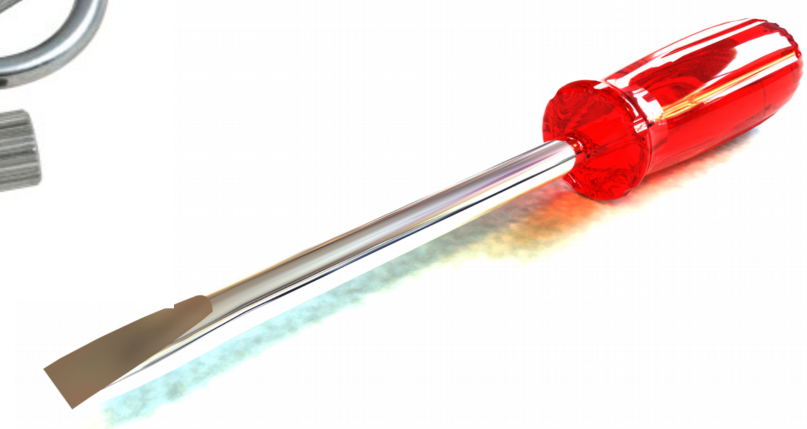
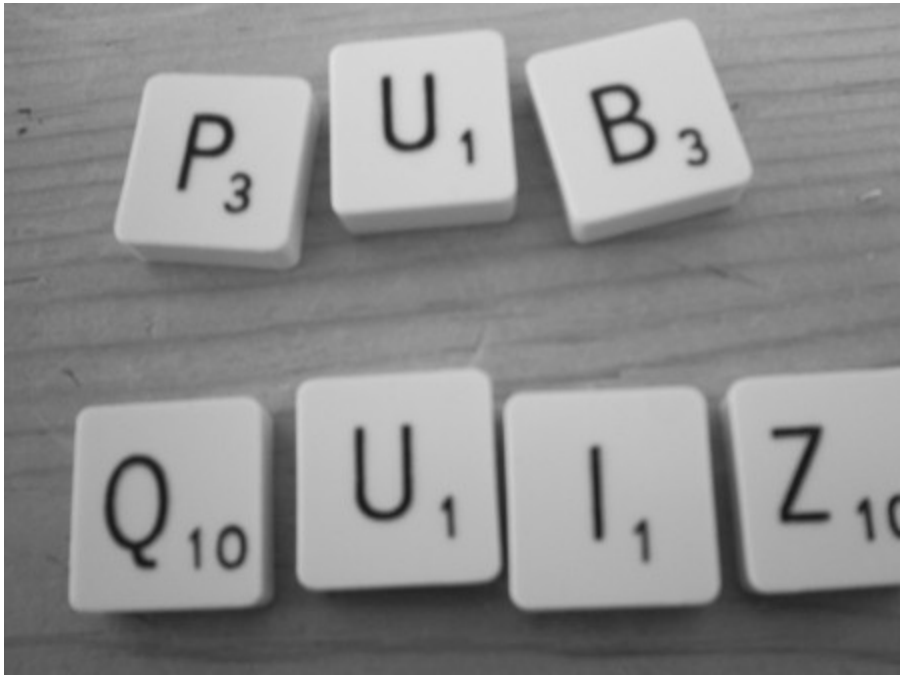
Tools



University of BRISTOL

Quiz Question

- What tool has been found—**empircally**—to be the most effective, both for **performance** and **debugging**?



University of BRISTOL

Answer

Pencil & paper

No, seriously!
and not because I'm
an old fuddy-duddy..

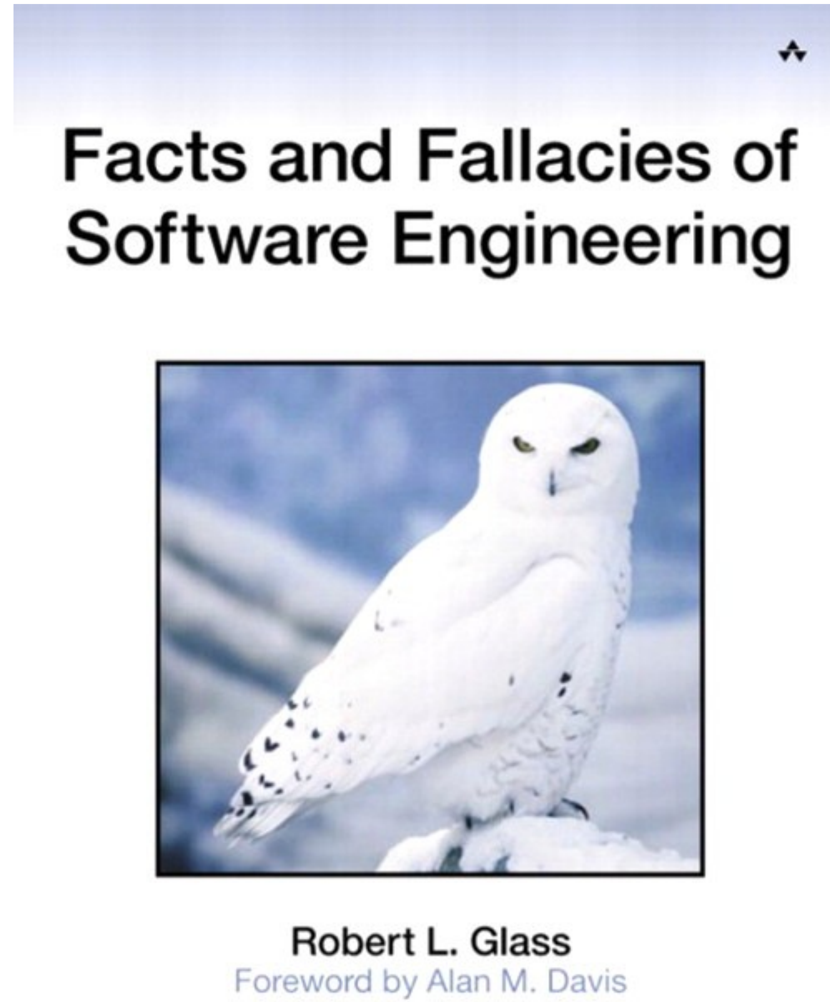
From 55, Empirically Derived Facts:

#36 Programmer-created, built-in debug code is an important supplement to testing tools.

#37 Rigorous inspections can Remove up to 90% of errors before the first test case is run.
(Review code in max 1hr chunks.)

#38 Rigorous inspections should not Replace testing.

#52 Efficiency stems more from good Design than good coding.

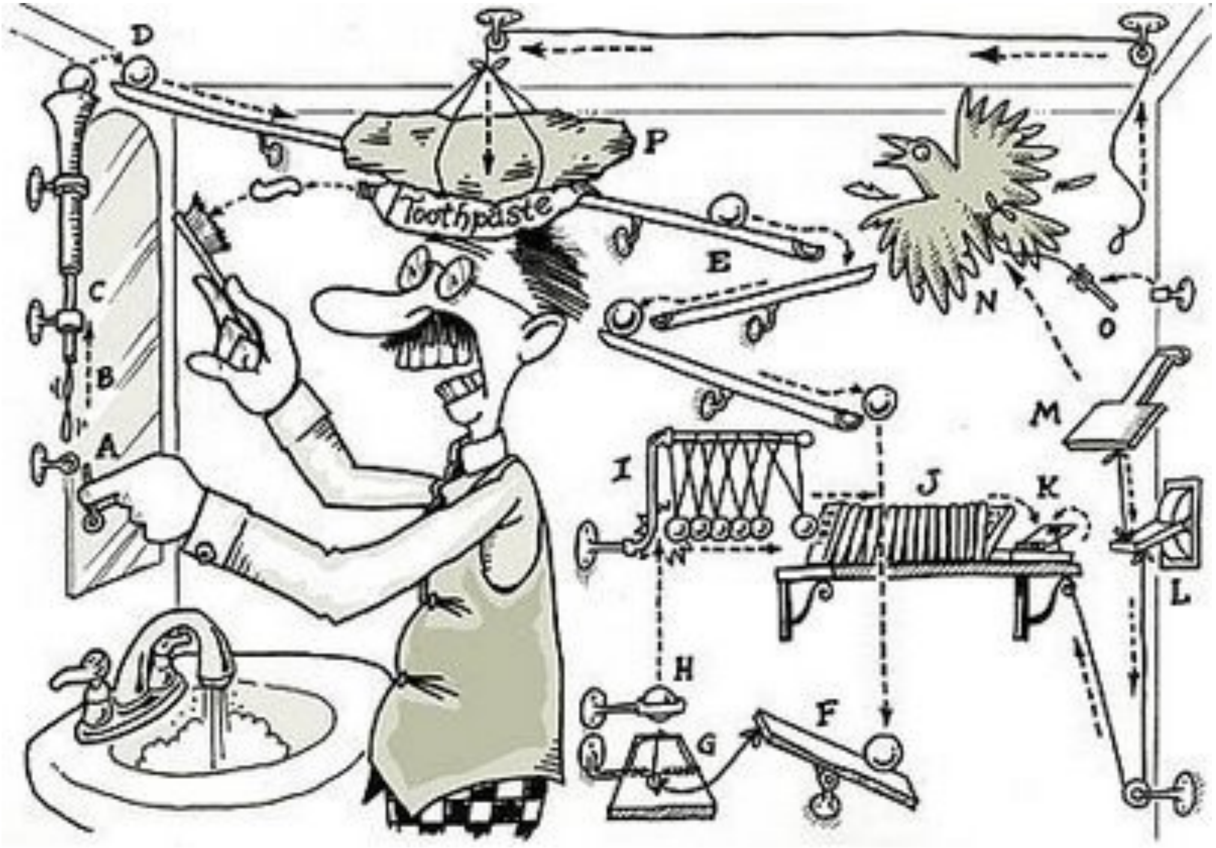


University of BRISTOL

Overview

- Design & Implementation
  - Don't go for a big bang!
- Profiling
  - Compiler reports.
  - Timing calls.
  - A number of good tools (several are open-source):
    - **`gprof, valgrind, tau, VTune, PAPI`**
- Debugging
  - Many bugs will be serial bugs, but some are of a new breed.
  - Compiler flags can help.
  - What tools do we do have?:
    - **`-wall, -traceback, gdb, electric fence, lint,`** ~~**`DDT`**~~

The Need for Design!



How often do we just open an editor and start typing?..
..and end up with something resembling the above?
Thank you, Rube Goldberg.

University of BRISTOL

Approaches to Implementation

- You've got deadlines..
- Other course work..
- Hobbies that keep you sane..
- A social life!..

...

Approaches to Implementation

When pushed for time it's tempting to go for...

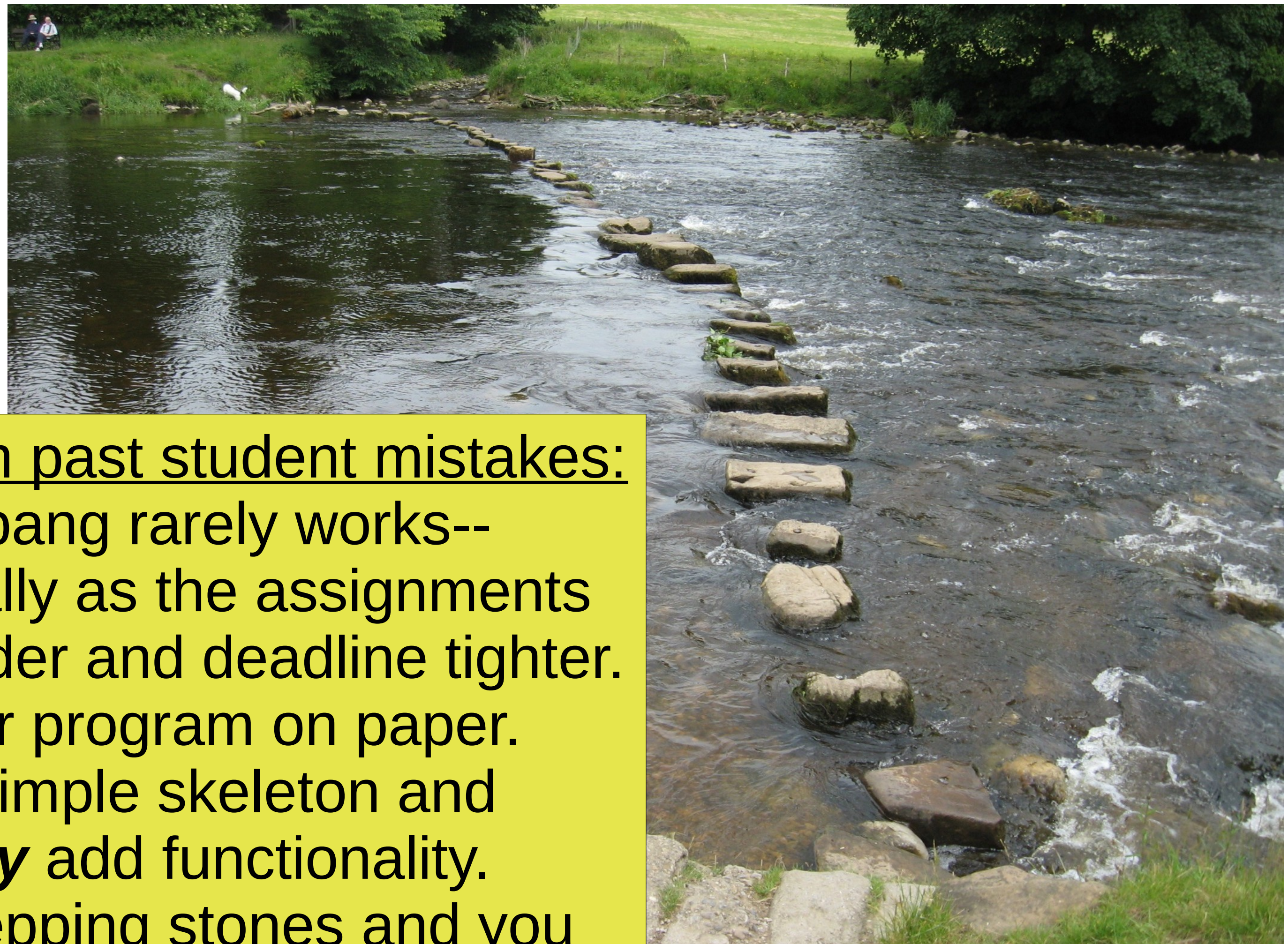
..the big bang approach

University of BRISTOL

Approaches to Implementation

Don't do it!

Approaches to Implementation



<u>Learn from past student mistakes:</u>

- The big bang rarely works-- especially as the assignments get harder and deadline tighter.
- Plan your program on paper.
- Write a simple skeleton and ***gradually*** add functionality.
- Think stepping stones and you ***will*** get there in the end.

University of BRISTOL

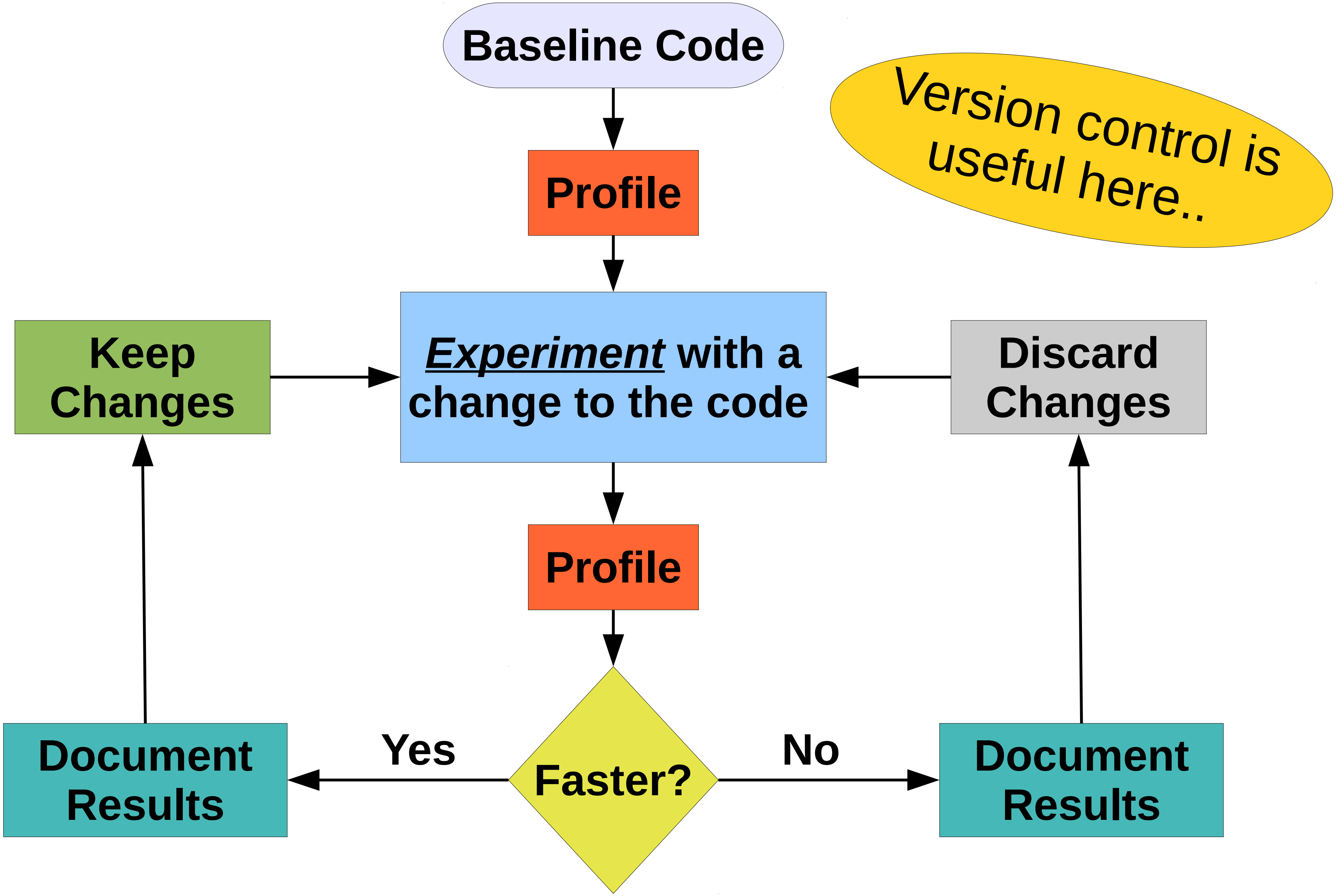
Approaches to Implementation

- Also consider using supporting tools such as:
  - Version control
    - Subversion, git etc.
  - Build systems:
    - Make, CMake, Ant, SCons etc.

Profiling

Profiling – Setting the Scene

- Code can ***always*** be improved.
- It makes sense to speed-up the portions of your program that take the most time (recall Knuth).
- Examples could be one spectacularly inefficient region of code, or a routine which is not bad, but is called many, many times.
- Optimising compilers are surprisingly good, so you will not be rewarded by trying to ***guess*** at which parts need improving.



University of BRISTOL

Compiler Reports

```
gcc -lm -Wall -O3 -ffast-math -ftree-vectorizer-verbose=2 d2q9-bgk.c -o d2q9-bgk.exe
d2q9-bgk.c:194: note: vectorized 0 loops in function.
...
d2q9-bgk.c:598: note: not vectorized: complicated access pattern.
...
d2q9-bgk.c:669: note: not vectorized: unhandled data-ref
...
```

GCC v4.4.7

https://gcc.gnu.org/projects/tree-ssa/vectorization.html

University of BRISTOL

Compiler Reports

```
icc -O3 -xHOST -vec-report=2 d2q9-bgk.c -o d2q9-bgk.exe
d2q9-bgk.c(150): (col. 3) remark: loop was not vectorized: vectorization possible but seems inefficient.
d2q9-bgk.c(150): (col. 3) remark: OUTER LOOP WAS VECTORIZED.
d2q9-bgk.c(150): (col. 3) remark: loop was not vectorized: not inner loop.
d2q9-bgk.c(150): (col. 3) remark: LOOP WAS VECTORIZED.
d2q9-bgk.c(150): (col. 3) remark: loop was not vectorized: not inner loop.
d2q9-bgk.c(150): (col. 3) remark: loop was not vectorized: not inner loop.
d2q9-bgk.c(150): (col. 3) remark: loop was not vectorized: unsupported loop structure.
```

e.g. icc 13.0.1

https://software.intel.com/en-us/articles/vectorization-and-optimization-reports

Compiler Reports

```
icc -O3 -xHOST -qopt-report d2q9-bgk.c -o d2q9-bgk.exe
LOOP BEGIN at d2q9-bgk.c(457,3) inlined into d2q9-bgk.c(150,3)
  remark #15542: loop was not vectorized: inner loop was already vectorized
  LOOP BEGIN at d2q9-bgk.c(458,5) inlined into d2q9-bgk.c(150,3)
    remark #15542: loop was not vectorized: inner loop was already vectorized
    LOOP BEGIN at d2q9-bgk.c(467,9) inlined into d2q9-bgk.c(150,3)
      remark #15301: MATERIALIZED LOOP WAS VECTORIZED
    LOOP END
  LOOP END
  LOOP BEGIN at d2q9-bgk.c(458,5) inlined into d2q9-bgk.c(150,3)
  <Remainder>
  LOOP END
LOOP END
```

e.g. icc 16.0.0

University of BRISTOL

Adding Timing Code

```
clock_t tic, toc;
tic = clock();
 ...
toc = clock();
printf("CPU time: %f\n", ((float)toc -(float)tic)/
(float)CLOCKS_PER_SEC);
```

- Beware! Clock() returns CPU time and ***not*** wall-clock time.
- Timing a parallel loop will return the ***total for all threads***.

```
time_t ttic, ttoc;
ttic = time(NULL);
 ...
ttoc = time(NULL);
printf("elapsed time (s): %d\n",
(int)difftime(ttoc,ttic)); /* NB nearest second */
```

See also: `gettimeofday()`, getrusage(), omp_get_wtime(), `MPI_Wtime()`

University of BRISTOL

Profiling Threaded Code

- **`gprof`** is unreliable (can only profile one thread in any case)
- **`valgrind`** is very slow and only emulates your code using a single cpu core.
- **`Tau`** is available on the cluster and works very well for both threaded code and MPI jobs...
  - http://www.cs.uoregon.edu/research/tau/home.php
  - Tuning and Analysis Utilities (TAU)
- Intel's **`VTune`** is also available on BCp3 ***but licenses are limited***.

Using Tau

- For OpenMP code:
  - **`module add tools/gnu_builds/tau-2.23.1-openmp`**
- To instrument some C code, replace **`gcc`** with a call to **`tau_cc.sh`**
  - e.g. in a Makefile: **`CC=tau_cc.sh`**

Comparing the output of gprof and Tau (pprof):

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 72.69     25.01    25.01   300000    83.37    83.37  collision
 15.03     30.18     5.17   300000    17.23    17.23  propagate
  9.13     33.32     3.14   300000    10.47    10.47  total_density
```

NB use a long run for better stats

```
---------------------------------------------------------------------------------
%Time  Exclusive  Inclusive   #Call    #Subrs  Inclusive  Name
           msec  total msec                   usec/call
---------------------------------------------------------------------------------
100.0     2,245     44,288        1    600004   44288530  int main(int, char **) C
 85.0     2,509     37,654   300000    900003        126  double timestep(..)
 63.5    28,138     28,138   300000         0         94  int collision(..)
 14.2     6,300      6,300   300000         0         21  int propagate(..)
  9.2     4,056      4,056   300000         0         14  double total_density(..)
```

Using Tau

Now we add in some OpenMP directives:

FUNCTION SUMMARY (mean):

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
74.8	20,438	51,918	0.75	225000	69224728	.TAU application
58.9	36,355	40,902	975000	675000	42	parallelfor [OpenMP location: file:d2q9-bgk.chk.c <217, 282>]
53.3	2,744	37,003	300000	300000	123	parallel begin/end [OpenMP]
46.7	2,293	32,393	300000	300000	108	for enter/exit [OpenMP]
29.6	1,578	20,529	300000	300000	68	barrier enter/exit [OpenMP]
25.2	549	17,506	0.25	150001	70024783	int main(int, char **) C
21.9	715	15,219	75000	225001	203	double timestep(..)
17.2	220	11,952	75000	75000	159	int collision(..)

- **`pprof`** reads files (in the same dir) called, **`profile.0.0.0`**, **`profile.0.0.1, profile.0.0.2`** etc.

Using Tau

Can also see per thread timings e.g.:

```
NODE 0;CONTEXT 0;THREAD 2:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call       #Subrs  Inclusive Name
              msec   total msec                         usec/call
---------------------------------------------------------------------------------------
100.0     27,217     1:09.225            1      300000    69225626  .TAU application
 60.7      2,190       42,008       300000      300000         140  parallel begin/end [OpenMP]
 57.5     36,104       39,817       900000      600000          44  parallelfor [OpenMP location:
                                                                     file:d2q9-bgk.chk.c <217, 282>]
 54.9      2,095       37,978       300000      300000         127  for enter/exit [OpenMP]
 36.8      1,618       25,444       300000      300000          85  barrier enter/exit [OpenMP]
```

See also **`paraprof`**, for a graphical representation.

University of BRISTOL

Profiling MPI Programs

- This time:
  - **`module add tools/gnu_builds/tau-2.23.1-openmpi`**
- and replace **`mpicc`** with **`tau_cc.sh`**, again
- this is the profile for example3/send_trapezoid.exe (N=400,000):

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time  Exclusive  Inclusive     #Call     #Subrs   Inclusive   Name
       msec       total msec                       usec/call
---------------------------------------------------------------------------------------
100.0        2       1,967          1         6.5   1967799   int main(int, char **) C
 54.1    1,065       1,065          1           0   1065480   MPI_Init()
 45.0      445         885          1      100001    885537   double trapezoid(..) C
 22.3      439         439     100001           0         4   double f(double) C
  0.7       13          13          1           0     13827   MPI_Finalize()
  0.0     0.0235      0.0235     0.75           0        31   MPI_Send()
  0.0     0.0217      0.0217     0.75           0        29   MPI_Recv()
  0.0     0.00225     0.00225    1              0         2   MPI_Comm_rank()
  0.0     0.00225     0.00225    1              0         2   MPI_Comm_size()
```


University of BRISTOL

Profiling MPI Programs

- and unsurprisingly—in this case—a very similar pattern for one of the other ranks.

NODE 3;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	0.225	1,973	1	6	1973838	int main(int, char **) C
54.2	1,069	1,069	1	0	1069833	MPI_Init()
45.3	454	893	1	100001	893450	double trapezoid(..) C
22.2	438	438	100001	0	4	double f(double) C
0.5	10	10	1	0	10293	MPI_Finalize()
0.0	0.031	0.031	1	0	31	MPI_Send()
0.0	0.003	0.003	1	0	3	MPI_Comm_rank()
0.0	0.003	0.003	1	0	3	MPI_Comm_size()

Profiling MPI Programs

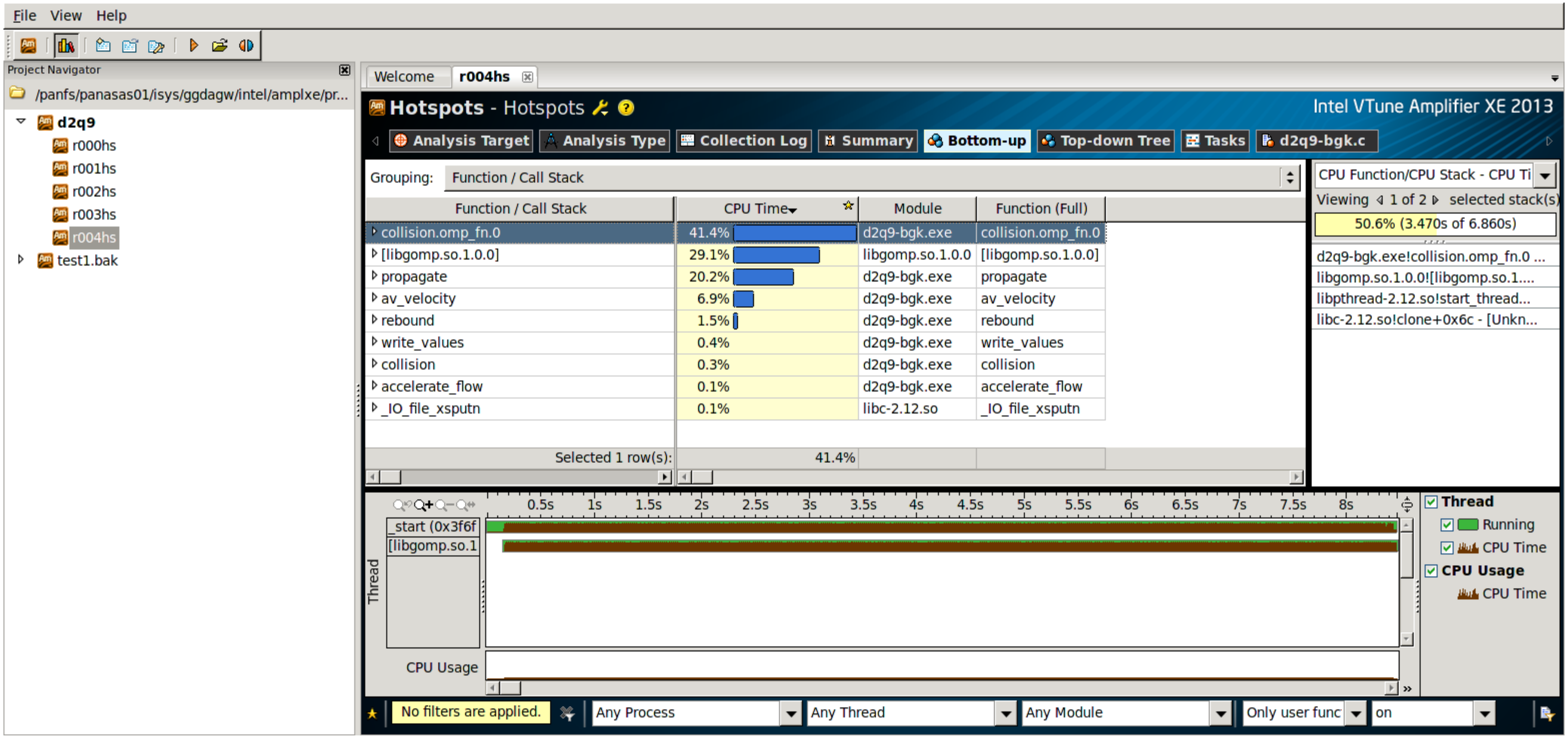
- For **`example4/blocking.exe`**, we see that the master process spends the vast majority of time in blocking receives (since we put calls to sleep() in the sending processes!):

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time   Exclusive    Inclusive     #Call   #Subrs  Inclusive Name
        msec          total msec                  usec/call
---------------------------------------------------------------------------------------
100.0          0.278  16,161          1        7   16161175   int main(int, char **) C
 92.8   15,004        15,004          3        0    5001396   MPI_Recv()
  6.8    1,096         1,096          1        0    1096404   MPI_Finalize()
  0.4       60            60          1        0      60302   MPI_Init()
  0.0          0.002        0.002     1        0          2   MPI_Comm_rank()
  0.0          0.002        0.002     1        0          2   MPI_Comm_size()
```


University of BRISTOL

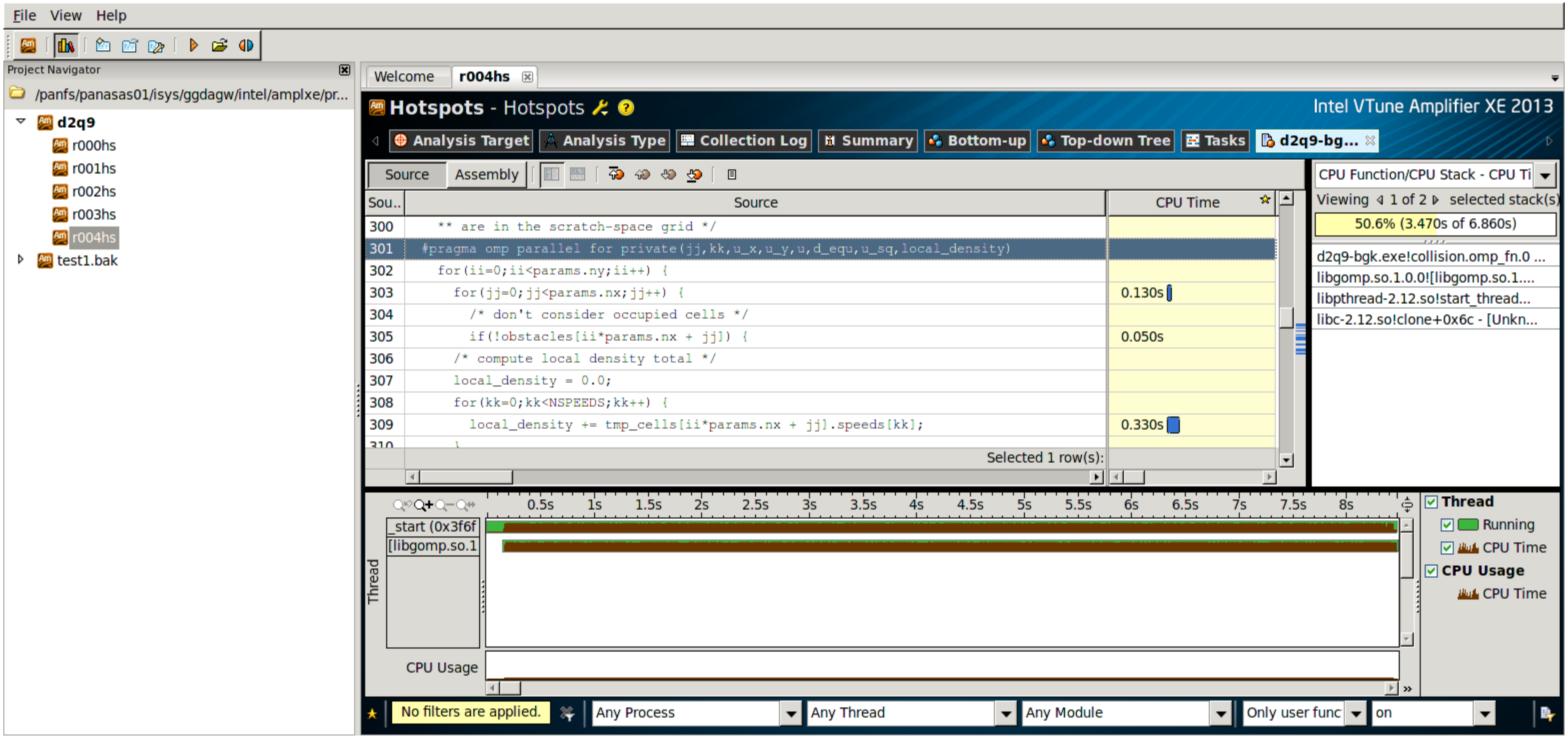
Profiling: VTune

- See https://www.acrc.bris.ac.uk/acrc/resources.htm
- Using Intel VTune Amplifier on BlueCrystal Phase 3



University of BRISTOL

Profiling: VTune



Offers line-by-line profiles

University of BRISTOL

Cache Profiling with PAPI: Example

```
> module add libraries/gnu_builds/papi-5.3.0
> papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.3.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Max Megahertz        : 2599
CPU Min Megahertz        : 2599
Hdw Threads per core     : 1
Cores per Socket         : 8
Sockets                  : 2
NUMA Nodes               : 2
CPUs per Node            : 8
Total CPUs               : 16
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------

    Name        Code    Avail Deriv Description (Note)
...
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
```

on a BCp3
compute node..

University of BRISTOL

Cache Profiling with PAPI: Example

```
#include <papi.h>
...
int main(int argc, char *argv[]) {
...
    long long counters[3];
    int PAPI_events[] = {
        PAPI_TOT_CYC,
        PAPI_L2_DCM,
        PAPI_L2_DCA };
...
PAPI_library_init( PAPI_VER_CURRENT );
...
i = PAPI_start_counters( PAPI_events, 3 );
...
/* your operations of interest here */
...
    printf("%lld L2 cache misses (%.3lf%% misses) in %lld cycles\n",
        counters[1],
        (double)counters[1] / (double)counters[2],
        counters[0] );
...
```

If you're interested in level 2 cache misses..

University of BRISTOL

Debugging

University of BRISTOL

Debugging

John Backus (the inventor of Fortran) on coding:

“***You need the willingness to fail all the time***.
You have to generate many ideas and then you have to work hard only to discover that they don't work. And you have to keep doing that over and over until you find one that does work.”

Take a moment to ask yourself, “does that sound like me?...”

Parallel Bugs: Floating Point Considerations

- Floating point addition is not commutative, meaning
- a + (b + c) ≠ (a + b) + c:

```
xx (0.43) is: 0.42999999999999999334
yy (0.67) is: 0.67000000000000003997
zz (0.37) is: 0.36999999999999999556
xx + (yy + zz) is: 1.46999999999999997335
but..
(xx + yy) + zz is: 1.47000000000000019540
```

- It's dangerous to compare a floating point number to a threshold.
- Consider comparing +/- a tolerance.
- “What every computer scientist should know about floating point artimetic”: http://dl.acm.org/citation.cfm?id=103163

University of BRISTOL

Debugging: Compilers Can Help

- Enable all warnings (e.g. for GNU, `-Wall`)
- Instrument code for debugging (`-g`)
- Enable stack traces (e.g. `-traceback`, Intel)
- But be aware: optimisation can reorder operations or even completely remove some.
- Useful links:
  http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
  http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
  http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html

Debugging Threaded Code with **gdb**

- **`info threads`**
  - gives, among other things, the **`threadno`** assigned to each thread by gdb.
- **`thread`** ***`threadno`***
  - switches the context to a particular thread.
- **`thread apply [threadno] [all]`** ***`args`***
  - apply a command to one—or all—threads.
- See tarball of debugging examples for more info on gdb, including a useful cribsheet.

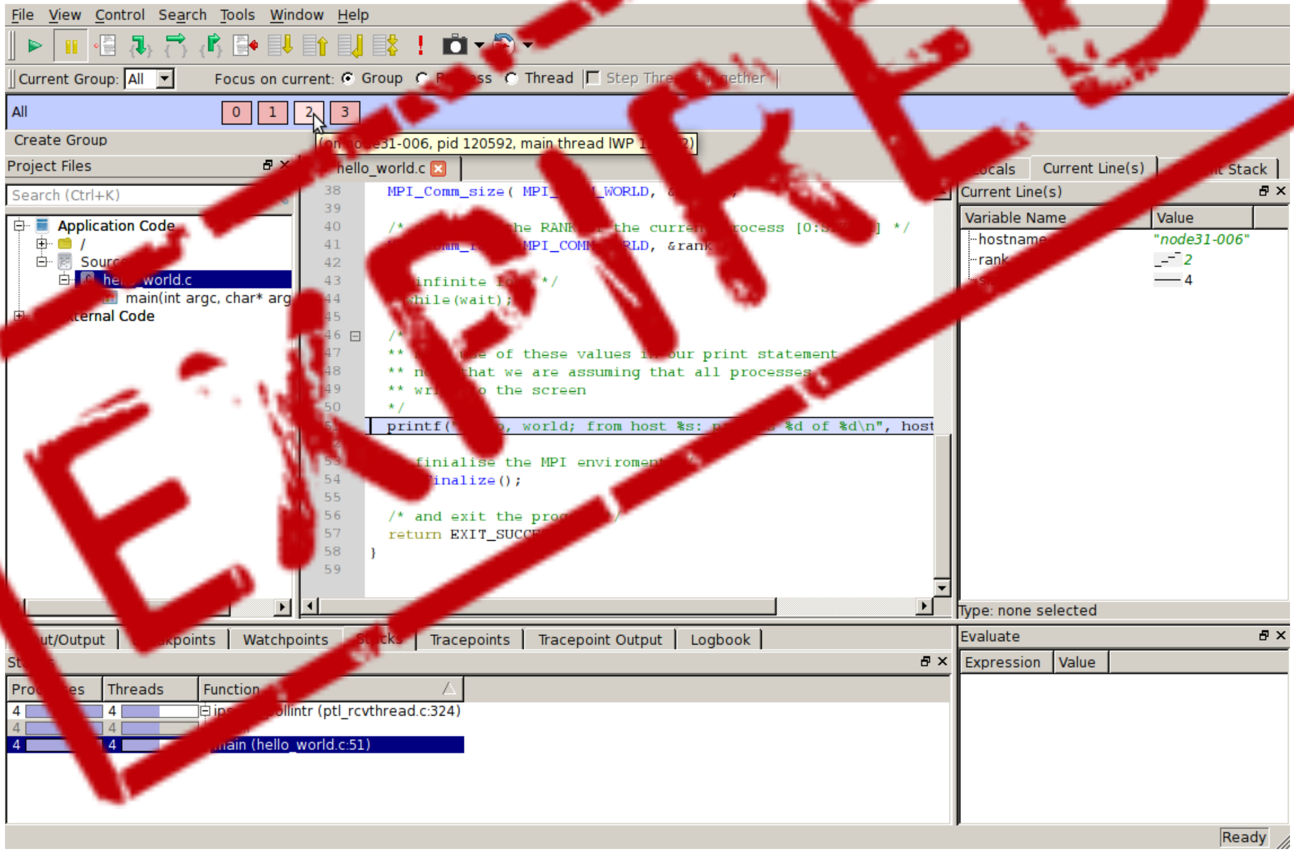
Debugging & the Queuing System: Threads

- You can request an interactive session on a compute node via the queuing system using:
  - **`qsub -q teaching -lnodes=1:ppn=<N> -I`**
  - **`[ggdagw@node001 ~]$`**
- It is then straightforward to run a debugger interactively.

University of BRISTOL

Debugging & the Queuing System: Distributed Memory – Using DDT

- See https://www.acrc.bris.ac.uk/acrc/resources.htm
- Recipes for Using Allinea's DDT Parallel Debugger ...



University of BRISTOL

Debugging & the Queuing System: Distributed Memory – Using GDB

<u>In lieu of a better tool, we need to attach debuggers to a process after it has queued.</u>

Add the following to your program so that your processes stall **(busy idle)** once they start running on a node:

```
int debugWait = 1;
…
while (debugWait); // infinite loop
```

Determine which nodes your processes are running on through the queuing system:

`qstat -n -u` ***`username`***

Debugging & the Queuing System: Distributed Memory – Using GDB

Connect to the node:

```
ssh node
```

Find the process number(s) for your running program:

`ps -elf | grep progname` (or `'top'`)

Launch gdb, attaching to a running process:

```
gdb progname procno
```

and 'liberate' the process:

```
(gdb) break lineno
(gdb) run
(gdb) set debugWait = 0
(gdb) cont
```

Simple! Right?

University of BRISTOL

MPI Programs: Some Leads..

- Common sense tells us that deadlock is going to be a common bug...
- ...and that a blocking call, such as **`MPI_Recv()`**, is a likely place to hang.
- It's amazing how many bugs can be found with good old **`printf()`**...
- **`__FILE__`** and **`__LINE__`** are useful macros to find out where you are in a program...
- See also: https://www.open-mpi.org/faq/?category=debugging

University of BRISTOL

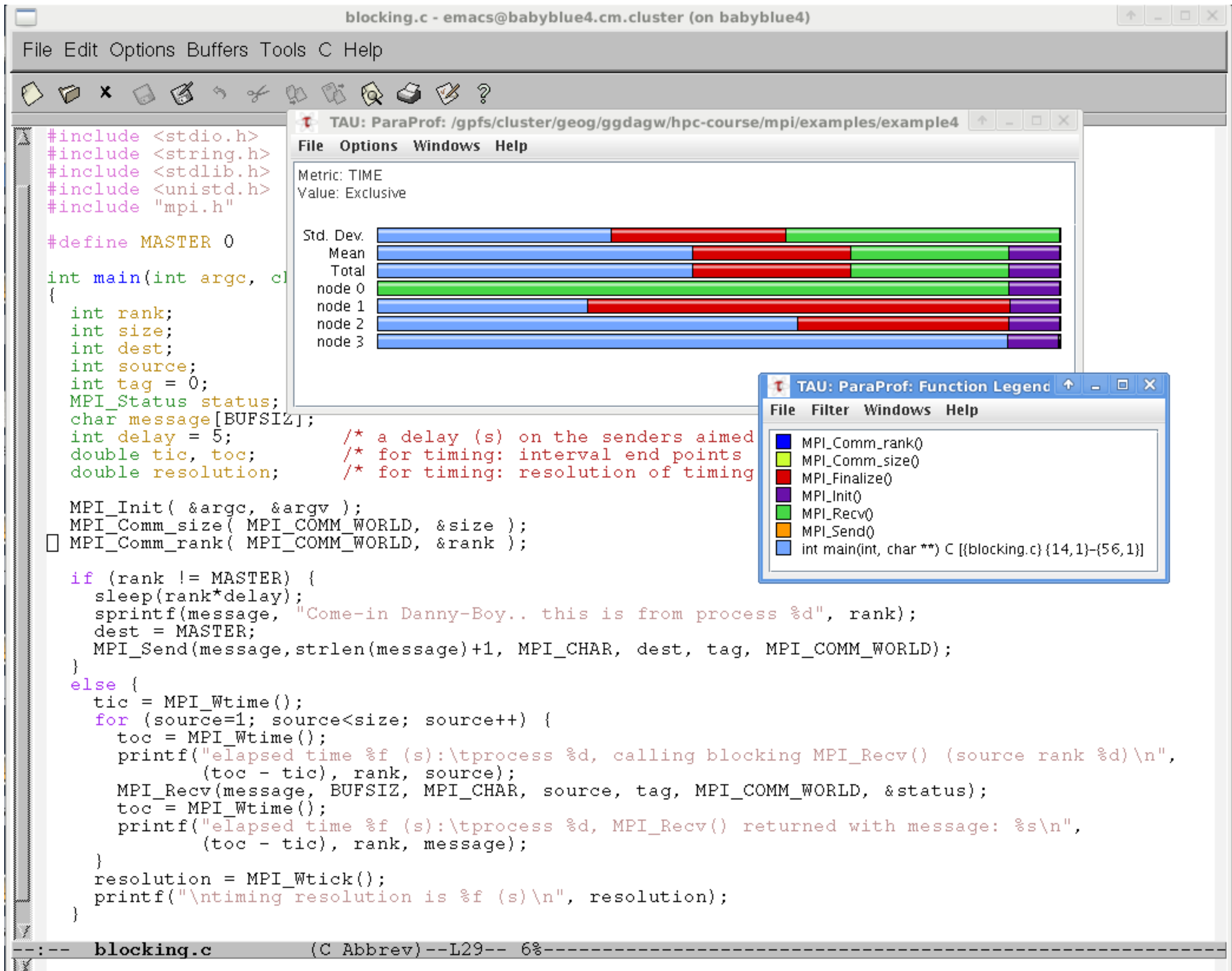
Recap

- Sometime all you have to debug is `printf()`.
- You will be rewarded many times over for time spent planning your code on paper.
- Many good profiling tools, including **`Tau`**—don't guess!
- You can use **`gdb`** for multi-threaded code and can attach it to running MPI processes.
- Use all your serial code debugging nouse.
- Race conditions are slippery!

University of BRISTOL

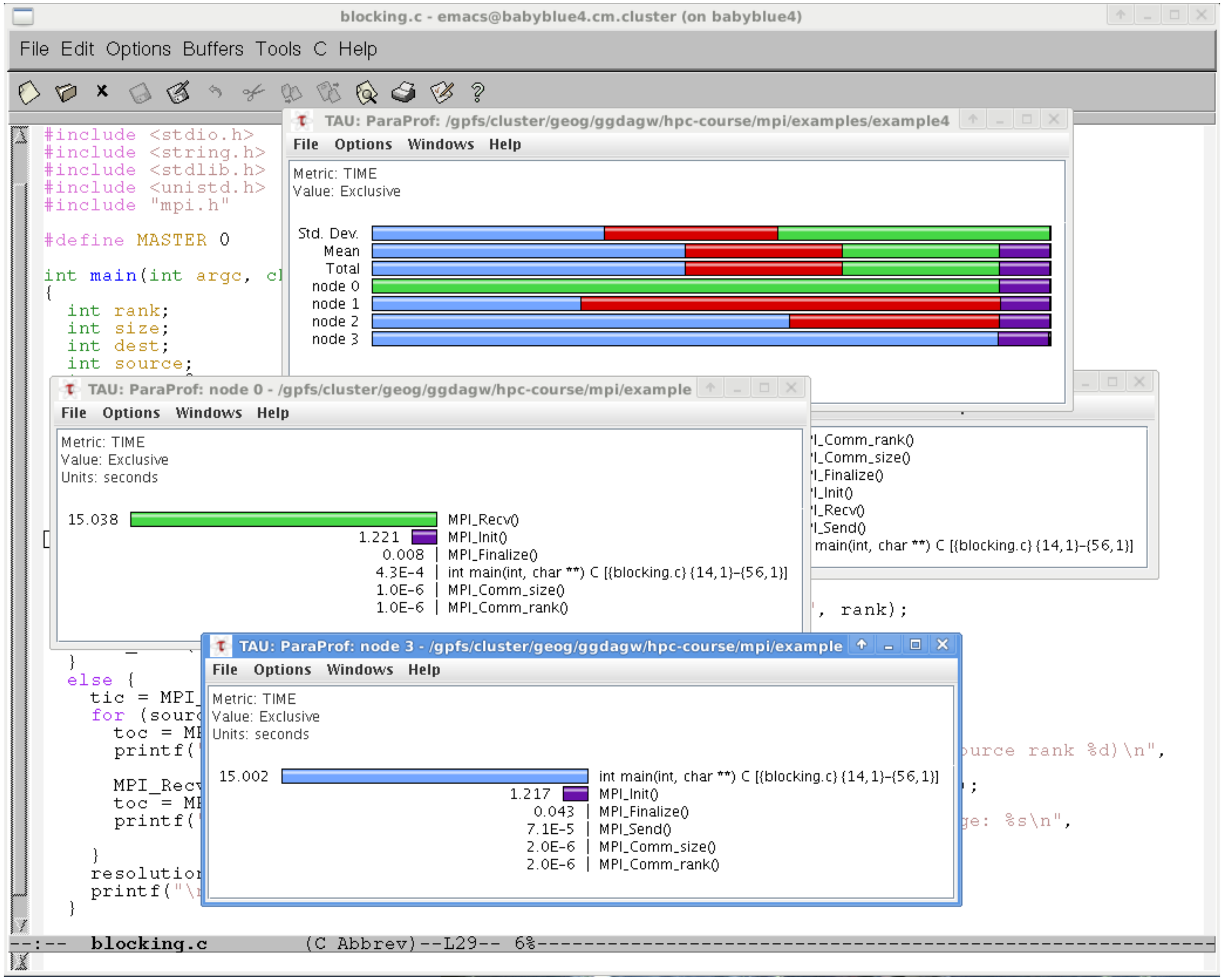
Extras

TAU ParaProf: Load Balance

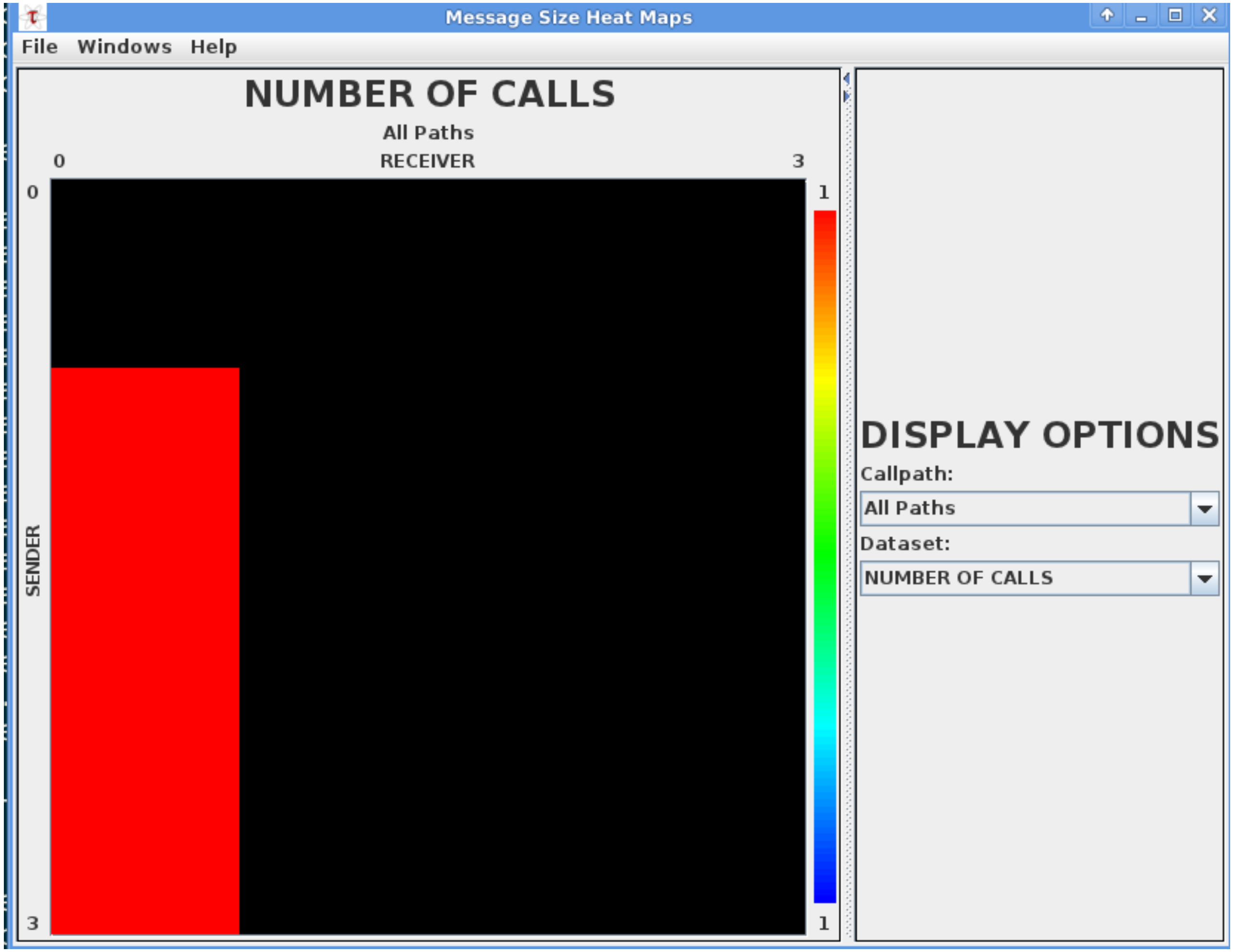


University of BRISTOL

TAU ParaProf: By Process



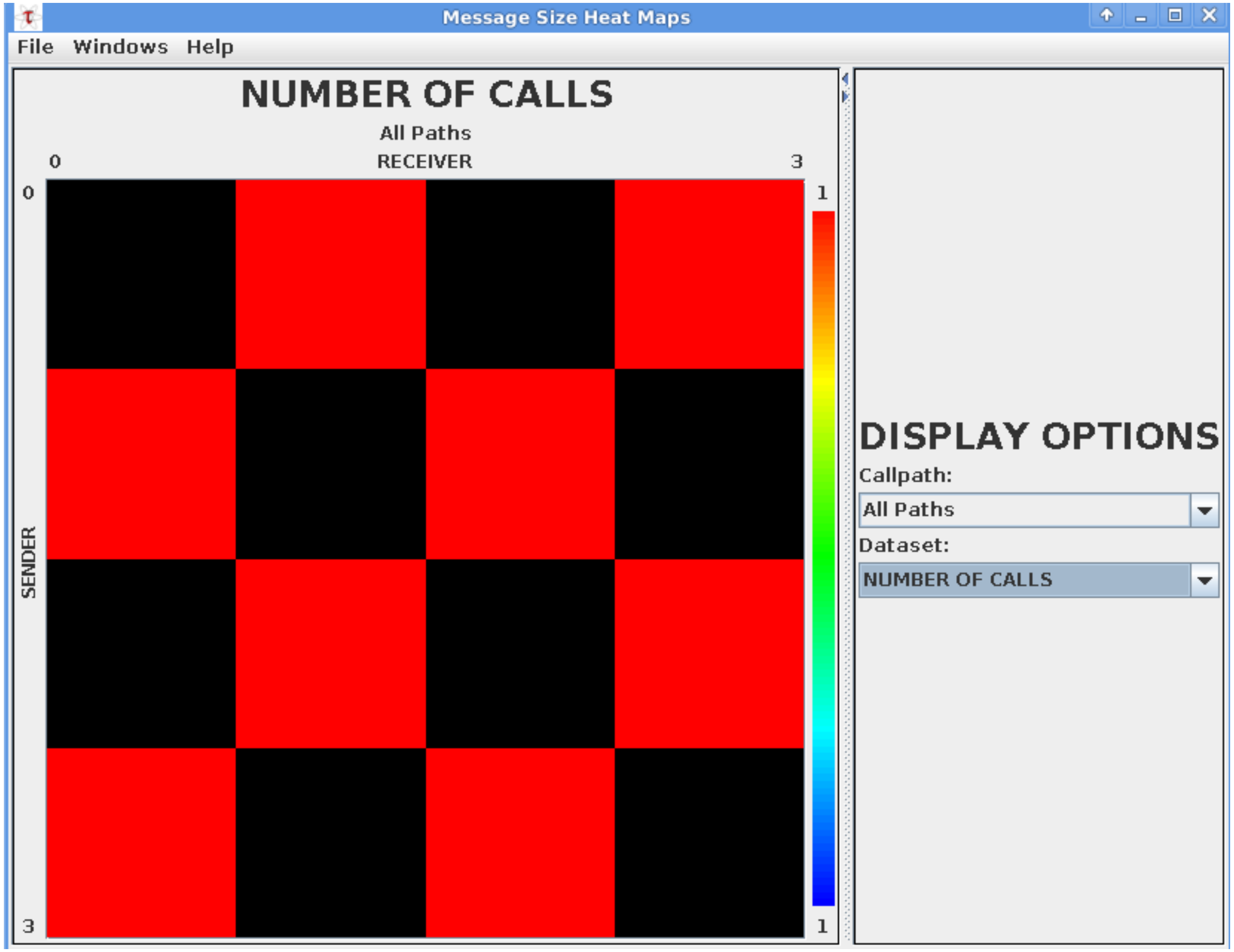
TAU ParaProf: Comms Pattern



Send to Rank 0: Master-worker pattern

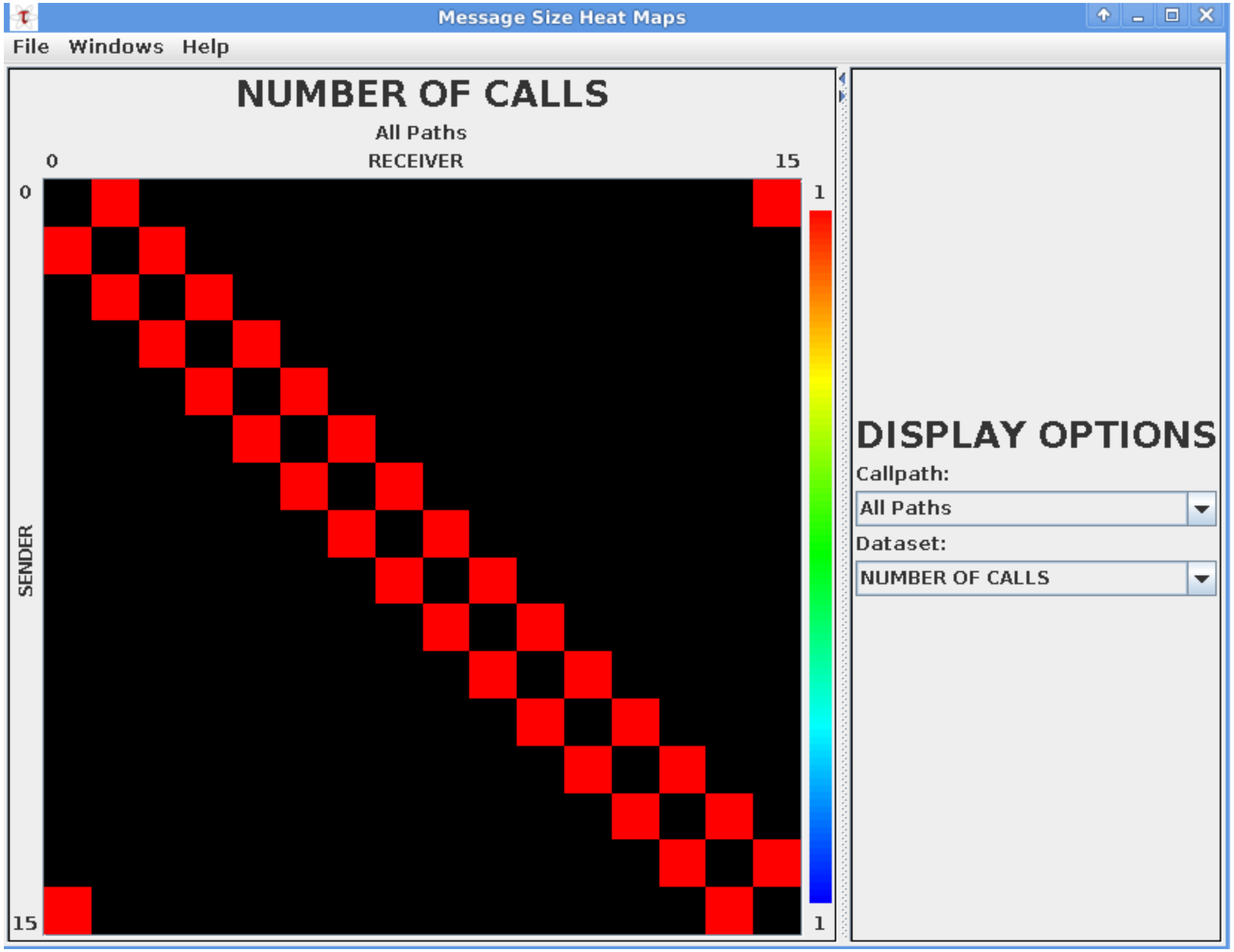
University of BRISTOL

TAU ParaProf: Comms Pattern



Send to Neighbours: Halo Exchange

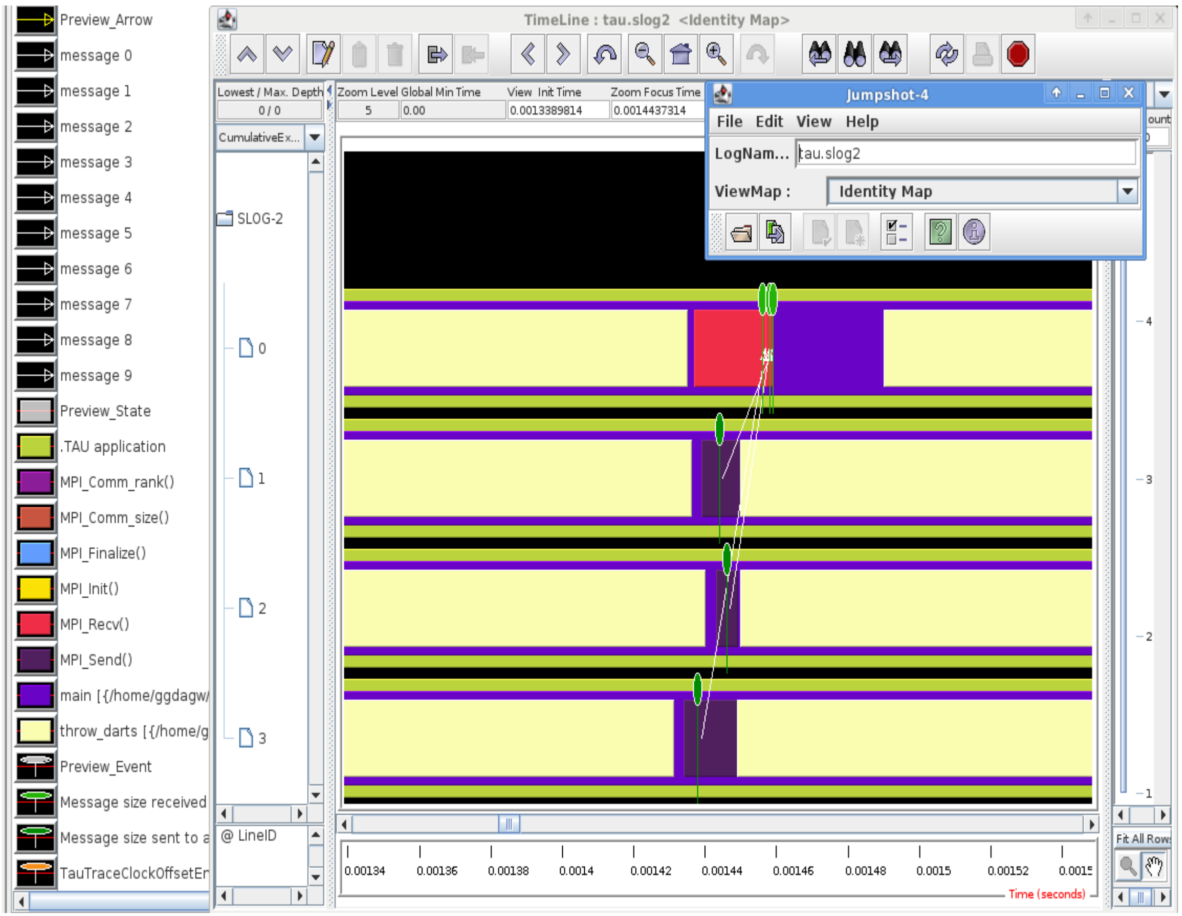
TAU ParaProf: Comms Pattern



Send to Neighbours: Halo Exchange

University of BRISTOL

TAU Jumpshot: Trace



See message pattern, timings, load balance..
BUT, v. slow over network

University of BRISTOL