Automatic Relative Debugging of OpenMP Programs

Gregory Matthews[*], Robert Hood[*], Haoqiang Jin,
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035 USA
{gmatthew,rhood,hjin}@nas.nasa.gov

Stephen Johnson, and Constantinos Ierotheou
Parallel Processing Research Group
Maritime Greenwich Campus
University of Greenwich
London, SE10 9LS UK
{S.Johnson,C.Ierotheou}@gre.ac.uk

NAS Technical Report NAS-03-014
October 2003

Abstract

In this work we show how automatic relative debugging can be used to find differences in computation between a correct serial program and an OpenMP parallel version of that program that does not yield correct results. Backtracking and re-execution are used to determine the first OpenMP parallel region that produces a difference in computation that may lead to an incorrect value the user has indicated. Our approach also lends itself to finding differences between parallel computations, where executing with M threads produces expected results but an N thread execution does not ($M, N > 1$, $M \neq N$). OpenMP programs created using a parallelization tool are addressed by utilizing static analysis and directive information from the tool. Hand-parallelized programs, where OpenMP directives are inserted by the user, are addressed by performing data dependence and directive analysis.

1. Introduction

Parallelization of existing programs continues to be of interest to a growing number of researchers and developers as parallel computers become commonplace. The porting process presents several difficulties though, such as making efficient use of parallel resources, minimizing the introduction of errors, and detecting and fixing any errors that are introduced. There are three general approaches to select from when parallelizing an existing program:

- manual translation, where explicit parallel and communication constructs are added to the code;
- compiler-based parallelization, where the user employs source code directives [8, 17] to steer the compiler into producing efficient parallel code; and
- fully interactive parallelization tools, where user inputs steer the parallelization process [2, 10].

None of these approaches does a perfect job of addressing all the difficulties of parallelizing an existing program, and in particular all of them fail to eliminate the introduction of errors because each requires (or greatly benefits from) user input at some level. Input from the user can be erroneous in any approach, thus it is beneficial to consider ways to assist the user in detecting and fixing bugs introduced in the parallelization process.

In previous work [11] we presented a new approach for detecting errors in distributed memory parallelized programs through the use of automatic relative debugging. This approach aids the user by comparing the execution of the serial program with that of the parallelized version, automatically finding the first difference in the parallel program that may have caused an incorrect value that the user has identified. Automation of the search for the first difference relies on the availability of sophisticated dependence analysis of the programs, and the ability to determine the computation mapping from the serial to the parallel program. These requirements were met with information collected from the parallelization tool used in the porting process.

In this paper we extend the automatic relative debugging approach in two important ways. First, we apply the approach to shared-memory OpenMP [13] programs, determining the first parallel region in the parallelized program

[*] Employees of Computer Sciences Corporation through NASA contract DTTS59-99-D-00437/A61812D.

that contains a difference that may have caused an incorrect value that the user has identified. Second, we enable automatic relative debugging for the two parallelization methods that apply to OpenMP – insertion of directives by the user, and directive insertion by a parallelization tool.

The rest of this paper is organized as follows. In Section 2 we discuss the different ways bugs may be introduced in the process of OpenMP parallelization. Section 3 presents an overview of our previous work, and how we adapt it to OpenMP debugging. Section 4 describes our prototype implementation, and Section 5 describes related work. Section 6 presents experimental results, along with a comparison to results from a prominent OpenMP debugging tool. In Section 7 we discuss directions for future research, and we conclude our discussion in Section 8.

2. OpenMP Parallelization Bugs

OpenMP parallelization is a non-trivial process that, regardless of the approach used, benefits greatly from user input. While the need for user input is apparent in manual directive insertion, it is actually important in tool parallelization as well. A tool can effectively act as a parallelizing compiler, but resulting parallel performance and scalability is hampered by the conservative analysis the tool performs in order to maintain correctness. User input to the tool is highly desirable to refine program analysis by providing information such as expected program inputs, externally defined functions, and clarification about algorithm structure.

When guiding the insertion of directives, the user has to work in an interprocedural context, considering dataflow and memory location re-use within routines possibly called from parallel regions. Achieving maximum performance can provide an incentive for the user to make overly optimistic decisions, such as adding incorrect directives or telling a parallelization tool to remove a dependence that is essential to the code operation. Such mistakes may lead to a number of common problems, for example:

- parallel execution where a loop-carried dependence exists;
- forcing a variable to be private even though its value flows beyond the parallel region;
- sharing a variable that is assigned a value by several threads in an unsynchronized fashion.

We want to address such problems with an approach that is highly automated, both to relieve the user of part of the debugging burden, and to avoid the possibility of further user mistakes that delay the diagnosis and correction of bugs.

3. Finding the First Difference

The next two subsections provide a general overview of automatic relative debugging, including the high-level algorithm and the information required to facilitate it. We then describe the modifications needed to apply automatic relative debugging to OpenMP programs.

3.1. Basic Algorithm

The user initiates automatic relative debugging by providing the name and location of a variable known to have a bad value in the parallel program. The relative debugger then begins a series of steps that effectively automate what the user might otherwise do by hand – look back through the execution history to determine what statement computed the bad value, test the values used in that definition, and repeat that search until the initial cause is found. Our previous work provides a thorough presentation of the algorithm [11]; here we give an overview to familiarize the reader.

The search for the first difference is performed in a depth-first search manner by repeating the following steps until no earlier difference is found:

⟨1⟩ Find the possible definition points of the earliest incorrect value we have seen so far, using dependence analysis information. (The user's initial bad value serves as a starting point.)

⟨2⟩ Examine the variable references on the right-hand sides of those definitions to determine a set of suspect variable references to monitor in a re-execution.

⟨3⟩ Instrument the suspect variable references in both the serial and parallel versions of the program.

⟨4⟩ Execute the instrumented programs, stopping when a difference (i.e., a bad value) is detected. If any bad value we encounter has not been seen before then use it as input for step ⟨1⟩, otherwise we are done.

For step ⟨1⟩, our prototype implementation of the algorithm uses a debugger to manage control of the serial and parallel programs at the location of the earliest known difference. We use this control to implement two optimizations that help us to reduce the number of re-executions necessary to search for differences. First, we observe that the set of suspect variables determined in step ⟨2⟩ were computed in previous program states, and are therefore candidates for examination by means of backtracking. Backtracking [1] is the retrieval of values without the use of re-execution. It is commonly implemented using tracing. The space overhead of tracing is undesirable, so we use a limited form of backtracking that utilizes only the current program state to retrieve values that have not been overwritten from

```
L1:  w2 = r1 + w1
     ...
     if ( ... ) then
L2:     w3 = r2 + w2
     else
L3:     w3 = r3 + u1
     endif
     ...
L4   u1 = 0
L5:  x = r4 + w3
```

Figure 1. Code for simple backtracking example.

previous program states. If any value retrieved in this way is found to be incorrect then we can use that value as input to step ⟨1⟩, saving us a re-execution.

For values of suspect variables that have been overwritten we use a second optimization. Such values are treated as if they were incorrect, and we speculatively apply steps ⟨1⟩ and ⟨2⟩ to the definition points of each to find if any of their right-hand side values can be retrieved with backtracking. Note that this optimization can be applied recursively for any right-hand side value that has been overwritten. If a bad value is found among the right-hand side values then we assume the overwritten value was incorrect as well, and continue our overall search for the first difference with the bad right-hand side value as input for step ⟨1⟩.

For example, suppose we have determined that the value of `w3` is bad in line L_5 in Figure 1. Suppose as well that the statements that could have computed `w3` are at lines L_2 and L_3. We would like to retrieve the values of `r2`, `w2`, `r3`, and `u1` to see if any of them may have caused the bad value of `w3`. Backtracking is successful for `r2`, `w2`, and `r3` because their values are still in the program state at line L_5. We cannot, however, retrieve the value of `u1` from line L_3 because it is overwritten in line L_4; we will need to instrument line L_3 to stop the program after a re-execution in order to access the program state at that line. Before we re-execute we continue backtracking if any of `r2`, `w2`, or `r3` are bad. Suppose `w2` is found to have a bad value, we then look to the statement at line L_1 that computed its value and apply backtracking to `r1` and `w1`.

This example uses only scalar values, but subscripted array expressions can be analyzed in a similar fashion assuming dependence analysis is employed. Such analysis may reveal that a single element of an array should be compared to proceed with relative debugging, but in other cases (e.g. array reductions) more than one element of an array is potentially incorrect. For the latter situation we simplify the approach by finding the "most incorrect" array value, where "most incorrect" is defined as the largest absolute difference. The dependence analysis utilized in our prototype handles backtracking for subscripted array expressions across procedural boundaries, even when the array is mapped to different numbers and sizes of dimensions.

3.2. Computation Mapping

The algorithm to find the first difference is a fairly straightforward utilization of interprocedural dependence information to navigate the potential causes of incorrect values. Determining whether a given value in the parallel computation is correct, though, may require extensive information about the mapping from the serial computation to the parallel one. We have broken down the computation mapping into several components that help to answer important questions we will have when performing comparisons.

- *Source-to-source mapping* provides information about where instrumentation should be inserted in the programs, such as file names and line numbers for breakpoints.
- *Execution mapping* builds upon the source-to-source mapping by revealing the parallel processes and/or threads that actually execute different portions of the program. For example, it is often the case that only one process/thread in a parallel execution performs file I/O, so any values related to file I/O would be retrieved from just that process/thread.
- *Iteration mapping* is used to address program statements that are executed more than once. The iterations of a loop in the serial program may be distributed across threads in the parallel program, for example, and this mapping answers the question "which thread performed iteration k?"
- *Data value mapping* builds on the execution mapping by describing how values in the serial computation are represented in the parallel one. In a trivial case, a serial scalar value might be replicated in each process and/or thread. A more complex case might involve a serial scalar value that is computed in parallel through some reduction of individual process/thread values. This mapping also accounts for any tolerance of inequality in values resulting from influences such as differing numerical methods or floating point representation.

Determining all these component mappings requires both static and runtime analysis. The source mapping is found statically, and the execution and iteration mappings are found at runtime. The data value mapping requires both analyses, since the type of parallelization (and therefore the representation of values in the parallel computation) is found in the source, but the actual distribution of values is of course performed at runtime.

3.3. Modifications for OpenMP

The use of OpenMP, or any other compiler-based parallelization that employs source code directives, complicates the search for the first difference in several ways. The execution model may differ across portions of the program, such as in OpenMP parallel regions that use multiple threads, complicating the determination of those components of the computation mapping that are found at runtime. Furthermore, comparing values inside regions affected by directives may require more sophisticated execution control and instrumentation to obtain values, and care must be taken to not affect execution with the “probe effect” of debugging.

Another complication is that directives may imply changes to the program beyond just the execution model. Directives can modify dataflow, such as OpenMP data scoping directives (e.g. FIRSTPRIVATE and LASTPRIVATE) that specify whether or not a given value flows into and/or out of a parallel region. Directives may also imply additional computation for the parallel program that is not represented in the source; we may need to know of any value changes associated with this computation. For example, OpenMP REDUCTION directives generate this type of computation and value change because the final reduction of individually computed thread values is nowhere represented in the source. The compiler generates instructions to perform a reduction across threads somewhere around the end of the reduction region.[1]

A comprehensive approach to automatic relative debugging of OpenMP programs must address all these issues in order to find the precise location of the first difference that may have caused the bad value the user has seen. Our work accomplishes an important first step in that approach by finding the first parallel region that contains a difference relative to the serial program. Finding the first troublesome parallel region can provide a focus for more intense analysis (see Section 7.1).

The simplified goal allows us to treat parallel regions as though they were single program statements, with values flowing into and out of the statements, because we do not attempt to resolve the cause of an incorrect value within a region. We can therefore restrict our value comparisons to serial sections of the parallel program. This avoids the need to understand the semantics of intra-region computation, even for computation that is implied by directives. No specialized execution control is required because no threads within parallel regions need to be inspected; values that flow into and out of parallel regions are checked only at region boundaries. Thus we effectively reduce the amount of the computation mapping needed to perform comparisons. The execution mapping is not needed at all, because parallel execution is limited to parallel regions. For the same reason we do not need the portion of the data value mapping that deals with representing serial values on multiple processes/threads.

```
      !$OMP PARALLEL PRIVATE(w2)
      !$OMP & SHARED(r1,r2,r3,w1,w3,u1)
L1:   w2 = r1 + w1
      ...
      if ( ... ) then
L2:      w3 = r2 + w2
      else
L3:      w3 = r3 + u1
      endif
      ...
      !$OMP END PARALLEL
L4    u1 = 0
L5:   x = r4 + w3
```

Figure 2. Code for OpenMP backtracking example.

For example, suppose we are debugging the code fragment in Figure 2 (created by adding OpenMP directives to the code fragment from Figure 1), and we have again determined that the value of `w3` is bad in line L_5. We would like to retrieve the values of `r2`, `w2`, `r3`, and `u1` that were used in lines L_2 and L_3, but at line L_5 only the values of `r2` and `r3` are unmodified. `u1` is overwritten in line L_4, and `w2` has an undefined value after the closing directive of the parallel region because `w2` is a PRIVATE variable within the region. After finding that `r2` and `r3` are correct we might want to re-execute to line L_3 to check `u1`, or to line L_2 to check `w2`, but both options require us to stop execution within the parallel region. Instead, we re-execute and stop the program before it has entered the parallel region. At this point we can check `u1`, because `u1` is SHARED in the region. The value of `w2` computed by each thread at line L_1 is never directly available to us because of the limited scope of `w2`, but we can check `w2` indirectly by looking at the values of `r1` and `w1`.

Our approach can also perform comparisons of parallel computations for the general case of M versus N threads ($M, N > 1$, $M \neq N$), since we only compare values in serial sections of the programs. Comparing two parallel computations can be simpler and take less time – simpler because the source mapping is likely trivial (typically the same source), and quicker because the amount of wall-clock time needed for re-execution of two parallel programs is likely less than for the case of a serial and parallel program. We expect our prototype implementation of the relative debugging approach will require little modification to support this general case, but more testing is needed.

[1]The final result of the reduction remains undefined until all threads have reached a barrier synchronization.

We need two pieces of information about parallel regions to successfully adapt the difference search and implement the steps described in the above example.

- We need to know about data scoping directives that modify dataflow at parallel region boundaries.
- We need the location of region boundaries to avoid stopping execution inside a region.

This information allows us to backtrack through parallel regions, re-executing when necessary to reach the beginning of a region. Our approach for obtaining this information is covered in the next section.

4. Prototype Implementation

We extended our distributed memory debugging prototype [11] to test our application of the relative debugging approach on OpenMP programs. The prototype comprises several modules:

- *ParaWise* (formerly called *CAPTools*) is a parallelization tool from the University of Greenwich [3, 10]. It performs sophisticated dependence analysis and array partitioning to convert programs for execution on distributed-memory machines.
- *CAPO* is a parallelization tool from NASA Ames [9], based on the *ParaWise* codebase. The user assists the tool in inserting OpenMP directives into a serial program based on the dependence analysis of *ParaWise*.
- *P2d2* is a debugger for parallel programs from NASA Ames [6]. It is portable across a variety of parallel machines and its user interface scales so that it is capable of debugging 256 processes or more.

P2d2 provides the user interface for the prototype, as well as execution control of the serial and parallel programs. A library encapsulation of *ParaWise* is used to obtain the necessary dependence analysis in one of two ways, depending on the method of parallelization of the program being debugged. For tool-parallelized programs created using *CAPO*, the dependence analysis is retrieved from a database created when the program was parallelized. For hand-parallelized programs the library performs dependence analysis, using a conservative level of analysis to reduce the time the user must wait for relative debugging to begin. This conservative analysis is fully automatic, it does not exploit user inputs (e.g. knowledge of program input) to refine the dependence graph.

A library encapsulation of *CAPO* is used to provide the necessary OpenMP directive information described in Section 3. For tool-parallelized programs the library derives directive information from the parallelization database. For hand-parallelized programs the library parses and analyzes directives in the program source code.

The user is prompted for information about an incorrect value they have found in the parallel execution, and the automated search for the first difference begins after all necessary dependence and directive information is obtained. The steps of the relative debugging algorithm are performed by the *ParaWise* library and *P2d2*; *ParaWise* guides backtracking and re-execution based on the dependence analysis and *P2d2* retrieves and compares values. When necessary, *ParaWise* makes use of directive information by asking the *CAPO* library two questions:

- *is variable* `V` *at line* `L` *shared?* – All variables outside of parallel regions are by definition shared[2]. Variables inside parallel regions are considered to be shared if their value flows in and out of the parallel region, allowing *P2d2* to retrieve the value when the program is stopped at some point outside of and after the region. See Figure 2 and its explanation in Section 3 for an example of this.
- *where is the beginning of the parallel region enclosing line* `L`*?* – If line `L` is within a parallel region the first line of that region is returned by *CAPO*. *ParaWise* uses this to make sure it does not instruct *P2d2* to stop execution and test values in a parallel region, which would violate our decision to treat parallel regions as if they were single program statements.

4.1. Determining the Computation Mapping

To maintain the automated nature of our prototype we must determine, without user involvement, those portions of the computation mapping that we need. The data value mapping is approximated by considering values to be equal if they are within some tolerance, to account both for differences in numerical method (for operations such as reductions) and for differences in compilers. The execution mapping is not required, because we do not compare values in parallel regions, and our constraints on the source mapping are described in the next subsection.

Our prototype's handling of the iteration mapping has advanced significantly since our previous work. We now appropriately match, in most cases, instances of program statements based on callsite as well as local iteration number (i.e. loop counters used in the right-hand side of the

[2]An exception to this is variables declared THREADPRIVATE, because such variables are allocated for each thread, and outside parallel regions only the master thread's variable is accessible/shared. We have no need to access the variables of non-master threads outside of parallel regions, so all variables outside parallel regions, from our perspective, are effectively shared.

```
      program test
        ...
L1:     call work(a,b,2)
L2:     call work(a,b,3)
L3:     call print_results(a)
      end
      subroutine work(a,b,k)
        ...
L4:     do i=1,imax
L5:       do j=1,jmax
L6:         a(i,j,k) = 1 / b(i,j,k)
L7:       enddo
L8:     enddo
        return
      end
```

Figure 3. Finding the Iteration Mapping.

program statement). For example, suppose we want to compare the value of $b(1,1,3)$ at line L_6 in Figure 3. Instances of line L_6 can be identified using both callsite and local iteration number. Static analysis reveals that only the call to `work` from line L_2 will make use of $b(1,1,3)$, and once inside that call we can evaluate i, j, and k to insure we are comparing at the correct instance.

The tests for callsite and local iteration number are performed whenever the serial or parallel programs reach a program statement that requires instance matching. *ParaWise* uses call graph information to determine the stack trace a desired statement instance will have, and passes this information to *P2d2* for each instrumentation that is requested as a part of step $\langle 3 \rangle$ in the algorithm from Section 3.1. *P2d2* uses this to test the stack trace each time a program arrives at a given statement. If the callsite is confirmed, *ParaWise* instructs *P2d2* to retrieve the loop counters used in that statement in order to test the local iteration number.

Placing instrumentation inside loops or other oft-repeated constructs can be extremely costly, so we have implemented an optimization for our instance matching that saves time in cases such as the nested loops in Figure 3. We observe that $b(1,1,3)$ is never overwritten in the nested loop beginning on line L_4, so it suffices to place instrumentation to check $b(1,1,3)$ at line L_4 and merely test the stack trace when that instrumentation is reached.

4.2. Limitations

Our prototype does not yet address a couple of possible program behaviors resulting from insertion of OpenMP directives. See Section 7 for our plans to address these issues.

- Non-determinism, both intended and unintended, may in some cases prohibit use of our approach due to execution order and value changes upon re-execution. In general, we require an approximation of replayability defined in the following way: if V is the set of all variables that we might retrieve values from in serial sections of the program, such that $v \epsilon V$ is the tuple (variable name, subscript, line, iteration, ...) that uniquely describes a variable in a single program state, then for all executions the value of v, $\forall v \epsilon V$, must be equal.

- Improper construction and/or placement of directives can result in use of uninitialized values. For example, a variable declared PRIVATE in a parallel region may be used immediately after that region without an intervening definition. In these cases the dependence graph is effectively altered, and our dependence analysis does not yet detect these changes.

Similar to the distributed memory debugging prototype, we assume the source mapping from the serial program to the parallel program is trivial. Relative debugging is performed on programs that have the same source code, for OpenMP this reduces to comparing a one-thread execution with an N-thread execution. Note that this guarantees we will only locate sources of difference in parallel regions, because if it were not for the multithreaded execution of parallel regions the two executions would be the same. Future work on source mapping determination will alleviate this restriction.

Our approximation of the iteration mapping does not yet handle statement instances that must be identified with a non-trivial global iteration number (i.e. loop counters identify a particular statement instance, but not all those counters are used in the right-hand side of that statement). For example, a program could perform many iterations of a number of statements to achieve some convergence requirement, but the iteration counter need not be used in the right-hand sides of those statements. In general, counters may exist as global variables, or even be implemented with GOTO statements, and therefore can be difficult to pinpoint; we are investigating how our current dependence analysis can be leveraged to address such scenarios.

The directive analysis we utilize to determine whether variables are shared (and therefore retrievable) is not yet comprehensive, in that we assume variables are only shared when their values flow in *and* out of a parallel region. More accurately, values may be retrievable before and/or after a region, depending on the storage class that defines the flow into/out of the region. We will enhance the prototype to query *CAPO* for the separate cases of values flowing into a region (e.g. FIRSTPRIVATE), out of a region (e.g. LASTPRIVATE), or both.

5. Related Work

Guard [18] is a relative debugger that relies on user assertions about value equivalence to direct the comparison of values between two programs. Assertions are given as input to the debugger, and *Guard* then runs the programs side-by-side, checking for any assertion violations.

DUCT [15] is a tool used with *VSGuard* [4] to analyze use define chains during relative debugging for Microsoft's Intermediate Language (MSIL). *VSGuard* and *DUCT* improve upon *Guard* by allowing the user to quickly formulate assertions for each step of the relative debugging process based on use define chains in a GUI.

Intel's *Assure* [8] attempts to predict errors that will occur in a parallel execution of an OpenMP program. Static and dynamic analysis of a serial execution of the program is used to identify potential storage conflicts and data races. See Section 6.1 for a comparison of results from *Assure* and our prototype for the same test case.

Sun ONE Studio 8 [16] offers an OpenMP-aware f95 compiler that can perform interprocedural dependence analysis to find any parallelization inhibitors, but this analysis seems limited to performance enhancement without providing bug detection.

HP *Visual Threads* [5], based on *Eraser* [14], provides a way to verify proper locking discipline when lock-based synchronization is used. OpenMP programs can be analyzed by *Visual Threads*, but it is unclear how helpful that analysis would be because many OpenMP programs do not make heavy use of lock-based synchronization.

IBM Rational *PurifyPlus* [7] performs memory and performance profiling, and some versions detect deadlocks and race conditions in multithreaded programs. OpenMP is not directly addressed by the tool, though, so the benefits to our intended users are unclear.

6. Example

We tested the prototype's ability to debug both tool- and hand-parallelized programs by applying it to the NAS Parallel Benchmark LU [12]. A serial version of LU was parallelized using *CAPO*, with erroneous user inputs given to *CAPO* so that a parallel program was produced that failed to generate correct output when executed with more than one thread. LU contains a subroutine `verify` that checks the output values and reports to the user whether the benchmark executed correctly. One of the values that triggers a failure in `verify` is used as the starting point of the search for the first difference.

Backtracking from the bad value in `verify` to the first parallel region that causes the bad value requires the prototype to search backwards through at least 3 subroutines. See Figure 4 for the relevant portion of LU's call graph.

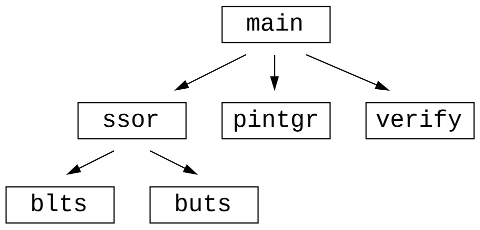

Figure 4. Portion of call graph of LU.

The guilty parallel region is in subroutine `buts`, which is called by subroutine `ssor`, and values from `ssor` are used in `pintgr` before they eventually reach `verify`. Re-execution must be used several times to access program states in these subroutines.

Testing the prototype on a tool-parallelized program is straightforward – we simply use the program and parallelization database generated by *CAPO*. After 3 iterations of inserting instrumentation and re-running one-thread and two-thread executions (plus the initial execution of each to confirm the user's bad value), the prototype correctly determines the guilty parallel region. On our test machine (dual Pentium III 933MHz), using LU's class S benchmark size, this debugging session takes about 20 seconds. For reference, complete execution of the serial and parallel versions takes 0.7 seconds and 0.5 seconds, respectively. Our prototype continues execution of only one program at a time, so an ideal iteration with no overhead would take at most 1.2 seconds, for an ideal session time (4 executions) of 4.8 seconds.

The prototype's operation on a hand-parallelized program is tested by treating the same tool-parallelized program as though it were hand-parallelized. The prototype is not given access to the database of parallelization information, and therefore must perform its own dependence and directive analysis to obtain the necessary information for the relative debugging algorithm. The correct result is again obtained after 3 iterations of inserting instrumentation and re-running the programs, but more backtracking is required due to the conservative dependence analysis. The increase in the number of values retrieved and compared is roughly five-fold over the tool-parallelized case. Total time for this debugging session is 75 seconds, of which about 26 seconds is dependence and directive analysis.

Our test version of LU actually contains two data races, one intended and one unintended. To our knowledge both have produced the same sequence of instructions for every parallel execution we have performed, and have therefore met our replayability requirements. The unintended data race is contained in the parallel region in `buts` that is reported to the user as the location of the first difference, and is indeed the cause of the bad value in `verify` that the user sees when running with more than one thread.

6.1. Comparison to *Assure*

We analyzed our LU test case using *Assure* [8] to see if its results matched, or were superior to, those given by our prototype. In order to make use of *Assure*, though, we first had to modify the test case to eliminate any uses of the OpenMP runtime library routine `omp_get_thread_num`. *Assure* performs its analysis by simulating a parallel execution using only an instrumented serial execution of the program, and use of `omp_get_thread_num` somehow prevents this simulation. Our original LU test case used `omp_get_thread_num` for point-to-point synchronization to achieve pipelining of threads in routine `blts`, to protect access to shared array `v`; we modified `blts` to instead achieve pipelining using a shared array of flags.

Assure found a total of six problems that our version of LU might have when run in parallel. Three of these related to variables declared PRIVATE in parallel regions, in practice these do not cause errors in parallel executions. The remaining three are all races – two races in `blts`, and one in `buts`. The races in `blts` are for the two shared arrays mentioned earlier, our array of flags and `v`. We fully intended the first race to occur (and tested to make sure it occurs correctly), and because of this we know the accesses to `v` are in fact synchronized. The last race, in `buts`, is the true cause of error in parallel executions of the program. The parallel region that encloses this race is the same region reported by our prototype. The *Assure* session takes over 200 seconds for the serial execution alone, plus additional time for the program modification, instrumentation and compilation, and processing of the output files generated by the instrumentation.

Assure's analysis is helpful when determining the cause of problems in parallel executions of this version of LU (suitably modified by the user), but even in this small test case it can be seen that *Assure's* error report potentially obscures the result the user needs to see. If there were many races such as those intended in `blts`, the user would have to verify each systematically in order to rule it out as a cause of the bad value they observed. Our prototype instead directs the user to a parallel region that definitely produces different values in a parallel execution than in a serial one. The user must decide if they require *Assure's* more general overview of possible issues in their program, our more accurate analysis of a particular issue, or the use of both these tools.

7. Algorithm and Prototype Extensions

We have plans to address non-determinism with a race-detection tool for OpenMP that would complement our relative debugging work. This tool could verify that our replayability conditions are met, and possibly enforce that replayability if a program proved to be more non-deterministic than we would like. A user could also address non-determinism through other tools such as *Assure*.

We believe that usage of uninitialized values, due to improper construction and/or placement of OpenMP directives, can be detected and readily fixed based on knowledge of correct dataflow in the serial program, directive information, and runtime information. It would be interesting to see how such analysis might compliment, or prove superior to, that of *Assure*.

7.1. Identifying Problems within OpenMP Parallel Regions

An important next step for OpenMP automatic relative debugging is to resolve problems within parallel regions. Checking for potential dependence violations in parallel regions could be performed using static analysis on the entire code, but this taxes the user with weeding out false positives due to the conservative nature of dependence analysis and lack of any runtime information. The identification of a problem in a particular instance of a specific parallel region should enable us to dramatically reduce the number of false positives.

Only those statements in a parallel region that are related by dependences to an erroneous variable need to be investigated, and the incorporation of runtime values provided by *P2d2* can even further reduce the number of potential errors presented to the user. In addition, the decisions made during OpenMP parallelization (whether user- or tool-based) can be checked against the automatic directive generation algorithms used in *CAPO* so that other potential problems may be identified. Constructs such as OpenMP reduction variables, and the use of optimizations like the NOWAIT directive modifier, can be checked for correctness to warn the user in suspect cases.

For tool generated parallel code, the decisions made by the user in the parallelization process (e.g. dependence deletions) can be checked against the potential problems identified in the parallel region. If any user decision directly affected the parallelization (e.g. allowing a variable to be privatized), then that decision can be relayed to the user for re-examination.

Performing backtracking and value comparisons inside parallel regions will be difficult for a couple of reasons. First and foremost, the execution mapping may be quite intricate due to directives such as SECTIONS, ORDERED, and WORKSHARE that allow a fine-grain division of work among threads. We may need to query threads at various locations to determine which is performing a given portion of the computation. For PARALLEL DO loops it may suffice to require a STATIC scheduling of iterations among threads, so that the distribution of iterations can be queried before the DO loop executes.

Another potential issue is barriers, both implicit and explicit, that block progress of threads we need to advance to reach the next instrumentation location. It should be possible to instrument most of these locations so that the prototype is notified when a thread cannot make further progress, but other scenarios such as point-to-point synchronization of threads may be hard to detect and instrument automatically.

8. Conclusions

In this work we have successfully extended the automatic relative debugging approach for use on shared-memory OpenMP programs, providing the user with a means to find differences in computation between both a serial program and an OpenMP parallel version of that program, and between M-thread and N-thread executions of an OpenMP program ($M, N > 1$, $M \neq N$). Using backtracking and re-execution we search through the execution history of both programs to find the first difference that may have caused an incorrect value the user has seen. The location of parallelization error(s) is bracketed to the first parallel region of the erroneous program that contains a difference relative to the reference program.

The approach is applicable to OpenMP programs created by either tool- or hand-parallelization methods. For tool-parallelized programs we utilize information from the tool to enable backtracking and value comparisons. Hand-parallelized programs are analyzed to obtain the necessary data dependence and directive information, allowing the user to begin relative debugging without having to furnish this information. In both cases the user need only provide an incorrect value from the parallel execution to initiate the difference search.

Acknowledgments

We thank Pete Leggett and Emyr Evans for their contributions to the prototype, both in implementation and testing. We greatly appreciate the help of Shirley Moore, Tom Cortese, and Angela McClure, at the University of Tennessee, for performing the Assure analysis. We also want to thank Parkson Wong and Rob Van Der Wijngaart for reviewing the paper and making suggestions for improvements.

References

[1] Agrawal, H. “Towards Automatic Debugging of Computer Programs.” Ph.D. Thesis, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1991. `http://citeseer.nj.nec.com/134139.html`.

[2] Computer Aided Parallelization Toolkit (*ParaWise*). `http://www.parallelsp.com`.

[3] Evans, E. W., Johnson, S. P., Leggett, P. F., and Cross, M. “Automatic and Effective Multi-Dimensional Parallelisation of Structured Mesh Based Codes.” *Parallel Computing 26*, pp 677-703, 2000.

[4] Guardsoft. *VSGuard*. `http://www.guardsoft.com`.

[5] Hewlett-Packard Company. *Visual Threads*. `http://h18000.www1.hp.com/products/software/visualthreads`.

[6] Hood, R. “The *p2d2* Project: Building a Portable Distributed Debugger.” *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996, Philadelphia, PA, pp. 126-137.

[7] IBM Corporation. Rational *PurifyPlus* Family. `http://www.rational.com/products/pqc/index.jsp`.

[8] Intel, Incorporated. *KAP/Pro*. `http://www.kai.com/parallel/kappro`.

[9] Jin, H., Frumkin, M., and Yan, J. “Code Parallelization with CAPO—A User Manual.” NAS Technical Report, `http://www.nas.nasa.gov/Research/Reports/Techreports/2001/nas-01-008-abstract.html`.

[10] Leggett, P. F., Marsh, A. T. J., Johnson, S. P., and Cross, M. “Integrating User Knowledge with Information from Parallelisation Tools to Facilitate the Automatic Generation of Efficient Parallel FORTRAN Code.” *Parallel Computing 22*, pp 259-288, 1996.

[11] Matthews, G., Hood, R., Johnson, S. P., and Leggett, P. F. “Backtracking and Re-execution in the Automatic Debugging of Parallelized Programs.” *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[12] The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB`.

[13] OpenMP Specifications. `http://www.openmp.org/specs`.

[14] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. “Eraser: A dynamic data race detector for multi-threaded programs,” *ACM Transactions on Computer Systems* (TOCS), 15(4):391-411, November 1997.

[15] Searle, A., Gough, J. K. and Abramson, D. A. “DUCT: An Interactive Define-Use Chain Navigation Tool for Relative Debugging,” *Proceedings of AADebug 2003*, Ghent, Belgium, 2003.

[16] Sun Microsytems, Inc. *Sun ONE Studio 8*. `http://docs.sun.com/source/817-0933/3_Compiling.html#pgfId-998346`.

[17] Veridian, Incorporated. *VAST*/Parallel Fortran and C Automatic Parallelizing Preprocessors. `http://www.psrv.com/vast_parallel.html`.

[18] Watson, G. and Abramson, D. “The Architecture of a Parallel Relative Debugger.” *Proceedings of the 13th International Conference on Parallel and Distributed Computer Systems*, Las Vegas, Nevada, August 2000.