Performance Tools

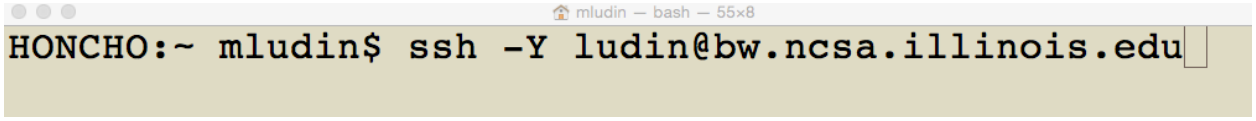
First things first: Login to Blue Waters

`$ ssh -Y`: There are tools available on BW for debugging, visualization, and performance tools that use a graphical user interface (GUI). To launch a GUI application remotely we need to pass the "`-Y`" to the ssh command, for example:

Training accounts:

```
ssh -Y ludin@bwbay.ncsa.illinois.edu
```

This will login to Blue Waters using the username "ludin". Keep in mind that temporary accounts login to `bwbay` but project accounts login to `bw.ncsa.illinois.edu`, as shown below.



Request an interactive session for two nodes and three hours:

`$ qsub -I`: The command below is asking for two XE interactive compute nodes, each node with 32 cores, and for 3 hours of total walltime.

```
qsub -I -l nodes=2:ppn=32:xe,walltime=03:00:00
```

Copy the performance tools directory from Mobeen's account on BW:

`$ cp -r source destination`: The "**`-r`**" is used for recursive copy. Meaning copy the directory and all other files and folders within it.

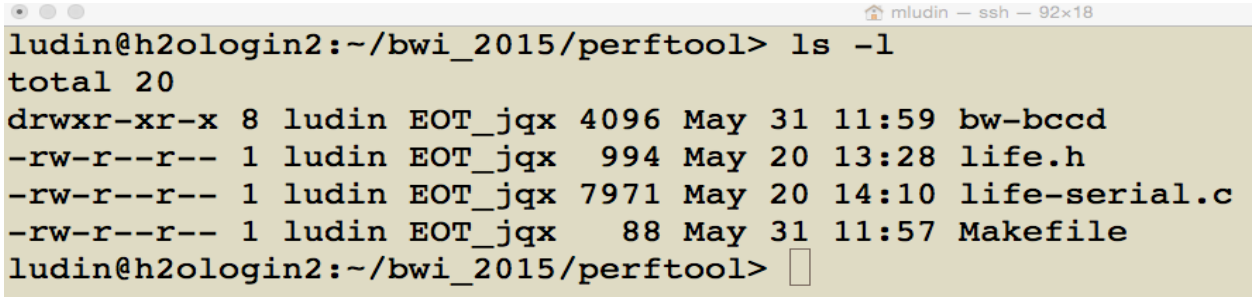
```
cp -r ~ludin/bwi_2015/perftool ~/scratch
```

NOTE: When you copy files/directories from another user's directory you must use just "**`~`**" followed by the user's username. The "**`~/`**" is used for your own home directory. The command above copies the "**`perftool`**" directory from Mobeen's account to the "**`scratch`**" directory on your account. For big projects make sure you use the "**`scratch`**" directory. Otherwise you might encounter **quota/limited disk space** issue.

Now go to the perftool directory:

```
cd ~/scratch/perftool
```

Note: use "**`ls -l`**" command to make sure you have the following files:



What is profiling:

To analyze the runtime behavior of the program, these are the questions one can ask oneself when profiling:

- Which parts (functions, statements, . . .) of a program take how long?
- How often are functions called?
- Which functions call which other functions?
    - Construct the dynamic call graph
- Memory consumption
    - Memory accesses, memory leaks
    - Cache performance

Stages of profiling:

On BW first make sure you have the right programming environment loaded. You will need to load the following modules:

- GNU gprof profiler: PrgEnv-gnu
- Cray Reveal profiler: PrgEnv-cray

Profiling has several steps:

1. You must compile and link your program with profiling enabled.
2. You must execute your program to generate a profile data file.
3. You must run the profiler to analyze the profile data.

GNU gprof Profiler (Manual: http://wwwcdf.pd.infn.it/localdoc/gprof.pdf) :

gprof provides several different methods for analyzing the program such as:

- **Flat Profile:**
    - How much time does the program spend in which functions?
- **Call Graph:**
    - Which functions call which functions, and how often?
- **Annotated Sources Listing:**
    - Annotate each source line with the number of executions

Steps for using GNU "`gprof`" profiler on Blue Waters:

You will want to do these steps in the `perftool` directory from earlier.

1. Switch to GNU programming environment from Cray:

```
module swap PrgEnv-cray PrgEnv-gnu
```

To make sure that the module was loaded:

```
module list
```

PrgEnv-gnu should be listed.

2. In order to compile and link our program with profiling enabled, the "-g -pg" options are needed for providing debugging and profiling information:

```
cd Life/
gcc -g -pg -o life-serial life-serial.c
```

The "**-pg**" should generate a file "**gmon.out**".

```
ludin@h2ologin2:~/bwi_2015/perftool> gcc -g -pg -o life-serial life-serial.c
ludin@h2ologin2:~/bwi_2015/perftool>
ludin@h2ologin2:~/bwi_2015/perftool>
ludin@h2ologin2:~/bwi_2015/perftool> ls
bw-bccd  gmon.out  life.h  life-serial  life-serial.c  Makefile
ludin@h2ologin2:~/bwi_2015/perftool>
```

3. Now that we have successfully linked and compiled our program for profiling, the next step is to run our program to actually generate the profiling information:

```
./life-serial -t 100 -r 100 -c 100
```

This command will generate a bunch of output in an attempt to visualize the program. We can ignore this output.

`gprof` is smart, but usually it still needs a bigger problem size to create meaningful profiling data.

4. Now let's finally use the "gprof" command to visualize the program profiled data:

```
gprof ./life-serial gmon.out > profiling_data
```

This command will save the visualization data into a file called `profiling_data`. The command itself will not show anything on the terminal unless there is a typo.

5. See the data using the `less` command:

```
less profiling_data
```

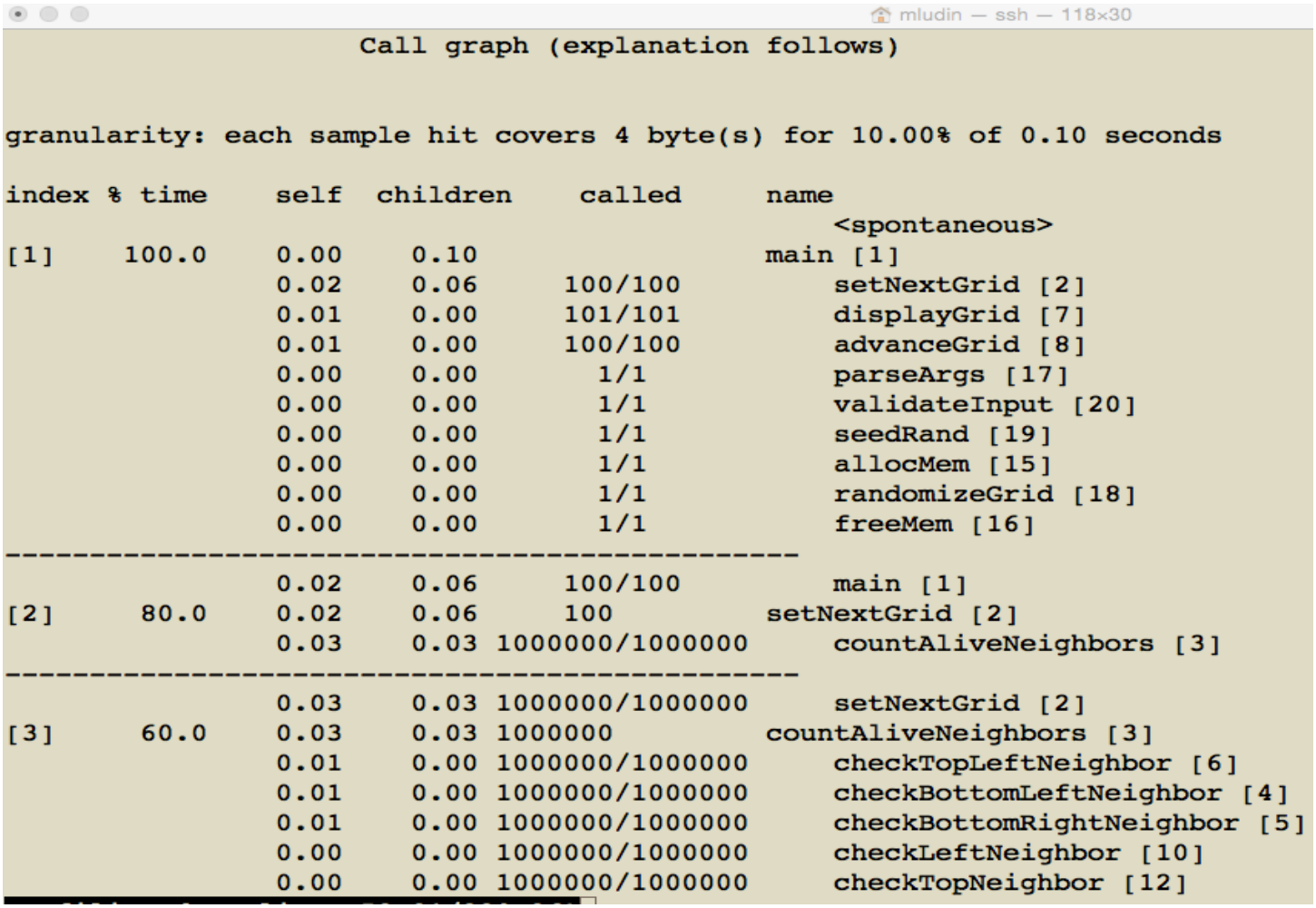
You can use the up/down arrow to move up and down.

6. **Flat Profile:** In which functions did your program spent most of its time?

A sample output is provided below.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 30.00      0.03     0.03  1000000     0.03     0.06  countAliveNeighbors
 20.00      0.05     0.02      100   200.00   800.00  setNextGrid
 10.00      0.06     0.01  1000000     0.01     0.01  checkBottomLeftNeighbor
 10.00      0.07     0.01  1000000     0.01     0.01  checkBottomRightNeighbor
 10.00      0.08     0.01  1000000     0.01     0.01  checkTopLeftNeighbor
 10.00      0.09     0.01      101    99.01    99.01  displayGrid
 10.00      0.10     0.01      100   100.00   100.00  advanceGrid
  0.00      0.10     0.00  1000000     0.00     0.00  checkBottomNeighbor
  0.00      0.10     0.00  1000000     0.00     0.00  checkLeftNeighbor
  0.00      0.10     0.00  1000000     0.00     0.00  checkRightNeighbor
  0.00      0.10     0.00  1000000     0.00     0.00  checkTopNeighbor
  0.00      0.10     0.00  1000000     0.00     0.00  checkTopRightNeighbor
  0.00      0.10     0.00    10000     0.00     0.00  getRandState
  0.00      0.10     0.00        1     0.00     0.00  allocMem
  0.00      0.10     0.00        1     0.00     0.00  freeMem
  0.00      0.10     0.00        1     0.00     0.00  parseArgs
  0.00      0.10     0.00        1     0.00     0.00  randomizeGrid
  0.00      0.10     0.00        1     0.00     0.00  seedRand
  0.00      0.10     0.00        1     0.00     0.00  validateInput

 %           the percentage of the total running time of the
time         program used by this function.
```

7. **Call Graph:** How much time was spent in each function and other functions that were called from it? The call graph helps us find out if the functions themselves spent a long time or if their children/subfunctions took most of the runtime. A sample output is provided below.

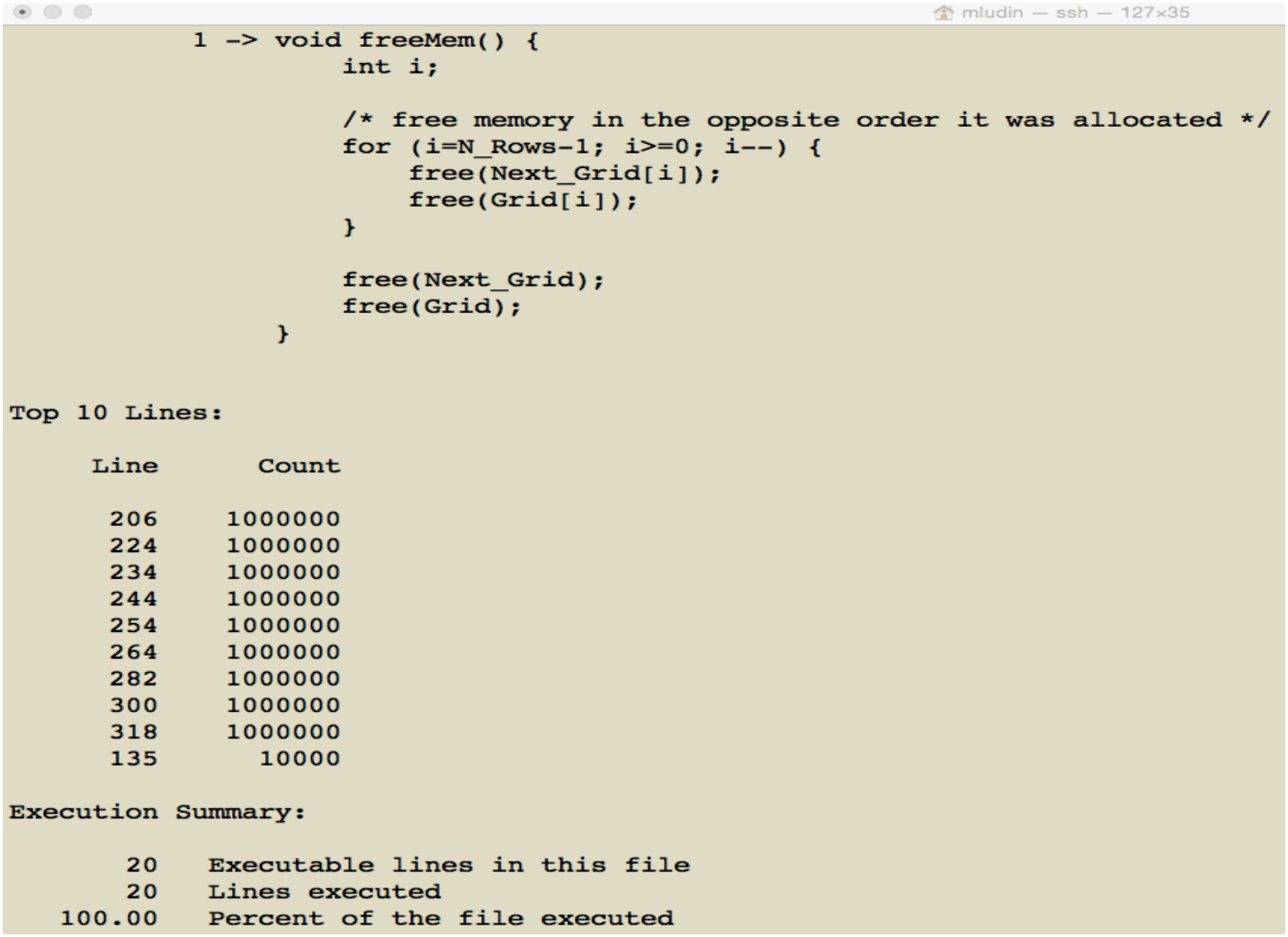


```
mludin — ssh — 118×30
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 10.00% of 0.10 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.10                 main [1]
                0.02    0.06     100/100         setNextGrid [2]
                0.01    0.00     101/101         displayGrid [7]
                0.01    0.00     100/100         advanceGrid [8]
                0.00    0.00       1/1           parseArgs [17]
                0.00    0.00       1/1           validateInput [20]
                0.00    0.00       1/1           seedRand [19]
                0.00    0.00       1/1           allocMem [15]
                0.00    0.00       1/1           randomizeGrid [18]
                0.00    0.00       1/1           freeMem [16]
-----------------------------------------------
                0.02    0.06     100/100         main [1]
[2]     80.0    0.02    0.06     100           setNextGrid [2]
                0.03    0.03 1000000/1000000     countAliveNeighbors [3]
-----------------------------------------------
                0.03    0.03 1000000/1000000     setNextGrid [2]
[3]     60.0    0.03    0.03 1000000           countAliveNeighbors [3]
                0.01    0.00 1000000/1000000     checkTopLeftNeighbor [6]
                0.01    0.00 1000000/1000000     checkBottomLeftNeighbor [4]
                0.01    0.00 1000000/1000000     checkBottomRightNeighbor [5]
                0.00    0.00 1000000/1000000     checkLeftNeighbor [10]
                0.00    0.00 1000000/1000000     checkTopNeighbor [12]
```

The call graph divides each function and its children as separate entries separated by lines of dashes. In each entry the main function/parent function has an index value in [ ], to the left.

8. **Annotated Sources Listing:** We have to give gprof the "**-A**" option to trigger the annotated source listing. This will print out the source code for each function and how many times it was called. It will also give you **top 10 lines** and **count** for how many times they were called. An example is shown below.

```
gprof ./life-serial gmon.out -A
```



Before you go to break, please make sure run `top` in your compute node terminal and login node too.

For Profiling GalaxSee with gprof:

- Make sure you request a compute nodes if you lost connection or don't have, and make sure you're on a compute node to execute these commands.

- **`export GMON_OUT_PREFIX=gmon.out-`**
  `(this will make every process to write its profile data in a separate file)`
- **`cd ~/scratch/perftool/GalaxSee`**
- **`make clean && make`**
- **`aprun -n 1 ./GalaxSee.cxx-mpi`**
- **`gprof ./GalaxSee.cxx-mpi gmon.out-.* > profile_data`**
- **`less profile_data`**

Cray Reveal Profiler:
Install X11 (Mac Users only): http://xquartz.macosforge.org/landing/

Cray Reveal is another profiler that is more specific for finding candidate loops for parallelism. It is mainly designed for shared memory, OpenMP, and is currently also being used for OpenACC. Documentation is available on the BW portal. First we will be using Cray's Apprentice2 tool for visualizing data captured during the program execution. Its a GUI tool that let us visualize some awesome information about our program. Second, we will use Cray Reveal with Apprentice2 to find out best candidate loops for OpenMP.

Step 1-4 generates loop statistics using CrayPAT to determine which loops have the most work. Step 5-7 is to visualize performance data captured during program execution, and the remaining steps to optimize code using Reveal.

Requirements: you must be compiling and running your program under the Cray programming environment (PrgEnv-Cray)

1. **Switch to Cray Programming environment if you haven't:**

```
module swap PrgEnv-gnu PrgEnv-cray
```

2. **Load perftools**: By default the BW system loads the "Darshan" profiler. We need to unload that so we can load the Cray's perftools.

```
module swap darshan perftools
```

3. **Compile and link the code**: Instead of "**-pg"**, we will need to pass "**-h profile_generate**" when compiling for loop work measurements. Also remember we will not be using the "**gcc**" compiler and instead will use "**cc**".

```
cd ~/scratch/perftool/Life
cc -h profile_generate -o life-serial life-serial.c
```

This will save some objects files in a hidden directory, so you will get an message like this "**WARNING: CrayPat is saving object files from a temporary directory into directory '/u/training/ludin/.craypat/life-serial/40149**"

4. **Instrument the executable for tracing**: Command below will generate a new instrumented executable "**life-serial+pat**"

```
pat_build -w life-serial
```

5. **Generate new data file**: Now on a compute node make sure you have the same modules (PrgEnv-cray, perftools) loaded, then run the instrumented code to generate new data file for example: "**life-serial+pat+27004-25430t.xf**"

```
./life-serial+pat -t 100 -r 100 -c 100
```

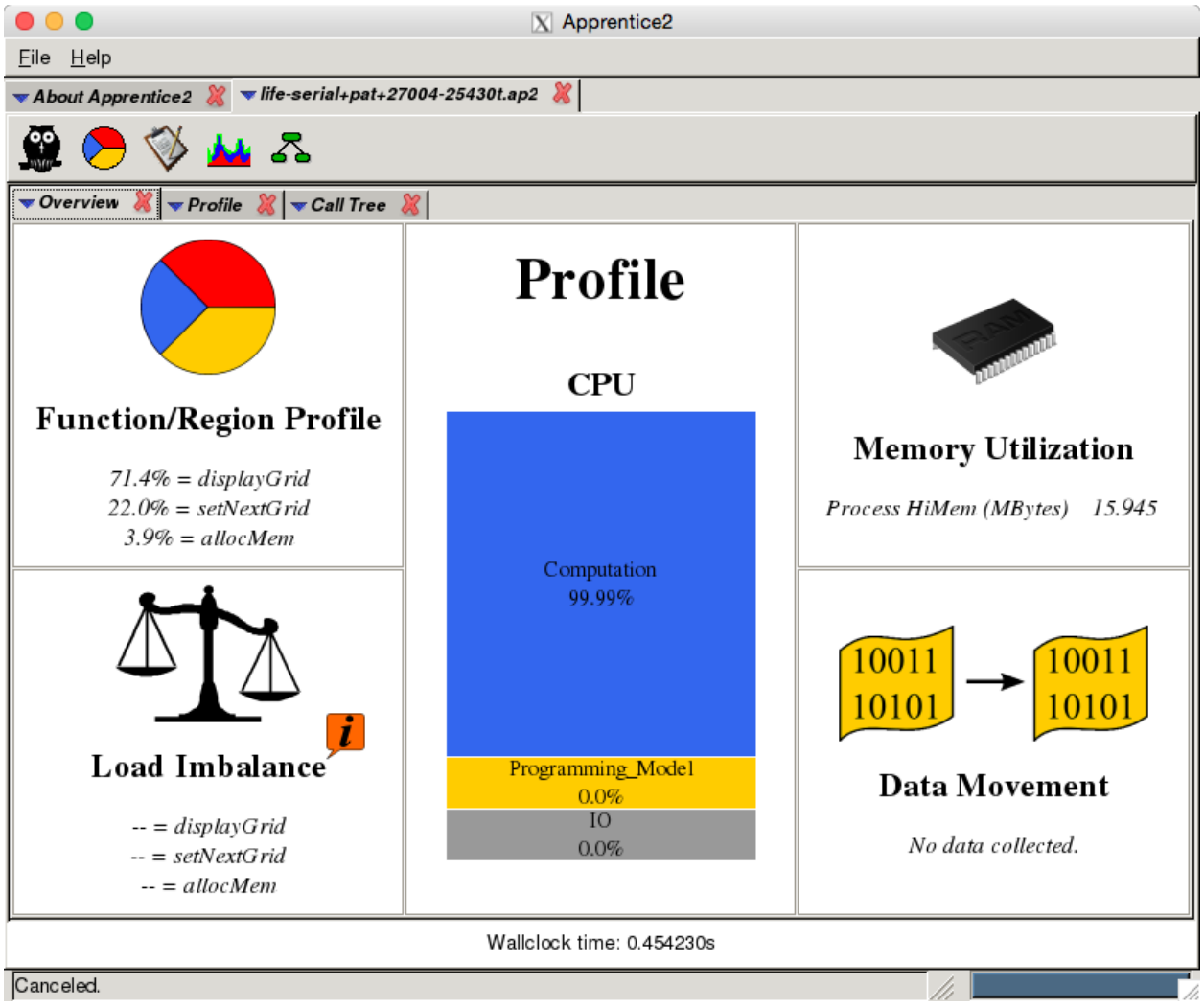
6. **Create report with loop statistics**: On the login node now, we have to process the **`*.xf`** file to create report with loop statistics. This generates "**`life-serial*.ap2`**" file that could later be used to visualize using Apprentice2.

```
pat_report -f ap2 life-serial*.xf > loop_report
```

7. We will use the output of "**`pat_report`**" to visualize with GUI analyzer "**Apprentice 2**". Now lets launch Apprentice2 (**`app2`**) on the login node:

```
app2 life-serial*.ap2 &
```

This will open the following display:



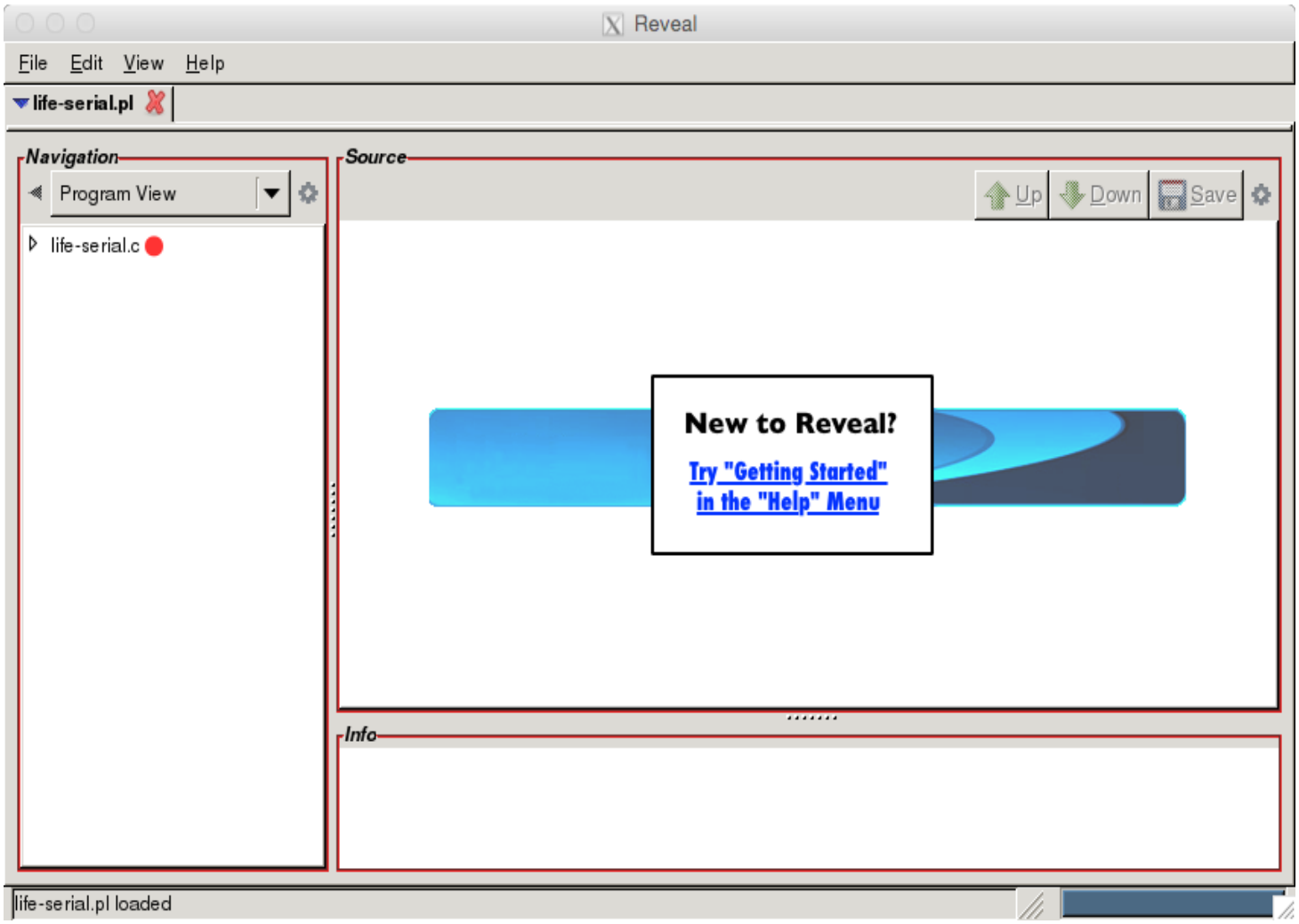
8. **Generate Program Library**: In order to use Cray Reveal, we have to recompile our code to generate a program library. This is necessary because the **`-h profile_generate`** disables most compiler optimizations. However, for Reveal want to know where the code does worst job, even when fully optimized.

```
rm life-serial
cc -O3 -h pl=life-serial.pl -h wp -o life-serial life-serial.c
```

This command should now generate a life-serial.pl file, which can be used with Reveal.

9. **Run Reveal**: First for compiler information only. It will open the following window:

```
reveal life-serial.pl
```

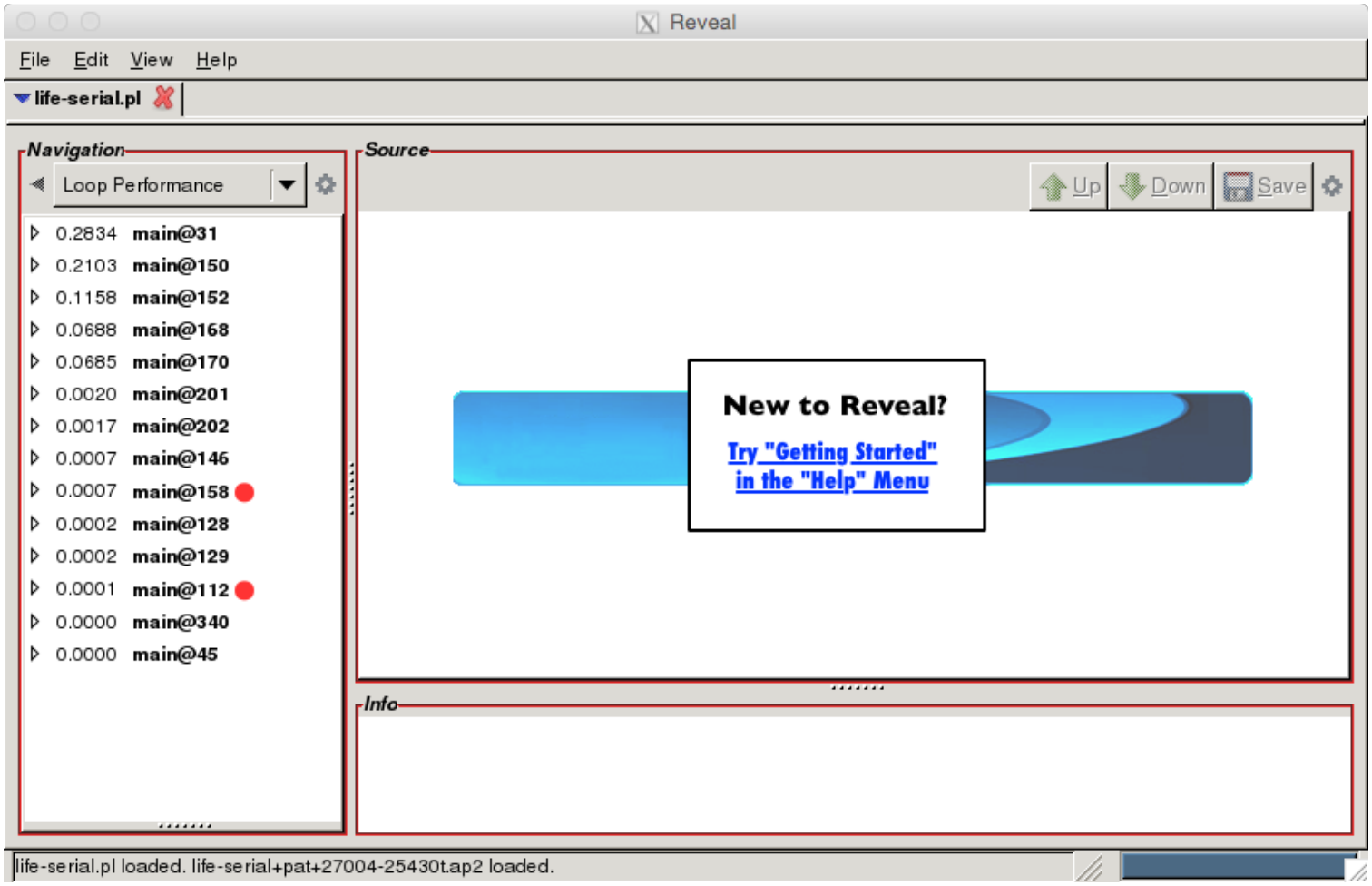


This GUI show what kind of automatic optimizations were performed by the compiler and suggest OpenMP directives for the loops.

10. **Visualize compiler and loop work estimates**: we will run Reveal by combining the program library with the performance data file (**`*.ap2`**)

```
reveal life-serial.pl life-serial+pat+*.ap2 &
```

This will open the following window:



Save the loops, open the file and resolve the unresolved variables to either shared or private.

Additional Resources:

Using Cray Reveal for OpenMP:
https://bluewaters.ncsa.illinois.edu/reveal-and-openmp

Cray C and C++ Refrence Manual: http://docs.cray.com/books/S-2179-83//S-2179-83.pdf

Cray Fortran Reference Manual: http://docs.cray.com/books/S-3901-83/S-3901-83.pdf

Cray C++ Compiler:
http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=id%3dcrayCC.1;f=man/xt_ccm/83/cat1/crayCC.1.html

Cray C Compiler:
http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=id%3dcrayCC.1;f=man/xt_ccm/82/cat1/craycc.1.html

Cray Fortran Compiler:
http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=id%3dcrayftn.1;f=man/xt_ftnm/83/cat1/crayftn.1.html

GCC Online Documentation: https://gcc.gnu.org/onlinedocs/

GNU Libgomp for OpenMP and OpenACC: https://gcc.gnu.org/onlinedocs/gcc-5.1.0/libgomp/

Intro-MPI for Cray XE System:
http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=;f=man/xe_mptm/72/cat3/intro_mpi.3.html

Tuning Parallel I/O on Blue Waters for Writing 10 Trillion Particles:
https://sdm.lbl.gov/~sbyna/research/papers/201504-CUG-VPICBW.pdf

Bunch of Cray Documentations:
http://docs.cray.com/cgi-bin/craydoc.cgi?mode=SiteMap;f=xe_sitemap

Optimization for the Cray XE6 Interlagos Architecture
http://www.erdc.hpc.mil/docs/Tips/optimizingForInterlagos.pdf

Using Cray Performance Measurement and Analysis Tools:
http://docs.cray.com/books/S-2376-622/S-2376-622.pdf

Overview of Gemini Hardware Counters: http://docs.cray.com/books/S-0025-10/S-0025-10.pdf

Cray Application Developer's Environment User's Guide:
http://docs.cray.com/books/S-2396-610/S-2396-610.pdf