Extending the OpenSHMEM Analyzer to perform synchronization and multi-valued analysis

Swaroop Pophale[1], Oscar Hernandez[2], Stephen Poole[2], and Barbara Chapman[1]

[1] University of Houston, Houston, Texas 77004, USA,
spophale, chapman@cs.uh.edu
[2] Oak Ridge National Laboratory, Oak Ridge, Tennessee, 37840, USA
oscar, spoole@ornl.gov

Abstract. OpenSHMEM Analyzer (OSA) is a compiler-based tool that provides static analysis for OpenSHMEM programs. It was developed with the intention of providing feedback to the users about semantics errors due to incorrect use of the OpenSHMEM API in their programs, thus making development of OpenSHMEM applications an easier task for beginners as well as experienced programmers. In this paper we discuss the improvements to the OSA tool to perform parallel analysis to detect collective synchronization structure of a program. Synchronization is a critical aspect of all programming models and in OpenSHMEM it is the responsibility of the programmer to introduce synchronization calls to ensure the completion of communication among processing elements (PEs) to prevent use of old/incorrect data, avoid deadlocks and ensure data race free execution keeping in mind the semantics of OpenSHMEM library specification. Our analysis yields three tangible outputs: a detailed control flow graph (CFG) making all the OpenSHMEM calls used, a *system dependence* graph and a *barrier tree*. The barrier tree represents the synchronization structure of the program presented in a simplistic manner that enables visualization of the program's synchronization keeping in mind the concurrent nature of SPMD applications that use OpenSHMEM library calls. This provides a graphical representation of the synchronization calls in the order in which they appear at execution time and how the different PEs in OpenSHMEM may encounter them based upon the different execution paths available in the program. Our results include the summarization of the analysis conducted within the middle-end of a compiler and the improvements we have done to the existing analysis to make it aware of the parallelism in the OpenSHMEM program.

1 Introduction

OpenSHMEM is a PGAS library that may be used with C, C++ or Fortran SPMD programs to achieve potentially low-latency communication via its *one-sided* remote direct memory access (RDMA) calls on shared as well as distributed systems that have the required hardware capability. Compilers are not aware

of the parallel semantics of the OpenSHMEM library and they treat it like a black box, thus hindering optimizations. OpenSHMEM analyzer (OSA) [1] is the first effort to develop a compiler-base tool aware of the parallel semantics of an OpenSHMEM library. OSA is built on top of the OpenUH compiler, that provides information about the structure of the OpenSHMEM code and semantic checks to ensure the correct use of the symmetric data in OpenSHMEM calls.

This paper describes how we built on top of the existing OSA framework to provide an in-depth synchronization analysis based on the control flow of the OpenSHMEM program that can be used to match collective synchronization calls, which are the building blocks for detecting the possible concurrent execution paths of an application. Our framework uses the concepts of multivalued seeds and multivalued expressions to build a multi-valued system dependence graph in the context of OpenSHMEM, which later is used to build a *barrier-tree* representation. 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* [2–5] and the particular variable used within the expression that causes this phenomenon is the *multi-valued seed*. Here we **implement** the critical parts of the framework proposed in [6] and provide insights into the the practical aspects of detection of *multi-valued* expressions by identification and tracing of *multi-valued* seeds.

Within the OpenUH compiler the source code is converted into an intermediate representation (IR) 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. We do our analysis within the inter-procedural analysis phase (IPA) 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, while the second graph structure is the *barrier- tree* where the leaves of tree are OpenSHMEM collective synchronization calls (*shmem_barrier* and *shmem_barrier_all*) and the nodes are operators (discussed in Section 4) 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.

This paper is organized as follows. We describe our motivation for extending the analysis capability of OSA in Section 2 and provide better understanding of OpenSHMEM's memory model and synchronization semantics in Section 3. We discuss the changes to the infrastructure and implementation details in Section 4 and present our results with the help of the Matrix Multiplication application in Section 5. Section 6 describes different static analysis techniques that have been explored for concurrency and synchronization detection in parallel programming

models. In Section 7 we summarize our contributions and our aspirations for the future of the OSA tool.

2 Motivation

The main characteristics of a program are captured by the control flow and data flow within the program. Especially in parallel programming it is often beneficial to be able to visualize the interaction of the different program components. PEs executing in parallel may take different paths depending on the explicit or implicit requirements set by the programmer. These are often expressed as conditions over variables that evaluate to different values on different PEs. Capturing this information in a simplistic visual manner can aid the users understand the concurrency in their applications. Figure 1 show a control flow graph (CFG) of an OpenSHMEM program described in Listing 1.1. Here each basic block has a single entry point and a single point of exit, and may contain OpenSHMEM calls. Using this CFG alone cannot convey the different paths that are possible within the program. To be able to distinguish these different paths we need to identify multi-value conditionals we need to essentially follow the propagation of the multi-value seeds through the program and mark the conditionals that are affected directly or indirectly by them [7]. To build such multi-valued CFG, we need to identify multi-valued seeds, determine the control flow and capture the effect of such seeds by analyzing the data flow. This results in a system dependence graph that can be used to do logical program slicing based on multi-valued conditions. This can later used to build a barrier tree to perform our synchronization analysis.

Listing 1.1: An OpenSHMEM program example with unaligned barriers.

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

```
28            shmem_barrier_all();
29           }
30      }
31     return 0;
32  }
```

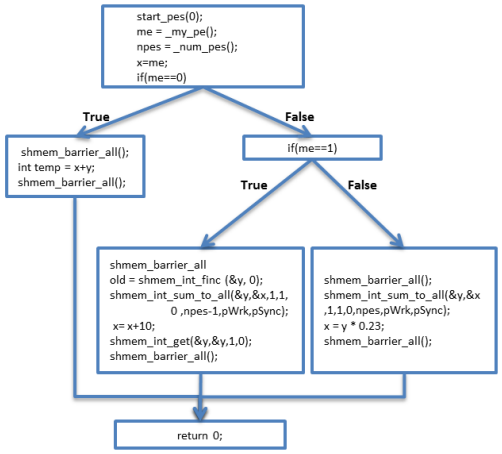


Fig. 1: Control Flow Graph of the OpenSHMEM program from Listing 1.1.

The CFG is represented as a sequence of *basic blocks* which shows the possible alternatives within the program but does not provide explicit information on what or how the control is determined and what data dependence may affect the concurrency relationship between different parallel paths of execution. This is better depicted by the system dependence graph [8] as shown in Figure 2. To determine the exact execution path or slice we look at the combination of the data flow and control flow information generated by the compiler. The *system dependence graph* is expressed in terms of statements and based on the CFG and the outcome of the conditional statements each control edge is either marked *true* (T) or *false* (F). In Figure 2 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 A2-A3-A4-A5-A6-B1-B2-B3-C or A2-A3-A4-A5-A6-C-D1-D2-D3-D4-D5-D6 or A2-A3-A4-A5-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 with the execution paths diverge. Synchronization analysis is another important aspect to understand the relationship between code regions. A *code phases* [9] can be defined as a valid (synchronization free) path enclosed within two global synchronization statements. Since OpenSHMEM has unaligned global synchronization, the first step is to identify these statements across different execution paths. This helps in the identification of potentially concurrent code or errors due to unmatched barriers. Our work in this paper addresses the challenges of extracting information from the compiler and merging

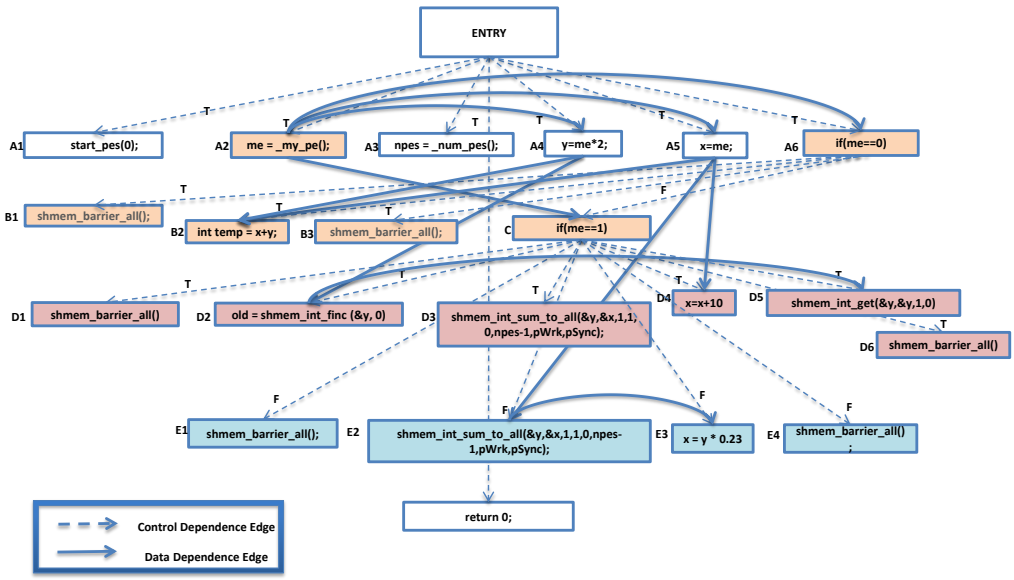


Fig. 2: System Dependence Graph of OpenSHMEM example (Listing 1.1) showing the control and data dependencies at statement level.

it with rules based off OpenSHMEM library semantics and presenting it to the user in a condensed meaningful format for visual inspection which can be used to detect synchronization mismatch, incorrect conditionals etc.

3 OpenSHMEM Library

As mentioned before OpenSHMEM is a PGAS library that provides routines for programmers using the SPMD programming paradigm. The OpenSHMEM Specification [10] provides the definition, functionality and expected behavior of these library calls. OpenSHMEM communication calls are *one-sided*, i.e. they do not require the involvement of the target PE for completion and when the underlying hardware allows RDMA, it can provide excellent opportunities for hiding communication latency by overlapping communication with computation.

OpenSHMEM introduces the concept of *symmetric variables*. By definition, *symmetric* data consists of variables that are allocated by an OpenSHMEM library (using special memory allocation calls) or defined as *global* or *static* in C/C++ or in *common* or *save* blocks in Fortran [10]. These variables have the

same name, size, type, storage allocation, and offset address on all PEs. The library calls *shmalloc* (C/C++) and *shpalloc* (Fortran) allocate and manage *symmetric* variables on the *symmetric heap*, which is remotely accessible from every other PE. Symmetric data allocation is a *collective* process and OpenSHMEM Specification requires that it appear at the same point in the code with identical *size* value [10]. This data is local to the PE and is not visible or directly accessible to a remote PE. Some OpenSHMEM library routines like *shmem put* and *shmem get* may use local variables as the *source* and *target* respectively. Hence for analysis for *multi-valued* seeds we must consider both types of data in conjunction with the OpenSHMEM calls they are used with.

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_pe***, and the ***PE_size*** [10]. OpenSHMEM allows for unaligned barriers, both the code listings, Listing 1.2 and 1.3, 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.

Listing 1.2: 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 1.3: C code with aligned barrier

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

By providing information at compile time the programmer can analyze the program structure **before** execution, thus preventing resource wastage.

4 Methodology

As discussed above, the two main concepts to consider for concurrency analysis of OpenSHMEM program are *multi-valued expressions* and *unaligned barriers*. In this section, we discuss the structure of the OSA and the additional analysis added to it to be able to do the *multi-valued* analysis and evaluate the system dependence graph and the barrier-tree for the entire program.

4.1 OSA Infrastructure

Figure 3 shows the different stages within the compiler and the shaded region are the phases where OSA tool does 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 [11]. We used the DU-manager, Alias Manager and control flow analysis data structures to build our system dependence graph to perform multi-valued analysis.

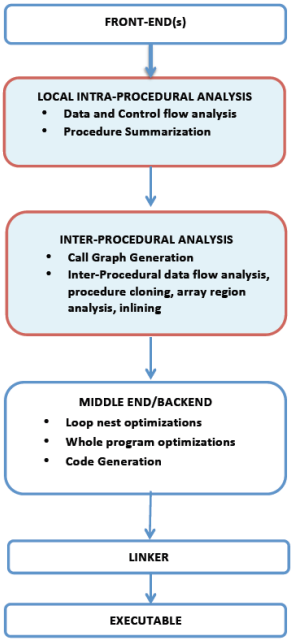


Fig. 3: Shaded blocks indicate OSA analysis within the OpenUH compiler.

4.1.1 Identification of Multivalued seeds As defined in Section 2 *multi-valued* expressions evaluate to different results on different PEs. The outcome of a multi-valued expression depends on a *multi-valued* seed. [5] states certain generic rules governing the multi-calued property of programming variables. For example, uninitialized data structures 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.

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 (else they result in *multi-valued target*). By

Placement of barriers	Operator used	Result
b1 followed by b2	$\cdot$	b1 $\cdot$ b2
if{ b1 } else { b2 };	|	b1 | b2
for(n times) b;	$\cdot$	b1$\cdot$ b2 $\cdot$ ... bn

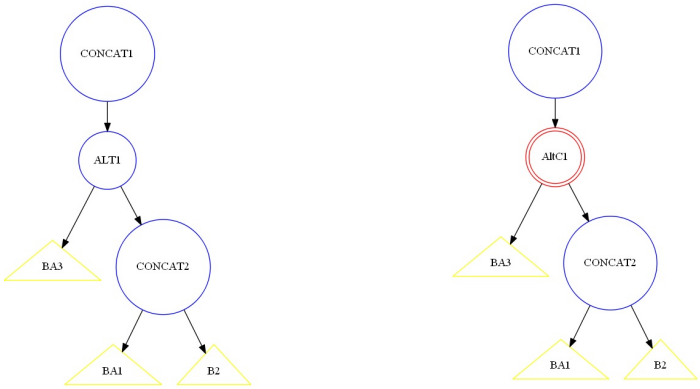
Table 1: **Rules for building a barrier expression**

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* or *multi-valued* seed. 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. 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. We append this information with the control dependencies extracted from the control flow graph with the help of the *dominator frontier* information. Both of this combined gives the system dependence graph which the OSA generates for the user to inspect.

4.1.2 Generating Barrier Trees [7] defines a *barrier expression* as being very similar to a path expression, with barriers connected by operators that best describe the control flow of the program. We extract the synchronization structure in a tree format 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 where the barriers are leaf nodes and the *operators* are the nodes of the tree. Like regular expressions, barrier trees use three types of operators: concatenation ($\cdot$), alternation (|), and quantification ($*$) [12]. Table 1 gives the rules that govern the barrier expression generation. It is important to note that if the result of a 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. Additionally we borrow the operator $|^c$ from [7] to indicate the operator *concurrent alternation*. This operator indicates that the different execution paths diverge from a *multi-valued* conditional.

Listing 1.4: 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  }
```



(a) Barrier-tree without multi-valued analysis

(b) Barrier-tree with multi-valued analysis

Fig. 4: Barrier trees generated by OSA for code Listing 1.4(where CONCAT=concatenation, ALT= alternation, AltC = alternate concurrent, B = shmem_barrier, and BA = shmem_barrier_all)

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. 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 1.4 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 4a. Purely based on Figure 4a 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 by AltC (alternate concurrent) label in Figure 4b. We augment the multi-value analysis to this providing a more meaningful representation of the program structure. Figure 4b depicts the barrier tree for the second scenario discussed above.

5 Results

We test our analysis framework on the Matrix Multiplication application which is part of the examples in the OpenSHMEM Validation and Verification Suite [13]. The application consists of three 2-D arrays A, B, and C, where C is used to store the product of two matrices A and B.

Listing 1.5: Matrix Multiplication application's main body.

```
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], &a_local[i][0], blocksize, 0);
12         shmem_barrier_all ();
13       }
14       else{
15         shmem_double_put (&a_local[i][0], &a_local[i][0], blocksize,
16             rank + 1);
17         shmem_barrier_all ();
18       }
19      ...
20   shmem_barrier_all ();
```

This program performs matrix multiplication based on 1D block-column distribution where in every iteration, the PE calculates the partial result of matrix-matrix multiply and communicates the current portion of matrix A to its right neighbor and receives the next portion of matrix A from its left neighbor. The main body of the benchmark is as shown in the code Listing 1.5.

Figure 5 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 Def-Us chains as discussed in Section 4) and present it as a **system dependence** graph in Figure 6.

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 when the branch is taken. This makes understanding the control and data dependence easier for the programmer.

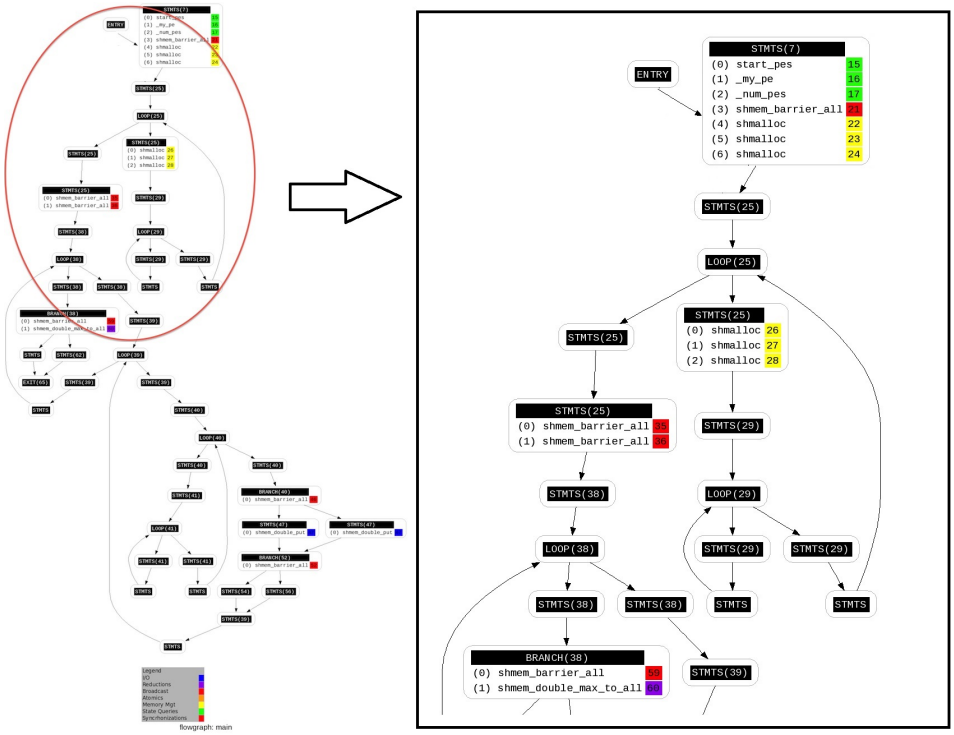


Fig. 5: Control flow representation with OpenSHMEM calls for Matrix Multiplication application.

We present the result of our multi-valued analysis in Figure 7. When we perform a logical *slicing* on the system dependence graph based on the PE number (stored in variable *rank*). In a multi PE execution scenario the statements in shaded boxes are executed only by **PE 0**. The program synchronization structure along with the multi-valued analysis is captured by the barrier tree generated by OSA in Figure 8. The entry into main() is indicated by operator **CONCAT1**. We follow the representation discussed in Section 4. 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 equal number of times. Thus, just by glancing at the barrier tree generated by OSA it is evident that the all PEs will encounter the same number of barriers. This makes the process of debugging and verification a trivial

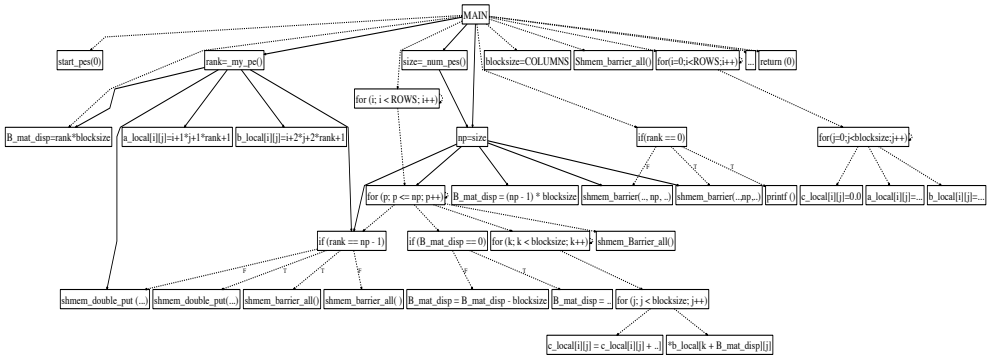


Fig. 6: System dependence graph as generated by OSA for Matrix Multiplication application.

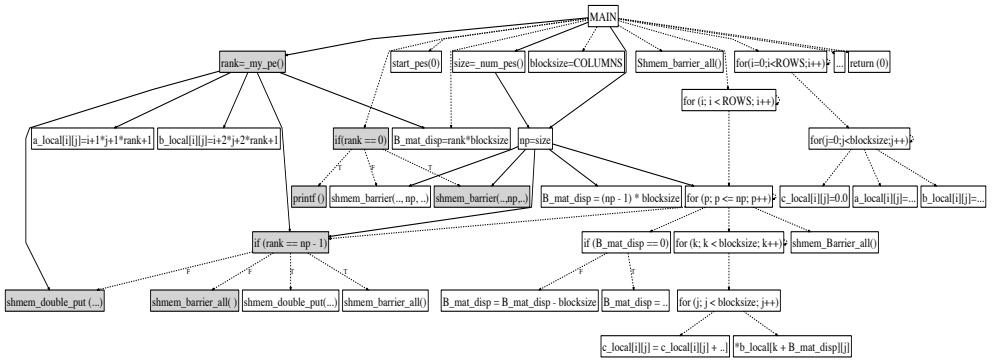


Fig. 7: Slicing of the System dependence graph on PE 0 indicating statements executed by PE 0 only.

task. This becomes more critical when applications become more complex with numerous branching statements involving multi-valued conditionals.

6 Related Work

Depending on the semantics of the parallel programming model, most applications rely on synchronization primitives to ensure updates or maintain ordering of different programming sub-tasks. Errors in synchronization could lead to incorrect or irreproducible execution characteristics. Hence research on synchronization and concurrency has always been an important aspect for the high performance programming community. One of the first research to verify program synchronization patterns and was done in [14] for Split-C. They analyze the effects of *single valued expression* on the control flow and concurrency character-

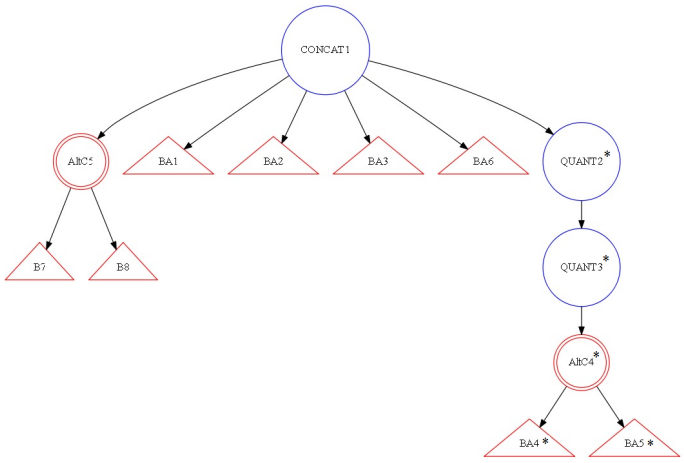


Fig. 8: Barrier tree as generated by OSA for Matrix Multiplication application (where CONCAT=concatenation, ALT= alternation, AltC = alternate concurrent, QUANT = quantification, B = shmem_barrier, and BA = shmem_barrier_all). * Indicates operators and barriers in code Listing 1.5.

istics of the program and define rules that govern the synchronization sequences. Like OpenSHMEM, Split-C has unaligned barriers and this work simplifies their identification with the use of keywords that are used for annotating the named barriers. [7] tries to identify and match unaligned barriers for MPI programs to uncover potential synchronization errors. They evaluate the different concurrent paths the processes in the MPI program may take by using multi-value conditional and barrier expression analysis and verify that each processes encounters an equal number of barriers. For other PGAS languages, like Titanium [15] is in identification of textually aligned barriers and was first proposed in [9]. They propose an inter-procedural algorithm that computes the set of all concurrent statements by first modifying the CFG and provide rules to perform a modified depth first search to ascertain pairs of concurrent expressions. Other parallel programming languages, such as X10 [16,17], Ada [18,19], and Java [20,21] have also explored analysis based on synchronization structure of a program.

7 Conclusions and Future Work

The main contribution of our work is to provide an enhanced OSA that presents more complex analysis in an easy to understand visual manner to an OpenSHMEM programmer. We provide CFG explicit with the OpenSHMEM calls, for providing detailed information about the usage and placement of OpenSHMEM calls, and a system dependence graph that clearly indicates the control and data dependencies prevalent in the application. The barrier tree provides a simplistic representation of the synchronization pattern along with information on concur-

rent execution paths available which 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 *active-sets* was challenging and we hope that the future library specification of OpenSHMEM will address this by providing implicit *active-sets* with handles. This will simplify the analysis considerably 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 OpenSHMEM barrier and barrier all synchronization calls but can be easily extended to account for other collective calls with implicit synchronization semantics. As future work we would like to integrate support for implicit synchronization and provide useful optimization hints to the user based on OpenSHMEM semantics. For example, if an application has no updates between two consecutive barriers on the same execution path we would want to indicate that there is no requirement for the extra synchronization to the application programmer/user at compile time thus helping achieve better performance.

8 Acknowledgments

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.

References

1. Oscar, H., Siddhartha, J., Pophale, S., Stephen, P., Kuehn, J., Barbara, C.: The OpenSHMEM Analyzer. In: Proceedings of the Sixth Conference on Partitioned Global Address Space Programming Model. PGAS '12 (2012)
2. Taylor, R.N.: A general-purpose algorithm for analyzing concurrent programs. Commun. ACM **26** (1983) 361–376
3. Lin, Y.: 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, Berlin, Heidelberg, Springer-Verlag (2008) 36–50
4. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming. PPOPP '93, New York, NY, USA, ACM (1993) 129–138
5. Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., Bershad, B.N.: Fast, effective dynamic compilation. In: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation. PLDI '96, New York, NY, USA, ACM (1996) 149–159

6. Swaroop, P., Oscar, H., Stephen, P., Barbara, C.: 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)
7. Zhang, Y., Duesterwald, E.: 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, New York, NY, USA, ACM (2007) 194–204
8. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. PLDI '88, New York, NY, USA, ACM (1988) 35–46
9. Kamil, A.A., Yelick, K.A.: Concurrency analysis for parallel programs with textually aligned barriers. Technical Report UCB/EECS-2006-41, EECS Department, University of California, Berkeley (2006)
10. OpenSHMEM.org: OpenSHMEM specification 1.0 (2011)
11. Chakrabarti, G., Chow, F., Llc, P.: Structure layout optimizations in the open64 compiler: Design, implementation and measurements (2008)
12. Kleene, S.C.: Representation of events in nerve nets and finite automata. Automata Studies (1956)
13. Swaroop, P., Oscar, H., Stephen, P., Barbara, C.: Poster: Validation and verification suite for OpenSHMEM. In: Proceedings of the Seventh Conference on Partitioned Global Address Space Programming Model. PGAS '13 (2013)
14. Aiken, A., Gay, D.: Barrier inference. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '98, New York, NY, USA, ACM (1998) 342–354
15. Luigi, K.Y., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance java dialect. In: In ACM. (1998) 10–11
16. Markstrum, S.A., Fuhrer, R.M., Millstein, T.D.: Towards concurrency refactoring for x10. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP '09, New York, NY, USA, ACM (2009) 303–304
17. Muller, S., Chong, S.: Towards a practical secure concurrent language. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '12, New York, NY, USA, ACM (2012) 57–74
18. Kaiser, C., Pajault, C., Pradat-Peyre, J.F.: Modelling 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, Berlin, Heidelberg, Springer-Verlag (2007) 192–207
19. Burns, A., Wellings, A.: Concurrency in Ada. Cambridge University Press, New York, NY, USA (1995)
20. Vakilian, M., Negara, S., Tasharofi, S., Johnson, R.E.: 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, New York, NY, USA, ACM (2011) 39–40
21. Magee, J., Kramer, J.: Concurrency: state models & Java programs. John Wiley & Sons, Inc., New York, NY, USA (1999)