Portable OpenMP debugging with TotalView

James Cownie[1] and Shirley Moore[2] *

[1] Etnus LLC., Framingham MA, USA
jcownie@etnus.com,
WWW home page: http://www.etnus.com/
[2] Innovative Computing Laboratory, University of Tennessee
browne@cs.utk.edu,
WWW home page: http://icl.cs.utk.edu/

Abstract. This paper discusses some of the issues in debugging OpenMP code due to compiler transformations, and the support provided for debugging OpenMP programs in the TotalView multi-process, multi-threaded debugger.

1 Introduction

The objective of TotalView's OpenMP support is to provide as much information to the users as possible so that they can debug their code. TotalView does *not* attempt to pretend that an OpenMP code is actually sequential, since this is both hard, and counterproductive: bugs in OpenMP code may be a result of the threaded nature of the OpenMP execution model. Instead TotalView aims to allow the user to see all of the state of their program just as they would be able to do for a sequential code.

The following problems arise in debugging OpenMP codes:

- simple debugging of threaded code (especially if a user level thread scheduler is involved),
- working out how execution reached the current point,
- accessing private and shared variables, and
- accessing threadprivate variables.

All of the functionality described in this paper (with the exception of `threadprivate` variable support on some platforms) works with vendor OpenMP compilers (Compaq, IBM, SGI) on their platforms, as well as with Kuck and Associates' Guide compiler on many different target machines.

* This work was partially supported by ARL MSRC under prime contract #DAHC94-96-C-0010, by ASC MSRC under prime contract #DAHC94-96-C-0005, and by ERDC MSRC under prime contract #DAHC94-96-C-0002.

2 OpenMP compilation

To understand the issues involved in debugging OpenMP codes, it is necessary to have some knowledge of the way that OpenMP codes are transformed by the compiler. We will discuss this here in the context of OpenMP Fortran. Similar transformations apply to OpenMP C.

Consider a canonical OpenMP example code like Fig. 4. In order to exploit the parallelism in the `do` loop, the OpenMP compiler must generate code for the loop body that can run concurrently in many threads, while still accessing the shared variables `w` and `sum`.

Most current OpenMP compilers achieve this by converting the loop body into a separate subroutine that can then be called by the OpenMP runtime in many threads simultaneously. Access to the shared variables is achieved either by making this "outlined" procedure into an F90 contained procedure of the parent and accessing the shared variables via the lexical link, or by passing the addresses of each of the shared variables referenced by the outlined procedure as arguments. In languages without lexical scoping (such as C or FORTRAN 77) only the second option is possible.

The outlined procedure is called through an OpenMP runtime routine that is responsible for ensuring that an appropriate thread is available to execute the code, and that the outlined procedure runs on that thread's stack. It is not possible to write this runtime routine in standard Fortran.

Therefore the code is converted into something conceptually like Fig. 5. This code assumes a static schedule of the loop body. Different code could be generated to handle other scheduling strategies. However, whichever scheduling strategy is used, the general style of the transformation, and the debugging issues it introduces, remain the same.

3 Thread debugging

To be able to debug OpenMP code a debugger must be able to debug threaded code, since the OpenMP runtime model is of multiple threads executing the code simultaneously.

Even before the specific work to support OpenMP TotalView was already a thread-capable debugger. No enhancements were therefore required to support basic thread level debugging of OpenMP parallelized code that uses one of the threading models supported by TotalView.

In the presence of a user level thread library that switches many threads between fewer kernel scheduled threads, thread debugging requires knowledge of, or assistance from, the user level thread library.

Without this it is impossible for the debugger to present the user with a thread list that corresponds to the user model. Fortunately user thread libraries were already supported by TotalView before work started on OpenMP support.

Unfortunately the user cannot single-step into an OpenMP parallel region using TotalView. Although the step looks to the user as if it is a simple step to the next line in the code in fact the next line is executed in a different thread, and is only reached after much code in the runtime library has executed. Even if TotalView were to have knowledge of the runtime library routines that are responsible for requesting execution in the worker thread, the spawning thread may never execute the apparently next line in the code. However provided a breakpoint has been planted inside the parallel region it is easy to run to that point in whichever thread executes the outlined routine.

The extent to which TotalView can then allow the user to control multi-threaded execution within the parallel region depends on operating system support for asynchronous thread control. On platforms with such support, the user can single-step a selected thread or single-step all user-level threads as a group.

4 Stack Backtraces

Consider the transformed code from our simple example, and assume that the user has set a breakpoint in the parallel loop that has been reached in one of the "worker" threads. The TotalView display will look like Fig. 1.

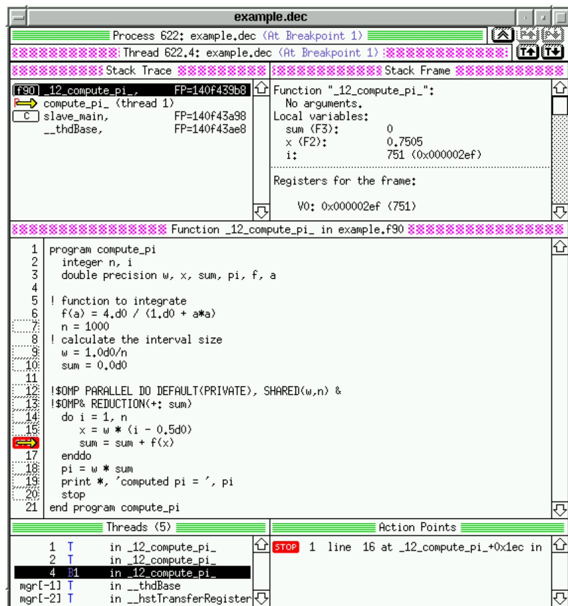

Fig. 1. Backtrace from a parallel loop

Here one can see that:

– The program has five threads, of which three are useful OpenMP worker threads and two are system threads (part of the thread library).
– All of the OpenMP threads are executing in the outlined routine, which has a name constructed by the compiler.
– The thread on which we have focused is a worker thread that was called from the OpenMP runtime system.
– TotalView has provided a "parent frame link" in the stack to show where this outlined routine was actually called from in the user's OpenMP code.

If the user selects this parent link, then the display is refocused onto the appropriate parent thread and stack frame (Fig. 2). It is thus easy to determine how a specific execution point was reached even when the breakpoint or failure occurs in an outlined routine.

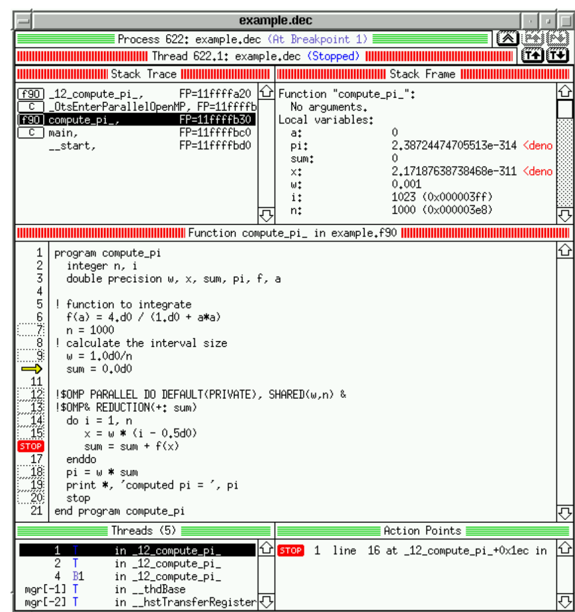

Fig. 2. Backtrace after following the parent link

This approach to the display of the backtrace from a worker thread is preferable to "faking up" a complete backtrace, because it both provides the user with the information required, and is less misleading. With the parent frame link, the user can see not only how execution reached this point, but also that the code is executing in a separate thread, and (if required) the runtime library routines that are involved.

TotalView does not demangle the names given by the compiler to the outlined routines since it is important that the user recognise that the code is executing in an outlined routine, rather than the parent function

itself, and there is no sensible user-level name which could be given to the outlined routine.

5 Data display

Displaying the values of OpenMP local, private variables in parallel constructs normally requires no special treatment, since these variables are stack-allocated, and the compiler outputs normal debug information for them. In Fig. 1 we can see that this is the case for the private variables `i` and `x`.

We can also see that the OpenMP compiler being used (Compaq's F90 compiler on Tru64) has chosen to implement access to shared variables via the static link (since there is no argument passing in `w`). However, this is not a problem for TotalView since it already understands the lexical scoping of Fortran 90 and how to access lexically scoped variables. If the user asks to display the variable `w` (by right clicking on it in the normal way within TotalView), TotalView will open a data window showing its value. If the compiler had chosen to pass the value of `w` as an argument TotalView would simply have displayed it as such. In either case the value is visible to the user of TotalView in a natural way.

When debugging an OpenMP code, it is often useful to see the value taken by every instance of a `private` variable. To make this easy, TotalView has the ability to display the value of a local variable in all threads that have a matching stack frame. (This is known as a "Thread Laminated" display). To support OpenMP codes, the knowledge of the call structure used to generate the parent link is also used when matching stack frames for the thread laminated display, so that stack frames match whether or not they have been invoked on the process' initial stack, or from inside a worker thread. For instance Fig. 3 shows the value of the loop control variable `i` in all four threads.

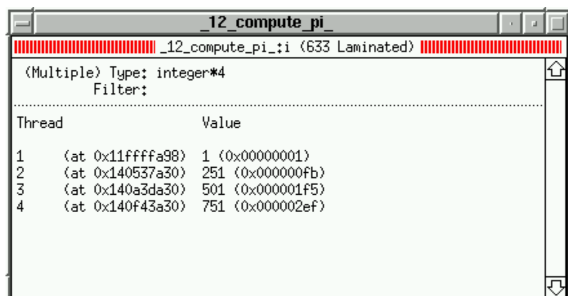

Fig. 3. Display of a local loop control variable in all threads

Access to OpenMP `threadprivate` variables such as `common` blocks or `static` variables is more complex. These variables exist for the duration of one OpenMP parallel region but are allocated only once in each OpenMP thread. Different compilers and run time systems use different implementation techniques for such variables.

5.1 Single virtual address multiple page mappings

On SGI Irix each thread has its own set of page tables. Therefore it is possible to map different physical pages to the same virtual address in each thread. A thread private variable can then be allocated by the linker to a virtual address in this thread-private region. Provided that the debugger is careful to access target store through the correct thread `/proc` file descriptor, the correct value will be displayed.

5.2 Multiple virtual addresses with linker support

On Compaq Tru64 Unix, the linker collects thread private variables into a single segment and generates their addresses as offsets relative to the base of this area. At run time the thread startup code allocates sufficient space for all of the thread private variables and stores the address of this area in the thread's runtime descriptor. The debugger can then address a `threadprvate` variable using the offset from the debug information and the base address of the thread's private store area obtained from the operating system.

5.3 Via pthread functions

On other operating systems on which TotalView runs there is, as yet, no specific linker support for thread private data. On these operating systems, or where portability is at a premium, the OpenMP runtime system has to make use of the `pthread_{get,set}specific` routines to store thread private data.

To support this TotalView has been extended to call vendor supplied routines that are dynamically linked into TotalView itself at runtime (through use of the `dlopen` library call). This allows details of the vendor implementation of `threadprivate` variables to remain in vendor code, and to change between different compiler releases without affecting TotalView. These routines can in turn make callbacks into TotalView to read values from target process' store, or (assuming such a callback exists in the vendor's thread debug library) to look up the value associated with a pthread key.

This dynamic library is also used to handle the name transformations generated by KAI Guide processed code, so that the user can see the original

names for variables, rather than the "mangled" names output by Guide into the intermediate source files.

6 Future directions

While we believe that TotalView in its current state is extremely useful for debugging OpenMP codes, we can see a number of possible extensions to aid in debugging more complex OpenMP codes.

Locks It would obviously be useful to display as much information as possible about an OpenMP lock, since this should make it easier for users to debug code that uses locks erroneously.

Barrier state It would be useful to be able to display the internal state of the data structures that represent an OpenMP barrier. This might allow the user more easily to determine which threads are failing to reach barriers, and thus why the program is deadlocking.

Variable information With some compilers it may be possible to add additional annotation to variable display to show whether the variables is `private`, `threadprivate` or `shared`. While this information is implicitly available form the way the debugger behaves and is explicit in the user's code, explicit display in TotalView might make it easier for the user to detect cases where they have shared a variable that they expected to be `private` because they have a omitted a `default(private)` clause on a parallel directive.

Asynchronous thread control Work is already scheduled to provide users with more intimate control of which threads execute when stepping code through a parallel region. This will allow users explicitly to control which threads are subject to source level `step` or `next` commands, or which threads are allowed to run as the result of a `go` command. Providing this level of control is hard (especially when interacting with a user level thread scheduler), but will be an important improvement.

As with other target dependent code these functions would be implemented by making calls to a compiler dependent library dynamically linked into TotalView.

7 Conclusions

We have discussed some of the major problems in debugging OpenMP code, and presented the solutions adopted in the TotalView debugger.

The addition of OpenMP debugging capability in TotalView is entirely orthogonal to its existing capabilities as a distributed debugger, making it possible to debug hybrid MPI codes with OpenMP node programs.

We believe that these solutions come close to making the debugging of OpenMP codes as tractable as the debugging of sequential codes, and make TotalView the debugger of choice for OpenMP applications on many platforms.

References

1. Etnus, Inc. TotalView Users' Manual. Available from http://www.etnus.com/.
2. Kuck and Associates Guide Users' Manual Available from http://www.kai.com/.
3. OpenMP Architecture Review Board OpenMP C and Fortran specifications Available from http://www.openmp.org/.


```
program compute_pi
  integer n, i
  double precision w, x, sum, pi, f, a

! Function to integrate
  f(a) = 4.d0 / (1.d0 + a*a)
  n = 1000
! Calculate the interval size
  w = 1.0d0/n
  sum = 0.0d0

!$OMP PARALLEL DO &
!$OMP& DEFAULT(PRIVATE), &
!$OMP& SHARED(w,n) &
!$OMP& REDUCTION(+: sum)
  do i = 1, n
     x = w * (i - 0.5d0)
     sum = sum + f(x)
  enddo
  pi = w * sum
  print *, 'computed pi = ', pi
  stop
end program compute_pi
```

Fig. 4. Sample OpenMP code

```
program compute_pi
  integer n, i, ichunksize, ibase, num_threads, ithread
  double precision w, x, sum, pi, f, a

  external compute_pi_outline, num_threads

! The function to integrate
  f(a) = 4.d0 / (1.d0 + a*a)

! The number of samples
  n = 1000

! The interval size
  w = 1.0d0/n
  sum = 0.0d0

! Work out the number of elements for each thread
  ichunksize = n/omp_num_threads()
  ibase = 1
! Spawn the threads
  do ithread = 1, omp_num_threads()-1
      call execute_in_thread (ithread, compute_pi_outline, ibase, &
&          ibase+ichunksize-1, w, sum)
      ibase = ibase + chunksize
  end do

  call compute_pi_outline (ibase, n, w, sum)

  call wait_for_threads()

  pi = w * sum
  print *, 'computed pi = ', pi
end program compute_pi

subroutine compute_pi_outline (ibase, iend, w, sum)
  double precision local_sum, f, w, sum, x
! The function to integrate
  f(a) = 4.d0 / (1.d0 + a*a)

  local_sum = 0.d0

! Evaluate this thread's contribution
  do i = ibase, iend
     x = w * (i - 0.5d0)
     local_sum = local_sum + f(x)
  end do

! Accumulate the final result
  call lock_variable (sum)
  sum = sum + local_sum
  call unlock_variable (sum)
end compute_pi_outline
```

Fig. 5. Transformed sample OpenMP code