
Compiling and Debugging Parallel Fortran

This chapter gives instructions on how to compile and debug a parallel Fortran program and contains the following sections:

- “Compiling and Running” explains how to compile and run a parallel Fortran program.
- “Profiling a Parallel Fortran Program” describes how to use the system profiler, *prof*, to examine execution profiles.
- “Debugging Parallel Fortran” presents some standard techniques for debugging a parallel Fortran program.

This chapter assumes you have read Chapter 5, “Fortran Enhancements for Multiprocessors,” and have reviewed the techniques and vocabulary for parallel processing in the IRIX environment.

Compiling and Running

After you have written a program for parallel processing, you should debug your program in a single-processor environment by calling the Fortran compiler with the *f77* command. You can also debug your program using the WorkShop Pro MPF debugger, which is sold as a separate product. After your program has executed successfully on a single processor, you can compile it for multiprocessing. Check the *f77(1)* reference page for multiprocessing options.

To turn on multiprocessing, add **-mp** to the *f77* command line. This option causes the Fortran compiler to generate multiprocessing code for the particular files being compiled. When linking, you can specify both object files produced with the **-mp** option and object files produced without it. If any or all of the files are compiled with **-mp**, the executable must be linked with **-mp** so that the correct libraries are used.

Using the `-static` Option

A few words of caution about the `-static` compiler option: The multiprocessing implementation demands some use of the stack to allow multiple threads of execution to execute the same code simultaneously. Therefore, the parallel **DO** loops themselves are compiled with the `-automatic` option, even if the routine enclosing them is compiled with `-static`.

This means that **SHARE** variables in a parallel loop behave correctly according to the `-static` semantics but that **LOCAL** variables in a parallel loop do not (see “ Debugging Parallel Fortran” on page 152 for a description of **SHARE** and **LOCAL** variables).

Finally, if the parallel loop calls an external routine, that external routine cannot be compiled with `-static`. You can mix static and multiprocessed object files in the same executable; the restriction is that a static routine cannot be called from within a parallel loop.

Examples of Compiling

This section steps you through a few examples of compiling code using `-mp`. The following command line

```
% f77 -mp foo.f
```

compiles and links the Fortran program `foo.f` into a multiprocessor executable.

In this example

```
% f77 -c -mp -O2 snark.f
```

the Fortran routines in the file `snark.f` are compiled with multiprocess code generation enabled. The optimizer is also used. A standard `snark.o` binary is produced, which must be linked:

```
% f77 -mp -o boojum snark.o bellman.o
```

Here, the `-mp` option signals the linker to use the Fortran multiprocessing library. The file `bellman.o` did not have to be compiled with the `-mp` option (although it could be).

After linking, the resulting executable can be run like any standard executable. Creating multiple execution threads, running and synchronizing them, and task terminating are all handled automatically.

When an executable has been linked with `-mp`, the Fortran initialization routines determine how many parallel threads of execution to create. This determination occurs each time the task starts; the number of threads is not compiled into the code. The default is to use whichever is less: 4 or the number of processors that are on the machine (the value returned by the system call `sysmp(MP_NAPROCS)`; see the `sysmp(2)` reference page). You can override the default by setting the shell environment variable `MP_SET_NUMTHREADS`. If it is set, Fortran tasks use the specified number of execution threads regardless of the number of processors physically present on the machine. `MP_SET_NUMTHREADS` can be from 1 to 64.

Profiling a Parallel Fortran Program

After converting a program, you need to examine execution profiles to judge the effectiveness of the transformation. Good execution profiles of the program are crucial to help you focus on the loops consuming the most time.

IRIX provides profiling tools that can be used on Fortran parallel programs. Both `pixie(1)` and `pc-sample` profiling can be used (`pc-sampling` can help show the system overhead). On jobs that use multiple threads, both these methods will create multiple profile data files, one for each thread. You can use the standard profile analyzer `prof(1)` to examine this output. Also, `timex(1)` indicates whether or not the parallelized versions performed better overall than the serial version.

The profile of a Fortran parallel job is different from a standard profile. As mentioned in “Analyzing Data Dependencies for Multiprocessing” on page 76, to produce a parallel program, the compiler pulls the parallel `DO` loops out into separate subroutines, one routine for each loop. Each of these loops is shown as a separate procedure in the profile. Comparing the amount of time spent in each loop by the various threads shows how well the workload is balanced.

In addition to the loops, the profile shows the special routines that actually do the multiprocessing. The `__mp_parallel_do` routine is the synchronizer and controller. Slave threads wait for work in the routine `__mp_slave_wait_for_work`. The less time they wait, the more time they work. This gives a rough estimate of how parallel a program is.

Debugging Parallel Fortran

This section presents some standard techniques to assist in debugging a parallel program.

General Debugging Hints

- Debugging a multiprocessed program is much more difficult than debugging a single-processor program. Therefore you should do as much debugging as possible on the single-processor version.
- Try to isolate the problem as much as possible. Ideally, try to reduce the problem to a single **C\$DOACROSS** loop.
- Before debugging a multiprocessed program, change the order of the iterations on the parallel **DO** loop on a single-processor version. If the loop can be multiprocessed, then the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order frequently causes the single-processor version to fail, and standard single-process debugging techniques can be used to find the problem.

Example: Erroneous **C\$DOACROSS**

In this example, the bug is that the two references to **a** have the indexes in reverse order. If the indexes were in the same order (if both were **a(i,j)** or both were **a(j,i)**), the loop could be multiprocessed. As written, there is a data dependency, so the **C\$DOACROSS** is a mistake.

```
c$doacross local(i,j)
  do i = 1, n
    do j = 1, n
      a(i,j) = a(j,i) + x*b(i)
    end do
  end do
```

Because a (correct) multiprocessed loop can execute its iterations in any order, you could rewrite this as:

```
c$doacross local(i,j)
  do i = n, 1, -1
    do j = 1, n
      a(i,j) = a(j,i) + x*b(i)
    end do
  end do
```

This loop no longer gives the same answer as the original even when compiled without the `-mp` option. This reduces the problem to a normal debugging problem. When a multiprocessed loop is giving the wrong answer, make the following checks:

- Check the **LOCAL** variables when the code runs correctly as a single process but fails when multiprocessed. Carefully check any scalar variables that appear in the left-hand side of an assignment statement in the loop to be sure they are all declared **LOCAL**. Be sure to include the index of any loop nested inside the parallel loop.

A related problem occurs when you need the final value of a variable but the variable is declared **LOCAL** rather than **LASTLOCAL**. If the use of the final value happens several hundred lines farther down, or if the variable is in a **COMMON** block and the final value is used in a completely separate routine, a variable can look as if it is **LOCAL** when in fact it should be **LASTLOCAL**. To combat this problem, simply declare all the **LOCAL** variables **LASTLOCAL** when debugging a loop.

- Check for **EQUIVALENCE** problems. Two variables of different names may in fact refer to the same storage location if they are associated through an **EQUIVALENCE**.
- Check for the use of uninitialized variables. Some programs assume uninitialized variables have the value 0. This works with the `-static` option, but without it, uninitialized values assume the value left on the stack. When compiling with `-mp`, the program executes differently and the stack contents are different. You should suspect this type of problem when a program compiled with `-mp` and run on a single processor gives a different result when it is compiled without `-mp`. One way to track down a problem of this type is to compile suspected routines with `-static`. If an uninitialized variable is the problem, it should be fixed by initializing the variable rather than by continuing to compile `-static`.
- Try compiling with the `-C` option for range checking on array references. If arrays are indexed out of bounds, a memory location may be referenced in unexpected ways. This is particularly true of adjacent arrays in a **COMMON** block.

- If the analysis of the loop was incorrect, one or more arrays that are **SHARE** may have data dependencies. This sort of error is seen only when running multiprocessed code. When stepping through the code in the debugger, the program executes correctly. In fact, this sort of error often is seen only intermittently, with the program working correctly most of the time.
- The most likely candidates for this error are arrays with complicated subscripts. If the array subscripts are simply the index variables of a **DO** loop, the analysis is probably correct. If the subscripts are more involved, they are a good choice to examine first.
- If you suspect this type of error, as a final resort print out all the values of all the subscripts on each iteration through the loop. Then use **uniq(1)** to look for duplicates. If duplicates are found, then there is a data dependency.