Chapter 20

Debuggers

A debugger is a very useful tool for developing codes, and in finding and fixing bugs introduced in the code development. Most compiler providers include a debugger as part of their software bundle. Since we are using the Portland Group compiler for the class, the discussion here will center on its debugger primarily, even though a lot of the information applies equally to other compiler/debugger systems. Check the manual for the specific compiler/debugger in case of problems.

20.1 Preparing the code for debugging

A debuggable executable must have extra information embedded in it for debugging purposes. The debugger then makes use of this information to report the values of variables, and allow the programmer to follow the execution of the program line by line. Compiler options must be used to instruct the compiler to embed this information in the object files. Here is a useful subset of these options pertinent to the PG compiler:

1. -g Generate information for the debugger, this is necessary on most computers to enable the printing of useful debugging information. Avoid using any optimization in the code development phase (so avoid the -O options). Occasionally you want to debug with optimization on, the -gopt would then be necessary.

2. -C or -Mbounds: generate code to check array bounds

3. -Ktrap=list-of-options: helps trap floating point problems. The list is a comma separates list of strings that controls which floating point operations to catch. These include:

   (a) -Ktrap=divz: trap divide by zero.
   (b) -Ktrap=ovef: trap floating point overflows.
(c) -Ktrap=unf: trap underflow (number too small to be representable).
(d) -Ktrap=inv: trap invalid operands (e.g. square root of negative numbers).

A commonly useful subset is to set -Ktrap=divz,inv,ovf

4. -Mchkptr: Check if code mistakenly references NULL pointers.

5. -Mchkstk: Check the stack for available space upon entry to and before the start of a parallel region. Useful when many private variables are declared.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>PGI</th>
<th>Intel</th>
<th>gcc</th>
<th>IBM</th>
<th>Pathscale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Debugging</td>
<td>-g</td>
<td>-g</td>
<td>-g</td>
<td>-g</td>
<td>-g</td>
</tr>
<tr>
<td>Bounds Checking</td>
<td>-C</td>
<td>-CB</td>
<td>-C</td>
<td>-C</td>
<td></td>
</tr>
<tr>
<td>Uninitialized variables</td>
<td>NA</td>
<td>-ftrapuv</td>
<td>-C</td>
<td>-qinitauto</td>
<td></td>
</tr>
<tr>
<td>Floating point trap</td>
<td>-Ktrap</td>
<td>-fpe0</td>
<td>-fpe-trap</td>
<td>-qflttrp</td>
<td></td>
</tr>
<tr>
<td>Floating point stack</td>
<td>-Mchkstk</td>
<td>-fpstkchk</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 20.2 Running the debugger

The command `pgdbg executable` will run the code `executable` under the control of the debugger. The default is to start the debugger under a Graphical User Interface (GUI). If you have a slow connection, the GUI can slow down your work because of the graphics overhead. Most debugger include a command, text line interface. For PG it is invoked with the `-text` option”. The command will start the debugger which in turn will “load” the executable and all its pertinent information. The debugger then passes control to the user and waits for further instructions. Most debuggers come with an on-line help facility to allow users to learn the command list interactively, the command to initiate is usually called `help`. Here we list briefly a subset of these commands and that have proved to be most useful for the author.

The debugger GUI is fairly intuitive; it will open up at least one window to display source lines, and another one for I/O operations. The GUI and text versions of the debugger can be controlled via command lines. Here we will cover some the most useful one, and we refer the user to the manual for further information.

1. **run** Will cause the code to start executing.
2. **list 10,40** will list the lines 10,40, **list** without argument will list lines 10 to 20 from the current statement.
3. **stop at xyz** will put a breakpoint at line xyz. Execution will stop at the line so the user can examine variables.
20.2. RUNNING THE DEBUGGER

4. **print sn** will print the value of the variable sn.

5. **assign sn=expression** assigns the expression `expression` to the variable sn.

6. **whatis sn** will report the data type of variable sn.

7. **where** will report the current statement.

8. **step** will execute the next source line, including stepping into a function or subroutine.

9. **stepi** will execute a single instruction (as opposed to the entire source line).
   *step `¡count`i* will execute `¡count`i instructions.

10. **display** list the expressions being printed at breakpoints. `display <exp1>,<exp2>` prints `<exp1>` and `<exp2>` at every breakpoint.

11. **next** will cause the debugge to skip over a function or subroutine, i.e. executing it in its entirety.

12. **continue** will cause the execution to resume from the point it stopped to the next breakpoint.