STATIC ANALYSIS TOOL FOR SYNCHRONIZATION ANALYSIS, REPRESENTATION, AND OPTIMIZATIONS FOR APPLICATIONS USING OPENSHMEM

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Swaroop Suhas Pophale

May 2014

STATIC ANALYSIS TOOL FOR SYNCHRONIZATION ANALYSIS, REPRESENTATION, AND OPTIMIZATIONS FOR APPLICATIONS USING OPENSHMEM

Swaroop Suhas Pophale

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Jaspal Subhlok
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

Dr. Shishir Shah
Dept. of Computer Science

Dr. Eric Bittner
Dept. of Chemistry

Dean, College of Natural Sciences and Mathematics

STATIC ANALYSIS TOOL FOR SYNCHRONIZATION ANALYSIS, REPRESENTATION, AND OPTIMIZATIONS FOR APPLICATIONS USING OPENSHMEM

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Swaroop Suhas Pophale

May 2014

Abstract

Programming models provide application developers abstraction from the underlying hardware. OpenSHMEM library follows the Partitioned Global Address Space programming model, which is characterized by local and global views of data. The OpenSHMEM library API provides synchronization primitives that require participation of some or all OpenSHMEM processes executing the application (*collective*). Since most distributed parallel applications spend 30-40% of their execution time performing synchronization, it is a constant struggle for most application programmers to relax the memory consistency constraints while guaranteeing reproducible and correct results. From our experience, we have seen that generally programmers tend to over-synchronize when in doubt, and the best approach towards creating correct, scalable, and performance driven applications is to help programmers leverage optimizations based on the semantics of the OpenSHMEM library. Unfortunately, most application developers are not well acquainted with all the nuances of the targeted programming libraries and spend most of their development time focused on the correctness aspect alone. This leads to a need for a framework to provide programmers better understanding of the applications and provide useful feed back making it easier for the application developer to incorporate basic and advanced optimizations into their applications with ease. For this we collaborated with the Oak Ridge National Laboratory (ORNL) to build a compiler-based tool called the OpenSHMEM analyzer (OSA), which makes the OpenUH compiler aware of the OpenSHMEM library semantics. Along with basic semantic checks, the analyzer provides useful feedback at compile time, leading to faster turn around time and lesser wastage of resources in terms of debugging time or failed execution runs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Until recently, all software has been written for serial computation where a problem is broken down to a series of instructions which are executed, one instruction at a time, on a single CPU. Parallel computation uses multiple compute resources to execute different parts of a problem simultaneously on different processing units, while obtaining the same correct result as obtained with serial execution. The popularity of parallel programming stems from its capacity to save time and resources. Computer systems based on their capability of handling data and instructions simultaneously are classified by M. J. Flynn [27] into four possible combinations: Single Instruction Stream Single Data Stream (SISD), Single Instruction Stream Multiple Data Stream (SIMD), Multiple Instruction Stream Single Data Stream (MISD), and Multiple Instruction Stream Multiple Data Stream (MIMD). Single Program Multiple Data (SPMD) is a special case of MIMD wherein a number of processing elements execute the same set of instructions on different sets of data. In parallel computing

environment certain assumptions are made about the data layout and the manner in which the processes communicate; these are usually defined by a programming model. With different programming models facilitating applications utilize the modern hardware architecture, getting maximal performance for applications is a very important concern. Our work aims to provide scientific application developers with a compiler-based tool to be more productive in their development process while giving critical information about their application that may have significant impact on its performance characteristics. The added benefit of providing this information at compile time is that the programmer does not have to execute the application to encounter potential bugs and bottlenecks. This saves valuable billable computing resources that can be put to better use.

1.1 Memory Models

Two distinct and widely used memory models are shared and distributed. These models are an abstraction over the hardware and the memory architecture.

1.1.1 Shared Memory Model

All shared memory parallel computers generally have the ability for all processors to access the entire system memory as global address space. The same memory resources are shared by all the processing elements. Depending upon their memory access, shared memory machines can either be uniform memory access (UMA) or

non-uniform memory access (NUMA). OpenMP [19] and pthreads [47] are the two most prominent application programming interfaces (API) used to implement multi-threaded applications on a shared memory systems.

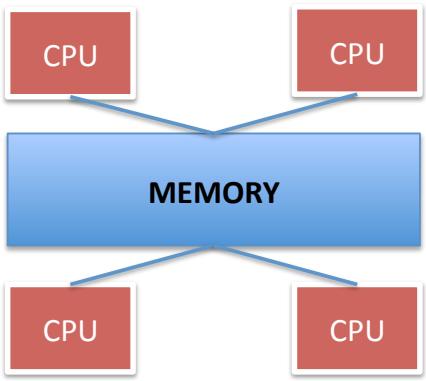


Figure 1.1: Uniform Memory Access

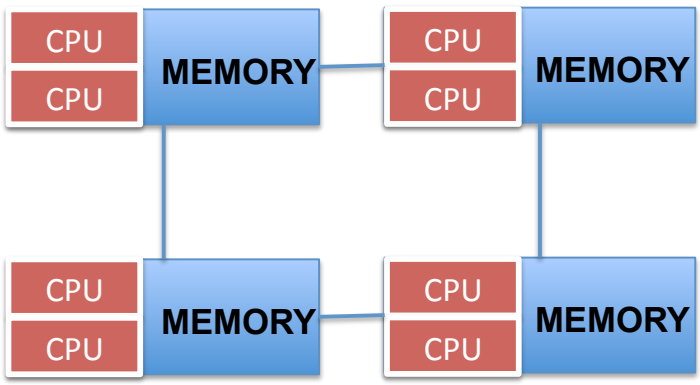


Figure 1.2: Non-uniform Memory Access configuration of Shared Memory System.

1.1.2 Distributed Memory Model

The distributed memory programming model is ideally suited for a memory systems where each processor maintains its own local memory and has no direct access to

another processor's memory. Data has to be shared explicitly via some form of communication. Due to the distributed nature of the memory the access time for data on the local memory of a processor is much less than accessing data that resides on another processor. The Message Passing Interface (MPI) has become a de facto standard for communication among processes running on a distributed memory systems.

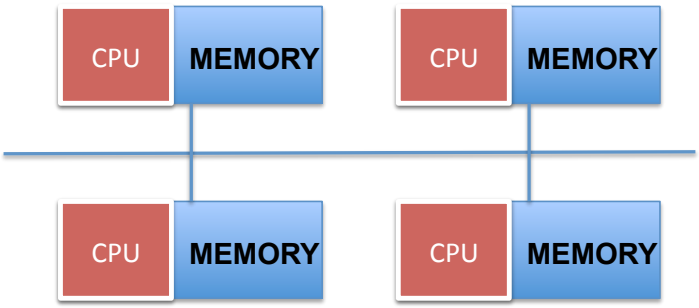


Figure 1.3: Distributed Memory System.

1.2 Parallel Programming Models

In this section we go over the most popular programming models used for scientific computing today. MPI has become the de facto standard for communication among processes executing over distributed memory systems, while OpenMP provides directive driven approach for shared memory systems.

1.2.1 Message Passing Interface (MPI)

MPI is a specification for all implementations of message passing library. Although primarily developed for communication over distributed memory systems, it supports shared memory multiprocessors, and networks of workstations. MPI provides seamless abstraction to the application, making it independent of network speed or memory architecture [10, 66]. In the message passing model, data are copied from the address space of one process to that of another process through cooperative operations on each process [73]. MPI library's popularity stems from the fact that it addressed the drawbacks of its predecessor, PVM (Parallel Virtual Machine), is portable, and provides many useful library routines for the distributed memory communication environment that facilitate development of High Performance Computing (HPC) applications. With many high quality open [28] and vendor specific implementations available it has become one of the most used parallel programming library.

Starting with MPI 1, the MPI standard has gone through a number of revisions, with the most recent version being MPI 3. As depicted in Figure 1.4, in MPI 1.0, during execution, if a process requires data from another process, it must request the data by sending a message to the destination process through the interconnecting network. When the data become available on the destination process, it is sent within a message to the process that requested the data. MPI programs generally follow a SPMD programming style.

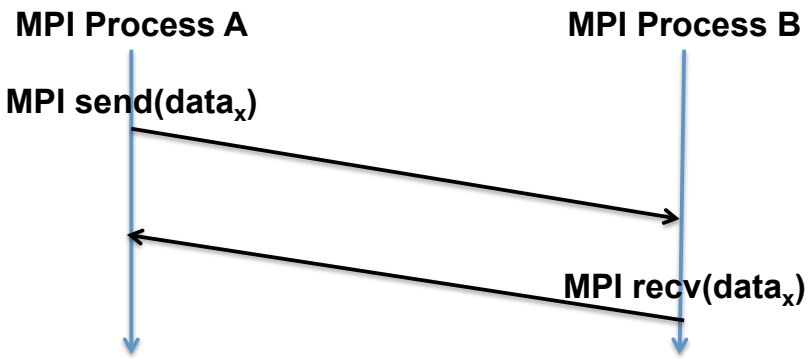


Figure 1.4: MPI 1 Send - Receive Semantics

MPI 2 introduced dynamic processes, one-sided communications, extended collective operations, external interfaces for debuggers and profilers, additional language bindings and an interface for parallel I/O. MPI 3, adopted in 2012, adds non-blocking functionality, improves on the one-sided communication semantics introduced in MPI 2, extends the collective API to provide neighborhood collectives, and provides Fortran 2008 bindings along with an improved tools interface.

1.2.2 OpenMP

OpenMP [19] is a specification, defined by the OpenMP Architecture Review Board (ARB), for a set of compiler *directives* along with library routines and environment variables for shared memory programming. These directives can be used to specify high-level parallelism in Fortran and C/C++ programs to efficiently leverage the parallelism possible on a shared memory platform through the use of threads. Threads are the smallest unit of processing that can be scheduled by an operating system and

they exist within the resources of a single process and cannot have independent existence. For parallelization, programmers look for regions of code whose instructions can be distributed amongst threads. When the different iterations may be executed independent of each other, programmers use special constructs so that these portions of the code are executed asynchronously.

OpenMP's execution model has one dedicated master thread, and the master thread executes the program sequentially until the first parallel region construct is encountered. At this point the master thread *forks* a team of concurrent threads. The statements enclosed by the parallel region construct are then executed concurrently among the threads belonging to the team. On completion of the statements in the parallel region construct, the threads in the team synchronize and terminate, leaving only the master thread as before the parallel construct. This is best described as a *fork-join* model as depicted in 1.5.

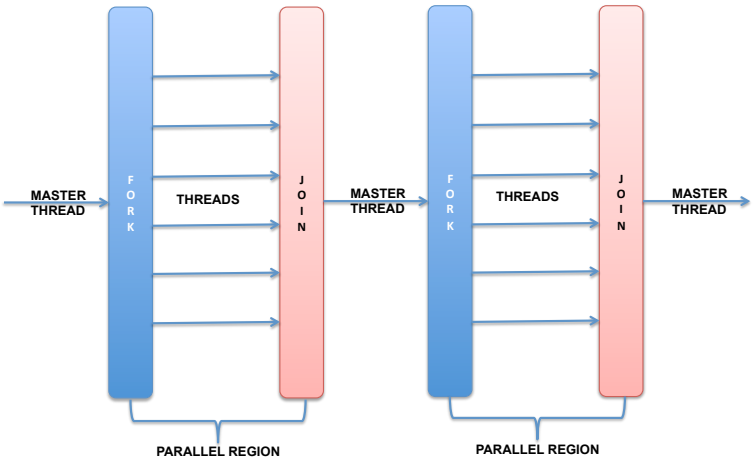


Figure 1.5: OpenMP's Fork-Join Model

Both shared and distributed programming models have their advantages and

disadvantages, but the ideal combination is the data locality features of distributed memory model, with the simplistic data referencing of a shared memory model. The want for this mixed features lead to the Partitioned Global Address Space (PGAS) model. The PGAS programming model provides a local-view programming style which differentiates between local and remote data, while also providing a logical global address space which is directly accessible by all executing process.

1.2.3 Partitioned Global Address Space (PGAS)

The PGAS programming model provides a local-view programming style which differentiates between local and global data. The local data are not directly accessible to all executing processes but the global data can be accessed by all without going through an elaborate acknowledgement-based communication pattern. Figure 1.6 shows the logical view of the memory model provided by the PGAS programming model. Because of its simple data accesses, PGAS languages like Unified Parallel C (UPC) [4], Co-Array Fortran (CAF) [51], and libraries such as Global Arrays (GA) Toolkit [49] and OpenSHMEM [18] are becoming increasingly popular. A slightly different category of PGAS model, termed Asynchronous Partitioned Global Address Space (APGAS) model, has recently emerged with additional capabilities such as remote method invocations, which enable nodes to invoke work on other nodes, and allows each node to execute multiple tasks from a task pool. IBM's X10 language [21] and Chapel [16] follow the APGAS programming model. They are a part of the DARPA HPCS [24] project and aim to improve programmer productivity on next-generation computing architectures by providing a richer execution framework

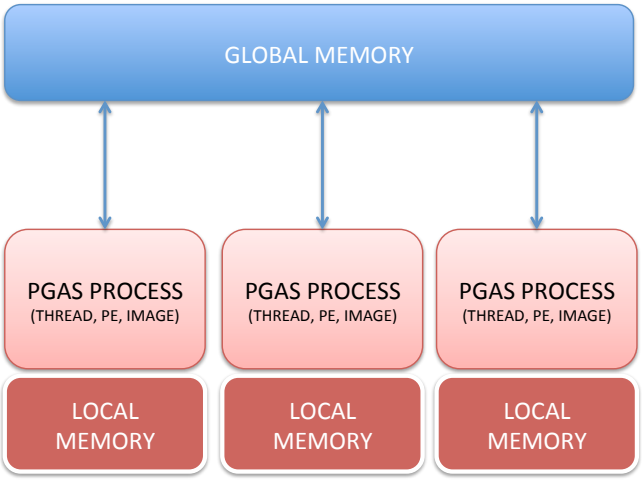


Figure 1.6: Partitioned Global Address Space Logical Memory Model.

than the SPMD style generally used by the traditional PGAS languages.

All the above mentioned PGAS languages and libraries either use the services of an underlying GAS runtime for their communication needs or communicate using the specialized hardware support where possible. Aggregated Remote Memory Copy Interface (ARMCI) [50] and Global Address Space Network (GASNet) [12] are two such low level GAS libraries that the PGAS languages can use at runtime as a compilation target to perform data transfers on distributed memory architectures. These GAS libraries have a translation layer that converts a memory access to corresponding data transfer calls that are specific to the underlying system hardware.

1.3 Factors affecting PGAS Applications

1.3.1 Collective Operations

To better understand the problems faced by application programmers while using the PGAS programming model we undertook a study to port the NAS parallel benchmarks [8] from their MPI 1 based implementations to those that use the OpenSHMEM library API. We test the same OpenSHMEM NAS kernels on three platforms with three different OpenSHMEM implementations; an SGI UV cc-NUMA shared memory system (with SGI's OpenSHMEM library), a Cray XT5 system (with Cray's OpenSHMEM compliant impelmentation) and an InfiniBand interconnect based Opteron cluster (with the OpenSHMEM Reference Implementation [61]).

We analyze their execution times to identify the categories of function calls that have the most impact on the performance of the benchmarks and identify productivity issues while programming with the OpenSHMEM library [60]. For completeness we also compare performance and scalability of these OpenSHMEM NAS benchmarks with their MPI 1, and in some cases, MPI 2 counter parts to analyze the strengths and weaknesses of the OpenSHMEM library implementations. To collect the OpenSHMEM performance profiles, we used Tuning and Analysis Utilities (TAU) [65] that collects relevant information through event-based sampling, on the SGI and Cray platforms. On the Cluster, we integrated the collector interface [30] into the OpenSHMEM reference library to collect the profiling information. We considered the following NAS benchmarks;

- Block Tridiagonal (BT)

  BT is a simulated computational fluid dynamics (CFD) applications that solves 3-dimensional compressible Naiver-Stokes equations. BT uses Alternating Direction Implicit (ADI) to find the finite difference solution to the problem. ADI involves solving three sets of uncoupled systems of equations in x, y, and z direction [8]. These equations are block tridiagonal in BT.

- Scalar Pentadiagonal (SP)

  Like BT, SP is also a simulated CFD application benchmark that solves 3-dimensional compressible Naiver-Stokes equations. The difference is that SP uses the Beam-Warming approximate factorization. The three sets of uncoupled systems of equations in the x, y and z direction are scalar penta-diagonal in SP. SP and BT are similar in many respects, but there is a fundamental difference is in their communication to computation ratio [9]. SP has higher communication to computation ratio, which allows for a better overlapping of communication with computation. The computation kernel in SP is different. However the communication kernel remains same for BT and SP.

- Integer Sort (IS)

  IS is the bucket sort based integer sorting kernel where each process sorts the numbers whose keys fall in its key range.

- Multi Grid (MG)

  MG uses a V-cycle Multi Grid method to compute the solution of a 3D scalar poisson equation. Each process uses a sequence of calls to routines *ready*,

give, and *take*. In the original MPI 1 version, the *ready* routine posts fake MPI requests so that the *take* routine can do a asynchronous wait on the corresponding *send* .

1.3.1.1 Performance Analysis

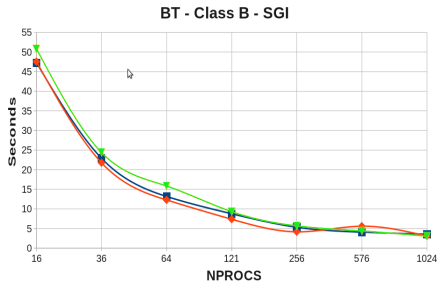
Figures 1.7 and 1.8 show the timings of the BT and SP benchmark implementations on the three different platforms. We have chosen class B as a common representative of the performance pattern, as the performance characteristics are consistent across different classes of input data. On SGI, all the versions show competitive performance and linear scaling from 16 to 1024 processes. On the maximum number of processes (1024), the performances of MPI 1 implementation of the BT benchmark and the OpenSHMEM version of the SP benchmark are the best. On the Cray platform, we see that the scaling is not so promising for both MPI 2 and OpenSHMEM implementations. On the Cluster, a similar performance trend is seen except, on maximum number of processes (256), the MPI 2 and OpenSHMEM versions perform better than MPI 1.

Figures 1.9 and 1.10 shows the top two time-consuming library calls for the BT and SP benchmarks. We see that the MPI 2 *fence*, and the OpenSHMEM *barrier* on the Cray platform causes degradation of performance on higher numbers of processes.

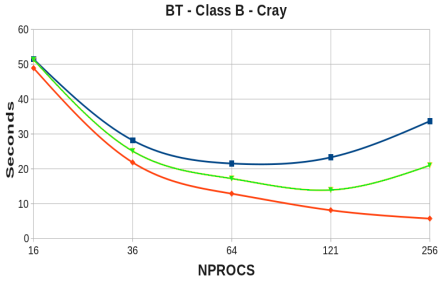
Figures 1.11 and 1.12 show the timing of the IS and MG benchmarks on the different platforms. We have chosen class C as a common representative of the performance pattern. On SGI the MPI 1 version of IS performs very slightly better

than the OpenSHMEM version, whereas the MPI 2 version fails to do well. A similar pattern is seen for the IS benchmark on the Cray platform where MPI 1 performs slightly better than the OpenSHMEM version, and the MPI 2 version does not do well. Figure 1.13 summarizes the top time-consuming calls of IS at the highest numbers of processes (256). Clearly the performance of MPI-IS is strongly dependent on having efficient *alltoall* and *alltoallv* implementations. The execution time of OpenSHMEM-IS is significantly affected by *shmem_put* which is used to simulate the *alltoall* communication pattern. As seen in Figure 1.11 (c) the OpenSHMEM library implementation on the Cluster is not optimized for performance but on the other two platforms the OpenSHMEM benchmarks obtain performance as good as MPI 1.

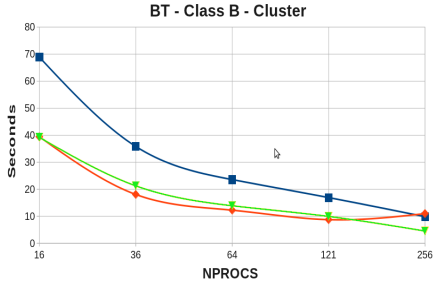
On SGI, the best performance is shown by the MG benchmark using OpenSHMEM, whereas on the other platforms, MPI 1 does slightly better. Figure 1.14 shows the impact of *mpi_wait* on the performances of the MPI 1 version of the MG benchmark. It also shows the effect of the *barrier* implementation on the performance of the OpenSHMEM-MG on the Cray platform. We observe that the reduction operation is the most time-consuming on the Cluster, which is in part due to the network latency of the underlying hardware. OpenSHMEM also uses more of the group synchronization calls (*shmem_barrier_all*) because the completion semantics of the one-sided *put* operation which are guaranteed only at synchronization. For the MG benchmark, the *shmem_barrier* call has a significant effect on the performance although the OpenSHMEM version performs as well as the MPI 1 version.



(a) BT using SGI OpenSHMEM



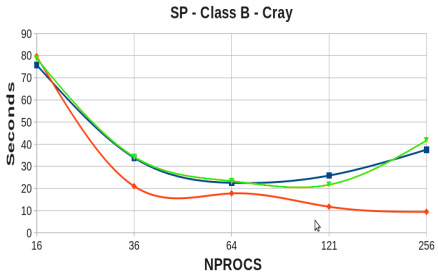
(b) BT using Cray OpenSHMEM



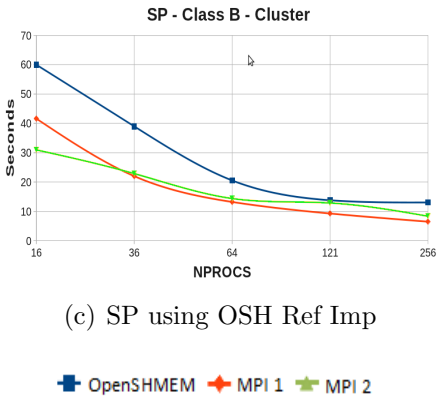
(c) BT using OSH Ref Imp

Figure 1.7: Performance of MPI 1, MPI 2, and OpenSHMEM implementations of the BT benchmark

(a) SP using SGI OpenSHMEM



(b) SP using Cray OpenSHMEM



(c) SP using OSH Ref Imp

Figure 1.8: Performance of MPI-1, MPI 2, and OpenSHMEM implementations of the SP benchmark

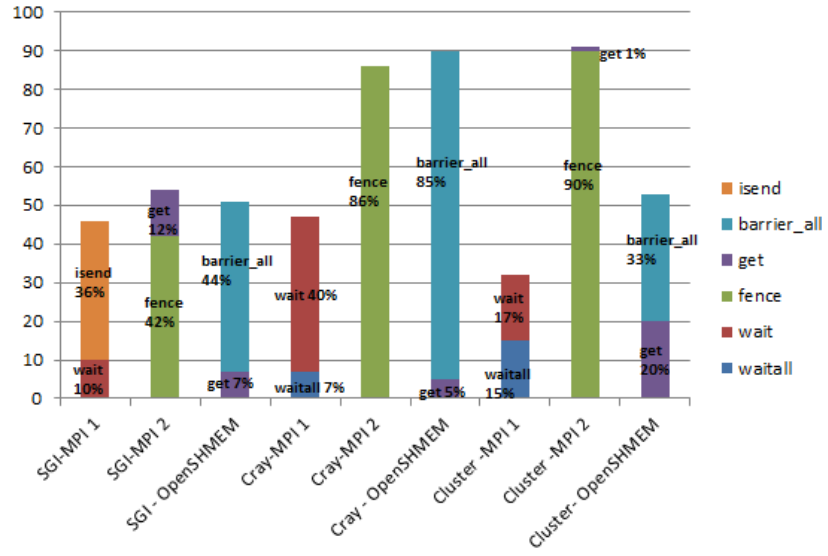


Figure 1.9: Two most time-consuming library calls in BT benchmark's implementations

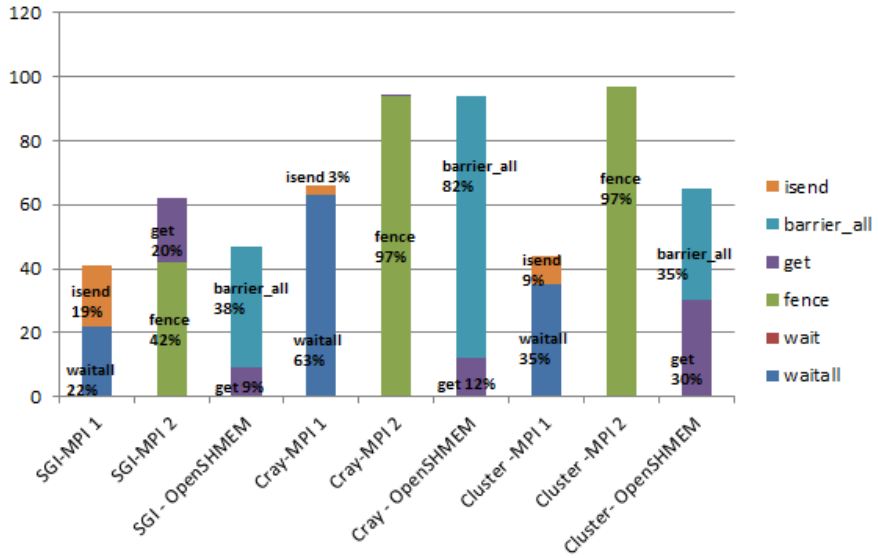
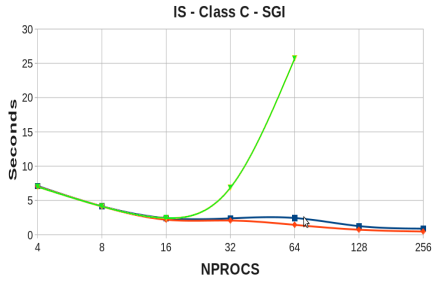
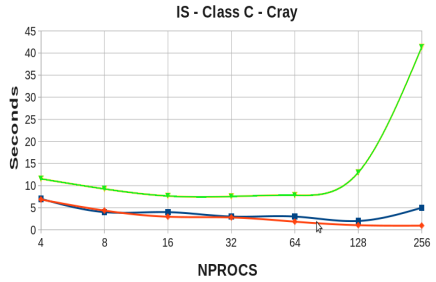


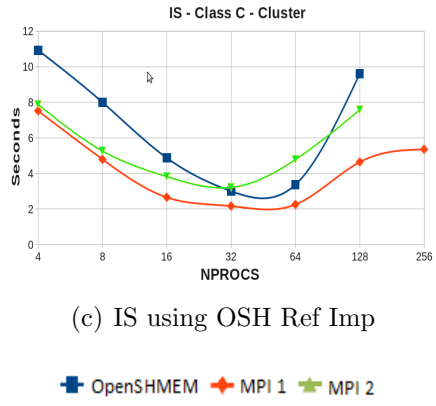
Figure 1.10: Two most time-consuming calls in SP benchmark's implementations



(a) IS using SGI OpenSHMEM

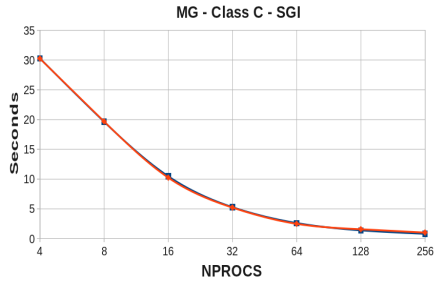


(b) IS using Cray OpenSHMEM

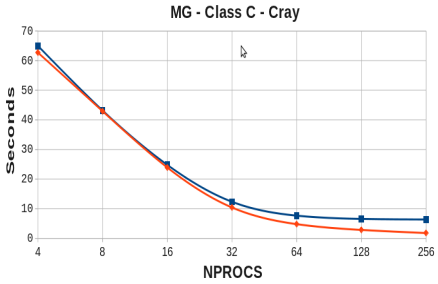


(c) IS using OSH Ref Imp

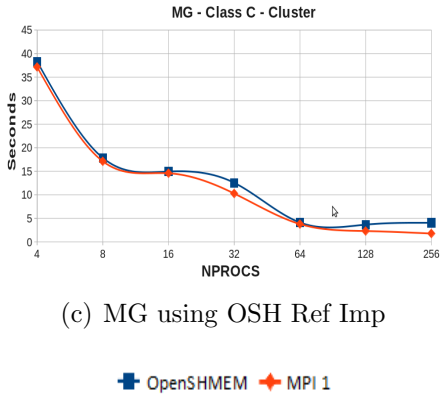
Figure 1.11: Performance of MPI 1, MPI 2, and OpenSHMEM implementations of the IS benchmark



(a) MG using SGI OpenSHMEM



(b) MG using Cray OpenSHMEM



(c) MG using OSH Ref Imp

Figure 1.12: Performance of MPI 1 and OpenSHMEM implementations of the MG benchmark

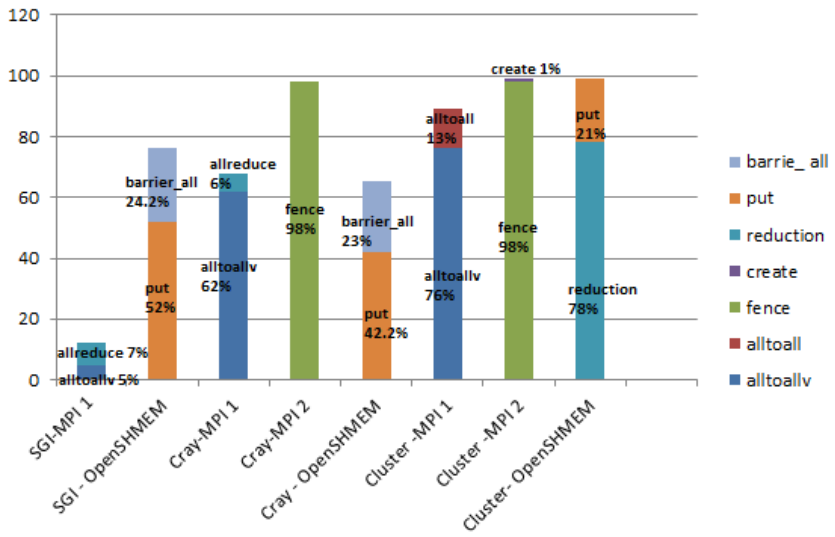


Figure 1.13: Two most time-consuming library calls in IS benchmark's implementations

We looked at performance of OpenSHMEM applications on three distinct platforms, ranging from specialized (SGI and Cray) to general purpose (Cluster) with varying degree of hardware support for OpenSHMEM functionalities to remove any bias introduced by any specific OpenSHMEM library implementation. The performance of the benchmarks using OpenSHMEM observed during this study is non-uniform, but results collected over all platforms indicate that OpenSHMEM applications are greatly impacted by the *collective* operations, especially group synchronization calls, such as *shmem_barrier* and *shmem_barrier_all* (refer to graphs in figures 1.9, 1.14, and 1.13). We use this information as a starting point for our compiler analysis.

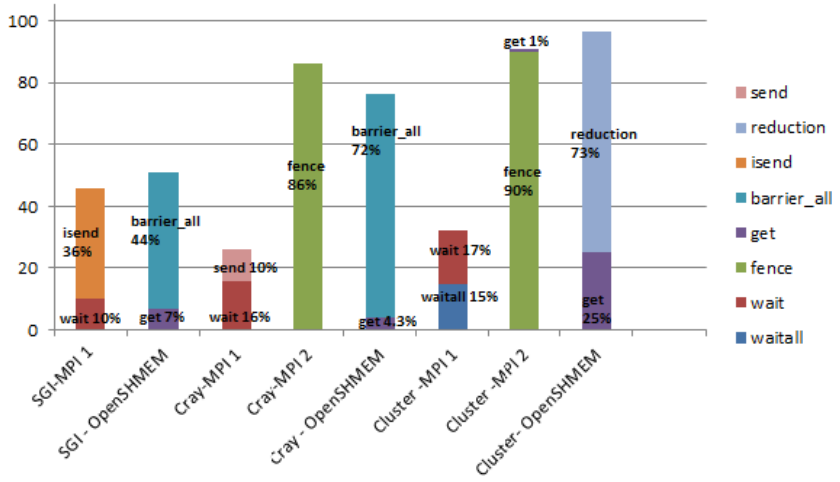


Figure 1.14: Two most time-consuming calls in MG benchmark's implementations

1.3.2 Locality

Topology mapping is a popular field of research and there are many contributions with respect to the optimal process placement. Most of these studies involve specific hardware platforms, and/or are limited to the distributed message passing paradigm [31]. For example, several vendor-distributed MPI implementations use graph theory algorithms to formulate a near optimum placement of resources. But these solutions are implementation specific. As the number of cores per node increase and the nodes per system increase, the PGAS programming model will have to cope with communication latency associated with physically distant processes communicating over underlying interconnect. A good library or language extension will have to find innovative solutions to hide the communication latency when communicating processes are on physically distant nodes.

To understand the effect of locality on OpenSHMEM applications we developed a portable scheme which works in conjunction with our OpenSHMEM reference library implementation. This methodology can be extended to all other PGAS libraries and language extensions. The methodology consists of profiling application with its production time parameters with TAU performance tool [65] and collecting communication information. This information is then fed at start-up to the OpenSHMEM library during the actual run or *production* run, thus ensuring that processing elements with the maximal communication are as proximally located as possible, given the capacity and limitations of the underlying hardware. This scheme is calibrated depending on the communication pattern amongst processes executing the application. The relative communication volume between processes as measured during the preliminary run and process re-naming is based on this information, since the pattern of communication does not change with change in the amount of data to be processed. As long as the ratio of the individual point-to-point communication is the same, the mapping will be optimum for all future runs using the same numbers of processes.

We experimented with the 2D heat transfer benchmark application adapted from the parallel MPI implementation of 2D heat conduction [1] on an InfiniBand cluster (called Crill [2]) with 16 nodes, with each node containing four twelve-core Opteron processors running the Linux operating system. The benchmark calculates finite difference over a regular domain using the Jacobi, Gauss-Siedel and SOR [63] methods. We observed that with prior knowledge of the communication pattern, the application executed 1.62 times faster, with 64 processing elements, using our process

remapping scheme. The drawback of such software approaches is that at least one execution of the application is required for collection of communication pattern. Also the scheme is applicable only for applications that have predictable and unchanging patterns of the communication over time and size of the input, hence this is not a general purpose solution.

1.3.3 Language or Library Specifics

All languages and libraries provide a substantial API to accomplish a large number of operations. These functions have a unique signature and very specific completion semantics. There are two aspects to using these functions, one is syntactic correctness and the other is optimization. Syntactic correctness involves using the correct parameters for a given API as directed by the governing Specification. But using the API optimally is a non-trivial challenge. The functions other than addressing common application needs are also designed specifically to leverage certain features provided by the programming model or the underlying hardware that they are designed for. But not all application developers are aware of the best ways to completely exploit the semantics of the API. For example, the OpenSHMEM library provides implicit synchronization at the end of every reduction operation, thus eliminating the need for explicit synchronization after these calls. As it is often observed, application programmers are not very well acquainted with these nuances, leading to missed opportunities for possible optimizations.

1.4 Goals

This dissertation aims at approaching the problem of development and performance of applications using OpenSHMEM in a holistic manner. We present a solution in the form of a compiler analysis tool in the form of the OpenSHMEM Analyzer. While it is not possible to address all problems through static analysis, the framework provided by the OpenSHMEM Analyzer (OSA) tool provides a robust, extensible and scalable prototype for aiding application development in the OpenSHMEM environment.

1. **Relaxing Memory Consistency without Impacting Correctness**

   As we saw from our experiments with the NAS Parallel benchmarks suite, the collectives are a necessary evil. From the graphs in Figures 1.9, 1.10, 1.13, and 1.14, we see that as much as 72-85 % of the time spent in OpenSHMEM calls is for shmem_barrier_all operation. Since this operation has both completion as well synchronization aspects it is sought out as a go to solution for all situations, even when less expensive **correct** options are available. We look at these operations as a starting point on which to build our synchronization analysis and optimization framework.

2. **No Common Intermediate Representation**

   All PGAS languages and libraries have similar functionality and are translated to some intermediate representation which in turn is lowered to a set of equivalent runtime library calls (discussed in detail in Chapter 2). We can view OpenSHMEM as a common intermediate representation to represent all

such language extensions as well as libraries. By having a common representation, all optimizations can be applied across all function calls with the same functional and completion semantics.

3. **Providing PGAS Specific Feedback for Correctness and Optimizations**

   Building a compiler-based static analysis tool provides a more comprehensive solution to developing applications using the PGAS programming model. We developed an OpenUH compiler-based tool which makes the compiler aware of the PGAS semantics of the OpenSHMEM library and gives feedback in the form of compile time messages, visual representation of the program synchronization structure, and the system dependence graph that better explains the control flow and data flow within the application.

Chapter 2

Partitioned Global Address Space (PGAS)

2.1 PGAS Languages

2.1.1 Unified Parallel C

UPC [4] is an extension to the C language to enable parallelism [70]. Extensions include global pointers, data distribution declarations for shared data, and work-sharing constructs. The first version of UPC, known as version 0.9, was published in May of 1999 as technical report at the Institute for Defense Analyses Center for Computing Sciences [70]. A UPC program is executed by a set of processes, referred to as *threads* by the UPC execution model, which may allocate both shared and private data objects. Except at declaration time there is no syntactic difference in the

access to local and global data objects. UPC follows the SPMD style of programming where a number of threads work independently and communicate through reads or writes to shared data in the global memory. Private objects belonging to a thread cannot be accessed by any other thread. The local data items reside in the private memory of the thread while the globally shared data values are placed in the global memory. Also, the shared data in the global memory are logically partitioned among the threads; i.e. there are sections of the global data which are *closer* to a thread, this leads to memory regions in the global address space to which threads have *affinity*. Figure 2.1 depicts the memory model in UPC.

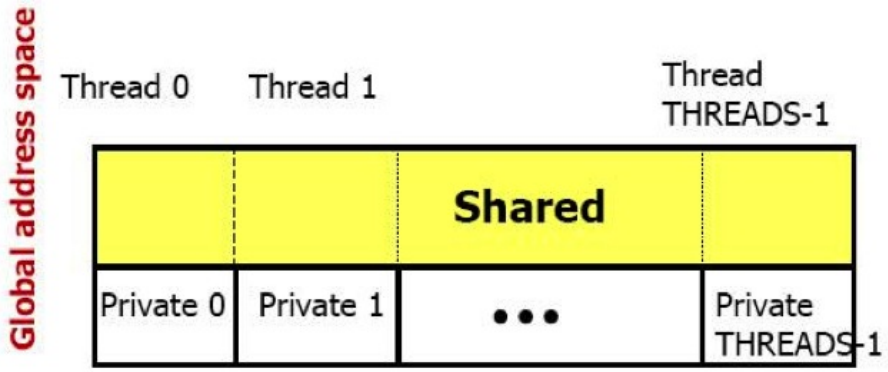


Figure 2.1: UPC Memory Model [38].

The UPC memory model supports three different kinds of pointers: private pointers pointing to the shared address space, pointers living in shared space that also point to shared data, and private pointers pointing to data in the thread's own private space. The access times depend on whether the data are local or remote and the UPC performance largely depends on the number of threads and how they access the shared space. UPC also provides synchronization mechanisms such as barriers, split-phase barriers, and other memory consistency controls. Better performance is

observed when a process accesses data which are held locally or data in the partition of the global address space to which the thread has affinity. This fact is exploited by the UPC work-sharing construct *upc_forall* which distributes iterations across threads in a way such that each thread operates on the local portion of shared arrays.

2.1.2 Co-Array Fortran

Co-Array Fortran (CAF) [52] was introduced as a small language extension to Fortran 95 which facilitates data decomposition by providing explicit notations. The syntax is independent of architecture and was finalized to be added to the Fortran standard by the ISO Fortran Committee in May 2005. CAF also follows the SPMD execution pattern where a copy of the same program is executed asynchronously by multiple *images*. Images are indexed starting from one to the total number of images executing the application. Inside a CAF program an image can retrieve their index through the intrinsic function *this_image()*. To access objects residing on another image CAF introduces the concept of *co-subscript* which is indicated between square braces. Like other PGAS languages (and libraries), all accesses to remote objects are *one-sided*, thus requiring no involvement of the image whose object is being accessed.

2.1.3 Chapel

Chapel is a parallel programming language that has been developed by Cray Inc. under the DARPA High Productivity Computing Systems (HPCS) program [16]. Initial goal for Chapel when it emerged from Cray's entry in the HPCS was to provide a

portable interface to improve programmer productivity on high-end parallel systems and provide a programming model that is attractive for commodity clusters, as well as desktop computing. Chapel is also being developed in an open-source manner at SourceForge. Chapel supports the PGAS programming model by providing global-view data aggregates with user-defined implementations. This allows for operations on distributed data structures. Chapel's parallel features are most directly influenced by ZPL [17], High-Performance Fortran (HPF) [62], and the Cray MTA/XMT [11] extensions to C and Fortran [23]. The latest version of Chapel, version 1.7.01, was released on April 18th, 2013.

Chapel supports a multithreaded execution model. For this purpose Chapel supports high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. To enable locality and affinity, Chapel introduces the concepts of *locales*, which enables users to specify the placement of data and tasks on a target architecture. Chapel upholds all object-oriented concepts by supporting code reuse, rapid prototyping, and type inference.

2.2 PGAS Libraries

2.2.1 Global Arrays

The Global Arrays (GA) Toolkit is a PGAS library for C and Fortran languages [48]. GA is implemented using the ARMCI runtime library. The runtime library is responsible for allocating global memory and accessing remote memory locations while

the *Memory Allocator* interface allows for allocating local memory, and providing memory availability and utilization information statistics. The Distributed Arrays (DA) layer detects the location of the data that an operation is accessing (physical shared memory or remote memory) and accordingly invokes the corresponding ARMCI functions.

GA allows for creation of global arrays with irregular distribution. This enables different processes to have different number of array elements. Like other PGAS languages and libraries, GA provides synchronization, environment query functions, remote read/write, non-contiguous data transfers, collective operations, and atomic operations. Functions and features unique to GA include; duplicating arrays, setting data, support for locality, scaling of array elements, and array copy features.

2.2.2 Titanium

Titanium is a dialect of Java [41] and follows the SPMD execution model, so all threads execute the same code image. Titanium does not use the Java Virtual Machine model, but instead, is first translated into C that is then lowered by the compiler to assembly code. The compiler generates calls to the runtime which is based on the lower level communications library GASNet. Titanium merges the PGAS programming model concepts and enriches Java's features by providing checked synchronization, support for complex data structures, and the use of object-oriented class mechanism along with the global address space to support large shared structures. Titanium provides a global memory space abstraction and the lightweight

communication layer exploits hardware support for direct remote reads and writes [53]. Titanium upholds the PGAS programming model by providing user-controllable processor affinity, when at the same time all parallel processes may directly reference each other's memory to read or write values or perform bulk data transfers.

Titanium runs on a wide range of platforms including uniprocessors, shared memory machines and distributed memory machines and a host of other specialized architectures including Cray X1, Cray T3E, SGI Altix, IBM SP, Origin 2000, and NEC SX6.

2.2.3 OpenSHMEM

An OpenSHMEM [18] library is based on the OpenSHMEM Specification [5], which is an open standard for all SHMEM library implementations. The OpenSHMEM standard is formulated with SGI's SHMEM library specification and implementation as the inception point. The OpenSHMEM Specification 1.0 was finalized by the OpenSHMEM community in early 2012. Existing OpenSHMEM libraries include those developed by SGI [3], Portals SHMEM [13], Scalable SHMEM developed by Mellanox and the OpenSHMEM Reference Library [61] (developed at the University of Houston in collaboration with Oak Ridge National Laboratory). The OpenSHMEM API [54] provides concise and powerful library calls for communicating and processing data. All OpenSHMEM read and write calls are *one-sided*, i.e. they do not require the involvement of the target processing element (PE) for completion and when the underlying hardware allows Remote Direct Memory Access (RDMA),

it can provide excellent opportunities for hiding communication latency by overlapping communication with computation. The OpenSHMEM API provides calls for data communication, point-to-point and group synchronizations, data collection and reduction operations, distributed locks, and process and data accessibility checking. Figure 2.2 depicts the logical memory model in OpenSHMEM.

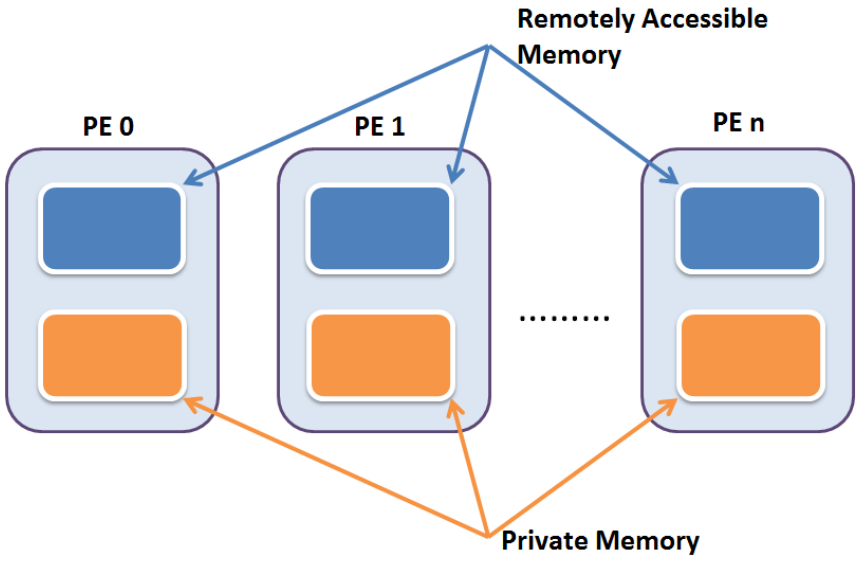


Figure 2.2: OpenSHMEM's Logical Memory Model.

The OpenSHMEM programming library allows a programmer to write parallel applications using a PGAS programming model, where all the processes operate on a globally accessible address space while allowing local data objects for individual computations. In order to support this programming model, OpenSHMEM has the concept of *remotely accessible memory* or *symmetric memory*.

OpenSHMEM library implementations are available for C, C++, and Fortran programs. OpenSHMEM is a library of choice for programs that perform computations

in separate address spaces and explicitly communicate data to and from different PEs in the program. Typically, *target* (data object to be written to) or *source* (data object to be copied from) data that reside on remote PEs are identified by passing the address of the corresponding data object on the local PE. With an evolving API that aims to provide user-friendly operations required by real world applications, OpenSHMEM has the potential to provide efficient solution to a multitude of HPC problems.

2.2.3.1 Key Concepts and Definitions

1. **Processing Element**

   Each executing *process* is called a *processing element* and is referred to as PE X, where

   $$X \in \{0, 1, , N-1\}$$

   and N is the total number of parallel processes executing the application.

   In the OpenSHMEM library environment the PE is an execution unit with associated memory, which is a combination of local memory and remotely accessible memory. The remotely accessible regions are the ones that the library uses for most operations. PEs do not have access to other PE's local memory, which is in congruence with the fundamental concepts of the PGAS programming model.

2. **Symmetric Memory**

   The key concept of OpenSHMEM is the use of symmetric data. Typically, one or more of the parameters of the OpenSHMEM data communicating/processing

calls is required to be *symmetric*. Symmetric or remotely accessible data can be allocated by a OpenSHMEM library (using special memory allocation calls) or is defined as *global* or *static* in C/C++ or in *common* or *save* blocks in Fortran. Symmetric variables have the same name, size, type, storage allocation, and relative address on all PEs.

Library allocation of symmetric data allocation is a *collective* process and has to appear at the same point in the code with the same *size* value. Additionally a dynamic memory allocation call cannot request more space than the size of the *symmetric heap*, which is the *remotely accessible heap memory* available to a PE and its value is implementation specific. Every PE in OpenSHMEM has symmetric memory associated with it. This area can be visualized as an area in memory which has identical structure on all processing elements. The location may or may not be identical, but the placement of the symmetric variables, allocated and managed by the library, is identical on each PE. Hence, address calculation for symmetric variables on other PEs becomes a trivial task. This enables OpenSHMEM to do *one-sided* communication, where the transfer of data between symmetric variables can be achieved without the involvement of the remote PE.

Listing 2.1: A C program to illustrate different symmetric variable categories in OpenSHMEM.

```
1  int aglobal;           /*symmetric variable*/
2  void main( ){
3    ...
4    int me, npes;        /*local variables*/
5    static int astatic;  /*symmetric variable*/
6    int *x;
7    int y;               /*local variable*/
8    ...
9    start_pes(0);
10   ...
11   x = (int *) shmalloc(sizeof(int));   /*dynamic symmetric allocation*/
12   ...
13   shmem_int_put(&astatic, x, 1, (me+1)%npes);
14   ...
15   shmem_int_get(&y, &aglobal, 1, (me+1)%npes);
16   shmem_barrier_all();
17   ...
18   shfree(x);
19   ...
20   return 0;
21 }
```

In code Listing 2.1, at line 1, the variable *aglobal* is a global variable and hence *symmetric*. At line 9 the OpenSHMEM library is initialized and all subsequent OpenSHMEM library calls must follow this call. The variable *astatic* declared in line 5 is also symmetric as per the OpenSHMEM Specification 1.0. A *symmetric* variable x is allocated in line 11 using the dynamic memory allocation call *shmalloc*. The symmetric variable *'x'* is allotted the requested amount of memory at the **same memory off-set** on the *symmetric heap* on each PE.

This mechanism facilitates fast remote address calculation at the source PE as the remote symmetric variable's address is computed by adding the base address of the symmetric heap on the target and the local offset corresponding to the same *symmetric variable*. This is managed internally by the OpenSHMEM library. An OpenSHMEM library implementation may choose to have symmetric variables at the same address (versus same off-set) to speedup target destination address calculation. Symmetric variables may contain different values on all PEs.

3. **Active Set**

   Some routines in OpenSHMEM involve participation of a sub-set of PEs. An *active set* is a logical grouping of PEs [54].

OpenSHMEM implementations expect that the programmer abides by these semantics. Any deviation from these results in undefined behavior which is implementation specific.

2.2.3.2 Functionality provided by OpenSHMEM library API

An overview of the OpenSHMEM operations is described below:

1. **Library Setup and Query**

   (a) Initialization: The OpenSHMEM library environment is initialized. No OpenSHMEM library calls may precede this call.

(b) Query: The local PE may get number of PE running the same application code and its unique integer identifier (starting at 0).

(c) Accessibility: Provides PE and data accessibility information.

2. **Symmetric Data Object Management**

(a) Allocation: Collective allocation of symmetric data by library provided routines.

(b) Deallocation: Collective deallocation of symmetric data by library provided routines.

(c) Reallocation: Collective reallocation of symmetric data by library provided routines.

3. **Remote Memory Access**

(a) Put: Write operation to the symmetric data object on a remote PE.

(b) Get: Read operation of symmetric data object on a remote PE to a data object (local or symmetric) on a local PE.

4. **Synchronization and Ordering**

(a) Fence: Ensures per-PE ordering of remote memory update operations.

(b) Quiet: Ensures completion of remote memory update operations.

(c) Barrier: Collective operation to synchronize and ensure completion of all remote and local updates.

5. **Collective Communication**

   (a) Broadcast: One to all copy from *root* PE to one or more remote PEs (not including itself) using symmetric source and target data objects.

   (b) Collection: Collective operation to achieve concatenated symmetric objects contributed by each of the PE, in another symmetric data object.

   (c) Reduction: Collective operation to get the result of associative binary operation over elements of the specified symmetric data object.

6. **Atomics**

   (a) Swap: Local PE reads the old value of the symmetric data object on a remote PE and copies a new value to it.

   (b) Increment: Local PE adds 1 to the symmetric data object on the remote PE.

   (c) Add: Local PE adds specified value to the symmetric data object on the remote PE.

   (d) Compare and Swap: Local PE reads the old value of the symmetric data object based on a value to be compared and copies a new value to the symmetric data object on the remote PE.

   (e) Fetch and Increment: Local PE reads the old value of the symmetric data object on the remote PE and adds 1 to the symmetric data object on the remote PE.

   (f) Fetch and Add: Local PE adds specific value to the symmetric data object on the remote PE and copies the old value.

7. **Mutual Exclusion**

   (a) Set Lock: Allows exclusive access to the region bounded by the symmetric *lock* variable.

   (b) Test Lock: Any PE may test the symmetric *lock* variable for availability.

   (c) Clear Lock: The PE which has previously acquired the *lock* releases it.

8. **Data Cache Control**

   (a) Mechanisms to exploit the capabilities of hardware cache if system has non-coherent cache.

Chapter 3

Related Work

3.1 Translation of PGAS Languages

All PGAS languages and libraries are lowered by the compiler to some intermediate representation for optimizations and finally replaced by a combination of runtime calls that perform the required operation. Since all of them go through a similar process, optimizations implemented for OpenSHMEM are applicable for other PGAS languages and libraries. These can be applied either by following the changes to the compiler infrastructure as a prototype and implementing them for individual language/library, or by targeting OpenSHMEM as a possible intermediate representation.

Figure 3.1 shows the overall structure of the Berkeley UPC Compiler, which is divided into three main components: the UPC-to-C translator, the UPC runtime

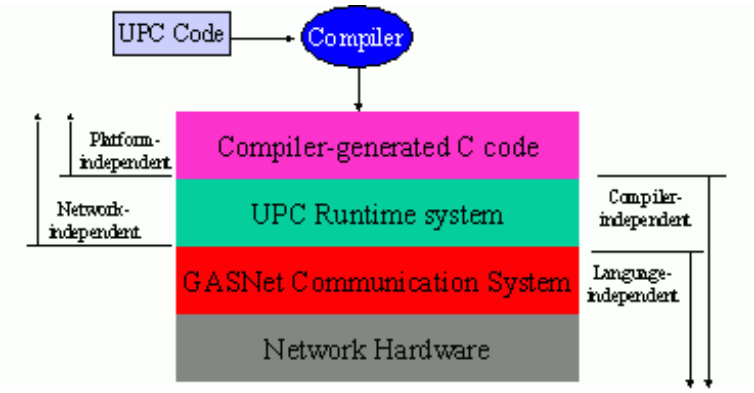


Figure 3.1: Berkeley's UPC Compiler [38]

system, and the GASNet communication system. The different phases of compilation allow the translation of UPC program code to C code which is platform independent. The UPC-related parallel constructs are converted into calls to the runtime library. The C compiler on the target machine translates the C code and links it to the runtime system. The runtime system handles the UPC specific tasks such as thread generation, allocation and management of shared data, etc. The Berkeley UPC [38] runtime relies on the GASNet communication library to perform its data allocation and communication. Main advantages of such a scheme is that GASNet provides an uniform interface for low-level communication primitives on a variety of networks, by directly accessing the individual hardware reduces communication overhead, and eliminates function call overhead (by using techniques such as inlining).

Berkeley UPC Compiler targets C as the intermediate representation and hence is available on commonly used hardware platforms that have an ANSI-compliant C compiler. The modular design of the compiler allows for aggressive optimization of the intermediate C output by the backend C compiler. The UPC-to-C translator

uses UPC-specific knowledge about shared memory access patterns to perform communication optimizations. This is helpful since application performance using UPC depends upon the number of accesses to shared data and how they are handled.

A similar flow is observed for Titanium, where compiler translates Titanium code into C that is then lowered by the compiler to assembly code [74]. We look at a popular open source CAF implementation (OpenUH CAF [25]) as another example of how PGAS languages are supported and translated by compilers.

The OpenUH compiler is based on the Open64 [15] compiler. Open64 is a high performance compiler that can generate binary codes for a range of architectures like, the Intel IA-64, Intel IA-32e, and AMD X8664. Open64 supports Fortran 77/95 and C/C++, as well as the shared memory programming model OpenMP. The Open64 compiler infrastructure is most recognized for its ability to perform high quality interprocedural analysis, data-flow analysis, data dependence analysis, and array region analysis [25].

To support CAF, OpenUH compiler's front-end is modified to generate intermediate code (IR), and during the translation phase the compiler analyzes the IR and translates it to produce appropriate communication code. The runtime library provides the actual implementation of the required communication primitives. Different optimizations are possible on different IR representations. Figure 3.2 shows the framework of the OpenUH CAF compiler and runtime system.

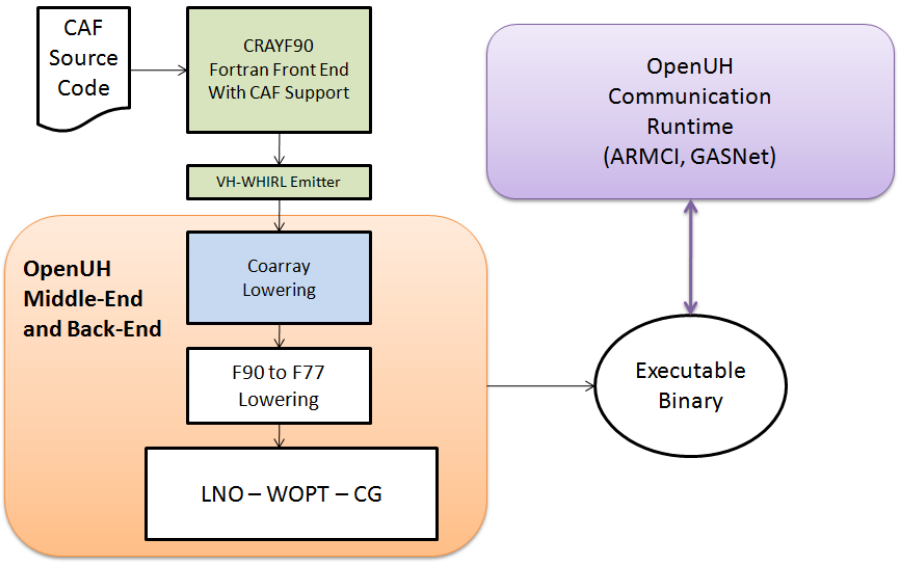


Figure 3.2: OpenUH CAF Framework [25]

3.2 Compiler-based Tools and Synchronization Analysis

Static analysis tools like Broadway [29] (built on top of the C-Breeze compiler [39]), aim at verifying programs by separating the code generation functionality of compilers from the awareness of the semantics of libraries. This is facilitated using annotations, which are used to convey the dependency relationships for code verification within the compiler. This means that the programmer needs to explicitly translate into annotations, the semantics for every single library function used. An incorrect representation of such dependencies may lead to excess false positives or worse, false negatives. To overcome this drawback we set out to embed the awareness of the OpenSHMEM library within the compiler infrastructure.

Concurrency analysis has a myriad of applications. Usually it is used as a stepping stone to facilitate complex analysis for process-level parallelism [34] of a program. This is done by identifying process segments (application code executed by a particular process) and then grouping the process segments into process phases that contain concurrent regions [36]. Such analysis avoids the problem of having to identify textually unaligned barriers by assuming that barriers are identified via unique barrier variables. Hence matching unaligned barriers and finding concurrent regions becomes trivial.

One of the first works to verify program synchronization patterns and the rules that govern the synchronization sequences was done in [6] for Split-C. Their work analyzes the effects of *single valued expression* on the control flow and concurrency characteristics of the program. Their methodology simplifies the identification of unaligned barriers and single valued variables by using the *single* keywords for annotating the named barriers (barriers statements that correspond to the same barrier).

An active testing framework called UPC-Thrille [56] was developed to find concurrency bugs in distributed memory programs. This framework works in two phases the race prediction phase and the race confirmation phase. In the first phase, an imprecise dynamic analysis finds program statements that could potentially lead to a race condition during the execution of the program. The next phase, called the race confirmation phase, tries to assert that the race condition that was detected actually exists for each pair of statements reported in the first phase. This involves executing the application at least two times. When the second phase successfully manages to get the individual threads to access the same memory location, and at least one of

the accesses is a write, data race is confirmed. For large scale applications requiring huge computational and data resources, execution for testing and profiling purposes may not be feasible.

Another concurrency analysis is proposed in [40] for shared memory programs using OpenMP programs based on *phase partitioning*. Their analysis is intra-procedural and relies on the OpenMP constructs to compute non-concurrency relationship across statements. The drawback of this method is that it does not account for multi-value expressions and synchronization across textually unaligned barriers, which is important to understand concurrency relationship in communication libraries like OpenSHMEM.

A concurrency analysis method for PGAS languages, like Titanium [41], with textually aligned barriers is proposed in [36]. It discusses an inter-procedural algorithm that efficiently computes the set of all concurrent statements by first modifying the Control Flow Graph (CFG) and then performing a modified depth first search to ascertain the pairs of concurrent expressions.

Work in [76] uses similar concepts as those used in [36] for concurrent analysis for shared memory programs written in OpenMP. The primary difference is that for MPI and OpenMP there is a need to match textually unaligned barriers first [76]. Consequently matching barriers that synchronize together are discovered and then a partitioning based on global synchronization calls is performed. An improvement of this work over [36] is that they consider the effects of unaligned as well as implicit barriers implied by OpenMP constructs and use this information to extend the CFG.

Since synchronization analysis is an essential step towards concurrency analysis for OpenSHMEM programs, we also look at [75] and their methodology to match unaligned barriers for MPI programs. They evaluate the different concurrent paths the processes in the MPI program may take (using multi-value conditional and barrier expression analysis) and check that each process encounters an equal number of barriers. Our approach to barrier matching is similar but we have to account for two different barrier calls provided by the OpenSHMEM library API, which makes the problem more complex. Our approach not only confirms if the barriers are matched but also can report to the user the exact location within the program where there is a possible mismatch. We also extend this analysis to detect possible optimizations in the parallel programs that are specific to the semantics of the OpenSHMEM library API.

Concurrency analysis for other parallel programming languages, such as X10 [43, 46], Ada [35, 14], and Java [71, 42] are also relevant.

3.3 Compiler and Runtime Optimizations

Compiler-based optimizations to applications is a widely studied area for improving application performance by relaxing memory consistency requirements and improving computation-communication overlap while maintaining program correctness. For UPC, research for communication optimizations by compilers for fine-grained UPC applications [22] and runtime system to search and exploit available overlap present at execution time [33] are well researched areas. Compiler-based optimizations like

redundancy elimination for shared pointer arithmetic (by replacing repeated computations of the same variable by temporaries), split-phase communication for reads (data is communicated as early as possible and completion is checked before the use of such data), and coalescing of communication calls (combining smaller data transfers into one large transfer) have been looked into. But like all static analyzes, there are inherent drawbacks, such as: data sizes may be available only at run-time, non-deterministic communication pattern with change in number of processes, or the compiler is unable to precisely determine the overlap due to multiple hardware targets. Here runtime analyzes tools like HUNT [33] may be more effective. But this would involve multiple runs of the application to collect information available only at runtime, thus using computational resources that may not be appealing to application programmers.

In the shared memory domain, compiler optimization techniques for explicit parallel programs using OpenMP [64] [45] have been researched. To enable optimization across threads, data-flow analysis techniques to model interactions between threads have been researched. Structured description of parallelism and relaxed memory consistency in OpenMP make the analyses effective and efficient. Algorithms for reaching definitions analysis, memory synchronization analysis, and cross-loop data dependence analysis for parallel loops have been researched to be able to provide aggressive compiler optimizations for software-implemented coherence schemes to obtain good performance on shared memory platforms.

Chapter 4

OpenSMEM Analyzer

Compilers are not aware of the parallel semantics of the OpenSHMEM library and they treat the library API like a black box, thus hindering optimizations. The OpenSHMEM analyzer (OSA) [55] is the first effort to develop a compiler-based tool aware of the parallel semantics of an OpenSHMEM library. OSA provides information about the structure of the OpenSHMEM code and semantic checks to ensure the correct use of the symmetric data in OpenSHMEM calls. The OpenSHMEM Analyzer uses intra-procedural and inter-procedural analysis information to report non-adherence to the semantics of the OpenSHMEM library specification, which can be an indispensable assistant for verifying correctness of OpenSHMEM programs.

The OpenSHMEM Analyzer has been integrated to OpenUH [20], at the University of Houston; which supports most of OpenMP 3.0 and Co-Array Fortran (Fortran 2008 Model). The front-end of the compiler accepts Fortran 77/90 and C/C++. OpenUH is a modular and robust research compiler with support for the IA-64,

IA-32e, and Opteron Linux ABI platforms. Major functional parts of OpenUH are the front-ends, the inter-language inter-procedural analyzer (IPA), and the back-end. The back-end is further sub-divided into the loop nest optimizer(LNO), global optimizer (WOPT), and code generator (CG). OpenUH, like Open64, uses five levels of Intermediate Representations (IR) in its back-end, called WHIRL. Each phase within the OpenUH compiler performs *lowering* of WHIRL (starting from Very High WHIRL) to finally get the executable in machine instructions. This facilitates different analysis and optimization on different levels of WHIRL.

For OSA, our goal is to perform the analyses at program-scope, and not just within procedure boundaries, the analyses are performed at two distinct stages; intra-procedural (IPL phase) and inter-procedural (IPA-Analysis phase). All analyses are performed at the High WHIRL IR level. The extension to the Data Flow Analysis includes the verification of the initialization of symmetric pointers and the status of the arguments. Checks are also performed after the IPA-Link stage following optimizations such as constant propagation, array-section propagation and inlining analysis. The intra-procedural Alias Analysis of symmetric buffers is performed using context-insensitive and flow-insensitive Steensgard-Alias Classification [68].

We extend the existing OSA framework to provide an in-depth synchronization analysis based on the control flow of the OpenSHMEM program. Since the semantics of the OpenSHMEM library function calls are relevant between global synchronization calls, the first step is to detect the possible concurrent execution paths of an application based on the synchronization structure of the application. Concurrency in an SPMD application results from distinct paths of execution which are effected

by providing conditions that evaluate to different values on different PEs. Such expressions within the conditionals are called *multi-valued expressions* [69, 40, 44, 7] and the particular variable used within the expression that causes this phenomenon is the *multi-valued seed*. Our framework uses the concepts of multivalued seeds and multivalued expressions to build a multi-valued system dependence graph in the context of OpenSHMEM. This graph is used to build a *barrier-tree* which represents the synchronization structure of the entire application. We implement the critical parts of the framework proposed in [59] and provide insights into the the practical aspects of detection of *multi-valued* expressions by identification and tracing of *multi-valued* seeds. Our analysis is performed within the IPA phase using the High WHIRL to help us build inter-procedural control flow and data flow graphs while preserving the high level constructs of a program, such as DO_LOOP, DO_WHILE, and IF, which are critical for multi-valued analysis.

We merge information available from different phases of the compiler and present the results of our analysis in the form of two graphs: a system dependence graph at the statement level clearly marking the control and data dependences across statements in the program, and the *barrier-tree* where the leaves of tree are OpenSHMEM collective synchronization calls (*shmem_barrier* and *shmem_barrier_all*) and the nodes are operators that represent the possible concurrent control flow within the program. This visual aid provides the programmer with necessary information to verify if there is congruence in the intent of the program with its actual execution structure. The OSA is based on extending existing OpenUH compiler technology to report errors and other useful feedback accurately in context of OpenSHMEM.

By providing other useful information at compile time the programmer can analyze the program structure **before** execution, thus preventing resource wastage. Currently the tool is tested for C programs only, but with minor modifications it can be extended to verify Fortran programs that use the OpenSHMEM library API.

4.1 OSA Infrastructure

Figure 4.1 shows the different stages within the compiler and the shaded region are the phases where OSA tool performs most of its analysis. Since we need the data flow information, alias analysis and the control flow information for each individual procedure we build our analysis at the local inter-procedural phase. At this phase all analyses is performed on the High WHIRL IR where variables and control flow statements are preserved and can be easily mapped to the source code and the control flow is fixed [15]. We used the DU-manager, alias manager, and control flow analysis data structures to build our system dependence graph to perform multi-valued analysis.

4.2 Semantic Checks

The current OpenSHMEM specification does not discuss detailed behavior of implementations of library calls that are used incorrectly. The individual OpenSHMEM libraries could choose different ways of handling such situations. While resilient implementations mask these, others may cause the application to crash or produce

incorrect results. Also, C, C++, and Fortran compilers cannot enforce the use of the local and symmetric variables in specific arguments of OpenSHMEM calls. This could lead to unintentional mixing of symmetric and automatic variables. To maintain reproducibility and portability of OpenSHMEM applications, the OpenSHMEM Analyzer can discover instances of incorrect usage and report them back to the application programmer through compile time static analyses. It keeps track of the storage classes of symmetric variables to ensure that their data are stored in the global or static sections of the code. It also performs type checks for symmetric variables used in 32-bit or 64-bit OpenSHMEM calls. The type-checks can also detect out-of-bound array accesses (in cases where the length of the access can be computed statically). These checks are performed after inter-procedural constant propagation. Other common errors that OSA is capable of detecting are the absence of or multiple calls to the initialization of the OpenSHMEM library using start_pes().

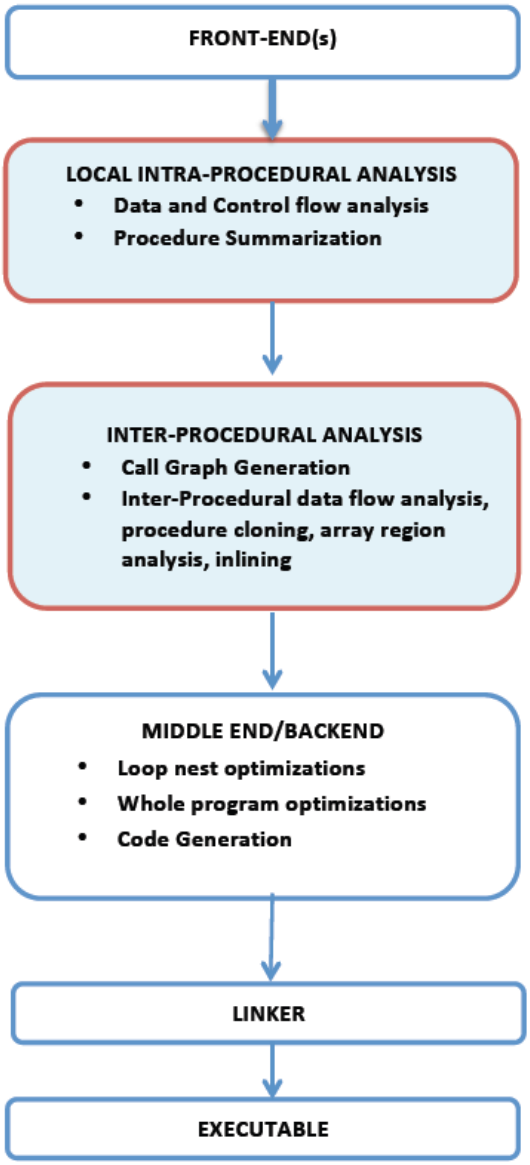


Figure 4.1: OSA analysis (shaded blocks) within the OpenUH compiler.

Chapter 5

Synchronization Analysis Framework

In this chapter we discuss the common programming mistakes that are encountered during an application development using the OpenSHMEM library. The OSA can help catch such conditions at compile time thus reducing development and debugging times. We discuss the different steps within the Synchronization Analysis framework of OSA here. The synchronization analysis takes place in two major steps; concurrency analysis and discovering synchronization phases. Each step in turn has to combine results from different analysis performed by the compiler to get meaningful information relevant to OpenSHMEM's synchronization model.

5.1 Common Programming Mistakes using OpenSHMEM

Consider the code in Listing 5.1. In the code example between barrier b1 and b2, (say) PE 1 executes an atomic statement (*shmem_int_finc*) and the data transfer statement (*shmem_int_put*). The atomic statement on line 6 increments the *symmetric variable target* on PE 2, while on line 5, PE 1 does a *put* on the same variable on PE 2.

Listing 5.1: Violation of OpenSHMEM semantics

```
1    start_pes(0);
2    ...
3    shmem_barrier_all();   //b1
4    ....
5    shmem_int_put(&target, &source, 1, (me+1)% npes);
6    shmem_int_finc(&target,(me+1)% npes);
7    ...
8    shmem_barrier_all();   //b2
9    ...
```

Since both these operations may happen simultaneously and the OpenSHMEM specification guarantees atomicity of atomic operations with respect to other atomics *only*, this code leads to non-deterministic results depending on which operation is executed first. The programmer may easily miss this class of errors leading to an application with irreproducible results. However, if the put operation is used in another code phase (after a barrier) with no atomics accessing the same variable in the code phase, it is correct to do so.

Listing 5.2: Deadlock due to missing synchronization.

```
1   ...
2   if(my_pe%2 == 0){        //then
3     ..
4     shmem_barrier_all(); //b1
5   }
6   else{
7     ..
8   }
9   ...
10  shmem_barrier_all();   //b2
11  ...
```

Consider the code in Listing 5.2. In the code we see that due to the *if* statement on line 2 all even numbered PEs will execute the *then* part and the odd numbered PEs will execute the *else* part. In OpenSHMEM the *shmem_barrier_all* function call must be executed by all PEs at the same point in their execution, so to be semantically correct there *must* be a *shmem_barrier_all* in the else block as well. In the above code the even PEs will wait at barrier b1 for all odd PEs while the odd PEs would have reached b2 and be waiting for even PEs thus resulting in a deadlock.

Consider the code in Listing 5.3. At line 8, value of *symmetric variable* ***src*** on PE 0 is updated and on line 12 PE 1 does a one-sided copy of the value of ***src*** from PE 0. Since no synchronization exists between the two actions, OpenSHMEM does not guarantee that PE 1 will get the latest value of ***src*** which may lead to the propagation of the old value of ***src*** and result in wrong and/or inconsistent program outcome.

Listing 5.3: Incorrect program behavior due to un-matched synchronization.

```
1  int src;
2  void main( ){
3      ....
4      start_pes(0);
5      ...
6      if(my_pe == 0){
7        ..
8        src = foo(src);
9        ..
10     }
11     if(my_pe == 1)
12       shmem_int_get(&target,&src, 1, 0);
13     ...
14     shmem_barrier_all();
15     ....
16     return 0;
17 }
18
19 int foo(int x){
20     x = x * 1.085 +10;
21     return (x);
22 }
```

In the code in Listing 5.4 the value of *symmetric variable* ***target*** is initialized to 10 on PE 1 at line 3. Later the value is updated by PE 0 (line 7) by the value in ***src***. At line 12, PE 1 waits on the value of ***target*** to change. Since the local and remote updates have been affected by the *shmem_barrier* on line 10, the point-to-point synchronization call is not needed, and could be eliminated.

Listing 5.4: Potential performance degradation due to excessive synchronization.

```
1    ...
2    if(me == 1)
3      target = 10;
4    shmem_barrier_all();
5    if(my_pe == 0){
6      ..
7      shmem_int_put(&target,&src, 1, 1);
8      ..
9    }
10   shmem_barrier_all();
11   if(my_pe == 1){
12     shmem_int_wait(&target, 10);
13     ..
14   }
```

5.2 Concurrency Analysis

In OpenSHMEM's execution model, all PEs execute the same program but may take different paths through the program based on some implicit or explicit conditions set by the programmer. Understanding the concurrent paths possible during execution is the first step for any advance analysis. In an SPMD application conditionals that result in different values on different PEs lead to concurrency. Variables that cause this phenomenon are called *multi-valued seeds* [76] . The first step for concurrency analysis is the identification of multivalued seeds.

OpenSHMEM Library Variable	Classification	Reason
_num_pes	*npes*	Single-valued
_my_pe	*me*	Multi-valued
PUT (elemental, block, strided)	*target*	Multi-valued
GET (elemental, block, strided)	*target*	Multi-valued
ATOMICS (fetch and operate)	*target*	Multi-valued
BROADCAST	*target*	Multi-valued
COLLECTS (fixed and variable length)	*target* array	Single-valued if *active set* = npes else Multi-valued

Table 5.1: **Effect of OpenSHMEM library calls on program variables**

5.2.1 Identification of Multivalued Seeds

Based on the rules stated in [7] there exist generic rules governing the multi-valued property of different variables used within an application. For example, uninitialized data structures and variables referred via pointers are marked as multi-valued. We extend certain assumptions about the expressions that generate from a known single-valued or multi-valued seed based on the OpenSHMEM programming model. We modify the classification scheme for *multi-valued* seed in presence of OpenSHMEM calls and their treatment of different program variables as shown in Table 5.1.

By analyzing the type of a variable and how it is modified (for example, if it is defined via a multi-valued OpenSHMEM call) we can then classify it as a *single-valued* or *multi-valued* seed. For example, the return value for the OpenSHMEM call *_my_pe()* is unique for every PE and hence is multi-valued. In contrast, *_num_pes()* returns the same value throughout the program for all PEs and hence the return

value (and the variable associated with it) is considered single-valued. Likewise, other OpenSHMEM library calls have an impact on the variable they modify. Generally, all PE-to-PE operations that modify data cause the variable to become *multi-valued*, while collective operations that modify *target* variables on **all** the PEs cause the target to be single-valued.

For every definition of a program variable there is a use-list associated with it and a set of statements that may directly or indirectly (via aliases) use the variable. The subsequent use of a variable after its declaration is analyzed by the compiler and saved in a specialized data structure called the Def-Use (D-U) chain within the compiler's back-end. A multi-valued seed may affect the value of other program variables or may only alter the control flow. The detection of resulting *multi-valued* variables is done by propagating the multi-valued seeds using the D-U chains generated by the backend. We append this information along with the control dependencies extracted from the control flow graph generated by the compiler . Combining the information from the the control flow graph and data flow, gives the *system dependence graph* [67].

5.2.2 Construction of the System Dependence Graph

We look at the example code listing in 5.5 to see how creating a logical system dependence graph enables extraction of concurrent paths within an application. Within the compiler control flow graph is a sequence of basic-blocks with dominator relationship. For example, if a basic-block BB5 cannot be reached without the control

passing through BB3, BB3 dominates BB5. Along with the dominator information we also need to find the *dominator frontier* information, which essentially gives the basic-block that immediately precedes the current basic-block.

Listing 5.5: Example OpenSHMEM C code with barrier synchronization

```
1  int main(int argc, char *argv[]){
2     ...
3     if(me==0){
4        shmem_barrier_all();
5        int temp = x+y;
6        shmem_barrier_all();
7     }
8     else {
9       if(me==1){
10          shmem_barrier_all();
11          old = shmem_int_finc (&y, 0);
12          shmem_int_sum_to_all(&y,&x,
13                 1,1,0,npes-1,pWrk,pSync);
14           x= x+10;
15          shmem_int_get(&y,&y,1,0);
16          shmem_barrier_all();
17         }
18      else{
19          shmem_barrier_all();
20          shmem_int_sum_to_all(&y,&x,
21                 1,1,0,npes-1,pWrk,pSync);
22          x=y*0.23
23          shmem_barrier_all();
24         }
25    ...
26    }
27   return 0;
28 }
```

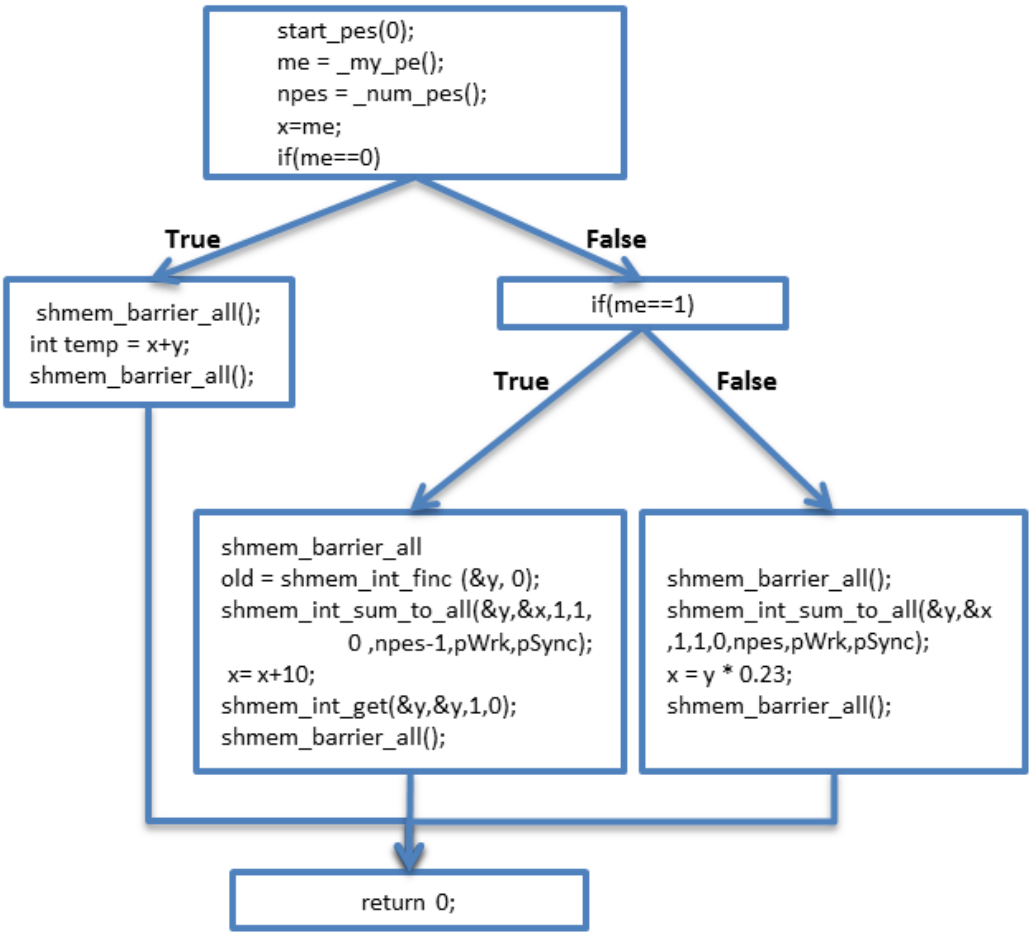


Figure 5.1: Control Flow Graph for code listing 5.5.

Figure 5.1 shows the CFG of the code listing in 5.5. Each basic-block has a single point of entry and exit. We append data flow information by finding *reaching definitions* of variables. From the compiler generated information for data flow for individual variables, we find definitions that reach a specific point in a program without being redefined at any intermediate point. We annotate the IR with this information. Merging this information we get a *system dependence graph* [32] shown in Figure 5.2 .

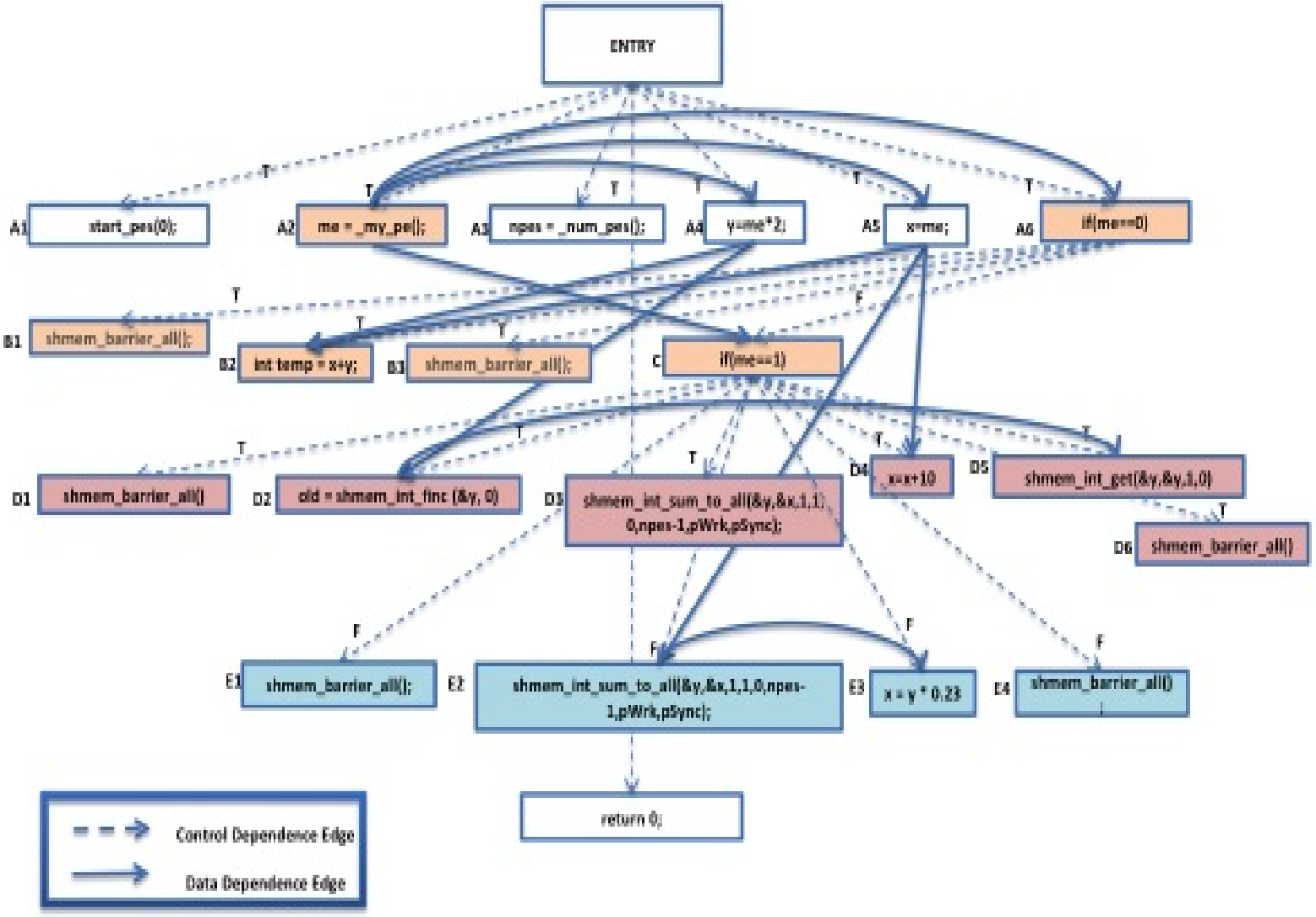


Figure 5.2: System Dependence Graph for code listing 5.5.

Based on the CFG and the outcome of the conditional statements each control edge is either marked *true* (T) or *false* (F). The data dependence edges are marked with bold arrows. Programming slicing is defined as a decomposition based on data flow and control flow analysis of the application [72] and can be viewed as a way to identify the reach of these multi-valued seeds. If we take a forward slice of the sample program based on the *multi-valued* PE number *me* at A6, then we get either A6-B1-B2-B3-C or A6-C-D1-D2-D3-D4-D5-D6 or A6-C-E1-E2-E3-E4 depending on the value of *me*. These slices help us identify the *multi-valued* conditionals in the program by finding the points at which the execution paths diverge.

5.3 Discovering Synchronization Phases

A synchronization free path between consecutive synchronization calls on the same execution path is defined as a *synchronization phase*. Identification of these phases is critical as before and after them the programming model promises consistent view of the memory to all executing PEs. Since we are focusing on synchronization analysis we need to identify the different synchronization constructs within a given OpenSHMEM application that a PE may encounter. Since global synchronization primitives in OpenSHMEM also provide a guarantee of memory consistency, detecting these phases allows us to apply different optimizations; since some are relevant only within a synchronization phase and some may be applied across synchronization phases.

By knowing the property of every conditional within the application (single-valued or multi-valued) we use the system dependence graph to get a logical slice of

the program statements that a PE may encounter and then register the synchronization structures on this path. OpenSHMEM specification defines two global synchronization calls *shmem_barrier* and *shmem_barrier_all*. These require participation of more than one PEs, hence these operations are *collective* in nature.

5.3.1 Synchronization Semantics

Collective synchronization is provided by *shmem_barrier* and *shmem_barrier_all* (over a subset of PEs and all PEs respectively) in OpenSHMEM. A barrier call guarantees synchronization as well as completion of all pending remote and local OpenSHMEM data transfer operations and leaves the memory in a consistent state. A *shmem barrier* is defined over an ***active set***. An *active set* is a logical grouping of PEs based on the triplet, namely, ***PE_start***, ***logPE_stride***, and the ***PE_size*** [54].

5.3.2 Textually Unaligned Barriers

Listing 5.6: C code with unaligned barriers

```
1 if(_my_pe() % 2 == 0){
2   ...
3   shmem_barrier_all();
4 } else{
5   ...
6   shmem_barrier_all();
7 }
```

Listing 5.7: C code with aligned barrier

```
1 f(_my_pe() % 2 == 0){
2   ...
3 } else{
4   ...
5   ...
6 }
7 shmem_barrier_all();
```

OpenSHMEM allows for unaligned barriers, both the code listings, Listing 5.6 and 5.7, are equivalent and valid as per OpenSHMEM Specification 1.0. This makes it easy to miss synchronization errors and may lead to unintended execution patterns or worse, dead lock. We now look into the details of discovering the synchronization structures and analyses that helps define relationships between these synchronization structures.

5.3.3 Barrier Detection and Generating Barrier Trees

We extract the synchronization structure in a tree structure by iterating over the IR generated by the compiler. By recording the barriers (both *shmem_barrier_all and shmem_barrier*), their relative position, and the control flow between them we generate a barrier tree for the entire program. The barriers are leaf nodes and the *operators* are the nodes of the tree and the edges connect operators to other operators or barrier synchronization calls. For simplicity both *shmem_barrier_all and shmem_barrier* are represented by b, and are appended by integers representing the relative position of the barrier statement with respect to other barrier statements. For example, if there are two barrier statements represented by bi and bj, such that j > i, indicates that bi is encountered before bj when traversing the CFG generated by the compiler's backend.

Like regular expressions, barrier trees use three types of operators: concatenation ($\cdot$), alternation ($|$), and quantification ($*$) [37]. Table 5.2 gives the rules that govern the barrier expression generation. It is important to note that if the result of a

Placement of barriers	Operator used	Result
b1 followed by b2	$\cdot$	b1 $\cdot$ b2
if(*single-valued conditional*) b1; else b2;	$\mid$	b1 $\mid$ b2
if(*multivalued-valued conditional*) b1; else b2;	$\mid^c$	b1 $\mid^c$ b2
for(n times) b;	$\cdot$	b1$\cdot$ b2 $\cdot$ ... bn

Table 5.2: **Rules for annotating nodes in the Barrier Tree**

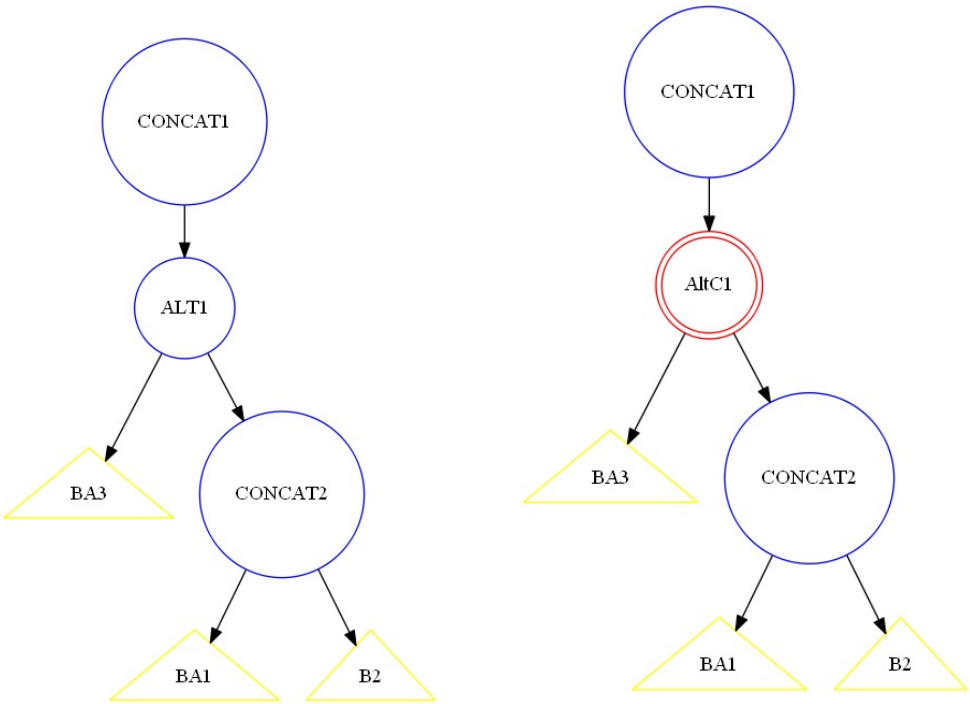
quantification operation can, at times, be statically non-deterministic, and we may not be able to compute the barrier-expression in terms of the exact number of barriers encountered for such a program. Our analysis does not handle these cases and leaves it to the user to verify program's synchronization structure from the graphical representation we provide for easy visual assessment. Additionally, we borrow the operator $\mid^c$ from [75] to indicate the operator *concurrent alternation*. This operator indicates that the different execution paths diverge from a *multi-valued* conditional.

We use the multi-value analysis saved in the system dependence graph to distinguish concurrent paths that may be present with a barrier tree. In our barrier tree representation the main function entry is indicated by the concatenation (CONCAT) as root. All operators are appended by a number which indicates their relative position of occurrence in the program's control flow.

Listing 5.8: OpenSHMEM example to explain concurrent alternate paths of execution.

```
1  int main(){
2      if( ){
3        shmem_barrier_all(); //b1
4             ..
5        shmem_barrier_all(); //b2
6      }
7      else {
8        shmem_barrier_all(); //b3
9             ..
10     }
11        return 0;
12  }
```

All barriers (barrier_all = BA, barrier=B) have independent numbering based on breadth first traversal ordering. This means that barriers in the *if-then* branch will have lower numeric labels than the *if-else* branch. Other operators are represented as follows: quantification (QUANT), alternation (ALT), and alternate concurrent (AltC). For example, the code in Listing 5.8 would evaluate to the barrier expression: (b1.b2) | b3 and would be represented by our compiler analysis (without multi-value information) by a barrier-tree in Figure 5.3(a). Purely based on Figure 5.3(a) the programmer has no way of knowing the different paths of execution that may be possible. Consider two possible scenarios, if the first **if** conditional in line 2 resulted in the same value on all PEs, then all PEs would either encounter barriers *b1-b2* **or** *b3*. But if the same conditional resulted in different values on different PEs, then some PEs would encounter barriers *b1-b2* and others would encounter *b3*, which is a obvious stall situation caused by un-matched synchronization calls. This is indicated



(a) Barrier-tree without multi-valued analysis

(b) Barrier-tree with multi-valued analysis

Figure 5.3: Barrier trees generated by OSA for code Listing 5.8

by AltC (alternate concurrent) label in Figure 5.3(b). We augment the multi-value analysis to this providing a more meaningful representation of the program structure. Figure 5.3(b) depicts the barrier tree for the second scenario discussed above.

5.3.4 Matching Synchronization Structures

Since the synchronization primitives in OpenSHMEM are textually unaligned, we need to match the different barrier statements to accurately determine the synchronization phases. Correct barrier matching is possible only if the barriers are well-matched [75]. We check the correctness of the synchronization structure of the program by applying the semantics of the OpenSHMEM synchronization calls on the barrier tree, starting with the smallest sub-tree. This is non-trivial for OpenSHMEM as there are additional considerations for the shmem_barrier call; since it has additional parameters. So for our analysis, not only the number of barrier statements encountered in concurrent alternating paths must match but the sequence of occurrence of the shmem_barrier and shmem_barrier_all calls must also match. Additionally, after the first two requirements are met, we need to match the arguments of the corresponding shmem_barrier calls to positively state that no synchronization imbalance exists. After verifying that all individual synchronization sub-trees are well-matched, we can be sure that there is no imbalance due to missing or extra synchronization call in the application [75].

We move to synchronization or barrier matching after verifying that all synchronization statements encountered on concurrent alternate paths have matching

synchronization primitive. At this phase in the analysis, we also generate the synchronization structure of the entire application and with the help of the Graphviz tool [26], so that the programmer may visually review the application's synchronization structure. Barrier Matching is done by first generating the legal sequences of barriers by a constraint driven pre-order depth-first-search of the barrier tree based on the operators discussed above.

Claim 5.1. In the presence of a alternate concurrent path and an absence of quantification sub-trees, there will be at least two barrier sequences with the same sequence of barrier statements.

Proof: Since alternate concurrent paths imply that some PEs will execute different set of statements concurrently. For the application to be well matched the number, sequence and ordering of the barrier statements has to be the same. Since we have already ensured that the tree is well-balanced and does not have quantification sub-trees, there will exist at least two barrier sequences that have the same order of barrier statements and are of the same length.

We validate the barrier sequences that have at least one other barrier sequence of the same length and find corresponding barrier statements that occur at the same position within a sequence. This gives us the matching barriers in the presence of unaligned barriers in an application. We discuss the algorithm via an example. Consider the skeleton code in Listing 5.9 and its barrier tree representation depicted in Figure 5.4.

Listing 5.9: Skeleton C code with barrier synchronization calls.

```
1  main(){
2    if(mv1){
3      b1;
4      b2;
5    }
6    else{
7      if(mv2){
8        b3;
9        b4;
10     }
11     else{
12       b5;
13       b6;
14     }
15   }
16   ...
17 }
```

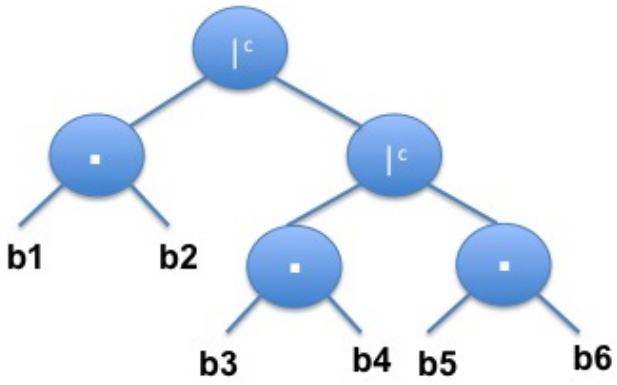


Figure 5.4: Barrier Tree for Listing 5.9.

An operator-driven pre-order depth-first-search of the barrier tree will give { b1.b2, b3.b4, b5.b6 } barrier sequences. By matching the sequences left to right (increasing order of index), we get the matching barriers set for each barrier. Here b1

corresponds to b3 and b5, similarly, b2 corresponds to b4 and b6. For more complex program flows an additional step for elimination of spurious barrier sequences is required, which is trivial, since they will not have an exact match and by following Claim 5.1 we can discard such barrier sequences. Hence the synchronization phases b1-b2 correspond to b3-b4 and b5-b6. The time complexity of depth first search without repetition is $O(|E|)$, where E is the number of edges, our analyses adds additional time based on the number of concurrent branches available. Since all analyses is done on the data structures that already exist in the compiler's back end, we are not adding any extra space overhead.

5.4 Optimizations

The final step in the synchronization analysis is to provide hints to the user about possible optimizations that they can incorporate in their applications that may positively affect the performance. At this point potential optimizations are suggested to the user as HINTS. The final goal is to effect safe transformations without user intervention. The major optimization classes we see the potential for are; Remove excessive synchronization, growing synchronization phases, identifying program statements that may be moved without side-effects, collective call decomposition, and improving communication computation overlap.

5.4.1 Excessive Synchronization and Growing Synchronization Phases

Often it is observed that completion semantics of one language or library call overlap or are a subset of another call. For example, in OpenSHMEM the *shmem_quiet* call is used to guarantee completion of all remote operation and memory stores [54]. This is a sub-set of the effect achieved by the barrier calls (barrier = quiet + synchronize), which not only guarantee memory updates across all PEs but also ensure that all PEs have reached the same point in execution. Hence, we may safely remove shmem_quiet if it is immediately followed by a barrier call as it would be equivalent to (quiet + quiet + synchronize), making the shmem_quiet redundant. Another common example of excessive synchronization would be if there is an absence of local or remote memory update within a phase, here inter-phase optimization is possible. The OSA tool indicates the trailing barrier and user may re-think the placement of the trailing barrier to a more appropriate location (where updates are required to be visible to all PEs). This grows the synchronization phase and eliminates the extra barrier call.

5.4.2 Collective Call Decomposition

Within a synchronization phase if there are one to many communication calls like shmem_broadcast but the result is used only by one or fewer PEs than those participating in the call, it is often a good strategy to replace the collective call by individual communication calls. For this we also have to ensure that the other PEs

do not use the broadcasted value in any later phase without update to it.

5.4.3 Improving Communication-Computation Overlap

The OSA tool can help identify statements within the program that may be moved without side effects. By initiating communication earlier within a phase and then following it with local computations, PEs can minimize the effects of communication latency. The goal of these category of optimizations is to perform communication in the background while the PE is busy processing data, hence when the PEs are ready to use the communicated data there is minimal waiting time. By simple rearrangement of unrelated statements within a synchronization phase we can allow for maximum communication-computation overlap.

Chapter 6

Results

We use the benchmarks/applications described in Table 6.1 to test the synchronization analysis framework we developed with the OSA. The testing methodology includes comparing the performance of the applications without the optimizations suggested by the OSA tool and then with the optimizations suggested. We use an InfiniBand cluster (called Crill [2]) with 16 nodes, with each node containing four twelve-core Opteron processors with Linux operating system. We present our results for execution over 128 PEs, keeping in mind the memory limitations imposed by the hardware. Since the improvement is over the original execution time, similar results will be realized if the platform or OpenSHMEM implementation is changed. As expected, not all applications were un-optimized. Table 6.2 shows the performance gain that was possible by implementing the hints provided by the OSA tool. We hand verified the synchronization structure of the applications as generated by the OSA tool. For all the applications tested our framework could positively indicate

that the barrier synchronization was well matched.

Benchmark	Description
CPI	Calculates value of PI
Matrix Multiplication	Performs matrix multiplication of two 2-D arrays
2D HEAT	2D heat transfer modeling
DAXPY	DAXPY like kernel
HEAT	Solving heat conduction task and generate the image file
SPING	Ping-pong test
IS	Integer Sort, part of NAS parallel benchmarks

Table 6.1: **Benchmarks/ applications used to assess OSA tool**

Benchmark	OSA Correctly Detects Synchronization Structure	Performance Gain
Matrix Multiplication	Yes	33% (excessive synchronization removal)
2D HEAT	Yes	0%
DAXPY	Yes	0%
HEAT	Yes	20.2% (improving overlap)
SPING	Yes	9% (extending synchronization phase)
IS	Yes	5.31% (excessive synchronization removal)

Table 6.2: **Results for Benchmarks used to evaluate OSA tool**

We give detailed results for the the Matrix Multiplication application which is part of the examples in the OpenSHMEM Validation and Verification Suite [58] as the structures generated for all applications are of similar nature, and hence redundant.

The Matrix Multiplication application consists of three 2-D arrays (of doubles) A, B, and C, where C is used to store the product of two matrices A and B. This program performs matrix multiplication based on 1D block-column distribution where in every iteration, every PE calculates the partial result of matrix-matrix multiplication and communicates the current portion of matrix A to its right neighbor (_my_pe() + 1) and receives the next portion of matrix A from its left neighbor (_my_pe() - 1) in a circular fashion. The main body of the benchmark is as shown in the code Listing 6.1.

Figure 6.1 shows the control flow as captured by our analysis which clearly marks out the OpenSHMEM calls and their placement. From the control flow analysis of the compiler, we use the *dominator frontier* information to extract control dependencies at the statement level. We merge this information with the data flow analysis (captured by D-U chains) and present it as a **system dependence** graph in Figure 6.2.

Here, the control dependencies are represented by light/dashed arrows while the data dependencies are represented by bold arrows. For conditionals, branches are marked with either **T** or **F** indicating if the branch is taken. This makes understanding the control and data dependence easier for the programmer.

We present the result of our multi-valued analysis by showing the logical *slicing* on the system dependence graph based on the PE number (stored in variable *rank*) for **PE 0** (shown in Figure 6.3).

Listing 6.1: Matrix Multiplication application's main body.

```
1
2   for (i = 0; i < rows; i++)
3   {
4     for (p = 1; p <= np; p++)
5     {
6       // compute the partial product of c[i][j]
7       ...
8       // send a block of matrix A to the adjacent PE
9       shmem_barrier_all ();
10      if (rank == np - 1){
11        shmem_double_put (&a_local[i][0], &tmp_local[i][0], blocksize, 0);
12        shmem_barrier_all ();
13      }
14      else{
15        shmem_double_put (&a_local[i][0], &tmp_local[i][0], blocksize,
16            rank + 1);
17        shmem_barrier_all ();
18      }
19     ...
20  shmem_barrier_all ();
```

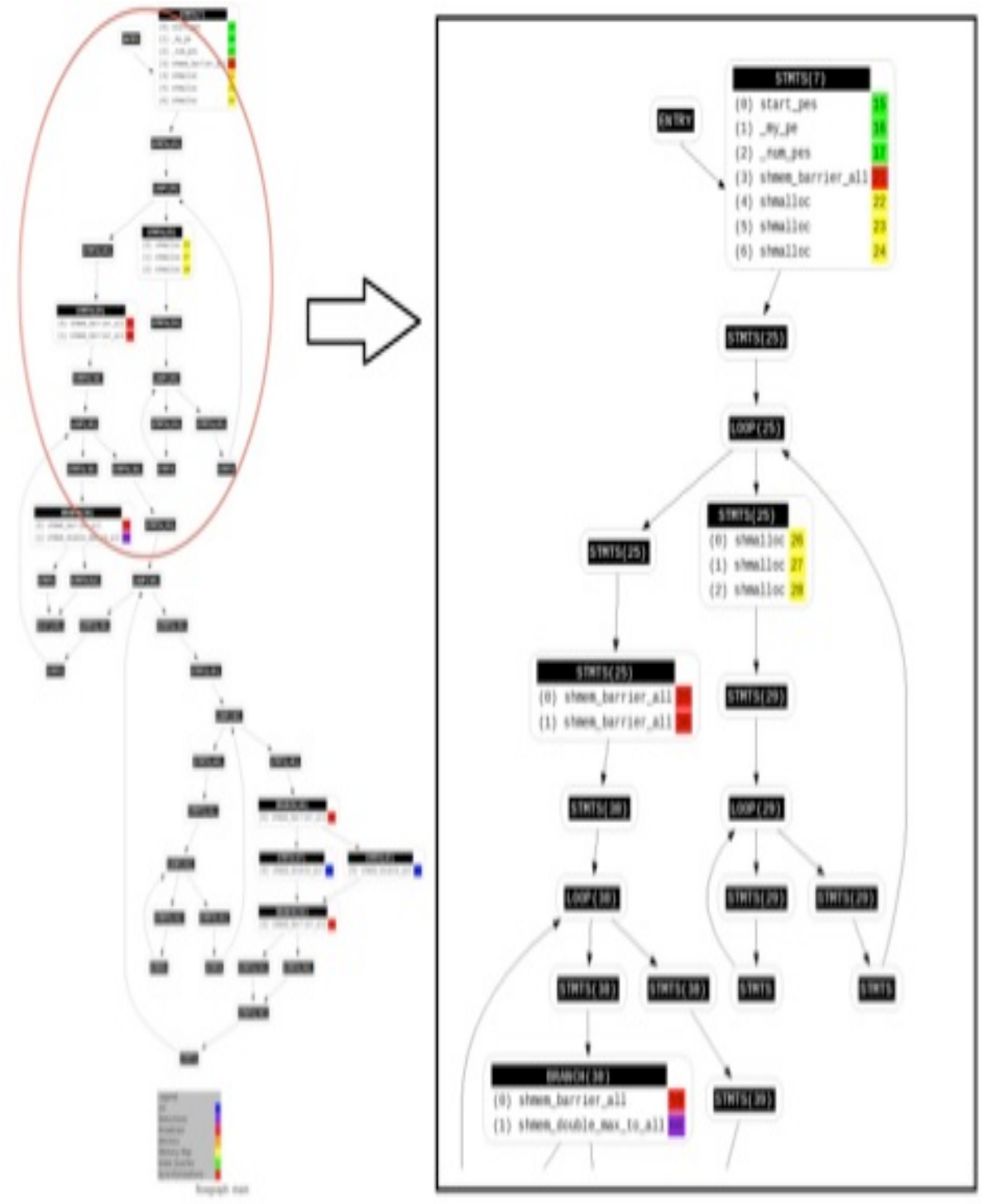


Figure 6.1: Control flow representation with OpenSHMEM calls for Matrix Multiplication application.

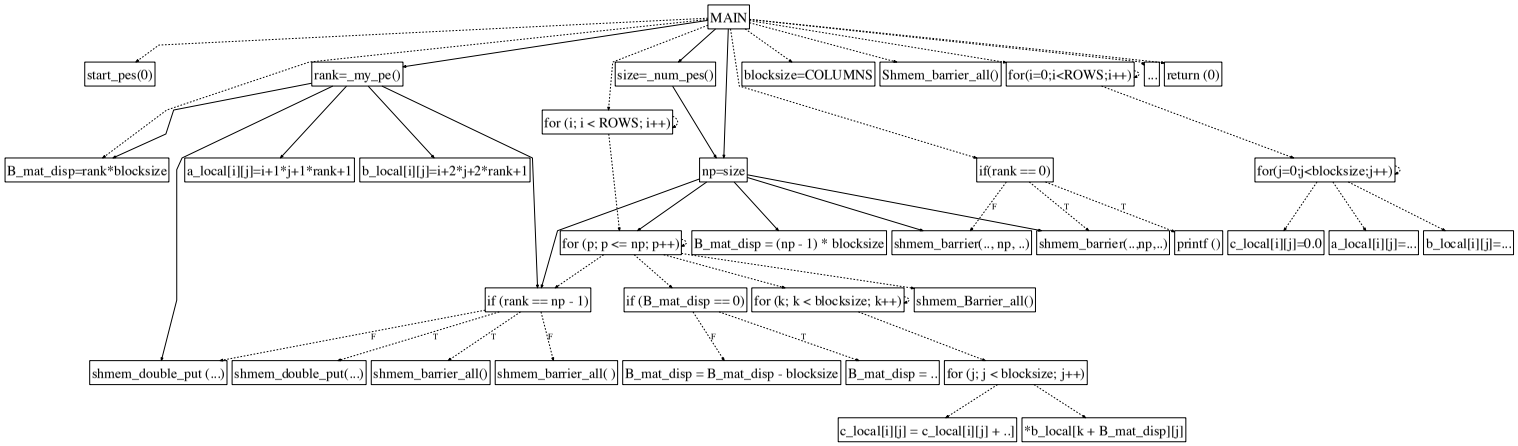


Figure 6.2: System dependence graph as generated by OSA for Matrix Multiplication application.

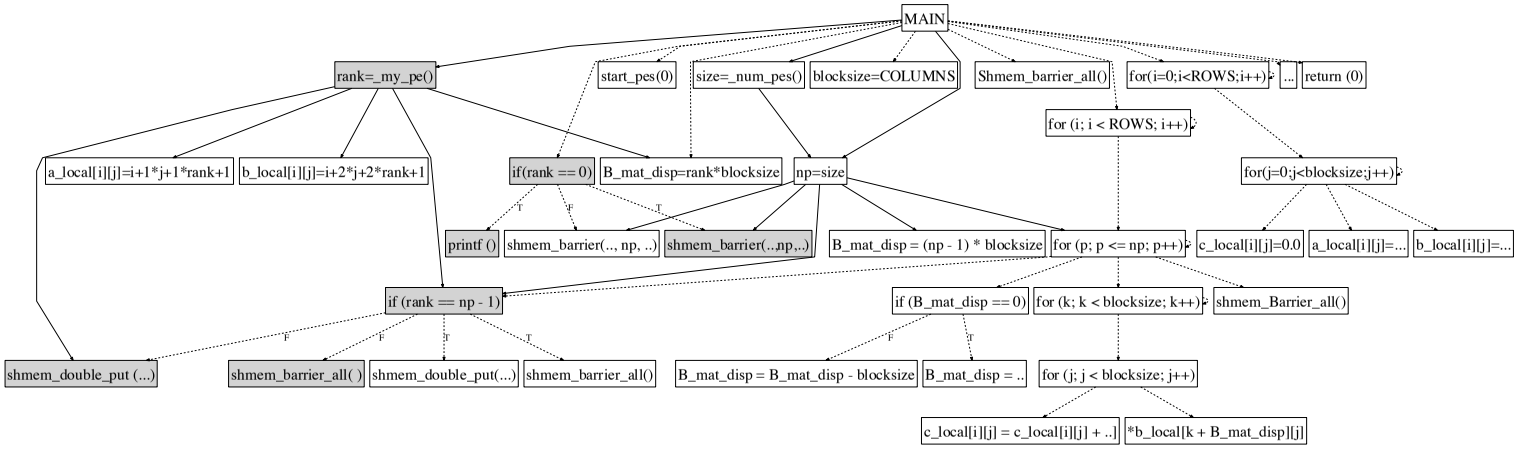


Figure 6.3: Slicing of the System dependence graph on PE 0 indicating statements executed by PE 0 only.

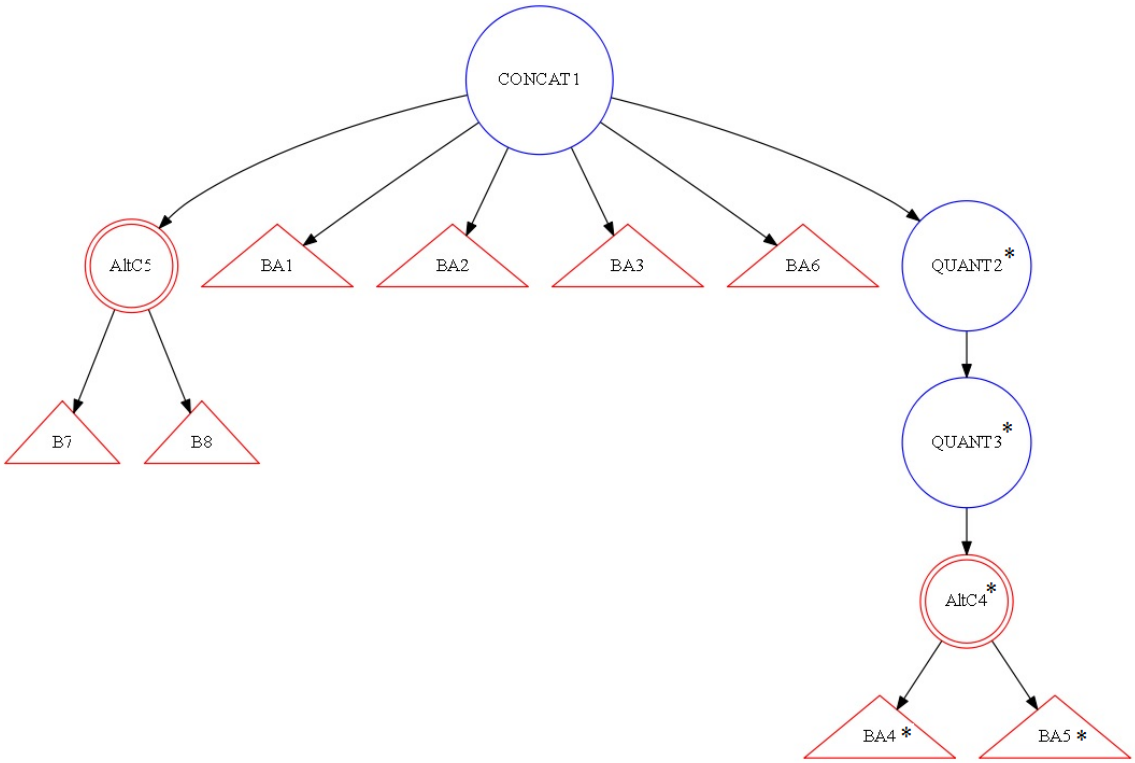


Figure 6.4: Barrier tree as generated by OSA for Matrix Multiplication application (Listing 6.1)

The program synchronization structure along with the multi-valued analysis is captured by the barrier tree generated by OSA in Figure 6.4. The entry into main() is indicated by operator **CONCAT1**. We follow the representation discussed in Table 5.2. The alternate concurrent paths are indicated by the double-circles labeled **AltC4** and **AltC5** and the two nested *for-loops* are represented by **QUANT2** and **QUANT3**. Since all loops run for the same number of times for all PEs, all PEs will encounter either BA4 or BA5 an equal number of times. Thus, just by visual inspection of the barrier tree generated by OSA tool, it is evident that the all PEs will encounter the same number of barriers. Providing simplistic representation to the

application developer makes the process of debugging and verification a trivial task. Providing useful feedback at compile time becomes more critical when applications become more complex with numerous branching statements involving multi-valued conditionals.

Chapter 7

Conclusion

The main contribution of this work is to provide an enhanced OpenSHMEM Analyzer that represents more complex analysis in an easy to understand visual manner to an OpenSHMEM programmer. We provide a graphical representation of the application's control flow, explicit with the OpenSHMEM calls, for providing detailed information about the usage and placement of all OpenSHMEM calls used by the programmer. The barrier tree provides a simplistic representation of the synchronization structure of the application along with information on the different concurrent execution paths that exist in the application. This makes discovering potential errors due to mis-aligned or missing synchronization easier for the OpenSHMEM programmer. We also pave the way for more complex analysis towards suggesting optimizations, which needs information like the system dependence graph along with the multivalued analysis and the synchronization analysis.

During the development of this analysis framework, tracking and evaluating parameters that define an *active-set* was challenging. From our experience we proposed an explicit active set handle [57] to the OpenSHMEM community, and is currently being considered as a valuable extension to the current OpenSHMEM specification. Explicit active set handles will greatly simplify the analysis resulting in better accuracy of predicting which PEs may take a particular concurrent path making it possible to provide specialized optimization feedback based on a particular PE or a group of PEs.

Our current implementation considers *shmem_barrier* and *shmem_barrier_all* synchronization calls but can be easily extended to account for other collective calls with similar completion semantics. As future work, we would like to integrate support for other OpenSHMEM API that require implicit synchronization and provide useful optimization hints to the user based on their completion semantics. Another avenue for research is to use user annotations as hints to the compiler for more accurate and speedy analysis. User annotations could either be based on the knowledge of the values of the variables (aiding multi-valued analysis) or of the synchronization structure (matching barriers).

Currently all optimization possibilities are presented as hints to the programmer, moving ahead we would like to automate the process such that the compiler performs transformation of the code without changing the application's behavior.

Bibliography

[1] 2D Heat Transfer using MPI.

[2] Crill cluster system description.

[3] *SHMEM API Man Pages*.

[4] *UPC* manual.

[5] OpenSHMEM specification.

[6] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.

[7] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 149–159, New York, NY, USA, 1996. ACM.

[8] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical report, NASA AMES Research Center, 1995.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[10] B. W. Barrett, J. M. Squyres, and A. Lumsdaine. Implementation of open mpi on red storm. Technical Report LA-UR-05-8307, Los Alamos National Laboratory, Los Alamos, New Mexico, USA, October 2005.

[11] S. H. Bokhari, B. H. Elton, and D. J. Mavriplis. The Cray MTA and Unstructured Meshes, 2000.

[12] D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[13] R. Brightwell, B. Lawry, A. B. MacCabe, and R. Riesen. Portals 3.0: Protocol Building Blocks for Low Overhead Communication. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, page 268, Washington, DC, USA, 2002. IEEE Computer Society.

[14] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, New York, NY, USA, 1995.

[15] G. Chakrabarti, F. Chow, and P. Llc. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation, and Measurements, 2008.

[16] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[17] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. Zpl: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26:2000, 2000.

[18] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[19] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[20] B. M. Chapman, D. Eachempati, and O. Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. volume 41, pages 825–854, 2013.

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[22] W.-Y. Chen, C. Iancu, and K. A. Yelick. Communication optimizations for fine-grained upc applications. In *IEEE PACT*, pages 267–278. IEEE Computer Society, 2005.

[23] Cray. Chapel language.

[24] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. Mcmahon, A. Snavely, J. Vetter, K. Yelick, S. Alam, R. Campbell, L. Carrington, T.-Y. Chen, O. Khalili, J. Meredith, and M. Tikir. Darpa's {HPCS} program: History, models, tools, languages. In M. V. Zelkowitz, editor, *Advances in COMPUTERS High Performance Computing*, volume 72 of *Advances in Computers*, pages 1 – 100. Elsevier, 2008.

[25] D. Eachempati, H. J. Jun, and B. Chapman. An open-source compiler and runtime implementation for coarray fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 13:1–13:8, New York, NY, USA, 2010. ACM.

[26] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies. Graphviz: Open Source Graph Drawing Tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[27] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972.

[28] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[29] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries, 2004.

[30] O. Hernandez, R. C. Nanjegowda, B. Chapman, V. Bui, and R. Kufrin. Open source software support for the openmp runtime api for profiling. volume 0, pages 130–137, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[31] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Traeff. The Scalable Process Topology Interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23(4):293–310, Aug. 2010.

[32] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Dependence Graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[33] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the Overlap. *In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:279–290, 2005.

[34] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '94, pages 171–180, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[35] C. Kaiser, C. Pajault, and J.-F. Pradat-Peyre. Modeling Remote Concurrency with Ada: Case Study of Symmetric Non-deterministic Rendezvous. In *Proceedings of the 12th International Conference on Reliable Software Technologies*, Ada-Europe'07, pages 192–207, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] A. A. Kamil and K. A. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. Technical Report UCB/EECS-2006-41, EECS Department, University of California, Berkeley, April 2006.

[37] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[38] LBNL and U. Berkeley. Berkeley upc : Unified parallel c.

[39] C. Lin, S. Z. Guyer, and D. Jimenez, November 2001.

[40] Y. Lin. Static nonconcurrency analysis of openmp programs. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, pages 36–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] K. Y. Luigi, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Java for High Performance Network Computing*, Concurrency: Practice and Experience, pages 825–836. John Wiley & Sons Ltd., 1998.

[42] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[43] S. A. Markstrum, R. M. Fuhrer, and T. D. Millstein. Towards concurrency refactoring for x10. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 303–304, New York, NY, USA, 2009. ACM.

[44] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 129–138, New York, NY, USA, 1993. ACM.

[45] M. Müller. Some simple openmp optimization techniques. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, WOMPAT '01, pages 31–39, London, UK, UK, 2001. Springer-Verlag.

[46] S. Muller and S. Chong. Towards a practical secure concurrent language. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 57–74, New York, NY, USA, 2012. ACM.

[47] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[48] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers. In *Proceedings of Supercomputing '94*, pages 340–349. IEEE Computer Society Press, 1994.

[49] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[50] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High Performance Remote Memory Access Communication: The ARMCI Approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, May 2006.

[51] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31.

[52] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[53] U. of Berkeley. Titanium.

[54] OpenSHMEM.org. OpenSHMEM specification 1.0, 2011.

[55] H. Oscar, J. Siddhartha, S. Pophale, P. Stephen, J. Kuehn, and C. Barbara. The OpenSHMEM Analyzer. In *Proceedings of the Sixth Conference on Partitioned Global Address Space Programming Model*, PGAS '12, 2012.

[56] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 51:1–51:12, New York, NY, USA, 2011. ACM.

[57] S. W. Poole, O. Hernandez, and P. Shamis, editors. *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - First Workshop, OpenSHMEM 2014, Annapolis, MD, USA, March 4-6, 2014. Proceedings*, volume 8356 of *Lecture Notes in Computer Science*. Springer, 2014.

[58] S. Pophale, O. Hernandez, S. Poole, and B. Chapman. Poster: Validation and Verification Suite for OpenSHMEM. In *Proceedings of the Seventh Conference on Partitioned Global Address Space Programming Model*, PGAS '13, 2013.

[59] S. Pophale, O. Hernandez, S. Poole, and B. Chapman. Static Analyses for Unaligned Collective Synchronization Matching for OpenSHMEM. In *Proceedings of the Seventh Conference on Partitioned Global Address Space Programming Model*, PGAS '13, 2013.

[60] S. Pophale, R. Nanjegowda, T. Curtis, B. Chapman, H. Jin, S. Poole, and J. Kuehn. OpenSHMEM Performance and Potential: A NPB Experimental Study. 2012.

[61] S. S. Pophale. SRC: OpenSHMEM library development. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 374–374, New York, NY, USA, 2011. ACM.

[62] C. Rice University. High Performance Fortran Language Specification. *SIGPLAN Fortran Forum*, 12(4):1–86, Dec. 1993.

[63] Y. Saad. *Iterative Methods for Sparse Linear Systems: Second Edition*. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2003.

[64] S. Satoh, K. Kusano, and M. Sato. Compiler Optimization Techniques for OpenMP Programs. volume 9, pages 131–142, Amsterdam, The Netherlands, The Netherlands, Aug. 2001. IOS Press.

[65] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[66] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. Infiniband scalability in open mpi. In *Proceedings of IEEE Parallel and Distributed Processing Symposium*, April 2006.

[67] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 432–441, May 1999.

[68] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[69] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):361–376, May 1983.

[70] UPC Consortium. Upc language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[71] M. Vakilian, S. Negara, S. Tasharofi, and R. E. Johnson. Keshmesh: A tool for detecting and fixing java concurrency bug patterns. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 39–40, New York, NY, USA, 2011. ACM.

[72] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[73] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalski, G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open MPI's TEG point-to-point communications methodology: Comparison to existing implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 105–111, Budapest, Hungary, September 2004.

[74] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen. Parallel Languages and Compilers: Perspective from the Titanium Experience. *The International Journal of High Performance Computing Applications*, 21:2007, 2007.

[75] Y. Zhang and E. Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM.

[76] Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 95–109. Springer-Verlag, Berlin, Heidelberg, 2008.