Answers

Exercise 1: Checking variable values

(glibc gives a stacktrace at the moment of the error, but only with memory references, not source lines. The top few stack frames are routines in the libc library, which are not very useful. The first stack frame pointer to our application can be used to locate the error using addr2line. This gives line 51, which frees the del-array. Although the double free happens on line 50, according to gdb from the core-dump, this already gives a good impression)

In the debugging example we had completed examining the values of *array2* computed by our function squareArray() but had found no errors in this function. In the solution illustrated below, we complete our debugging of variablePrinting.c simply by examining variable values within the remainder of this program.

If we look back at our lesson example, we can see from the printout of our code that, following the call to squareArray(), our program computes the difference between the element values in the two arrays, *array2* and *array1*,

del[indx] = array2[indx] - array1[indx] , for indx = 0, 1, . . . , (nelem – 1)

beginning at the for() loop on line 41.

If *array1* has elements *{ a1k }* and *array2* has the elements *{ a2k} = { (a1k)² }* then *{ delk } = { 0 }* only if *{ (a1k)² } = { a1k }*, which tells us that *{ a1k } = { 1 }*, for *k = 0, 1, . . . , (nelem – 1)*. But we can see from our output that not a single one of the elements of *array1* has the value 1. Therefore, we must have a coding error in our program.

To debug this next section of code, we step through this for() loop and examine the values of each variable on both the left- and right-hand sides of our expression for *del[indx]* to see if these values look correct for each value of *indx* in the this loop. We print out the values of *indx*, *array2*, *array1*, and *del*, using the GDB 'display' command.

```
(gdb) next
75      }
(gdb) n
main () at variablePrinting.c:41
41         for (indx = 0; indx < nelem; indx++)
(gdb) step
43            del[indx] = array2[indx] - array1[indx];
(gdb) disp indx
3: indx = 0
(gdb) disp array2[indx]
4: array2[indx] = 4
(gdb) disp array1[indx]
5: array1[indx] = 4
(gdb) disp del[indx]
6: del[indx] = 0
(gdb) s
41         for (indx = 0; indx < nelem; indx++)
6: del[indx] = 0
5: array1[indx] = 4
4: array2[indx] = 4
3: indx = 0
(gdb) s
43            del[indx] = array2[indx] - array1[indx];
6: del[indx] = 0
5: array1[indx] = 9
```

```
4: array2[indx] = 9
3: indx = 1
```

Clearly, the values for *array1* do not look correct at all. When we debugged our function in the lesson example, we found that the values of *array2* passed to the function squareArray() were correct and that the values of *array2* returned from this function were also correct, viz., { a2k } = { (a1k)2 }. That is, we found that the value of each element in *array2* was to equal the square of the values of the corresponding element in *array1*. But as we see from the above printout of our variables, the elements of *array1* and *array2* are equal, { a2k } = { a1k }. Somehow the values of the elements in *array1* have been squared in addition to the elements in array2.

So how could this have happened? The array, *array1*, was never passed to the function squareArray(); only *array2* was passed in line 38 of our code. If we think about it a bit, this sounds very much like a pointer error. If for some reason *array2* and *array1* are pointing to the same location in memory and the value of one of the elements of *array2* is modified, then the value of the corresponding element in *array1* will equal the modified value in *array2*. To confirm our suspicion, we compare the memory address of both *array1* and *array2*. If we look back at the example in this lesson, when we first stepped into our function squareArray( ) from line 38, GDB gave us the address of when it was passed to this function,

```
(gdb) s
squareArray (nelem_in_array=12, array=0x80498d8) at variablePrinting.c:70
70          for (indx = 0; indx < nelem_in_array; indx++)
(gdb) s
72              array[indx] *= array[indx];
```

If we print out the address pointed to by *array1* and compare it with the address to which the pointer *array2* was pointing when it was passed to squareArray(), we find that the two addresses are identical.

```
(gdb) disp array1
1: array1 = (int *) 0x80498d8
```

If we want, we can confirm this by printing out the address to which *array2* is pointing in the expression for *del[indx]* on line 43 of our code,

```
(gdb) disp array2
2: array2 = (int *) 0x80498d8
(gdb)undisp 1
(gdb)undisp 2
```

As we can see, this is the same address as that passed to squareArray() on line 38.

The question is: How can these two arrays have the same memory address when the memory for these arrays were allocated separately on lines 17 and 18 of our code?

```
16      /* Allocate memory for each array */
17          array1 = (int *)malloc(nelem*sizeof(int));
18          array2 = (int *)malloc(nelem*sizeof(int));
19          del = (int *)malloc(nelem*sizeof(int));
20
```

Because the memory for *array1* and *array2* were allocated separately, they should have different addresses. We can confirm this by setting a new breakpoint at line 20 and printing out the values of *array1* and *array2* immediately after this memory allocation is made,

```
(gdb)clear 38
Deleted breakpoint 1
(gdb) b 20
Breakpoint 2 at 0x80483e9: file variablePrinting.c, line 20.
```

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: variablePrinting

Breakpoint 2, main () at variablePrinting.c:22
22          initArray(nelem, array1);
(gdb) p array1
$7 = (int *) 0x80498d8
(gdb) p array2
$8 = (int *) 0x8049910
(gdb)
```

This confirms that, initially, the addresses of the two arrays are not the same. Since the address of *array1* is exactly the same here as in the computation of *del[indx]* on line 43, but the address of *array2* does change then the error in our code must be modifying the address of *array2*. To find out exactly where in our code the address of *array2* changes, we move down the code line-by-line, starting at line 20, examining the value of array2.

```
(gdb) n
24          for (indx = 0; indx < nelem; indx++)
(gdb) disp array2
1: array2 = (int *) 0x8049910
(gdb) s
26              array1[indx] = indx + 2;
1: array2 = (int *) 0x8049910
(gdb) s
24          for (indx = 0; indx < nelem; indx++)
1: array2 = (int *) 0x8049910
(gdb) s
26              array1[indx] = indx + 2;
1: array2 = (int *) 0x8049910
(gdb) s
24          for (indx = 0; indx < nelem; indx++)
1: array2 = (int *) 0x8049910

  .   .   .
  .   .   .

30          printf("\n");
1: array2 = (int *) 0x8049910
(gdb) s

31          printf("   array1 = ");
1: array2 = (int *) 0x8049910
(gdb) s
32          printArray(nelem, array1);
1: array2 = (int *) 0x8049910
(gdb) s
printArray (nelem_in_array=12, array=0x80498d8) at variablePrinting.c:79
79          printf("[  ");

  .   .   .
  .   .   .

85          printf("]\n\n");
(gdb) s
   array1 = [  2  3  4  5  6  7  8  9  10  11  12  13  ]

86       }
(gdb) s
```

```
main () at variablePrinting.c:35
35          array2 = array1;
1: array2 = (int *) 0x8049910
(gdb) s
38          squareArray(nelem, array2);
1: array2 = (int *) 0x80498d8
(gdb) l
```

What we find is that the address of *array2* changes after line 35, just before *array2* is passed to the function squareArray( ). If we look carefully at our code, the reason is rather obvious. When we wrote the code to copy *array1* to *array2* on line 35, our intent was to equate the corresponding elements in the two arrays, *array1* and *array2*. The code on line 35 does exactly this, but it does it by pointing the first element in *array2* to the address of the first element in *array1*. That is, writing

```
array2  =  array1
```

is equivalent to writing

```
&array2[0]  =  &array1[0]
```

And, since the elements in each array are contiguous, then this is also equivalent to writing

```
&array2[ k ]  =  &array1[ k ]  , for  k  = 0, 1, . . . ,  (nelem − 1) .
```

So, we now have two pointers, *array1* and *array2*, pointing to the same data (the data at 0x80498d8). Therefore, when the value of the data element at 0x80498d8 is modified by passing to the function squareArray(), both arrays will now point to the same modified data at 0x80498d8. Consequently, when the elements of *array2* are squared through the call

```
squareArray(nelem, array2)
```

on line 38, the elements of *array1* will also be squared and

```
del[ k ]  =  array2[ k ]  − array1[ k ]  =  0 , for  k  = 0, 1, . . . ,  (nelem
− 1)
```

What we actually meant to do on line 35 is not set the addresses of the two arrays equal,

```
array2  =  array1  ,
```

but, instead, to set the element values of the two arrays equal,

```
array2[ k ]  =  array1[ k ]  ,  for  k  = 0, 1, . . . ,  (nelem − 1)
```

Therefore, to correct our code, we need to replace the statement on line 35 with the following code,

```
for (indx = 0; index < nelem; indx++)
{
  array2[ k ]  =  array1[ k ]
}
```

When we make this correction and then recompile and rerun our code, we obtain the following correct output.

```
$ gcc -g -o variablePrinting variablePrinting.c

$ ./variablePrinting

   array1 = [  2  3  4  5  6  7  8  9  10  11  12  13  ]
```

The difference in the elements of *array2* and *array1* are:

```
del = [  2  6  12  20  30  42  56  72  90  110  132  156  ]
```

(e. It is possible to write multiple elements of dynamically-allocated arrays using the binary @ operator. To print the first ten elements, use 'print array1[0]@10'. Static arrays can be printed simply as 'print arraystatic'.)

Exercise 2: arrays

In this exercise we use the Intel compiler and the Intel-version of gdb, called gdb-ia.

```
$ icc -g -o array array.c
$ gdb-ia -tui array
```

a. Now we run the code line-by-line up to the beginning of the summation loop and then stop. To do this, we set a breakpoint at line 19 using the 'break' command:

```
(gdb) break 19
Breakpoint 1 at 0x10778: file array.c, line 19.
```

Then we type the 'run' command to execute the program to the point that we just entered:

```
(gdb) run
Starting program: /nfs/homea/dje/WebCT/debug/array
Breakpoint 1, main (argc=1, argv=0xffbefb54) at array.c:19
19          for(i=0;i<(N-1);++i) {
```

b. To do this, use the 'print' debugger command, which shows the value of a variable. The values look ok.

```
(gdb) print tock[2]
$1 = 1
```

```
(gdb) print tock[7]
$2 = 4
(gdb)
```

c. Use '*disp evensum*'

d. After 1 iteration we see that *evensum* is much higher than expected so we now suspect that the value of *tock[0]* is not correct. Using the 'print' command to check the value confirms that it is indeed wrong:

```
(gdb) print tock[0]
$4 = 224676
```

We have found the problem: the array element *tock[0]* was not initialized correctly. In fact, by looking at the initialization loop you can see that it was not initialized at all. The array index was set to start at 1 (like in Fortran) and not 0 (like in C). Therefore, the array element *tock[0]* contained a random, and incorrect, value rather than zero as was intended.

e. simply '*print tock*' in this case. The variable tock is a static array, so the debugger knows how to print it.

Exercise 3: learning valgrind

a.

```
$ valgrind ./variablePrinting
==30658== Memcheck, a memory error detector
==30658== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
```

```
==30658== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==30658== Command: ./variablePrinting
==30658==
array1 = [  2  3  4  5  6  7  8  9  10  11  ]

The difference in the elements of array2 and array1 are:  [  0  0  0  0  0  0  0
0  0  0  ]

==30658== Invalid free() / delete / delete[]
==30658==    at 0x40248B6: free (vg_replace_malloc.c:325)
==30658==    by 0x80485E9: main (variablePrinting.c:50)
==30658==  Address 0x4187028 is 0 bytes inside a block of size 40 free'd
==30658==    at 0x40248B6: free (vg_replace_malloc.c:325)
==30658==    by 0x80485DD: main (variablePrinting.c:49)
==30658==
==30658==
==30658== HEAP SUMMARY:
==30658==     in use at exit: 40 bytes in 1 blocks
==30658==   total heap usage: 3 allocs, 3 frees, 120 bytes allocated
==30658==
==30658== LEAK SUMMARY:
==30658==    definitely lost: 40 bytes in 1 blocks
==30658==    indirectly lost: 0 bytes in 0 blocks
==30658==      possibly lost: 0 bytes in 0 blocks
==30658==    still reachable: 0 bytes in 0 blocks
==30658==         suppressed: 0 bytes in 0 blocks
==30658== Rerun with --leak-check=full to see details of leaked memory
==30658==
==30658== For counts of detected and suppressed errors, rerun with: -v
==30658== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 16 from 11)
```

It shows an invalid free at line 50 of our program. The memory is already freed in line 49. Furthermore, valgrind shows 40 bytes that are definitely lost, but we need to rerun valgrind to see the details:

```
$ valgrind --leak-check=full ./variablePrinting
...
==30661== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==30661==    at 0x4024C9C: malloc (vg_replace_malloc.c:195)
==30661==    by 0x80484B7: main (variablePrinting.c:18)
...
```

which shows that the memory allocated for array2 is lost. This indicates that the pointer to array2 was lost after the allocation, which we found out to happen at line 34. So, valgrind detects the same problems as our manual debugging in the first exercise.

c. valgrind observes and detects uninitialized values, also if these are copied or used in other variables. A complaint is issued only when your program attempts to make use of uninitialised data in a way that might affect your program's externally-visible behaviour. That is in this case the printing of the variable evensum. Valgrind detects the use of the same uninitialized variable at several locations in the printing routine and each time it prints a warning. The amount of warnings is therefore not directly related to the amount of errors in your code. The option --track-origins=yes indicates that the uninitialized values come from the array tock, which again agrees with the conclusion in the second exercise after manual debugging.

Exercise 4: Dynamic summation

The input file only has 25 values in it, but the array *tock* can hold 50 integers. So after the array values are input in the while loop, what values are in the last half of *tock*? In C, arrays are not

intialized to anything at declaration. Therefore, if only the first half of *tock* is filled with values, the last half contains whatever happened to be in those memory locations, which is junk. So when the summation is done over all 50 elements, 25 unwanted integers are addded to the sum (and some of these junk integers can be quite large). That explains why the incorrect sum is so large.

There are many ways to fix this problem. A rather simple approach is to initialize all 50 elements of *tock* directly to zero:

```
int tock[N]={0};  /* HERE IS WHERE THE ONLY CHANGE IS MADE */
```

Therefore, only zeros from the last 25 elements are added to the sum, which have no effect. Another approach is to add another loop variable *j* for the summation loop that sums the indices *0..i-1*. This approach is implemented in the solution code.

valgrind would show the uninitialized values.

Exercise 5: Fortran I/O

a. the problem is that the file does not even exist. Although there is a file with the name sectionsCT.txt after running the executable, it is empty and newly created. The creation of a new file is the default behaviour of Fortran for opening files without additional arguments.

b. the command should be: OPEN(UNIT=10,FILE="sectionsCT.txt",**status='OLD')**. An extra argument ACTION='READ' opens the file read-only.

c. The following table shows the expected and actual results obtained:

	Expected	**Actual**
Id	100103	100103
Name	"Rufenacht "	" Rufenacht "
Country	"SWZ"	"SWZ"
Rating	2493	24
Title	"GM"	"93"
Experience	254	0

when printing strings, it is good to print special characters before and after the input, which explicitly show where the string begins and ends, e.g. print*, 'Name: *',name,'*'

d. READ(10,FMT='(I6,TR2,A14,A3,TR2,I4,TR1,A2,TR1,I3)')test(I)

e. there are still problems :

```
$ ls -l
total 96
-rw-r-----    1 ian      aef               348 Jun 03 15:24 SectionsCT.txt
-rwxr-x---    1 ian      aef             11174 Jun 03 16:41 badcode*
-rw-r-----    1 ian      aef              1482 Jun 03 16:41 bugs.f
-rw-r-----    1 ian      aef                 0 Jun 03 16:41 final.data
-rw-r-----    1 ian      aef               216 Jun 03 16:41 fort.22
-rw-r-----    1 ian      aef               396 Jun 03 16:41 out.data
```

The output file "final.data" is a zero byte file and there is an unexpected file "fort.22". This file contains the information expected to be written to "final.data". The suffix, ".22" is a hint that there is a problem with the unit, in this case unit 22. Looking at lines 41 and 49:

```
Line 41: OPEN(UNIT=21,FILE="final.data",STATUS='new')
Line 49: WRITE(22,FMT='(A2,A,A16,I3)')test(I)%title," ",test(I)%name,
```

we find another error — unit 21 is being opened, but the write statement is using unit 22.

Exercise 6: argument and type checking

The solutions can be found in the 'solution' directories.

Exercise 7: C++ pointer arithmetic

- addr2line: with the address you'll see that the error occurs at or just before the last line, when freeing newArray. It doesn't give enough information to fix the code straightaway. When compiling with optimization, addr2line gives an even less accurate answer.
- valgrind: it finds the invalid use of bytes after the allocated space for newArray when it is printed and notices that it tries to free an address that is 32 bytes into a block of 40 allocated bytes.

- gdb

- totalview

Exercise 8: real numbers

a. We want to examine the value of $x2$, which we have seen to be incorrect the first time through the loop. First, set a breakpoint at line 20, after $x2$ has been computed, and run the code to that point.

```
(dbx) stop at 20
[1] stop at 20

(dbx) run
   i       x       f(x)     int(f(x))
   1     0.00     1.00        0.00
[1] stopped in trapezoid at line 20
   20        f1 = cos(x1)
(dbx)
```

The expected value of $x2$ is 0.1*pi=0.31. Examine the actual value:

```
(dbx) p x2
0.0
```

Something is clearly incorrect here. Print the values used to calculate $x2$:

```
(dbx) p i,nm,pi
2 10 3.14159274
(dbx)
```

All these numbers are correct. Examining the source line where $x2$ is computed,

```
(dbx) l 19
   19        x2 = ((i-1)/nm)*pi
(dbx)
```

shows the problem. All the variables on the right-hand side have the correct values, but the resulting calculation is incorrect. This leads us to suspect some kind of type mismatch, and we indeed see that we are performing integer arithmetic rather than the required floating-point arithmetic. There is an additional caveat here: The debugger will not necessarily perform the same type conversions as the compiler. Going back to the version of the code with the bug, suppose we had chosen to print the value of the expression used to compute $x2$:

```
(dbx) p i,nm,pi
```

```
2 10 3.14159274
(dbx) p (i-1)/nm
0.10000000000000001
(dbx) p ((i-1)/nm)*pi
0.31415927410125732
```

These results are correct, even though the code computes an incorrect result for the same operations! You must exercise caution when computing expressions within the debugger, since the debugger may not handle variable typing in the same way as the source code.

b. you can use the GNU compiler flag -Wconversion-extra and see the error immediately. Of course the problem would be to know the correct flag, but searching through the man-page of your favourite compiler(s) can help

Exercise 9: dynamic memory

a) No, valgrind finds no problem in the user code when it is compiled with the Intel compiler. The Intel compiler reallocates the allocatable array on the left before the assignment. In the Fortran 2003 standard the left hand side would be assumed to be the right size and errors occur. The current standard reallocates the left hand side, as the Intel compiler defaults to.

For the gfortran compiler it depends on the version: 4.4.7 follows the Fortran 2003 standard, 6.3.0 follows the current standard

Array assignments only copy as many elements as available in the destination in the Fortran runtime. If the array is allocated but too small, even array bound checking does not find this problem.

b) As long as the array is not allocated, the optional array bound checking (-C) stops the program and warns you that the array is not yet allocated.

c) Test your application with different compilers and possibly on different platforms.

Compiler options to detect bugs

1) none available

2)

for C version, Intel:

```
[donners@login4.salomon array]$ icc -ftrapuv -o array array.c
[donners@login4.salomon array]$ ./array
oddsum=5.000000
evensum=-107374176.000000
```

for Fortran version, Intel:

```
[donners@login4.salomon array]$ ifort -init=snan,arrays -g -traceback -o array
array.f90
[donners@login4.salomon array]$ ./array
forrtl: error (182): floating invalid - possible uninitialized real/complex
variable.
Image              PC                 Routine           Line        Source
array              000000000047BF81   Unknown           Unknown     Unknown
array              000000000047A0BB   Unknown           Unknown     Unknown
array              0000000000447244   Unknown           Unknown     Unknown
array              0000000000447056   Unknown           Unknown     Unknown
array              0000000000426839   Unknown           Unknown     Unknown
array              00000000004031FC   Unknown           Unknown     Unknown
```

```
libpthread-2.12.s   00007FFFCEC0B7E0   Unknown                        Unknown   Unknown
array               00000000004029CA   MAIN__                              22   array.f90
array               000000000040282E   Unknown                        Unknown   Unknown
libc-2.12.so        00007FFFCE682D5D   __libc_start_main              Unknown   Unknown
array               0000000000402729   Unknown                        Unknown   Unknown
Aborted (core dumped)
```

for Fortran version, GNU:

```
[donners@login4.salomon array]$ gfortran -g -finit-real=snan -ffpe-
trap=zero,inv,over array.f90
[donners@login4.salomon array]$ ./a.out

Program received signal SIGFPE: Floating-point exception - erroneous arithmetic
operation.

Backtrace for this error:
#0  0x7fffcea5f69f in ???
#1  0x400947 in array
      at /home/donners/src/array/array.f90:22
#2  0x400a8d in main
      at /home/donners/src/array/array.f90:29
Floating point exception (core dumped)
```

Note that the GNU compiler does not initialize allocatable arrays with signalling NaN, so the use of this option is limited.

Answers parallel debugging

Answers to exercise 10: outsourced

a) when not using the libmpiwrap-library from Valgrind, we see many false positives inside the OpenMPI library itself. If we use libmpiwrap, the only call that uses uninitialized memory from our own program is the MPI_Send call. It finds that the buffer sdata is not initialized on the sending process. This is logical, since the buffer sdata is initialized on process 1 and sent from process 0.

b) Change the process that initializes sdata. See solution.

Answers to exercise 11: crossover

a) The code works fine with 100 or 1.000 elements, but it hangs at 10.000 elements. Small messages are usually sent straight-away to the receiver, which then buffers it and waits for the matching MPI_Recv call. The MPI_Send call then becomes a sort of asynchronous send and it can finish without having started the matching MPI_Recv call on the other process. This method is usually called 'eager messages'. Larger messages are not sent, to prevent running out of buffer space. The exact value where the MPI processes hang, will depend on the MPI library and the default settings for eager messages.

b) The order of the MPI_Send and MPI_Recv on one of the MPI ranks needs to be changed for it in order to work. See solution.

Answers to exercise 12: OpenMP

When serial and parallel results differ in an OpenMP code, it is often useful to apply the following simple heuristic test:

1. Run a serial version of the code (with no OpenMP directives or compiled so it does *not* interpret OpenMP directives)
2. Run the OpenMP version on a single thread.
3. Compare the serial and OpenMP results.
4. If the results are the same, suspect a data dependency.
5. If the results differ, suspect a scope error.

Running the OpenMP version of the code on one processor yields:

```
1.105E+01    4.8600    4.364E-09   -1.373E-05    2.403E-04   -7.563E-01

1.153E+01    5.0672    2.883E-09   -1.374E-05    1.588E-04   -7.569E-01

1.202E+01    5.2827    1.872E-09   -1.375E-05    1.031E-04   -7.573E-01


edge momentum factor =   0.000E+00
```

These results differ from the serial (correct) result. Using our simple heuristic test, we suspect a scope error since the serial run and the single-processor run of the OpenMP version give different results.

Add all relevant variables to your expression window.

Check all the values inside the OpenMP loop for their correctness.

Step now through the code until the incorrect output is written to the screen.

Note that the value of *xmom* has changed upon leaving the loop. This is because *xmom* was declared private, since each thread requires its own copy of *xmom*, and it explains the bug. Upon examination of the loop, it is clear that *xmom* actually should have been declared *lastprivate*, so the last value in the serial version of the loop would have been retained after leaving the loop.

Answers to exercise 13: prefix

We found that MPI rank 0 was not behaving in the same way as the other MPI processes. Whereas the other processes completed the program successfully, rank 0 was hung in an MPI_RECV() fairly early on in the program. Something in the construction of either our communication scheme or our binary tree configuration is flawed.

If we take another look at values returned by the functions parent(), child0(), and child1(), as shown in the following table, we find the answer:

mype	parent(mype)	child0(mype)	child1(mype)
i	i/2	2*i	2*i+1
0	0	0	1
1	0	2	3
2	1	4	5
3	1	6	7

Rank 0 thinks it is both its own parent **and** its own child, which causes it to try to receive a message from itself that it never has a chance to send! If we reexamine the tree functions (including is_leaf() and is_down()), they are all based on the idea of the process ranks starting at 1, but in MPI the ranks always start at 0. This problem can be easily fixed, by replacing the code:

```
c This is the code for what a normal node does
      else
```

with:

```
c This is the code for what a normal node does
      else if (mype.ne.0) then
```