Static Analysis for Unaligned Collective Synchronization Matching for OpenSHMEM

Swaroop Pophale\textsuperscript{1}, Oscar Hernandez\textsuperscript{2}, Stephen Poole\textsuperscript{2} and Barbara Chapman\textsuperscript{1}

\textsuperscript{1} University of Houston Houston, Texas, USA
spophale, chapman@cs.uh.edu
\textsuperscript{2} Oak Ridge National Lab Oak Ridge, Tennessee, USA
oscar, spoole@ornl.gov

Abstract

The most critical step for static concurrency analysis of OpenSHMEM is to detect matching unaligned collective synchronization calls within an OpenSHMEM program. Concurrency analysis will be able to detect the regions in the code where two or more OpenSHMEM calls run concurrently and help identify parallel programming errors due to incorrect usage of OpenSHMEM library calls. For the concurrency analysis to be accurate the collective synchronization matching process must be accurate. This task is particularly challenging for OpenSHMEM programs since the OpenSHMEM library provides textually unaligned barriers over an active set in addition to the traditional barrier all statement. An active set is essentially a logical grouping of processing elements. In this paper we discuss our effort towards discovering and matching barrier calls in OpenSHMEM. We extend the OpenSHMEM Analyzer (OSA) to discover potential synchronization errors due to unaligned barriers in OpenSHMEM programs.

1 Introduction

Parallel programming libraries are critical for High Performance computing applications. OpenSHMEM is a PGAS library that can be employed on shared as well as distributed systems to aid SPMD programs to achieve potentially low-latency communication via its onesided data transfer calls. Other than data transfer an OpenSHMEM library provides library calls for collective operations (broadcast, reductions, collections and synchronization), atomic memory operations, distributed locks and data transfer ordering primitives (fence and quiet). As all libraries go, OpenSHMEM is limited by the lack of compiler support to ensure the correct use of the library in a parallel context. A part of the burden is alleviated by the syntactic and basic semantic checks already present in the OpenSHMEM Analyzer (OSA) \cite{10}. OSA extends the existing compiler technology to report errors accurately in context of C and OpenSHMEM. In this paper we extend the OSA to perform barrier matching analysis and make the control flow graph (CFG) aware of OpenSHMEM calls and its the SPMD semantics \cite{11}. Studies have determined that multi-valued expressions that affect control flow statements the cause of concurrency \cite{8,9,13} and are needed for barrier matching analysis. Multi-valued expressions are those that evaluate to different values on different processes \cite{3}. A multi-valued expression depends on a multi-valued seed/variable. A generic example of multivalued seed is the process id (which we know is unique for different processes). Another level of complexity exists for OpenSHMEM where the the collective synchronization statements are textually unaligned and may be applicable only over a portion of the total processing elements (PEs) executing the same code. Barrier matching \cite{15} across the different concurrent regions within an OpenSHMEM program is needed to detect possible synchronization errors. We present a framework\textsuperscript{1} for modifying

\textsuperscript{1}This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.
and analyzing the CFG to build a program dependence graph and perform unaligned barrier matching in presence of different collective synchronization calls, namely, `shmem_barrier` (which is over an active set) and `shmem_barrier_all`.

## 2 Related Work

Collective synchronization matching has a myriad of applications. It is used as a stepping stone to facilitate complex analysis for process-level parallelism analysis of a program in [6]. Their analysis avoids the problem of having to identify textually unaligned barriers by assuming that barriers are identified via unique barrier variables. One of the first works to verify program synchronization patterns and the rules that govern the synchronization sequences was done in [2] for Split-C. They analyze the effects of single valued expression on the control flow and concurrency characteristics of the program. They simplify the identification of unaligned barriers and single valued variables by using the single keywords for annotating the named barriers. Barrier matching for MPI [15] evaluates the different concurrent paths the processes may take (using multi-value conditional and barrier expression analysis) and check that each processes encounters an equal number of barriers. Our approach to barrier matching is similar but we distinguish our approach by avoiding the use of barrier subtrees to match barriers. This makes our analysis more resilient to unstructured code.

## 3 Motivation

OpenSHMEM is a PGAS library that provides routines for programmers using the SPMD programming paradigm. The OpenSHMEM Specification [1] provides the definition and functionality of these concise and powerful library calls for communicating and processing data. We first describe the collective calls and then discuss their implication and potential error scenarios they could lead to where programs may be syntactically correct but are either semantically wrong or may result in parallel behavior unintended by the programmer. Collective synchronization in OpenSHMEM is provided by `shmem_barrier` and `shmem_barrier_all` (over a subset of PEs and all PEs respectively) in OpenSHMEM.

### 3.0.1 shmem_barrier_all

A `shmem_barrier_all` (referred to as `barrier_all`) is defined over all PEs. OpenSHMEM requires all PEs to call `shmem_barrier_all` at the same point in the execution path. It provides global synchronization and its semantics guarantee completion of all local and remote memory updates once a PE returns from the call.

### 3.0.2 shmem_barrier

A `shmem_barrier` (referred to as `barrier`) is defined over an active set. An active set is a logical grouping of PEs based on three parameters (passed as arguments), namely, `PE_start`, `logPE_pe` and `PE_size` triplet [1]. OpenSHMEM requires all PEs within the active set to call `shmem_barrier` at the same point in the execution path. When barriers are not matched it is a obvious dead-lock situation, but even in cases where all barrier (barrier all and barrier) statements are well matched the compiler needs to differentiate between the two to make sure that the semantics of the OpenSHMEM library are not violated. In the code below we observe one such situation.
In this example all barrier statements are matched. We see that on line 5 all even numbered 
PEs update their value of the variable `source` and immediately after, update the value of 
the symmetric variable `target` on the next PE in a circular fashion. Simultaneously all odd 
numbered PEs are updating a local variable `x` with the value in their symmetric variable `target` (line 12). Since the semantics of OpenSHMEM only guarantee the completion of `puts` after 
synchronization, on line 12, the odd numbered PEs may or may not have the updated value of 
`target`. The programmer may easily miss such errors leading to an application with inconsistent 
results. For this type of analysis barrier matching is essential to detect concurrent execution 
phases for reads and writes to the same variables along with data flow analysis.

4 Unaligned Collective Synchronization Matching Framework

To be able to discover matching barriers in OpenSHMEM program we need to first locate 
and mark the multi-valued expressions which cause different processes to follow different 
execution paths. Multi-valued expressions evaluate to different results on different PEs. The 
outcome of a multi-valued expression depends on a multi-valued seed. Extending the rules 
stated in [3] we can make certain assumptions about the expressions that generate from a 
known single-valued or multi-valued seed. Depending on the semantics of OpenSHMEM and 
its treatment of different program variables, the rules for determining multi-valued seed have 
to be modified. Different library calls cause changes in the values that may make them single 
or multi-valued. This information is required for a finer analysis and to avoid overestimation 
of concurrency. Table 1 lists the OpenSHMEM library operation categories that cause a change 
in the value of a program variable (this includes all variables used in the program irrespective 
of their class) and the effect they have on the variable. Explanation of the API and the 
nomenclature is beyond the scope of the paper (refer to OpenSHMEM Specification 1.0 [1]).

4.1 Identifying Multi-value Conditionals

To identify multi-value conditionals we need to essentially follow the data flow propagation of 
the multi-value seeds through the program’s CFG and mark the conditionals that are affected 
directly or indirectly by them [15]. A system dependence graph [5] needs to be built. We look 
at program slicing as a way to identify the reach of these multi-valued seeds. Programming 
slicing is defined as, "A decomposition based on data flow and control flow analysis" [14]. To
### OpenSHMEM Library Operations

<table>
<thead>
<tr>
<th>Variable Affected</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>_num_pes</td>
<td>npes</td>
</tr>
<tr>
<td>_my_pe</td>
<td>me</td>
</tr>
<tr>
<td>PUT (elemental, block, strided)</td>
<td>target</td>
</tr>
<tr>
<td>GET (elemental, block, strided)</td>
<td>target</td>
</tr>
<tr>
<td>ATOMICS (fetch and operate)</td>
<td>target</td>
</tr>
<tr>
<td>BROADCAST</td>
<td>target</td>
</tr>
<tr>
<td>COLLECTS (fixed and variable length)</td>
<td>target array</td>
</tr>
</tbody>
</table>

Table 1: **Effect of OpenSHMEM library calls on program variables**

do the program slicing, we look at the data flow and control flow information generated by the compiler and build a system dependence graph [5] as shown in Figure 1. If we were to take a forward slice of the sample program based on the multi-valued PE number me at A2, then we get either A3-B1-B2-B3-B4-B5-D or A3-C1-C2-D depending on the value of me. These slices help us identify the multi-valued conditionals in the program by finding the points at with the slices diverge.

![System Dependence Graph](image)

**Figure 1: System Dependence Graph**

### 4.2 Synchronization Matching

To facilitate analysis between and across barrier regions we need to identify code phases (a valid synchronization free path enclosed within barriers) that lie between matching barriers. We use the approach in [15] for developing barrier expressions and barrier trees as the first step towards barrier matching. As defined in [15] barrier expressions are very similar to path expressions and can be generated from them by replacing the node labels by barrier labels. Like regular expressions these expressions use three types of operators: concatenation (·), alternation (|), and quantification (∗) [7]. Additionally we borrow the operator ∣c from [15] to indicate the operator concurrent alternation, which essentially indicates that the different execution paths diverge.
from a multi-valued conditional and that different PEs may take different paths from this point on. Table 2 gives the rules that govern the barrier expression generation. It is important to note that if the result of a quantification operation is statically non-deterministic (like a barrier enclosed within a while loop based on a value available only at run-time) we cannot determine with confidence any concurrency relationship for such programs. Using the operators mentioned above we can derive the barrier expression and the barrier tree. The barrier expressions can be generated by first generating the path expressions using methods in [4] or [12]. The barrier expression for the entire program is generated by first evaluating the barrier expressions of the individual procedures and then connecting them together using the inter-procedural program dependence graph. Barrier matching is done by first generating the legal sequences of barriers by a constraint driven depth-first-search of the barrier tree and validating barrier sequences that have at least one other barrier sequence of the same length.

5 Evaluation

As of this writing we are testing our methodology using the OpenSHMEM versions of NAS Parallel benchmarks (in C). We have been successful in detecting matching barriers in all the smaller test codes provided with the OpenSHMEM Validation and Verification Suit. For some codes we intentionally added unmatched barriers and our algorithm was able to detect it. In severely unstructured codes or with conditional values which are runtime dependent (which are defaulted to multivalued) our algorithm generates a warning even if barriers are matched.

References

Synchronization Matching for OpenSHMEM

Pophale, Hernandez, Poole and Chapman


