5-1-2016

Software Testing of Parallel Programming Frameworks

Beverly Klemme

Beverly Klemme
Candidate

Computer Science
Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dorian Arnold, Chairperson

Patrick Bridges

Patrick Kelley

SOFTWARE TESTING FOR PARALLEL PROGRAMMING FRAMEWORKS

By

BEVERLY KLEMME

B.S., Physics, University of California, Los Angeles, 1986
M.S., Physics, University of Illinois, Urbana-Champaign, 1988
Ph.D., Physics, University of California, Los Angeles, 1997

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

May, 2016

Dedication

For Calypso: My first horse.

Acknowledgments

I would like to acknowledge my colleagues on the UNM Scalable Systems Lab Cray testing team: Evan Dye and Whit Schonbein for their support and assistance.

Special thanks to my thesis adviser Dorian Arnold; without his mentorship this would have not been possible.

Thanks to Professor Patrick Bridges for his support and guidance.

Many thanks to the UNM Computer Science Faculty who were all first-rate and provided me with the skills, training, and knowledge to be able to complete this work and go on to an exciting new career in Computer Science.

Also, I would like to thank Mark Pagel at Cray Inc. for directing the projects and Krishna Kandalla for his assistance with the MPI Thread Safety Tests.

The projects in this thesis were conducted at the University of New Mexico with funding and support provided by Cray, Inc.

Disclaimer:
The software developed for this thesis was designed and tested for in-house use at Cray and not intended nor available for public distribution.

Software Testing for Parallel Programming Frameworks

by

Beverly Klemme

B.S., Physics, University of California, Los Angeles, 1986

M.S., Physics, University of Illinois, Urbana-Champaign, 1988

Ph.D., Physics, University of California, Los Angeles, 1997

M.S., Computer Science, University of New Mexico, 2016

Abstract

Parallel programming frameworks rapidly evolve to meet the performance demands of High Performance Computing (HPC) applications and the concurrent evolution of supercomputing class system architectures. To meet this demand, standards and specifications that outline the semantics and required capabilities of parallel programming models are being developed by committees of government and industry experts and then implemented by third parties. OpenMP and MPI are particularly prominent examples of such programming models and specifications, and are in common use in the HPC world. Comprehensive testing is required to be sure that any given implementation adheres to the published standard. The type and degree of testing depends on the goals of the developers. In particular, commercial implementations developed by companies for specialized applications (like HPC) will have much more stringent requirements than those for general applications.

This thesis describes the development of test suites targeted toward a subset of OpenMP and MPI features, namely processor affinity and thread safety, as implemented in Cray compilers and libraries. These tests, the focus of which were robustness, re-usability, and detailed error output, contributed to software quality for the wide range of applications for which Cray compilers are used, and continue to help Cray ensure correctness in their OpenMP and MPI implementations as their compilers and libraries continue to evolve.

Contents

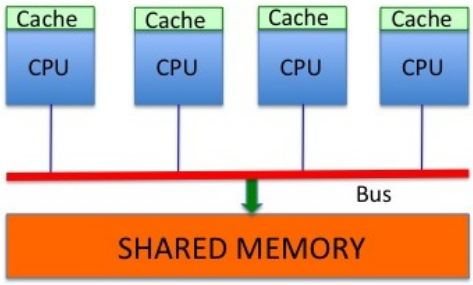
List of Figures

List of Tables

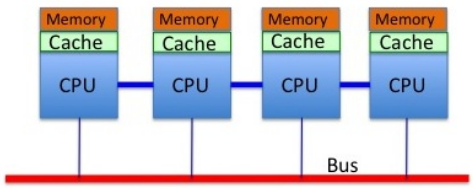
Chapter 1

Introduction

High performance computing (HPC) hardware and applications are rapidly evolving to meet demand for breakthroughs in fields such as biology, medicine, astrophysics, and climate modeling. These exciting and diverse domains call for ever higher performance and drive increases in complexity and heterogeneity of networking and memory architectures. For example, the current road-map for the development of exascale computing systems calls for an order of magnitude reduction in power budget from our current capabilities [4] . Because of power and cooling limitations, the speed of CPUs is no longer increased by upping clock frequency, but by increasing the number of cores per chip [12]. In HPC systems, multi and many-core CPUs are bundled together on nodes, which are connected by high speed interconnects. In addition, each core may run many threads. These systems can be heterogeneous in design, and use a combination of shared and distributed memory (Fig. 1.1). Memory access speed is currently the major limitation in application performance, and therefore communication mechanisms and pathways will determine the ultimate performance achievable. This, in turn, has driven developments in networking paradigms and chip architecture such as remote direct memory access (RDMA). These architecture developments require changes in communication and data access patterns that then must be implemented in software.



(a) Distributed memory architecture



(b) Shared memory architecture.

Figure 1.1: Two common memory architectures. In shared memory, for which OpenMP was designed, every CPU has access to a common pool of memory through a single bus and communication between processes is implicit. In distributed memory architectures, the bus connects the CPUs, but each CPU has it's own local memory. Communication in distributed memory architectures must be done with an explicit message passing mechanism (such as MPI). Most systems for high-performance computing will incorporate some combination of these two architectures.

To fully take advantage of innovations and architectural changes, parallel programming models such as *Open Multi-Processing* (OpenMP) and *Message Passing Interface* (MPI) have become pervasive HPC. These take the form of standards and specifications developed by committees of experts and implemented as compilers, libraries, and runtime frameworks for C, Fortran, and C++. The quality, efficiency, and correctness of results from these parallel programming models critically depend on software testing efforts that can keep up as new versions of these specifications are being continually rolled out [3][5]. The topic of this thesis is the development of test suites for key features of the new OpenMP and MPI standards.

1.1 The Need for Software Quality and Correctness in HPC Applications

It is important that the performance increases in HPC applications made possible by rapid innovations not come at the expense of basic features such as software quality, security, resilience, and correctness. However, test suites tailored to these quality features tend to take a back seat to the results of performance benchmarks and software correctness is assumed. The amount of effort put into ensuring that a given implementation conforms to a standard and/or produces a correct result is not as visible or appreciated; the danger in this is that mistakes could become apparent after-the-fact leading to huge economic and human losses [6]. This becomes even more important when considering the complexity of HPC applications: machine crashes or deadlocks are expensive and inconvenient but an incorrect result may remain undetectable for years, leading to erroneous conclusions resulting in a cascade of invalid research results, policy mistakes, and the resultant miss-appropriation of resources and scarce research dollars.

1.2 Project Motivation and Scope

This thesis describes the design and development of test suites to verify the correctness of several new features in OpenMP 4.0 and MPI 3.0 as implemented in the latest Cray libraries and compilers. Cray is a key player in HPC, and provides HPC systems to commercial and government agencies to support research into such fields as energy, financial services, health care, and weather forecasting. These systems have stringent quality assurance requirements because of the nature of their applications and the customers that they serve. So, although test suites for both OpenMP and MPI exist and can be found on the web, there is no assurance that they meet the targeted needs of Cray, or that documentation, error output, and the quality of the tests themselves is adequate.

To fully appreciate the need for internally developed test suites, targeted to specific compilers and applications, contrast the current situation in software quality to that of the system in place to assure the quality of physical measurements: For every system that purports to "traceably" measure a definable physical or electrical quality, traceability to a national standard (itself traceable to a agreed upon international standard at the International Bureau of Weights and Measures, France) is required. So in the US, high-stakes measurement systems used in government and key industries will only utilize instruments that produce measurements that are "NIST-treacable" (NIST - The National Institute of Standards and Technology - is the official national metrology institute for the US). The system of standards and traceability is supported by a sophisticated system of auditors and certification requirements and exists at huge government expense. Although NIST has a software quality division [10] and software quality assurance is beginning to be recognized [2], no comparable system currently exists for software quality, and there is not an external authority to verify the correctness of any software or testing suite that is developed by third parties.

Cost and schedule issues can also make the use of third party test suites infeasible: The standards themselves change every few years, and re-usability of the tests is a factor: For example, it may be advantageous from a cost standpoint to be able to evolve the tests along with the standards, and have them run on an in-house test harness so that code and infrastructure are both re-used to the extent possible. Intellectual property, export controls, and licensing issues may also come into play and can be hard to keep track of and document for third party software. Schedule is also an issue: tests must be completed in time for the next scheduled production release of the compiler.

Given these constraints, companies like Cray often direct the development of their own tailored test suites, so that the tests can be targeted toward their specific needs, and quality assurance can be closely monitored and controlled. The tests described here are the result of a collaboration between Cray and the University of New Mexico to develop robust

and comprehensive test suites for the latest OpenMP 4.0 and MPI 3.0 releases, which are being implemented as libraries in evolving Cray compilers. As part of that team, I was responsible for two major areas of testing: 1) The development of a new feature in the OpenMP 4.0 standard that allows for thread affinity policies that allow the programmer to specify exactly on which processor (or set of processors) a software thread will run; and 2) Thread-safety testing in MPI 3.0 for one-sided communication and non-blocking collectives. Both are necessary to allow for fine-grained performance tuning in OpenMP 4.0 and are especially important in ccNUMA systems [8].

1.3 Test Development for OpenMP and MPI

1.3.1 Choosing a Test Model

Software testing is still an art but the field has converged around a specific set of software testing models and strategies, and can be seen as existing on two levels: 1) General models that describe how testing and development should be scheduled and co-ordinated throughout the entire development process (for example, the Waterfall or General V-model) 2) Specific strategies (such as static code analysis or black-box testing) that describe the exact method of examining or evaluating code. Although no single model or strategy may fit a given testing scenario exactly, each one is targeted toward a specific software lifecycle design, programming team structure, risk analysis, application domain, and roll-out schedule. A very brief and general description of some of these models is provided in Chapter 3, but to summarize: This project mainly relied on black-box testing, and test cases were developed either through examination of typical use cases (for OpenMP 4.0 processor affinity) or simply to achieve a coverage of basic functionality (for MPI 3.0 thread-safety).

1.3.2 The Parallel Programing Frameworks

OpenMP is designed for shared memory systems and uses implicit communication, while MPI uses a completely different process-based model of parallelism and communication between processes is achieved through explicit message-passing [12]. These programming models use compiler directives or functions to allow the programmer to direct exactly how parallelism will be done. However, the strict division between shared and distributed memory within architectures has become increasingly blurred so there has been increased use of hybrid programming models such as MPI+OpenMP. With this increased flexibility and performance potential comes extra complexity in programming as well as the need for thread-safe implementations of MPI [7]. For this reason, the MPI thread-safety tests were both motivated by, and necessitated MPI+OpenMP hybrid programming.

1.3.3 Results Overview

The Tests for OpenMP 4.0 processor affinity were completed for C, C++, and Fortran Cray implementations and packaged into an automated test suite delivered to Cray. Thread safety tests for MPI 3.0 one-sided communication and non-blocking collectives were completed in C and forwarded to Cray for use in further optimization and development. Ultimately all tests passed using the latest Cray compilers and libraries. More importantly, these suites of tests provided Cray with a comprehensive and reliable method to validate OpenMP and MPI implementations that are in a process of continual optimization.

1.3.4 Outline of This Document

The outline of this document is as follows: Chapter 2 presents an overview of OpenMP and MPI, and challenges of each. Chapter 3 then outlines software testing models that are in common use, and how they relate to the testing undertaken for this theses. The rest of

the document is devoted to details of the OpenMP and MPI test development effort itself: Chapter 4 describes test planning, implementation and results for the development of the OpenMP tests. Chapter 5 summarizes the same for the MPI thread safety testing. Chapter 6 provides a summary and ideas for future work.

Chapter 2

Background

This section first briefly describes the parallel programming models and the specific features that were tested. Then it provides a level of technical detail about those features sufficient to understand the scope of the tests and the challenges involved in designing them. For OpenMP it is particularly important to understand how threads are created when reaching a parallel region. The MPI tests focused on thread safety, so general issues in MPI thread safety are also explored. The interested reader may explore more details of the OpenMP and MPI standards and parallel programming in [3][5][12][11][9][8].

2.0.5 OpenMP

OpenMP [3] is a standardized, flexible, portable, and easy to use parallel programming model specifically designed for shared-memory systems. Software threads are identified by parallelizing the underlying code. To use OpenMP programmers must specify which portions of the code should be parallelized using compiler directives.

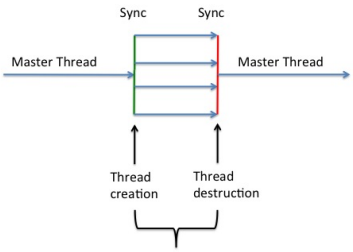
In all cases, threads are launched according to a "fork-join" model (Fig. 2.1), where a master thread launches a team of threads when encountering a parallel region. When

coming to the end of a parallel section, the thread team is synchronized and re-assigned to the pool. The master thread then continues until encountering another parallel section. In the case of nested parallel regions, the child threads created at the outer level each launch teams of child threads that go to work on the inner sections, so several generations of threads can exist. Communication speed can suffer if threads end up on processors far from the data that they must access. Therefore it is necessary to keep the underlying processor architecture in mind when developing programs that use OpenMP.

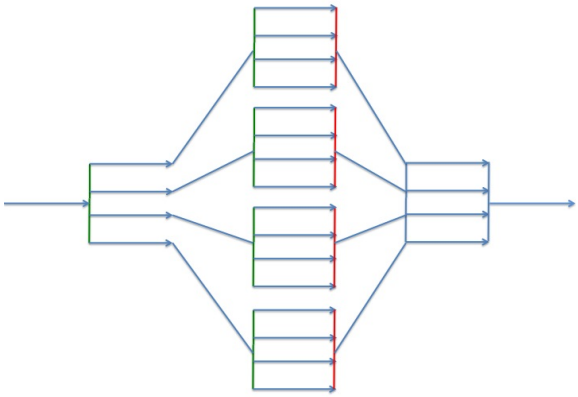
The Motivation for Thread Affinity Policies in OpenMP

The OpenMP specification that describes how to launch software threads identified by the programmer in a particular application. In contrast, hardware threads are usually managed by the underlying operations system or processor. For example, a given set of software threads in user space could be scheduled in any particular way but the OS may bind those threads according to its own policy. One common policy called "first touch policy", is for the OS to bind a thread to the first processor on which it first allocates memory. Hardware thread implementations, such as hyper-threaded processors, will in turn, have their own thread-scheduling models. The pattern of thread binding according to this scenario may not be efficient for high-performance applications: Threads could eventually end up far from their data that they must access, because data locations can change in the course of an application.

To achieve optimum performance in OpenMP applications, the assignment of threads to processors must be able to adapt to both changes in data location, and features of the underlying hardware, which usually does not have a flat computational or memory topology. In OpenMP this is achieved by giving the programmer control over the binding of threads to processors. "Thread Affinity" is a term used to describe the process of binding a thread to a specific processor (or set of processors).



(a) Fork-join model for non-nested parallel structure.



(b) In nested parallel sections the number of threads can increase exponentially. This figure shows only two generations.

Figure 2.1: Illustration of the fork-join model in OpenMP: Time proceeds horizontally (not to scale). The horizontal blue arrows represent thread execution, and each arrow in a vertical stack represents a thread. The master thread launches a thread team upon encountering a parallel region (marked by a compiler directive). The master thread continues processing throughout the parallel region as on of the threads in the team. Thread creation and destruction occur at the green and red vertical line respectively (although many implementations will maintain thread pools to lower thread creation overhead). There are implicit synchronization barriers at the thread creation and destruction points. In pre-OpenMP 4.0 implementations there is no guarantee that the threads will run on any specific set of processors or that they will not be moved to another processor at any point along the path of execution.

Implementation Details of Thread Affinity in OpenMP 4.0

OpenMP features a rich variety of directives and constructs that specify the details of thread creation and processor affinity. These include an environment variable that specifies

groupings of processors, and clauses that specify binding patterns. These are summarized below:

Processor groupings can be specified as either *threads*, *cores*, or *sockets*. In OpenMP terminology, a *thread* is the most fine-grained definition of a processor corresponding to a single hyperthread on a single CPU. A *core* is a CPU. A *socket* is a grouping of CPUs. Explicit groupings can be specified by setting an environment variable (OMP_PLACES). The mapping of threads to cores and sockets, as well as the numbering of processors, is implementation defined.

The mapping of software threads to processor groups (once the groups have been specified) is set using the parameters “master”, “close”, and “spread”. These determine where the threads are placed among the various processor groups. The “master” policy will run all threads on the same processor that the master thread runs on. The “close” and “spread” policies are more distributed (as the names imply). Additional policies are enacted when an outer parallel region encounters another parallel region in the case of nested parallelism. More details on these policies can be found in the test implementation section.

The specification also states that once a thread is assigned to a processor it cannot be transferred to another by the OpenMP runtime, or OS.

2.0.6 Message Passing Interface (MPI)

In contrast to OpenMP, MPI was initially designed for use on distributed memory systems and requires programmers to incorporate explicit parallelism into their code. Instead of using threads, MPI starts multiple concurrent processes (identified by “ranks”), each with their own address space, that communicate via explicit message passing. Like OpenMP, MPI is a standard, not an implementation. For any particular implementation, the standard specifies what it must supply and the semantics it must follow.

Overview of Remote Memory Access (RMA) in MPI

Remote Memory Access (RMA or "one-sided communication") is separate from what is known as "Remote Direct Memory Access" (RDMA) but related to it: RDMA is way of allowing direct access of remote memory locations while bypassing the CPU and OS. In some implementations of RDMA, hardware used for memory access is also offloaded onto the NIC, enabling an even more efficient means of pipelining memory access to allow for more efficient computation/communication overlap. RMA operations in MPI, on the other hand, were developed as means to take advantage of these hardware modifications. (Earlier MPI Standards that only included 2-sided communications were a poor match for RDMA since both origin and target processes had to explicitly be involved in communication, in effect nullfiying the advantages of RDMA.) All RMA operations in MPI-3 are non-blocking and can be used effectively in situations where all of the transfer parameters are available to one process beforehand, so that additional synchronization and broadcasts of parameters are not necessary before transfer (which would completely obliterate any performance advantages). The ability to initiate several message transfers and complete them later, while doing other work, reduces effective latency. In addition, if RDMA is implemented in the hardware, message transfers can proceed without tying up the CPU while computation is being performed.

There are three main different methods of exposing memory to other processes and two different synchronization methods, some with multiple choices of synchronization functions depending on how fine-grained control is desired. Which of these is most appropriate for a given application will depend on the capabilities of the hardware, the pattern and changing-nature of data access, and the tradeoffs in performance vs memory consistency. A review of the various memory window creation and synchronization methods that were tested is provided in Appendix A.

Non-Blocking Collective Operations in MPI

Collective operations in MPI are broadcast and scatter/gather operations that involve all processes in a communicator. For example, MPI_IBcast distributes the value a one process to all other processes. MPI_IScatter divides the data at one process into equal size chunks and distributes them amongst all other processes in a communicator. MPI_IGather does the inverse operation of MPI_IScatter: It gathers data distributed amongst processes into one location at the root process. These are just a few examples of the many collective operations available in MPI. These operations are useful because it is difficult and awkward to mimic the same behavior using simple point-to-point communication. specialized functions that do the same thing are simpler to use in the code and can be implemented more efficiently. Collective operations have been a part of MPI before the MPI 3.0 release. New to MPI 3.0 are non-blocking collective operations, which were added to reduce communication latency.

Thread Safety in MPI

MPI provides 3 levels of thread support:
MPI_THREAD_SINGLE: Only a single thread runs in the entire application. Obviously, thread safety is not an issue at this thread support level.
MPI_THREAD_FUNNELED: Multiple threads run throughout the program, however only one thread makes MPI calls. This eliminates the need for thread safety withing MPI functions, however thread safety of the application as a whole may have to be addressed.
MPI_THREAD_MULTIPLE: This is the thread support level that requires thread safety testing. Multiple threads run both throughout the application as a whole, as well as in MPI functions. This is the level that was tested for this project.

Gropp and Thakur [7] provide a convenient classification of thread-safety issues that can

arise in MPI functions. to summarize the most salient points for this thesis, a subset of these are:

Access to the same object or data structure: Multiple MPI functions may need to allocate or deallocate the same data structure. One example would be a read-only property such as a function that returns the number of MPI ranks. In this case, race conditions could arise if one process frees an object that is then needed by another. Another is reference counts: Thread conflicts in this case could lead to erroneous results if multiple processes accessed the count variable in non-thread-safe ways.

I/O System: Multiple threads in MPI functions access the communication or I/O system. This will be the case for remote memory access (RMA) functions such as MPI_Put or MPI_Get. Since a substantial portion of the MPI thread safety tests developed for this thesis involve RMA operations, this form of thread safety will be the major factor for the tests.

Allocation of Objects or Memory: MPI functions allocate memory using malloc, or other objects such as windows or handles. MPI RMA operations allocate memory windows as well as communicator and request handles. If these functions are not designed for thread-safety, one thread may use memory already de-allocated by another, or access to the memory may overlap in undesired ways. In the case of handles, one handle may be accessed by more than one thread, causing confusion in data destinations or synchronization operations.

Chapter 3

Testing Models and Strategies

The entire software development and testing of the Cray OpenMP and MPI implementations is comprised of three levels: 1) The application of a general software development model used for large scale projects: *The General V-Model*. This stage also defines the software testing goals and priorities, so it informs other models and strategies used in the project. 2) Development of a set of strategies that the UNM test team uses for the testing of the OpenMP and MPI standards: *Dynamic Analysis; Regression Testing; Functional Testing; Verification* 3) Implementation of the specific test strategy used for the test cases that are the topic of this thesis: *Black-Box Testing*. These three levels provide a good framework for explaining test models and strategies, and help to explain where this project fits into a larger picture that involved Cray and a team of test developers at UNM. This chapter will present details the models used at the levels, and also analyze some of their weaknesses.

3.1 Level 1: Software Development and Testing Models for Large Scale Projects

There are two main models of software development: the "Waterfall Model" and the "General V Model" [1]. I call these software development models because they prescribe ways that testing can be integrated into a software development project as a whole (not just specific testing phases). In the Waterfall Model test planning and execution proceed after all of the other development stages have been completed. In the V-Model, test design and implementation is synchronized with various development stages and the two branches feedback on one another.

The development stages are [1]: Requirements Definition, Functional System Design, Technical System Design, Component System Design and Programming. In the V-Model, they are completed along side of the test planning for each stage. Then, after the programming stage, testing is done as the projects proceeds back up the V (See Fig. 3-1 of [1]).

In most cases the General V-Model is the most desirable approach, however not all test environments, software designs, and project management considerations allow it to be implemented in detail. The spirit of the approach is to not leave all test planning and implementation until the step just before a new software project is released to a customer.

I describe the software development models here, because they can be used to see where the UNM testing team fit into the Cray OpenMP and MPI development project as a whole. Whether the Waterfall or the V-Model was used, the best place the UNM testing team would fit into the software development process is right after the Programming Stage (in the Waterfall model this would also be after all coding has been completed). This stage (which is called "Component Testing" in the V-Model) is where actual code is tested to verify its functionality. Verification of code, to ensure that it meets functional requirements

is the domain of "Black Box testing". Black Box testing itself, is subsumed into a more general test technique called "Dynamic Testing". Both Black Box and Dynamic Testing are techniques specifically used by the UNM testing team so they will be described in the following sections.

However, this still does not completely describe the situation: The Cray OpenMP and MPI libraries and compilers are in a process of continual development and are either being optimized, or changed to meet new specifications that are released on a regular basis. So regardless of when they were originally developed, they came to us as already developed entities that require software maintenance as they are continually updated. Also, our test suites had to be automated and available for re-use to check that one modification did not have unintentional side-effects on another feature that had already been tested in a previous release cycle. These testing activities are called "Software Maintenance" and "Regression Testing".

So far, I have mentioned "Dynamic Testing", "Verification", "Black Box Testing", "Functional Testing", "Software Maintenance", and "Regression Testing" as techniques that were used by the UNM Testing Team to verify the Cray implementationsof OpenMP and MPI. These terms are briefly summarized below:

Dynamic Analysis: Testing that is done with code actually running on a computer, comparing the output for a given input is *dynamic*. The alternative, *Static* testing, is done by analysis of the code when not running.

Verification Testing: This includes testing to verify compliance to a specification. The specification relevant to this project is the OpenMP 4.0 and MPI 3.0 specifications.

Black-Box Testing: A subset of Dynamic Analysis, Black-Box Testing views the code as black box. The test is defined in terms of input/output behavior, not on the underlying code or structure, which cannot be viewed by the tester.

Functional Testing: This is to test that a program complies to specifically defined input/output behavior. This behavior is described in the relevant specification document, which constitutes the *requirements definition* for the tests. By contrast, non-functional testing would pertain to performance, or quality. We did not do non-functional tests for this project.

Software Maintenance. Unlike mechanical systems, software does not gradually degrade, so "maintenance" refers to the gradual process of uncovering bugs, or testing additional functionality. In particular, any upgrades or added functionality must not break the existing code.

Regression Testing: Testing that is performed throughout the software life-cycle. It can also be intended to be sure new features do not effect older ones that have already been tested in past releases.

3.2 Level 2: Testing by of the UNM Cray Compilers by the UNM Testing Team

As mentioned in the previous section, Dynamic Analysis and Black Box testing were the main techniques used to do functional verification of the Cray OpenMP and MPI implementations after the programming stage. In this section I will go through Dynamic Analysis in more detail, because it relates to how the test planning was done at UNM Test Team level. Although Black Box Testing also pertains to this stage, it is more relevant to the design and implementation of individual tests, which is more relevant to Level 3, described in the following section.

3.2.1 Dynamic Analysis

Description of Dynamic Analysis

Dynamic analysis is the process of testing code by executing it on a computer. Therefore it requires data to be executed, a method of knowing the correct result of execution for that data, and a test-bed for the execution. For this project, the test bed consisted of a test harness as well as the Cray computing environment and compilers themselves. Data is related to the selection of test cases and was specific to each test. This will be described for the specific test case of thread affinity later on. In general, numerical input data was randomly generated. Other data could be the selection of environment variables, configuration and parameters of the test. The correct result was determined from interpretation of the specifications and translations of that interpretation into code.

3.2.2 Dynamic Analysis Weaknesses

In Dynamic Analysis, the correctness of the tests depends on the complex interplay of many components, each of which themselves and can be in error. For example, the test bed must compile, link, and run the code correctly and set the configuration in a known and reliable manner. The computing environment also presents a unique infrastructure and computer architecture that can change the results and therefore must be taken into account. The test bed may not be inherently stable: the system can be updated, environment variables changed, and compiler version changed on a whim. In addition, interpretation and translation of specifications brings in the possibility of human error: mis-reading and coding mistakes.

Given the inherent complexity of Dynamic Analysis, verification of the tests themselves becomes more important. One can imagine that both false positives and false negatives are possible. Presumably software testing engineers will be astute enough to evaluate

their own software (ironically one of the major pitfalls mentioned in most software testing texts). Ideally, a third party or additional method would be used for verification of the tests and testing process. For this project, test results and code reviews were done by test managers at both UNM and Cray. Also, the structure of the tests and test cases themselves made the possibility of a false negative (here meaning that the tests evaluates faulty software as correct) improbable because of the large number of configurations of the output data, each individually checked. However, the lack of systematic verification of the tests, driven by cost and schedule constraints, remains one of the more significant weaknesses of the method as it was applied to this project.

3.3 Level 3: Black Box Testing OpenMP Thread Affinity and MPI Thread Safety

3.3.1 Application of Software Testing Strategies to the Test Development

As part of the UNM testing team I was assigned the responsibility of writing test cases for OpenMP 4.0 and MPI 3.0 Thread Safety. My tools were the test bed, specification documents, the Cray computing environment, and a Linux computer. The method was Black Box Testing and the goals of the tests were functional verification of the Cray OpenMP and MPI implementations.

The OpenMP project had a traceability matrix (a document that keeps track of the relation between the test cases and the specification written by the test manager) from which I picked the "proc_bind" compiler directive to work on. This is the command that specifies processor affinity in OpenMP. My task was to write the required test cases to achieve comprehensive coverage of the various ways that processor binding can be speci-

fied in OpenMP 4.0 for the C, C++, and Fortran compilers. This required me to examine and understand the specification, determine a set of inputs to the tests, find out from the specification the expected output for a given set of inputs, and translate it all into code. The input data for the proc_bind directive consists of a set of environment variables and clauses that specify the method of processor binding. I initially exercised the compiler with a wide variety of input settings to try to tease out any faults or misinterpretations of the specification. Eventually the test cases were narrowed down to a few commonly used configurations and ultimately integrated into the test harness that ran the full suite of OpenMP 4.0 tests developed by the UNM team.

The MPI Thread Safety tests used the same test bed and tools as the OpenMP tests. They also used Black Box testing with the goal of functional verification. Again, I had to understand the MPI 3.0 specification. However the test themselves were packaged as a set of files that were forwarded to the developer for use in development efforts. The tests were validated using the MPICH library on the Cray systems (whichever one was current) and reviewed by the developer for correctness and suitability.

The following sections will describe Black Box Testing, the main method that I used for developing the tests.

3.3.2 Black Box Testing

Black Box Testing Description

In Black Box testing, the code being tested is not visible at all to the tester, and appears as a "Black Box". The code is evaluated using known inputs and pre-conditions. The result of the test is the comparison of the actual output of the test with the expected output for the given input. The expected output is derived from specifications and requirements. As such it is similar to Dynamic Analysis (as described above) except that it exclude testing

where the code is available (known as "White Box Testing").

Black Box Testing Weaknesses

Black Box testing can efficiently verify that code gives output consistent with a specification, but it cannot evaluate the suitability of those specifications themselves. Also, extra functionality in the component being tested will not be discovered by the tester [1].

3.4 Summary

In this section I reviewed the various test models and strategies used in the project. I started by describing the over-arching testing models used in large software development efforts. Then I discussed test models, planning, and tools used by the UNM team to implement Dynamic Analysis. Finally, I provided details of the writing of individual tests using Black Box Testing. The goals of the tests were functional verification. The tests themselves were validated by reviewing the code, and analyzing the behavior of the tests themselves using the Cray libraries while running the test cases.

Chapter 4

OpenMP 4.0 Thread Affinity Tests

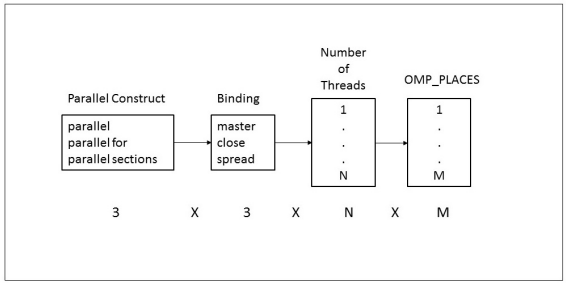
This chapter provides details the design, planning, implementation, and results of the tests. The designs for both the the OpenMP and MPI tests are explained in terms of the Black Box Testing Model.

4.1 OpenMP 4.0 Processor Binding Test Planning and Design

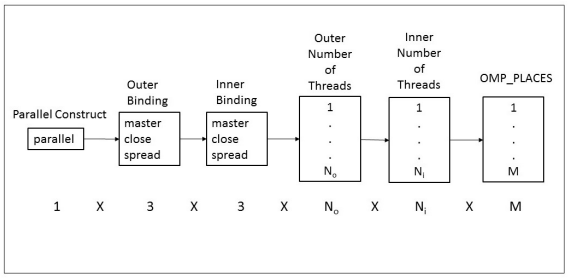
4.1.1 Design of the Black Box Testing of OpenMP Thread Affinity

Input Data to the Black Box Testing Model

Recall that in *black box testing* the code under is test is not available to the tester. Instead the behavior of the code is analyzed by providing a set of input data, and comparing the resulting output data after it has been passed through the "black box". The expected output data for a given input is determined from the specification. The set of all possible input data for the OpenMP Thread Affinity tests is shown in Fig. 4.1. Two diagrams are shown:



(a) Input for one level of parallelism.



(b) Input for 2 levels of nested parallelism.

Figure 4.1: Input data for black box testing of OpenMP proc_bind for one level of parallelism (top) and two levels of nested parallelism (bottom). The number of choices for each input category is shown in the line below each figure, and are multiplied to get the total number of configurations.

one for the cases where there was just one level of parallelism; the other is for the more complicated case of "nested parallelism" where threads are called in the outer level, and then each spawn a number of child threads to work on an inner level. (See Chapter 2 and the fork-join model for details on nested parallelism.) Here I explain the input variables for the Fig. 4.1 in detail:

Parallel Construct: OpenMP offers many ways to divide up work between threads in a parallel region. The constructs `parallel`, `parallel for`, and `parallel section` are a subset of those. In `parallel`, every thread executes every instruction in the parallel region. In `parallel for` the threads are divided between iterations of a for loop. In `parallel section`, each thread executes a separate section delineated by the programmer. The original traceability matrix for the OpenMP 4.0 test suite had

entries for all parallel constructs, but only these three constructs were chosen because the remaining construct (`parallel workshare`) has a very restricted set of function types that are allowed in the parallel region, precluding the system calls and array assignments that were necessary to obtain the thread affinity. For the nested cases, only one construct (`parallel`) was used to avoid redundancy.

Binding: This clause specifies exactly how the threads are bound to the processors. Exactly how it works depends on the setting of the OMP_PLACES environment variable (explained below). The specification for the *master* clause states that all the child threads will be bound to the master thread. The *close* policy places the threads in sequential order starting at the master place, using wrap-around. The *spread* policy attempts to spread the threads out as much as possible between the different processors. The OMP_PLACES variable for the master and close cases does not change for the master and close policies when encountering another level of nested parallelism. However, the spread policy requires a re-computation of the OMP_PLACES variable which will then be different for each child thread, which is relevant for nested cases. See the OpenMP 4.0 specification for more details on these policies [3].

Number of Threads (outer and inner): This is just total the number of threads (for non-nested) and the number of threads in the outer and inner loops (for nested). The number of threads can greatly effect running time. Also, it has a large impact when combined with the OMP_PLACES variable: Some values if the number of threads may divide evenly into the number of processors, others not.

OMP_PLACES Environment Variable: Several test cases can pertain to the ability of the compiler to parse the OMP_PLACES variable correctly. The environment variable, OMP_PLACES specifies the order and grouping of processors (see Fig.4.2). There is more than one way to set the OMP_PLACES variable. For example, the syntax: OMP_PLACES=

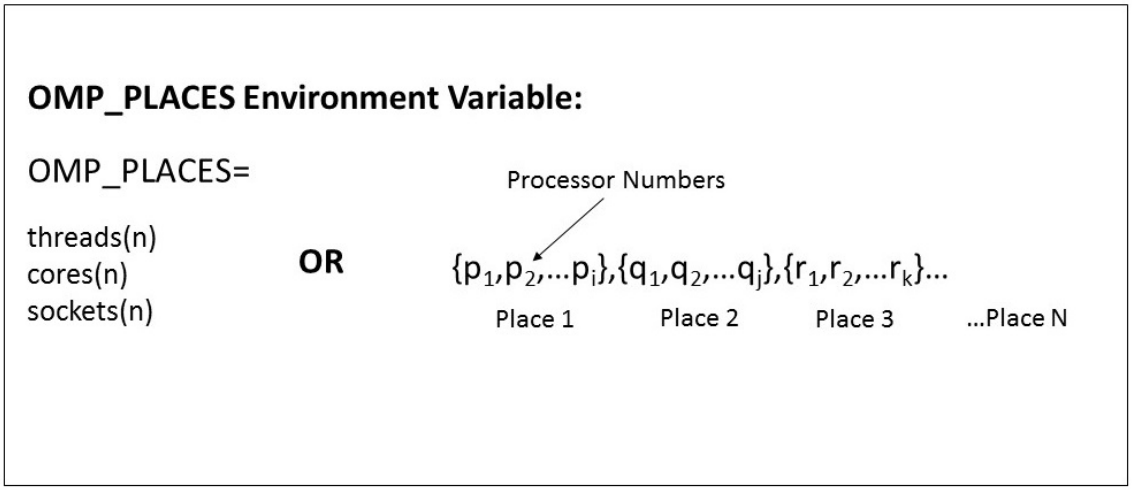


Figure 4.2: Ways of setting the OMP_PLACES environment variable.

$\{1, 2\}, \{3, 4\}$ specifies that the first "place" is processors 1 and 2 (a thread bound to this place can run on either processor), and the second place contains processors 3 and 4. It is important to note that threads are bound to places, not processors. Another, equally correct syntax, is OMP_PLACES=threads(n), OMP_PLACES=cores(n), or OMP_PLACES=sockets(n). These correspond to implementation-defined processor groupings (as outlined in Section 4.2). The tests must be able to parse this environment variable, compute the expected result of the setting and compare that with results for each of the other runs.

All of the ways of combining the number of threads, the OMP_PLACES Environment variable, and the processor architecture would create many more test cases than can be run in a reasonable amount a time. A more organized, systematic way of choosing the most relevant test cases would entail division of this space into equivalence classes, ideally selecting one case from each class [1]. In fact, an intuitive understanding of this motivated an initial exploratory selection of test cases shown in 4.1. This list was not used in the final test suite because it took over 2 hours to run which was not practical or acceptable to Cray. But the initial exploration turned up issues that are describe in Section 4.2. The final set of test cases used in the suite included common use cases. In the end, I added a feature

allowing the user to chose exactly which test cases to run, so the program is flexible and new test cases can be added as needed.

num_threads/num_places = k	num_threads/num_places != k
num_places/num_threads = k	num_places/num_threads != k
repeat processor numbers in a place	no repeats of processor numbers
non-consecutive processor numbers	consecutive processor numbers
processors appear in different places	processors only appear in one place
threads $>$ places	threads $<$ places

Table 4.1: **Some Possible Test Cases for OpenMP proc_bind:** An attempt to divide a portion of the test case matrix into equivalence classes. k is an integer. The program can set any of these configurations through the setting of an environment variable. These initial test cases were used for a smoke test to and helped uncover miss-interpretations of the specification.

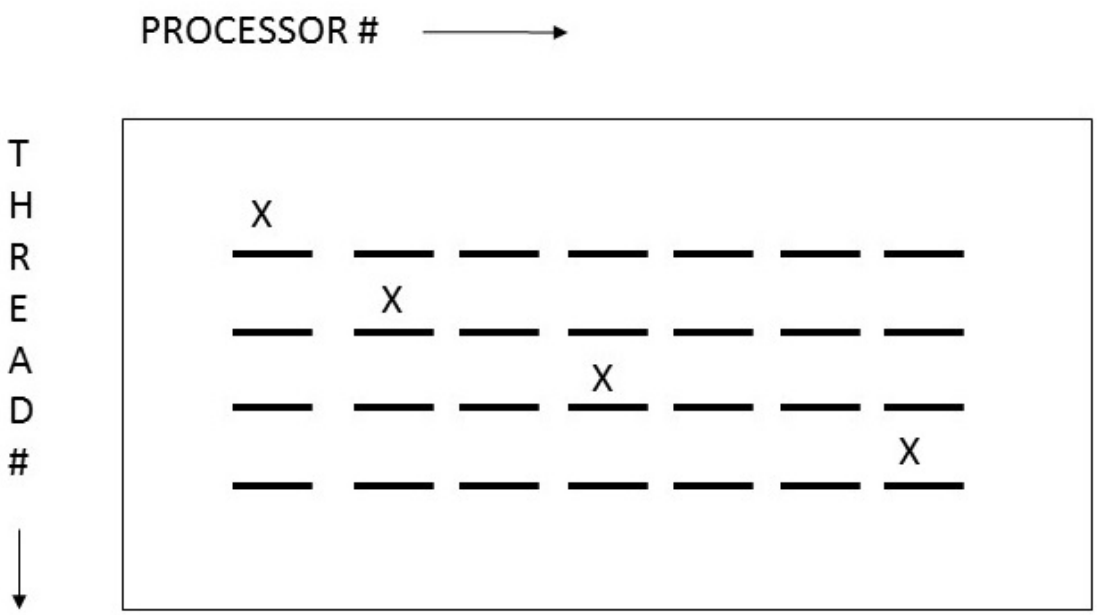


Figure 4.3: A possible thread binding output for a black box test of proc_bind in OpenMP.

Output Data to the Black Box Testing Model

The output data for the Black Box Testing was the binding of the threads to the processors. This took the form a of a matrix. Fig. 4.3 is an illustration of a thread binding scenario that could form the output of the test. Eactly how these bindings were obtained is discussed in the next section on implementation.

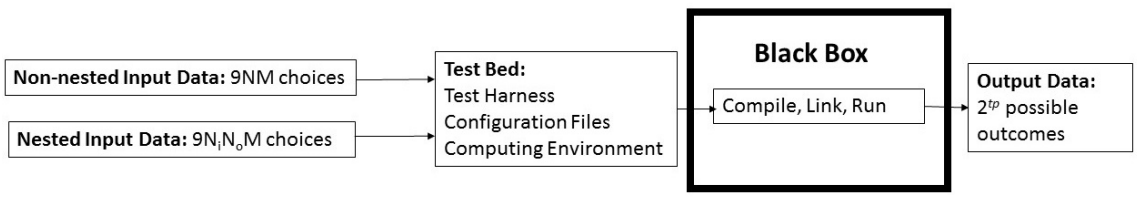


Figure 4.4: The Black Box Test Design for OpenMP Thread Affinity showing the input data, test bed, system under test (the black box), and output data.

Overall Black Box Design

The Black Box Testing paradigm and the input and output data outlined in the previous sections combine to make the overall test design of the thread affinity test as shown in Fig. 4.4.

4.1.2 Software Design Principles

The complexity the design of the thread affinity tests necessitated the adherence to sound software engineering principles when designing how the code for the test itself should be written. The code was designed around the principles of *portability*, *modularity*, and *code*

re-use. Specific of these how these three design strategies were implemented are outlined in the next chapter.

4.1.3 Test Requirements and Plan

The goal of these tests was to verify the Cray Compiler implementation of the OpenMP 4.0 specification which was released in July, 2013 [3]. This release contained major changes from the previous OpenMP 3.1 release (July 2011). Because the test requirement is to verify performance to a specification, the test plan consisted of a document that outlines which OpenMP constructs and clauses differed from the previous 2.5 specification. The task of the testing team was to test only these features.

4.1.4 Test Infrastructure

The tests themselves consisted of a set of routines in C, C++, and Fortran. Since there was a test team (each working on a different OpenMP 4.0 feature) a local svn repository was setup for version control. An additional svn repository was located at the customer site. All members of the test team had access to development systems at Cray through ssh, and local linux-based machine to use for test development.

The entire suite of tests for the OpenMP 4.0 release was run on a test-harness that could run the entire suite, or any selection of tests which were specified through as script file. A complete test to be run on the harness, consisted of the source code, as well as configuration files that set environment variables.

4.2 OpenMP Test Implementation

Here I discuss detail of the OpenMP Thread Affinity test implementation. First I present an overview and some of the challenges which motivated the software engineering strategies described in the following section: *Modularity, Code Re-Use, and Portability*. Following these sections, implementation details are presented. The last section describes some more specific challenges that arose during the implementation, specifically ambiguities resulting from implementation-defined portions of the specification.

4.2.1 OpenMP Thread Affinity Test Development Challenges

The OpenMP 4.0 thread affinity tests are inherently complex in that they have to take into account many different thread affinity scenarios. In addition, some have implementation-defined results in which case the test designer must be able to either 1) ascertain exactly how the developer has implemented it 2) design test cases that cover all possible implementations consistent with the specification. For this project, Cray wanted the latter so as to be able to have the option to re-use the same tests in the event that the preferred implementation was changed, either in future revisions or for running on different architectures. These relaxed requirements required the tests to be more general and cover more cases than would be necessary for a more rigid specification. So even though I was testing only one feature of OpenMP 4.0, the variety of results that I had to take into account, combined with the information about the thread-affinity for many different ways of specifying processor binding in the language resulted in one longer program with several modules and sub-functions. The challenge in writing this tests was in overall program design and software re-use as the test became increasingly complex an unwieldy without careful attention to these software engineering principles.

4.2.2 Implementation of Software Design Principles

Modularity and Code Re-Use

Modularity is a feature of all well-written code. The main way to realize modularity is to structure the code so that one function accomplishes one well-defined task, and to limit the number of returned values. I adhered to this whenever possible and greatly assisted debugging, made it simple to make changes, and assisted in code re-use. Utility functions that might be used by more than one program were combined into Fortran modules. Functions that made system calls were written in C, and put into a single C file which was used by all 3 compilers (C, C++, and Fortran).

Another strategy used in this project was to separate the routines that used OpenMP directives to run the test cases from the code that was used to setup the tests, evaluate the results, and print output. This made it possible to use the Fortran code written for these functions in both the C and Fortran implementations (for logistical reasons the Fortran code was developed first). To implement this, bindings for C and Fortran interoperability were used. This was not done for the C++ version because I was not able to get the Fortran bindings to work properly for C++. This worked out fine in the end though, since C++ is the only object-oriented (OO) language of the three, so writing an OO version from scratch was both instructive, and would have allowed the C++ compiler to be tested more rigorously.

Portability

A customer requirement was that the tests should be portable, since there was a plan to run the test suites on different machines, each with different processor topologies as well as machines with either AMD or Intel processors. AMD and Intel number their processors separately, and therefore the smallest unit of a process (a thread) will be numbered differ-

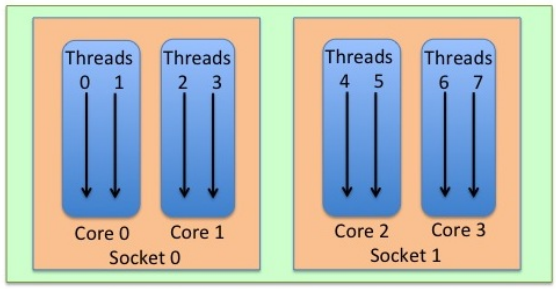
ently in the OMP_PLACES environment variable (Figs. 4.5a and 4.5b). For any general OMP_PLACES test this might not matter, but it *does* matter in the case of tasting the setting threads, cores, and sockets. This is because the tests for these settings were done by converting them to the bracketed notation. To solve this issue, I wrote a C function that parsing the files in "sys/devices/system/cpu/cpu(n)topology/" and sorting the output to get the correct thread numbers for the topology in use.

However, this did not completely solve the portability issue. A hidden requirement was that the test be able to run on ARM processors. These processors expose their architecture differently so the code did not run on them. Unfortunately, no ARM processors were made available to the test team during development.

4.2.3 Implementation Details

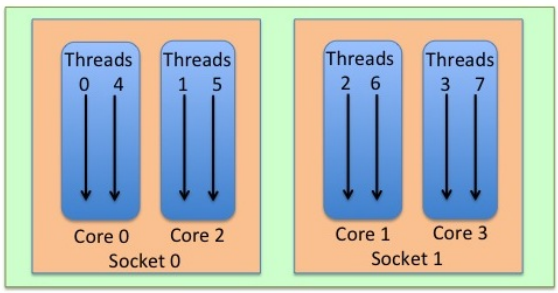
Overall Program Flow

The overall program flow is shown in Fig. 4.6. The first step was to parse environment variables that provided the test parameters as inputs to the program. Then information was extracted through either system calls or files to translate the test parameters to run on the architecture. Next the tests were run using the appropriate OpenMP commands and compiler directives. The computed and actual thread affinity masks obtained and compared. Finally, differences between the computed and actual masks were determined and an analysis of the results provided. The first 2 sections, as well getting the affinity mask (in the third section) were written in C, either because they tended to be closer to the machine and required system calls, or they utilized C parsing functions. All other functions were in C++ or Fortran, with the exception of the step that utilizes OpenMP. Since OpenMP is what is being tested, this section had to have separate routines in all 3 languages.



OMP_PLACES = threads -> {0,1,2,3,4,5,6,7}
OMP_PLACES = cores -> {0,1},{2,3},{4,5},{6,7}
OMP_PLACES = sockets -> {0,1,2,3},{4,5,6,7}

(a) AMD Processor Numbering



OMP_PLACES = threads -> {0,4,1,5,2,6,3,7}
OMP_PLACES = cores -> {0,4},{2,6},{1,5},{3,7}
OMP_PLACES = sockets -> {0,4,1,5},{2,6,3,7}

(b) Intel Processor Numbering

Figure 4.5: AMD and Intel number their processors differently and this information must be extracted from linux system files for a portable implementation of the tests. Below each figure, the OMP_PLACES equivalent bracket notation for thread, cores, and sockets is shown.

Testing of Master, Close, and Spread Policies

After consultation with the test team, it was decided that all three non-nested cases (master, close, and spread), as well as all possible combinations of nested cases for two levels of nested parallelism should be tested (i.e. master/master, master/close, master/spread, close/master, close/spread, etc.). The non-nested cases where the number of threads divided evenly into the number of processors (or vice-versa) were relatively straightforward and is illustrated in Fig. 4.7. This procedure is illustrated for the nested cases (Fig. 4.8) where neither (P/T) or (T/P) has a remainder.

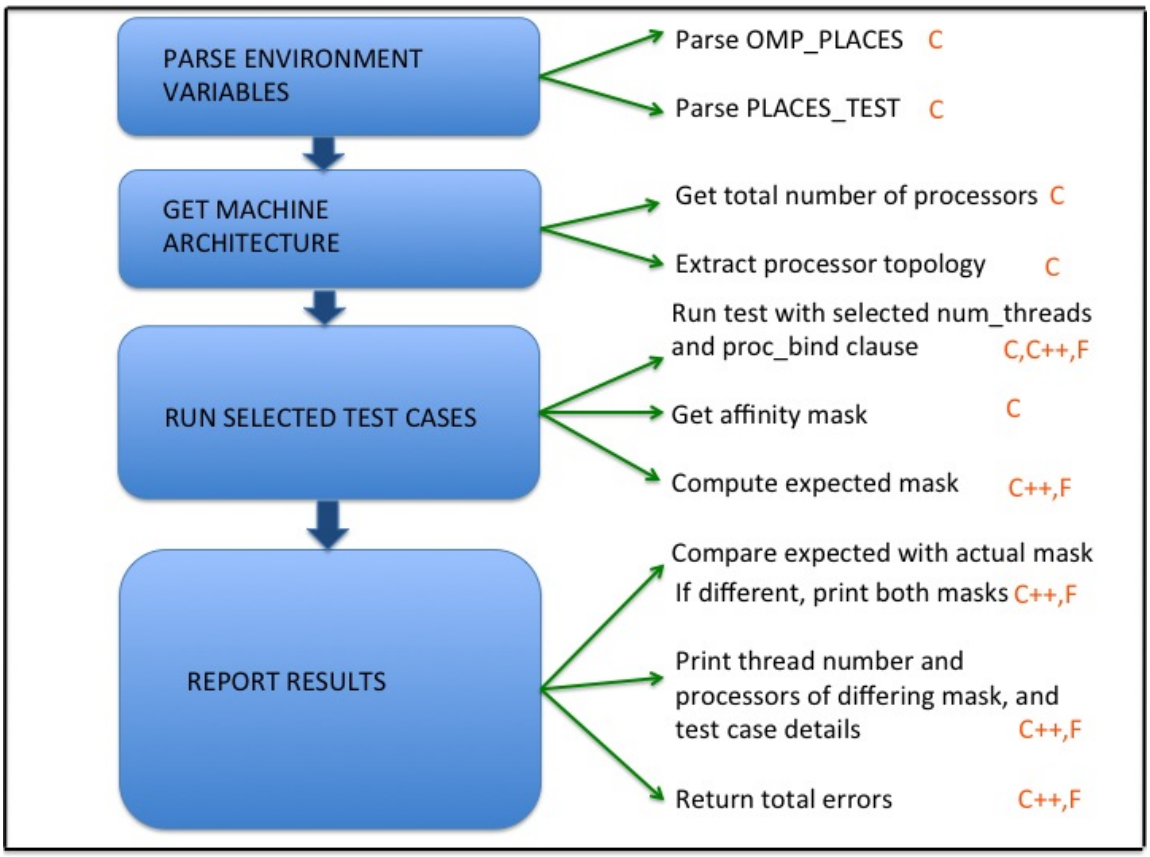


Figure 4.6: The general outline of the program. The major sections are shown on the left, with green arrows pointing to subtasks within that section. The red letters on the right indicate the languages that each section was coded in.

4.2.4 Additional Challenges and Implementation-Defined Features in OpenMP

Abstract Names and Processor Numbering

Some challenges arose in the case of portions of the OpenMP 4.0 specification that were listed as "implementation defined". One set of variables that were implementation-defined is the definition of abstract names like "thread", "cores", and "sockets" in the OMP_PLACES variable, any other abstract names the developer choses to add, and the meaning of the

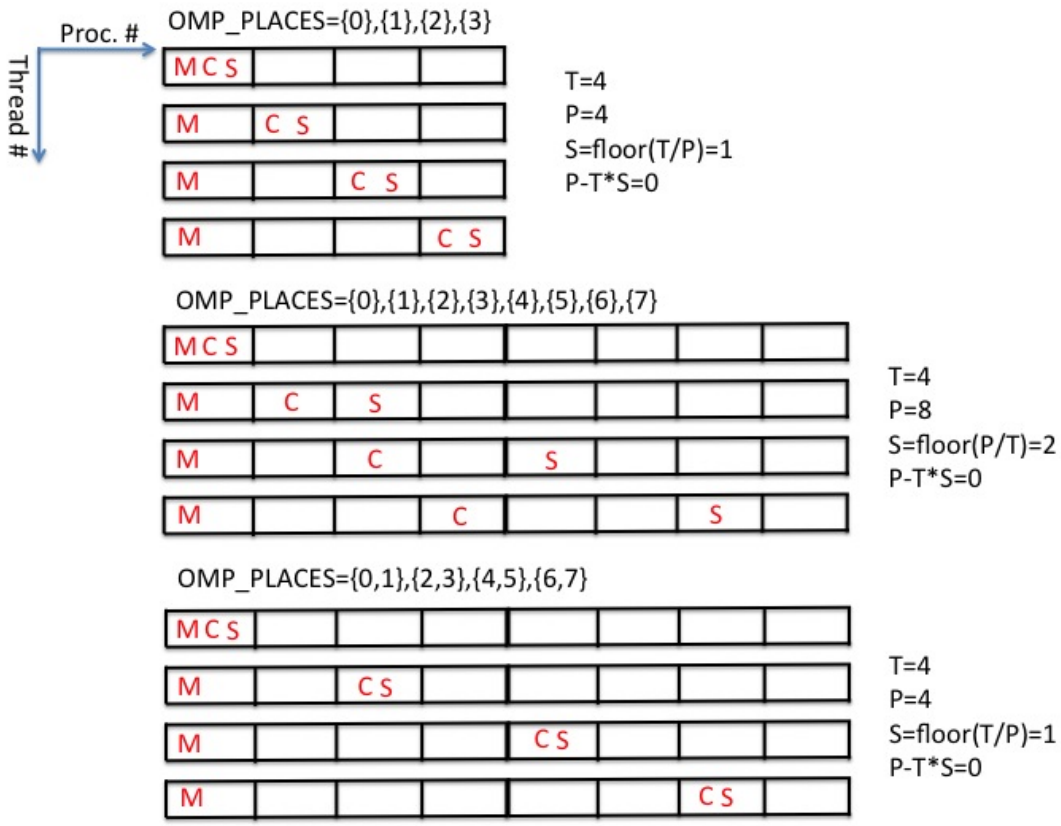


Figure 4.7: Thread affinity mask for single parallel loop cases of master, close, or thread. P is the number of places and T is the number of threads. M,C,S show the binding for the master, close, and spread cases respectively. The case for AMD processors, with sequential thread numbering, is shown for simplicity.

numbers in the comma-separated ordered list (when using that syntax). The compiler developers for this project did not add any additional abstract names. The specific definitions of the threads, cores, and sockets, and numbering schemes were specified by the developer and adhered to, however it did entail obtaining the processor numbering scheme of the architecture, as I described in the previous section.

The Spread Affinity Policy

Another implementation-defined feature is the thread affinity for the spread policy. This arose in cases where the number of threads did not evenly divide into the number of places (or vice-versa). The specification states that the placement of the remaining $P - T * S$ partitions is implementation-defined. In this case, there are two or more configurations that produce maximum separation between threads (see Fig. 4.9). This kind of definition is a little tricky to test because the developer may want to leave the option open to change the implementation in future releases, but want the test to still work without revision. (In fact, this was a specific request from the customer.) The task then, is to figure out a rule that can cover all configurations that are consistent with the specifications. The rule that I chose was to check that fractional places filling was no processor has more than T/P (the number of threads divided by the number of processors). This produces the maximum separation between threads, but is general enough to encompass all such configurations.

The next challenge for this policy was figuring out what to do in the nested case, where an outer spread policy case meets an inner case of any other policy. The problem here, is that when thread adhering to a spread policy encounter another inner nested loop, the specification requires that they each obtain a new set of places (essentially a new, individual OMP_PLACES), derived from the parent set of places. Not only is this also implementation-defined, but I had no way of knowing what the exact affinity scheme was used in the outer case. This was because, although I tested for a general property, it could correspond to more than one affinity scheme, each consistent with the specification. The developer declined a request to provide that information in real time due to performance reasons, so I had to write a routine that deduced the actual affinity that was used according to the rule that a thread is bound to the first place in the OMP_PLACES) partition.

This still left open what to do when one arrives at the last place partition but still has remaining threads. The obvious solution is to use warp-around, (which I did) but it failed

cases according to a mysterious pattern. After communicating with the developer I found out that the policy for this case was variable, and chosen on a case-by-case basis according to no discernible, or readily-described pattern that could be used as a basis for testing. At this point I found to different patterns that were being used and added an extra loop to retest to the second pattern if the first failed.

4.3 OpenMP Tests Runs

Initial testing of the test-codes were done using the GNU C, C++ and Fortran compilers. If the tests passed that step, they were then uploaded to a Cray machine for testing on the actual compilers under test. These compilers were updated nightly, so each logon to a customer machined required loading of the latest compiler. This was necessary because the compilers were in a process of continuous development alongside the testing. Sometimes tests were ready before the compiler had implemented the feature. This illustration of "ad-hoc integration" created a dynamic environment.

4.4 OpenMP Tests: Validation, Results, and Deliverables

4.4.1 Test Validation and Results Reporting

The tests were validated by expert review and static code analysis. Tests that produced unexpected results (if they failed using the most current Cray compiler version) were examined in more detail to detect any possible mistakes in the test itself. If none could be found, communication with the developer was necessary to verify the correct interpretation of the specification. In theory, non-passing tests that had undergone these stages of review would be reported as a bug in the Cray implementation (in practice this did not happen). False positives are harder to detect, and mostly relied on code review and detailed

knowledge of the test itself.

All tests past of the most current version of the Cray OpenMP and MPI compiler and libraries at the time of this writing. However, one goal of the testing was to have detailed error output so that the cause of failures could be clearly and accurately determined. For these tests, the most relevant and useful set of diagnostic parameters for any failing case included; the parallel constructs (both inner or outer), the clause that was used (master, close, or spread), whether it was an inner or outer case of nested parallel construct, which thread was failing, the expected and actual thread affinity for the failing thread, if the failing thread is on an inner nested case, what the parent thread is. The challenge was to format it in a way that the information could be digested and scanned easily. Fig. 4.10 shows the sample output for a test rigged to fail on the 1-node/2-core/4-thread linux system that I mainly used for test development.

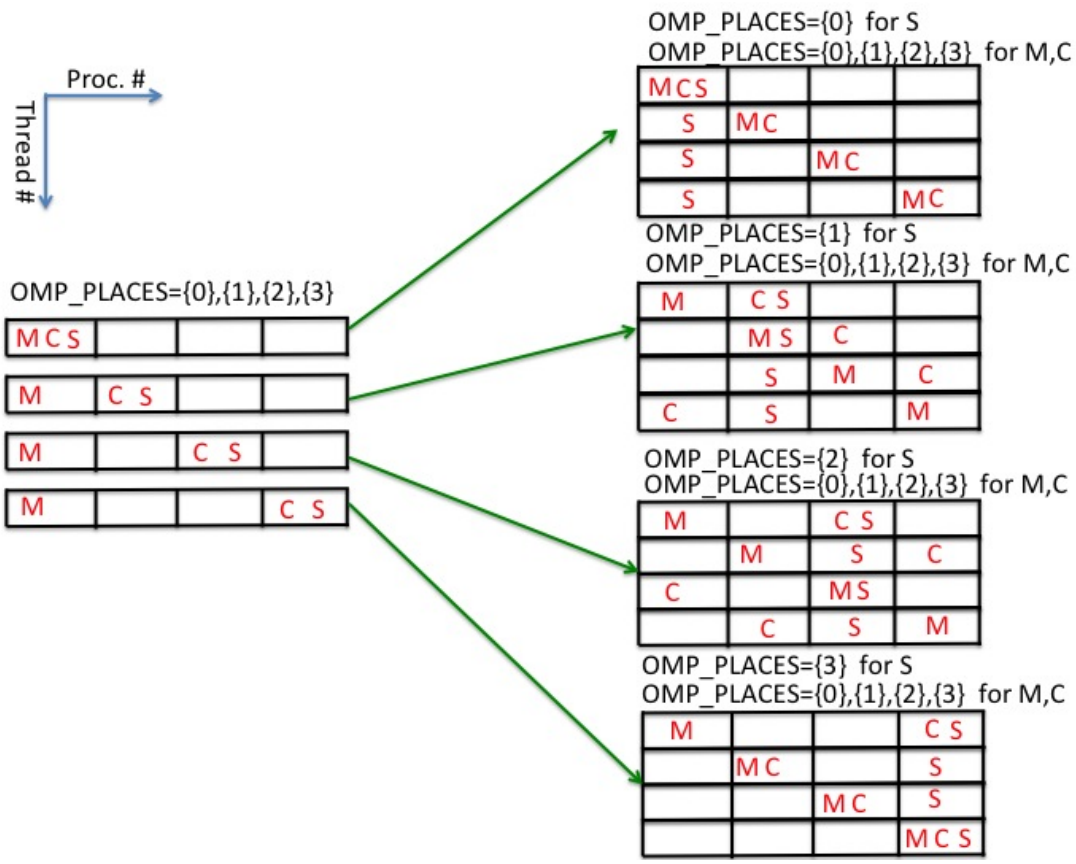


Figure 4.8: Thread affinity mask for an outer parallel loop of master, close, or thread (left). The thread affinities when the outer loop encounters an inner loop marked with proc_bind(close) (right). M,C,S show the binding for the master, close, and spread cases respectively. The green arrows show the mapping of the parent threads to the child threads (this is showing the case for 4 outer loop threads and 4 inner loop child threads). Note that the letters on the right indicate the case for the parent thread (they all use the close policy).

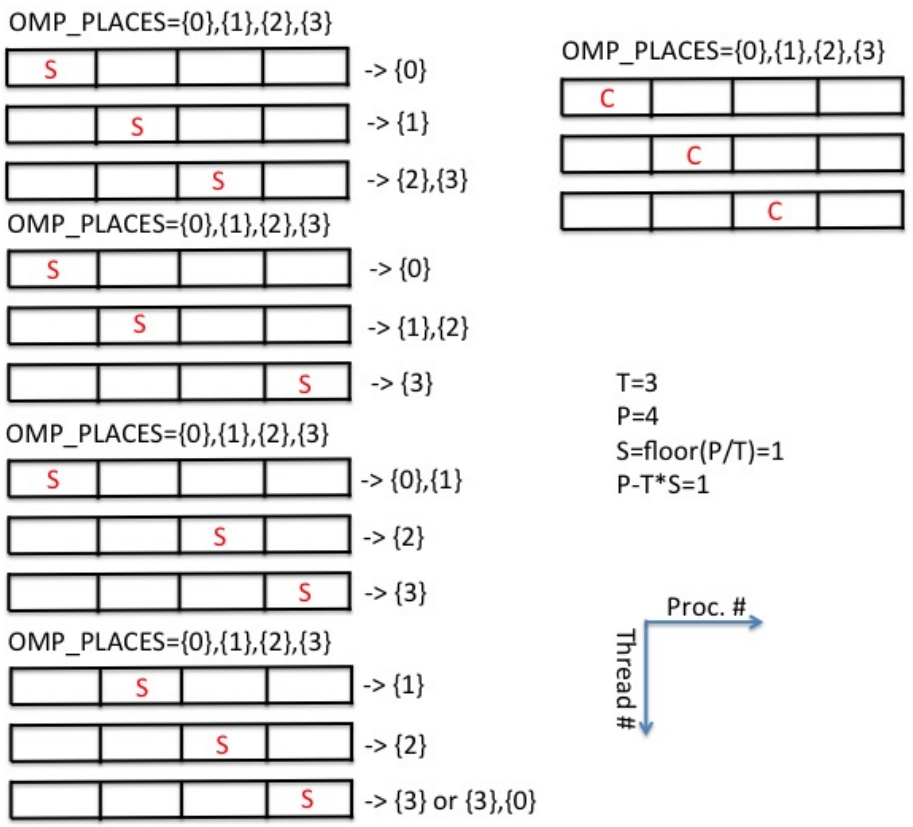


Figure 4.9: Thread affinity mask for the spread policy, where the number of threads does not divide evenly into the number of places. The bindings are implementation-defined in this case, and 4 possibilities for the outer loop are shown on the left, along with the new OMP_PLACES for the child threads. For the last option, there is ambiguity as to how to re-compute the PLACES for the last thread (wrap-around or cut-off). The developer wanted to leave open the option to use either arbitrarily, so each case was tested twice -once for each strategy. The figure on the right shows the non-ambiguous close policy for comparison.

```
parallel_proc_bind_modules.F90:1389 : FAIL - affinity_mask incorrect for omp parallel inner nested case close in master/close 4 outer threads 4
inner threads.
 Actual Mask:
 Thread Number =           0 ; Thread aff =           1          0          0          0
 Thread Number =           1 ; Thread aff =           1          0          0          0
 Thread Number =           2 ; Thread aff =           1          0          0          0
 Thread Number =           3 ; Thread aff =           1          0          0          0
 Computed Mask :
 Thread Number =           0 ; Thread aff =           1          0          0          0
 Thread Number =           1 ; Thread aff =           0          1          0          0
 Thread Number =           2 ; Thread aff =           0          0          1          0
 Thread Number =           3 ; Thread aff =           0          0          0          1
 Outer test was master. Parent thread #           1
 Outer Mask (for reference):
 Thread Number =           0 ; Thread aff =           1          0          0          0
 Thread Number =           1 ; Thread aff =           1          0          0          0
 Thread Number =           2 ; Thread aff =           1          0          0          0
 Thread Number =           3 ; Thread aff =           1          0          0          0
```

Figure 4.10: Error output for a nested test case rigged to fail

Chapter 5

MPI 3.0 Thread Safety Testing

This chapter will follow some of the basic structure as the previous chapter on the OpenMP Thread Affinity Tests: Test Planning, Design, Implementation, and Results will be presented, and will be framed in terms of the Black Box Testing Model. However, the emphasis and organization will be slightly different: First, there will be a section on design challenges and the overall test strategy and less emphasis on software engineering principles. This is because these tests were implemented differently than the OpenMP tests: Instead of one long program that could run a multiple of input configurations for one tests, the MPI thread safety tests consisted of many individual files, written as C programs, each of which tested one function from the specification for either RMA or non-blocking collectives.

The test-bed and validation steps were identical between the OpenMP and MPI tests, so they will not be repeated in here; Refer to the previous chapter for details on the test harness and validation procedure.

5.1 MPI 3.0 Test Planning and Design

5.1.1 Design of the Black Box Testing of MPI Thread Safety

Challenges in MPI Thread Safety Testing Design

In testing thread-safety of hybrid MPI+OpenMP programs both MPI and OpenMP programming, along with the various MPI window creation methods, consistency semantics, and synchronization operations must be understood to be able to write the tests and also determine what is the correct result of a test.

In MPI+OpenMP, OpenMP creates threaded regions for each MPI rank. For each thread a unique handle to the memory window and communicator is necessary. MPI has methods to return handles to memory windows created earlier in the program using `malloc`, but it can also allocate memory windows using MPI functions (see Appendix A). The returned memory regions can be an address space unique to each rank, or shared amongst ranks. Combined with each window creation method are a set of synchronization functions which must be called in a specific sequence for each individual rank and thread. Individual threads cannot be addressed directly in MPI functions, so some ingenuity is required to design test cases that explore typical use cases and are useful to the customer, all within a complex environment that encompasses two levels and types of parallelism simultaneously.

For tests of the non-blocking collectives, these challenges were combined with the necessity of translating collective calls into point-to-point communication calls. This was necessary because Cray did not want to use any MPI calls similar to those being tested when evaluating test results. Similarly, for the RMA testing, methods had to be devised to send the expected result to all ranks for evaluation of the correctness of the tests. Cray did not want RMA calls used for this purpose even outside of the threaded region, so the data

had to either be coded using a mathematical calculation, or sent to the other ranks using point-to-point communication.

For the tests developed for this work the challenges combined: 1) Developing the knowledge and ability to write parallel programs with 2 different models and levels of parallelism 2) Understanding the basic OpenMP and MPI syntax and semantics and how they interact. 2) Understanding Cray requirements and the wording of the MPI 3.0 standard.

Challenges 1) and 2) were a matter of practice, and study. With regard to 3): In the MPI 3.0 Standard itself there is only one section (12.4) that addresses thread-safety and what constitutes a "thread-compliant" implementation:

"All MPI calls are thread-safe, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved."

This is a fairly abstract statement, and any MPI programming operation that could reveal whether this is actually true or not has an undefined result. For example, an obvious thing to try would be to have two threads attempt to write long arrays of integers to the same memory location and see if one or the other of the written values arrives uncorrupted. The designers of the standard had programming flexibility of this kind as a specific goal so concurrent writes to the same memory location are allowed. Unfortunately, the result is undefined, so no test can be designed that uses this operation. This is an example of how relaxed consistency semantics in MPI 3.0 can both add to the flexibility and richness of the programming environment while at the same time make testing goals hard to define.

The weak consistency in MPI, the abstract specification regarding thread-safety in the standard, and the nature of black-box testing made it necessary to work according to a set

of strategies that would challenge thread-safety in a wide variety of contexts and inputs. With this in mind, a set of strategies to be used as guidelines in the tests were developed and included in the test plan:

1) Increasing the length and size of the input for functions that can accept arrays. The idea is that non-atomicity of write operations to memory could cause the input from different threads to interleave in a non-thread safe implementation.

2) Allow for scaling and changes in number of processes and threads where possible.

3) Utilize mechanisms to address individual threads as a parameter in MPI functions in threaded regions whenever possible.

Input Data to Black Box Testing Model for MPI Thread Safety

The input data to the Black Box Model consisted of a data array of length, M, its MPI datatype, the number of ranks, the number of threads, the MPI memory window and synchronization method, and the MPI function under test. The Black Box Model In most cases the data consisted of an array of a length, M, that was specified as a parameter in the program. The output data also consisted of arrays. The structure of both the input and output the arrays depended on the exact operation of the MPI function being tested. Fig.5.1 shows the Black Box design of the RMA thread safety tests and Fig.5.2 shows the input data for the Non-Blocking Collectives.

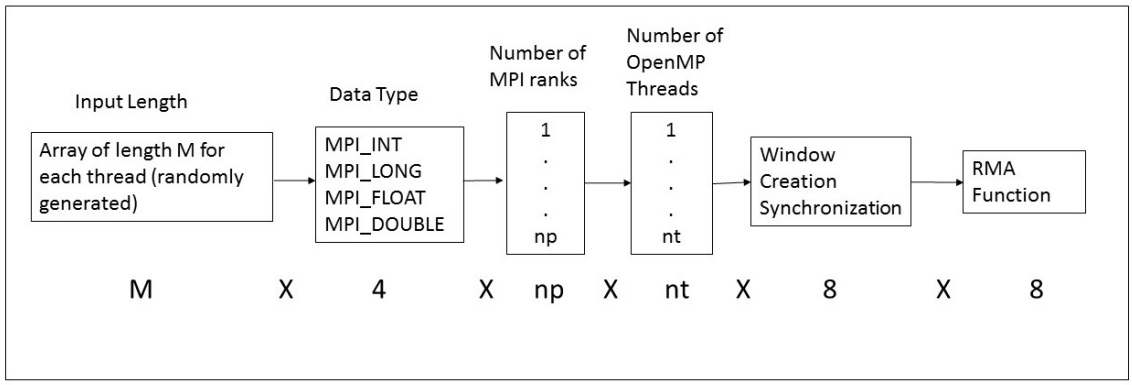


Figure 5.1: Input Data for RMA Black Box Testing: The boxes that list "Window Creation and Synchronization" and "RMA Function" are functions listed in the MPI 3.0 specifications document and are too many to list.The number of choices for each input category is shown in the line below each figure, and are multiplied to get the total number of configurations.

Output Data to the Black Box Testing Model

The output data to the tests consisted of a matrix that contains the values that were provided in the input. Where exactly the output data is placed and the exact form it takes is dependent on the details of the MPI function being tested. The section on implementa-

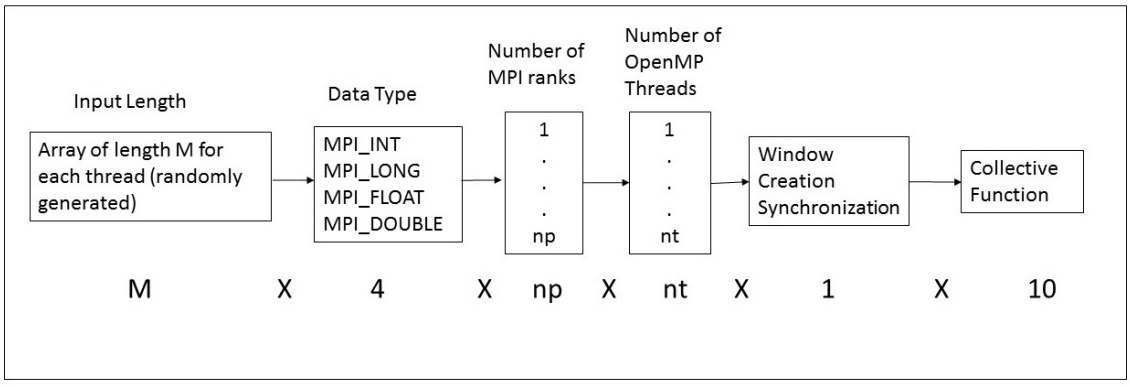


Figure 5.2: Input Data for Black Box Testing of the Non-Blocking Collectives: The boxes that list "Window Creation and Synchronization" and "Collective Function" are functions listed in the MPI 3.0 specifications document and are too many to list, but for these tests only the MPI_Win_allocate() and MPI_Win_fence() were used. The number of choices for each input category is shown in the line below each figure, and are multiplied to get the total number of configurations.

tion shows an example of input and output data matrices for a sample RMA and a sample Non-Blocking Collectives test in Figs. 5.4 and 5.5.

Overall Black Box Design

The Black Box Testing design is presented in Fig. 5.3.

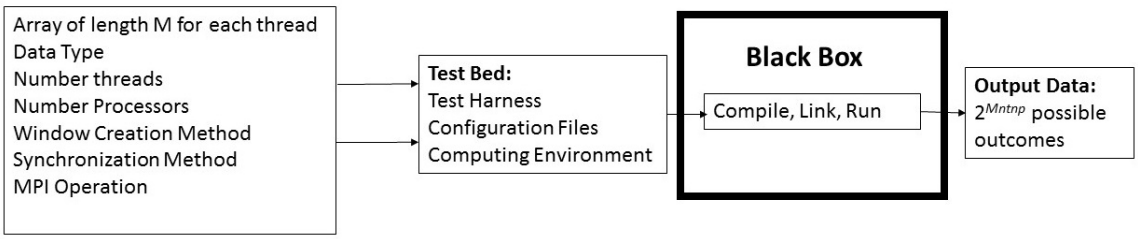


Figure 5.3: The Black Box Test Design for MPI Thread Safety showing the input data, test bed, system under test (the black box), and output data.

5.2 MPI 3.0 Thread Safety Test Suite Implementation

5.2.1 Implementation of the RMA Functions

The MPI 3.0 thread safety test involved the development of a test suite compromising individual test files, each examining a separate one-sided communication or non-blocking collective operation. Each file consisted of exactly one C function and the threaded regions were implemented using OpenMP 4.0 via "pragma omp parallel" directives. The tests used arrays of randomly assigned or calculated values to be passed between processes. Each thread wrote to and from separate portions of the array. The resulting tests were therefore scalable in terms of the length of input and the number of threads used. Fig.5.4 shows the

input and output array designs for a representative RMA function, MPI_Put. The test for this particular function instructed the ranks to send the data array to the next rank (using wrap-around).

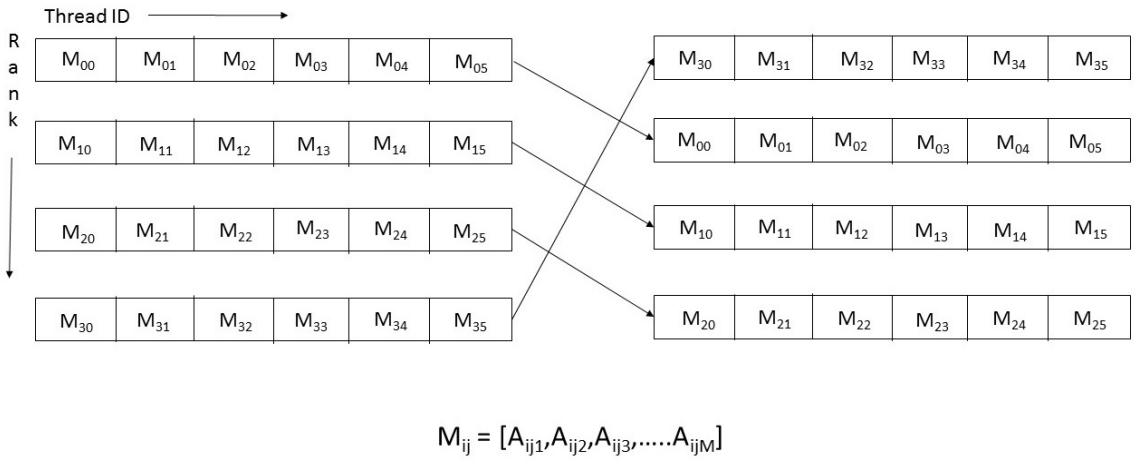


Figure 5.4: Input and Output Matrice for MPI_Put(). The M are arrays of values of the specified MPI datatype.

5.2.2 Implementation of the Non-blocking Collective Functions

Non-blocking collectives in MPI have the performance advantages of non-blocking functions and at the same time provide collective operations that package the process of distributing data to other processes from a root process, or collecting data from other processes. Because they are non-blocking, a request handle is provided at each initiation of each collective operation so the process can check when it has been completed. MPI 3.0 provides many collective operations that enable the programmer to easily implement different patterns of communication that are in common use in HPC applications. Input output data for the MPI_Ialltoall() function test is shown in Fig.5.5.

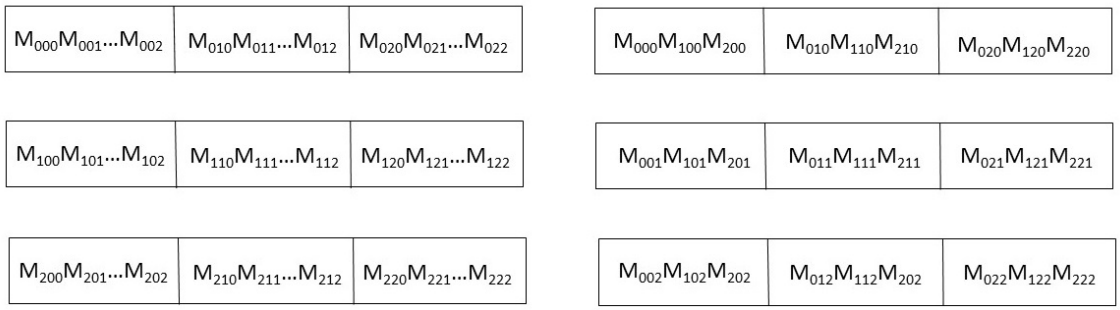


Figure 5.5: Input and Output Matrices for MPI_Ialltoall(). The M are arrays of values of the specified MPI datatype. Here they need to be repeated np times, so that the data can be divided and sent to multiple processors.

5.3 MPI 3.0 Thread Safety Test Results and Deliverables

5.3.1 Test Results and Deliverables for One-Sided Communication

The list below summarizes the tests produced, memory allocation and synchronization method for the RMA tests. These tests were reviewed and accepted by the test manager at Cray and the developer. The Cray implementations are still currently under development, so tests were evaluated on the latest version of the MPICH library. However they are ready, and intended for use on future, optimized implementations that are in the process of being developed.

RMA Tests:

MPI_Put, MPI_Get, MPI_Accumulate, MPI_Get_Accumulate

MPI_Compare_and_swap, MPI_Fetch_and_op

RMA Request Based Tests:

MPI_Rput, MPI_Rget, MPI_Raccumulate, MPI_Rget_accumulate

Window Creation Methods:

MPI_Win_Create, MPI_Win_allocate, MPI_Win_allocate_shared

Synchronization:

MPI_Win_start/complete/post/wait, MPI_Wait/Waitall

MPI_Lock/Unlock, MPI_Unlock/Unlockall

5.3.2 Test Results and Deliverables for Non-Blocking Collectives

The tests developed for the non-blocking collectives are as follows:

Non-blocking Collective Functions Tested:

MPI_Ibcast, MPI_Iscatter, MPI_Igather, MPI_Iallgather

MPI_Ialltoall, MPI_Iallreduce, MPI_Ireduce_scatter_block,

MPI_Ireduce_scatter, MPI_Iscan, MPI_Iexscan

These tests went through the same review and acceptance process as the RMA tests, and all passed with the most recent version of the Cray MPI implementation.

Chapter 6

Conclusions

6.1 Contributions of This Work

This work resulted in the development of regression tests for the validation of the key aspects of the OpenMP 4.0 ad MPI 3.0 standards as implemented by Cray. The emphasis was in correctness, comprehensiveness, easy of use, and detailed error output. Cray implementations of OpenMP and MPI are used in HPC applications that effect scientific research, as well as human health and safety. Software correctness is the primary characteristic for these applications to be useful, and ensures software quality in this large and important domain.

6.2 Future Work

HPC and parallel programming are exciting and rapidly evolving fields that continually generate new requirements and challenges in the area of software testing. In particular, the OpenMP and MPI standards are in a continual process of revision. Regression testing will

be required to assure that new implementations still provide correct results for old features as well as new ones that will be inevitably incorporated into them.

As mentioned in Sect. 3.3, test validation is important in black box testing, and also a weakness of this project. Given more time, I would like to devise a method for validation of the OpenMP thread affinity tests. The initial division of test cases shown in 4.1 would be a good place to start: I would like to do a more thorough and systematic analysis, and then run the tests several times to be sure that all cases pass. Even though it would require several hours to complete, I think the resulting increased confidence would be worth the effort. Then a more systematic selection of which cases to be included in the final test suite could be made. I think the ad-hoc selection of a handful of use cases that actually occurred was not very well thought out and did not do justice to the sophistication of the test.

This thesis just touched on hybrid programming models, however they are becoming increasingly popular and are used with other thread models besides OpenMP (pthreads is one obvious example). Memory affinity is being explored as an addition to OpenMP and, if incorporates into the standard, will become an interesting area of testing along the lines of the thread affinity tests developed here. In general, the field of software quality has not kept up with performance testing, and new tests suites and methods will be needed as HPC enters the exascale era.

Appendix A

MPI RMA Window Creation and Synchronization

Window Creation

MPI 3.0 has several different ways to expose "Windows" of memory to other processes and to allocate it. The most straightforward method is for each process to allocate a buffer using malloc and then to use the MPI_Win_create function to create a reference to the window which is used in subsequent calls by other processes in the communicator group to access the window. This allows for control over who can access the window and some isolation to prevent corruption of the memory space. MPI also includes methods to allocate shared memory and dynamic memory. Each MPI Window creation function uses it's own set of synchronization functions.

Synchronization

The two general methods of synchronization are called *active target synchronization* and *passive target synchronization*. Both circumscribe the allowed times for processes to ac-

cess memory into *epochs*: The target process opens an *exposure epoch* which control when other processes can read or write it's memory. Correspondingly, the source process uses an *access* epoch which defines when it can access the memory of other processes.

Active Target Synchronization

For active target synchronization both an exposure and access window must be open at the target and source simultaneously for the RMA functions to proceed. As the name implies, the target process actively participates in the timing of access to it's window. Active target synchronization is used when memory access patterns change frequently within an application. There are two synchronization models used with active target synchronization: Fence and General.

Using Fence synchronization, exposure and access epochs are opened and closed for all processes simultaneously using the MPI_Win_fence function. The MPI standard stipulates that a communication operation will complete at the source process before it is called. "Completion" in this context means that the associated memory buffers can either be accessed or re-used (whichever situation is applicable). For example, with MPI_Put, MPI_Fence at the source process means that the data has been transferred out of the buffer and the buffer can safely be used. It says nothing about the completion of the transfer at the target process. When the target process calls MPI_Fence, it means that the data has been moved to the receive buffer at the target process. Therefore a given communication operation cannot be said to have completed in a way that ensures sequential memory consistency semantics until MPI_Fence has been called at all processes.

By contrast, general synchronization allows for more fine-grained communication between process groups. Unlike fence synchronization and exposure and access epochs specify the specific processes that they apply to. General synchronization functions pose a special problem for thread-based applications because the matching MPI_Post/MPI_Wait

and MPI_Start/MPI_Complete must complete in a specified order for each process. In the event that there is not a one-to-one relationship between threads and process groups this can be challenging to keep track of.

Passive Target Synchronization

In passive target synchronization the target process is not involved in co-ordinating communication so, in effect, there is always an exposure epoch open. Lock and Unlock operations control when specific groups of processes can access a given window. Shared memory window allocation will only work with passive target synchronizations and operations on these windows use the same lock/unlock functions. There are two kinds of lock/unlock functions: one that targets specific processes (MPI_Win_lock/MPI_Win_unlock) and one that involves all processes (MPI_Win_unlock/MPI_win_unlockall).

Bibliography

[1] T. Linz A. Spillner and H. Schaefer. *Software Testing Foundations, 4th Edition*. Santa Barbara, CA 93103: Rocky Nook Inc., 2014.

[2] P.E. Black and E. Fong. *Report on the Metrics and Standards for Software Testing (MaSST) Workshop*. Tech. rep. NISTIR 7920. Gaithersburg, MD 20899-8970: National Institute of Standards and Technology, 2012.

[3] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2013. URL: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[4] Advanced Scientific Computing Advisory Committee Subcommittee on Exascale Computing. *The Opportunities and Challenges of Exascale Computing*. Tech. rep. US Dept. of Energy, Office of Science, 2010.

[5] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. 2012. URL: `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[6] U. W. Ghazali. *Software Testing: Essential Skills for First Time Testers*. This compact and comprehensive guide has a list of several software bugs (and their economic and human consequences) at the end of Chapt. 1. Bayaeditions@gmail.com, 2014.

[7] W. Gropp and R. Thakur. "Thread Safety in an MPI Implementation: Requirements and Analysis." In: *Parallel Computing* 33.9 (2007), pp. 595–604.

[8] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, FL: CRC Press, 2011.

[9] P-Fr. Lavallee and P. Wautelet. *Hybrid MPI-OpenMP Programming*. 2015. URL: `http://www.idris.fr/data/cours/hybride/form_hybride_en_proj.pdf`.

[10] *NIST Software Quality Group*. URL: `www.nist.gov/itl/ssd/cs/index.html`.

[11] T. Hoefler P. Balaji and M. Schulz. *Advanced Parallel Programming with MPI*. 2013. URL: `http://www.mcs.anl.gov/~balaji/2013-06-16-isc-mpi.pptx`.

[12] P. Pacheco. *An Introduction to Parallel Programming*. Burlington, MA: Elsevier Inc., 2011.