TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

# Optimization, Profiling, & Debugging

For Modern Multicore Processors


**PRESENTED BY:**

Virginia Trueheart, MSIS

HPC Applications

Texas Advanced Computing Center

# An Overview

## Optimization & Profiling

- Write organized code
- Address architecture needs
- Address software needs
- Make good code better

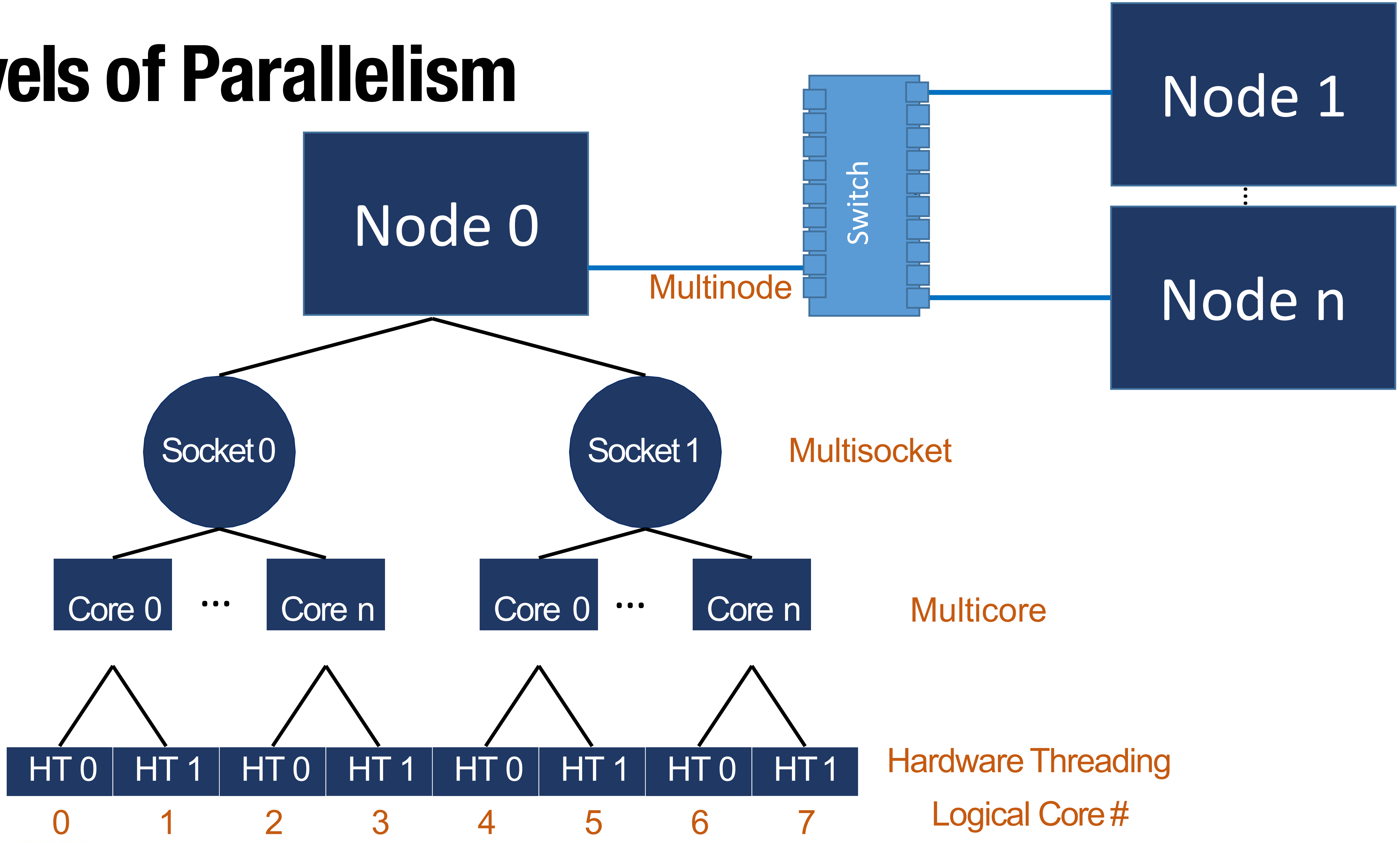
## Debugging

- Find problems in your code
- Fix said problems
- Find more problems
- (╯°□°)╯︵ ┻━┻

# Basic Optimization

- Parallelism
  - Divide computational tasks and vectorize
- Cache Usage
  - Smaller but faster storage
- Load Balancing
  - Maximize hardware and cache usage

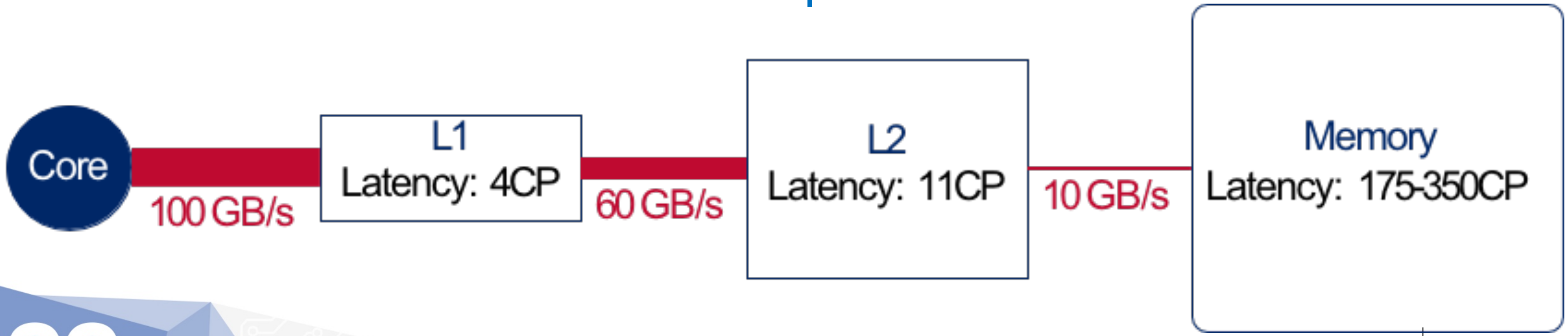
# Levels of Parallelism



# Caching

## Memory Hierarchy

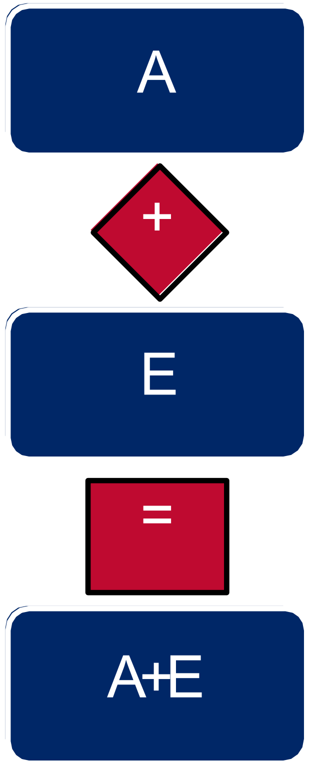
- Caches are smaller but faster than memory (higher bandwidth, less latency) than memory
- May have multiple levels of cache: L1, L2, LLC . . .
- Keeps cores fed with data
- Cache reuse is often critical to performance



# Vectorization

## Basics

- Cores have registers
- Data must be moved from cache/memory into registers before operating on
- To add 2 integers:
    1. Move first integer A from cache to a register
    2. Move second integer E from cache to a different register
    3. Add integers and place result in yet a different register
    4. Move result to cache/memory
- Frequencies are limited so they can only go so fast



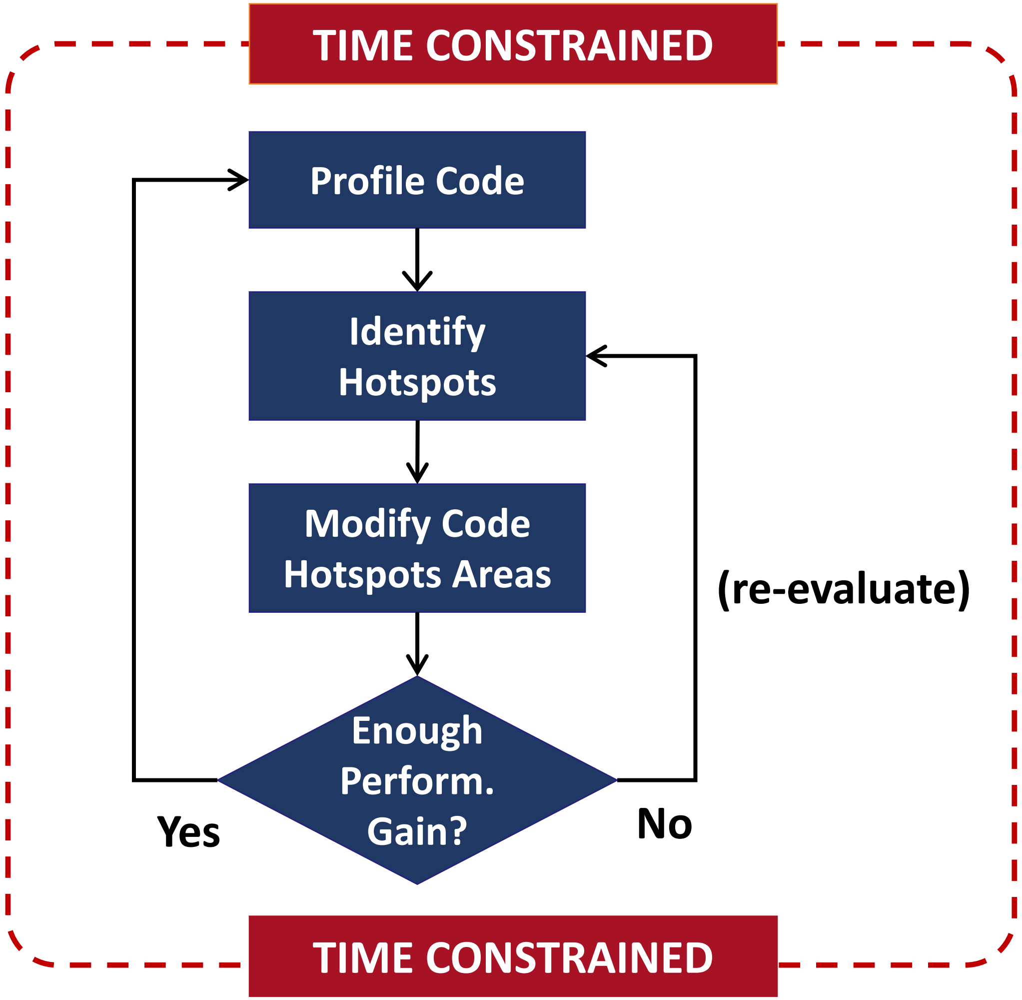
# Why Vectorization Matters

```
$ icc vector.c -no-vec
-o no-vec

$ ./no-vec
run time = 45.704987s
sum of a =
8388608000.000000
```

```
$ icc vector.c -xcore-
avx512 -qopt-zmm-
usage=high -o avx512-
vec-skx-zmm

$ ./avx512-vec-skx-zmm
run time  = 1.839410s
sum of a  =
8388608000.000000
```

TACC

# Optimization Improvement Process




- Iterative process
- Application dependent
- Different levels
  - Compiler Options
  - Performance Libraries
  - Code Optimizations

# Profiling Basics

## Controlled measurements are essential

- **Control**

  Don’t let the O/S decide process & memory affinity – **control it** !

- **Repeat**

  No single measurement is reliable

- **Automate**

  Don’t try to remember how you did something – **script it** !

- **Document**

  Save all these:

  ✓ Code versions
  ✓ Compilers / flags
  ✓ User inputs
  ✓ System environment
  ✓ Timers / counters
  ✓ Code output

Record Details:

Important routines should track work done and time taken

TACC

# Basic Profiling Tools

**CML Timer | Code Timer | gprof**

# Timer: Command Line

The command **time** is available in most Unix systems.

It is simple to use (no code instrumentation required).

Gives total execution time of a process and all its children in seconds.

```
$ /usr/bin/time -p ./exe
```

POSIX format

| | | |
|---|---|---|
| Elapsed wall clock time | | real 9.95 |
| Time spent in | user mode | user 9.86 |
| | kernel mode | sys 0.06 |

Leave out the `-p` option to get additional information:

```
$ /usr/bin/time ./exe
$ 9.860u 0.060s 0:09.95 99.9%    0+0k 0+0io 0pf+0w
```

| 9.860u | 0.060s | 0:09.95 | 99.9% | 0+0k | 0+0io | 0pf+0w |
|---|---|---|---|---|---|---|
| User | System | Real | %CPU | Shared Unshared | Input Output | Page faults Swap |

# Timer: Code Section

C

```c
#include <time.h>

double start, stop, time;
start = (double)clock()/CLOCKS_PER_SEC;

 /* Code to time here */

stop = (double)clock()/CLOCKS_PER_SEC;
time = stop - start;
```

FORTRAN

```fortran
INTEGER :: rate, start, stop
REAL    :: time

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
CALL SYSTEM_CLOCK(COUNT = start)

! Code to time here

CALL SYSTEM_CLOCK(COUNT = stop)
time = REAL( ( stop - start )/ rate )
```

C ( MPI )

```c
#include <mpi.h>

double start, stop, time;
start = MPI_Wtime();

 /* Code to time here */

stop = MPI_Wtime();
time = stop - start;
```

Python

```python
import time

start=time.time()

# Code to time here

end=time.time()
time = end - start
```

# gprof: GNU Project PROFiler

**Flat Profile**

- CPU time spend in each function (self and cumulative)
- Number of times a function is called
- **Useful to identify most expensive routines**

**Call Graph**

- Number of times a function was called by other functions
- Number of times a function called other functions
- **Useful to identify function relations**
- Suggests places where function calls could be eliminated

**Annotated Source**

- **Indicates number of times a line was executed**

# Profiling with gprof

Use the `-pg` flag during compilation:

```
$ gcc -g -pg -o exeFile ./srcFile.c
$ gfortran -g -pg -o exeFile ./srcFile.f90
```

Run the executable. An output file `gmon.out` will be generated with the profiling information.

Execute `gprof` and redirect the output to a file:

```
$ gprof    ./exeFile gmon.out > profile.txt
$ gprof -l ./exeFile gmon.out > profile_line.txt
$ gprof -A ./exeFile gmon.out > profile_anotated.txt
```

The code must be compiled with "-g"

Enable line-by-line profiling

Print annotated source code

# gprof: Flat Profile

In the flat profile we can identify the most expensive parts of the code (in this case, the calls to **`matSqrt`**, **`matCube`**, and **`sysCube`**).

| % time | cumulative seconds | self seconds | calls | self s/call | total s/call | name |
|---|---|---|---|---|---|---|
| 50.00 | 2.47 | 2.47 | 2 | 1.24 | 1.24 | matSqrt |
| 24.70 | 3.69 | 1.22 | 1 | 1.22 | 1.22 | matCube |
| 24.70 | 4.91 | 1.22 | 1 | 1.22 | 1.22 | sysCube |
| 0.61 | 4.94 | 0.03 | 1 | 0.03 | 4.94 | main |
| 0.00 | 4.94 | 0.00 | 2 | 0.00 | 0.00 | vecSqrt |
| 0.00 | 4.94 | 0.00 | 1 | 0.00 | 1.24 | sysSqrt |
| 0.00 | 4.94 | 0.00 | 1 | 0.00 | 0.00 | vecCube |

# gprof: Call Graph Profile

```
index % time    self  children    called     name
                0.00    0.00       1/1           <hicore> (8)
[1]     100.0   0.03    4.91       1          main [1]
                0.00    1.24       1/1           sysSqrt [3]
                1.24    0.00       1/2           matSqrt [2]
                1.22    0.00       1/1           sysCube [5]
                1.22    0.00       1/1           matCube [4]
                0.00    0.00       1/2           vecSqrt [6]
                0.00    0.00       1/1           vecCube [7]
-----------------------------------------------
                1.24    0.00       1/2           main [1]
                1.24    0.00       1/2           sysSqrt [3]
[2]      50.0   2.47    0.00       2           matSqrt [2]
-----------------------------------------------
                0.00    1.24       1/1           main [1]
[3]      25.0   0.00    1.24       1           sysSqrt [3]
                1.24    0.00       1/2           matSqrt [2]
                0.00    0.00       1/2           vecSqrt [6]
-----------------------------------------------
```

Called by

Called

This table describes the call tree of the program, and was sorted by the total amount of **time** spent in each function and its children.

The lines above it list the functions that called this function, and the lines below it list the functions this one called.

# gprof: -l and -A

```
$ gprof -l ./exeFile gmon.out
```
Enable line-by-line profiling

```
$ gprof -A ./exeFile gmon.out
```
Print annotated source code

Line-by-line profiling (-l)

The code must be compiled with "-g"

```
Flat profile:

Each sample counts as 0.01 seconds.

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name

100.00     47.21    47.21                             StaticFunc
(gprof_test.c:23 @ 40195a)

  0.00     47.21     0.00      100     0.00     0.00  StaticFunc
(gprof_test.c:18 @ 401927)

  0.00     47.21     0.00        1     0.00     0.00  TestFunc
(gprof_test.c:7 @ 4018c8)
```

Line number

Annotated source listing (-A)

```
                #include<stdio.h>
                void TestFunc();
                static void StaticFunc();
                void TestFunc()
           1 -> {
                        int i = 0;
                        printf("In TestFunc\n");
                        for (i=0; i<100; i++)
                                StaticFunc();
                }
                static void StaticFunc()
         100 -> {
                        int i = 0;
                        printf("In StaticFunc\n");
                        for (i=0; i<100000000; i++);
                }
                int main(void)
       ##### -> {
                        printf("In main\n");
                        TestFunc();
                        return 0;
                }
Top 10 Lines:
     Line      Count
       18        100
        7          1
Execution Summary:
        3   Executable lines in this file
        3   Lines executed
   100.00   Percent of the file executed
      101   Total number of line executions
    33.67   Average executions per line
```

# Advanced Profilers

When gprof isn’t enough you can consider some of the following tools:

- IPM (Integrated Performance Monitoring)
- HPCToolKit
- TAU
- VTune

Tutorials are available online for many of these and they can produce far more information about the jobs and help you generate graphs and other visualizations.

- Free
- Paid (but the viewer can be downloaded for free to your desktop)

# Basic Debugging

- Show backtraces
  - View the history of the code up to the point of interest
- Set breakpoints
  - Pause your code in various states
- Display the value of individual variables
  - Like print statements, this allows you to see the state of the code
- Set new values
  - Controlled values make mistakes easy to see
- Run individual steps of the program
  - Ensure that every function and subsection runs properly independently

# gdb: GNU Debugger

As with the profile, recompile with the `-g` flag

```
> gcc -g -o hello hello.c
```

gdb can be started 4 ways

| | |
|---|---|
| `> gdb` | Start an interactive gdb session |
| `> gdb a.out` | Spawn a new instance of ./a.out |
| `> gdb a.out corefile` | Examine trapped state in corefile |
| `> gdb a.out 1234` | Attach gdb to a running program with PID |

# hello.c

```
#include <stdio.h>
void foo();

int main ()
{
        printf("inside main\n");
        foo();
        return;
}

void foo ()
{
        int i, total=0;
        printf("inside foo\n");
        for (i=0; i<1000; i++)
                total += i;
}
```

# gdb Commands

`run` – starts program

`print` – print a variable located in current scope

`next` – executes current command and moves to next command

`break` - set a break point

`continue` – run until next break point or termination

`delete` – delete a break point

`condition` – make a break point conditional

`where` – show current function in stack trace

# An Example

```
$ gdb ./hello
GNU gdb (GDB) Red Hat Enterprise Linux
…<http://www.gnu.org/software/gdb/bugs/>
Reading symbols from
/scratch/03658/vtrue/training/petascale19/hello…done
(gdb) run
Starting program:
/scratch/03658/vtrue/training/petascale19/./hello
inside main
inside foo
```

# Cont.

```
(gdb) break main
Breakpoint 1 at 0x4005ae: file hello.c, line 6.
(gdb) run
Starting program:
/scratch/03658/vtrue/training/petascale19/./hello

Breakpoint 1, main () at hello.c:6
6    printf("inside main\n");
```

# Other Kinds of Bugs

## Floating Point Errors

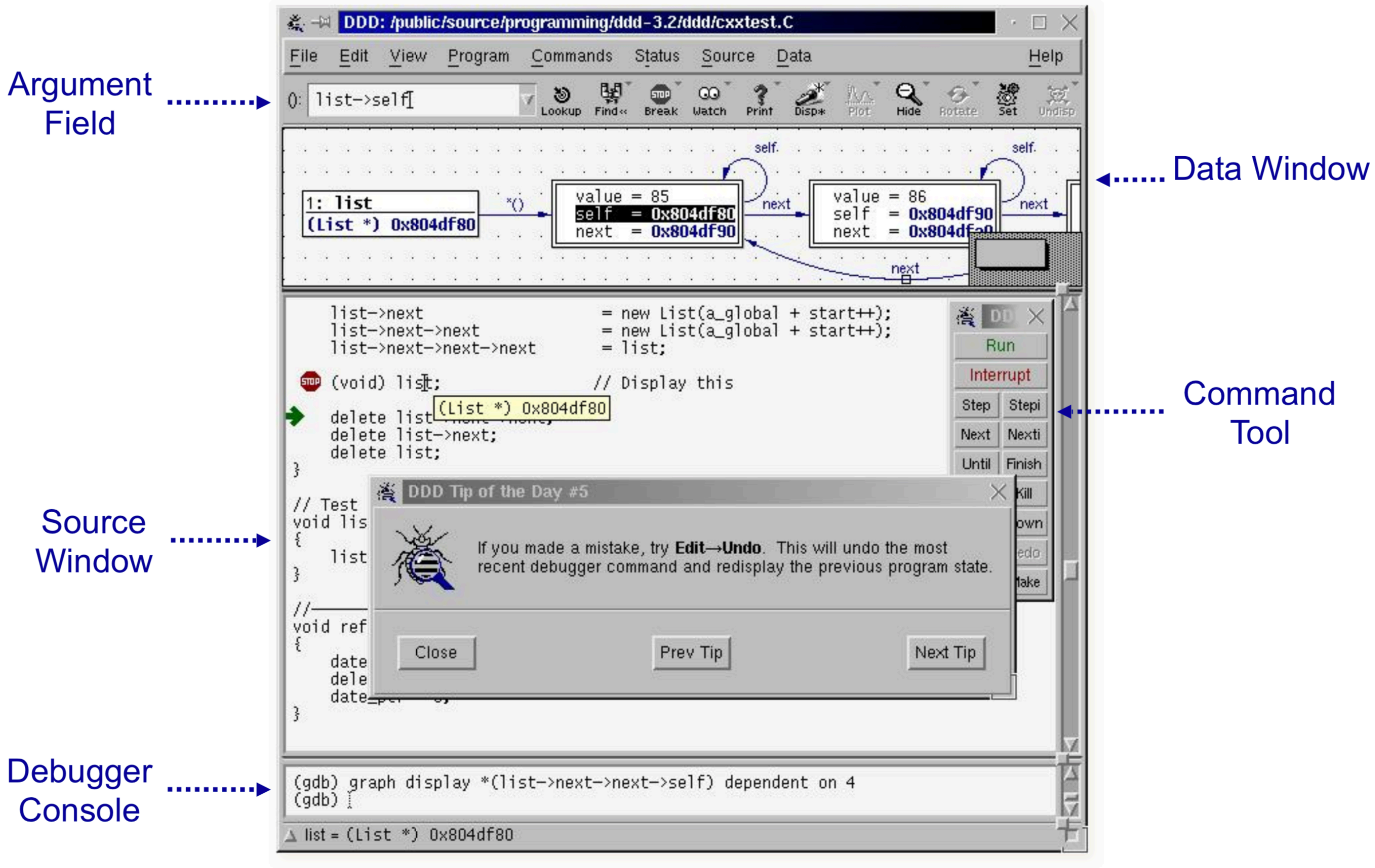
- Result in `NaN` (not-a-number) issues
- Different compilers may have different outputs
- Test for these periodically to preemptively trap a diverged solution

## Memory Errors

- Difficult to trap: use glibc, dmalloc, Electric Fence, or Valgrind to help
- These runtime checks will slow down the performance of the code
- Restrict memory checks to non-production runs

# Advanced Debuggers

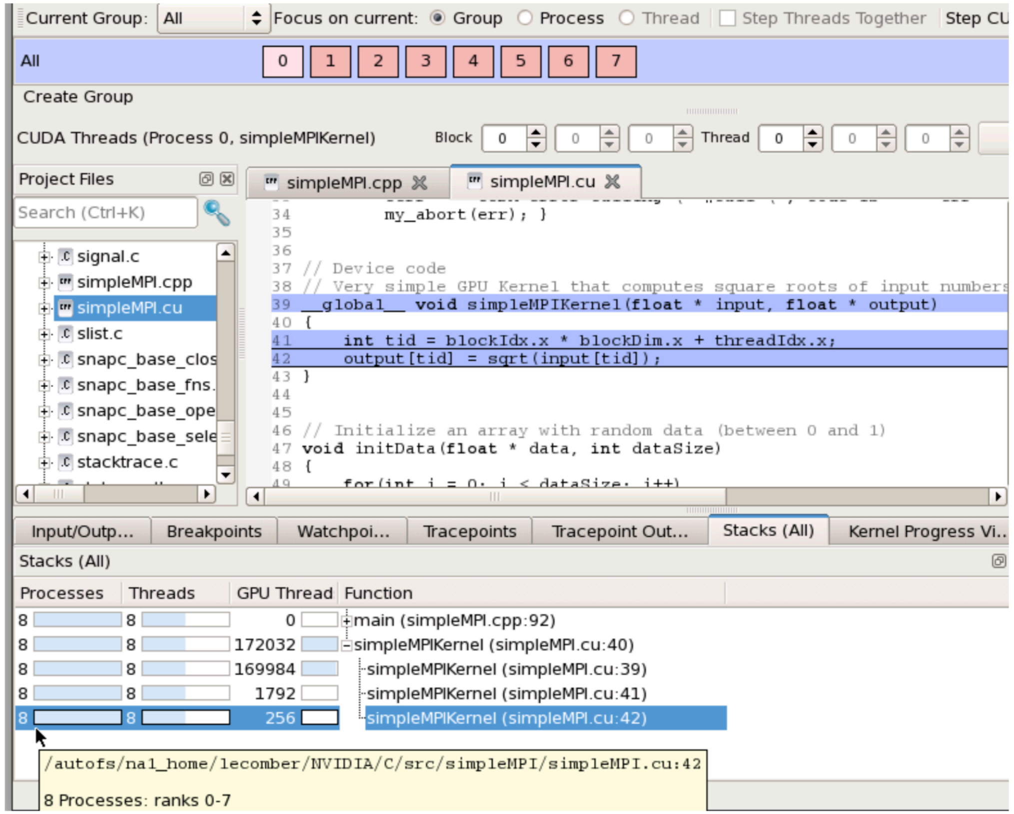
## Data Display Debugger – DDD



# Cont.

## Arm DDT

- This is a commercial debugger but is available on some systems like Stampede2
- Can interrogate over 100k MPI tasks
- Includes some basic memory debugging tools



# Resources

Vectorization - https://learn.tacc.utexas.edu/mod/page/view.php?id=48

gprof - https://web.eecs.umich.edu/~sugih/pointers/gprof_quick.html & https://sourceware.org/binutils/docs/gprof/

gdb - http://www.sourceware.org/gdb/current/onlinedocs/gdb.html

DDD - http://www.gnu.org/manual/ddd/

# License