6 ccc 84
P. 12

Backtracking and Re-execution in the Automatic Debugging of Parallelized Programs

Gregory Matthews, Robert Hood,
Computer Sciences Corporation
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035 USA
{gmatthew, rhood}@nas.nasa.gov

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

Abstract

In this work we describe a new approach using relative debugging to find differences in computation between a serial program and a parallel version of that program. We use a combination of re-execution and backtracking in order to find the first difference in computation that may ultimately lead to an incorrect value that the user has indicated. In our prototype implementation we use static analysis information from a parallelization tool in order to perform the backtracking as well as the mapping required between serial and parallel computations.

1. Introduction

As the number of parallel computers increases, so does the demand for converting existing programs into parallel form. There are three general approaches to the conversion process:

- manual translation, where explicit parallel and communication constructs are added to the code;
- compiler-based parallelization, where the user employs source code directives [10, 14] to steer the compiler into producing efficient parallel code; and
- fully interactive parallelization tools, where user inputs steer the parallelization process [3 11].

Regardless of the approach used, the porting process is error-prone. Even in the more automatic alternatives the user is providing information, and it is mistakes in that information that leads to bugs in the parallel program. For example, the user might incorrectly indicate that a loop can be safely run in parallel by deleting an essential edge in the dependence graph. The resulting program may have race conditions in a shared-memory version, or use values of variables that are not up-to-date in a distributed-memory version.

Finding bugs introduced during parallelization can be very complicated. Often the user must manually run the serial and parallel programs side-by-side to try to find out where the two computations differ. This technique likely involves numerous executions of the programs in an attempt to locate the first difference that potentially led to all subsequent differences.

In this paper we describe our approach for automating the manual debugging technique described above, as applied to distributed-memory programs. We begin by discussing our general approach, including the information required and the difference detection algorithm. In Section 3 we describe our prototype implementation, and following that we give an example of its use. In Section 5 we discuss ways to extend our work. We then discuss related work and draw conclusions.

2. Finding the First Difference

The manual technique of running the serial and parallel programs side-by-side involves deciding where to put breakpoints and, at breakpoints, deciding what values to compare. For example, if the parallel program is printing a wrong value, the user can look at the source code and follow enough statements backward from the print statement to see where the incorrect value was calculated. He could then insert a breakpoint at that definition point and re-execute the program to see what values used in the calculation are incorrect. He can repeat this process of following statements backward until the source of the error is reached.

Automating this search for the first difference between the computations of a serial program and its parallelized version requires two important elements:

1. a method to drive the search that decides which value comparisons are useful, and
2. the ability to make comparisons between the serial and parallel for the values we choose to compare.

The first element can be provided by combining the user's observations of program behavior with *data dependence analysis*, which describes how values get created and used within a program. The possible definition points of a known bad value can be found, and further comparisons can be performed on the values involved in those definition points.

The second element requires determining how the serial computation has turned into a parallel one. This *computation mapping* answers the important comparison questions:

- Where to compare — the program statements where the serial and parallel processes should be instrumented to perform comparisons;
- In whom to compare — the processes in the parallel execution containing the values to be compared;
- When to compare — the iteration count at which the comparison should be made in each process; and
- How to compare — the comparison function that describes how to construct the values to be compared (e.g., obtain the checksum of a distributed array), and then describes how to determine "equality" of the values.

In the remainder of this section we discuss the computation mapping information we need, then describe our algorithm and how it makes use of this information.

2.1. Computation Mapping

The computation mapping from a serial program to a parallelized version of that program can be broken down into several components. Here we discuss each component, and the manner in which they facilitate answering the comparison questions listed earlier.

- *Source-to-source mapping* can be used to answer the "where to compare" question. It is a description of location correspondence in the serial and parallel source code where we can expect values to be comparable. For example, we may want to place instrumentation breakpoints at one line in the serial code and at a corresponding line in the parallel code. This mapping information likely goes beyond simple line correspondence, however, since the parallel source code may differ significantly from the serial source code
- *Execution mapping* answers the "in whom to compare" question. This information builds on the source mapping given above, and describes which parallel processes actually execute the program statements in the parallel source code that map to a certain location in the serial source code.

  For example, if the serial program performs an initial phase of file I/O, a typical execution mapping will inform us that the corresponding file I/O code in the parallel program is confined to the first process. Often this mapping can only be determined at runtime, when the number of parallel processes is known and the computation has been split up among those processes.
- *Iteration Mapping* answers the "when to compare" question. If a value we wish to compare is located in a program statement that is executed more than once, we rely on iteration mapping information to tell us which of those executions should be instrumented to compare the value. Multiple executions of the same statement may occur because of loop constructs, or perhaps because of multiple subroutine executions (e.g., recursion).
- *Data value mapping* is used to answer the "how to compare" question. This mapping builds on the execution mapping by providing a description of how serial-side values may be represented in the parallel computation, and also provides an equality function with which to compare values. For example, if the computations perform a sum reduction on a vector, the serial program may have a variable `S` whose value is actually the sum of the `S` values found in the parallel processes. Furthermore, when comparing the value of variable `S` in the serial process to the sum of `S` values in the parallel processes, we might consider them equal if they agree within some tolerance that accounts for differences in numerical method.

  A notable subset of this mapping is data distribution information. The description of how variables, arrays in particular, are distributed across multiple address spaces is important for debugging parallelized programs. For example, it must indicate how a serial-side array index expression gets mapped into a process number and a list of indices in the parallel computation. If data decomposition is employed in the parallel computation, the data distribution description must contain information about "ghost points" and other issues relating to data on the boundaries of the decomposition.

We next describe how these pieces of information are utilized to produce an effective approach to relative debugging.

2.2. Algorithm

In this work we have automated the manual technique for comparison debugging that was described at the beginning of this section. When a user indicates a bad value in the execution of the parallel program, we perform the following steps.

⟨1⟩ Find the possible definition points of the incorrect value using dependence analysis information.

⟨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⟩ While there are new suspect variable references to monitor

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

⟨3b⟩ Execute the instrumented programs, stopping when a difference (i.e., a bad value) is detected.

⟨3c⟩ If the bad value has not been seen before then use it to determine a new set of suspect variable references to instrument (as was done in steps ⟨1⟩ and ⟨2⟩ above); otherwise allow loop to terminate.

When the loop stops we have identified the first difference between the two computations that may lead to the bad value identified by the user. While this strategy works, we can improve upon it by using a limited form of backtracking [2]. In particular, if we can determine the value that a new suspect variable would have at a potential instrumentation point, we can avoid the re-execution that takes place in step ⟨3⟩.

Although backtracking improves our efficiency substantially, its typical implementation—checkpointing program variables before they get overwritten during execution—is too costly. Instead, we are satisfied to limit the scope of our queries about previous program states to those that can be answered using only information in the current state. If a variable value created at one point in the program has not been killed by subsequent execution, we can evaluate it and determine either that it is OK or bad. If it is OK, then we can ignore it and concentrate on other values. If it is bad, we can look for values used in its definitions (as is done in steps ⟨1⟩ and ⟨2⟩). If a variable value has been killed we can simply add the variable to the list of suspect references to be monitored during a re-execution.

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 definitions of w3 that may reach that point are at L_2 and

```
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. Simple backtracking.

L_3. In that case, we will attempt to evaluate the variables r2, w2, r3, and u1[1]. If we cannot evaluate one, say u1, because its value is killed between lines L_3 and L_5, then we will add the use of u1 at line L_3 to the list of suspect references that need to be instrumented in the next re-execution. If we evaluate a variable, for example r2 or r3, and find that the value matches the serial value, then we can ignore it. If the value of one, say w2 in line L_2, is not the same, then we look at the definitions of w2 that reach the use in L_2. In this case suppose the only definition is at line L_1. We would then look at the variables r1 and w1 to see if they need to be added to the list of suspect references. Suppose r1 is correct and that w1 is incorrect. Then the algorithm will proceed by evaluating the definition points of w1 that reach line L_1.

We can potentially further reduce the number of re-executions by speculatively examining the definition points of unknown values. In the example given above, the value of u1 in line L_3 is killed by the assignment at line L_4. Since we couldn't directly determine the value used at line L_3, we simply instrumented the use of u1 at that point in order to do a comparison in the next re-execution. However, we can look at the definitions of u1 that reach line L_3 to see if any of them use bad values. If we find a bad value, we can continue the search for differences at that point, potentially saving us a re-execution of the program.

Note that the accuracy of the dependence information plays an important role in the efficiency of the algorithm. In particular, having accurate interprocedural information will allow us to examine backwards through procedure calls. Consider the example in Figure 2. Suppose we find out that the value of w7 is incorrect at line L_8. Accurate interprocedural information could tell us that the statement at line L_9 might define the value of w7 that reaches L_8. (Note that the test for a definition kill of the variables used at L_9 must check for kills from L_9 to L_{10} as well as check from L_7 to L_8.) If no kills are found, we can evaluate b and r7.

[1] In this and the following example, we use a mnemonic for the reader's convenience: the names of variables with wrong values begin with a "w," right values with an "r," and untestable values with a "u."

```
              ...
L6:           w6 = u6 + u7
              ...
L7:           call sub(w7, w6)
              ...
L8:           x = w7 + r6
              ...
        end
        ...
        subroutine sub(a, b)
              ...
L9:           a = b + r7
              ...
L10:    end
```

Figure 2. Interprocedural backtracking.

Suppose we find that `r7` is correct, but `b` is not. The interprocedural information could tell us that the statement at L_6 defines a value for b that might reach L_9. We can then continue the backtracking by looking at the variables on the right-hand side of L_6. In the absence of accurate interprocedural information we would have had to treat the call at L_7 as a potential definition, and use, of all actual parameters as well as any global variables (i.e., those in common blocks in Fortran.) In that case we might have had to instrument every statement in `sub` in order to find an erroneous definition of w7.

Although our examples so far have used only scalar variables, our approach also works when subscripted array expressions are present. Having the results of a sophisticated data dependence analysis phase is critical to being able to follow the flow of values in arrays from their definition to their use.

In summary, the algorithm we have proposed utilizes information from static analysis in three ways:

1. to find the possible definition points of a value V being used at some location L,
2. to enumerate the values used in a definition at some location L, and
3. to determine if a value V which is used in location L_1 is still available in the program state at location L_2.

For the correctness of our algorithm, the answers to these questions must be conservative. In particular, we must get *all* of the possible definition points in (1) and *all* of the values used in a definition in (2). For (3), we must only get a "yes" answer if the value definitely survives. The more accurate this static information is, the more efficiently our difference finding algorithm will perform. Overly conservative responses could result in extra instrumentation points or re-executions.

3. Prototype Implementation

We prototyped the algorithm of the previous section by extending two existing tools to cooperate with each other.

- Computer Aided Parallelization Tools (*CAPTools*) is a parallelization tool from the University of Greenwich [5, 11]. The user assists the tool in converting serial programs into a form suitable for execution on a distributed-memory machine. It performs sophisticated dependence analysis, partitions array data, and inserts needed communication calls.

- *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.

In this section we first describe how these existing tools were extended to produce the prototype. We then discuss some shortcuts we took in the implementation in the interest of speedy development.

3.1. Putting the Pieces Together

In the prototype, *CAPTools* is used to create the parallel program whose correctness is dependent upon correct user interactions. The results of dependence analysis, data partitioning, and parallel communication insertion are deposited into a database. A library encapsulation of *CAPTools*, with modifications to accommodate the information needed in this prototype, is used as the comparison algorithm proceeds.

P2d2 acts as the user interface. The user is queried for an initial variable name and a location where that variable is used but appears to have the wrong value. *P2d2* runs the serial and parallel programs to verify this difference exists, then

- retrieves a list of instrumentation points from *CAPTools*,

- inserts breakpoints in the serial and parallel programs at those points, and

- reruns the programs, checking the appropriate values at each breakpoint.

These steps are repeated as long as new, earlier, differences of interest are found in the values being checked at breakpoints. If a re-execution results in the same first difference detected, then the algorithm terminates.

To perform the first step above, *p2d2* holds a conversation with *CAPTools*. The debugger initiates by requesting from *CAPTools* a list of instrumentation points that might lead to the bad value just found, where the bad value is described with the pair

⟨variable name, location in source code⟩

for the serial version of the program. In response, *CAPTools* examines its database of dependence information, using the algorithm described in the previous section, looking for the possible definition points of the bad value. It handles this even if the value flows across procedure boundaries and gets remapped in going from caller to callee. For each of the definition points it uses a range of dependence information throughout the relevant call stacks to determine if the values used in the definition are still live, and then makes requests of *p2d2* for different evaluations to determine the correctness of values in the parallel processes.

Evaluations in the serial program are straightforward. In the parallel program, however, there are varying levels of complexity to overcome in determining the data value mapping needed for performing evaluations. In the simple case, a value may be duplicated in all parallel processes, or perhaps in a statically defined subset of these processes. *CAPTools* need only indicate to *p2d2* the processes of interest, and the stack frame in each process where the value can be found. A more complex case involves a dynamically defined mapping from a serial value to a parallel value. *CAPTools* must then provide *p2d2* with a description of the mapping in order to facilitate such evaluations. For example, a request for an evaluation of an array that is partitioned in the parallel program would include a description of the partition boundaries that hold for each parallel process.

When *CAPTools* has finished its examination of the static and dynamic information, having exhausted all backtracking possibilities from live variables, it returns to *p2d2* a list of 3-tuples

⟨variable, location in source code, process⟩

that need to be instrumented in a rerun of the serial and parallel programs. Each entry in this list takes into account the computation mapping from serial to parallel, using previously evaluated dynamically defined mapping attributes if necessary. The conversation between *CAPTools* and *p2d2* for the first step of this iteration of the algorithm ends at this point. The remaining steps of inserting breakpoints at the instrumentation locations and rerunning the programs are then performed by *p2d2*.

When there are no new interesting differences found between the computations, *p2d2* returns to the user the most recent difference discovered by the algorithm, which is the first that occurs between the computations that may have caused the bad value reported by the user.

3.2. Limitations of the Prototype

In the interest of speedy implementation, we limited the need for the computation mapping information described in Section 2.1.

- We temporarily eliminated the need for a source-to-source mapping by restricting comparisons to computations of programs that have the same source code. Essentially we take a parallelized program that has correct behavior when run with one process but incorrect behavior when run with N processes, and compare the two computations.

- The prototype does not yet perform sophisticated iteration mapping. The test executions that we are making are constructed so that the mapping is either not needed, or the effect of not having the mapping is minimal (i.e., we know what the result would be if we did have iteration mapping).

- The prototype does only a partial job of data value mapping. *CAPTools* takes care of mapping variable references from the sequential execution to their location in the distributed address space. We do not yet, however, perform any comparisons other than tests on simple expressions. For example, we do not aggregate values from the parallel execution into a single value for comparison against a value in the sequential run. In addition, we currently perform only strict equality tests of the values being compared. Furthermore, the tests are done on the ASCII strings the debugger uses to print the values, not the bit representations of the values.

In Section 5 we describe our plans for removing these restrictions.

4. Example

In order to illustrate the power of automated relative debugging, consider the following scenario. The user has parallelized a serial version of the NAS Parallel Benchmark program LU [12] using *CAPTools*, implementing a 1-D decomposition in the "J" dimension. In doing so, he inadvertently introduces errors in the parallel program by instructing the parallelization tool to ignore some dependences. When the resulting parallel version is executed using just one process, program output exactly matches that of the serial version and every verification test in the benchmark succeeds. Executing the parallel version with two processes, however, results in different program output and failure of every verification test.

```
subroutine ssor()
   ...
   do k=nz-1,2,-1



      call buts(v,k)
   enddo
   ...
   return
end
subroutine buts(v,k)
   ...
   call cap_receive(v(1,1,high+1,k),
                    ...,cap_right)
   do j=high,low,-1
      do i=nx-1,2,-1
         do m=1,5,1
            tv(m,i,j) = f(v,tv)
            tmat(m,1) = d(m,1,i,j)
            ...
         enddo
         ...
         tv(1,i,j) = tv(1,i,j)/tmat(1,1)
         v(1,i,j,k) = v(1,i,j,k)-tv(1,i,j)
         ...
      enddo
   enddo
   ...
   call cap_send(v(1,1,low,k),...,cap_left)
   return
end
```

Pipelined (unmodified dependences)

```
subroutine ssor()
   ...
   do k=nz-1,2,-1
      call cap_exchange(v(1,1,high+1,k),
                        v(1,1,low,k),
                        ...,cap_right)
      call buts(v,k)
   enddo
   ...
   return
end
subroutine buts(v,k)
   ...



   do j=high,low,-1
      do i=nx-1,2,-1
         do m=1,5,1
            tv(m,i,j) = f(v,tv)
            tmat(m,1) = d(m,1,i,j)
            ...
         enddo
         ...
         tv(1,i,j) = tv(1,i,j)/tmat(1,1)
         v(1,i,j,k) = v(1,i,j,k)-tv(1,i,j)
         ...
      enddo
   enddo
   ...

   return
end
```

Fully parallel (removed dependences)

Figure 3. Effects of dependence modifications on parallelization.

4.1. User Interaction with the Prototype

The user first looks at the verification subroutine to find what values might have caused failures in the two-process execution. One of the verification tests is a check of whether a scalar variable in the program has the correct value, and so this variable and source code location is entered into *p2d2* to begin the relative debugging algorithm.

After our prototype has performed five iterations of inserting instrumentation and re-running the one-process and two-process executions, it reports that the first difference between the computations is in the array `tv` at line 272 of subroutine `buts`. (Note that the first difference the prototype should have reported is in the array `v` at this location. See the next subsection for our discussion of this.)

With this information in hand, the user opens *CAPTools*, loads the database of information generated when the parallel version of the benchmark was created, and begins to scrutinize the data dependence choices that were made for subroutine `buts`. *CAPTools* reports that the original analysis of the serial source code, before any user modification of dependences, indicated the subroutine might be amenable to partial parallelization using a pipeline, but not to a full parallelization. During the parallelization process, however, the user erroneously removed dependences for several variables (including `tv`) within the subroutine. This allowed a full parallelization without the need for communication within the subroutine body to update values. See Figure 3 for a code comparison.

These modifications are plausible candidates for the cause of the incorrect behavior. A parallelized version of the subroutine would behave similarly to the serial version when executed with one process, because the degenerate case of one parallel process does not encounter the difficulty of stale values. If a dependence modification was indeed made incorrectly, then stale values could become a problem

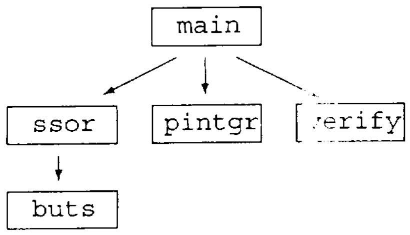

Figure 4. Portion of call graph of LU.

when two or more processes were used in the parallel execution. Parallelizing the serial source code without these user-introduced errors generates a parallel program that behaves correctly when executed with any number of processes.

4.2. Behind the Scenes

Our tests with the LU benchmark were conducted with the "sample" class size (i.e., class S). It uses a 12 × 12 × 12 mesh and is executed on 2 processes. Thus, for a three dimensional array A that represents the entire mesh, process 0 owns A(1:12,1:6,1:12) and process 1 owns A(1:12,7:12,1:12).

The automatic debugging algorithm starts by examining the definition points of the user-indicated incorrect value of variable xci in routine verify. (See also the relevant part of the call graph shown in Figure 4). The variable is not defined there, but rather is an argument passed in, so the search moves to the calling routine. This in turn leads to the routine pintgr in which the variable (now named frc) is defined. The definition statement is:

```
frc=0.25d+00*(frc1+frc2-frc3)
```

where frc1, frc2, and frc3 are not overwritten before the current execution state (i.e., at a breakpoint in verify). However, they are variables local to pintgr and since it has already exited, they cannot be evaluated. Therefore, an instrumentation point is set in routine pintgr at the point where frc is set and a re-execution is performed.

On reaching the definition of frc, the value of frc3 in the parallel execution proves to be incorrect when *p2d2* compares it to the serial value. The search therefore continues for its defining statements. One is found in the statement:

```
frc3=deta*dzeta*frc3
```

where the values of deta and dzeta prove correct. The incorrect value must therefore be frc3. However, the old value was overwritten by this very assignment and thus cannot be tested. Rather than issuing another instrumentation for this variable (especially since this is the statement immediately prior to the previous instrumentation point), the definitions of the frc3 used here are inspected. This brings us to a communication call that performs a global summation of this variable, so the definitions of frc3 used in this summation are then inspected. The definition encountered is:

```
frc3=frc3+(phi1(j,k)   +phi1(j+1,k)   +
           phi1(j,k+1)+phi1(j+1,k+1)+
           phi2(j,k)   +phi2(j+1,k)   +
           phi2(j,k+1)+phi2(j+1,k+1))
```

and the arrays phi1 and phi2 are inspected to determine if they are correct. When more than one location in an array is wrong, we find the "most incorrect" element in the array on any process. In this case, both phi1 and phi2 prove incorrect with the "worst" values identified being:

phi1(6,7) (on process 1)
phi2(6,6) (on process 0)

The search then continues for the definition of these incorrect values, leading to the statement:

```
phi2(j,k)=c2*(u(5,ifin,j,k)-
              0.50d+00*
                (u(2,ifin,j,k)**2+
                 u(3,ifin,j,k)**2+
                 u(4,ifin,j,k)**2)/
                u(1,ifin,j,k))
```

still in routine pintgr. Evaluation establishes that c2 is correct and that ifin = 11, so the search then focuses on u(1:5,11,6,6) on process 0.

In subroutine pintgr the array u is in a common block. The definition of u is found to be in routine ssor at the statement

```
u(m,i,j,k)=u(m,i,j,k)+tmp*rsd(m,i,j,k)
```

where all used variables are potentially overwritten before reaching the state available to *p2d2*, so instrumentation points are set and a re-execution performed.

On reaching an instrumentation point, we search for the statements that define rsd(1,11,6,6) on process 0. It is defined in routine buts at the statement:

```
v(1,i,j,k)=v(1,i,j,k)-tv(1,i,j)
```

The values of tv and v cannot be tested in the current state, so we search for their definition points. Array tv is defined in the statement:

```
tv(1,i,j)=tv(1,i,j)/tmat(1,1)
```

where, although tmat cannot be checked, it is defined as

```
tmat(m,1)=d(m,1,i,j)
```

within the same `i` and `j` loops. Furthermore, the value of `d(1,1,11,6)` on process 0 proves correct so `tmat` does not need to be instrumented. Obviously, the used value of `tv` has been overwritten, so an instrumentation point for `tv` is set and a re-execution performed. The definition of `tv` that is then located is the assignment

`tv(m,i,j)=`f`(v,tv)`

in Figure 3, and this is the statement the prototype reports to the user as the location of the first difference.

The current implementation of the algorithm reports the use of `tv` in the statement as the problem. The actual problem, an incorrect value for `v(1,11,7,6)` on process 0 due to the missing pipeline communication, will be found when the prototype has the iteration mapping information described in Section 2.1. This extension to the prototype is discussed in Section 5.1.

For simplicity, the above example operation of the algorithm omits many other successful variable comparisons and instrumentation points that were never reached. These were essential to ensure that the problem would not be missed, but proved unnecessary as the algorithm progressed.

5. Extending the Prototype

While the prototype described in Section 3 establishes the proof of concept of the use of backtracking and re-execution in the debugging of parallelized programs, it doesn't yet meet our vision for a comprehensive automatic debugger for parallelized programs. In this section we describe how we plan to extend the functionality of the prototype to handle a wider variety of problems.

5.1. Iteration Mapping

Our current prototype assumes that only a trivial iteration mapping is needed to compare the serial and parallel computations. Any statement that is executed more than once is assumed to be instrumentable in both the serial and parallel programs without having to worry about aligning iterations.

In terms of loop iteration matching, a first-order approximation of this mapping will describe loop transformations and unrolling performed at the source code level. More precise approximations might also include those loop optimizations performed by the compilers used to generate the serial and parallel executables. In addition, the iteration mapping will need to describe program statements that are executed multiple times because of repeated calls to the same function.

Once an iteration mapping is produced there are still further issues to consider. In our prototype, *CAPTools* will communicate to *p2d2* the necessary iteration conditions for each instrumentation point. These conditions can then be checked each time the instrumentation is triggered. For a nested loop this might be unacceptably slow, though, and to solve this we could instrument locations progressively closer to the actual location. For example, instrumentation could be placed in the outer-loop of the loop nest to check when its iteration condition is met, and then each successive inner-loop would be instrumented in the same manner until our desired location and iteration is reached.

Additional improvement can be obtained by looking into different types of instrumentation. Conditional breakpoints offer an easy way to accommodate iteration condition checking, but typically execute much slower than the program itself. Instrumentation that runs at full program execution speed (such as offered by Dyninst [4]) is desirable, assuming the required types of condition checking can be implemented.

5.2. Different Program Sources

In the interest of minimizing implementation time, we limited our prototype to comparing one-process and multiprocess executions of the same program. That is, given an MPI code produced by *CAPTools*, we can compare a single process execution with a multiprocess one. The user may, however, be more interested in comparing an execution of the original serial program (with no MPI calls in it) to a multiprocess execution of the MPI version.

There are two issues that arise if we are to relax this limitation of our prototype. First, the parallelization tool will need to provide the source-to-source mapping discussed in Section 2.1. This mapping may be difficult to produce, because, for example, the parallelization tool may have introduced new functions as a result of either outlining or cloning.

A second issue that comes up concerns the order of execution of statements in the two versions of the code. While this issue has been studied by Watson and Abramson in Guard [16], the addition of backtracking introduces further complexity to the situation. Consider the following two situations addressed by Guard.

- If we have multiple instrumentation points in the program, there may be no guarantee that they will be encountered in the same order in the serial and parallel executions. The problem then is that we must continue past one of the instrumentation points in order for the execution to make progress. By continuing we may destroy values that are needed for comparison at the point when the second execution reaches the corresponding point. Thus, the implementation must take care to checkpoint information required for comparison.

- *Temporal displacement* concerns the possibility that a collection of values in one program may not exist all at the same time during execution of the other program. This situation can arise, for example, as a result of scalar expansion or loop fusion. To address this situation Watson and Abramson suggest an array constructing technique that collects and checkpoints the values needed in a comparison.

In Guard, the values being compared are explicit. Thus, if checkpointed values or constructed arrays are needed, it is clear which values should be saved. If we add backtracking to relative debugging, the values that we may want to examine are not known *a priori*. Instead as our comparison algorithm backtracks we determine the values of interest. The question we need to address, therefore, is which values should we checkpoint? One extreme would be to checkpoint an entire program state. Alternatively, we might want to anticipate the backtracking that will be done and only checkpoint values that might be investigated.

5.3. Manually Parallelized Programs

In order to extend the prototype to manually parallelized programs, we need to acquire the types of information described in Section 2 that are currently provided by the parallelization tool: data dependence and computation mapping information. There are two possibilities for each class of information. Either we can try to acquire the information automatically or we will need to ask the user for it. In our estimation, automatic methods are clearly preferable, because the user will likely find the process of providing information to be tedious and he may make mistakes.

Fortunately, it seems straightforward to collect the dependence analysis information. By running a variant of the *CAPTools* analysis phase on both the serial and parallel versions of the code, we should be able to answer the dataflow questions that come up during the relative debugging.

It is not as straightforward to collect the computation mapping. For example, acquiring the source mapping information automatically may be an interesting research problem, depending on the similarity of the serial and parallel programs. If, for example, functions in the serial code are present in the parallel code, then there is a natural starting point of correspondence. It may be possible to use incomplete mapping information that is automatically derived to bracket an error, and then rely on the user to provide more complete mapping information in order to zero in on the bug.

Determining the other mapping information automatically seems more problematic. There has been previous work done on straightforward ways for a user to describe the data distribution [7, 9, 15, 17]. Similar approaches may work for other aspects of data value mapping. When the data value mapping is combined with dependence analysis information we may be able to construct execution and iteration mapping automatically.

5.4. Shared-Memory Parallelism

One of the restrictions we placed on the prototype implementation concerned the parallel programming paradigms supported. In particular we limited it to handling only MPI programs[2]. In the future we would like to relax this restriction by providing support for shared-memory parallel programs, such as OpenMP programs.

One obstacle to relaxing this restriction is in the implementation of *p2d2*. The debugger uses a client-server architecture [6], and the current implementation of the server is layered on top of *gdb*, the debugger from the Free Software Foundation. Unfortunately, *gdb* does not support a full set of thread control operations. For example, there is no *gdb* command to single-step one thread (and leave others where they are).

In order to compare the computations of an OpenMP program and its serial counterpart, we must be able to control individual threads in the OpenMP program. To do this using the *gdb* command set, we need to be able to hold some threads when we continue others. One way to do this is to modify the program counter of each thread to be held so that it effectively busy waits while the non-held threads execute their normal instruction stream. We have tested this technique in a prototype debugger server and it seems promising. Some work remains to make it robust enough to use in the general case.

Besides the underlying debugger work needed, our prototype would need to find sources for the dependence analysis and computation mapping information currently provided by *CAPTools*. Fortunately there is a tool, *CAPO* [8], which is based on the *CAPTools* code base and can generate OpenMP programs. It should thus be straightforward to get the information we need.

There are also paradigm issues to consider in automatic relative debugging of shared-memory programs. In particular, user errors, such as incorrectly indicating that it is safe to run a loop in parallel, could lead to race conditions in the program that cause nondeterministic behavior. Since it will be important for the relative debugging tool to produce consistent answers, detecting and handling nondeterministic execution in the target code will be critical.

[2]While the prototype currently handles only MPI programs, extending it to other message-passing libraries, such as PVM [13], is straightforward. There are two issues to address: tool generation of the code and debugging codes of that type. *CAPTools* already produces codes containing calls to CAPLib, a generic message-passing library. With the exception of process startup, *p2d2* is independent of the message-passing library used. Thus, accommodating a new distributed communication library reduces to implementing CAPLib in terms of the library and having the library's process creation mechanism notify *p2d2* when there are new processes to debug.

5.5. Identifying and Correcting Bugs

Detecting the first difference is not necessarily the end of the story for our relative debugging mechanism. In the parallelized version of a serial program we can expect to encounter certain classes of bugs that are symptomatic of mistakes made in manual parallelizations and incorrect user inputs in the parallelization process. A mechanism that not only isolates the location of differences, but also identifies the type of bug that caused the difference along with a potential corrective action, could be of tremendous value to the user.

We would like to expand the scope of our difference detection mechanism to include an analysis of the difference. The following are some common bug types in distributed-memory message-passing programs we believe can be identified using this analysis:

- a missing communication;
- a communication that does not convey all required data;
- a communication that overwrites correct data with incorrect data; and
- missing computation (i.e., where a computation is performed in serial but where no matching computation is performed in any parallel process, perhaps due to errors in distributed loop limits, etc.)

For example, suppose that in one process of the parallel execution there is a variable that is first assigned a correct value and then used. In another process of the parallel execution, though, the same variable is used without being assigned, triggering the detection of a difference between the parallel and serial computations. Analysis of the definition and use of the variable among the parallel processes might indicate that a communication is missing between those processes.

Additionally, difference analysis in a prototype that has been extended as described in Section 5.4 might help isolate the following bug types in shared-memory OpenMP programs :

- invalid parallel execution of a serial loop, producing nondeterministic behavior due to data races;
- mis-declaring a shared variable to be private, leading to data being lost when the parallel region is exited;
- mis-declaring a private variable to be shared, leading to overwriting of values by other threads; and
- missing synchronization, leading to the use of stale values.

The definition, use, and sharing of values could be analyzed to identify these bugs, similarly to the way definition, use, and communication of values might be analyzed in distributed-memory programs.

6. Related Work

Backtracking has existed in debuggers for more than 30 years. Agrawal's thesis [2] surveys several approaches that roll back execution from an erroneous state, looking for the original bug in the program. Typically, these mechanisms use a combination of dependence information from a static analysis pass and trace information collected during an execution.

Agrawal also reviews filtering techniques that use static analysis information. Program slicing and program dicing attempt to obtain the subset of a program that *may* have had an effect on a given variable. A slice reduces the search space for bugs, but does not provide, on its own, a comprehensive way to isolate them. Program dicing improves upon slicing by narrowing down the search space of a slice using information about correct variables.

There has been considerable work done in the last few years on Relative Debugging [1]. For example, the Guard debugger [16] allows the user to indicate comparisons that should be made at runtime between two executions. During execution, it performs the comparisons, even taking care of potential execution order changes between the two programs, and stops when a difference is detected. As part of this work, Watson and Abramson [15, 17] detail an algebra for describing data distributions.

Relative debugging has been applied previously to the specific problem of finding differences between serial and tool-generated parallel programs [7, 9]. In those efforts, the user indicates what variable is wrong and where it is wrong in the program. The debugger then queries the parallelization tool to find out which routines modify the variable and under what name it is modified. With that data, the debugger then inserts comparison instrumentation at entry and exit of those routines. When the serial and parallel programs are then run side-by-side, the debugger is able to narrow down the difference to a single subprogram.

7. Conclusions

The automation of relative debugging for parallelized programs can provide a significant reduction in the effort required of users to find where a serial and parallel computation diverge. The steps typically performed by the user can be more quickly and comprehensively performed using mapping and data dependence information, potentially from a parallelization tool, to inform the actions of a parallel debugger.

In this work we have described the implementation of a practical mechanism that uses backtracking and re-execution to automate a relative debugging session. Instrumentation points are determined using existing static analysis information along with dynamically retrieved program state, without requiring that trace information be collected during executions. This allows us to avoid both excessive re-execution and excessive instrumentation by exploiting as much current information as possible at every stage.

The practicality of using automated relative debugging continues to be of interest to us, and our prototype implementation helps to shed light on where algorithmic improvements and better utilization of currently available information can lead to further reductions in required user knowledge and time. This issue is of particular concern to the high performance community due to the great need for porting of large- scale, long-running programs to parallel forms.

Acknowledgments

We wish to thank Cos Ierotheou and Emyr Evans for their helpful inputs on both technical and presentation issues. We also want to express appreciation to Henry Jin and Parkson Wong for their reviewing the paper and making suggestions for improvements.

This work was partially funded through NASA contract DTTS59-99-D-00437/A61812D.

References

[1] Abramson D., Foster, I., Michalakes, J., and Sosic, R. "Relative Debugging and its Application to the Development of Large Numerical Models." *Proceedings of IEEE Supercomputing 1995*, San Diego, December 1995. `http://citeseer.nj.nec.com/abramson95relative.html`.

[2] 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`.

[3] Computer Aided Parallelization Tools (*CAPTools*). `http://captools.gre.ac.uk/`.

[4] The Dyninst API. `http://www.cs.umd.edu/projects/dyninstAPI/`.

[5] 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.

[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] Hood, R. and Jost, G. "Support for Debugging Automatically Parallelized Programs." *Proceedings of AADEBUG'2000*, Munich, Germany, 2000.

[8] 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`.

[9] Jost, G. and Hood, R. "Relative Debugging of Automatically Parallelized Programs." To appear in the *Journal of Automated Software Engineering*.

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

[11] 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.

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

[13] Sunderam, V. S. "PVM: a Framework for Parallel Distributed Computing." *Concurrency, Practice and Experience 2* (4), pp. 315–340, 1990. `http://citeseer.nj.nec.com/sunderam90pvm.html`.

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

[15] Watson, G. "The Design and Implementation of a Parallel Relative Debugger." Ph.D. thesis, Monash University, Melbourne, Australia, 2000.

[16] 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.

[17] Watson, G. and Abramson, D. "Programming Language Array Constructs For Parallel Relative Debugging." `http://citeseer.nj.nec.com/watson98programming.html`.