|epcc|

Thread Safety for Hybrid Programming in Thread-as-Rank Model

Gaurav Saxena

August 22, 2013

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2013

Abstract

Hybrid Programming involves the use of a distributed memory programming model in conjunction with a shared memory programming model. Typically MPI is used with a threading library like OpenMP or POSIX threads. They key concept and motivation is to utilize the shared memory across various cores of a SMP node and to speed up the application by exploiting this shared memory for communication and work distribution, thereby reducing the total volume and size of inter-node messages. Due to the presence of multiple threads we must prevent deadlocks, race conditions, corruption of shared data structures by using coarse-grained or fine-grained locks or other lock-free synchronization methods. Making a hybrid application thread-safe involves carrying out the actions stated above in scenarios wheres multiple threads can simultaneously invoke conflicting MPI functions in a MPI_THREAD_MULTIPLE environment. The current popular implementations of the MPI specification do not address a thread as having a separate rank i.e. all threads in a given process share the same rank as the process itself. An orthogonal approach assigns a rank to each thread/MPI process residing in an OS (Operating System) process and hence creates a *thread-as-rank* model. The current project is an attempt to address thread-safety in a hybrid environment and to produce necessary and sufficient rules for making an application thread-safe in a *thread-as-rank* model. It explores thread-safety issues from low, medium and high contention MPI operations e.g., manipulating the *MPI_Info* object, thread-unsafe *malloc()*, point-to-point and collective operations while providing preventive or corrective suggestions and solutions. It contributes to the field of thread-safety by establishing that there is an imperative need to prevent dead-locks, races and protect shared data-structures for a hybrid application to run *correctly* in a MPI_THREAD_MULTIPLE environment and proposing algorithms/pseudo-codes for a subset of such scenarios.

Contents

List of Figures

List of Algorithms

Authorship declaration

I, Gaurav Saxena, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Matriculation no: s1266736

Signed:

Date: 23^{rd} August, 2013

Acknowledgements

I profusely thank *Daniel Holmes* for his unending support and encouragement throughout the project work. The resting point of all my doubts were the interesting discussions I had with him during the course of the project. With infinite patience and a genuine will to help and clear my doubts, he is a humble academician who I have always looked upto and would love to follow.

I remain indebted to *Fiona Reid* for adjusting her busy schedule for doubt-sessions, painstakingly reading the project report and offering valuable suggestions.

I would further like to thank *William D. Gropp* for taking out the time to solve a set of my doubts.

Thanks is a small word for my amazing friends whose never-ending support carried me across oceans and the barriers of life. To my survival kits in Edinburgh-India - *Rahul Arora, Vasuda Arora, Sachin Gupta, Rashmi Shakya, Swapnil Laxman Gaikwad, Kanika Malik* and *Jyoti Balwani*- you people are too amazing and deserve better than the best.

In the end and the most, to my *parents*, *family* and my *grandmother* who always believed in me and forged my character.

Chapter 1

Introduction

Hybrid programming involves the use of MPI (Message Passing Interface) [1] programming along with an implementation of a shared memory programming paradigm like OpenMP (Open Multi Processing)[2] or POSIX (Portable Operating System Interface) [3] threads (Pthreads). OpenMP is the de-facto standard for shared memory programming although internally OpenMP threads compile to POSIX threads. The reason and motivation for using a combination of these is to enhance the performance of communication and computation by simplifying work-distribution on shared memory nodes in a distributed symmetric multiprocessor architecture/environment. Having several threads running (one thread per core) per OS (Operating System) process reduces the total number of processes and the total number/size of inter-node messages. Due to the presence of optimized shared memory communication, the message start-up cost and latency decrease and may enhance the efficiency of execution of an application. On the other hand, the contention for common resources due to several threads operating in parallel may adversely affect the performance and possibly need more effort in terms of coding to synchronize the contention. Due to the non-deterministic timing and numerous possible permutations of thread schedules, simultaneous conflicting MPI function calls on different threads can corrupt a shared data structure, an object may be freed/deleted incorrectly in a race between threads, deadlocks may arise etc. and hence resulting in a need for making the application thread-safe. The performance gains, if any, by hybridizing a code should be thoroughly weighed against the induced overhead and the complexity of coding from a software-engineering perspective.

Most of the current popular MPI implementations like *MPICH2* [4], *OpenMPI* [5] etc. do not address the thread by a separate rank and all the threads in a process share the same rank as the process itself. The opposite of this approach is to make each thread addressable by a separate rank and hence create a *thread-as-rank* model where a thread is equivalent to an MPI process residing in an OS process. The *thread-as-rank* model is supported by implementations like *TMPI* (Thread MPI) [7], *TOMPI* (Thread Open MPI) [8] and *McMPI* (Managed code

MPI) [9]. *USFMPI* [10] is an implementation of MPI which uses a separate thread for communication and *MiMPI* [11] is a thread-safe implementation but does not assign a rank to a thread separately. The MPI specification standard does not acknowledge the thread-as-rank model and all its specifications concern a model where threads share the rank of the process (Section 12.4.1 of [1]).

This project aims to highlight thread-safety issues, scenarios and formulate preventive/corrective measures in the form of advice and concrete solutions at an application level while providing some insight at the implementation level in a thread-as-rank model. Due to the exhaustive list of functions and interactions between functions in MPI, it is not possible to examine each and every function and its possible interactions. The unique contribution of the project is the creation and exploration of some thread-safe algorithms to prevent shared data-structures from becoming corrupted or unusable and to prevent or solve such situations. Further, whenever such an algorithm cannot be devised to solve the problem under consideration, code snippets in the C language try to bring out the thread-unsafe scenario. This project gives an insight as to how to think about the problem of thread-safety in a multi-threaded environment. It also emphasizes that the difference between a thread-unsafe program and a thread-safe program is correctness, which needles to say is the prime requirement in solving any problem. As an ideal consequence of program correctness, it is unwise to compare the performance of a thread-unsafe application with its thread-safe sibling. The road-map of the project is as follows:

Chapter 1 mentions and introduces various topics like hybrid programming, thread-as-rank model, thread-safety, MPI process, OS process which are explored in detail in the subsequent chapters. It outlines the layout of the report and emphasizes the contribution of the project towards the field of thread-safety in a *thread-as-rank* model.

Chapter 2 creates a background by introducing the terms and concepts in detail and puts the problem into perspective by describing previous work carried out in this field. It emphasizes as to why this problem must be addressed for correctness of applications when multi-threaded shared memory and distributed memory programming models go hand-in-hand towards building future systems of the *Exascale* level. Further, this chapter also explores some basic problems with threads which are more general and are not necessarily specific to MPI.

Chapter 3 discusses various thread-unsafe scenarios beginning with a low contention operation involving the *MPI_Info* object, a medium contention non-reentrant ANSI C function *malloc()* and high contention operations like the point-to-point, collective and Remote Memory Access (RMA) operations. It is not always possible that a thread-unsafe scenario is preventable or solvable by using lock-based or lock-less synchronization methods (to enhance concurrency among non-conflicting multi-threaded accesses). This is especially true when the user violates the MPI specification standard and produces a completely erroneous program. A thread-safe strategy to dynamically assign ranks to newly spawned threads is dis-

cussed as an alternative to distribution of ranks using MPI endpoints along with the advantages that it offers as compared to the latter.

Chapter 4 makes an attempt to quantitatively analyze the algorithms which have been introduced in terms of time complexity whilst touching upon the space requirements of data structures i.e. variables which the algorithm utilizes to make the scenario thread-safe. The former however, has been dealt with in more detail as it forms the most important part of complexity analysis as far as performance measurement is concerned. In a few cases the analysis supports the *Occam's razor* i.e. among all the solutions which are feasible, the simplest and most easily implementable is the best. Further, an abstract reference model is constructed and proposed to approximate the lines of source code needed to make a given application/code-snippet thread-safe.

Chapter 5 concludes the project and gives the major findings, inferences, a critical evaluation and conclusions which can be derived from the problems and their proposed solutions while also presenting some ideas for future work.

Chapter 2

Background and literature review

The traditional use of MPI [1] involves spawning one process per CPU core and communicating between various MPI processes using explicit MPI communication calls even when some of the cores have shared memory between them. Hybrid programming combines the use of threads on a shared memory system with processes that span one SMP (Symmetric Multiprocessor) node/multicore chip. It leads to a reduction in the number of heavy-weight processes and use of light-weight MPI processes/threads to optimize communication and distribute computational tasks in an intuitive way. There are several methods of creating threads but two libraries which are seemingly popular and used in majority of applications with MPI are OpenMP [2] and Pthreads [3]. OpenMP offers additional automatic work distribution constructs and user-friendly implementation of locks, critical sections, atomic clauses etc. and has become the indisputable standard for shared memory programming. This mixed mode of programming/hybrid programming can be an advantage when MPI scales poorly due to an increasing number of processes, has irregular distribution of load leading to load imbalance, becomes communication dominated or has mammoth amounts of replicated data. A computationally intensive multi-dimensional domain decomposition in problems like a 3-dimensional *Fourier* transform or poor intra-node optimization [6] can also lead to performance deterioration with a pure MPI process approach. Figure 2.1 shows a typical architecture for running hybrid codes where one MPI/OS process per node (running four threads) and one thread per SMP core is run.

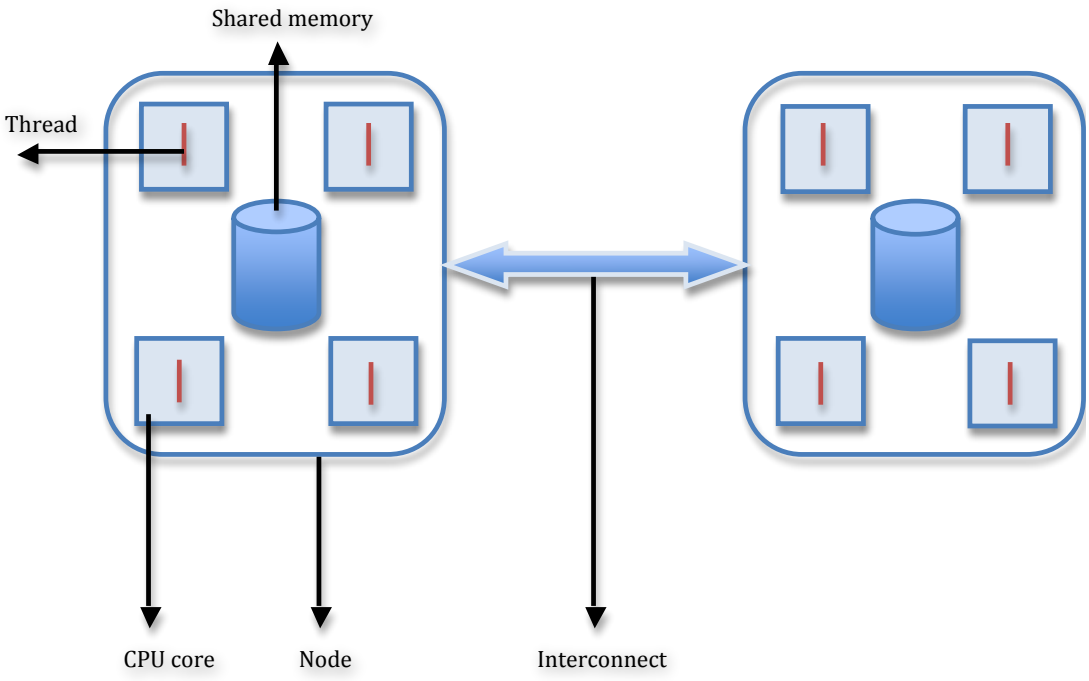

Figure 2.1: Two SMP nodes running one MPI process per SMP node and one thread per core

2.1 Threads and MPI

A thread is a lightweight process having its own program counter, registers, stack and shares the same address space as the process in which it is created [13]. Threads are desirable in shared memory systems due to the low cost of creating them and a lesser overhead involved in context switching as compared to a process. There are majorly two types of threads: *Kernel* threads and *User* threads - threads running in kernel space or scheduled by the kernel and threads created by the user application, respectively.

According to the MPI specification, implementations should support a maximum of four levels of thread support [1] (represented by four named constants).

- MPI_THREAD_SINGLE allows only a single thread of execution and is the same as a process/single threaded process initializing the MPI library with the function *MPI_Init()*.
- MPI_THREAD_FUNNELED lets the user create multiple threads but only the master thread i.e. the thread which initializes MPI can invoke MPI library functions.
- MPI_THREAD_SERIALIZED allows any thread to call an MPI function

but no two threads can call MPI functions at the same time i.e. simultaneously.

- MPI_THREAD_MULTIPLE is the most complex and yet the most intuitive support level that allows any number of threads to call MPI functions simultaneously.

The thread support level in MPI is initialized through a call to *MPI_Init_thread()* function whose syntax in C language is shown below:

*MPI_Init_thread(int *argc, char **argv[ ], int required, int *provided)*

required is a field which is set to any of the four named constants above and *provided* is an out parameter which is set equal to *required* (requested level) if the MPI implementation supports the *required* thread level or the greatest level below the *required* thread level, if it is not supported. Not all MPI implementations support MPI_THREAD_MULTIPLE in general and hence a correct program must query the returned thread support level.

2.2 Thread-as-Rank vs. process model

Most MPI implementations do not provide separate ranks to threads and thus support the rank-less/process based model from the perspective of threads. Tags used in point-to-point communications can be used to distinguish between threads because all threads share the rank of the process in which they reside. Tags are not inherent identifiers (like the thread ID or the rank of a process) and their attachment to the thread is not created by the MPI implementation but by the application programmer. For example, two threads can post individual *MPI_Receive()* MPI calls with an identical signature and either of them can receive a message meant for the other thread. As an alternative approach if the threads can be made addressable by rank then we move from a black box model towards a glass box model and an OS (Operating System) thread can be mapped to a single MPI process i.e. ranks specified in point-to-point communications will refer to individual threads and not OS processes. The MPI specification states that a process cannot call *MPI_Init_thread()* more than once. If each thread has a separate rank then it should be possible to call *MPI_Init_thread()* multiple times and thus violating the condition stated in the MPI specification. Hence in a thread-as-rank model a thread should be treated as a separate MPI process that resides in an OS process [9]. Further the threads are allowed to define their own private variables/data-structures in addition to sharing the variables/data-structures global to the OS process. Figure 2.2 illustrates the process and the thread-as-rank model.

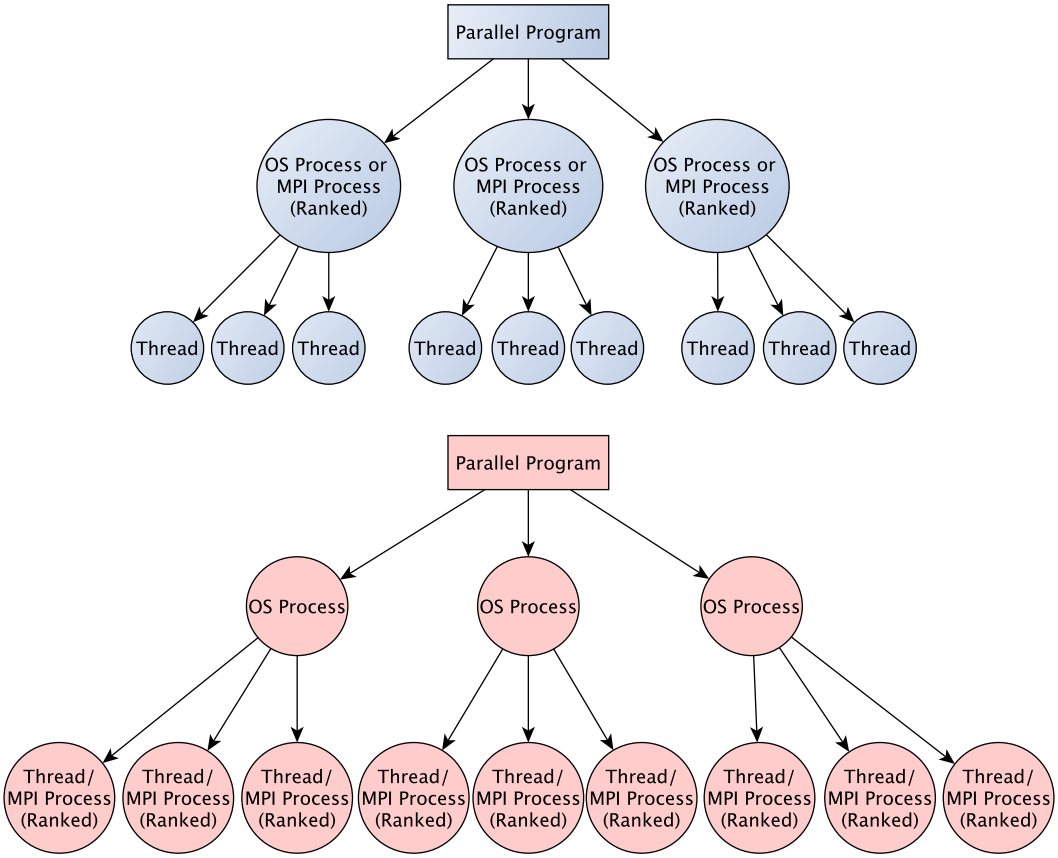

Figure 2.2: Process model (top) vs. Thread-as-rank model (bottom)

An implementation of MPI named *TMPI* [7] (Thread MPI, last update: May 2002) that implements the *thread-as-rank* model uses a technique called *NSD* (Node Specific Data) as opposed to *parameter-passing* or *array-replication*. This technique is based on the concept of *TSD* (Thread Specific Data) in POSIX threads which is used to convert all global and static data of an OS process to private data of the thread and hence, in effect, replicates all the variables and data structures per MPI process/thread. Each global variable is associated with a pointer sized value and a uniform key across all threads which identifies a particular variable. Given a key value, a thread can retrieve the value of its private copy of the global variable residing with it. This approach is not useful when the size of the variables and data-structures being replicated is large or/and when threads frequently modify shared data, the changes to which naturally result in exchange of large number of messages to make the thread private copies of global variables/data-structure coherent. Thus, *TMPI* attempts to create a complete OS process out of a thread and hence, does not justify the usage of threads as means to optimize shared memory communication/computation. Moreover *TMPI* does not support the thread level MPI_THREAD_MULTIPLE.

McMPI (Managed-code MPI)[9] is an MPI library written completely in *C#* lan-

guage. *McMPI* extends the four thread-support levels specified in the MPI standard by adding two more thread support levels. MPI_THREAD_AS_RANK level is an extension of the MPI_THREAD_SINGLE level when *all* the threads (MPI processes addressable by individual ranks) in an OS process are initialized using *MPI_Init()* (or *MPI_Init_thread()*). Another level of thread support called MPI_THREAD_FILTERED is an extension of MPI_THREAD_FUNNELED and restricts the invocation of MPI calls only on threads initialized with *MPI_Init()* (or *MPI_Init_thread()*) while allowing some threads to remain uninitialized. This MPI library does not replicate the global or static data of the OS process and allows threads/MPI processes to be used in the most natural way to share global variables/data structures while letting the threads define their own thread-private variables/data-structures.

2.3 Thread-Safety

When operating simultaneously in a shared memory environment threads can cause disruption of data structures, blocking of other threads, deadlocks/livelocks etc. due to variable scheduling and can in effect make the application thread-unsafe. The thread-safety specification in MPI provides advice to application developers to avoid such scenarios. The current popular implementations like MPICH2 [4] and OpenMPI [5] being used to produce massively parallel applications are not thread-safe when the thread support level is raised to MPI_THREAD_MULTIPLE. Two basic rules for thread safety are [1]:

1. **Rule 1**: MPI calls are thread safe if an order can be imposed on the conflicting calls i.e. the two calls execute serially in any possible order.
2. **Rule 2**: A thread giving a blocking call blocks only the calling thread and not any other thread.

The two main issues that must be dealt with in order to make an application thread-safe are: serializing conflicting accesses to *global states* and making the library and user functions *re-entrant*. Most implementations leave it to the user to serialize accesses to common data structures and hence the user bears a partial responsibility of making the application thread-safe.

A support level of MPI_THREAD_MULTIPLE by an implementation does not mean that the application is thread-safe, it just indicates that multiple threads are allowed to call MPI library functions simultaneously. An appropriate analogy can be that of a runway being used by multiple aeroplanes. It is the responsibility of the pilots (application developer) to make sure that only one plane (thread) uses the runway (common variable/data-structure) at a time (serialized access) and prevent any possible accidents (dead-locks/races/corruption of global states).

Quasi-thread-safety [12] implies leaving the synchronization between threads to the user as opposed to *complete-thread-safety* where the synchronization is hidden from

the user and implemented completely in the library. *MiMPI* (Multithread implementation of MPI) [11] claims to be thread-safe but does not describe the process that makes it thread-safe. Its architecture uses three layers: the first layer hides the hardware details from the upper layers, a lightweight communication microkernel layer called *XMP* (interface consists of: *XMP_Init(), XMP_Finalize()* and *XMP_Request(struct request*)*) and an MPI Interface layer.

MPI specifies that it is the responsibility of the user to prevent race conditions among threads when they post conflicting MPI calls which interfere with each other [1][23]. For example, the user must ensure the correct ordering of collective communications on the same communicator and file handles on different MPI processes that are part of the communicator. To understand the thread-safety requirements of functions in *MPICH2*, more than three hundred functions have been examined by researchers and classified into a particular thread-safety class based on a function's primary requirement [23], but the details of what features in them require thread-safety or the solutions which have been used to make them thread-safe largely remain hidden. The classes form a list of intersecting classes i.e. a function may have thread-safety requirements falling in more than one class. A brief explanation of these classes is as follows [23]:

- **Comm I/O**: The function needs to access the communication or I/O system in a thread-safe way. Majorly the point-to-point and file modification functions constitute this category. For e.g., *MPI_Send(), MPI_Win_unlock()*, and *MPI_File_write_at()* etc.
- **Collective**: The thread-safety in this class concerns the order of invocation of collectives on different threads for the same communicator. They also require *Comm I/O* thread-safety but this is handled by separate locks protecting the critical section implementing communication. For example, *MPI_Alltoall(), MPI_Comm_accept()* and *MPI_File_close()* etc.
- **Access Only**: These functions access read-only fixed data like a communicator size or might modify a value like setting the name of a communicator. Thread-safety is required because a race condition might delete the object before it is accessed. For example, *MPI_Cart_coords(), MPI_Comm_rank()* and *MPI_Get_count()* etc.
- **Update Ref**: This class includes functions which return a handle to an object resulting in an increase in the reference count of the MPI object. For example, *MPI_Win_get_group() and MPI_File_get_view()* etc.
- **Read List**: Functions in this category return an attribute or info value. A race condition with another thread may result in data corruption due to update (write) or deletion of the attribute. For e.g., *MPI_Attr_get(), MPI_Info_get()*, and *MPI_Win_get_attr()* etc.
- **Update List**: Functions can update a particular list of attributes and would need thread-safety as multiple threads can invoke the same function resulting

in corruption of values. For example, *MPI_Attr_put()* and *MPI_Info_set()* etc.

- **Allocate**: This class of functions need some form of dynamic memory allocation and may require thread-safety due to a non-re-entrant routine like *malloc()* function in the ANSI C language. For example, *MPI_Bsend_init(), MPI_Group_free()*, and *MPI_Type_vector()* etc.
- **Own**: These functions require some special thread-safety management due to the presence of global states. For example, *MPI_Buffer_attach()* and *MPI_File_seek()* etc.
- **None**: The functions in this category have no thread-safety issues in a correct program. For example, *MPI_Wtick()* and *MPI_Address()* etc.
- **Other**: These form a special category and have different thread-safety requirements. For example, *MPI_Init_thread()* and *MPI_Abort()* etc.

2.4 Protecting the critical section of code

The critical section of code containing common variables/data-structures must be prevented from being executed by multiple threads simultaneously and hence either *lock-based* or *lock-free* approaches (making use of atomic operations) must be used. Locks may vary in their level of implementation i.e. granularity of a lock can vary. For example, a *global-lock* locks the critical section for the entire period of execution of a function by a thread but gives up the lock before a blocking I/O/communication operation starts. It again obtains the lock after the blocking operation is complete. A *brief-global* lock holds only the critical section of a function as and when required and not for the entire duration or complete execution of the function. This permits greater concurrency but at the same time calls for a more complex logic in terms of implementation. A *per-object* lock allows different critical sections to be locked for different objects or different classes of objects and permits even greater concurrency. Last and the most complex class of algorithms use *lock-free* approaches implemented though *atomic* operations like *Compare-And-Swap, Test-And-Set* and *Fetch-And-Add* (see chapter 3) etc. to permit the highest level of concurrency among threads [14].

2.5 Specification, implementation and application

In general several entities are involved in writing a complete thread-safe program. Figure 2.3 shows the entities involved and the arrows show the dependence of one entity on another.

First the specification of the MPI standard should be thread-safe. Upon examination of the MPI specification only two functions *MPI_Probe()* and *MPI_Iprobe()* were found to have thread-unsafe semantics [22] and hence should not be used in multi-threaded scenarios or should be made thread-safe. The latest MPI specification MPI 3.0 [1] incorporates two variants of the probe function, namely *MPI_Mprobe()* and *MPI_Improbe()*, which have thread-safe semantics. These two functions guarantee that a specific message probed with either of these is the only one that will be received by a subsequent call to *MPI_Mrecv()* or *MPI_Imrecv()*. This eliminates the possibility of another thread receiving the message that was probed by a call to *MPI_Mprobe()* or *MPI_Improbe()*. As a contradiction, detailed studies by researchers [23] bring out the thread-safety needs and categorize functions according to their thread-safety requirements into various classes (see section 2.3).

Secondly, the implementation must adhere to the MPI specification [1] and hence create functions, modules and layers which are thread-safe. Some implementations claim to be thread safe but there is no test-suite which can validate their claim [23]. Some implementations which use *MPI_Get_count()* to return the length of the last message received are definitely not thread-safe as between receiving a message and calling *MPI_Get_count()*, another thread can receive a message and the length of the latter message could be returned to the thread calling this function (section 3.2.5 of [1]). As another example an algorithm used for finding the *context-ID* of a new communicator in a multi-threaded environment when *MPI_Comm_dup()* is executed in *MPICH2* [4] was found to be thread-unsafe [23]. Further, a recent bug report concerning *MPICH2* identifies *MPI_Bsend()* and *MPI_Ibsend()* as not being thread-safe as the static data in a file *bsendutil.c* is prone to becoming corrupted and there is no lock for MPIU_THREAD_GRANULARITY_PER_OBJECT [28]. A description of these terms and concepts is beyond the scope of the project. The purpose of citing the bug is to bring out the fact that even if the specification is thread-safe, an implementation may not be thread-safe.

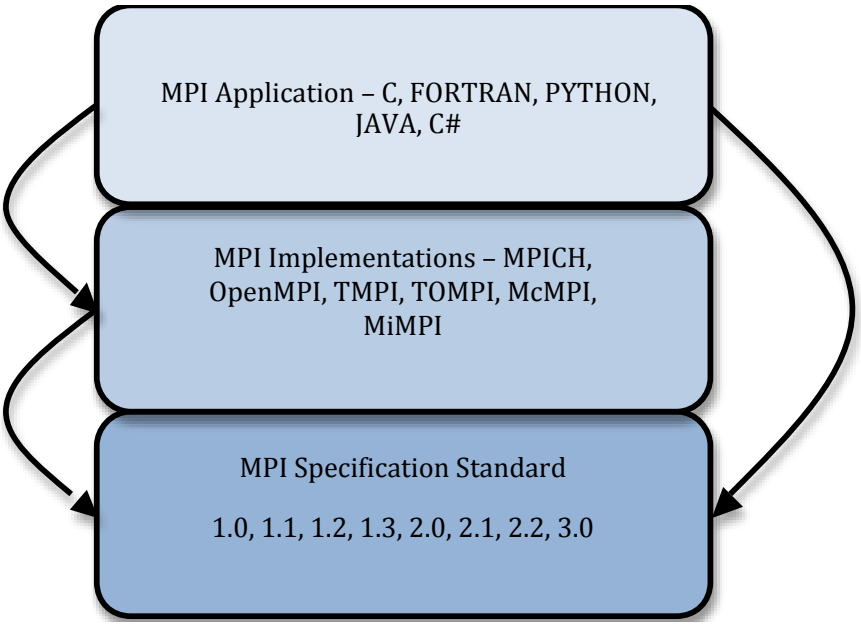

Figure 2.3: Three layers/components of building any MPI application

Thirdly, the application needs to be thread-safe. For example, it is possible to receive any message for a particular process by any thread which posts a matching receive in a process model. The application programmer must be careful if there are competing threads posting receives in the same process as a thread can starve in such a scenario. The user must also be aware of the protocols being used internally by MPI implementations and optimization in a shared memory environment to produce a portable correct program. A wrong order of receives using the wild-card MPI_ANY_SOURCE could result in a dead-lock (see section 3.4.2 chapter 3). Further, the serialization of simultaneous conflicting calls issued on multiple threads of the same OS process is solely the responsibility of the user [1].

2.6 Notion of correctness of programs

We define three types of program correctness levels and discuss thread-safety in accordance with the type of program. We feel that this is useful because it is not always that a program can be made thread-safe or there is no use of making a program thread-safe if it is inherently erroneous. The discussion assumes non-overlapping thread schedules and instantaneous execution of a statement/group of statements by a thread (analogous to an apparent instantaneous execution of atomic operations). Assuming n threads in an SMP, suppose thread i calls f_{n_ik} MPI functions in the k^{th} SMP. Let ϕ (specifically ϕ_{nk}) denote the total number of possible valid permutations of all functions of all nk threads of the system. The function ϕ is not a simple permutation or combination of functions of nk threads. Some restrictions like the serial ordering of functions on a particular

thread cannot be violated i.e. the function ordering on thread i of the k^{th} SMP must be $f_{1k}, f_{2k}, ..., f_{n_ik}$ but these functions can be interspersed with functions from any other SMP/SMP's to form a valid schedule of threads. For example a valid schedule can be: $\langle f_{12}, f_{11}, f_{21}, f_{31}, f_{22} \rangle$ for *any* two threads of SMP's numbered 1 and 2 where one thread calls three functions in all and the second thread calls two functions in all. We hypothesize that the function ϕ (specifically ϕ_{nk}) can be modeled as:

$$\phi_{nk} = \prod_{j=1}^{k} \phi_j (\sum_{i=1}^{n} f_{n_ij}) \tag{2.1}$$

ϕ_{nk} gives the total number of thread schedules for n threads on k SMP's i.e. the total possible thread schedules for a total of nk threads. The value of ϕ cannot be easily computed even for a small number of threads and the number of functions that are invoked but it helps us to mathematically express the notion of correctness of programs. Let ϕ'_{nk} denote the number of permutations of thread schedules which lead to a successful termination of the program under consideration. Let a ratio α be defined as:

$$\alpha = \frac{\phi'_{nk}}{\phi_{nk}} \tag{2.2}$$

The levels of program correctness can be expressed in terms of the ratio α as follows:

1. **Completely-correct programs**: The ratio $\alpha = 1$ i.e. the program executes successfully for all thread schedules and is hence completely-correct. The program under consideration has no application thread-safety requirements.

2. **Quasi-erroneous/partially-correct**: The ratio $\alpha \in (0, 1)$ i.e. there is at least one permutation which does not lead to correct execution or there is at least one permutation for which the program executes correctly. Programs for which $\alpha \to 0$ can be said to be *quasi-erroneous* and are harder to make thread-safe. For $\alpha \to 1$ the program falls in the category of a *partially-correct* program and requires less effort for making it thread safe as compared to a *quasi-erroneous* program. It is this class of programs that require thread-safety.

3. **Completely-erroneous**: Programs for which $\alpha = 0$ i.e. programs for which no permutation of thread schedules takes the program to a successful completion without producing an error have no need for thread-safety as they are *completely-erroneous*.

2.7 Thread-Safety issues in implementation

The implementation must ensure that no global variables in re-entrant functions are incorrectly modified and produce erroneous results when being accessed by multiple threads. Some general thread-safety issues, which also are a concern in MPI applications, are as follows:

- **Updating reference count of objects**: MPI guarantees that an object will not be freed/deleted until its usage is complete. For example, a buffer or user defined data-type being used in a non-blocking point-to-point call can be freed immediately after the call returns, but MPI guarantees that it will be freed/deleted only after all its uses are complete i.e. if it has been used by any other thread in a MPI call, then it would be freed/deleted only after the call/use is complete. A reference counting method is used for such cases and a count for each object is incremented each time this object is used and decremented each time the object's use or the call using it is complete. In a multi-threaded environment the reference count must be incremented or decremented atomically [23].

- **Thread private memory**: It is possible that a global variable may have different values on different threads and hence each thread needs to maintain a private copy of the global variable. Functions like *pthread_getspecific()* in the Pthreads package serve the same purpose but even threading libraries not having a similar function should be able to maintain these private copies of global variables. As an example consider a MPI function that calls another MPI function internally. Such is case with *ROMIO's* (*parallel MPI-IO implementation*) function *MPI_File_open()* which calls *MPI_Allreduce()* internally using the communicator passed by the user in the former call. In case of an error, the error handler for *MPI_Allreduce()* should not be invoked and instead the error handler for *MPI_File_open()* should be called. This creates a need to maintain the nesting level of a function so that the top level error handler should be called and not the error handler of the current function. An implementation cannot simply reset the error handler of the nested function before invoking it because another thread might invoke the same function and expect the error handler to become active, if an error occurs. Hence, the solution is to maintain the depth of nesting of functions for each thread and not invoke the error handler if the nesting depth is greater than one [23].

- **Thread cancellation/failure**: A problem occurs with threads which hold a lock and fail or are deliberately canceled with a function like *pthread_cancel()*. These cases give rise to a non-deterministic behavior and the application may crash, hang or exhibit a *byzantine*[1] fault. A plausible solution is to use

[1] A scenario where the same input to a system/application produces different responses at different times.

lock-free data-structures but the complexity of implementing them is much higher as compared to a *lock-based* approach [23].

- **Fine-grained Vs coarse-grained locks**: An implementation using fine-grained multiple locks (and *lock-free* methods) might fall in a deadly embrace when two threads hold the locks required by each other. Since there are usually multiple locks in fine-grained approaches, it is not always possible to request locks in the same order on two threads to (increase the concurrency) avoid a deadly embrace [23].
- **Restarting/Notifying a thread**: It is possible that a thread which is waiting for an event to occur (e.g., a thread blocked in *MPI_Recv()*, waiting for a message) may yield to another thread so that the other thread can proceed. The first thread must be woken up by some signaling mechanism to resume the execution of the function but it is difficult to *notify* the thread if a low latency method for communication is being used by threads/MPI processes on the same SMP node [23]. The MPI specification recommends (not imposes) that an implementation should not use the signals SIGALRM, SIGFPE, and SIGIO. Further, the signals should be caught/handled on threads that do not invoke MPI functions to prevent the mixing of signal handling with the execution of MPI library functions (section 2.9.2 of [1]).

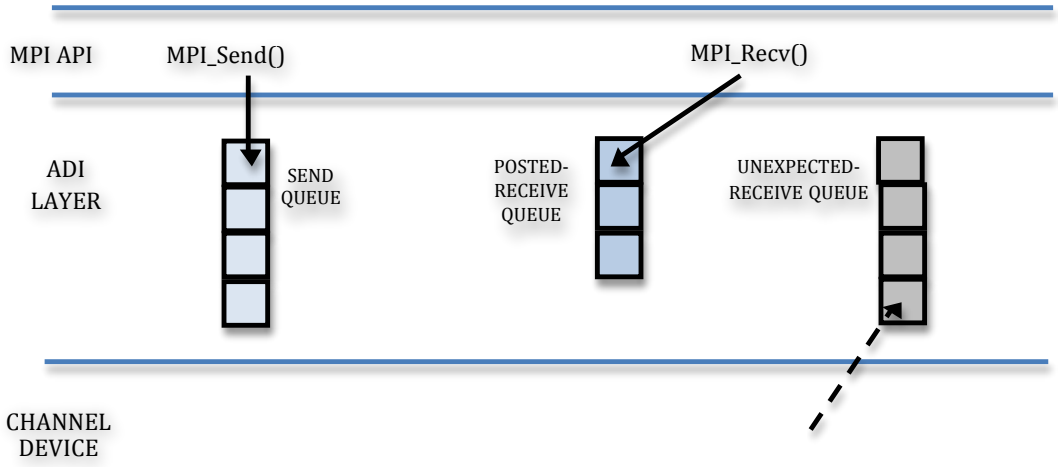

Figure 2.4: Send and Receive queues in Abstract Device Interface layer

- **Queue locking in ADI (Abstract Device Interface)**: When *MPI_Send()* or *MPI_Recv()* is posted, a data-structure representing the instantaneous description of the communication call called *Transfer Description* is constructed and an *opaque* (not accessible by a user) pointer called the *handle* pointing to it is posted in the appropriate queue in the *ADI* (Abstract Device Interface) layer [26]. The *ADI* layer implements the message passing engine and acts as an interface between the MPI *API* (Application Programming Interface) and the *Device* layer.

There are three types of queues: A *Send-queue* storing the send handles, a *Posted-receive* (henceforth abbreviated as *P-r*) queue storing the handles for the receives which are posted and an *Unexpected-receive* (henceforth abbreviated as *U-r*) queue which contains handles for receives for which a request has arrived but the receives have not been posted as yet. The communication device at the receivers side locks both the *P-r* and the *U-r* queues, searches the *P-r* queue for any matching receives handles and if not found, constructs and puts a request *handle* in the *U-r* queue. Locking both the queues is essential as a posted receive might be searching the *U-r* queue to match any *handles* and an incoming request could be searching the *P-r* queue to match any *handles* and hence would not find each other's entry thus resulting in an incorrect insertion of *handles* in the *P-r* and *U-r* queue, respectively. There is also a possibility that when multiple threads post identical receives, if both the queues are not locked together, the threads might traverse the *U-r* queue together and find the same *handle* pointing to the same *transfer description* at the same time causing it to be returned to both the threads (and hence an error). In general a *MPI_Recv()* operation is more complex than a *MPI_Send()* operation and the use of wild-cards MPI_ANY_SOURCE and MPI_ANY_TAG further complicates the process of receiving. Figure 2.4 shows the various queues.

In a *thread-as-rank* model a per-thread queue can be implemented in a thread-safe manner to enhance the performance by increasing the concurrency. Each thread has two receive queues, the *P-r* and the *U-r* queue (in addition to the *send-queue*). This is possible as the send and receive calls specify the thread rank and hence the *handle* of *transfer description* can be posted directly in the queue specific to a thread. The incoming request locks and searches for a handle in the *P-r* queue and posts it in the *U-r* queue of the thread, if it fails to find the handle. Locking both the queues (*P-r* and *U-r*) can be avoided if *only* the owner thread of the queue is allowed to access it, otherwise both the queues must be locked here as well. The former strategy thus, allows for a completely lock-free implementation [32]. The increase in concurrency comes from having per-thread queues and hence results in reduction of contention and increase in performance. Even a receive posted with a wild-card need not be synchronized with the other queues as the *thread-as-rank* model uses a more fine-grained *push* model[2] controlled by the send operation in the sense that all sends have clearly defined receivers, which are threads. A thread can decide using some additional criteria as to which handle is to be picked to match a wild-card receive.

[2]In a rank-less thread model the senders must specify the rank of a process and hence push data to a unique receiver. A receiver using wild-cards can pull data from any source. Thus, MPI uses a *push* model rather than a *pull* model [1].

2.8 Endpoints in MPI

The concept of endpoints in MPI has not been incorporated as yet in the MPI 3.0 specification standard [1] and relates to relaxing the one-to-one mapping between MPI processes and threads i.e. allowing threads to have individual ranks and decoupling them from processes by making them an MPI process in their own right. An endpoint has been formally defined as a set of logical or physical resources which help an entity like a thread to communicate independently i.e. one or more threads can attach to an endpoint and communicate without the need for a process interface. Further, this strengthens the interoperability between MPI and data-parallel, offload programming and node-level parallelism models [18]. The current MPI specification [1] interface for threads needs the use of tags and communicators for efficient communication. However, tags cannot be used for collective communications and creating one communicator per parent process thread has high computational cost. Further, the use of wild-cards in receive operations makes the matching non-deterministic. The reasons described above serve as motivation for creating and formalizing a concept like endpoints.

Endpoints offer two interfaces for attaching a rank to a thread: *static* and *dynamic*. The *static* interface approach allows the user to request a number of endpoints just once during application launch or initialization. If done during the launching of the application then all the endpoints are contained in the MPI_COMM_WORLD communicator. If the request for endpoints is made during initialization then a separate communicator called MPI_COMM_ENDPOINTS is created containing all the processes and the endpoints. The endpoints can either be represented by objects called *MPI_Endpoint* or simple indexes represented by integers and returned as an array of objects after initialization or possess the facility of querying for a handle using an appropriate routine. The following is the syntax for creating endpoints:

*int MPI_Init_endpoints(int *argc, char *argv[], int count, int tl_requested, int *tl_provided)*

count is the number of requested endpoints, *tl_requested* is the requested thread level and *tl_provided* is the thread level which is returned by the application. The endpoints can be attached to a thread with another function having the following prototype:

int MPI_Comm_endpoint_attach(MPI_Comm comm, int index)

where the index ranges from 0 to *count* - 1. A static approach is not flexible and can have disadvantages like not being able to operate with dynamic multi-threading libraries and restricted to having equal number of endpoints for each process even when the number of threads in each process may vary considerably. The inability to make a good initial guess about the number of endpoints by leaving it to the user instead of automating the procedure is another drawback. The endpoints cannot be transferred from one process to another process even when the threads

in one process have completed execution.

The second method to create endpoints is the *dynamic* interface method which creates a new communicator and returns an array of handles to this communicator. Ranks in the new communicator are ordered serially as the ranks in the parent communicator. The routine for the same has the following syntax [18]:

int MPI_Comm_create_endpoints(MPI_Comm parent_comm, int my_num_ep, MPI_Info info, MPI_Comm out_comm_hdls[])

The last argument is an array of *my_num_ep* handles to a single communicator and the i^{th} handle corresponds to the i^{th} rank. A complete description of MPI endpoints is beyond the scope of the current project but it suffices to say that the *MPI Forum* is thinking and proposing concepts pertaining to threads being allocated ranks for the purpose of carrying out communication that does not use the process interface.

Chapter 3

Thread-Safety scenarios

This chapter explores several scenarios which require thread-safety. It is not possible to come up with an exhaustive list of such scenarios due to the large number of functions in the MPI specification [1] and their interactions. The cases that we discuss are not stand-alone or isolated cases but meant to represent a general category and can further be combined with other scenarios to cater to the thread-safety requirements of a new code-snippet/application.

3.1 Thread-Safety with *MPI_Info*

MPI_Info is an opaque[3] object [1] which helps to provide hints for improving the efficiency of an operation. The *info* object consists of sets of unordered key-value pairs, where each key and value is a string. In addition to the keys defined in the MPI specification, the implementation can use its own sets of keys and values. Implementations should support the maximum length of a key and value upto MPI_MAX_KEY_VALUE (minimum value 32 and maximum value 255) and MPI_MAX_INFO_VALUE. Further, the functions should be able to ignore a key if the key is not implemented by an implementation, hence making it portable across platforms.

The problem with a shared *MPI_Info* object is that multiple threads may simultaneously try to access the object and hence possibly corrupt the data/data structure. For a pure read operation no thread-safety is required but due to the presence of functions like *MPI_Info_set()*, *MPI_info_delete()*, and *MPI_Info_free()*, a thread may update/free a shared *info* object which is being simultaneously operated upon (like being read) by another thread. For example, a thread sets three keys for a shared *MPI_info* object while a second thread invokes *MPI_Info_free()*.

[3]Objects in the system memory managed by MPI and whose size and shape is not visible to the user. They are accessed in user space using *handles*.

Such a scenario is depicted in Figure 3.1 where an 'X' symbol shows the invalid operation at that stage of execution.

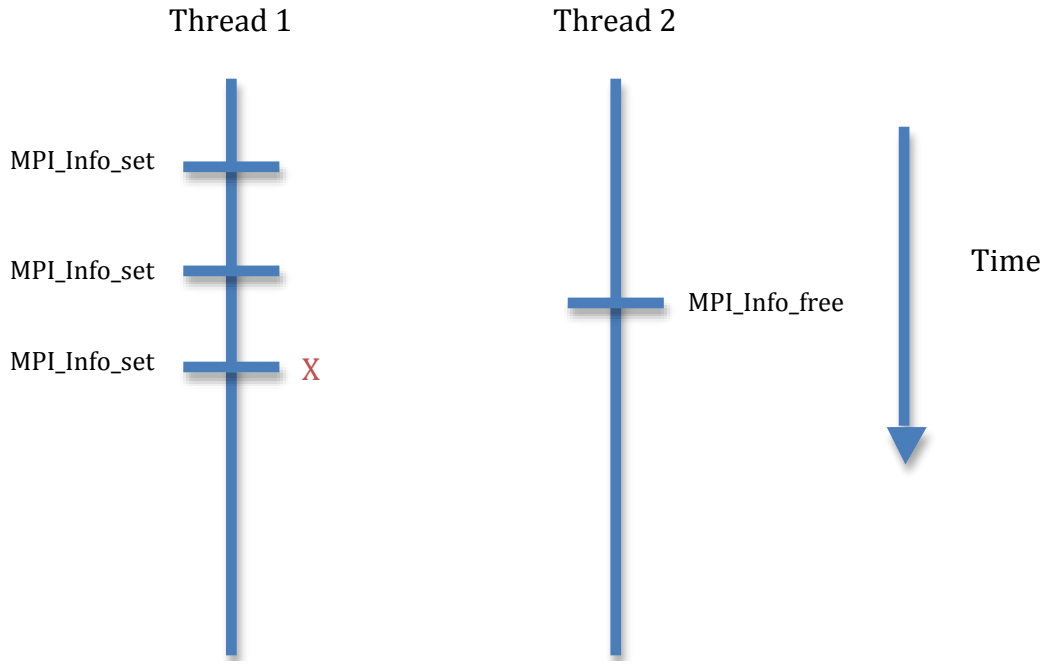

Figure 3.1: Thread unsafe scenario with *MPI_Info*

A simple locking mechanism does not help because if the second thread in Figure 3.1 executes *MPI_Info_free()* before the third call to *MPI_Info_set()* on the first thread, it destroys the *info* object and hence when the first thread tries to set the value of any key in it results in an error. This requires that in addition to obtaining a per-object lock [14] on the *info* object, each thread after obtaining the lock must check whether the object exists or not. The two operations cannot be exchanged because there is a possibility that the object is deleted immediately *after* a thread checks the object's existence and *before* the lock is obtained. This thread-safe code is shown in Algorithm 3.1.

The *reference counting* mechanism of MPI (see section 2.7 chapter 2) does not prevent the *MPI_Info* object from being deleted as the third call to *MPI_Info_set()* is executed *after* the call to *MPI_Info_free()* on the second thread in this particular ordering of thread schedules. The *reference counting* mechanism ensures that an MPI object is not deleted if the use for that object is not complete i.e. if the *reference count* is more than one. But in this case the *reference count* of *MPI_Info* object is one when *MPI_Info_free()* is called. This program/code-snippet is *quasi-erroneous/partially-correct* as if the three calls to *MPI_Info_set()* are executed before the call to *MPI_Info_free()*, the program would successfully execute. Specifically, the value $\phi_{21} = 4$ and the ratio $\alpha = \frac{1}{4}$ for the example shown in Figure 3.1.

Algorithm 3.1 Thread Safety for *MPI_Info*

Require: shared: MPI_Info *info*, lock_t *info_lock*
Ensure: one lock variable for each *MPI_Info* object

```
1: Function IsExists(MPI_Info info)
2: if info == MPI_INFO_NULL then
3:     return false
4: else
5:     return true
6: end if
7: end Function
```

Thread Safety for *MPI_Info*

```
8: ∀ instances of info
9: set_lock(info_lock)
10: if IsExists(info) then
11:     Manipulate(info)
12: end if
13: unset_lock(info_lock)
```

MPI states [1] that the *MPI_Info* object takes the value MPI_INFO_NULL when *MPI_Info_free()* function is called on it. This value is checked for in the *IsExists()* function which returns *true* if the object exists and *false* otherwise. The *lock_t* datatype is a hypothetical datatype for a theoretical machine. This could evaluate to an OpenMP lock datatype i.e. *omp_lock_t* or a *mutex* variable of type *pthread_mutex_t* while using Pthreads (POSIX threads). For each instance of the *info* object being manipulated, we need to wrap the instance with *set_lock(info), IsExists(info)* and *unset_lock(info)*. The granularity of the lock is at the level of the object as we need a separate lock for each *info* object and not for different occurrences of the same object. *Manipulate()* is any user defined function which operates on the *info* object.

3.2 Generalization

Algorithm 3.1 is a representative element of a general category of functions which require thread-safety. The three basic operations associated with any object are: read, write and update. There is a fuzzy difference between the update and write operation but the result of both the operations is that an existing value (or no valid value) is changed to a different value (or the same value is overwritten). A fourth operation which can be associated with some objects is a deletion or freeing operation. For example, an *MPI_Info* object can be freed after which it ceases to exist in the sense that it contains a value MPI_INFO_NULL when *MPI_Info_free()* is called on it.

Operation 1	Operation 2	Conflict
Read	Read	No
Read	Write/Update/Free	Yes
Write	Read/Write/Update/Free	Yes
Update	Read/Write/Update/Free	Yes
Free	Read/Write/Update/Free	Yes

Figure 3.2: Conflicting vs. non-conflicting operations

Figure 3.2 shows various combinations of operations which lead to a conflict (or no conflict) when two or more threads are performing MPI operations on the same attribute/value of an object/location. As a first step to perform any operation on an object, a lock must be obtained to ensure that only one thread at a time is operating on that object. There are two levels of existence: object level and attribute level. It is possible that another thread might have deleted/freed the object and hence it is necessary to query for the existence of the object on which we have obtained a lock. For example, in the case of *MPI_Info* the object contains a value MPI_INFO_NULL if the function *MPI_Info_free()* has been called on it. The *MPI_Comm_free()* operation deallocates the communicator object and sets its value to MPI_COMM_NULL and hence it is erroneous to call a function that reads, writes, or updates the value of an attribute associated with the communicator object.

The second level deals with the existence of an attribute of an object and prohibits any read, write or update operation called subsequently after the deletion/freeing of the attribute. Figure 3.3 gives the flowchart for carrying out manipulation of a shared object (or its attribute values) when conflicting calls depicted in Figure 3.2 are operational. Most objects in MPI provide functions which free level one of existence (object) but a function like *MPI_Comm_delete_attr()* deals with level two (attribute existence). A function *MPI_Type_delete_attr()* falls in the same category and may clash with an operation like *MPI_Type_get_attr()*.

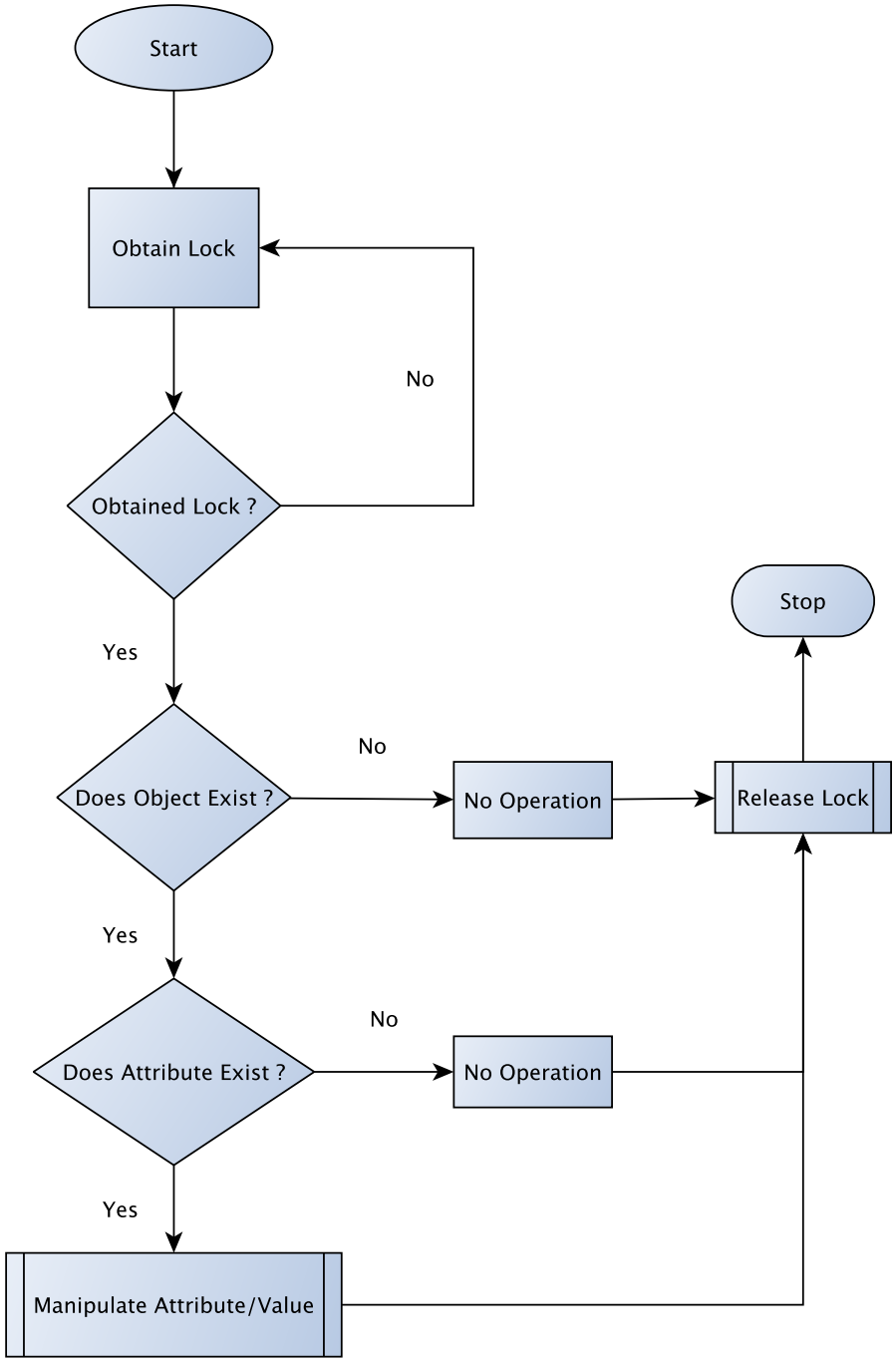

Figure 3.3: Flowchart depicting the manipulation of objects/attributes with conflicting Read/Write/Update/Free operations.

3.3 The function *malloc()*

The function used in C language for dynamic/run-time allocation of memory is *malloc()* [15], the syntax for which is shown below:

*void *malloc(size_t size)*

malloc() allocates space for an object whose size is specified by *size* and whose value is indeterminate. It returns the starting address of the contiguous block allocated on success and returns a NULL pointer on failure. The memory is allocated from the heap - which is a continuous piece of memory defined with three bounds: the starting point (START), an end point (BREAK) and the maximum limit (RLIMIT). As shown in figure 3.4 the current end point of the heap memory is called BREAK and the maximum limit is denoted by RLIMIT.

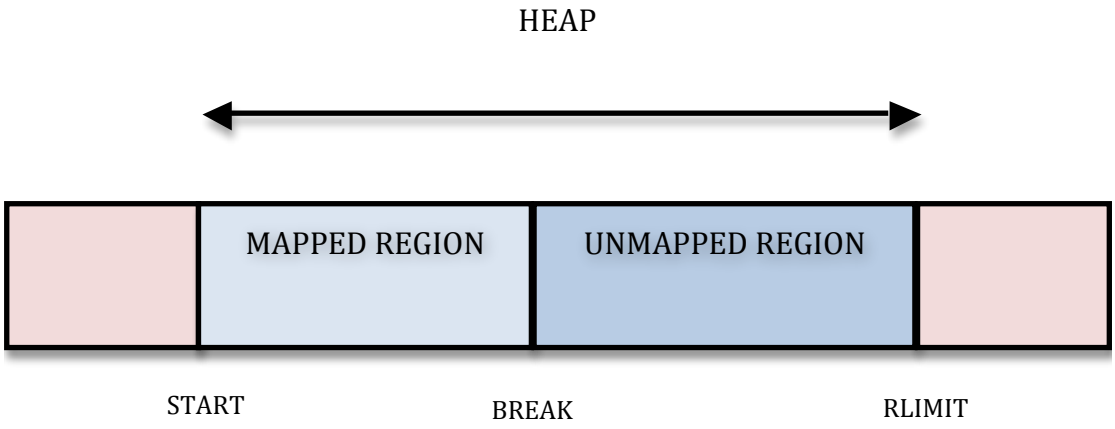

Figure 3.4: Heap memory

The space above RLIMIT and below the starting point START is not part of the heap memory. Two ANSI C functions for manipulating the BREAK are *brk()* and *sbrk()* for placing the BREAK at a given address and incrementing the BREAK by a certain amount, respectively. Depending on the implementation, the *sbrk()* function either returns the current BREAK address or the BREAK address after advancing the end point by the offset provided as its argument. In all implementations, *sbrk()* called with an argument of zero (0) returns the current BREAK address. The functions for manipulating RLIMIT are *getrlimit()* and *setrlimit()*. The memory allocated by *malloc()* on success is always more than the memory requested by the user as some bookkeeping meta-data regarding the memory must also be stored along with the data but the pointer returned to the user always points to the starting address of the data block having the appropriate requested data size. The meta-data consists of at least three fields: the address of the next chunk, size of the chunk, and whether the current chunk of memory is free or allo-

cated. This pattern of storing a pointer to the data block, having a pointer to the next chunk, a data block and other implementation dependent fields suggest that the memory inside heap is represented as a linked list of allocated and unallocated nodes.

3.3.1 Issues with thread-safety in *malloc()*

Several MPI routines like *MPI_Comm_split(), MPI_Comm_dup()* etc. call *malloc()* internally to allocate memory[4]. Further, the developer can issue calls to *malloc()* for dynamic memory/run-time allocation in an application. Depending on the implementations of *malloc()*, it may or may not be thread-safe and/or re-entrant. A re-entrant routine is a routine which cannot be safely restarted after interruption at a certain point due to a jump, call, hardware or software interrupt, without guaranteeing the safety of data. A re-entrant routine/function cannot use static data, return a pointer to static data or call a non-re-entrant function. Further if it uses global data then it must protect it in order to keep the values consistent from a modification by another invocation of the same function possibly from another routine. For example a routine may be interrupted due to delivery of a signal which starts a signal handler installed in the memory and which in turn might call the same function which was interrupted in the first place. Discussing re-entrant procedures in detail is beyond the scope of the current project.

Various flavors of *malloc()* have been made both thread-safe and efficient. For example *TCMalloc()* [16] which stands for *Thread Cache malloc()* uses a thread-cache for satisfying the memory requirements of small objects ($\leq$ 32K) and utilizes the central heap using a page-level (page $\approx$ 4K) memory allocator for large objects. Each small object maps to one of the (approximately) 170 classes. For example, allocations in the range 961 to 1024 bytes are rounded to 1024 bytes and the size classes are spaced so that smaller sizes are separated by 8 bytes, larger by 16 bytes, and so on, till the maximum separation of 256 bytes for classes $\geq$ 2k bytes. It is efficient due to virtually zero contention when allocating small objects and uses fine-grained efficient spin locks for large objects. *ptmalloc()* is another flavor of *malloc()* that improves upon the original *Doug Lea malloc()* [17] by using per thread arenas but with the disadvantage that the memory can never move from one arena to another and hence results in wasted space. *SmartHeap* for SMP systems offers another solution by allowing multiple threads to make concurrent requests to the heap.

The memory allocated by using *malloc()* can be freed by a function called *free()*. This function adds the allocated memory to the free pool by altering the meta-data and making it available for other requests. Several policies like best fit, first fit etc.

[4]Functions other than *malloc()* can be used to allocate memory dynamically. Further, the routines can be implemented without allocating dynamic memory (see Appendix D).

are used while allocating memory. Since a usual implementation of *malloc()* uses some static and global variables, it is considered both thread-unsafe and non-re-entrant in general. Hence, our aim is to make *malloc()* thread-safe by treating it as a black box i.e. not bothering about its implementation and creating conditions such that no two threads are allocated the same address/intersecting addresses.

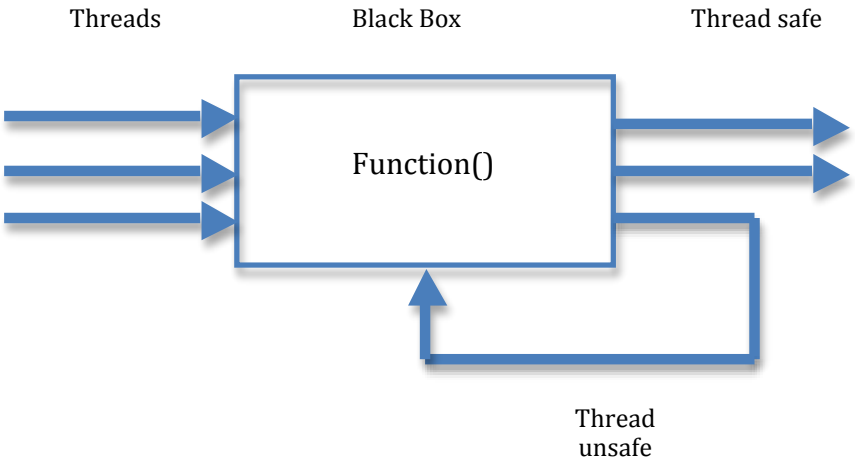

Figure 3.5: *malloc()* as black box

Figure 3.5 shows the methodology behind making *malloc()* thread-safe where we take post-*malloc()* actions to make the allocation thread-safe i.e. we let the *malloc()* function execute without any intervention/changes to its source code and recover from the situation when two or more threads are allocated the same address. If more than one thread is assigned the same address, re-allocation is done by calling *malloc()* again.

3.3.2 Making *malloc()* thread-safe

The easiest and (possibly) the most in-efficient solution to making *malloc()* thread-safe is to use a global lock to ensure that no two threads call the *malloc()* function at the same time. This results in a serialized thread schedule and all the calls are executed in some (any particular) order. The complexity of implementing this in terms of *Lines of Code* (LOC) is negligible. However the execution performance is possibly the worst. This solution should therefore be considered as a trivial but correct solution and is presented in Algorithm 3.2.

Algorithm 3.2 Thread-Safety for *malloc()* using a global lock

Require: shared: lock_t *global_lock*
Ensure: pointer variable local to a thread
1: ∀ instances of *malloc()*
2: set_lock(*global_lock*)
3: *var* ← *malloc(size)*
4: unset_lock(*global_lock*)

Algorithm 3.2 presents a solution based on a single global lock named *global_lock* which is shared among all the threads and is locked by a thread before it executes a call to *malloc()*. Irrespective of a successful or unsuccessful call to *malloc()*, the lock is unset and hence can be used by another thread. Considering that thousands of threads might be executing on an SMP, this strategy is not scalable as it allows only one thread at a time to obtain memory using *malloc()*. What is needed in practice is an algorithm which allows multiple threads to execute *malloc()* concurrently and also guarantee that no two threads are assigned the same address. We design an alternative strategy where any number of threads, say n can execute *malloc()* concurrently and in the event of a subset $\overline{n}$ threads being assigned the same starting address, $\overline{n} - 1$ threads back-off and only one thread out of $\overline{n}$ threads advances ahead with the allocated piece of memory. The performance of the algorithm depends heavily on $\overline{n}$ being $\ll n$ i.e. the number of threads acquiring the same address should be much less in number than the total number of threads concurrently executing *malloc()*.

3.3.3 A non-blocking and wait-free thread-safe algorithm for *malloc()*

The strategy that we present makes *malloc()* a thread-safe function by taking a post-*malloc()* action and involves a shared queue-like-structure implemented by a single dimensional unsigned integer array that supports the insert/enqueue operation. Each thread after being assigned an address by *malloc()* checks if the starting address (which has been returned to it) is present in the shared array or not. If the thread finds an identical entry in the shared array, it does not make an entry into the array as another thread which has been assigned the same address has already inserted its address into the table. Further, it calls *malloc()* again to request another non-conflicting piece of memory. This checking, removal of entries from the globally shared array and re-invocation of *malloc()* is done after *malloc()* is called and before any thread attempts to use the address space given to it. Before calling *free()* on the allocated memory for deallocation, a thread must reset the value of stored by it in the global shared array to zero, which is the value of a constant NULL defined in the ANSI C standard. Algorithm 3.3 presents this approach.

Algorithm 3.3 Non-blocking and wait-free thread-safe algorithm for *malloc()*

Require: shared: Q[MAX_THREADS*MAX_MALLOC], *grear*, private: *size*, boolean *allocate, flag*, ⟨data-type ⟩ *ptr*
Ensure: 1≤i≤ MAX_THREADS * MAX_MALLOC, *Q[i]* ← -1, *grear* ← 0

```
1: Function Search(p)
2: index←0
3: flag ← false
4: while index ≤ MAX_THREADS * MAX_MALLOC ∧ Q[index++] ≠ -1 do
5:    if Q[index] == p then
6:       flag← true
7:       break
8:    end if
9: end while
10: return flag
11: end Function
```

Non-blocking and wait-free thread-safe algorithm for *malloc()*

```
12: ∀ instances of malloc()
13: allocate← false
14: while allocate ≠ true do
15:    ptr ← malloc(size)
16:    if ptr≠ NULL then
17:       while true do
18:          flag ← Search(ptr)
19:          if flag == false then
20:             if CAS(Q[grear], -1, ptr) then
21:                FAA(grear,1)
22:                allocate←true
23:                break
24:             end if
25:          else
26:             break
27:          end if
28:       end while
29:    end if
30: end while
```

The constants MAX_THREADS and MAX_MALLOC (Algorithm 3.3) denote the maximum number of threads which are created per OS process and the maximum number of times any thread calls *malloc()*, respectively. The problem in storing data in a dynamic data structure is that memory for that data structure would need to be allocated dynamically and hence would again need a *malloc()* to be called which would need a thread-safe code again. Hence, we use an al-

ternative approach which uses a fixed size shared data structure i.e. a shared one-dimensional array having size MAX_THREADS * MAX_MALLOC, where each element is of the size of a *uint64_t*[5]. The only drawback of this approach is that when threads create threads which call *malloc()*, the fixed size of the array might not be sufficient. To accommodate for this situation to a certain extent, some extra space must be allocated to the shared array while declaring it.

Algorithm 3.3 which uses a *Search()* function to find if the address being allocated to a thread is present in the shared array or not. Initially all the threads call *malloc()* and obtain a starting address which is returned/contained in thread-private *ptr* variable. This starting address can be the same for several threads. An invalid address NULL can be returned by *malloc()* when there is insufficient space in the heap even after advancing BREAK further than the current limit. If the returned address is not NULL, the threads after obtaining the address execute the *Search()* function to search for the address allocated to them in the shared array *Q*. If the address is present, then it clearly means that there is another thread which has been assigned the same address and which has already made its entry into the shared array *Q*. In such a case the variable *flag* is returned as *true* and the variable *allocate* retains its value as *false*, causing *malloc()* to be called again for reallocation of memory. Figure 3.6 shows the traversing of the shared array *Q* by threads.

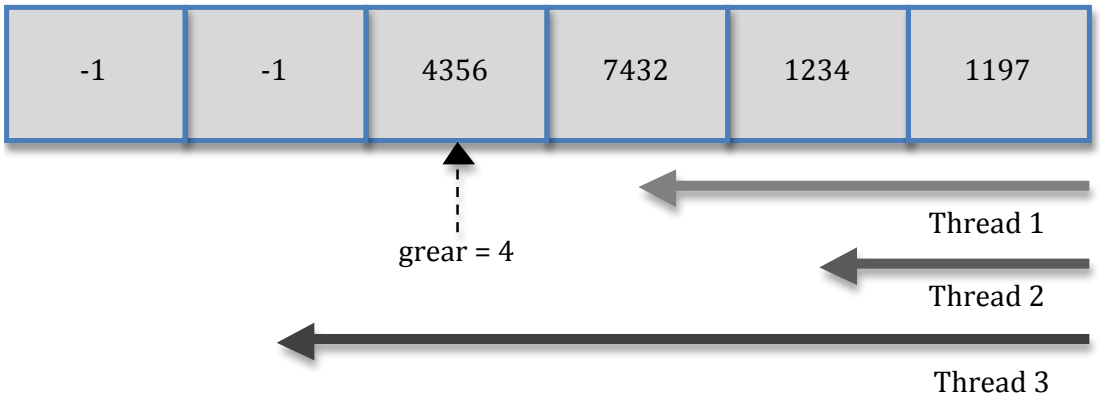

Figure 3.6: Traversing of shared array *Q* by threads

The value of the variable *grear* is initialized to *zero* and acts like the rear pointer of a queue that points to the location where values can be inserted/enqueued. This global rear *grear* is incremented after the atomic CAS (Compare and Swap) operation is executed by any thread. The atomic CAS operation takes three arguments: the register or the memory location, an expected value to be compared with the value in the register/memory location and the value which should replace

[5] *uint64_t* is a data-type that represents an unsigned integer in 64 bits.

the current value if it contains the expected value. It returns *true* if the replacement operation is successful else returns *false* in case of failure. The function CAS can be written as a code snippet in C language as shown in Listing 3.1. The entire function is assumed to execute instantaneously.

Listing 3.1: Code snippet implementing CAS

```
bool CAS(register r, int old, int new)
{
if (r==old)
 {
  r=new ;
  return true ;
 }
else
  return false;
}
```

The property of an atomic operation is that it *appears* to happen in an instant and if multiple threads execute the CAS operation simultaneously then only one of the threads will execute it successfully. The rest of the threads will fail. This guarantees that the global rear variable *grear* is only incremented by the thread which succeeds in completing the CAS operation. There are several algorithms lock-free algorithms which make use of atomic operations for a queue data structure in literature but almost all of them use a linked list, which in turn requires *malloc()* for dynamic memory allocation in the C language. Clearly, we cannot use *malloc()* as it is the function which is being made thread-safe here on the assumption that it is thread-unsafe. Using a concurrent lock-free data structure prevents non-conflicting operations from serialization and hence enhances the performance [20]. Further, the FAA (Fetch and Add) atomic instruction fetches and adds one to the current value of *grear*. The term register for the first operand of the CAS operation has been used for historical reasons and could be anything from a register to a memory location [21].

Theoretical argument leads us to the possibility of two threads obtaining the same address again and again and hence emerges a need for a method to break up this infinite chain. An *Exponential Back Off* strategy can be used after a thread executes the *Search()* function and finds that the address allocated to it is already present in the shared array so that the threads have to wait a random amount of time and the probability of a collision reduces i.e. the probability of them invoking *malloc()* at the same time and acquiring the same piece of memory reduces. This algorithm is typically used to reduce the probability of collision in CSMA/CD (Carrier Sense Multiple Access/Collision Detection)[6] [19] networks. In the context of threads the number of collisions is taken as the number of times any thread is assigned a conflicting address as another thread. The advantage is that the

[6]Multiple systems connected to a shared medium use decentralized control to transmit data and delay their transmission randomly if a collision occurs by choosing a multiple of the Round Trip Time (RTT) of the network.

scheme is lock-free and randomly delays the threads executing it so that *malloc()* is possibly not invoked simultaneously. Each thread chooses a random number from the interval $[0, 2^{collisions} - 1]$, which is multiplied by a system dependent time to obtain a random delay. This is shown in Algorithm 3.4.

Algorithm 3.4 Exponential Back Off policy addition to thread-safe *malloc()*

```
1: Function Expo_Delay(integer collisions)
2: x ← random([0, 2^collisions-1])
3: delay← x * T                ▷ T is system dependent and in units of time
4: end Function
```

Due to an entry in the shared array Q of the addresses assigned to threads - we need to carry out some form of pre-processing before the *free()* function is called to release the memory into the free pool. The only need is to reset the entry in the shared array to a *zero*. A value of *zero* has been carefully chosen so that no thread which is reading the value at any instant of time sees a value of -1 due to the internal boolean representation of 0 and -1. A -1 in *2's complement* is represented by sixty four 1's (assuming a 64 bit architecture) and a 0 is represented by sixty four 0's. Hence at no point in time a value of 0 can be interpreted as -1. This idea is expressed in Algorithm 3.5.

Algorithm 3.5 Algorithm for *free()* compatible with Algorithm 3.3

```
Require: private: ⟨ data-type ⟩ ptr
Ensure: ptr ≠ NULL
1: Function Nullify(ptr)
2: index ← 0
3: while Q[index] ≠ p do
4:     index ← index + 1
5: end while
6: Q[index] ← 0
7: end Function
```

Algorithm for *free()* compatible with Algorithm 3.3

```
8: ∀ instances of free()
9: Nullify(ptr)
10: free(ptr)
```

3.4 Thread-Safety with MPI Send and Recv

MPI_Send() is a point-to-point MPI operation used to send messages to a destination MPI process. The MPI standard [1] defines several types of send calls

namely: blocking, non-blocking, buffered, ready and standard. Abstractly, the blocking send call blocks until the data is transferred to the destination or to an intermediate system buffer. A non-blocking send call returns immediately after being posted and its completion involves using some form of a wait operation. A buffered call attaches the send buffer to a user provided buffer for (possibly) later delivery. A ready send can only be posted if the destination has already initiated a receive call. A standard send may be either synchronous (blocking) or buffered. The C language syntax for a standard send is shown below.

*int MPI_Send(const void *buffer, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm)*

The receive call is used to receive messages from a particular or from any source. The latter in a receive call is specified by using a wild-card denoted by a named constant MPI_ANY_SOURCE. There are two flavors of receive namely, blocking and non-blocking. A blocking receive operation blocks until the process/thread receives the complete data and frees the resources upon return. A non-blocking receive returns immediately after being posted irrespective of whether data has been received or not. It is the user's responsibility to check if the non-blocking receive has completed by calling some form of wait. The syntax for a blocking receive is shown below.

*int MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm)*

The syntax for both the calls look identical except for the fact that the *buffer* parameter in the *MPI_Recv()* call is modified during the process of receiving (except when receiving message of size zero) and the *source* parameter is used instead of the *destination* in *MPI_Send()*. There are two parts to the message: the data and the message envelope. The particulars of data are specified by the first three arguments in the *MPI_Send()* or *MPI_Recv()* functions and the envelope consists of the *source, destination, tag* and the *intra* or *inter-communicator* on/through which the communication takes place [1]. *count* gives the number of data elements to be received or sent, *datatype* is a user defined datatype or a pre-defined datatype like MPI_INTEGER, MPI_CHARACTER etc. and the *tag* can take any valid integer value and can be used to identify a particular thread in a process model.

A special value of *tag* in the receive operation can be MPI_ANY_TAG which specifies that a message with any *tag* can be received by the receiving process. A similar value denoted by MPI_ANY_SOURCE allows to accept messages from any arbitrary source. In *MPICH2* [4] the blocking synchronous send operation is specifically named as *MPI_Ssend()* and the *MPI_Send()* function can be blocking (synchronous) or buffered. In this report we specifically mention the word blocking or non-blocking before the operation to avoid any ambiguity.

3.4.1 Thread-Safety issues with shared buffers in Send and Recv

In the presence of multiple threads the shared buffer being used in the send or receive function call can be modified by a thread which is executing concurrently on another core of the SMP. This problem is almost but not exactly the same as the thread-unsafe scenario of *MPI_Info()*. The difference here is that *MPI_Send()* and *MPI_Recv()* depend on each other to complete and hence are not necessarily local to the SMP threads. These can be taken as a *lock-key pair* or a complimentary set of calls which are incorrect if executed in isolation (except for a send/receive to/from MPI_PROC_NULL). For every *MPI_Send()*, a matching *MPI_Recv()* must be posted by another thread. Further, it is not necessary that the *MPI_Recv()* is executed as soon as the *MPI_Send()* call is executed and vice-versa. MPI specification delegates the responsibility to the user for synchronizing the use of resources involved in a send or receive operation in case a thread is de-scheduled due to a blocking send/receive and another is scheduled in the same address space (section 3.4 of [1]).

To demonstrate and discuss thread-safety issues in point-to-point operations, we assume two threads T_A, T_B are on the same SMP, and T_C is on a different SMP, operations Receive (R), Send (S) and Write(W) and a shared buffer BUF. We do not discuss a read operation on a shared buffer separately as it is meted out the same treatment as a Write(W) operation. An arbitrary thread is denoted by X or Y where X $\neq$ Y and a buffer is denoted by an empty paranthesis ( ), if the argument does not matter. An execution of a Receive(R) by T_A on shared buffer BUF is denoted by T_A^R(BUF). The following conditions should be allowed and observed:

1. $T_X^W(\text{BUF}) \Rightarrow \neg\ (T_Y^R(\text{BUF}) \vee T_Y^S(\text{BUF}))$
2. $T_X^S(\text{BUF}) \vee T_X^R(\text{BUF}) \Rightarrow \neg\ T_Y^W(\text{BUF})$.
3. $T_A^R(\text{BUF}) \Longleftrightarrow T_C^S() \vee T_B^S(\text{BUF}) \Longleftrightarrow T_C^R()$.
4. $(T_A^R(\text{BUF}) \Longleftrightarrow T_B^S(\text{BUF})) \wedge \neg\ T_X^W(\text{BUF})$.

The first condition implies ($\Rightarrow$) that when any receive or ($\vee$) send operation has been initiated by some thread, no other thread in the same SMP can initiate a write operation or vice-versa. The second condition is obtained from the first condition by exchanging the LHS (Left Hand Side) of the equation with the RHS (Right Hand Side) and transposing the negation operator ($\neg$).

The third condition states that only one of the operations i.e. either receive or send can be executing on the same buffer when communicating ($\Longleftrightarrow$) with a remote thread (and not both the operations).

It is possible that two threads on the same SMP may communicate through MPI point-to-point calls using the same buffer i.e. one thread sends a shared buffer to

another thread which receives the message in the same shared buffer. Such a call is theoretically and practically possible although it may not serve any purpose as the information is already available in the shared buffer. For the sake of correctness and completeness such scenarios are allowed and the fourth condition above encapsulates this case. We use the terminology local calls for such scenarios. As opposed to this a thread can send or receive to/from another thread in a different SMP (remote call). In the latter case only one of receive or send operation using the same shared buffer should be allowed to progress.

Threads must be able to determine if the communication between the corresponding thread is a local or a non-local one. Hence, a shared buffer is used for each OS process in which a thread after initializing, records its presence in that process by entering its rank in the global array (shared among threads/MPI processes of an OS process). Using this scheme a thread when communicating with a destination can check the shared array to determine if the communication is a local or a non-local one. Algorithm 3.6 shows how the conditions described above can be handled in a thread-safe way.

Algorithm 3.6 Thread-Safety for shared buffer in point-to-point communication

Require: shared: *Rank*[MAX_THREADS], int *sr_test*, ⟨datatype⟩*buffer*, lock_t *w_lock, r_lock*

Ensure: ∀ threads ∈ SMP initialize *Rank*[*thread_ID*] ← *thread_rank*, *sr_test* ← -1

```
1: Function Query(rank)
2: index ← 1
3: while index ≤ MAX_THREADS ∧ Rank[index] ≠ rank do
4:    index ← index + 1
5: end while
6: if index > MAX_THREADS then
7:    return false
8: else
9:    return true
10: end if
11: end Function
```

Thread-Safety for shared buffer in point-to-point communication

12: ∀ instances of *MPI_Send()*
13: *Is_local* ← Query(*destination*)
14: **if** *Is_local* == *false* **then** ▷Remote Communication
15: set_lock(*w_lock*)
16: MPI_Send(*buffer*,...,*destination*)
17: unset_lock(*w_lock*)
18: **else** ▷Local Communication
19: set_lock(*w_lock*)
20: **while** ¬ CAS(*sr_test*, -1, *source*) **do**
21: **end while**
22: MPI_Send(*buffer*, ..., *destination*)
23: unset_lock(*w_lock*)
24: **end if**

Thread-Safety for shared buffer in point-to-point communication

25: ∀ instances of *MPI_Recv()*
26: *Is_local* ← Query(*source*)
27: **if** *Is_local* ≠ *true* **then** ▷Remote Communication
28: set_lock(*w_lock*)
29: MPI_Recv(*buffer*,...,*source*)
30: unset_lock(*w_lock*)
31: **else** ▷Local Communication
32: **while** ¬ CAS(*sr_test*, *source*,-1) **do**
33: nop ▷No operation
34: **end while**
35: set_lock(*r_lock*)
36: MPI_Recv(*buffer*, ..., *source*)
37: unset_lock(*r_lock*)
38: **end if**

Thread-Safety for shared buffer in point-to-point communication

39: ∀ instances of *Write(buffer)* ▷ Also applies to any *Read(buffer)* operation
40: set_lock(*w_lock*)
41: set_lock(*r_lock*)
42: Write(*buffer*)
43: unset_lock(*r_lock*)
44: unset_lock(*w_lock*)

The shared array *Rank*[MAX_THREADS] for a particular OS process contains the ranks of all the threads that share the address space of the process. The

threads while getting initialized, can store their rank at an index which is equal to the thread ID of the thread and is specific to the threading library being used. For example, the OpenMP threading library gives thread ID's starting from zero to the maximum number of threads minus one. It is recommended that some extra space in the shared array be reserved as threads may create other threads which store their ranks in the shared array *Rank* while getting initialized. The function *Query()* takes as input the rank of a thread and returns *true* if that rank is found in the shared array *Rank*, else returns a value of *false*.

For all instances of *MPI_Send()/MPI_Recv()*, we first determine whether the communication will be local (intra-OS process ranked-thread communication) or non-local (inter-OS process ranked-thread communication) by using the function *Query(rank)*. For *MPI_Send()*, if the communication is non-local, a lock *w_lock* is obtained which can only be held by a thread doing remote/local communication or a thread doing a Write(W) on the shared buffer. The same sequence of operations apply to a thread performing a non-local receive operation and guarantees mutual exclusion from a thread intending to do a remote/local send or a Write(W) on the shared buffer. Thus, the remote communication pseudo-code of *MPI_Send()* and *MPI_Recv()* looks almost identical. Clearly, a non-local send or a non-local receive or a local write on the same shared buffer is guaranteed not to interfere with each other due to mutual exclusion implemented by acquiring a lock *w_lock*. Figure 3.7 illustrates a local versus non-local communication.

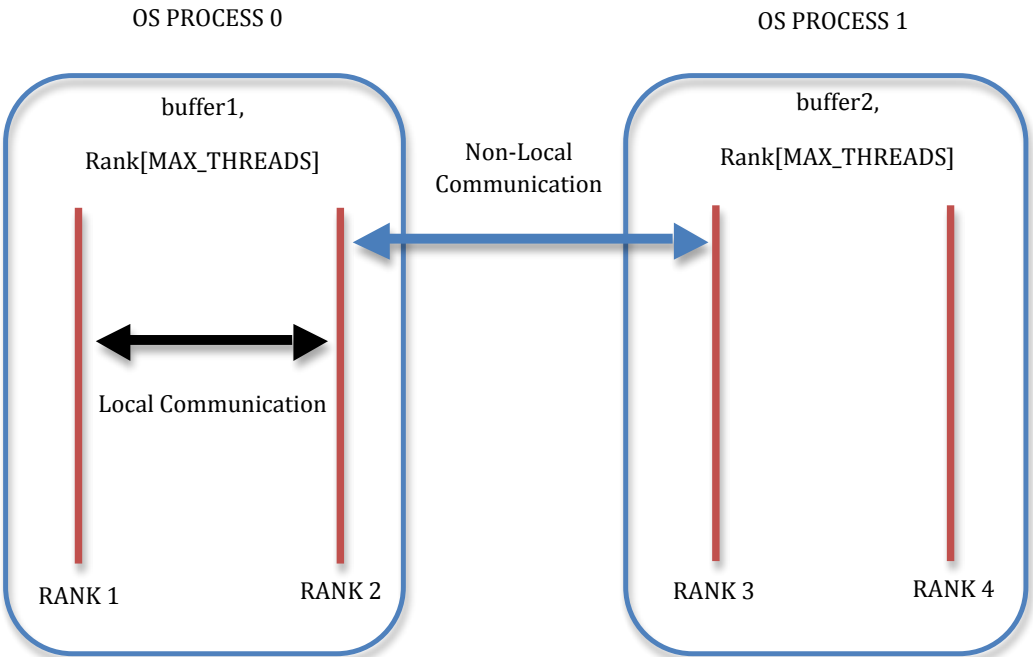

Figure 3.7: Local vs. non-Local communication

The algorithm must also guarantee that if there is a local communication between two threads then both the local receive and local send must execute but prohibiting

a write to occur or vice-versa. This case is different from a non-local send or receive as in the non-local case only one of send or receive is executed. This problem is taken care of by *MPI_Send()* when it sets the variable *sr_test* after acquiring the *w_lock* which in turn is done *after* determining whether the communication is local or not. *MPI_Recv()* on the other hand tests the value of *sr_test* to see if it has been set to its rank by the sending MPI process/thread. If it finds its own rank in the shared variable *sr_test*, it proceeds towards acquiring the *r_lock*. Acquiring *r_lock* guarantees that a write on this buffer will not interfere with this either receive or send as a thread that wants to write to this shared buffer must acquire two locks i.e. *w_lock* followed by *r_lock* precisely in that order. Further the *r_lock* is released when the receive function returns and hence a write on the shared buffer can only be initiated after receive returns. This describes and proves the correctness of the algorithm when a send is posted *before* the corresponding receive operation.

In the case when a receive is posted before the corresponding local send operation, the receive does not acquire the lock *r_lock* as the value of the variable *sr_test* is initialized to -1. During this time period it is possible for a write operation by another thread to acquire the *r_lock* and proceed with the modification. A local send which wants to start while this writing operation is going on cannot proceed until the *w_lock* is released by the thread performing the write. After the thread is done writing it releases both the locks and at some point in time the sending thread can acquire the *w_lock* and set the variable *sr_test* to the rank of the receiving MPI process/thread.

There is also a possibility that two or more threads are spinning on the variable *sr_test* to check if it has become equal to the their rank and waiting for a local communication. But even if multiple threads post the local send operation to send messages to their counter-parts, only one of is able to acquire the *w_lock* and sets the value of *sr_test* to the appropriate rank. The thread corresponding to this rank (rank = value of *sr_test*) would then proceed towards acquiring *r_lock* and posting a matching receive. There is possibility that due to a 64 bit[7] read operation, a thread with the incorrect rank may misinterpret the value of *sr_test* to be equal to its own rank and hence acquire *r_lock*, resulting in a dead-lock. To prevent this situation, a CAS operation does the comparison and the thread performing it proceeds if a value of *true* is returned. The other CAS operation in the *MPI_Send()* pseudo-code ensures an atomic write on the *sr_test* variable so that the CAS operation in the receive operation pseudo-code reads an atomically set value of *sr_test*. Since in a thread-as-rank model the ranks of threads are unique, only one CAS operation executed by the receiving thread will succeed and the others will fail.

The utility of a send and receive operation between two threads of the same

[7]A 64 bit read possibly is done in two 32 bit reads and hence a value being written into a variable can be misinterpreted, if it is being read simultaneously. For example, different Intel processors guarantee atomicity of different operations [27].

SMP comes into the picture when one thread wants to send a particular state of the buffer to another thread without intervention from any third thread which might possibly perform a write on the shared buffer. In such cases if a pure shared memory model is used then the thread intending to receive the buffer must acquire a lock on the buffer which in itself is a complex procedure as there are a set of contending threads. Instead, the sending thread acquires a lock on the buffer and the send lock and waits for the correct thread to acquire a read lock on the buffer after the receiver examines the value of the variable *sr_test*. To further clarify the scenario, the sending thread can transfer the contents of the buffer to another private buffer preserving the exact state and then send it on to the receiving thread. In such a case only the receiving thread needs to acquire a write (*w_lock*) and read lock (*r_lock*) on the shared buffer.

3.4.2 Thread-Safety with MPI_ANY_SOURCE

The wildcard MPI_ANY_SOURCE when specified in a receive operations allows an MPI process/thread to accept messages from any source. The send operation on the other hand must specify a particular destination and hence a receiver posting a wildcard receive can accept *any* message from the set of all the messages which have destination as the rank of the receiver MPI process. This can potentially cause a mismatch in the sequence of messages that are to be received and hence in effect, may lead to a deadlock. This is shown in Figure 3.8 in which two threads send a message to a destination thread and the application logic fails to match the messages to the correct receive.

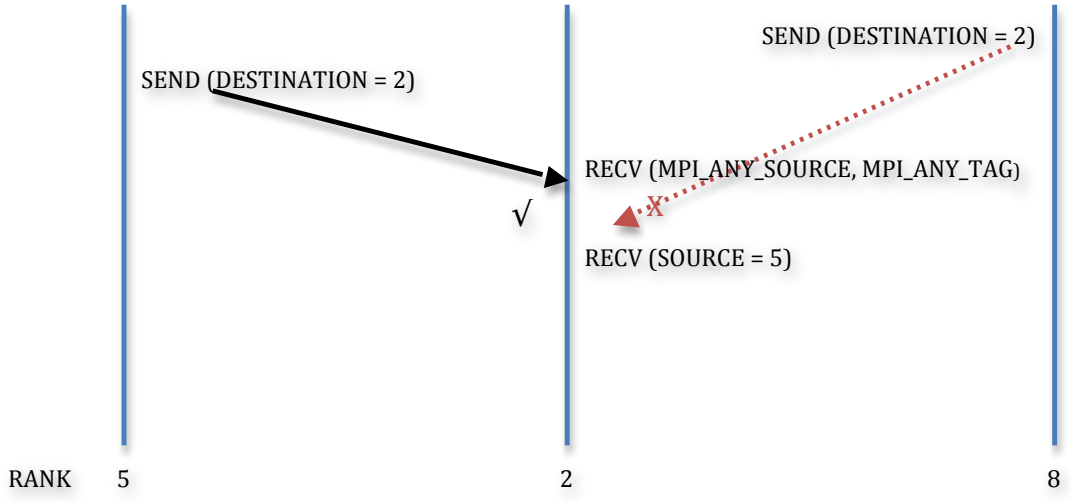

Figure 3.8: Wrong order of receives using wildcard MPI_ANY_SOURCE

Figure 3.8 shows a thread-unsafe scenario where a message from thread ranked 5 meant for destination thread having rank 2 is received by the receiver thread

using a wildcard receive which actually is meant for some other thread (according to the application logic). This message is received without any issues but when another thread ranked 8 sends a message to destination thread 2, it does not find a matching receive and hence the application deadlocks. A clear distinction here is made that this thread-unsafety because of MPI_ANY_SOURCE comes into picture because of the way it is used in the application and not because of any thread-unsafe code within the implementation or specification [29].

The thread-unsafe scenario can be avoided by making use of the *MPI_Probe()* function to find the source of the message and taking an appropriate option based on the value of the source. The *MPI_Probe()* function has the following syntax (section 3.8 of [1]):

*int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)*

The function has arguments which are exactly the same as the arguments in a receive call which constitute the message envelope. *MPI_Probe()* can return the particulars of a message in the *status* argument i.e. *tag, source* and *error* (MPI_TAG, MPI_SOURCE and MPI_ERROR). The value of MPI_SOURCE allows us to inspect (with reference to Figure 3.8) whether the message sent from thread ranked 8 is available or not. Listing 3.2 shows a solution for the scenario. We place the probe operation in a loop (which does not look elegant) to avoid this scenario. Instead we could have reversed the order of operations to receive the message from source ranked 5 followed by a message from MPI_ANY_SOURCE. The latter solution of simply reversing the two calls would lead to an erroneous solution when the receives are *non-blocking* as there would always be a possibility of accepting a message from thread rank 5 on the receive with MPI_SOURCE_ANY which would lead to a deadlock. Hence, Listing 3.2 makes sure that the message from thread rank 5 is received first, followed by a wildcard receive of any message (though this approach leads to a lower performance at the cost of correctness).

Listing 3.2: Using Probe to prevent deadlock

```
if ( rank == 2)
{
 flag = false ;
 comm = MPI_COMM_WORLD ;
 while( flag == false)
 {
 MPI_Probe(5,MPI_ANY_TAG, &flag,  MPI_COMM_WORLD) ;
 }
 MPI_Recv(buf, cnt, MPI_INT, 5, MPI_ANY_TAG, comm , &status);
 MPI_Recv(buf, cnt, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status)
 }
}
```

3.4.3 Thread-Safety with buffered Send

A buffered send requires a user defined shared buffer to be attached for calls to *MPI_Bsend()* using a *MPI_Buffer_attach()* function call. This buffer is used by MPI in the user space to store the outgoing buffered mode messages. Another function call *MPI_Buffer_detach()* detaches the buffer after the pending messages are sent (reference counting mechanism ensures this). The syntax for both the functions is shown below [1].

int MPI_Buffer_attach(void buffer, int size)*
int MPI_Buffer_detach(void buffer_addr, int* size)*

A limitation of these functions is that multiple threads cannot call any of the functions simultaneously. This is different from the shared buffer problem as the problem is independent of a particular buffer. Even when two threads call *MPI_Buffer_attach()* concurrently with different buffers, an error is thrown. It must be ensured that these calls are serialized. The simplest method is to call the functions outside the parallel regions i.e. function calls should be made only by the main thread. If they are to be called by multiple threads then a single global lock shared by the OS process can serialize the calls.

3.5 Thread-Safety with collective operations

Collective operations occurring on a communicator involve all the processes which are a part of it. A communicator defines a group or a group of processes [1]. Collective operations may have a root process which is the single sending or receiving process in that operation. The amount of data sent to a process must be received in some exact sized buffer specified by the receiver unlike the point-to-point operations and hence imposes a stricter check in collective operations. A blocking collective operation finishes execution as soon as the call returns but a non-blocking collective must to be tested for completion by using some form of wait operation as in point-to-point non-blocking operations. MPI further guarantees that messages generated on the same communicator by collective and point-to-point operations will not be intermixed. The collective operations may or may not have the effect of synchronizing the various processes and it is up to an implementation to choose the methods used for collective operations. The following discussion concerns only intra-communicators and not inter-communicators to limit the scope of the project. The four major categories of collective communications are:

1. **One-to-All**: One process dissipates data to all other processes and acts as the root in the collective communication. Examples include *MPI_Bcast()*, *MPI_Scatter()* and their respective non-blocking versions.
2. **All-to-one**: The root process receives information from all other processes.

Examples include *MPI_Gather(), MPI_Reduce()* and their respective non-blocking versions.

3. **All-to-All**: Each process in the communicator sends and receives information to/from all other processes. Examples are operations like *MPI_Alltoall(), MPI_Allgather(), MPI_Allreduce()* and *MPI_Barrier()* etc. *MPI_Barrier()* is always a synchronizing collective operation in which all processes must execute the barrier at some point in time but can exit asynchronously and no user-level data is exchanged by processes.

4. **Others**: These do not pertain to any of the above categories and include operations like *MPI_Scan()* and variations of it.

The non-blocking collective operations help to overlap computation with background communication and are not synchronizing in general. An non-blocking operation which is synchronizing is *MPI_Ibarrier()*. The immediate return from the call and testing their completion in a separate call gives the same advantage as non-blocking point-to-point operations. Further, two non-blocking collectives can be outstanding at the same time on the same communicator whereas two blocking collectives on the same communicator cannot be outstanding at the same time i.e. a blocking collective operation must be completed before the processes in the communicator can proceed further. In case of intersecting communicators, a blocking collective would lead to a dead-lock if there exists a cycle created by a circular wait condition and hence is another area where non-blocking collectives score over their blocking counter-parts.

3.5.1 Collective semantics in a rank-less thread model

Two threads belonging to the same process cannot issue two collective calls (same or different) on the same communicator concurrently. MPI prohibits the aforementioned scenario due to the possibility of mixing messages in any two or more collective operation over the same communicator. This is especially true as the messages may be fragmented, with a part of some message being matched incorrectly with a collective operation at a process. Listing 3.3 shows an erroneous code snippet.

Listing 3.3: Erroneous code

```
if ( threadID == 0)
 MPI_Bcast(buffer, count, MPI_INT, 0, MPI_COMM_WORLD) ;
if ( threadID == 1)
 MPI_Bcast(buffer, count, MPI_INT, 0, MPI_COMM_WORLD) ;
```

Listing 3.3 shows two threads in the same process posting an identical broadcast call over the communicator MPI_COMM_WORLD. Since a blocking call blocks only the calling thread, the other threads are not affected by these calls and

all processes in the communicator have two threads waiting to receive messages from process 0, which is the root in this operation. This communication fails even though any of the threads i.e. threads having threadID 0 or 1 in the other processes in MPI_COMM_WORLD can receive a message from threadID 0 or 1 of process 0. But there are reasons why this communication fails and hence should not be allowed.

Considering an uneven fragmentation of messages in the event of a large message, a thread may receive a message whose size does not equate to the buffer size specified in the collective operation. The application fails due to a mismatch of message sizes as MPI imposes strict matching of message sizes in case of collective operations. MPI *may* try to succeed by creating a duplicate communicator for one of the collective *MPI_Bcast()* calls analogous to separating the library communication communication with user communication but doing so would again require a collective operation on the same communicator which cannot be permitted as it would lead to the same problem. In practice there are several algorithms which are used in collective communication like message transfer based on *binary* trees or *binomial* trees. If such algorithms are implemented then the intermediate processes used to relay messages have no way of distinguishing between messages coming from two different broadcasts (no tags being used in collective operations). Hence, the MPI library disallows posting of any two or more collective operations by different threads of the same OS process/MPI process (rank-less thread model).

3.5.2 Blocking collectives in a thread-as-rank model

The semantics of blocking collectives differ in a thread-as-rank model from that of a rankless-thread model (described above) as the various threads/MPI processes of an OS process are individually addressable by ranks. All the threads which are part of a communicator must call the blocking collective operations in the same order on that communicator. In case of intersecting communicators i.e. one or more threads is common to two or more communicators, all blocking collectives on all the communicators must be called in the same order on all the threads. To illustrate possible thread-unsafe scenarios we assume two OS processes having three MPI processes/threads each (shown in Figure 3.9).

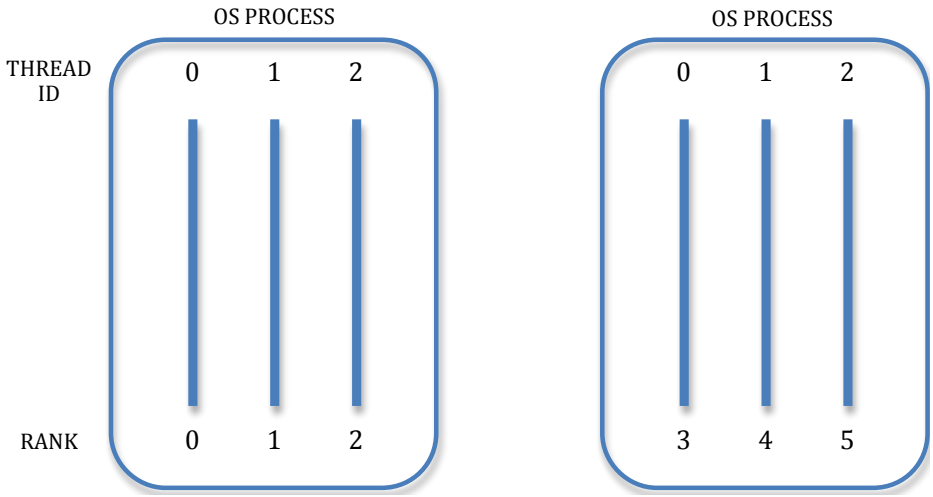

Figure 3.9: Three threads/MPI Processes in two OS processes

To show the disparity between the rankless-thread model (process model) and the thread-as-rank model, assume that a communicator *Comm0* contains all the the MPI processes. In the process model each MPI process is assumed to correspond to a single OS process and hence the communicator *Comm0* has two MPI processes ranked 0 and 1 whereas in a thread-as-rank model the communicator *Comm0* contains six MPI processes/threads ranked from 0 to 5, which in turn are contained in two OS processes. The following code snippet is a valid snippet under the process model but not under the thread-as-rank model.

Listing 3.4: Deadlocked code

```
if(ThreadID == 0)
 MPI_Alltoall(sbuf,cnt1,MPI_INTEGER,rbuf,cnt1,MPI_INTEGER,Comm0);
```

The reason is that in the process model this code snippet results in each thread of an OS process calling an *MPI_Alltoall()* collective operation which in effect is equivalent to both the processes participating in the same collective call over the same communicator. In the thread-as-rank model, all ranked threads must call the same collective operation in the same order on the same communicator but a threadID of 0 corresponds to threads ranked 0 and 3 only and hence threads 1,2,4 and 5 do not participate in the collective operation. Since not all MPI processes in *Comm0* call the execute the collective operation, this code deadlocks in a thread-as-rank model.

It is not necessary that a scenario that is successful in a process model always fails in a thread-as-rank model or vice-versa. As an example the following outlines a scenario which fails in both the models. Assume that a communicator *Comm1* contains both the processes for the process based model and contains threads with ranks 0, 1, 3 and 4 for the thread-as-rank model. Posting the following operations would lead to a dead-lock as MPI disallows any two threads of the same OS

process posting *any* two or more collective operations on the same communicator concurrently. This is shown in Listing 3.5.

Listing 3.5: Erroneous/Deadlocked code

```
if(ThreadID == 0)
 MPI_Alltoall(sbuf,cnt1,MPI_INTEGER,rbuf,cnt1,MPI_INTEGER,Comm1);
if(ThreadID == 1)
 MPI_Allreduce(sbuf2, rbuf2, cnt2, MPI_INTEGER, MPI_AND, Comm1);
```

Clearly, in a process based model, two threads in the same OS/MPI process calling different collectives are not allowed by the MPI standard (see section 3.5.1). Further, in a thread-as-rank model, not posting/executing the same collective on *all* the threads/MPI processes of the same communicator causes a deadlock. In general, a conclusion can be drawn that blocking collectives on *all* communicators must be called in the same order on every MPI process in the communicator in the thread-as-rank model. We conclude that there is no possibility of creation of an algorithm which can solve the fundamental problems described above until and unless the semantics of the blocking operations are changed.

It is also possible for a set of MPI processes to deadlock when a cycle is formed by calling collectives on different communicators that have at least one MPI process common to them i.e. the communicators are intersecting. As an example, consider three communicators consisting of two threads each defined as *C0* ={0,1}, *C1* = {1,2}, and *C2* = {2,0}. The following code snippet will deadlock.

Listing 3.6: Deadlocked code

```
if(rank == 0)
 MPI_Alltoall(sbuf,cnt1,MPI_INTEGER,rbuf,cnt1,MPI_INTEGER,C0);
if(rank== 1)
 MPI_Allreduce(sbuf2, rbuf2, cnt2, MPI_INTEGER, MPI_AND, C1);
if(rank== 2)
 MPI_Bcast(sbuf3, cnt3, MPI_INTEGER, 2, C2);
```

The code in Listing 3.6 produces a cycle in which thread 1 is waiting for thread 2 to finish which in turn is waiting for thread 3 to finish which in turn is waiting for thread 1 to finish. This is illustrated in Figure 3.10. Even if an algorithm detects blocking calls on intersecting communicators, no corrective action can be taken as the needed (correct) collective calls cannot be automatically inserted by the algorithm. Thus, the only way that application level thread safety can be maintained here is that the programmer must follow correct rules for placing blocking calls on same or distinct communicators. Such programs if created can be said to be completely erroneous as they surely deadlock or produce a certain error for any possible permutation of thread-schedules (see section 2.6).

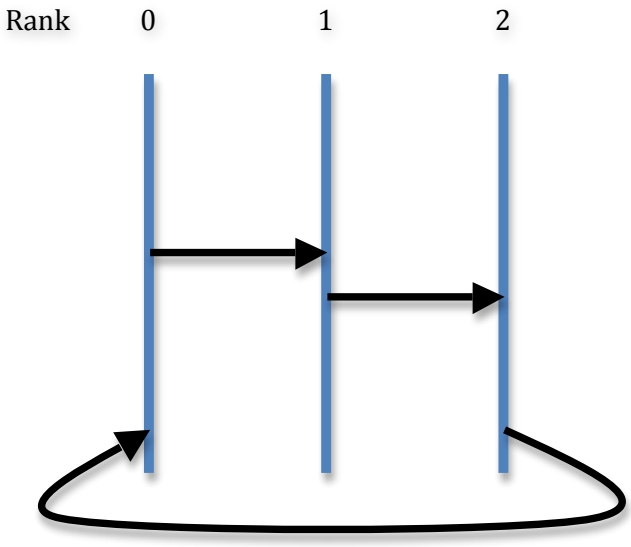

Figure 3.10: A cycle created by collective calls on intersecting communicators

3.5.3 Non-blocking collective semantics in a thread-as-rank model

The behavior of non-blocking collectives is very similar to their behavior on thread-as-rankless model/process model in the sense that the order of non-blocking collectives on a particular communicator must be preserved. For example the following code snippet indicates a valid piece of code if applied to Figure 3.9 where the communicator *C0* and *C1* are defined as consisting of threads ranked 0 to 5 i.e. $C0 = C1 = \{0,1,2,3,4,5\}$.

Listing 3.7: Valid code

```
if(rank == 0 || rank == 1 || rank == 2)
{
 MPI_Ialltoall(sbuf,cnt1,MPI_INTEGER,rbuf,cnt1,MPI_INTEGER,C0);
 MPI_Iallreduce(sbuf2, rbuf2, cnt2, MPI_INTEGER, MPI_AND, C1);
}
if(rank== 3 || rank == 4 || rank == 5)
{
 MPI_Iallreduce(sbuf2, rbuf2, cnt2, MPI_INTEGER, MPI_AND, C1);
 MPI_Ialltoall(sbuf,cnt1,MPI_INTEGER,rbuf,cnt1,MPI_INTEGER,C0);
}
```

The order of non-collectives must be preserved per communicator as it is possible to have two same collectives having identical signatures being called one after the other and hence would become impossible to distinguish between them if the order preservation condition is taken away unless some form of tags are used. The current MPI specification [1] does not specify the use of tags with collectives. A

future MPI specification might remove the constraint of maintaining the order on non-blocking collectives but the relaxation must be weighed against the potential benefits of this modification.

3.6 Dynamic rank assignment to threads in a thread-as-rank model

Threads in threading libraries like Pthreads, OpenMP etc. are capable of spawning threads. For example, the *pthread_create()* function can be used by any previously created thread to create more threads and a parallel region within a parallel region leads to the creation of threads by threads in OpenMP. In the following discussion when threads are created for the first time, we refer to them as the first-level threads. When the first-level threads create other threads, they are addressed as second-level threads. The threads which are created by the second-level threads are in general called n^{th}-level threads. For example, the number of threads which are specified with the help of OMP_NUM_THREADS environment variable before the launch of a hybrid application are first level threads. These threads can spawn more threads (second level) within the application code by using another parallel region within a parallel region i.e. using *#pragma omp parallel*.

3.6.1 Dynamic process management in MPI

MPI currently provides a detailed specification of the creation and management of processes created dynamically. Due to applications in which processes need to be created dynamically [13], for example, servers creating separate processes to cater to client requests, server task farms etc., MPI has provided several functions for dynamic process creation and management. The function *MPI_Comm_spawn()* creates an *intercommunicator* connecting two groups, the spawning process group (*local* group) and the spawned process group (*remote* group). It is a collective operation over the parents i.e. the spawning process group and also with respect to the calls of *MPI_Init()* function in the processes that are created. The new processes reside in a different MPI_COMM_WORLD i.e. the communicator containing the parent and the communicator containing the spawned children/child is different. An inference of the previous discussion is that the size of the parent communicator does not change. For a collective operation to be effective on both the communicators, it would require two different calls - one for the parent *intracommunicator* and the other for the newly created *intercommunicator*. The remote group of processes and the local group can be merged into one *intracommunicator* using *MPI_Intercomm_merge()* and hence, take advantage of the collective operations. But this newly created *intracommunicator* is still different from the parent's original *intracommunicator*.

3.6.2 Assigning ranks to threads with endpoints

The concept of endpoints is an attempt to assign ranks to various threads in the OS process in a process-based model so that the threads can communicate independently of the process interface. The problem lies in a limited set of endpoints being distributed to an OS process. If there are a large number of threads contending for a rank from a small set of endpoints, the delay in obtaining a rank can be nondeterministic and possibly cause *starvation*[8]. Further, it is only when the request for ranks is issued at application launch time that the granted ranks for threads become part of MPI_COMM_WORLD. At the time of initialization (irrespective of a static or dynamic interface) the ranks are a part of MPI_COMM_ENDPOINTS. This is not intuitive as the threads now communicate over a separate communicator and cannot be attached to the parent communicator. It is unclear as to what happens when a thread in a particular process, which has attached itself to an endpoint, accidentally addresses a rank which is not attached to a thread in some other OS process. The threads may create other threads which would need ranks again but this problem is not addressed in the endpoints proposal [18] (not incorporated in MPI 3.0 standard [1]) and possibly the only solution is to request another set of ranks or use the existing ranks and thus increase the contention.

3.6.3 Management of second-level threads by McMPI

McMPI [9], a rank based implementation of threads, does not allocate ranks to second-level threads and assigns the responsibility to the parent thread to communicate on behalf of these second-level threads. Hence, effectively in McMPI if a thread creates other threads, the thread support level of these set of threads becomes MPI_THREAD_FUNNELED as all the requests of the second-level threads are channeled through their first level parent. Implementations like TMPI [7] do not support dynamic thread creation of the second-level.

3.6.4 Dynamic assignment of ranks using manager processes

We propose a solution to the problem of dynamic assignment of ranks from second-level to n^{th}-level threads by uniformly dividing the rank-space in non-intersecting intervals among hidden manager processes. When multiple threads spawn multiple threads, it must be made sure that each of them gets a unique rank and hence it requires serialization of accesses to the rank database. Further, it can be shown that there is no necessity of creating an intercommunicator of the newly created

[8]A thread which wants to communicate is *always* denied a rank by a competing thread. The term has its origin in scheduling of OS processes where a lower priority process in priority based scheduling is denied CPU time by higher priority threads.

MPI processes/threads and they can become a part of the existing communicator of the parent (except for MPI_COMM_SELF). The architecture of the scheme is shown in Figure 3.11.

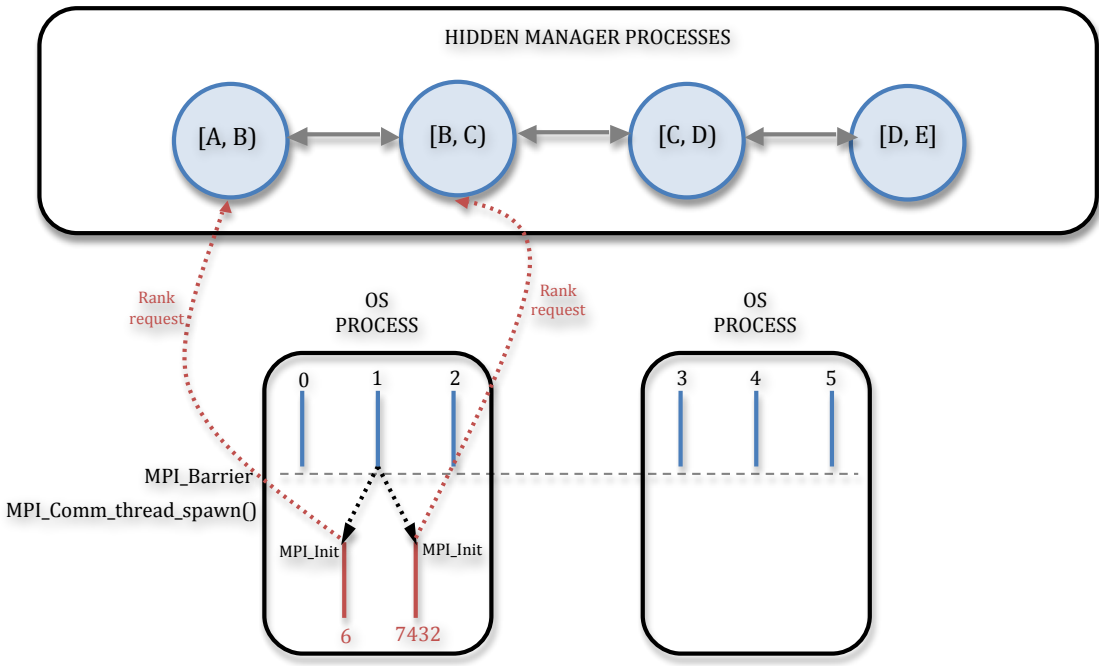

Figure 3.11: Assignment of ranks to second-level threads by manager processes

The manager processes start as soon as the application is launched and are not accessible/visible by/to the user. Assuming a rank space being represented by a 64 bit unsigned integer, each manager process keeps a particular range of ranks. For example Figure 3.11 shows four manager processes and each contains $\frac{2^{64}}{2^2}$ distinct ranks forming a continuous interval of unsigned integer values. The manager processes can communicate with each other in case a request for a rank arrives at a manager process which has run out of ranks. Such a communication is analogous to a primary DNS server forwarding a request to a secondary server for a name resolution.

At the implementation level each rank can be represented by the position of a bit in an array and the value of the bit specifies whether a particular rank has been allocated or not. For increasing the concurrency these arrays can further be divided so that locks of finer granularity can be applied to parts of the array and multiple requests can be processed simultaneously. The implementation must lock the sub-part of the particular array while searching for a rank for a newly created thread. Not acquiring the lock can possibly result in returning the same rank to two threads and hence making the rank distribution thread-unsafe. An implementation can choose to dynamically tune the number of manager processes depending on the number of threads which are initially spawned (in the thread-

as-rank model) at the application launch time. Figure 3.11 shows ranges of ranks in the form of a closed-open/open-closed/closed-closed non-intersecting intervals [X,Y)/(X,Y]/[X,Y].

The policy for assignment of ranks to threads remains uniform for first-level or second-level threads. Each spawned thread calls the *MPI_Init()* function (more specifically the *MPI_Init_thread()* function) which establishes a TCP connection with any manager process to request a rank (or alternatively a single UDP[9] request packet to speed up the process). The manager process returns a free rank or communicates with other manager processes (if it does not have a free rank) to obtain a rank and forward it to the thread.

A key decision is to choose a communicator for the newly ranked-thread. The most intuitive choice is to place the thread in the *intracommunicator* of the parent so that the need for merging any *intercommunicators* can be avoided and we can immediately start taking advantage of collective communications. The problem is that there could be other threads performing point-to-point/collective or other operations on the same communicator. A possible solution is to use a synchronizing barrier i.e. *MPI_Barrier()* on the parent thread's intracommunicator and let all the threads/MPI processes belonging to the communicator finish all operations on that communicator. This is followed by a call to a collective function *MPI_Comm_thread_spawn()* (hypothetical function - MPI does not have a function to create threads) from the *root/parent* thread and increment the size of communicator *atomically* by the number of new threads which are spawned. The collective communication ensures that each thread knows the new size of the communicator as soon as it returns from the blocking collective call *MPI_Comm_thread_spawn()*. A possible syntax for this call could be:

int MPI_Comm_thread_spawn(MPI_Comm comm, int numthreads, int root_rank)

The first argument specifies the communicator of the parent thread and it can have several values depending on which communicators the parent thread belongs to. If the value is MPI_COMM_SELF, then logically the size of the MPI_COMM_SELF should increase for this thread. Clearly, this would take away the meaning of MPI_COMM_SELF and hence the implementation can prohibit this particular value of the communicator. The second argument gives the number of threads which are to be spawned. The third argument is significant only at the *root* and gives the rank of the thread creating the second-level threads. It allows an implementation to maintain a hierarchical relation between threads. The biggest advantage is that the newly created threads belong to one of the communicators that the parent thread belongs to and can immediately start participating in collective communication. The choice of calling the *MPI_Init()* function within the function *MPI_Comm_thread_spawn()* as the last statement and incorporating a barrier after *MPI_Comm_thread_spawn()* would ensure that

[9] *User Datagram Protocol* (UDP) is an unreliable, connection-less, message oriented protocol in the third layer of the TCP/IP protocol stack.

all threads become aware of the new size of the communicator before they start any operations.

3.6.5 Problem of hardware resources in dynamic threading

When an application is launched on HPC machines, it usually reserves the number of cores, the time for which it will run, the number of processes that will be spawned or the distribution of processes and threads. Creating a thread dynamically would ideally create a need for new resources like a CPU core, memory resources, coordination with the operating system and possibly a restructuring of the topology. A possible method of addressing this problem is to estimate the resources in advance and overbook the resources in case the application needs to create threads and may need additional resources.

3.7 One-sided communication

A *Remote Memory Operation* (RMO) allows one process to directly access the memory of another process and accomplish data transfer without the remote process invoking any exclusive routines. It is analogous to the concept of *Direct Memory Access* (DMA) where an external processor can directly write into the memory of another processor, without the intervention of the second processor. The *Bulk Synchronous Parallel*[10] (BSP) style of programming also uses remote memory operations for communication with other processes. RMO can achieve greater performance as opposed to the pure *Message Passing Model* (MPS) in certain applications but requires special support from hardware. For example, commercial systems like the Cray T3D and Cray T3E used fast hardware support for RMO and an interface called *shmem* [30] (shared memory). The RMO model is different from shared memory programming model where simple variable names are used for referencing any memory address in a shared memory address space accessible to all threads of execution (of an OS process) [13].

Typically, remote memory operations are called Remote Memory Access (RMA) or One-sided Communication in MPI, as a single routine (usually) is equivalent to two routines in a pure message passing model. A typical RMA has three steps.

1. **Defining a memory window**: This defines a local contiguous memory of a process that will be used for RMA. This creates a new MPI object of type *MPI_Win*. The window object has the same role as that of a communicator in the message passing model. This operation is similar to declaring dynamic

[10] A model of programming where the computation and communication sections are separated by using barriers. Remote memory operations are used to get and put the data. Since the completion of communication and computation is handled by a single blocking barrier, these are called "bulk synchronous" [13].

memory in *shmem* as opposed to the usual approach of using statically allocated memory variables which are guaranteed to have the same address on each process (ensured by the Cray compiler and loader).

2. **Moving the data**: by specifying details of the sender and the receiver. The MPI RMA routines used for this purpose are: *MPI_Put(), MPI_Get()* and *MPI_Accumulate()*. The location in the remote memory window is a combination of offset provided in the origin process and the displacement unit specified in the target process.

3. **Determining completion of data transfer**: An *MPI_Win_fence()* operation is like a blocking *MPI_Barrier()* which guarantees that all pending operations will complete before the process enters the barrier. In the simplest model for ensuring data completion local writes to buffers in the window should be separated by *MPI_Win_fence()* from any *MPI_Put()*, *MPI_Get()* and *MPI_Accumulate()* operations as in the BSP model. This method is called *Active Target Synchronization* (ATS) as the target process (and all other processes that are part of the window object) must call *MPI_Win_fence()* to complete the RMA calls. A second method for carrying out ATS involves the origin process issuing a call to *MPI_Win_start()*, followed by a *MPI_Win_complete()* and the target process synchronizes[11] by issuing a *MPI_Win_post()*, followed by a call to *MPI_Win_wait()*.

   Passive Target Synchronization (PTS) makes use of *MPI_Win_lock()* and *MPI_Win_unlock()* routines at the origin process to synchronize it with the target process, without involving the other processes in the window object group and without the target process calling any exclusive RMA routine. For an atomic access a value of MPI_LOCK_EXCLUSIVE is used as opposed to MPI_LOCK_SHARED as an argument to *MPI_Win_lock()*. The latter allows multiple RMA operations simultaneously on the same window and the lock/unlock routines just imply completion of a RMA operation at the origin process. With MPI_LOCK_SHARED, the application developer must ensure that there are no conflicting concurrent operations modifying the same or overlapping parts of the window (see Figure 3.12). Implementations are allowed to restrict the use of PTS with a window allocated with *MPI_Alloc_mem()* routine which has the same function as *malloc()*. This particular memory is freed with *MPI_Free_mem()*.

3.7.1 Memory coherency with RMA

In general local operations and RMA operations can be conflicting. For example, if a local load operation on window is not separated from an *MPI_Put()* being carried out by a process on this window, it might produce a non-deterministic

[11] RMA operations between the starting synchronization call and the completing synchronization call are said to constitute an *epoch*.

result. Local load and store operations are not guaranteed to be atomic and hence a partial load and a partial *MPI_Put()* will produce an erroneous value of a variable (except for data being accessed atomically).

Operation 1	Operation 2	Conflict
MPI_Put	MPI_Put/MPI_Accumulate/load/store	Yes
MPI_Get	MPI_Put/MPI_Accumulate/store	Yes
MPI_Accumulate	MPI_Put/MPI_Accumulate/load/store	Yes
Store	MPI_Put/MPI_Accumulate/MPI_Get	Yes
Load	MPI_Put/MPI_Accumulate	Yes
Load	MPI_Get	No

Figure 3.12: Conflicting vs. non-conflicting RMA operations

As can be seen from Figure 3.12, only one combination of operations i.e. an *MPI_Get()* and a local load are non-conflicting. These operations are conflicting independently of the presence of multiple threads because a particular window could be the target for two or more remote processes. Hence using the simplest approach of the BSP model, the conflicting operations must be separated by an *MPI_Win_fence()* barrier. We do not consider conflicting load-load operations in the same window (this requires multiple threads) here as they do not involve any RMA operation.

3.7.2 Thread-Safety semantics of RMA operations in the thread-as-rank model

We first consider the ATS method of performing RMA operations using the synchronization forced by *MPI_Win_fence()* and assume the BSP model of programming. *MPI_Win_fence()* is a collective operation over the entire set of ranked threads that are a part of the communicator over which it is called. We suggest and explore two methods to ensure thread-safety.

The application of the first method demands that no thread in the origin OS process apart from the thread issuing the RMA call or any other thread in any OS process which is part of the communicator, should perform any RMA operation on the window which is being accessed by the origin thread, between the two calls to *MPI_Win_fence()*. Secondly, no thread of the target OS process can perform any local store/load operation. The second condition is accounted for in the BSP model of programming where local computation and RMA operations cannot be interspersed in the same *epoch*. Trivially, the second condition also implies that no local load/store operation can be done by any thread on the local window, as the either the data is being written or being read from the local window. Figure 3.13 shows illustrates the first method using three OS processes each of which contains

three ranked MPI processes/threads. The origin thread performs RMA operations on the window in the target OS process. As can be seen from Figure 3.13, the other threads perform a "no operation (nop)" which is practically equivalent to having an empty code block. Although this is thread-safe, it deteriorates the performance as the remaining threads (except for the origin thread) do not perform any useful computation. Further, several threads can participate in the operation *MPI_Accumulate()* with the usual restrictions as illustrated in Figure 3.12.

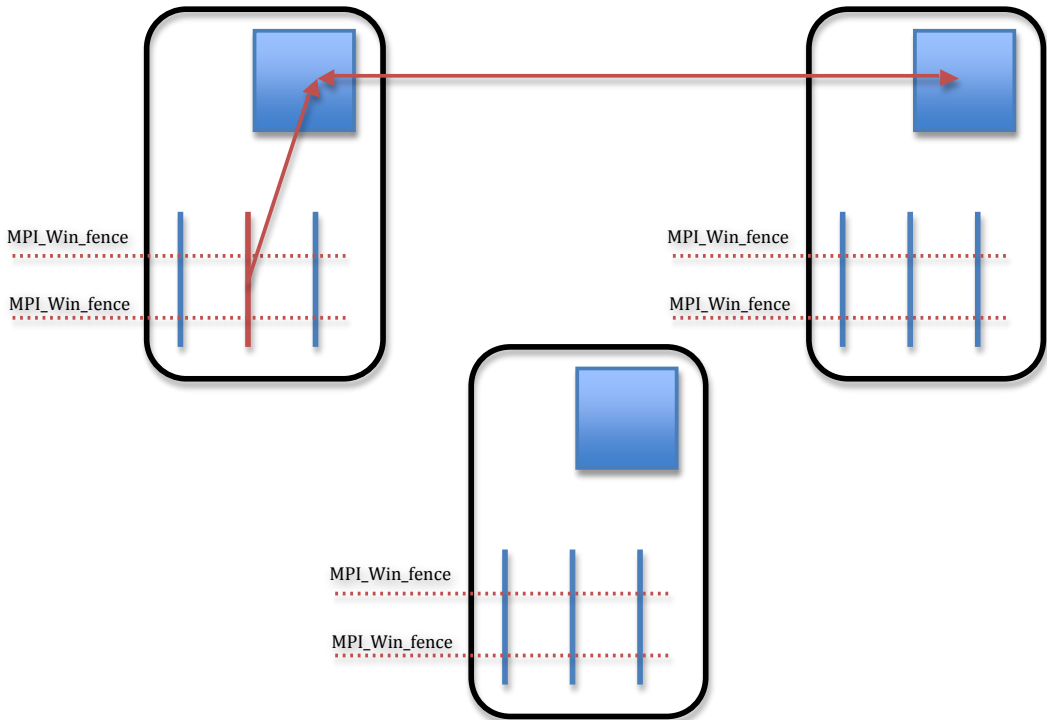

Figure 3.13: Thread-Safety with ATS using *MPI_Win_fence()* and empty *epoch*

The second approach we explore combines the ATS with the PTS model but subtracts away the BSP model of programming. What this means in effect is that now within an *epoch* initiated by *MPI_Win_fence()* operation, the origin thread can perform RMA operations while the other threads can perform non-conflicting MPI or non-MPI operations. We proceed to enumerate the salient points in this hybrid strategy.

1. The origin thread calls *MPI_Win_lock()* with MPI_LOCK_EXCLUSIVE on the local window followed by *MPI_Win_fence()* which in turn is followed by a *MPI_Win_lock()* on the remote OS process window. This ensures no thread in the OS process in which the origin thread resides performs any load/store on the local window as it is locked. Instead of calling *MPI_Win_lock()* on the local window, the origin thread can also acquire a per-object or global lock on the window-objects/window. The idea is that none of the other threads should be able to access the window during the

epoch, neither for reading or loading (conflicts with *MPI_Get()*) nor for writing or storing (conflicts with *MPI_Get()* and *MPI_Put()*).

2. The other threads in the OS process which contain the origin thread can perform any local computation using local non-window resources or RMA operations on any OS process window but they must obtain an exclusive lock. They can even request an exclusive lock on the target OS process which is currently being manipulated which, needless to say would be serialized.

3. The threads of the target OS process must not perform any operation in an *epoch* bounded by *MPI_Win_fence()*. To ensure this one and only one of the threads must obtain a per-object lock on all the objects in the window. This would ensure that an RMA operation can proceed but not a local load or store operation on the window objects. The locks can be released before the *epoch* finishes.

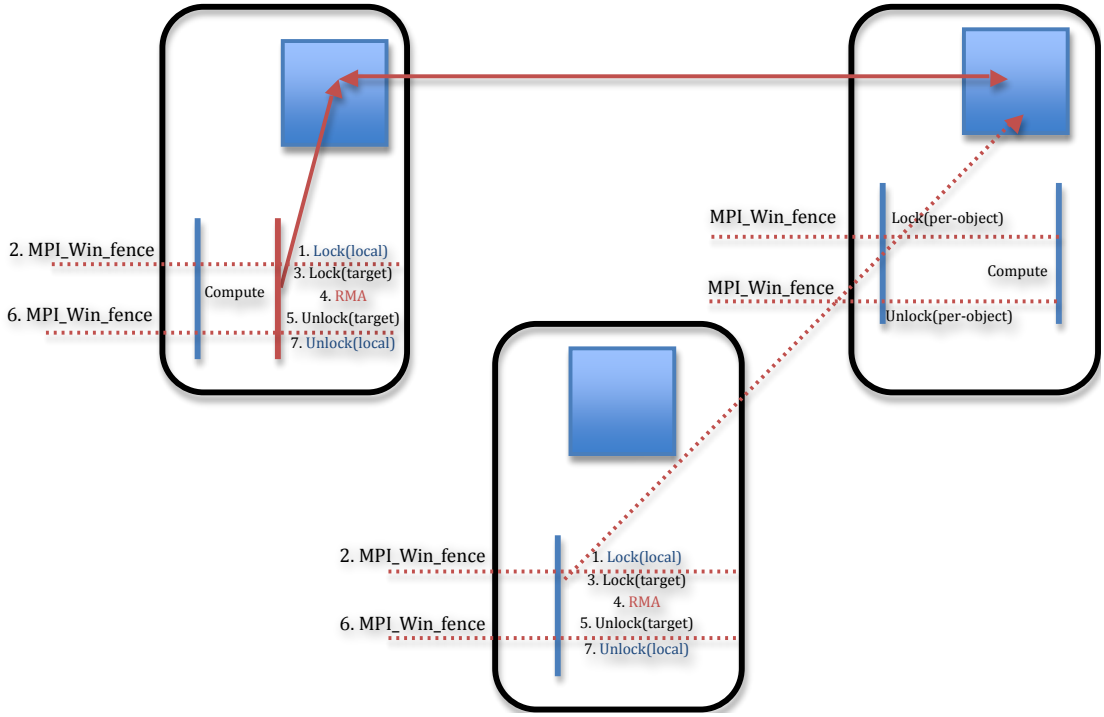

Figure 3.14: Thread-Safety with ATS-PTS using *MPI_Win_fence()* and non-empty *epoch*

Figure 3.14 shows the second method of making RMA thread-safe. The order of operations, wherever relevant (and complex), is indicated by numbers preceding the operation. The dotted lines show an operation which is not currently active.

A thread-safe scheme for a completely *Passive Target Synchronization* (PTS) method of RMA should also be devised. However, due to its expected complexity, it is not being explored due to limited time and to limit the scope of the project but has been added to the future work.

Chapter 4

Quantitative analysis

The two main types of computational complexities used in the analysis of algorithms are (1) *Space* and (2) *Time* complexity. The latter of the two i.e. time complexity enjoys a wider attention in literature as the space complexity is of diminishing importance due to a steady increase in the working memory. Hence, an increase in the memory footprint of an application is becoming more and more acceptable. The space complexity can be defined as the number and size of objects used in the solution used to resolve contention and restrict access to the critical section. *Time* complexities are calculated with regard to a hypothetical machine where the time taken to execute an operation is assumed to be $\mathcal{O}(1)$ (read as '*Big Oh*' of 1) for a single comparison, assignment, evaluation and other types of operations. The aim is to quantify the the *worst* case or the *average* performance in terms of the number of influencing entities, for example, like the number of keys in a sorting algorithm.

An additional measure of complexity related to the number of lines of code used in software projects is *Lines of Code* (LOC) which only brings to light the increase in the *source code* size of a particular piece of code. This is misleading in terms of the resources which are devoted to running the piece of code which certainly have an impact on the scalability, performance and load on the system. Examples in such classes include *recursive* algorithms which incur a heavy cost of computation and resource allocation on the system stack as opposed to their *iterative* counter-parts whose LOC might far exceed the former.

In algorithms which use locks, atomic lock-free operations or a combination of these, two cases are of particular interest: *deadlock* freedom and *starvation* freedom. The latter is stronger than the former and implies that all the threads trying to enter their critical section will eventually enter their critical section. A critical section is piece of code/pseudo-code which can only be executed by only one thread at a time. An algorithm is supposedly fast if in the absence of contention the critical section can be entered in a constant amount of time [24]. The contention-free time complexity is not always $\mathcal{O}(1)$ for all implementations, as is

mostly the case in lock-free algorithms and certain lock-based algorithms. For example, the contention-free time complexity in our thread-safe *MPI_Info()* algorithm Algorithm 3.1 is $\mathcal{O}(1)$. But for the lock-free algorithm using *malloc()* i.e. Algorithm 3.3, the complexity depends on the number of elements to be traversed in the shared array Q and hence is certainly not constant-time ($\mathcal{O}(1)$) when the number of elements start growing.

4.1 Complexity analysis of thread-safety algorithm for *MPI_Info*

The space complexity of Algorithm 3.1 which solves the mutual contention between threads to grant access to a single thread using a lock based approach is proportional to the total number of shared locks of type *lock_t* used for every shared *MPI_Info* object. Hence, the total space complexity is equal to the size occupied by the total number of lock objects created. The computation of this complexity is both trivial and computationally unintensive.

To compute the worst case time complexity when threads compete against one another to acquire the lock, we assume n as the total number of threads in an SMP and k shared objects of type *MPI_Info*. Clearly the total number of lock objects is $n_L = k$. We further assume that the total number of operations is of the order of $\mathcal{O}(p)$ for some finite positive integer p and the time taken to perform each of these operations is $\mathcal{O}(1)$. These assumptions are valid as the implementation limits the number of keys which can be set for an *MPI_Info* object and further, setting the value for these keys, querying the value of a key, finding the N^{th} key etc. can be done in a very small constant amount of time. For a calculation on a real system, these values can be replaced with the actual system time the implementation uses. In general these assumptions are sufficient for calculating the time complexity on a hypothetical machine running on a homogeneous architecture.

In the worst case scenario, all the threads i.e. n are competing to acquire a lock to manipulate a particular object. The total time for performing these operations by a thread can be divided into two parts:

1. Time spent in waiting before the lock is acquired (T_{wait})
2. Time spent in performing the actual manipulation (T_{work}).

In general the upper bound on the waiting time for each thread is $T_{wait} = \mathcal{O}(np)$ i.e. no thread waits for a time longer than the product of maximum number of operations and the maximum number of threads in the SMP. For a detailed upper bound calculation and a precise total waiting time, the following can be noted regarding the waiting times (T_{wait}) and working times (T_{work}) of threads:

1^{st} thread: $T_{wait} = 0$ and $T_{work} = p$
2^{nd} thread: $T_{wait} = p$ and $T_{work} = p$

3^{rd} thread: $T_{wait} = 2p$ and $T_{work} = p$

.

.

n^{th} thread: $T_{wait} = (n-1)p$ and $T_{work} = p$

Summing up the various values of T_{wait} and T_{work}, the total time for all the threads (T_{total}) can be written as:

$$T_{total} = \frac{n(n-1)p}{2} + p \tag{4.1}$$

The second term in the equation above is not np i.e $T_{total} \neq T_{wait} + T_{work}$ exactly as depicted above because while a thread is performing p operations, the other threads are waiting for this interval of time. Hence the waiting time of a thread and operating time of another thread *overlap* and hence should correctly contribute only p to the total time and not $2p$. The average taken over the total number of threads i.e. n gives us the average time taken by any/one thread to perform the operation presented in Equation 4.2.

$$T_{average} = \frac{(n-1)p}{2} + \frac{p}{n} \tag{4.2}$$

For the case when there is only one thread performing the operation, this time reduces to $\Omega(p)$ for the lower bound and $\mathcal{O}(p)$ for the upper bound. This is correct as a single thread takes $\mathcal{O}(1)$ time to acquire the lock and performs p operations. Clearly, from the above analysis the worst case total time (T_{total}) is bounded by $\mathcal{O}(n^2p)$ when $n >> p$ (in Equation 4.1).

A more realistic/practical case would not have all the threads n competing for the same operation at the same time and hence would in-turn reduce the contention at any other instance of *MPI_Info* by spinning on the lock on the previous instance of *MPI_Info*. Further, if the k^{th} thread frees the *MPI_Info* object, the remaining $n-k$ threads would just query the function *IsExists()* in $\mathcal{O}(1)$ time and not perform any operation. The total time T_{total} in this case becomes $(k-1)p+1+(n-k)$ as the last $n-k$ threads acquire the lock and perform operations in $\mathcal{O}(1)$ time. Clearly, when $k = 1$ i.e. the first thread frees the *MPI_Info* object and the expression is bounded below (tight lower bound) by $\Omega(n)$.

4.2 Analysis of thread-safe algorithms for *malloc()*

To analyze the non-blocking and wait-free thread-safe algorithm for *malloc()* i.e. Algorithm 3.3, we assume that for each call of *malloc()* there are n threads which participate in that call. At each of these calls the participating threads are divided

into k classes denoted by s_i such that each class s_i, where $1 \leq i \leq k$ has an equal number of threads and the address allocated to *all* the threads in class s_i is A_i i.e. $\forall$ threads $t \in s_i$, $Address(t) = A_i$. Hence, for classes s_i where i is from 1 to k the following two properties hold.

$$\sum_{i=1}^{k} |s_i| = |s_n| \tag{4.3}$$

In words, Equation 4.3 states that the sum of total number of threads from each class is equal to the total number of threads participating in the *malloc()* call at any specific point in time. The second property is as follows:

$$\bigcup_{i=1}^{k} s_i = s_n \tag{4.4}$$

Equation 4.4 implies that the union of all sets of threads gives back the original set of threads participating in the *malloc()* call at any particular point in time.

We further assume, without any loss of generality, that the operation of the threads is *synchronous* in the sense that for each class s_i, only one of the threads proceeds after registering its address in the shared array Q and the others back-off and call *malloc()* again. Synchronicity further implies that these threads wait for other threads which have backed-off from other classes at *malloc()* and the function is called by all the threads which have backed off together. These assumptions have been taken to make the analysis more fluent and intuitive. A part of the time taken is spent traversing the shared array Q and searching for the value of the address which is assigned to a thread. If the address is found then the thread backs-off, otherwise it continues its search and executes the CAS (Compare-and-Swap) operation in order to register its address in the shared array. In the first part of execution, k threads in total, one from each class s_i register themselves and proceed. The remaining $n - k$ threads call *malloc()* again. The number of comparisons made by all threads in the i^{th} class before and including the time when one thread registers itself is given by:

$$\frac{n}{k}[\frac{i(i+1)}{2}] + (\frac{n}{k} - 1)i \tag{4.5}$$

The first term $\frac{n}{k}$ in Equation 4.5 gives the number of threads in partition class s_i and is same for all the classes as mentioned above. Each of threads traverses until the i^{th} position in the array as the previous $i - 1$ positions are occupied by the addresses assigned to the other threads from the first $i - 1$ classes. Finding the i^{th} position empty, the threads execute the CAS instruction and after one of them succeeds - it registers itself in the i^{th} position. The remaining threads in this class then take one more iteration and traverse until the i^{th} position to find out that

their address has already been taken and hence back-off. These two comparison cycles are taken into account by the terms $\frac{i(i+1)}{2}$ and $\frac{n}{k}-1$ in Equation 4.5. Clearly, the total number of comparisons to assign addresses to the first k positions in the shared array Q by one thread each from class s_i is then given by Equation 4.6.

$$\sum_{i=1}^{k} \frac{n}{k}[\frac{i(i+1)}{2}] + (\frac{n}{k} - 1)i \tag{4.6}$$

After the first collective/parallel execution of *malloc()*, k threads are successfully assigned memory and the remaining $n-k$ threads must call the function again in search of a different address. Equation 4.7 expresses the condition that holds just before the next set of k threads are assigned memory.

$$\sum_{i=1}^{k} |s_i| = n - k. \tag{4.7}$$

These threads are again partitioned into k classes with each class having a total of $\frac{n-k}{k}$ threads each. Repeating the same procedure again, the thread in the i^{th} class occupies a position at the $(k+i)^{th}$ index. Hence the total number of comparisons performed in order to register k addresses in the shared array, one from each class s_i where $1 \leq i \leq k$ and $|s_i| = \frac{n-k}{k}$ is given by Equation 4.8.

$$\sum_{i=1}^{k} \frac{(n-k)}{k}[ik + \frac{i(i+1)}{2}] + [\frac{(n-k)}{k} - 1](k+i) \tag{4.8}$$

$\exists\ x \geq 0$ and $x \in I^+$, denoting the number of sets of k threads each such that $\frac{n-xk}{k} = 1$, which after solving gives the value of x as:

$$x = \frac{n-k}{k} \tag{4.9}$$

Clearly, after x sets of execution of *malloc()*, $|s_i| = 1$ and hence only k threads remains having a different address A_i, each of which needs to register in Q and thus complete the assignment of different addresses to all threads. Hence, the total cost in terms of comparisons in the shared array Q is given by the following:

$$\acute{c} = \sum_{x=0}^{\frac{(n-k)}{k}} \sum_{i=1}^{k} \frac{(n-xk)}{k}[xki + \frac{i(i+1)}{2}] + [\frac{(n-xk)}{k} - 1](xk+i) + \sum_{i=1}^{k}(n-k)i + \frac{i(i+1)}{2} \tag{4.10}$$

In Equation 4.10, the first (double summation) term accounts for the comparisons taking place until and including the x sets of k threads each and the second term

(single summation) sums the number of comparisons in k sets having one thread each. Equation 4.10 only gives the cost of comparisons.

The total cost of memory assignment must also include the number of times *malloc()* is executed. Assuming that the order of execution time of *malloc()* is $\mathcal{O}(p)$ i.e. the function *malloc()* is assumed to be made up of p operations each of which takes approximately $\mathcal{O}(1)$ amount of time, the following is the total cost of executing *malloc()*:

$$np+(n-k)p+(n-2k)p+...+(n-xk)p+kp = p[nx-k(1+2+3+...+x)+k] \quad (4.11)$$

which on solving and substituting $x = \frac{(n-k)}{k}$ from 4.9 gives the total cost as:

$$T_{malloc} = p[\frac{n(n-k)}{2k} + k] \quad (4.12)$$

In practice, it is desirable to have a large value of $k \approx n$, indicating a high level of non-conflicting concurrency and reducing the number of times *malloc()* is executed due to an increased number of threads which back-off. Hence, it is important to realize the level of contention for *malloc()* in the application though it it unpredictable due the non-deterministic thread-schedules. This helps to decide whether one should incorporate a wait-free non-blocking *malloc()* strategy, as this implementation is not only complex but may back-fire due to the number of CAS based comparison operations which add towards the total execution time.

In general, any atomic instruction would employ locks at the hardware level which incurs some additional cost as compared to a simple comparison operation and also limits the number of non-competing threads which can register their addresses into the shared array Q. An improvement over this algorithm would be to alternate the entry of threads of each of these sets from the *rear* and *front* of the shared array Q and maintain two entry-point indexes. This would lead to an increase in the concurrency as two CAS operations can be carried out successfully on two different indexes and thus, two registrations of addresses in the shared array Q can be made at the simultaneously.

Algorithm 3.2 which uses a global lock for serializing the calls to *malloc()* can be a worthy competitor of Algorithm 3.3 if the value of k assumed in calculations above is not large enough. Further, the global lock approach is easy to implement, reducing the burden of the application developer to a few lines of source code. Assuming n threads and the order of operations in *malloc()* to be $\mathcal{O}(p)$, the total complexity of the global approach is given by the sum of wait times of n threads and the time taken by the last thread to execute *malloc()*, as depicted by Equation 4.13.

$$T_{total} = (n-1)p+(n-2)p+(n-3)p+...(n-(n-1))p+p = \frac{n(n-1)p}{2}+p \quad (4.13)$$

Equation 4.13 is clearly of the order of $\mathcal{O}(n^2p)$.

Let $T(n)$ denote the time taken to solve a problem of size n, which in this case is the time taken to allocate memory to n threads. The time complexity of non-blocking and wait-free *malloc()* can also be expressed as:

$$T(n) = T(n-k) + \mathcal{O}(p) + c_1 \quad (4.14)$$

where $T(n)$ is the total cost of allocating memory to n threads in terms of time, $T(n-k)$ is the problem size after k threads have been successfully allocated memory, $\mathcal{O}(p)$ is the cost of executing *malloc()* and c_1 is the total number of comparisons done in the shared array Q in the process of allocating memory to the first k threads in Equation 4.14. Expanding this expression gives:

$$\begin{aligned} T(n) &= T(n-k) + \mathcal{O}(p) + c_1 \\ &= T(n-2k) + 2\mathcal{O}(p) + c_1 + c_2 = ... \\ &= T(n-xk) + x\mathcal{O}(p) + c_1 + c_2 + ... + c_x \end{aligned} \quad (4.15)$$

where $x-nk = k$ at the beginning of the $(x+1)^{th}$ iteration indicating that only k threads remain to be allocated memory. Clearly, from Equation 4.9 $x = \frac{n-k}{k}$ and the expression in Equation 4.15 is equivalent to $T(n) = T(k) + (\frac{n-k}{k})\mathcal{O}(p) + c_1 + c_2 + ... + c_x$. $T(k)$ is the time taken to assign memory to the last set of threads and $T(k)$ can be solved in $\mathcal{O}(p) + c_{x+1}$ time or more precisely (substituting the value of x) in $\mathcal{O}(p) + c_{\frac{n}{k}}$ amount of time. Also, the sum $c_1 + c_2 + c_3 + ... + c_{\frac{n}{k}}$ is nothing but $\acute{c}$ in Equation 4.10. Hence, $T(n)$ can be precisely expressed as:

$$T(n) = \frac{n}{k}\mathcal{O}(p) + \acute{c} \quad (4.16)$$

which is of the order of $\mathcal{O}(np/k)$ if $\acute{c}$ is negligible. In the best possible case $k = n$ and hence $T(n) = \mathcal{O}(p) + \acute{c}$, which clearly shows that in the case when *malloc()* assigns diffrent memory locations to all the threads executing *malloc()* concurrently, the operation is perfectly parallel and uses only $\mathcal{O}(p)$ time. The analysis shows that there a gain of factor of $\mathcal{O}(n/k)$ over the global *malloc()* scheme which is of the order $\mathcal{O}(n^2p)$. For an *exact* comparison, the value of $\acute{c}$ should be computed precisely. The same mathematical treatment can be applied to Algorithm 3.4 which uses the Exponential Back-Off policy to generate a random time interval which causes threads to arrive at different times at *malloc()* after backing-off. Analysis of Algorithm 3.4 is deliberately being avoided here to avoid unnecessary repetition and to limit the scope of the project.

4.3 Time complexity of *free()* compatible with non-blocking and wait-free *malloc()*

Algorithm 3.5 first nullifies the value in the global shared array Q and then calls the standard ANSI C function *free()* to deallocate the memory assigned to a thread. The complexity of these operations should be dependent on the time taken to search for an entry in the shared array Q and then the order of operations in *free()*.

Assuming ℓ = MAX_THREADS * MAX_MALLOC i.e. ℓ denotes the maximum length of the array Q. Also, let MAX_THREADS = n and MAX_MALLOC = m. In the worst case all the threads have called all the *malloc()* functions on their path of execution and call *free()* at the end. Without any loss of generality, we assume that *thread 0* has all its entries in the first m = MAX_MALLOC positions (1^{st} block), *thread 1* has all its entries in the m = MAX_MALLOC positions in the 2^{nd} block and so on. The first thread must make $(1+2+3+...+m)$ comparisons, the second thread must make $(m+1)+(m+2)+...+(m+m)$ comparisons and so on. Clearly the n^{th} thread must make $((n-1)m+1)+((n-1)m+2)+...+((n-1)m+m)$ comparisons. In general the i^{th} thread needs $(i-1)m^2 + \frac{m(m+1)}{2}$ comparisons. Summing up the comparisons we get:

$$T_{comparisons} = \frac{n(n-1)}{2}m^2 + \frac{m(m+1)}{2} = \frac{(n-1)}{2}\ell m + \frac{m(m+1)}{2} \tag{4.17}$$

In addition to $T_{comparisons}$ as given in Equation 4.17, each thread calls the ANSI C function which incurs a cost of $T_{free} = \mathcal{O}(p)$ operations per thread and hence the total cost of freeing all the chunks of memory is $T_{total} = T_{comparisons} + T_{free}$.

4.4 Analysis of shared buffer problem in point-to-point communication

The analysis of Algorithm 3.6 can be divided into three logical parts, one each for a Send, Receive and Write/Read problem. We first exhaustively derive the time complexity for a non-local *MPI_Send()* operation and then logically argue to show that the time complexity for an *MPI_Recv()* and *Write()* operation is approximately the same. In all the parts we assume $\bar{n}$ threads which participate in that operation, where $\bar{n}$ is less than the total number of threads n in the SMP node. An approximation is taken here to ensure that there is a conflicting operation which is not the same as the operation being analyzed. For example, we cannot have n threads carrying out the local send operation as there would be no thread which posts the receive operation.

Considering the case when $\bar{n}$ threads want to perform a non-local send operation, the *Query()* function takes $\mathcal{O}(1)$ time to execute as the rank can just be mapped to an index in the *Rank* array and the value at that index returned as the result. Assuming that the order of operations in both *MPI_Send()* and *Write()* is of the order of $\mathcal{O}(p)$, in the worst case all remaining $n - \bar{n}$ threads contend to perform a write operation and thus spin on the *w_lock* as well. In such a case if the *MPI_Send()* operation is performed *after* the remaining threads perform the *Write()* operation and keep performing this operation before any thread performs the *MPI_Send()* operation, the T_{wait} and T_{work} times of $\bar{n}$ threads is the sum of the following:

1^{st} thread: $T_{wait} = (n - \bar{n})p$ and $T_{work} = p$
2^{nd} thread: $T_{wait} = 2(n - \bar{n})p$ and $T_{work} = p$
.
.
$\bar{n}^{th}$ thread: $T_{wait} = \bar{n}(n - \bar{n})p$ and $T_{work} = p$
Summing up for all the $\bar{n}$ threads we get the total time T_{total} as:

$$T_{total} = p(n - \bar{n})\frac{\bar{n}(\bar{n} + 1)}{2} + \frac{\bar{n}(\bar{n} - 1)p}{2} + p \tag{4.18}$$

Equation 4.18 clearly has a tight upper bound of $\mathcal{O}(p(n-\bar{n})\bar{n}^2)$. Further, it can be verified by substituting $\bar{n} = n$ that the total time taken by n threads to perform *MPI_Send()* is of the order of $\mathcal{O}(n^2p)$ as T_{total} reduces to $\frac{n(n-1)p}{2}+p$. In the case of a local *MPI_Send()* being attempted by $\bar{n}$ threads, the time complexity is again of the order of $\mathcal{O}(p(n-\bar{n})\bar{n}^2)$ as it can be logically argued that the remaining $n-\bar{n}$ threads perform a *Write()* operation first and then post an appropriate local receive operation. In this argument based proof for time complexity we assume that the send operation after performing $\mathcal{O}(p)$ operations releases the *w_lock* and allows another thread to acquire the lock which is enough time for the receive operation to complete. If we do not make this assumption then the time complexity of a local send by $\bar{n}$ threads increases by $\bar{n}p$. The time complexity for an *MPI_Recv()* and *Write()* operation can be similarly derived and expressed as in Equation 4.18. In both the cases i.e. *MPI_Recv()* and *Write()*, we can assume that there are $n-\bar{n}$ threads trying to perform a conflicting *Write()* or a local/non-local *MPI_Send()*, respectively.

4.5 Abstract reference model for estimating the complexity of thread-safe source code

We attempt to estimate the number of lines of code (LOC) in the source code which will make a thread-unsafe code thread-safe. The motivation is to judge the

complexity of coding while making a code thread-safe. The model is abstract because the exact number of LOC would depend on the actual application, methods for acquiring locks, usage of shared variables/data-structures and the total number of thread-unsafe scenarios. For building this model we assume the following and give the reason for the assumption along with it.

1. **Assumption**: A lock-based approach will be used for with two basic operations of locking and unlocking an object/function. The declaration/initialization of a variable of *lock_t* type will not be counted towards the lines of source code. Further, the algorithms established in this project can be used as the details of these are completely available in the project.
   Reason: Lock based approach is simple and deterministic as compared to a lock-free approach, which can be implemented in several ways. Using a lock object automatically implies it has been declared/initialized.

2. **Assumption**: The performance factor is not taken into account i.e. code optimization is not our aim.
   Reason: To account for performance we again need to choose between numerous lock-free approaches maximizing the concurrency or lock-based approaches. Since the former is not being used, we cannot guarantee or emphasize on performance of code.

3. **Assumption**: No reordering of code is permitted.
   Reason: If code reordering is permitted then for example all *malloc()* calls of a thread can be clubbed together and made thread-safe by using a single lock. This is not possible for operations like *MPI_Send()* etc., and hence not permitted.

4. **Assumption**: Some functions in the implementation are not thread-safe at the implementation level.
   Reason: The assumption is realistic as a function like *MPI_Buffer_attach()* is not thread-safe in *MPICH2*.

5. **Assumption**: The function calls are either point-to-point, collective, RMA, functions which require dynamic allocation, completely-thread unsafe functions or MPI functions which manipulate or read attribute values of an object which can be freed.
   Reason: The assumption has been made as the current project addresses thread-safety scenarios related to the categories of functions mentioned above only. Further, this is an approximation model and not an exact estimate.

We describe the model using a flow-chart like approach (but not exactly a flow-chart due to a loose ordering of operations). The estimated lines of code for a particular type of function can be calculated by going from the function type till the end of flow. The following discussion is with reference to Figure 4.1 showing the Abstract reference model.

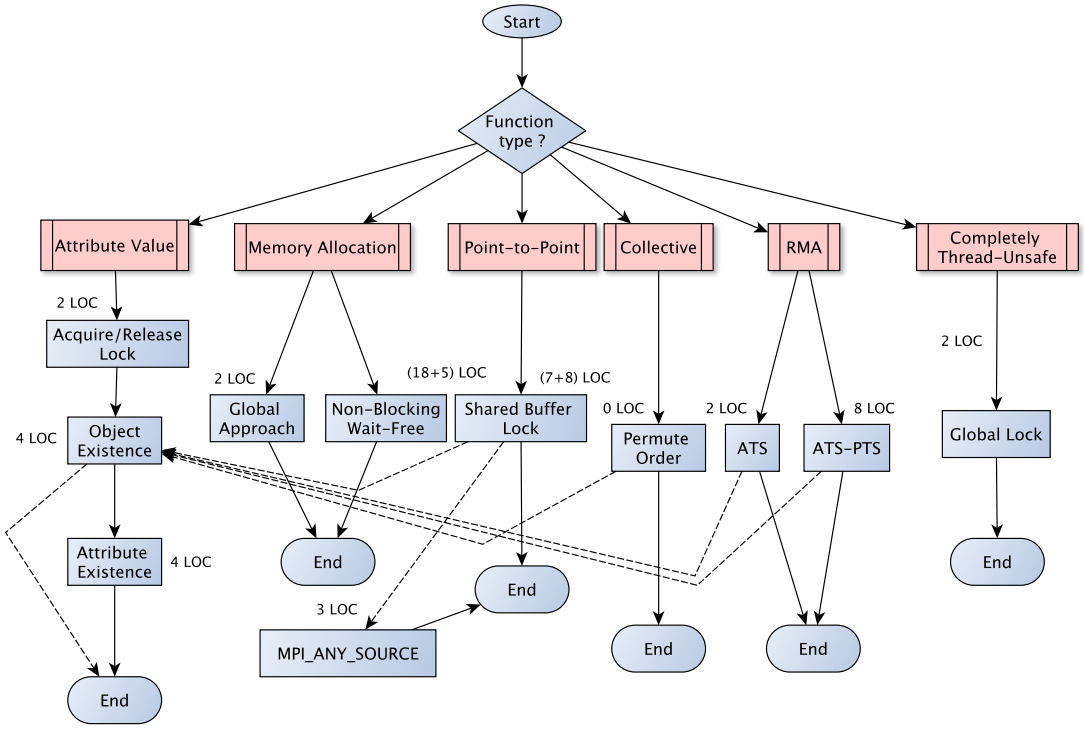

Figure 4.1: Abstract reference model for calculating approximate LOC

There are six high level function types namely (1) *Attribute Value* (AV) (2) *Memory Allocation* (MA) (3) *Point-to-Point* (P2P) (4) *Collective* (C) (5) *RMA* (6) *Completely Thread-Unsafe* (CTU).

The AV class consists of functions like *MPI_Info_set()* and *MPI_Comm_size()* etc. that set the value of an attribute, query the value of a fixed attribute of a hidden object etc. It becomes important ascertain whether the object or the attribute being manipulated exists or not. This was elaborated in Algorithm 3.1. The Lines of Code (LOC) marked along the arrows are estimated from Algorithm 3.1 and also can be inferred from the generalized flow-chart shown in Figure 3.3. If the attribute is not dynamically added or created, then we can move directly from the node marked "Object Existence" to the node marked as "End" (shown by a dashed arrow).

The MA class caters to functions which call dynamic memory allocation functions or simply allocating objects/variables dynamically. This specifically refers to the C language function *malloc()*. There is a sub-category of functions which call *malloc()* internally for e.g., *MPI_Comm_create()*, *MPI_Comm_split()* and *MPI_Bsend_init()* etc. The global approach to locking as described in Algorithm 3.2 takes only 2 LOC and the non-blocking wait-free method takes about 18+5 = 23 LOC, including memory allocation and deallocation as described in Algorithm 3.3 and 3.5, respectively.

The P2P class can be taken care of by Algorithm 3.6 but there is subtle minor scenario which should be taken care of. A simple buffer/user datatype being used inside a point-to-point operation can be freed and hence calls for checking the existence of the object. This takes about 15 LOC for an *MPI_Send()* and *MPI_Recv()* (separately). The point-to-point may further display an MPI_ANY_SOURCE thread-unsafe scenario, which can be taken care of by pseudo-code shown in Listing 3.2 and takes approximately 3 LOC.

The C class can be made thread-safe by changing the order of collectives to get the correct order, by removing multiple collectives on the same communicator by multiple threads and hence takes 0 LOC. The only issue here is that if it is using a dynamic object which can be freed, it must be tested for its existence and hence the arrow going from C class to the node "Object Existence".

The RMA class can be made thread-safe using the methods described in section 3.7.2 and Figures 3.13 and 3.14. The ATS (empty epoch 3.13) approach takes only 2 LOC and the ATS-PTS approach, which is more flexible takes about 8 LOC. Further, a routine like *MPI_Mem_alloc()* which might have allocated memory to the window objects would fall in the MA category. It is again important to check for the existence of objects in the window and hence an arrow from nodes labeled "ATS", "ATS-PTS" to "Object Existence".

The last class CTU has "some" function which is not thread-safe implementation-wise and hence should be called by a single thread only. The lock approach yields at least a LOC of 2.

Chapter 5

Conclusion

This chapter summarizes the project by giving an overall view of the various investigations carried out and the ideas proposed in the current project. The generalized conclusions which can be drawn from the study are summed up. The summary is occasionally supplemented with a conclusion and thus fragments of the latter can be found alongside the summarized work. Further, it ends with a critical evaluation of the project and various ideas for future work.

5.1 Summary of the project

This project is a thought experiment which proposes and discusses some algorithms for various scenarios requiring thread-safety due to MPI functions and attempts to generalize them to make them applicable to a wide range of functions. Code snippets have been used instead of pseudo-codes to highlight more details when a proper algorithm does not suit the scenario. A brief summary of the project is as follows:

An idea of the correctness of a program is constructed and proposed that mathematically defines a correct/partially correct/quasi-erroneous or a completely-erroneous program. Shared resources like allocated buffers, objects defined by MPI and their associated attributes or attribute values can be modified or freed/deleted by threads and thus produce a thread-unsafe program if there are concurrent conflicting operations being carried out by other threads simultaneously. The *MPI_Info* object is used to illustrate a general category of scenarios requiring thread-safety. A simple lock-based scheme is used to first query the object's existence and then carry out any modification/updation/addition of any attribute or value. The existence of an object and its attributes can be treated separately and this is generalized into a sequence of operations and illustrated by a flow-chart. A lock-based and lock-free approach to make *malloc()* thread-safe is explored. Though the complexity of implementation of a non-blocking wait-free *malloc()* is

estimated to be much higher than the scheme using a global lock, the former is expected to give higher performance as it exploits concurrency of non-conflicting accesses.

MPI_Send() and *MPI_Recv()* are used to illustrate the shared buffer problem when multiple threads in an SMP produce conflicts by writing, posting local/non-local *MPI_Send()/MPI_Recv()* routines on shared buffers. An application thread-unsafe scenario is created when MPI_ANY_SOURCE is used in *MPI_Recv()* function and can be averted by the use of *MPI_Probe()* in an inefficient manner. The semantics of blocking and non-blocking collective operations are discussed using pseudo-codes. It is shown that it is not possible to rectify a thread-unsafe blocking collective scenario due to a wrong permutation of calls being posted on the same or intersecting communicators. Prevention is suggested as the best cure for the same. A future specification of the MPI standard might remove the order of calls on non-blocking collective operations using additional tag fields. Implementations are free to use an additional tag field as per the advice of the current MPI 3.0 standard.

A thread-safe way of assigning ranks to threads spawned dynamically is presented by using hidden manager processes and a hypothetical collective operation namely, *MPI_Comm_thread_spawn()*, which adds the newly ranked thread to the communicator of the parent thread specified as an argument to the function. The time complexities of various proposed algorithms are derived and analyzed which serve as a basis for recommending lock-based or lock-free thread-safety solutions in various scenarios. Two methods for making RMA operations thread-safe are proposed. The first method uses a mutually agreed upon empty *epoch* surrounded by two calls to *MPI_Win_fence()*, with only the origin MPI process/thread issuing RMA calls. The second method uses a combination of *Active Target Synchronization* (ATS) and *Passive Target Synchronization* (PTS) techniques to provide a more flexible approach by letting the non-origin and non-target threads perform independent computations and further, allowing them to issue RMA operations on the target thread but protected by exclusive locks. An abstract reference model for approximating the number of lines of code (LOC) for making a given source code thread-safe is constructed.

5.2 Conclusions

We divide the conclusions into logical levels viz.: abstract or higher level conclusions and specialized or lower level conclusions. The abstract conclusions emphasize the importance of thread-safety as a whole and the specialized conclusions concentrate on the what can be learnt from the specific scenarios discussed in this project.

- **Abstract conclusions**: Several general/high-level conclusions can be drawn from the project. The semantics of a thread-as-rank model are different from

the rank-less-thread model/process-model that is defined in the MPI specification. Thread-safety is a broad field and when applied to MPI, involves resolving simultaneous conflicting MPI function calls by multiple threads by using any lock-based/lock-free or a hybrid approach. It is a clear fact and inference that a thread-unsafe program cannot be compared to its thread-safe counterpart for performance purposes as there is no comparison between a correct program and a non-deterministic erroneous program. It is not always possible that a program can be made thread-safe until the underlying application logic is changed and hence in practice not all programs should be made thread-safe. Thread-safety comes with a cost and implies that synchronization among threads incurs additional overhead. In addition to the application, the implementation should support thread-safe functions and states. The MPI specification tries to minimize the global variables/states and thus, in effect, tries to make the specification of functions thread-safe. Due to the wide variety of functions and their interactions, it is not possible to form an exhaustive set of rules for each and every function and hence functions should be categorized into generic classes to investigate thread-safety issues.

- **Specialized conclusions**: The following is a discussion pertaining to specialized or low-level conclusions. An object/attribute of an object which can be freed or deleted, when used in an MPI function, must be checked for its existence. This can be done by acquiring a per-object lock on that object to guarantee atomic access. This is true for any MPI function being used by any thread on a SMP. If the implementation of a function is thread-unsafe then its use must be locked in addition to the per object lock for the shared resources being used by it. Non-re-entrant ANSI C functions used by several MPI routines must be made thread-safe inside the implementation, as well as in the user code by using a lock-based or lock-free strategy. The global scheme of locking limits the access to a single thread and thus does not scale well when the number of threads are increased. Further, the time that threads spend spinning on the lock can be used to perform some useful computation if a lock-free non-blocking approach is used.

  The use of MPI_ANY_SOURCE wildcard can cause a mismatch if a message is overtaken by a wrong message and hence causing a dead-lock. The routines *MPI_Probe()* or *MPI_Mprobe()* can be used to match messages in a deterministic manner, thus making the application thread-safe. Two separate locks can be used to allow simultaneous local Send and Recv operations to proceed on a SMP and to incorporate a thread-safe non-local *MPI_Send()/MPI_Recv()* scenario, where only one of them can proceed at a time. Further, any conflicting read or write operations can be controlled with the same strategy. Blocking collectives can only be made thread-safe by correcting the permutation of posted collective operations to the same sequence on all communicators and also avoiding cycles due to intersecting

communicators. Removal of the order constraint for non-blocking collectives must be weighed against the usefulness of the concept and currently the author sees no significant advantage in doing so.

MPI endpoints propose a method to distribute a set of ranks to threads but may cause a serious contention problem in an increasing multi-core era. To circumvent this a thread-safe framework to assign ranks to various dynamic threads has been proposed which has an additional advantage of adding the threads to the communicator of the parent thread, thereby letting them take advantage of collective communications immediately. However a disadvantage can be seen when the size of the communicator is large and a synchronizing *MPI_Barrier()* possibly lessens the performance of execution. Thread-safety in RMA operations can be ensured by not letting any local load/store operations to proceed on the window while a one sided communication is under progress. Hence, it is the responsibility of the programmer to protect the objects in the window of the origin and target thread/MPI process from local load/store operations and RMA operations from other MPI processes in the communicator/window object. It is very difficult to estimate the number of lines of source code which are needed to make a thread-unsafe code thread-safe. Non-trivially it depends on the application, the thread-unsafe MPI routines being used, the frequency of usage of shared resources like buffers or objects and the method being used to make the routine thread-safe. Lock-free and hybrid methods for thread-safety are being explored to exploit the concurrency between non-conflicting operations and thus, enhance the performance of execution.

5.3 Critical evaluation

This section analyzes the level to which the aims in the final proposal were fulfilled and mentions some difficulties which were faced during the course of the project.

1. **First aim**: The first aim of the project was to formulate the necessary and sufficient rules for the thread-safety of a program. This aim was satisfied to a good extent in the form of formulation of algorithms, generalization and the illustration of some thread-unsafe scenarios with the help of code snippets. *Ideally*, each and every function in the MPI standard should have been examined (more than 300 functions) along with the methods to make them application thread-safe. But given the time for the project, it was only possible to examine the fuzzy representatives of the thread-unsafe abstract classes. Further, the MPI specification only gives an understanding of the rank-less thread model/process model, the semantics of which differ from a thread-as-rank model. The MPI standard currently does not recognize the thread-as-rank model and hence there are no established guidelines from bodies like the *MPI Forum* for the same that can limit or guide the think-

ing when applying standard MPI function semantics to the thread-as-rank model.

2. **Second aim**: The second aim was to quantify the difficulty in changing an existing pure MPI code to a thread-safe code which was explored by an abstract lock-based model determining the source code complexity in terms of LOC. The aim was found to be abstract in the sense that the code complexity non-trivially depends on the actual application code and further in practice while using lock-free methods, there are several ways and several data structures which can be used to make the code thread-safe. A lock based approach is not always preferred due to a possible performance bottleneck when scaling to thousands of threads but is considered in the abstract model due to the limited number of primitive statements which allows us to make the code thread-safe i.e. locking a lock object and unlocking it. Apart from skimming the surface while describing the abstract model, a time complexity analysis of the various algorithms created as part of the project was carried out in detail to give an estimate of the worst case execution time of an algorithm.

3. **Third aim**: The third aim of the project was to compare the performance of thread-safe snippets with thread-unsafe snippets. This aim was found to be redundant as mentioned in the conclusion and this was realizable throughout the project work. Thread-safety is about correctness and hence a thread-unsafe program cannot be compared to its thread-safe sibling. Thread-safety produces deterministic programs at the cost additional overhead incurred by using lock-based and/or lock-free methods. Ideally, this is not an overhead as the correct execution of the program is paramount.

4. **Difficulties faced/Aim fulfillment level**: A difficulty was the unavailability of a thread-as-rank MPI library in C language. TMPI (Thread MPI) was last updated in 2002 and does not support the thread support level MPI_THREAD_MULTIPLE. IBM MPI claims to be thread-safe but does not support the thread-as-rank model. MiMPI (Multithreaded Implementation of MPI), another thread-safe library became obsolete in 1999. The author of this project report being incompetent with C# language, could not use McMPI (Managed-Code MPI), though it supports the thread-as-rank model. Although the risks were known before the project began but a need for an implementation was still felt for possible application development and testing of ideas to input a feedback into the thought process. Overall, the main aims of the project that were to investigate/explore the necessary and sufficient rules to make a code thread-safe in a thread-as-rank model and to construct pseudo-codes/algorithms for the same, were met to a good extent. In addition to exploring some thread-unsafe scenarios in depth with the help of algorithms/pseudo-codes, the project also expanded breadth-wise by including an elaborate discussion on the three main communication models *in* MPI namely, point-to-point, collective and RMA operations and

consolidating the explored thread-unsafe classes into an abstract reference model for calculating the complexity of making a given code thread-safe in terms of lines of code (LOC).

5.4 Future work

The number of cores on SMP's are growing with time to gather more compute power and because the clock speeds of processors have reached a threshold due to high power dissipation. The increasing amount of shared memory of SMP nodes, developments in threading libraries for automatic work distribution, efforts to increase interoperability of MPI with other programming models etc. all point in a direction where threads will play a major role. Threads not only share the address space of the process, offer lower context switching time as compared to processes, utilize fewer resources during creation, they also remove the process-interface level interaction in MPI when equipped with a rank. Reaching the *Exascale* level is not just about running a large number of processes but about successfully running fully enabled applications which can solve a given task correctly in the shortest possible time, which would need correct thread-safe and semantics. We discuss some ideas pertaining to thread-safety in the thread-as-rank model which can be regarded as natural extensions to the current project.

- **Thread-Safe framework**: There is an immediate need to build a thread-safe framework for application thread-unsafe library functions found in the current implementations (process based model) compliant with the MPI specification and abstract away the details of making a function thread-safe from the user. Every function in the MPI library should be inspected to identify its thread safety requirements and a wrapper function/separate library doing the needful can be created. This leads to fulfillment of a three fold goal: maintaining correctness, making MPI more fault tolerant and reducing thread-safe application development time by reinventing the wheel. Automatic generation of help messages when an operation cannot proceed due to a conflicting operation by another thread can give an insight into the execution of the application. This is true specially for small to medium sized programs with the LOC being the measure. Incorporating such messages does not reduce the role of debuggers, visual viewers as the purpose for they have been created are many-fold for example, identifying bottlenecks, identifying/isolating errors in a single-threaded/multi-threaded process based programs, creating a graphical representation of execution etc. If the user so desires, the thread-safety features can be switched on/off at an application or individual function level. The latter supports usage of the thread-safe framework at a finer granularity and implies that whenever needed, a user can implement thread-safe logic for a function on his/her own, hence providing a flexibility in implementation in case the framework offers

a restrictive wrapper for a particular function.

- **Dynamic one-to-one rank assignment**: MPI endpoints offer a method for assigning ranks to threads but the contention among threads for a rank can become a performance bottleneck. A scheme for dynamically assigning ranks to threads on a one-to-one basis as suggested in the framework developed in the current project is recommended for realization. This not only solves the contention between threads by utilizing the rank space fully but also provides a method of making the ranks renewable and the possibility of incorporating the threads in the parent thread's communicator, thus letting them take immediate advantage of collective operations.

- **Passive Target Synchronization (PTS) thread-safety**: A scheme for making a scenario thread-safe when using only *MPI_Win_lock()* followed by a *MPI_Win_unlock()* for RMA should be devised and explored in detail. The complexity in this case arises because the target process does not actively issue an RMA call, because of which it is difficult to know the period during which the local-window should be protected from being corrupted by any target OS process threads.

- **Thread-Safe Thread-as-Rank MPI**: A thread-safe thread-as-rank (TSTR) MPI implementation in C language where a thread inherently has a rank needs to be realized. Statistically speaking, C language seems to be the preferred language for implementation. There have been several attempts to implement either a thread-safe or a thread-as-rank MPI library but not both. Further, the latest MPI standard prescribes bindings for functions in C, Fortran but C++ bindings stand deprecated. Internally, thread-safety can be achieved by using a lock-free, lock-based or a hybrid approach depending on the performance achieved. Functions for dynamic creation and manipulation of threads should be a part of the MPI implementation but the interoperability of MPI with other threading libraries must be maintained to cater to the needs of various applications and developers.

Bibliography

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. September 12, 2012.

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 3.1*. July 2011.

[3] *POSIX.1c, Threads extensions IEEE Std 1003.1c 1995*.

[4] Mpich.org. *MPICH | High-Performance Portable MPI*. [online] Available at: http://www.mpich.org/ [Accessed: 31^{st} July 2013].

[5] Open-mpi.org. 2013. *Open MPI: Open Source High Performance Computing*. [online] Available at: http://www.open-mpi.org/ [Accessed: 10^{th} June 2013].

[6] Smith, Lorna, and Mark Bull. *Development of mixed mode MPI/OpenMP applications*. Scientific Programming 9.2 (2001): 83-98.

[7] Tang, Hong, and Tao Yang. *Optimizing threaded MPI execution on SMP clusters*, In Proceedings of the 15th international conference on Supercomputing, pp. 381-392. ACM, 2001

[8] Erik D. Demaine. *A Threads-Only MPI Implementation for the Development of Parallel Programs* .In Proc. of the 11th International Symposium on High Performance Computing Systems (HPCS'97), pages 153-163, Winnipeg, Manitoba, Canada, July 1997.

[9] Daniel Holmes. *McMPI - a Managed-code Message Passing Interface Library for High Performance Communication in C#*. P.hD. Thesis, University of Edinburgh, October 2012.

[10] Caglar, Sadik G., Gregory D. Benson, Qing Huang, and Ch-W. Chu. *USFMPI: a multi-threaded implementation of MPI for Linux clusters*. In Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems, pp. 674-680. 2003.

[11] García, Félix, Alejandro Calderón, and Jesús Carretero. *MiMPI: A multithread-safe implementation of MPI*. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 207-214. Springer Berlin Heidelberg, 1999.

[12] Plachetka, Tomas. *(Quasi-) thread-safe PVM and (quasi-) thread-safe MPI without active polling*. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 296-305. Springer Berlin Heidelberg, 2002.

[13] William Gropp, Ewing Lusk, Rajeev Thakur. *Using MPI-2, Advanced Features of the Message-Passing Interface*. Scientific and Engineering Computation Series, 1999, ISBN-13 978-0-262-57133-3.

[14] Balaji, Pavan, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. *Fine-grained multithreading support for hybrid threaded MPI programming*. International Journal of High Performance Computing Applications 24, no. 1 (2010): 49-57.

[15] ISO/IEC 9899:201x, Committee draft N1570. International Standard, Programming Languages C, April 12, 2011.

[16] Goog-perftools.sourceforge.net. *TCMalloc : Thread-Caching Malloc*. [online] Available at: http://goog-perftools.sourceforge.net/doc/tcmalloc.html [Accessed: 25^{th} June 2013].

[17] Ftp. *Untitled*. [online] Available at: http://ftp://g.oswego.edu/pub/papers/C++Report89.txt [Accessed: 6^{th} July 2013].

[18] Dinan, James, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. *Enabling MPI Interoperability Through Flexible Communication Endpoints*. Conference Paper, EuroMPI 2013, Madrid, Spain.

[19] Andrew S. Tanenbaum. *Computer Networks*. 4^{th} Edition, Prentice Hall, International Edition (6 July 2009), ISBN-13: 978-7302172758.

[20] Keir Fraser. *Practical lock-freedom*. P.hD. Thesis, University of Cambridge, February 2004, ISSN 1476-2986

[21] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsevier 2008, Morgan Kauffman Publishers, ISBN: 978-0-12-370591-4.

[22] Skjellum, Anthony, Boris Protopopov, and Shane Hebert. *A thread taxonomy for MPI*. MPI Developer's Conference, 1996. Proceedings., Second. IEEE, 1996.

[23] Gropp, William, and Rajeev Thakur. *Thread-safety in an MPI implementation: Requirements and analysis*. Parallel Computing 33.9 (2007): 595-604.

[24] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson Education Limited 2006, ISBN: 978-0-13-197259-9.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Second Edition*. ISBN 0-262-03293-7 (hc.: alk. paper, MIT Press).-ISBN 0-07-013151-1 (McGraw-Hill).

[26] Protopopov, Boris V., and Anthony Skjellum. *A multithreaded message passing interface (MPI) architecture: Performance and program issues*. Journal of Parallel and Distributed Computing 61, no. 4 (2001): 449-466.

[27] *Intel® 64 and IA-32 Architecture, Software Developer's Manual, Volume 3A, Systems Programming Guide, Part-1*, June 2013.

[28] Wiki.mpich.org. 2013. *Developer Documentation - Mpich*. [online] Available at:http://wiki.mpich.org/mpich/index.php/Developer_Documentation [Accessed: 10^{th} August 2013].

[29] William D. Gropp. *Personal Communication*. 1^{st} July 2013.

[30] Cray Research. *Application Programmer's Library Reference Manual*, Vol. 1, November 1995. Publication SR-2165 2.0.

[31] Wiki.nss.cs.ubc.ca. 2013. *FineGrainMPI - NssWiki*. [online] Available at: http://wiki.nss.cs.ubc.ca/FineGrainMPI [Accessed: 21^{st} August 2013].

[32] Daniel Holmes. *Personal Communication*. 21^{st} August 2013.

Appendix A

List of symbols used

Symbol	Name	Usage	Meaning
==	Equal to	x == y	Is x Equal to y ?
$\forall$	For all	$\forall$x	For all x
$\leftarrow$	Assignment	x$\leftarrow$a	x takes the value a
$\leq$	Less than Equal to	x $\leq$ y	x is less than or equal to y
*	Product	x*y	Product of x and y
$\wedge$	AND	x$\wedge$y	x AND y
++	Increment	x++	x $\leftarrow$ x + 1
$\neq$	Not Equal to	x$\neq$y	x is Not Equal to y
[]	Inclusive Range	[x,y]	Including and between x and y
$\triangleright$	Comment	$\triangleright$ A comment	Not part of pseudo-code logic
:	Colon	private: x, y	Starting of a block
$\Rightarrow$	Implies	x $\Rightarrow$ y	x implies y
$\neg$	Negation/Not of	$\neg$ y	absence of y
$\vee$	OR	x $\vee$ y	x OR y
$\Longleftrightarrow$	Mutual Communication	x $\Longleftrightarrow$ y	x and y communicate
$\in$	In	x $\in$ y	x Belongs to/In y
$>$	Greater than	x $>$ y	x is Greater than y
$\gg$	Greater than greater than	x $\gg$ y	x is Greater than greater than y
$\sum_{i=a}^{b}(expr)$	Summation	$\sum_{i=1}^{n} i$	Sum of first n numbers
$\bigcup_{i=1}^{k}(set)$	Union	$\bigcup_{i=1}^{k} s_i$	$s_1 \bigcup s_2 \bigcup ... \bigcup s_k$
$\approx$	Approximately	x $\approx$ y	x is approximately equal to y
$\geq$	Greater than equal to	x $\geq$ y	x is Greater than or equal to y
$\exists$	Exists	$\exists$x	There exists some x
$\prod$	Product	$\prod_{i=1}^{5} i$	1*2*3*4*5
$\rightarrow$	Tends to/Limit	x $\rightarrow$0	x tends to 0 or x approaches 0
| |	Set cardinality	|x|	number of elements in set x
I^+	Positive integers set	x $\in I^+$	x is positive integer

Appendix B

Asymptotic notations

B.1 $\mathcal{O}(g(n))$ - *Big Oh* notation

Definition: $\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 \geq 0 \text{ such that } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$
$\mathcal{O}(g(n))$ is a set and all functions $f(n)$ which satisfy the definition above and formally we say $f(n) \in \mathcal{O}(g(n))$. The *Big Oh* notation gives the tight upper bound of a function within a constant factor [25].
For example:

$$\sum_{i=1}^{n} i = \mathcal{O}(n^2)$$

B.2 $\Omega(g(n))$- *Omega* notation

Definition: $\Omega(g(n)) = \{f(n) : \exists c, n_0 \geq 0 \text{ such that } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$
$\Omega(g(n))$ is a set of functions $f(n)$ which satisfy the definition above and formally we say $f(n) \in \Omega(g(n))$. It gives the tight lower bound of a function within a constant factor [25].
For example:

$$\sum_{i=1}^{n} i = \Omega(n^2)$$

Appendix C

Working sample code for Algorithm 3.1

```
/* (1) AUTHOR: Gaurav Saxena                */
/* (2) PROGRAM: Thread-safety for MPI_Info  */
/* (3) COMMENTS: Removed to manage space    */


#include<mpi.h>
#include<stdio.h>
#include<omp.h>

int IsExists(MPI_Info info);

void main(int argc, char *argv[])
{
    MPI_Info info ;
    omp_lock_t info_lock;
    int required = MPI_THREAD_MULTIPLE, provided ;
    int rank, size, threadID, threadProcRank ;
    MPI_Comm comm = MPI_COMM_WORLD ;

    MPI_Init_thread(&argc, &argv, required, &provided);
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);

    if(rank == 0 )
    {
        if(required != provided)
            MPI_Abort(comm, 1);
        else
```

```
            printf("\nMPI_THREAD_MULTIPLE was successfully initialized");
    }

    MPI_Info_create(&info);
    omp_init_lock(&info_lock);

    #pragma omp parallel private(threadID,threadProcRank) shared(comm)
    {
        threadID = omp_get_thread_num();
        MPI_Comm_rank(comm, &threadProcRank);
        printf("\n Thread ID %d, Process rank %d", threadID, threadProcRank);

        if(threadID == 0)
        {
            omp_set_lock(&info_lock);
            if(IsExists(info))
            {
            MPI_Info_set(info, "striping_factor","4");
            MPI_Info_set(info, "striping_unit","65536");
            MPI_Info_set(info, "start_iodevice","2");
            }
            omp_unset_lock(&info_lock);
        }
        else
        {
            omp_set_lock(&info_lock);
            if(IsExists(info))
            {
            MPI_Info_free(&info);
            }
            omp_unset_lock(&info_lock);
        }
    }

    MPI_Finalize();
}

int IsExists(MPI_Info info)
{
    if(info == MPI_INFO_NULL)
        return 0;
    else
        return 1;
}
```

C.1 Hardware

The program "Thread-safety for MPI_Info" was executed on Morar, a 128 core machine which is partitioned into two shared memory nodes of 64 cores each. Each processor is an AMD Opteron (TM) Processor 6276 with a speed of 2.3 GHZ approximately. The cache size is 2048 KB with a cache alignment of 64.

C.2 Method of job submission

The following SGE (Sun Grid Engine) script was submitted using the command:

qsub -cwd -pe mpi 4 Info.sge

The number of processes specified were four and the *Info.sge* file having the following contents specified the number of threads in each process using the environment variable OMP_NUM_THREADS.

```
#!/bin/bash
#$ -cwd -V

MPIEXE=‘basename $REQUEST .sge‘

echo "------------------------------------------------"
echo "Running MPI program <$MPIEXE> on" $NSLOTS "processes"
echo "------------------------------------------------"
echo

export OMP_NUM_THREADS=4

(time mpiexec -n $NSLOTS ./$MPIEXE) 2>&1

echo
echo "--------------------"
echo "Finished MPI program"
echo "--------------------"
echo
```

Appendix D

Implementing MPI routines without using *malloc()*

It is not necessary that an implementation needs to/must use the *malloc()* function for allocating memory to MPI routines. Elementary but customized implementations of *malloc()* can be implemented *easily* by using the basic *sbrk()* system call. As an example, the *MPI_Comm* structure can be statically allocated memory in the application code by the developer. To elaborate it further, assume a *binary tree* structure or a *binary hyper-cube* structure for collective communication as shown in Figure D.1.

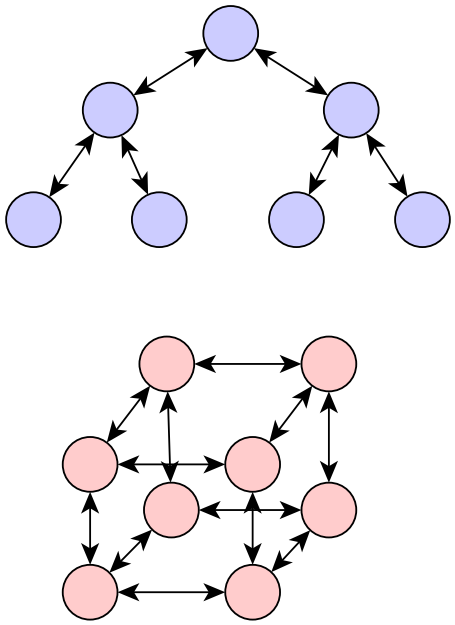

Figure D.1: Top: Complete Binary tree (n=7 nodes), Bottom: Binary Hyper-cube *(n=8 nodes)*

Each MPI process/thread/rank needs $\mathcal{O}(log_2(n))$ information to locate the remaining $(n-1)$ ranks. Assuming a million MPI processes i.e. $n = 1000000$, the value of $log_2 n$ is ≈ 19.93. If we assume that the user *statically* allocates approximately 32 structures per MPI process/ranked-thread which contain information about other ranks/MPI processes/ranked-threads, then the MPI process can locate and communicate with $2^{32} = 4294967296$ or approximately 4.2 billion MPI processes/ranked-threads [32]. Recent efforts using *FG-MPI* (Fine-Grain MPI) [31] have lead to a successful execution of more than a 100 million MPI processes. Clearly, using 32 *statically* allocated information structures for storing information regarding location of other MPI processes leads to a scale of at least 4 billion MPI processes (4000 Million) and hence, an MPI library may not be required to allocate space dynamically [32].