Lab Report

ECPE 170 – Computer Systems and Networks – Spring 2021

Name: Kaung Khant Pyae Sone

Lab Topic: Performance Measurement (Lab #: 5)

Question #1:
Create a table that shows the real, user, and system times measured for the bubble and tree sort algorithms.
Answer:

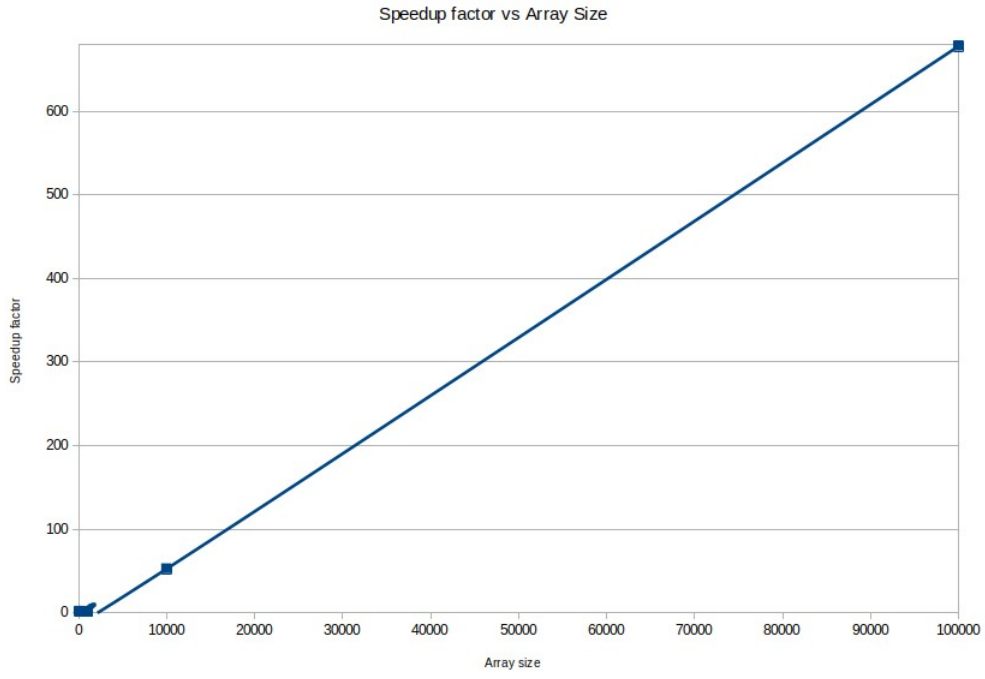
Times	Bubble Sort	Tree Sort
Real	25.579s	0.035s
User	25.271s	0.031s
System	0.288s	0.004s

Question #2:
In the sorting program, what actions take user time?
Answer:
Running the user application code

Question #3:
In the sorting program, what actions take kernel time?
Answer:
Reading or writing data from memory or other storage

Question #4:
Which sorting algorithm is fastest?
Answer:
Tree Sort

Array Size	Bubble Sort	Tree Sort	Speedup
100	0.002s	0.001s	2
1000	0.003s	0.002s	1.5
10000	0.211s	0.004s	52.75
100000	25.758s	0.038s	677.84



With array sizes 100 and 1000, the speed of bubble sort and tree sort are relatively close, with a speedup factor of 2 and 1.5. The smaller speedup value of array size 1000 may be caused by rounding, so we will assume that bubble sort and tree sort have the same performance at that array sizes 100 and 1000. At array size 10000, we can see that tree sort is faster than bubble sort by nearly 53 times, and at array size 100000, tree sort is now faster by nearly 678 times. Looking at the graph, the speedup factor seems to be linearly correlated to the array size positively.

Question #5:
Create a table that shows the total Instruction Read (IR) counts for the bubble and tree sort routines. (In the text output format, IR is the default unit used. In the GUI program, un-check the "% Relative" button to have it display absolute counts of instruction reads).
Answer:

Array Size: 100000	**Total Instruction Read counts**
Bubble Sort	164,784,619,051
Tree Sort	79,342,759

Question #6:
Create a table that shows the top 3 most active functions for the bubble and tree sort programs by IR count (excluding main()) - Sort by the "self" column if you're using kcachegrind, so that you see the IR counts for *just* that function, and not include the IR count for functions that it calls.
Answer:

Array Size: 100000	**Most active functions**	**Total Instruction Read counts**
Bubble Sort	bubbleSort	164,775,225,637
	verifySort	1,900,001
	initArray	1,100,022
Tree Sort	insert_element	25,744,194
	inorder	5,199,972
	free_btree_subnodes	3,599,984

Question #7:
Create a table that shows, for the bubble and tree sort programs, the most CPU intensive line that is part of the most CPU intensive function. (i.e. First, take the most active function for a program. Second, find the annotated source code for that function. Third, look inside the whole function code - what is the most active line? If by some chance it happens to be another function, drill down one more level and repeat.)
Answer:
Assuming if statements are functions:

Array Size: 100000	**Most CPU intensive line**	**Code**
Bubble Sort	helper_functions.c:43	`array_start[j-1] = array_start[j];`
Tree Sort	your_functions.c:118	`currNode = currNode->leftnode;`

Question #8:
Show the Valgrind output file for the merge sort with the intentional memory leak. Clearly highlight the line where Valgrind identified where the block was originally allocated.
Answer:

```
==15794== Memcheck, a memory error detector
==15794== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15794== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==15794== Command: ./sorting_program merge
==15794== Parent PID: 15587
==15794==
==15794==
==15794== HEAP SUMMARY:
==15794==     in use at exit: 400,000 bytes in 1 blocks
==15794==   total heap usage: 2 allocs, 1 frees, 401,024 bytes allocated
==15794==
==15794== 400,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==15794==    at 0x483DD99: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==15794==    by 0x109344: main (sorting.c:42)
==15794==
==15794== LEAK SUMMARY:
==15794==    definitely lost: 400,000 bytes in 1 blocks
==15794==    indirectly lost: 0 bytes in 0 blocks
==15794==      possibly lost: 0 bytes in 0 blocks
==15794==    still reachable: 0 bytes in 0 blocks
==15794==         suppressed: 0 bytes in 0 blocks
==15794==
==15794== For lists of detected and suppressed errors, rerun with: -s
==15794== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Question #9:
How many bytes were leaked in the buggy program?
Answer:
400,000 bytes

Question #10:
Show the Valgrind output file for the merge sort after you fixed the intentional leak.
Answer:
==16083== Memcheck, a memory error detector
==16083== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16083== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==16083== Command: ./sorting_program merge
==16083== Parent PID: 15587
==16083==
==16083==
==16083== HEAP SUMMARY:
==16083== in use at exit: 0 bytes in 0 blocks
==16083== total heap usage: 2 allocs, 2 frees, 401,024 bytes allocated
==16083==
==16083== All heap blocks were freed -- no leaks are possible
==16083==
==16083== For lists of detected and suppressed errors, rerun with: -s
==16083== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Question #11:
Show the Valgrind output file for your tree sort.
Answer:
==16235== Memcheck, a memory error detector
==16235== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16235== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==16235== Command: ./sorting_program tree
==16235== Parent PID: 15587
==16235==
==16235==
==16235== HEAP SUMMARY:
==16235== in use at exit: 0 bytes in 0 blocks
==16235== total heap usage: 100,001 allocs, 100,001 frees, 3,201,088 bytes allocated
==16235==
==16235== All heap blocks were freed -- no leaks are possible
==16235==
==16235== For lists of detected and suppressed errors, rerun with: -s
==16235== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Question #12:
How many seconds of real, user, and system time were required to complete the amplify program execution with Lenna image (Lenna_org_1024.pgm)? Document the command used to measure this.
Answer:
Command: `time ./amplify IMAGES/Lenna_org_1024.pgm 11 1.1 2`

```
real    0m0.569s
user    0m0.532s
sys     0m0.036s
```

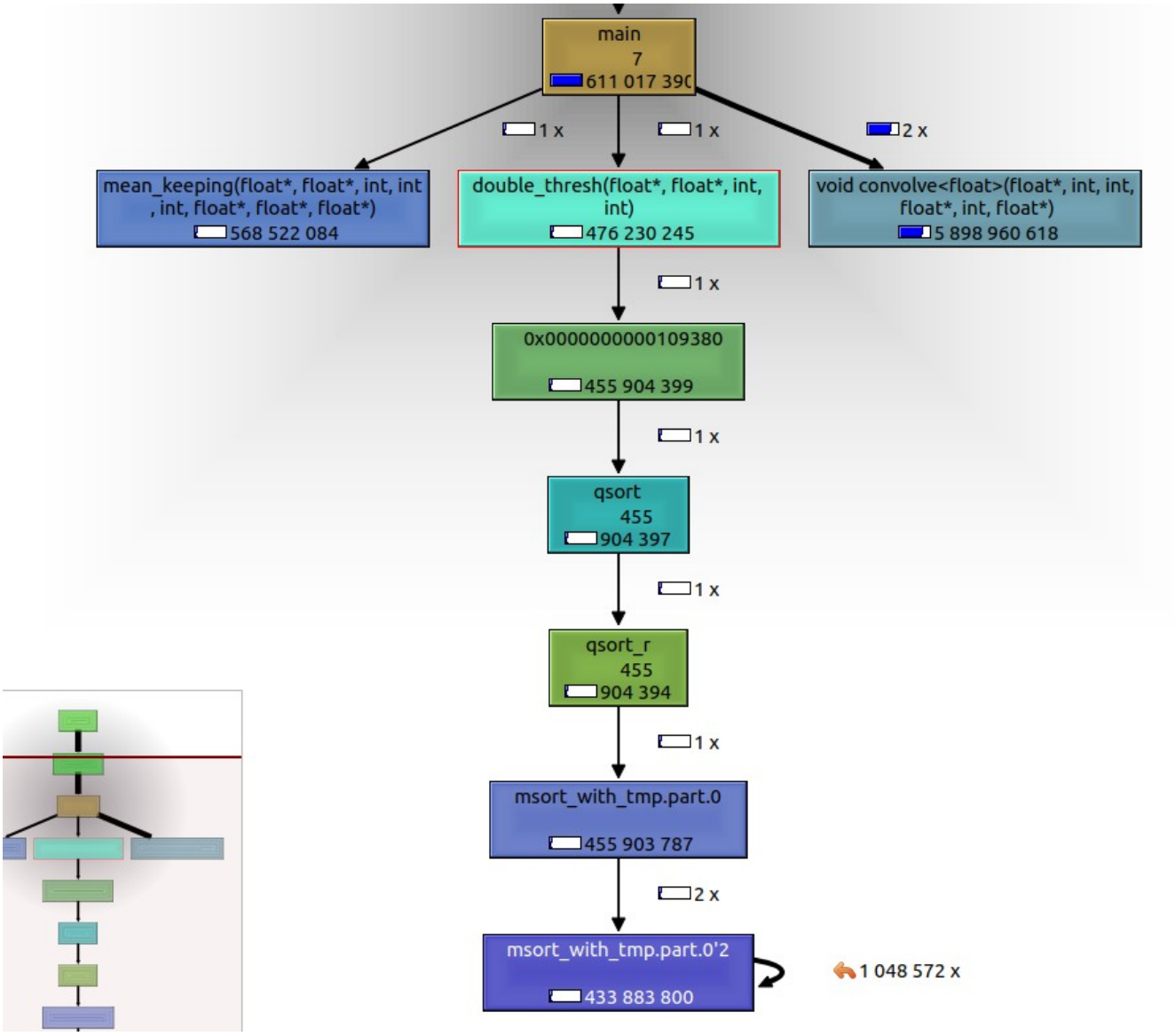
Question #13:
Research and answer: why does real time != (user time + system time)?
Answer:
When the CPU is idling, while tasks like waiting to retrieving data from the memory or storage, user and system time is not counted, but the real time is.

Question #14:
In your report, put the output image for the Lenna image titled output<somenumber>.pgm.
Answer:

Question #15:
Excluding **main()** and everything before it, what are the top three functions run by amplify executable when you do count sub-functions in the total? Document the commands used to measure this. Tip: Sorting by the "Incl" (Inclusive) column in kcachegrind should be what you want.
Answer:

Most active functions	Total Instruction Read counts
void convolve<float>(float*, int, int, float*, int, float*)	5,898,960,618
mean_keeping(float*, float*, int, int, int, float*, float*, float*)	568,522,084
double_thresh(float*, float*, int, int)	476,230,245

Question #16:
Include a screen capture that shows the kcachegrind utility displaying the callgraph for the amplify executable, starting at main(), and going down for several levels.
Answer:



Question #17:
Which function dominates the execution time? What fraction of the total execution time does this function occupy?
Answer:
void convolve<float>(float*, int, int, float*, int, float*)
77.48%

Question #18:
Does this program show any signs of memory leaks? Optional: Remove as many memory leaks as possible for 5 extra credits. Document the specific leak(s) you found and the specific code change(s) you made in your lab report.
Answer:
Yes.

```
definitely lost: 16,778,236 bytes in 8 blocks
==20642==    indirectly lost: 0 bytes in 0 blocks
==20642==      possibly lost: 33,554,432 bytes in 2 blocks
==20642==    still reachable: 0 bytes in 0 blocks
==20642==         suppressed: 0 bytes in 0 blocks
```

Fixes:

- Free patch3x3 and patchaxa created at amplify.h:24 and amplify.h:26 at the end of mean_keeping function.
  - `free(patch3x3);`
  - `free(patchaxa);`
- Free Gx_mask and Gy_mask created at main.c:60 and main.c:61 at the end of main function.
  - `free(Gx_mask);`
  - `free(Gy_mask);`
- Free org_img created at main.c:13 at the end of main function.
  - `free(org_img);`
- Free Vmap and Hmap created at main.c:18 at the end of main function.
  - `free(Vmap);`
  - `free(Hmap);`
- Free supp created at main.h:127 at the end of double_thresh function.
  - `free(supp);`
- Free HRV and HRH created at main.c:74 and main.c:75 at the end of main function.
  - `free(HRV);`
  - `free(HRH);`

Summary:

```
==22500== HEAP SUMMARY:
==22500==     in use at exit: 0 bytes in 0 blocks
==22500==   total heap usage: 89 allocs, 89 frees, 118,633,132 bytes
allocated
==22500==
==22500== All heap blocks were freed -- no leaks are possible
==22500==
==22500== Use --track-origins=yes to see where uninitialised values
come from
==22500== For lists of detected and suppressed errors, rerun with: -s
==22500== ERROR SUMMARY: 10000000 errors from 8 contexts (suppressed:
0 from 0)
```