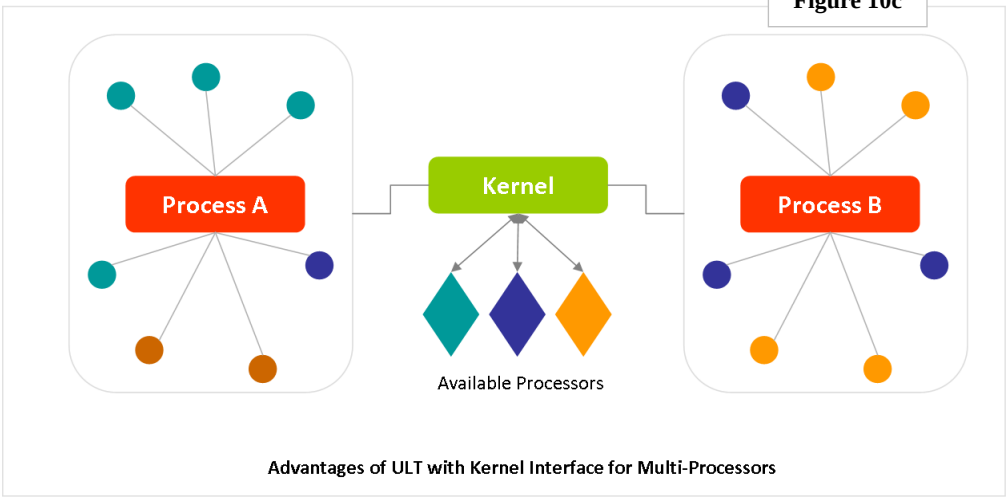
Figure 10b



Optimized Kernel by Using User Level Threads (ULT)

I have tested my results on single core, dual core and quad core machines manufactured by Intel Corporation, but the results I got were very close to each other as compared to different APIs and libraries used in this project. By having more processors and cores the efficiency can be increased if one uses user level threads because that maintains the kernel a bit less saturated, as it can be observed in the following figure.

Figure 10c



Advantages of ULT with Kernel Interface for Multi-Processors

There are some significant facts I came across about all these libraries and API specifications, which are mentioned in more detail in the table shown below. It explains the limits, learning time, difficulty to perform certain tasks and overall performance.

Table 10a

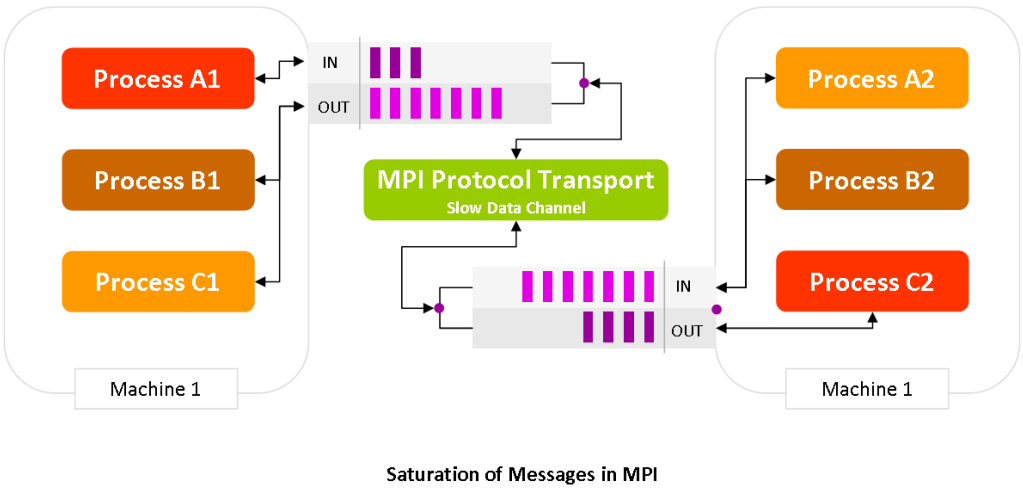
Features	OpenMP	OpenMPI	PThreads	OpenMP + OpenMPI	PThreads + OpenMPI
Max Threads / Processes	380	252	Tested up to 50,000.	95,760	Tested up to 50,000.
Synchronization	Easy	Tough	Easy	Toughest	Toughest
Lock Structure	Easy	Medium	Easy	Hard	Hard
Learning Curve	Low	High	Medium	Highest	Highest
Memory Usage	Less	More	Less	Lesser than MPI	Lesser than MPI
Performance	High in shared memory architecture.	High in distributed memory architecture.	Moderate in shared memory architecture.	Highest in distributed memory architecture.	Highest in distributed memory architecture.
Lines of Code	Less	More	More than OpenMP.	More than MPI.	More than MPI.

Significant Facts about libraries and API specifications

In case of hybrid memory, I faced a problem. MPI messages are quite slow as compared to threads. I am getting non-deterministic bugs in my code such as deadlocks and race conditions, due to lack of synchronization, so for this I have come up with a solution. What I did was

simple. *I didn't send MPI calls from the threads*; instead, I used the main process to send the messages. This way each thread sends and receives messages to a virtual message queue, and then the main process takes that queue and processes it one message at a time. This is how I am able to achieve proper synchronization in hybrid memory architecture. The slower performance of MPI is due to its message passing through the network interface, causing a very unpleasant and long latency when several messages are sent, because the internal message queues become very full as it can be seen in the following figure.

Figure 10d



Saturation of Messages in MPI

While running my code on Quad-Core machines, I found several non-deterministic bugs in one of my IPC problems, so I had to modify the code and test the results again over a different platform including MTL. Now I can say that my code is scalable over N number of Cores as I tested it on Intel 32 Core Machine. This work was made possible because of machine time provided by Intel on their Manycore Testing Lab (MTL).

Here both white box and black box testing is done effectively by using extensive test cases and the codes are highly optimized in so that I can achieve the best performance from them.

This project is helpful for future programmers because it explains the problems that they will face when using any of these libraries; tricky problems such as the MPI passing interface working only from the main-thread; or that the OMP has a limited amount of threads that can be created. Using this information, a programmer will be able to decide if the library will actually be of use for his project.

The code written for the test cases has been tested for correctness in several ways, first by checking that the output of each IPC problem was indeed correct. Each problem was done by

hand, one at a time. Val grind along with the Memcheck utility for MPI and the rest of the libraries has been used in order to detect any kind of misuse of memory pointers, corrupted memory, null-pointer assignments and memory leaks. This was necessary because multithreading applications are very susceptible to memory corruption and one small corruption will create an enormous amount of errors. No more than the standard C-libraries were used in order to maintain the code as clean and as optimized as possible.

CONCLUSIONS 11.0

The results show that much better efficiency is obtained with higher number of data items due to the divide-and-conquer technique used. Larger sizes of input data are ideally tackled by parallel programming as more elements will be easily divided between threads/processors. Since these are multiprocessing techniques, single-core, dual-core, and even quad-core machines always show a slow performance for all the three test cases. This is because these types of processors do not run processes and threads in symmetrical fashion. This causes the directly-proportional relation between the increase in the number of processes and the execution time. This indicates that an X cored machine is not enough for **true** multiprocessing.

In terms of speed and memory usage, we can say that the performance of OpenMP is slightly better than PThreads in most cases, as mentioned earlier. This might be due to the fact that OpenMP hides the functionality and provides a rather simple interface, taking care of the initialization and manipulation internally by the library.

The performance and memory usage of OpenMPI + OpenMP are also much better than when using OpenMPI + PThreads, again, due to the better performance of OpenMP over PThreads.

Furthermore, last but not least, The MPI, which despite its amazing power, when it comes down to writing distributed applications easily, still runs very slow, uses a large amount of memory, and has overall poor performance in all the test cases. The MPI can be used for distributed applications such as server-clusters or clusters in general. The primary rule when using MPI is to maintain the global communications between each process at a minimum whenever it might be possible, since reducing communications will reduce the overhead caused by the message passing, improving the application performance to a great extent.

In this later case (MPI library), the memory usage increased significantly when the number of processes was incremented. Another problem is that MPI message passing is very slow due to usage of the network protocol to communicate. Messages are very slow and cannot be sent from threads. Only the master thread of the process is able to transmit and receive messages. This reminds me of the previously mentioned rule: "MPI is great, but please *reduce* the global process communication to attain a much better performance" [6].

The performance of a multithreaded application is mutually inclusive with the amount of processing cores available to it; i.e. the more cores that are free, the higher the performance our application will be able to reach.

REFERENCES

1. Arun Kejariwal, Alexander V. Veidenbaum, Alexandru Nicolau, Milind Girkar, Xinmin Tian, Hideki Saito, *On the Exploitation of Loop-level Parallelism in Embedded Applications*, ACM, New York, NY, USA, 2009, ISSN: 1539-9087.

2. Barbara Chapman, Gabriele Jost, Juud Van De Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, 978-0-262-53302-7, Massachusetts Institute of Technology, 2008.

3. Berkeley University of California (2009), *2009 Par Lap Boot Camp – Short Course on Parallel Programming*, Available at: http://parlab.eecs.berkeley.edu/bootcampagenda

4. Banerjee, U., B. Bliss, Z. Ma, and P. Petersen, "A Theory of Data Race Detection." *Proc. Of Workshop on Parallel and Distributed Systems: Testing and Debugging* (PADTAD), pp. 69-78, ACM, Portland, USA, July 2006.

5. Billard, T. 2001. *Resource allocation graph*, Available at the following source web site: http://www.sci.csueastbay.edu/~billard/cs4560/node10.html

6. Blaise Barney. (2010). *Message Passing Interface (MPI)*. Available at the source web site: https://computing.llnl.gov/tutorials/mpi/#Abstract. Last accessed 19 May 2010.

7. Blaise Barney. (2010). *OpenMP*. Available: https://computing.llnl.gov/tutorials/openMP/. Last accessed 19 May 2010.

8. Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, Massachusetts: The MIT Press, 2008.

9. Dagum, L., Menon, R., "OpenMP: An Industry-Standard API for Shared Memory Programming," *Computational Science and Engineering*, 5(1): 46-55, IEEE, January-March 1998.

10. Detecting and Ending Deadlocks - SQL Server 2008 Books Online. Nov. 2009 http://technet.microsoft.com/en-us/library/ms178104.aspx

11. Faculty of Computational & Cybernetics - University of Nizhni Novgorod (2006), *Introduction to Parallel Programming*. Available at the source web site: http://www.software.unn.ru/ccam/mskurs/ENG/HTML/cs338_pp_materials.htm

12. Geraud Krawezik, Franck Cappello, *Performance Comparison of MPI and three Programming Styles on Shared Memory Multiprocessors*, ACM Symposium on Parallel Algorithms and Architectures, San Diego, California, USA, 2003, ISBN: 1-58113-661-7.

13. Hsin-Chu Chen, Alvin Lim, Nazir A. Warsi, *Multilevel master-slave parallel programming models*, Clark Atlanta University, DAAL-O3-G-92-0377, 2006.

14. Jorge Luis Ortega Arjona, *Architectural Patterns for Parallel Programming – Models for Performance Estimation*, Department of Computer Science, University College London, November 2006.

15. Luis Moura E Silvay, Rajkumar Buyyaz, *Parallel Programming Models and Paradigms*, Monash Univerity, Melbourne, Australia 2000.

16. Michael J, Quynn, *Parallel Programming in C with MPI and OpenMP*, 0-07-282256-2, McGraw-Hill, New York 2004.

17. Ryan Eccles, Deborah A. Stacey, *Understanding the Parallel Programmer*, University of Guelph, HPCS' 06, 0-7695-2582-2/06, IEEE Xplore, 2006.

18. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *ACM Transactions on Computer Systems*, 15(4), 391–411, 1997.

19. *Static code analysis - Wikipedia, the free encyclopedia*. 30 Aug. 2009. 30 Aug. 2009. Available at: http://en.wikipedia.org/wiki/static_code_analysis.

20. Timothy Mattson, Beverly Sanders, Berna Massingill, (2004) *Patterns for Parallel Programming*, Addison-Wesley Professional.

21. Yu, Y., Rodeheffer, T., and Chen, W. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. *In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 23 - 26, 2005). SOSP '05. ACM, New York, NY, 221-234. DOI= http://doi.acm.org/10.1145/1095810.1095832