Profiling and Debugging

Aaron Birkland
Consultant
Cornell CAC

With contributions from TACC training materials

Parallel Computing on Stampede

June 18, 2013

Introduction

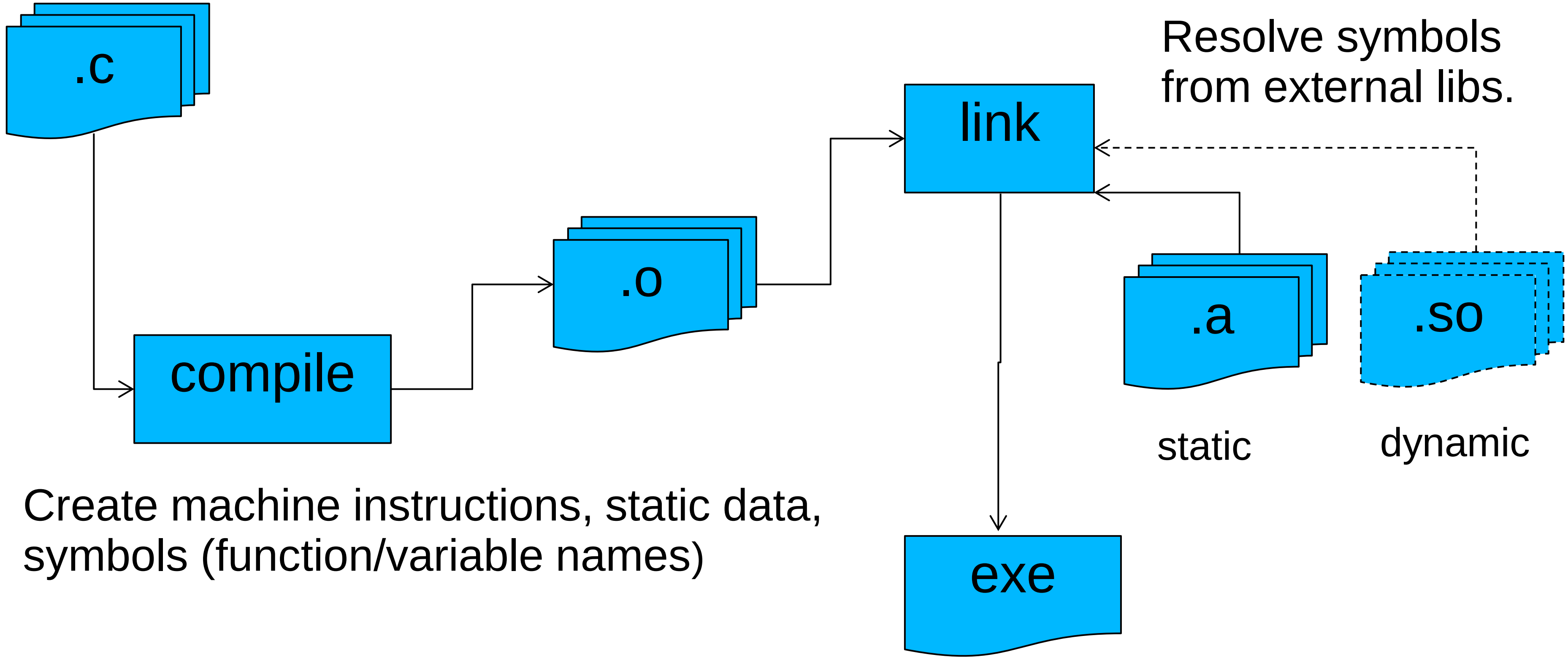
Debugging

- Find defects, analyze failures, verify expected program flow.
- Debugger tools: Inspect or modify state of running program, port-mortem analysis of memory dumps.
- Harder in parallel!

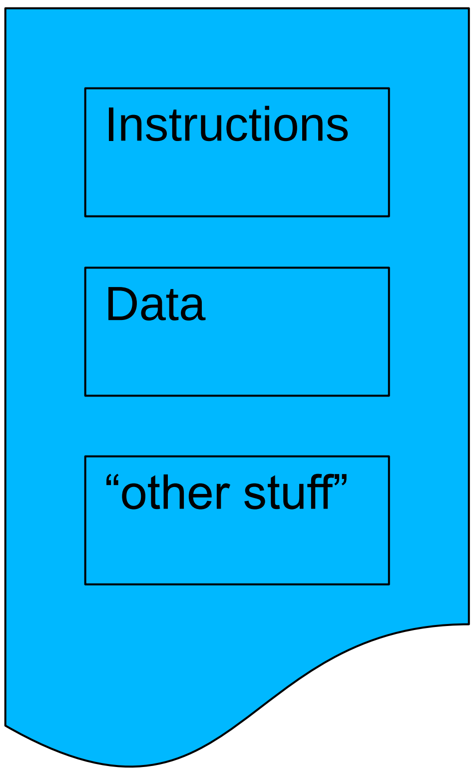
Profiling

- Measure performance characteristics, Identify areas for improvement.
- Profiler tools: collect performance measurements of a running program, analyze afterward.
- Harder in parallel!

Background: Compiling/Linking



Background: Executable Files

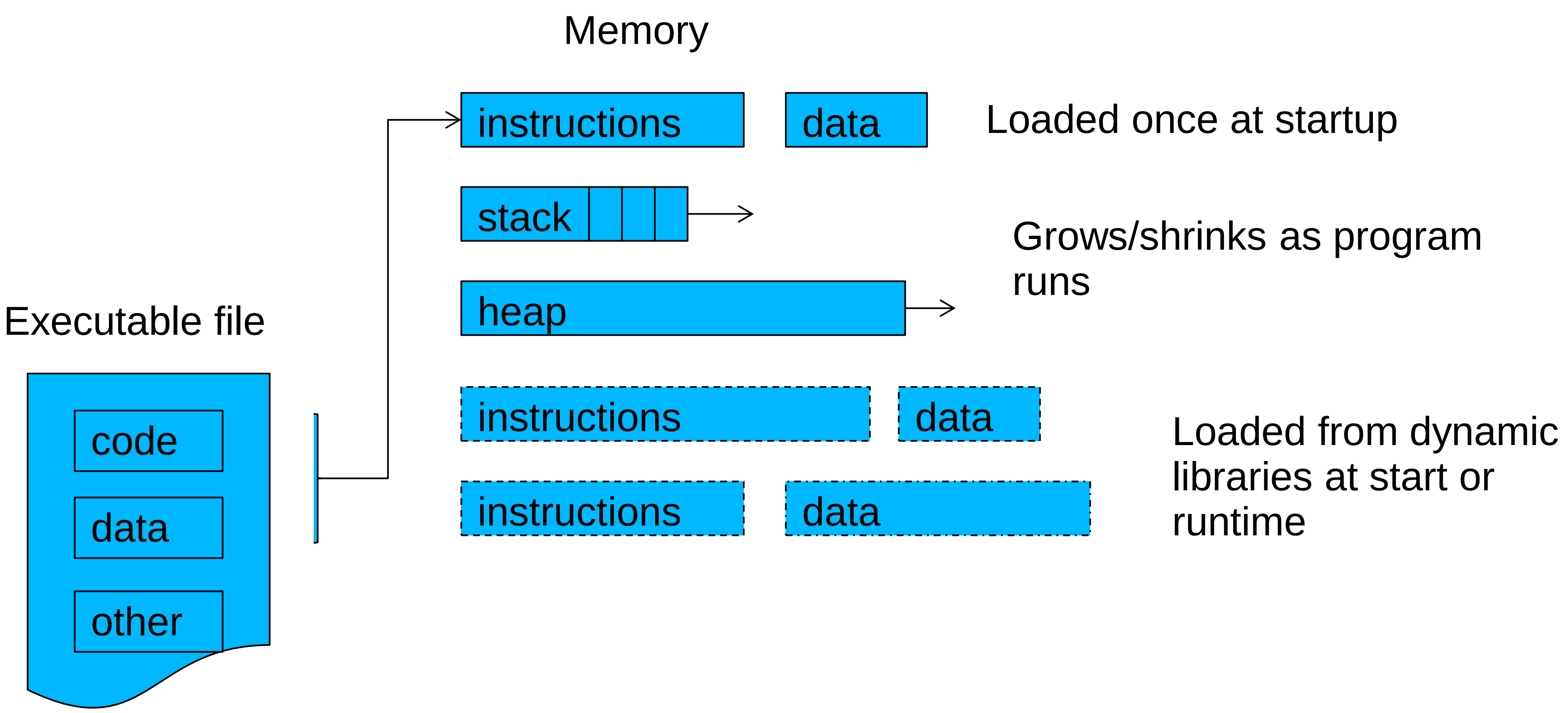


Machine instructions, memory addresses

Global and static variable data

Symbol table, linked library filenames, compiler version, other metadata.

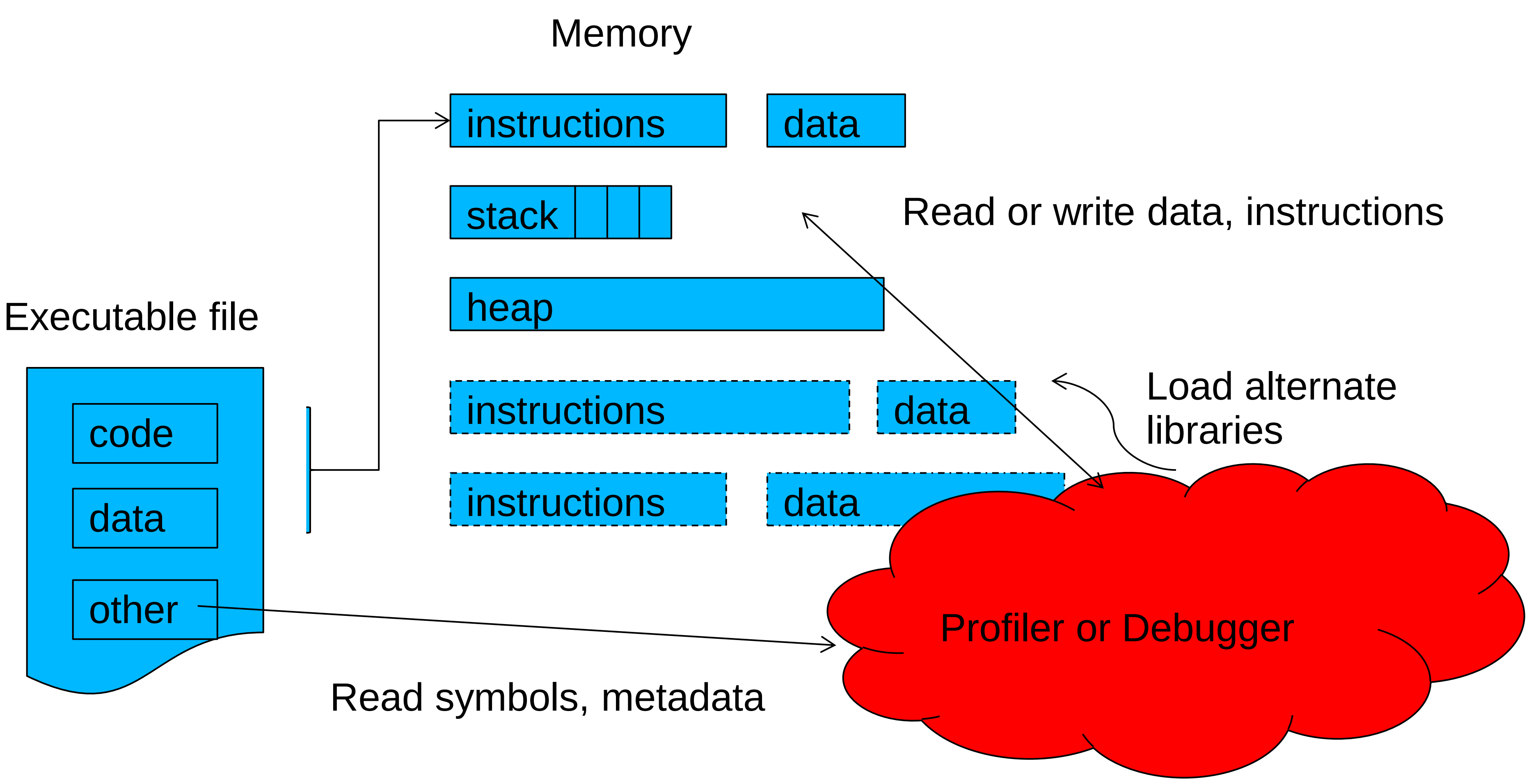
Background: Execution & Memory



Background: OS and Hardware

- OS can provide API for inspecting and controlling process execution
- Wrap a program at startup or attach to running process
- Example: Linux `ptrace()`
  - Pause execution
  - Modify in-memory instructions
  - Inspect or modify data memory or registers
  - Catch signals and traps
- CPU can provide hardware counters
  - Cache hits/misses, TLB hits/misses, FLOPs, etc

Background: Profilers and Debuggers in control



Debugging

- Inspect program state, compare to one's own assumptions and expectations
  - Step through code line by line
  - Inspect variables/memory at specific points
  - Inspect memory and call stack after a crash
- For MPI, OpenMP 'state' gets more complex
  - Many remote processes with own memory
  - Message status and timing
  - Step through individual processes or thread independent of rest (while others may still be running!)

Debugging: printf and logging

```
int main (int argc, char** argv) {
        printf("Starting main…");
        int iterations = 5;
        int val = 0, val2=0;
        printf("Initialized val to %d and val2 to %d", val, val2);
        while (iterations --) {
                val = sometime();
                print("Sometime() returned %d\n", val);
                val2 = moretime();
                printf("moretime() returned %d\n", val);
        }
        printf("Exiting main, iterations ==%s\d", iterations);
```

Debugging: printf and logging

- Easy and intuitive
  - Target specific sections of code, under specific conditions
  - Simply analyze log(s) after execution, even for parallel or multithreaded jobs
  - Great for rare/transient or timing related bugs
- Invasive and messy
  - Need to re-compile when logging statement added/removed
  - Can slow down execution
  - Easy to forget statements are there
  - Can be hard to correlate output with statements.
  - Jumbled output with threads printing simultaneously

Debugging: printf and logging

- Logging frameworks an improvement over printf (e.g. Log4c)
  - Filter by log levels (WARN, INFO, DEBUG)
  - Timestamps, formatting, runtime configuration changes
  - Control over where/how log is written (console, large file, rolling file, remote server, database, etc)

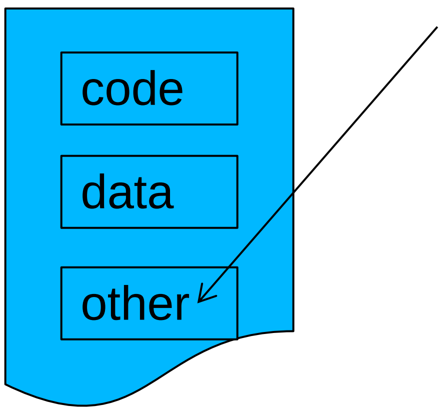
Debugging: printf and logging

```
int main (int argc, char** argv) {
        log4c_init();
        mycat = log4c_category_get("sillyapp.main");
        int iterations = 5;
        log4c_category_log(mycat, LOG4C_PRIORITY_DEBUG,"Debugging app 1 - loop %d", iterations);
        int val = 0, val2=0;
        log4c_category_log(mycat, LOG4C_PRIORITY_ERROR, "Some error"
        printf("Initialized val to %d and val2 to %d", val, val2);
        …
```

```
[Header]
2009-05-13 15:21:14,315 [11] WARN  Logger.Program Pretty sure I'm getting ready to die!
2009-05-13 15:21:14,331 [11] ERROR Logger.Program uh-oh, no I wasn't!
2009-05-13 15:21:14,331 [11] FATAL Logger.Program blech.  Out
[Footer]
```

Debugging: symbolic debugging

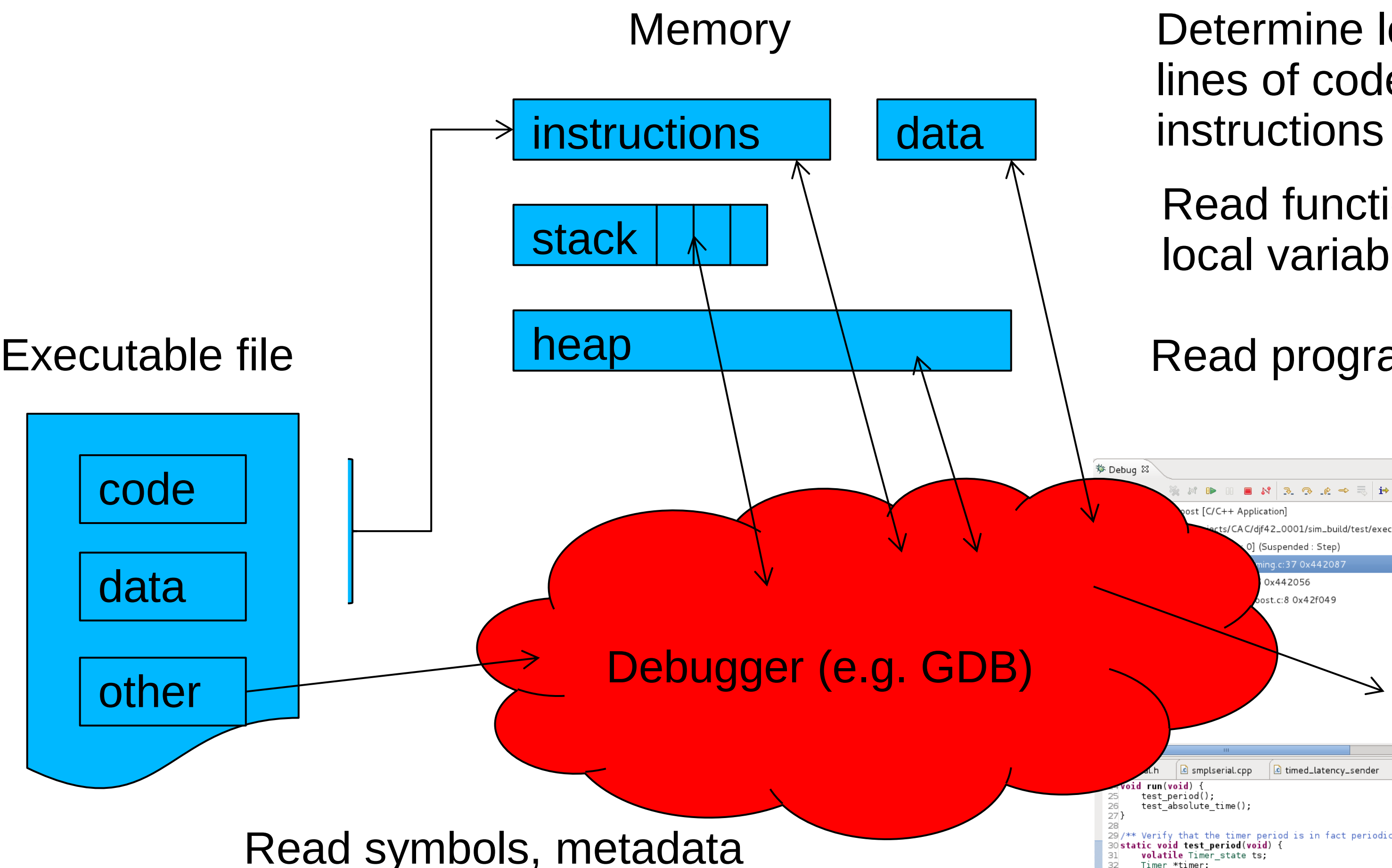
- Inspect process memory, correlate instructions & memory addresses with *symbols* from source code.
- Compiler option (`-g` for gcc, intel) tells compiler to store debugging symbols in the executable file



Human-readable symbols and correlation data stored in one of the “other” segments in an executable file.

- Not loaded into memory (no runtime overhead)
  - Some compilers MAY disable some optimizations
- Available for inspection by debugging tool
- Provides a very useful “map” for inspecting core dumps

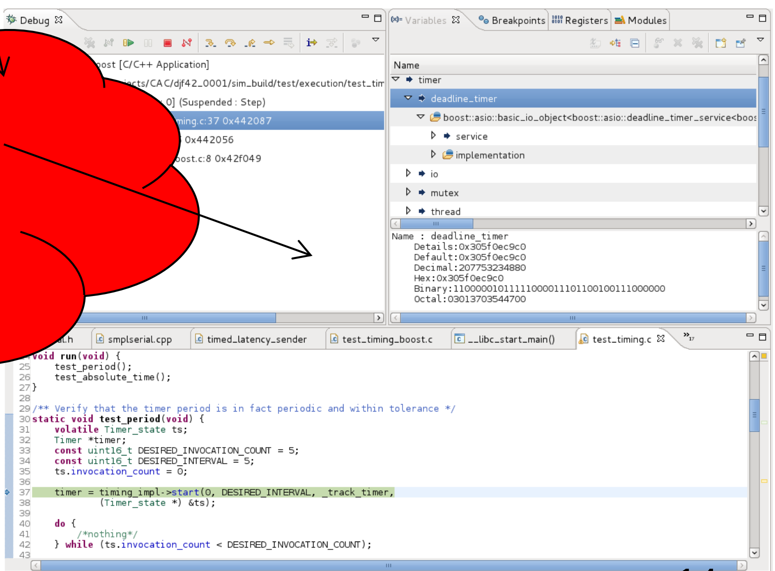
Debugging: symbolic debugging: serial, threaded



Determine location, step through lines of code by manipulating instructions

Read function call sequence and local variables from stack

Read program data from heap



Debugging: symbolic debugging: serial, threaded

- GDB (Gnu, almost ubiquitous), IDB (Intel)
  - Launch a program, analyze a dump, or attach to running process
  - Set conditional breakpoints, start/stop execution at will
  - Inspect and modify variables

Launch a process: `gdb <executable>`

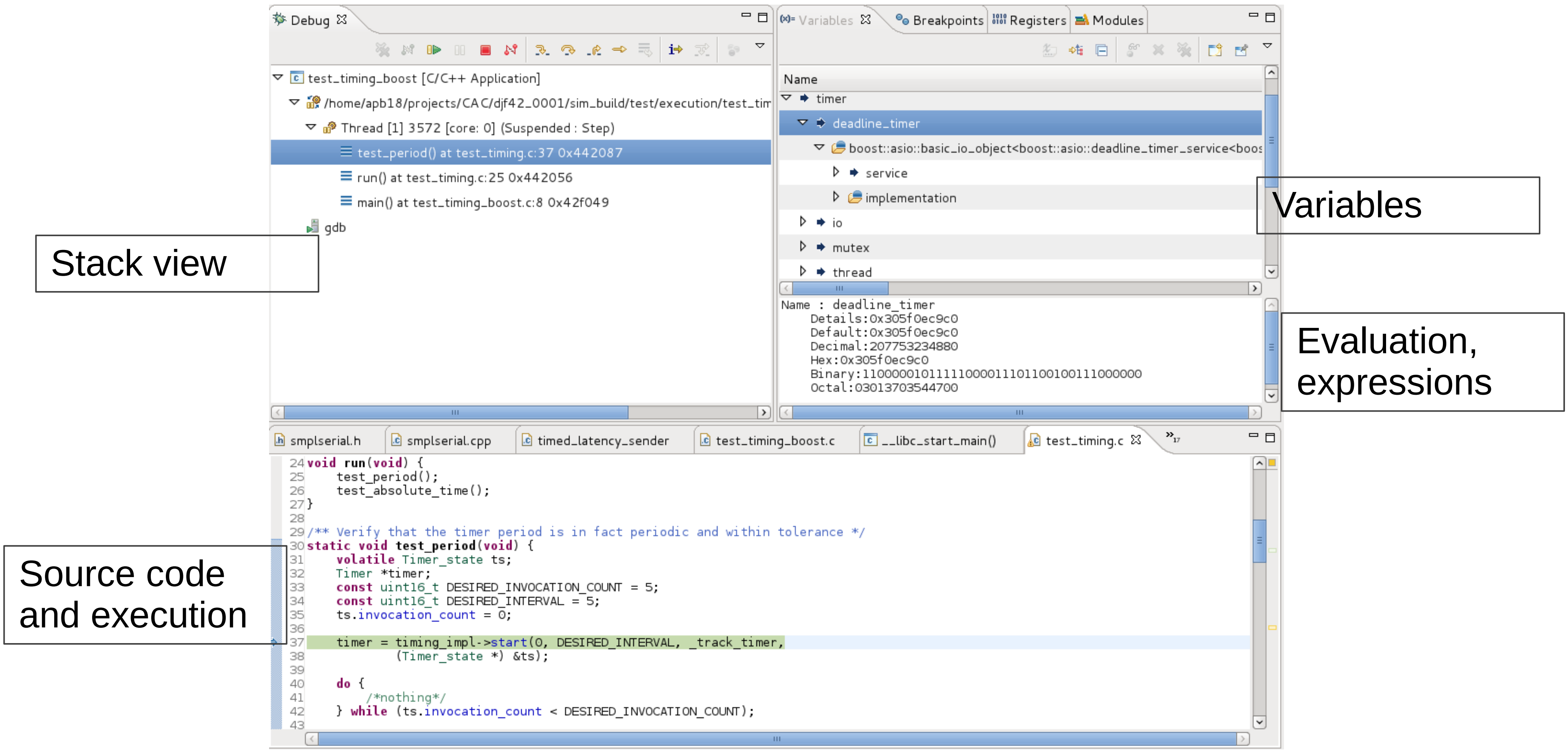
Attach to process: `gdb <executable> 1234`

Analyze a dump: `gdb <executable> core.1234`
(check ulimit setting for max core file size!)

Debugging: symbolic debugging: GDB

- run – execute the program from beginning.
- backtrace – produce the backtrace from the last fault
- break <line number> or break <function-name> - break at the line number or at the use of the function
- step – step to next line of code (step into function if possible)
- next – step to next line of code (do not step into function)
- print <variable name> - print the value stored by the variable
- continue – run until next break point

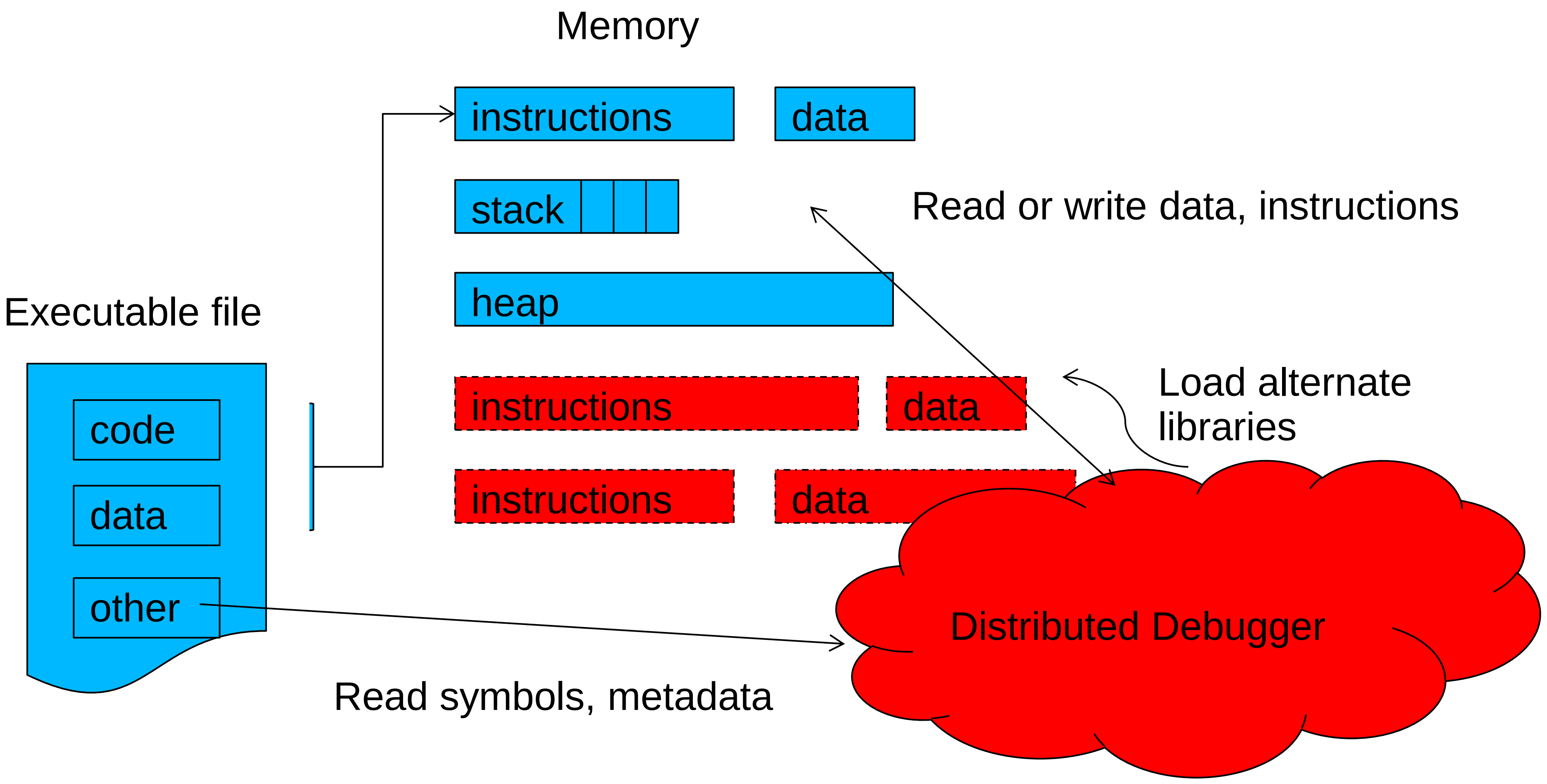
Debugging: symbolic debugging



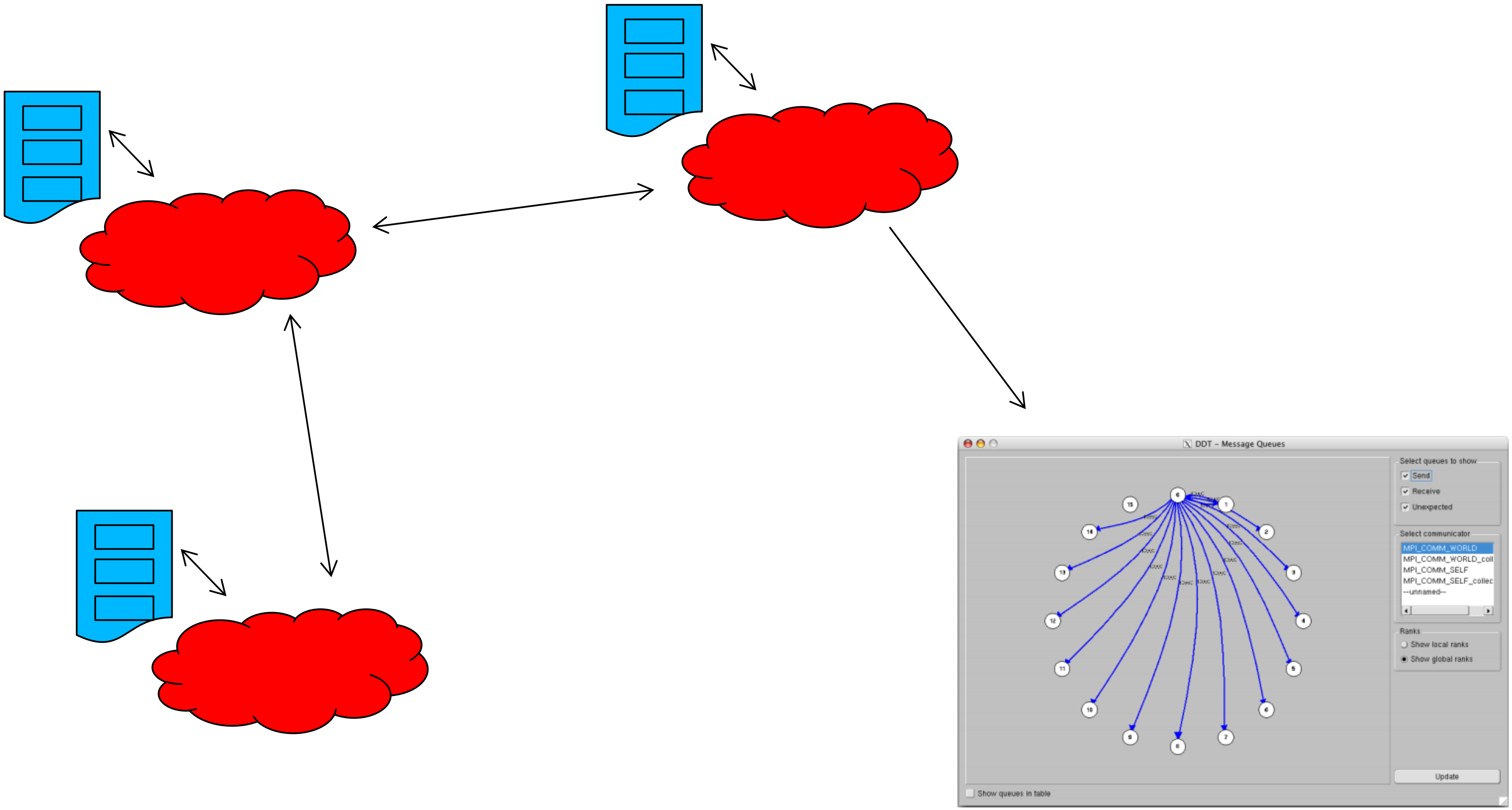
Debugging: symbolic debugging: Optimized code

- Aggressive optimizations (e.g. `-O3`) cause machine instructions to diverge from machine code!
  - Loop unrolling, function inlining, instruction re-ordering, optimizing out variables, etc
- Effects: debugger much less predictable
  - Setting some breakpoints are impossible (instructions optimized out or moved)
  - Variables are optimized out, or appear to change unexpectedly
  - Stepping through code follows arbitrary execution order
- Easiest to debug with NO optimizations (`-O0`)

Debugging: symbolic debugging: Distributed



Debugging: symbolic debugging: Distributed



Debugging: symbolic debugging: distributed: DDT

- DDT (Allinea Distributed Debugger Tool)
- Proprietary, GUI-oriented
- Large-scale OpenMP, MPI debugging
  - MPI message tracking
  - View queues and communication patterns for running procs
  - Supports all MPI distributions on Ranger
- Jobs submitted through DDT
  - Remember, it needs to “wrap” and control each task
- Usage: Compile with `–g`, then `module load ddt`, then `ddt <executable>` and go from there.

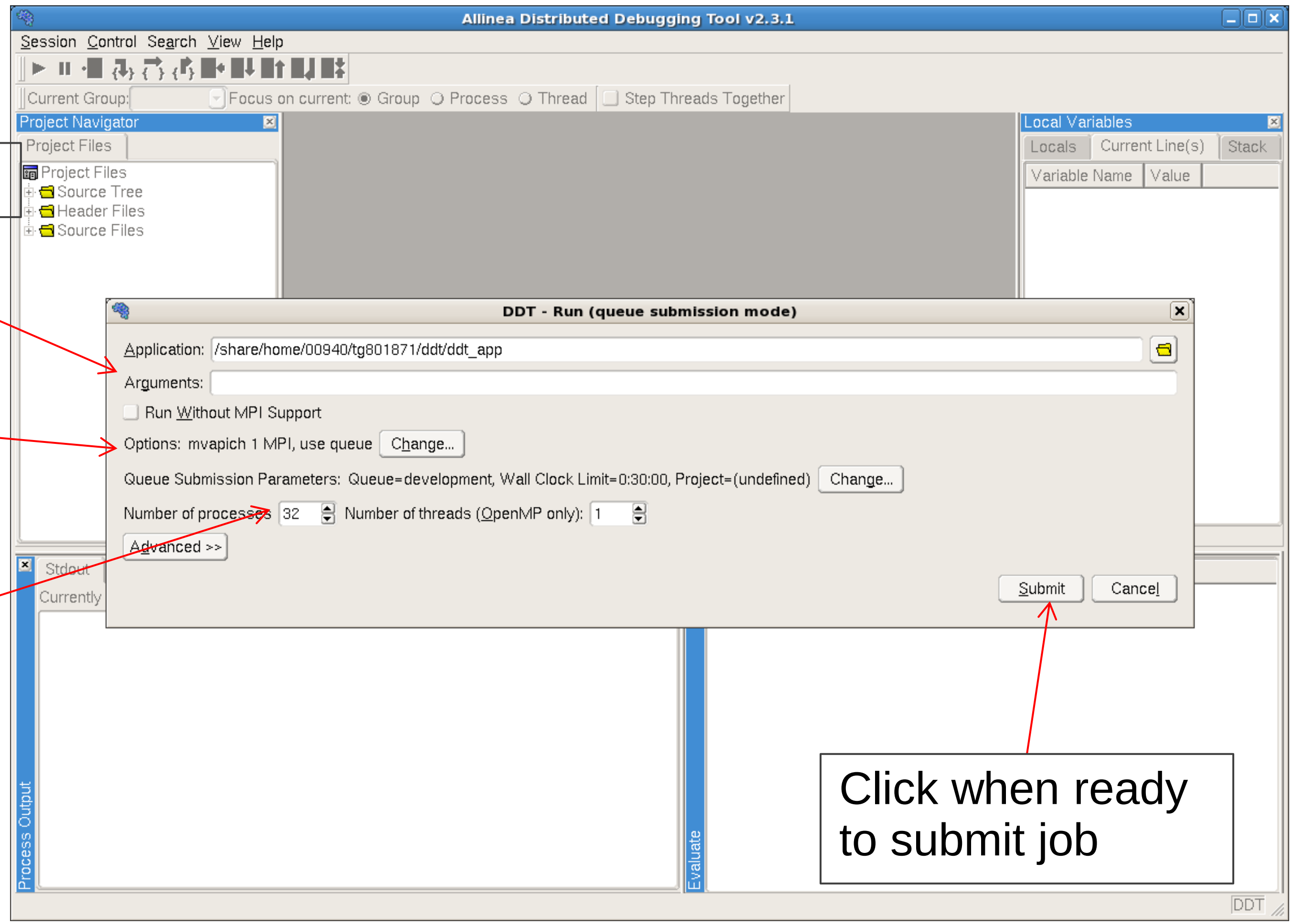
- Need local X server (ssh –X), or use vnc on

Debugging: symbolic debugging: distributed: DDT

Add any arguments

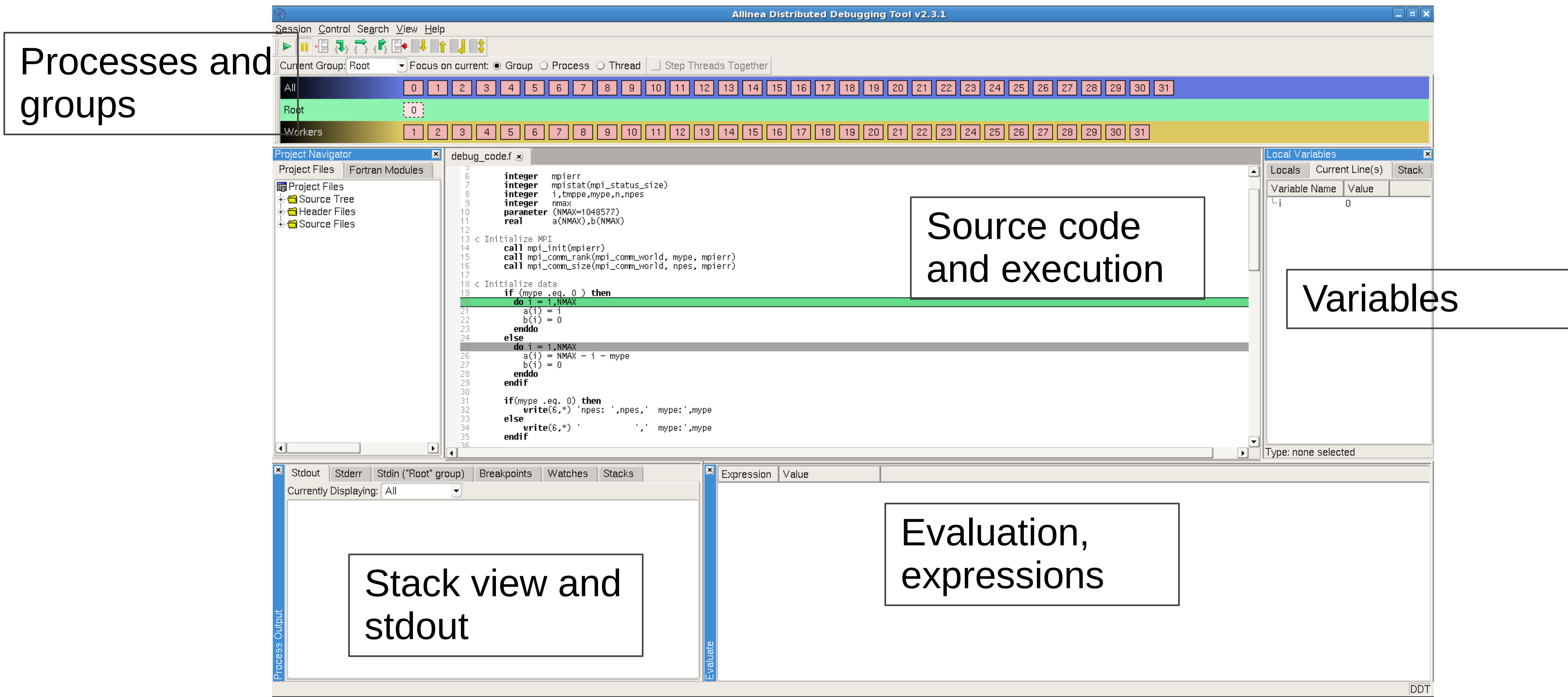
Stampede default

Sets number of nodes

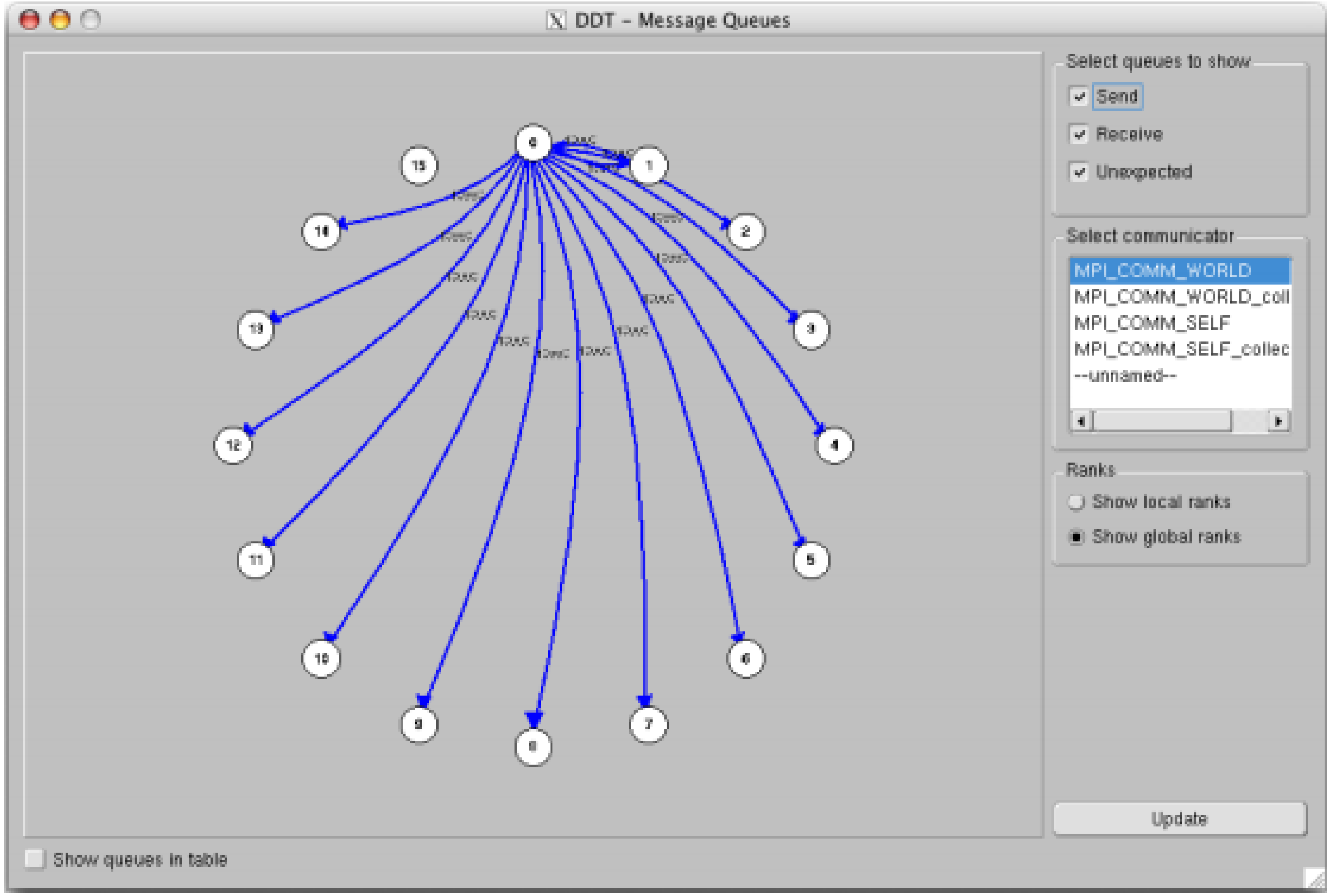


Click when ready to submit job

Debugging: symbolic debugging: distributed: DDT



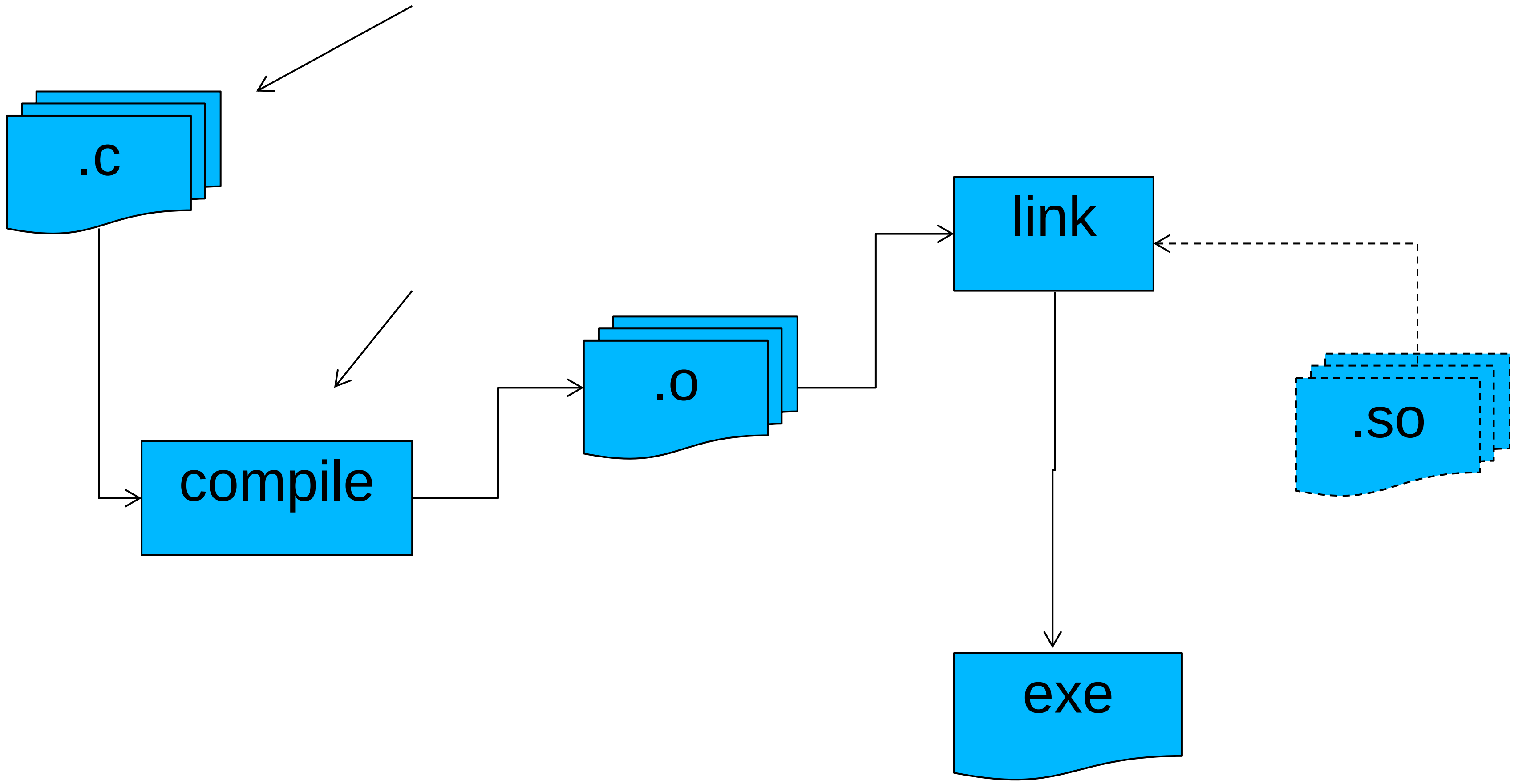
Debugging: symbolic debugging: distributed: DDT



Profiling

- Measure performance characteristics, identify compute-intensive areas (e.g. “hot spots”) that *may* be worth improving
- Can suffer from “observer effect” – collecting performance data significantly degrades performance
- Two main approaches: instrumentation and statistical sampling
  - Instrumentation: add instructions to collect information (function call duration, number of invocations, etc)
  - Sampling: Query state of unmodified executable at regular intervals

Profiling: Instrumentation



Profiling: Instrumentation: printf and timers

- Check system time and printf at appropriate points
  - SYSTEM_CLOCK or clock() for fortran, C
- Very simple, great for targeting a specific area.
- Problem: printf statements are expensive, especially if there are many
- Problem: Timer precision and accuracy is system/implementation dependent.

Profiling: Instrumentation: GPROF

- GPROF (GNU profiler)
- Compile option `-pg` adds debugging symbols and additional data collection symbols
  - Slows program down, sometimes significantly
- Each time program is run, output file `gmon.out` is created containing profiling data
  - This data is then analyzed by gprof in a separate step, e.g. gprof <executable> gmon.out > profile.txt

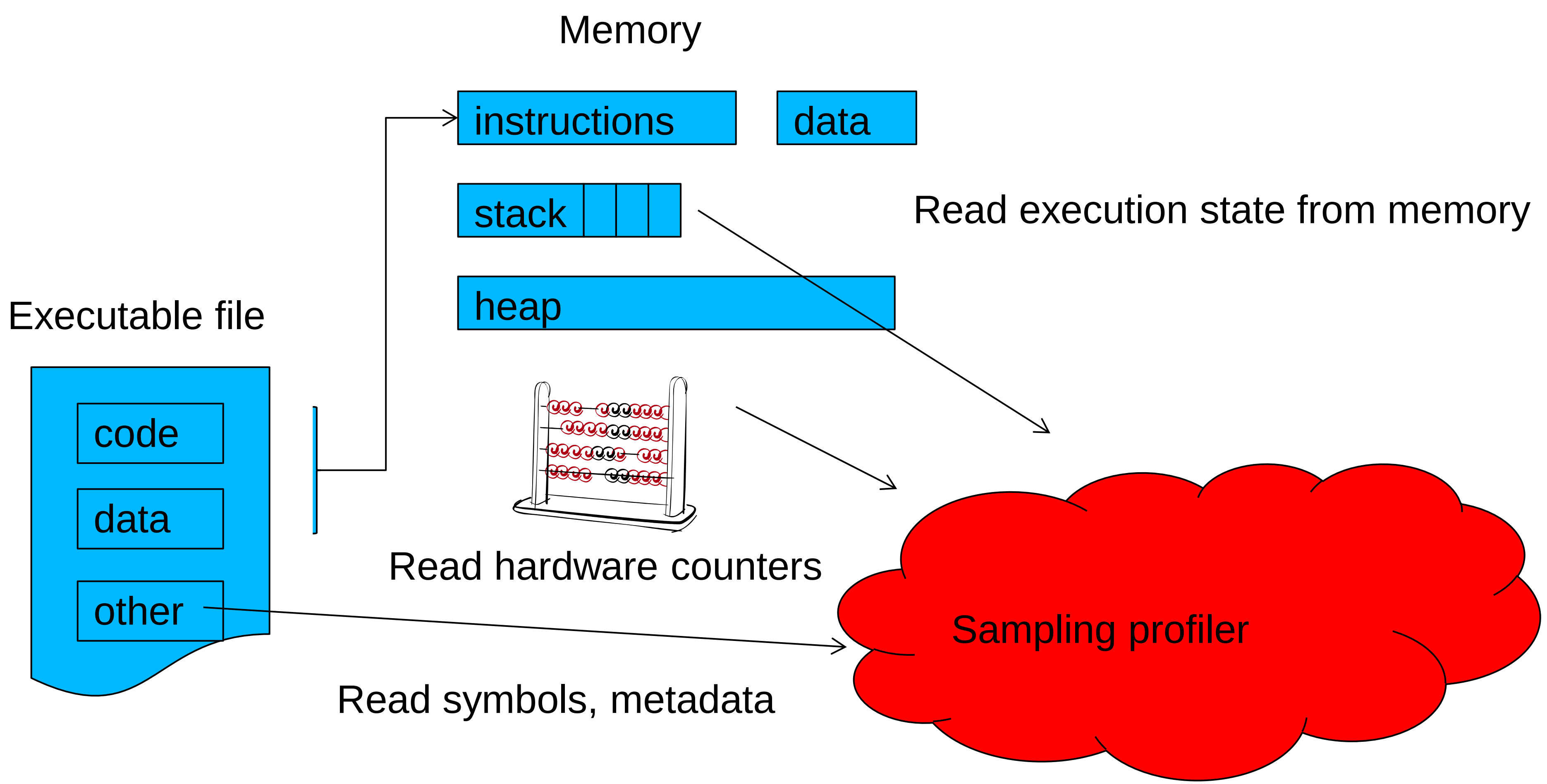
Profiling: Instrumentation: GPROF

- Flat profile
  - Lists each function with associated statistics
  - CPU time spend, number of times called, etc
  - Useful to identify expensive routines
- Call Graph
  - Number of times function was called by another, called others
  - Gives a sense of relationship between functions
- Annotated Source
  - Number of times a line was executed

Profiling: Instrumentation: TAU

- Specialized in multithreaded and/or MPI applications
- Compile with special wrappers
  - `tau_cc.sh, tau_f90.sh`
- Set environment variables to gather certain statistics
  - `export COUNTER1=GET_TIME_OF_DAY`
  - `export COUNTER2=PAPI_FP_OPS`
- Text UI `pprof`
- GUI via `paraprof`
- Integrates with (i.e. can access data from) sampling libraries such as PAPI
- Can also perform statistical sampling via `tau_exec`

Profiling: sampling



Profiling: sampling: HPCToolkit, PAPI

- PAPI: Provides access to hardware counters
  - API hides gory details of hardware/OS platform
  - Cache accesses, hits, misses
  - FLOPS
  - The kinds of data available depend very much on hardware
- HPCToolkit
  - Asynchronous sampling of running processes
  - Supports OpenMP, MPI, and hybrid
  - Supports running against optimized code
  - http://hpctoolkit.org

Profiling: sampling: PerfExpert

- Developed at TACC
- Easy to use interface over data collected via HPCToolkit and PAPI
- Provides suggestions and “what to fix”
- Runs against fully optimized code with debugging symbols
- http://www.tacc.utexas.edu/perfexpert

- Profile with `perfexpert_run_exp`, creates results file experiment.xml
- View results with `perfexpert <threshold> experiment.xml`
- Get recommendations with `perfexpert -r <threshold> experiment.xml`

Profiling: sampling: PerfExpert

```
Loop in function main() at Integrator.c:81  (98.9% of the total runtime)
===============================================================================
ratio to total instrns      %  0.........25..........50.........75........100
   - floating point      :   30 **************
   - data accesses       :   71 **********************************
* GFLOPS (% max)         :    1 *
-------------------------------------------------------------------------------
performance assessment    LCPI good......okay......fair......poor......bad....
* overall                :  4.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
upper bound estimates
* data accesses          : 33.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits            :  2.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
   - L2d hits            :  2.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L2d misses          : 28.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses   :  0.4 >>>>>>>>
   - L1i hits            :  0.4 >>>>>>>>
   - L2i hits            :  0.0 >
   - L2i misses          :  0.0 >
* data TLB               :  0.0 >
* instruction TLB        :  0.0 >
* branch instructions    :  0.1 >>
   - correctly predicted:  0.1 >>
   - mispredicted        :  0.0 >
* floating-point instr   :  1.1 >>>>>>>>>>>>>>>>>>>>>>>
   - fast FP instr       :  1.1 >>>>>>>>>>>>>>>>>>>>>>>
   - slow FP instr       :  0.0 >
```

overall loop performance is bad

biggest problem is data accesses that miss in the L2 cache

remaining performance categories are good

Profiling: sampling: PerfExpert

```
Code Section: Loop in function main() at Integrator.c:81  (98.9% of the total runtime)
==========================================================================================
change the order of loops
 loop i { loop j {...} } → loop j { loop i {...} }
------------------------------------------------------------------------------------------
 employ loop blocking
 loop i {loop k {loop j {c[i][j] = c[i][j] + a[i][k] * b[k][j];}}} →
 loop k step s {loop j step s {loop i {
   for (kk = k; kk < k + s; kk++) {
     for (jj = j; jj < j + s; jj++) {
       c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];}}}}}
------------------------------------------------------------------------------------------
 apply loop fission so every loop accesses just a couple of different arrays
 loop i {a[i] = a[i] * b[i] - c[i];} →
 loop i {a[i] = a[i] * b[i];} loop i {a[i] = a[i] - c[i];}
```