

HPC Programming

Debugging, Part I

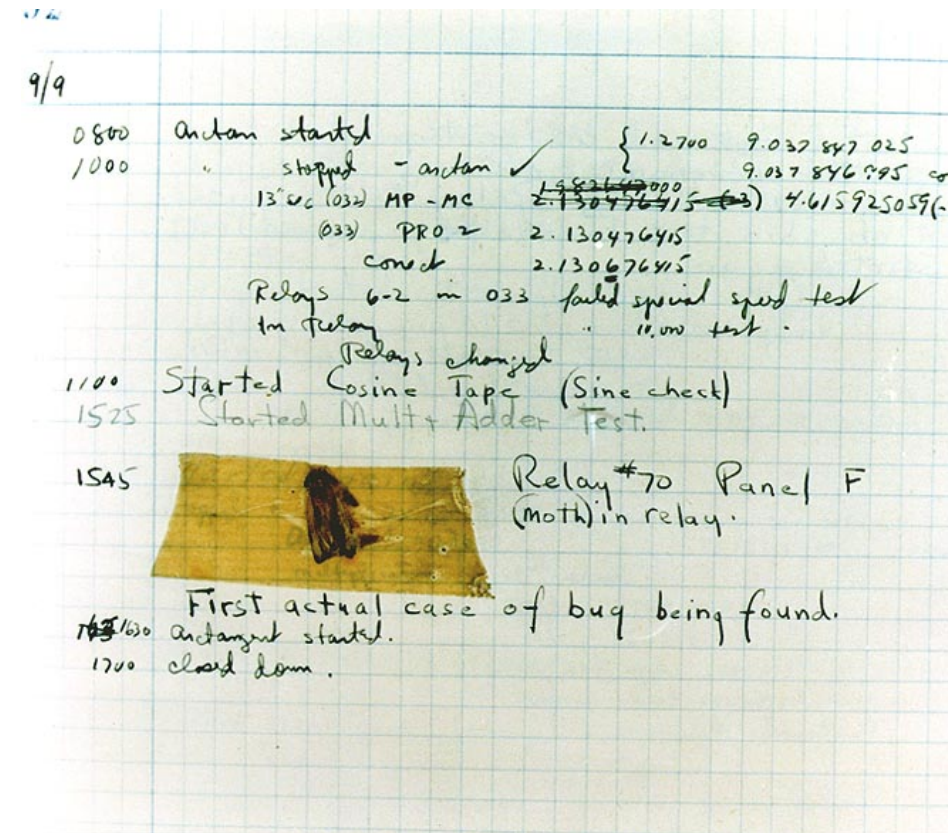
Peter-Bernd Otte, 14.1.2020

Debugging

1. Introduction / General Debugging
2. Typical bugs
3. Tools Overview
4. Introduction TotalView
5. Debugging with TotalView OpenMP
6. Debugging with TotalView MPI

Definition of a bug

- “bug” := errors or glitches in a program
→ incorrect result.
- most difficult part of debugging: finding the bug.
Once found, correcting is relatively easy
 - prove: bug bounty programs
 - debuggers: help programmers locate bugs by:
executing code line by line, watching variable values
- locating bugs is something of an art:
 - why? a bug in one section of a program cause failures in a completely different section
 - there is no defined right way to debug



1946, moth removed from relay

What's it all about

- humans write high level code, e.g. in C
BUT
- hardware understands assembler

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <sys/time.h>
4 #ifndef _OPENMP
5 # include <omp.h>
6 #endif
7
8 #define f(A) (4.0/(1.0+A*A))
9
10 int n = 100000000;
11
12 int main(int argc, char** argv)
13 {
14     int i;
15     double w,x,sum,pi;
16     clock_t t1,t2;
17     struct timeval tv1,tv2; struct timezone
18     # ifdef _OPENMP
19         double wt1,wt2;
20     # endif
21
22     # ifdef _OPENMP
23         int myrank, num_threads;
```

compiler
& linker

```
:::  _start:      xorl    %ebp, %ebp
0x00400521:
0x00400522:  mov     %rdx, %r9
0x00400523:
0x00400524:
0x00400525:  popl   %rsi
0x00400526:  mov     %rsp, %rdx
0x00400527:
0x00400528:
0x00400529:  andl   $-16, %rsp
0x0040052a:
0x0040052b:
0x0040052c:
0x0040052d:  pushl  %rax
0x0040052e:  pushl  %rsp
0x0040052f:  movl   $0x400810, %r8
0x00400530:
0x00400531:
0x00400532:
0x00400533:
0x00400534:
0x00400535:
0x00400536:  movl   $0x4007a0, %rcx
```

- error during execution? → today's topic

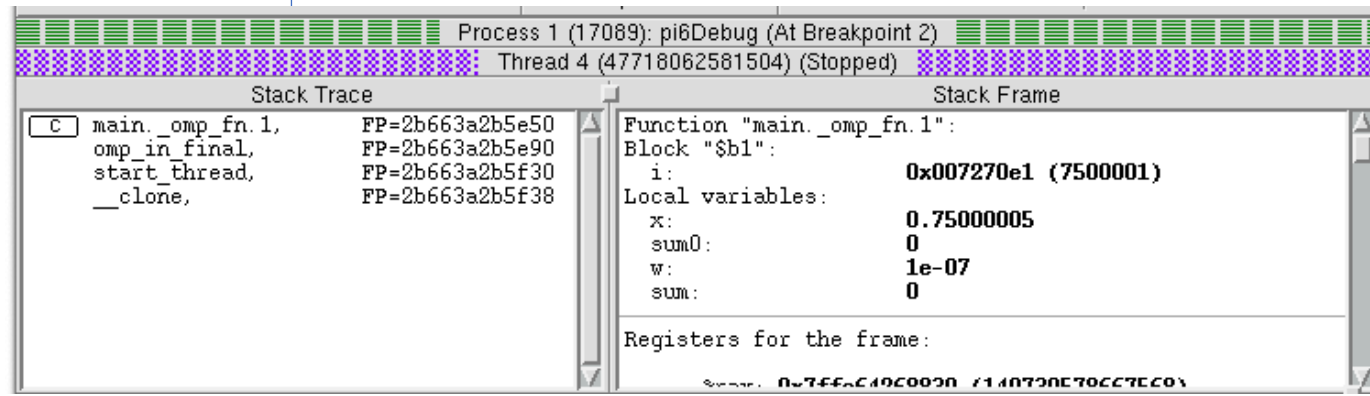
today's lecture topics

- Today we concentrate on following classes of bugs:
 - Arithmetic
 - Logic
 - resource
- Next lessons:
 - Multi-threading in OpenMP and multi-processing in MPI
 - Deadlock, Race condition, concurrency errors
- We concentrate on run-time and logical errors, no syntax or semantic (→ compiler) nor linker errors.

Call stack

- Call stack = stack of “stack frames”
- function call → new stack frame.
Removed when call ends
- “stack frame”:
 - local variables (in example: “c”)
 - argument parameters (in example: “a, b”)
 - return address (in example: “1st line in main()”)
 - saved copies of registers modified by subprograms which might get restored (in example: none)
 - has Frame Pointer (FP)
- LIFO (last in, first out)

```
int myfunc(int a, int b) {  
    int c;  
    //do some calculation  
    return;  
}  
int main () {  
    myfunc(a,b);  
}
```



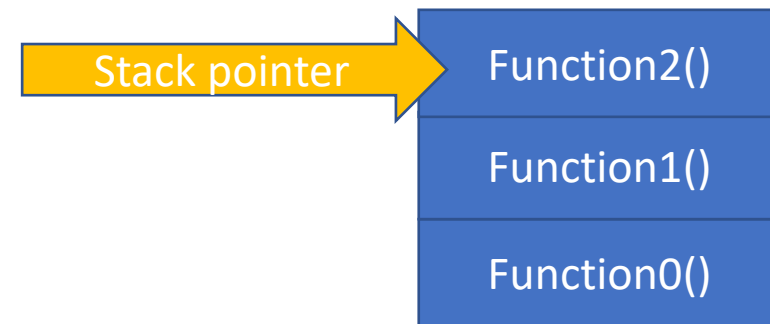
The screenshot shows a debugger window with the following content:

Process 1 (17089): pi6Debug (At Breakpoint 2)
Thread 4 (47718062581504) (Stopped)

Stack Trace		Stack Frame	
C	main._omp_fn.1, FP=2b663a2b5e50	Function "main._omp_fn.1":	
	omp_in_final, FP=2b663a2b5e90	Block "\$b1":	
	start_thread, FP=2b663a2b5f30	i:	0x007270e1 (7500001)
	__clone, FP=2b663a2b5f38	Local variables:	
		x:	0.75000005
		sum0:	0
		w:	1e-07
		sum:	0
		Registers for the frame:	
			0x7ff6e49c0020 (140720e70cc7ec0)


Program counter and Stack pointer

- Program Counter (PC):
 - Hardware register in processor, indicating the actual point in program sequence.
 - Stack Frame includes a return address
→ PC can be reset at end of called sub function
- Stack pointer:
 - Address register, that points to the top of the call stack



Debugging

1. Introduction / General Debugging
2. Typical bugs
3. Tools Overview
4. Introduction TotalView
5. Debugging with TotalView OpenMP
6. Debugging with TotalView MPI



let's focus on
the
problems...

Common bugs in C

- Arithmetic
 - div 0, over- or underflow, loss of precision
- Logic
 - infinite loops, infinite recursion, off-by-one error, syntactically correct “errors”
- Resource
 - null pointer dereference, uninitialized variable, wrong data type, access violations and use-after-free error, resource leaks, buffer overflow

- few examples to warm up...

uninitialized memory

```
double d;  
switch (i) {  
    case 0: d = 1; break;  
    case 1: d = 2; break;  
}  
printf("value of d: %f", d);
```

- value of d?
 - is arbitrary and depends on what is stored in memory before program launched
- “safety initialisation” recommended

value outside the domain

```
int x, y, z;  
//some calculation  
if ( (x+y) < z )  
    return 1;  
else  
    return 0;
```

- what is the result for $x=y=z=2E9$?
→ $(x+y)$ outside of int range → overflow → gets negative.

buffer overflow

```
main () {  
    char comment[100];  
    int *myPhDResult;  
    char sqlCommand[200] = "SELECT comments FROM users";  
    gets(comment);  
    //SQLExeceute(sqlCommand);  
    printf("My PhD Result: %i", myPhDResult);  
}
```

memory address



- any code which puts data in some buffer without checks → possible buffer overflow
- when $\text{size}(\text{entered value}) > \text{size}(\text{comment}) + 1$ → adjacent memory gets overwritten
- in C: no reliable error message during compilation or runtime! → debugger with memory checks helps OR use C++ String class

Arithmetic exceptions

- divide by zero (misnomer: “floating point exceptions” do cover int arithmetic errors too)
- off by one: starting a loop at 1 instead of 0, writing `<=` instead of `<`, etc...
(Mathlab and Fortran start at 1, python and C at 0)

Syntactically correct “errors”

single statement not in loop:

```
for (int i=0; i<10; i++); x++;
```

Using a single equal sign to check equality:

```
char x='y';  
while (x='y') { //Assigns 'y' to x, tests if x is zero  
    printf("continue? "); gets(y);  
}
```

- syntactically correct, but most likely different programmer intention
- stick to code formatting rules

Memory leaks I

- frequent in C, no automatic garbage collection
(check new techniques like smart pointer in C++11)
- more memory gets allocated during runtime (and halts when all is eaten up)

```
for (;;) {  
    char *out = (char*) malloc (size);  
    /*do some stuff  
    and forget to free*/  
}
```

Memory leaks II


- Overstepping array boundaries

```
int array[10];
int myPhDResult;
for (int i=1; i<=10; i++)
    printf(“%d ”, array[i]);
```

- No hint or halt during runtime
- Only a memory checker finds this error.



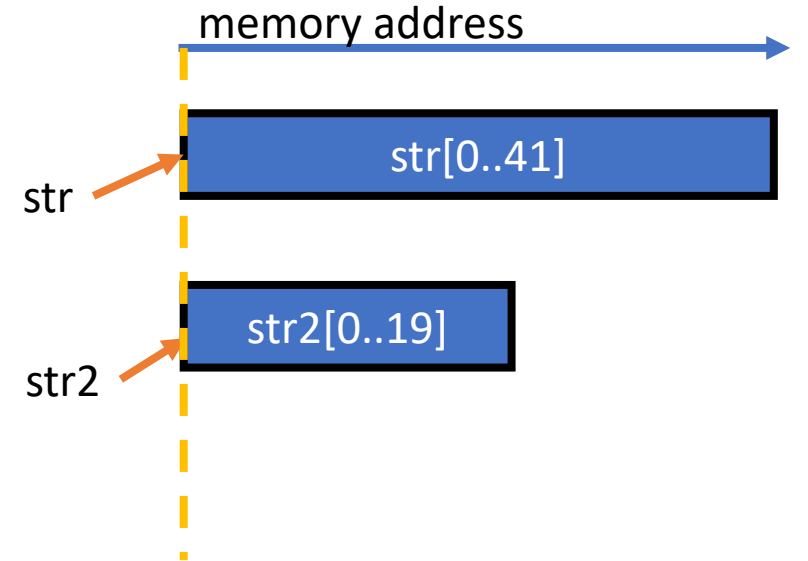
return temporary value

```
char *myfunc() {  
    char ch;   
    //so some stuff  
    return (&ch);  
}
```

- local variable address from stack is returned
- solution: declare the variable as public before calling myfunc()

free the already freed resource

```
int myfunc(int global) {  
    char *str = (char*) malloc(42);  
    if (global == 0) free(str);  
    //some more statements  
    char *str2 = (char*) malloc(20);  
    free(str);  
    //use of str2 problematic now  
}
```



- free the already freed resource, but str still points to the old address
- affects by chance the newly allocated variable

NULL dereferencing

```
char *c; //might be NULL
if (x>0) c='h';
printf("The character c is: %c", *c);
```

- if c is NULL → dereferencing fails
- when dereferencing an object, makes sure it initialised in any path.

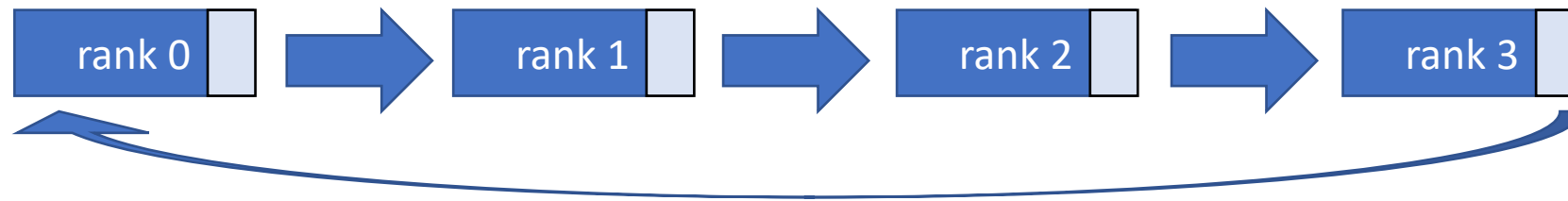
Aliasing

```
char str[42] = "Test Str";  
char *str2 = &str; //bad: str2 is now an alias of str  
strcat(str, str2);
```

- we **may** get an runtime error
- (strcat is no safe function, buffer overflow when 2nd argument > 1st)
- Aliasing creates problems when different addresses are expected. → try to avoid
- functions which expect parameters to be in certain format → be cautious!

Deadlocks, Race condition

- see lectures from OpenMP and MPI
- Deadlock: cyclic list, all threads proceed when receive OK from predecessor



- Race condition:
multiple threads (min. one write access), shared resources, result depends on scheduler
- Debugging OpenMP and MPI → next lecture.

How to avoid bugs

- Switch on compiler warnings! ... and pay attention to them
- Use of simpler methods
 - Split larger methods into small, cohesive ones!
 - Intuitive idea of what's being done
 - few parameters only, best with named parameters
- Mixing up various operations in a single expression → confusion
 - Split complicated expressions!
 - Make it easy for the debugger

```
if ( (42.*b/c++) > (43.+(10.*e/f)) ) {  
    /*do some operation A*/  
} else {  
    /*do some operation B*/  
}
```

```
double a = 42. * b / c++;  
double d = 43. + ( 10. * e / f);  
if (a > d) {  
    /*do some operation A*/  
} else {  
    /*do some operation B*/  
}
```

Hints

- Problem?
 1. remove all object, intermediate or temporary files (“artefacts” in git terms)
 2. Rebuild with debugging info on (-g) and optimisation off (-O0)
 3. Still problematic? --> debugger!
- Debug first a serial version of your program
- Some errors only occur:
 - with optimized code (possible reasons: initialized variables? Wrong pointers? Buffer overflow?)
 - outside of debug session (possible reason: different timing?)
 - with many processes

Debugging

1. Introduction / General Debugging
2. Typical bugs
3. Tools Overview
4. Introduction TotalView
5. Debugging with TotalView OpenMP
6. Debugging with TotalView MPI

Debuggers in general

- test and debug a target program
- Common features:
 - flow control (run, step, into)
 - actions points
 - view registers values
 - view call stack
 - inspect and edit program memory

Debug Tools Overview for C

- **GDB** (OpenSource):
 - **Minus:** not optimal for beginners, multi-thread and multi-process possible
- **Valgrind** (OpenSource):
 - **Plus:** detect memory leaks or cache misses, works also for threads
 - **Minus:** does not run programs in parallel, threads are serialised. Only minimal MPI support
 - Modules on Himster2: `debugger/Valgrind/<version>-<toolchain>`
- **Intel's Vtune**
 - Profiler for serial and parallel code, OpenMP and MPI
- **RogueWave's TotalView** (Closed Source)
 - More in this lecture
 - **Plus: User friendly; serial, threaded and multi-process programs**
 - **Minus:** Costs you "an arm and a leg"
 - Modules on Himster2: `debugger/TotalView/2018.0.5_linux_x86-64`
- **For Python:** `pdb`, TotalView or included ones in GUIs

- Which to chose? Availability on your platform and fits your needs.

Debugging

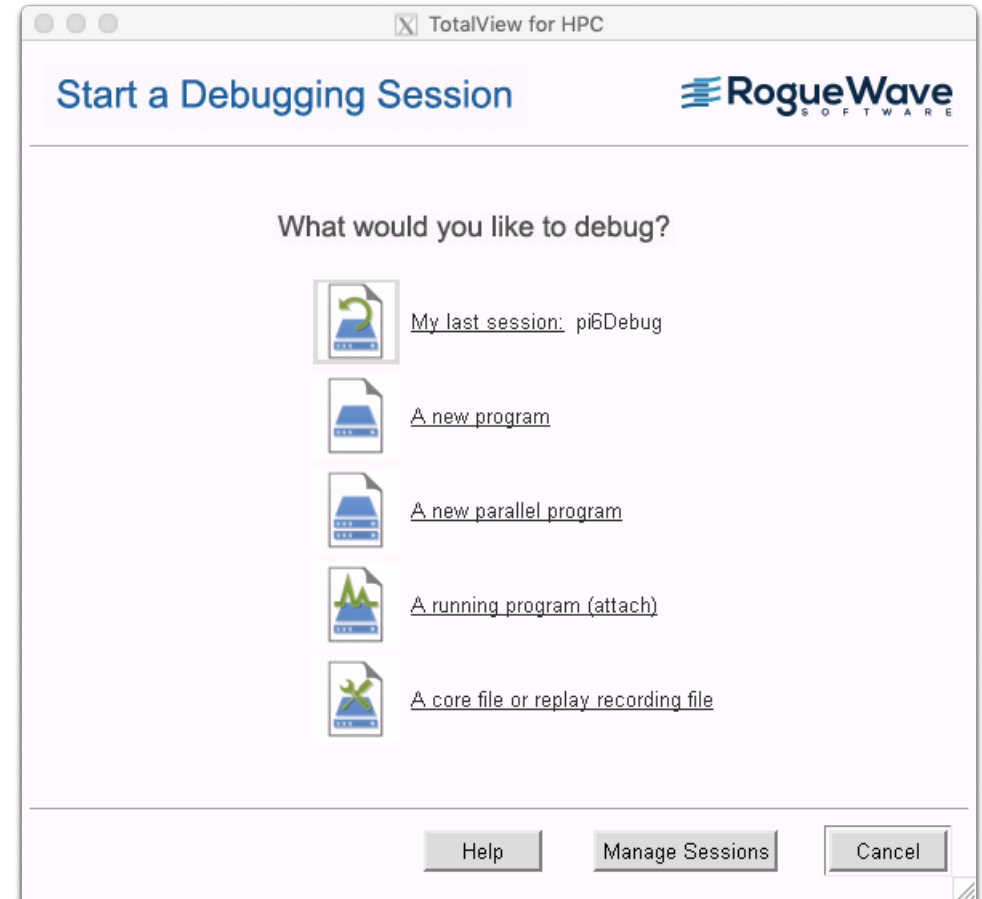
1. Introduction / General Debugging
2. Typical bugs
3. Tools Overview
4. Introduction TotalView
5. Debugging with TotalView OpenMP
6. Debugging with TotalView MPI

How to use totalview

- via command line, on Himster head node:
`$ module load debugger/TotalView/2018.0.5_linux_x86-64`
- interactively:
`$ totalview &`
- Normal:
`$ totalview [totalviewArgs] executable [-a executable_args]`
- Attach to running program:
`$ totalview [totalviewArgs] executable -pid [PID#] [-a executable_args]`
find out PID# with
`$ps ax`
- Attach to a core file:
`$ totalview [totalviewArgs] executable coreFileName [-a executable_args]`

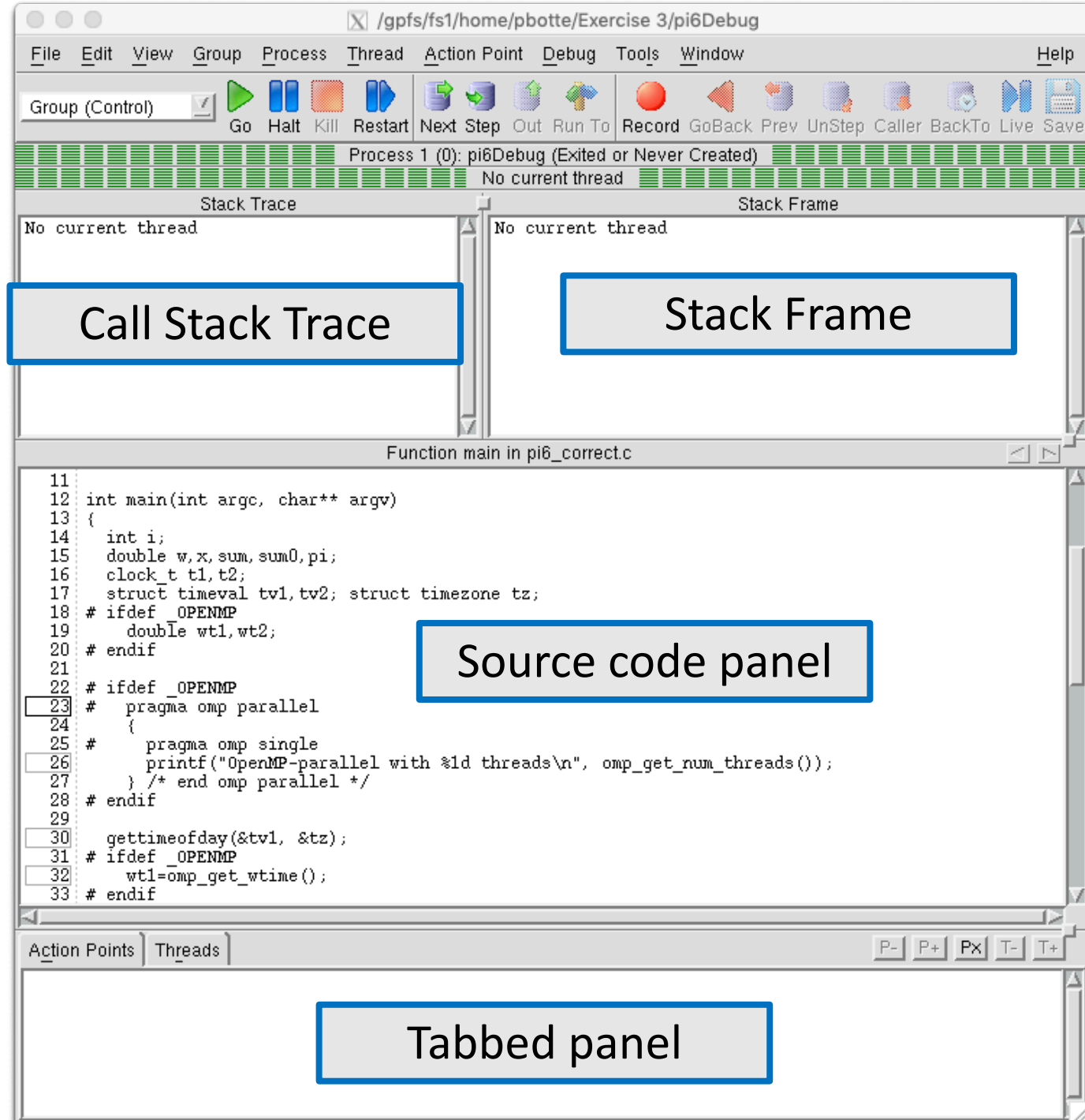
Totalview and Preparations

- Main features:
 - Interactive debugging
 - Attaching to a process
 - Analyse core-dumps
 - reverse debugging (reverse anytime during debug)
- To enable debugging
 - debug enabled compilation: `-g`
 - creates pointers to your source code lines
 - source code still needs to be available at the path during compilation
 - 1st step: no optimisations: `-O0`
 - later use `-O3`
 - may change the behaviour of your program with different errors



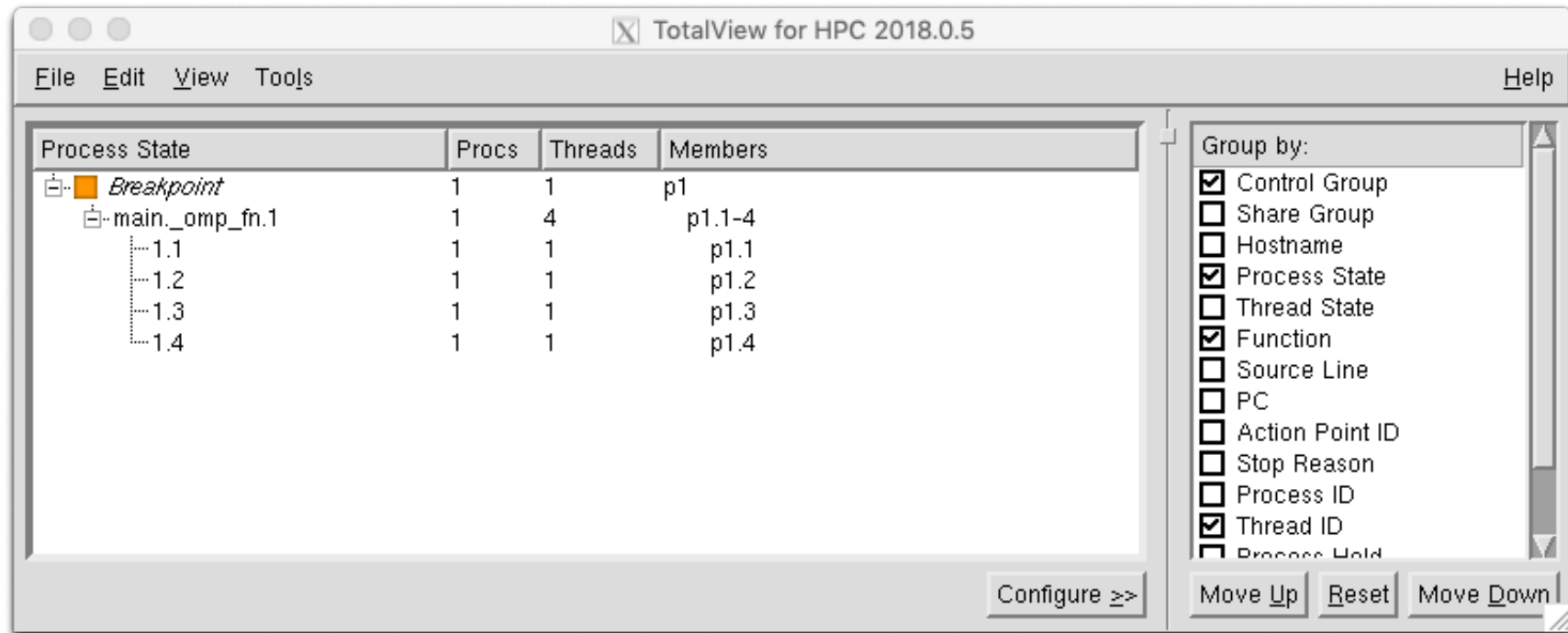
TotalView

- “Standard tool” for parallel debugging (OpenMP, MPI, CUDA)
- Wide compiler (Python, C, Fortran) and platform support (Linux, Unix, MacOS, no Windows)
- Process window:
 - State of one process / thread



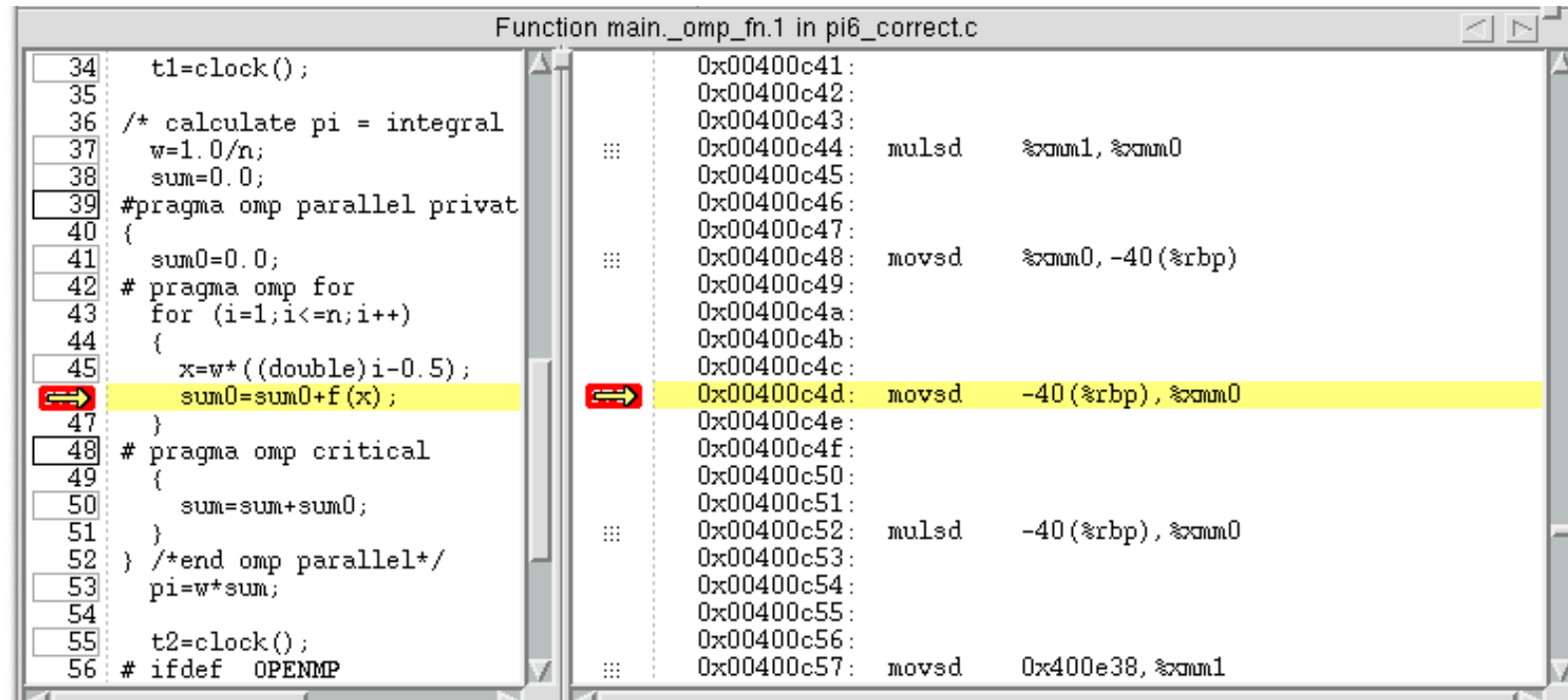
TotalView Root Window

- provides details of state of all processes and threads → important for next lecture



TotalView Source Code Panel

- Toggle Source: Code and/or Assembler (View > Source) (make sure to use “-g”)

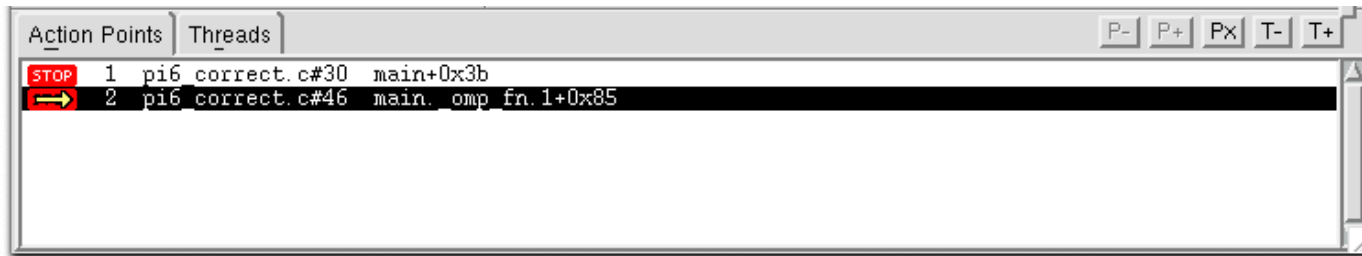


The screenshot shows the TotalView Source Code Panel for the function `main_omp_fn.1` in `pi6_correct.c`. The panel is split into two columns: Source Code on the left and Assembly Code on the right. The source code is C code with OpenMP pragmas. The assembly code is x86_64 assembly. The line `sum0=sum0+f(x);` in the source code and the corresponding assembly instruction `movsd -40(%rbp), %xmm0` are highlighted in yellow. A red arrow points from the source code line to the assembly instruction.

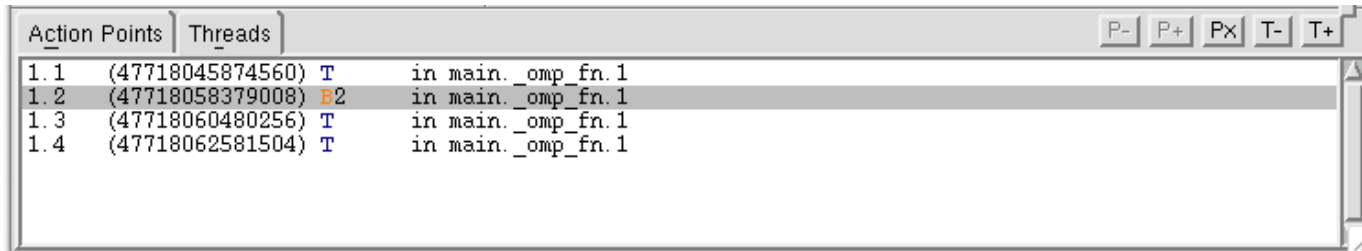
```
Function main_omp_fn.1 in pi6_correct.c
34  t1=clock();
35
36  /* calculate pi = integral
37  w=1.0/n;
38  sum=0.0;
39  #pragma omp parallel privat
40  {
41  sum0=0.0;
42  # pragma omp for
43  for (i=1;i<=n;i++)
44  {
45  x=w*((double)i-0.5);
46  sum0=sum0+f(x);
47  }
48  # pragma omp critical
49  {
50  sum=sum+sum0;
51  }
52  } /*end omp parallel*/
53  pi=w*sum;
54
55  t2=clock();
56  # ifdef OPENMP
57  0x00400c41:
58  0x00400c42:
59  0x00400c43:
60  ::: 0x00400c44:  mulsd   %xmm1, %xmm0
61  0x00400c45:
62  0x00400c46:
63  0x00400c47:
64  ::: 0x00400c48:  movsd   %xmm0, -40(%rbp)
65  0x00400c49:
66  0x00400c4a:
67  0x00400c4b:
68  0x00400c4c:
69  => 0x00400c4d:  movsd   -40(%rbp), %xmm0
70  0x00400c4e:
71  0x00400c4f:
72  0x00400c50:
73  0x00400c51:
74  ::: 0x00400c52:  mulsd   -40(%rbp), %xmm0
75  0x00400c53:
76  0x00400c54:
77  0x00400c55:
78  0x00400c56:
79  ::: 0x00400c57:  movsd   0x400e38, %xmm1
```


TotalView Tabbed Panel

- Action Points (Right click: Dive, Delete, Disable, Modify)
 - Add: Click on line number in source panel (only code after optimisation possible to add)

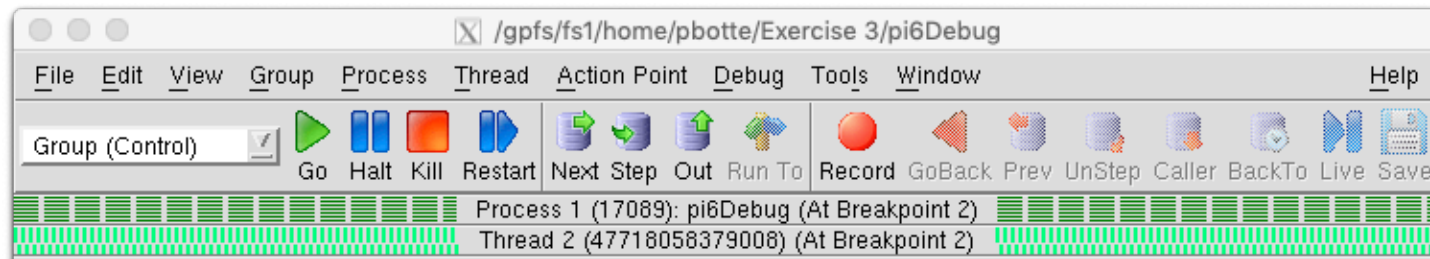


- Threads and Processes with their status



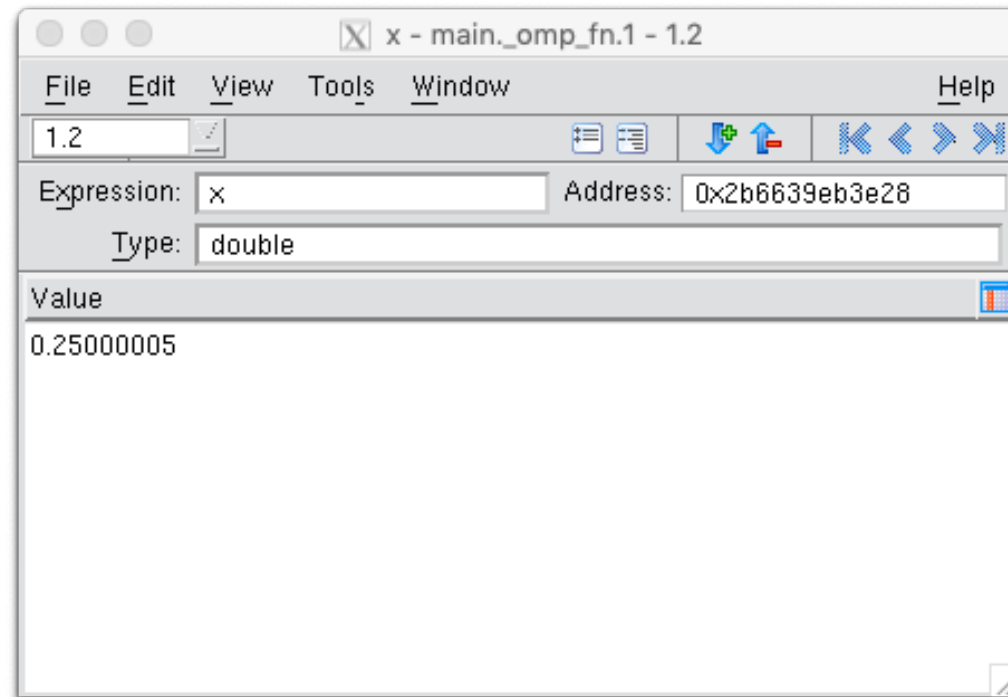
TotalView Stepping Commands

- Select how to proceed from actual PC location
 - next: Next line in same function
 - step: go into sub function
 - return to: go into end of sub function
 - out: leave current function
- Select group of threads / processes affected



TotalView Diving

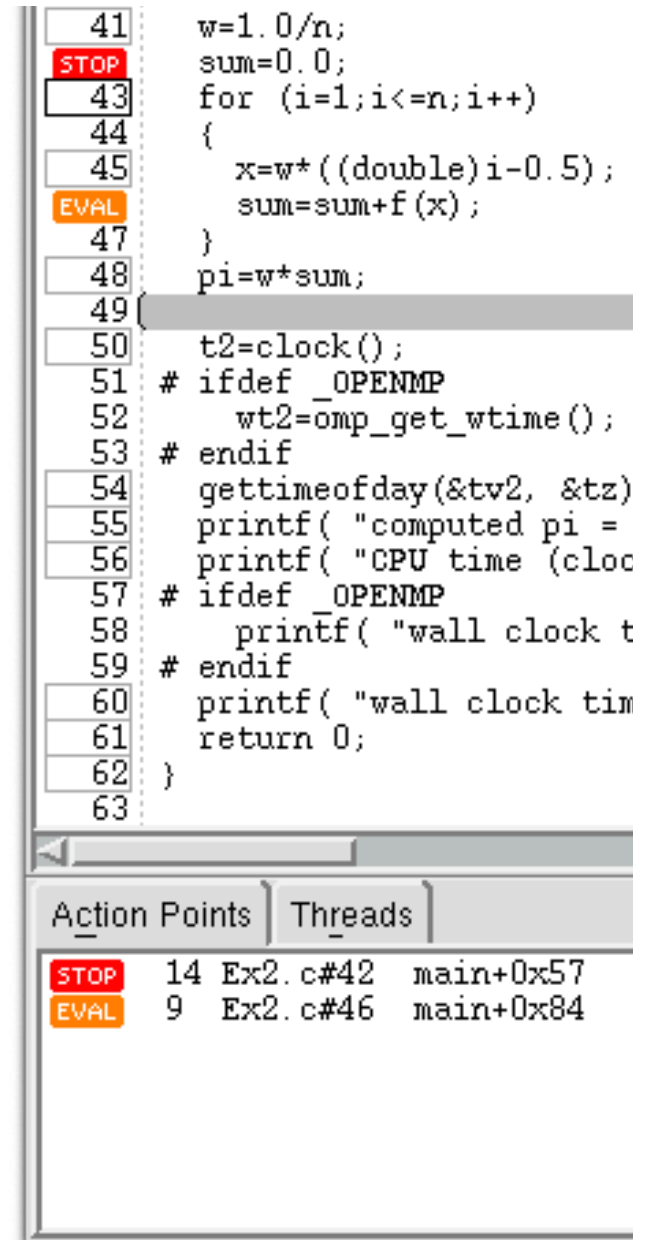
- Double click on variable: “Dive” to get more information



- Right click on value and change it. → live update!

TotalView action points

- click on the source code line surrounded by a small grey box
 - make sure you compiled with the correct parameters (-g -O0)
- Execution will stop once this line is reached.
To proceed, press Go or other stepping controls.



The screenshot shows a code editor with the following C code:

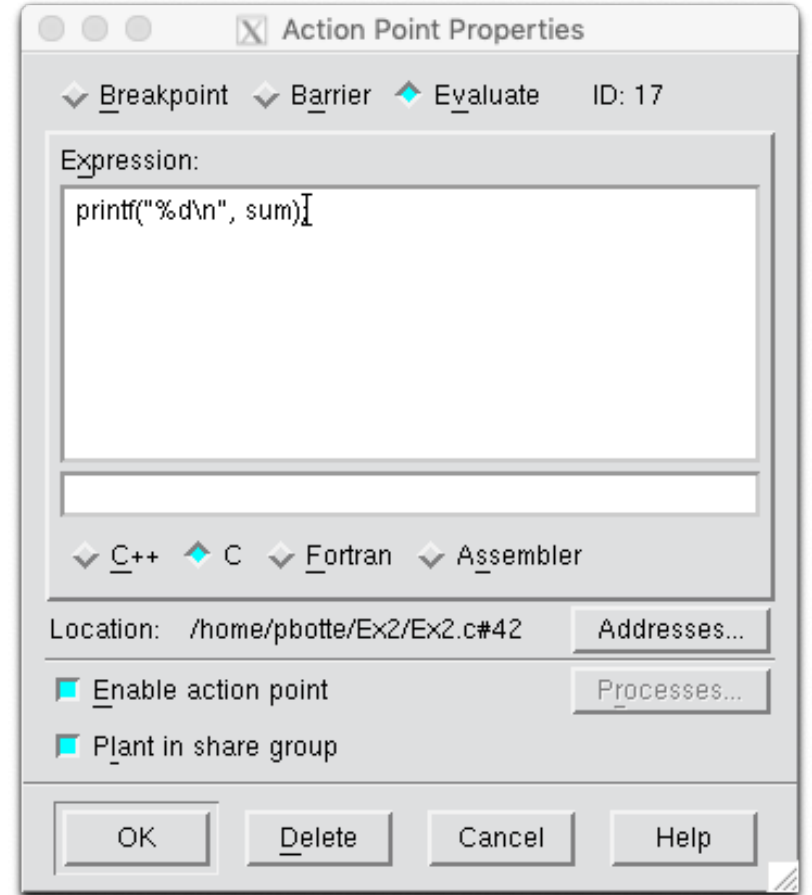
```
41 w=1.0/n;  
42 sum=0.0;  
43 for (i=1;i<=n;i++)  
44 {  
45     x=w*((double)i-0.5);  
46     sum=sum+f(x);  
47 }  
48 pi=w*sum;  
49  
50 t2=clock();  
51 # ifdef _OPENMP  
52     wt2=omp_get_wtime();  
53 # endif  
54 gettimeofday(&tv2, &tz)  
55 printf( "computed pi =  
56 printf( "CPU time (cloc  
57 # ifdef _OPENMP  
58     printf( "wall clock t  
59 # endif  
60 printf( "wall clock tim  
61 return 0;  
62 }  
63
```

Below the code editor is a table with two tabs: "Action Points" and "Threads". The "Action Points" tab is active and shows the following data:

Action Point	Count	File	Address
STOP	14	Ex2.c#42	main+0x57
EVAL	9	Ex2.c#46	main+0x84

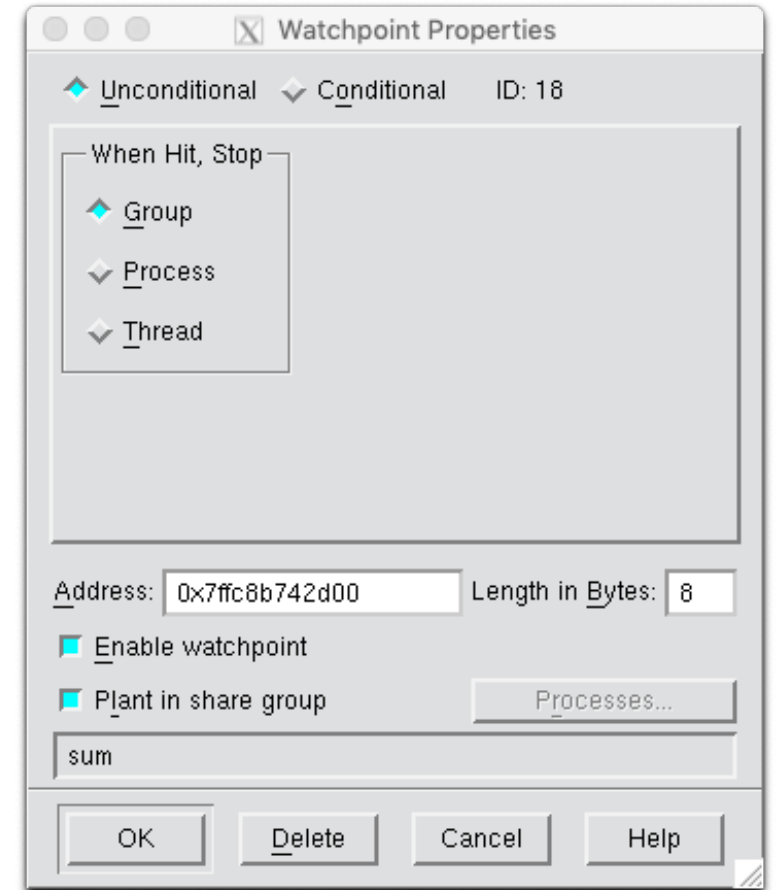
TotalView Evaluation

- Create a Action Point and change its properties to transform it into a Evaluation point.
- The Expression is executed, once the point is reached.
 - You can write full programs: change variables, conditions, etc.
 - Do a full test drive without recompiling.



TotalView Watchpoint

- Break point, when a register (variable value) changes
 1. Run your program from inside TotalView, halt it.
 2. From the menu select: “Action Point > Create Watchpoint” and enter your variable name.
- You can add conditional statement in the properties of the Action Point.
- These Watch Points can also be saved (see Menu Action Point), but by default they get deleted after execution.



Post-Mortem Analysis

Process does segmentation fault etc.

1. In bash: "ulimit -c unlimited" (check with ulimit -a and look for "core file size")
 2. Build your app with -O0 and -g and run
 3. Test: "kill -s SEGV <PID>"
 4. Core file will be generated in same directory
 5. Analyse with
"totalview executable coreFileName"
(or "gdb executable coreFileName")
- Supported on Himster2
 - Hint: With "gcore <pid> -o <filename>" a core dump is being generated and program remains running.

Live Demo I

1. Login into Himster 2 Headnode
2. Load module debugger/TotalView/2018.0.5_linux_x86-64
3. Run interactively totalview &
4. Provide Application Name and arguments
5. (Post-Mortem Debugging: Provide core file)
6. Setting Breakpoints
 - click in source pane
 - conditional breakpoints possible

Set up your workbench

- Connect two times via SSH to Mogon2 / HIMster2 and work on the head node
 1. Use the first SSH connection for editing (gedit, vi, vim, nano, geany) and compiling
\$ compiling: `gcc -g -O0 -o ExecutableName SourceFileName.c`
 2. Use the second connection for the interactive usage of TotalView:
\$ `module load debugger/TotalView/2018.0.5_linux_x86-64`
\$ `totalview &`

Exercise 1:

Learning objectives:

- Familiarise with TotalView
- Add temporal test code to your program

Steps:

1. Download the skeleton from OpenMP exercise 2 from the git repo:
2. Compile WITHOUT `-fopenmp` and open these programs in totalview. With and without:
 1. Debug flag: `-g`
 2. Optimisation: `-O0` and `-O3`
Check for source code panel and possible lines to set a break point.
3. get familiar with TotalView: Set a breakpoint, dive into variables, add variables to your expression list, step through your program
4. Change the number of iterations `n` after you launched your program to `n=10`. Why does it not work?
5. Compile your program again with a variable `n`. Change its value to 10 after your program has been launched.
6. Add an Action Point which evaluates the following: print out the value of `sum`.
hint: add `printf("%d\n", sum);`
→ Congratulations! You changed your program, did a test drive, without recompiling it!
7. Add an Watchpoint to be notified when the `pi` is changed.