ECE454 Lab 1

By
Shafaaf Khaled Hossain - 998891515
Sai Kiran Varikooty - 998440307

3.1 Setup

Question 1

ReadOptions - This function has a loop and so can implement loop unrolling.

SetupVPR - This functions makes certain function calls which could possibly be optimized through function inlining.

CheckOptions - This function contains multiple loops which could be optimized through loop unrolling.

ShowSetup - Has nested loops. Can possibly apply Loop Invariant Code Motion and loop unrolling to optimize code.

Place_and_route - This function has loops and so can apply loop unrolling.

3.3 Measuring Compilation Time

Question 2

Changing opt flags and using these commands to measure compilation time -
make clean
/usr/bin/time make

Compiler Options	Compilation Time (seconds)	Speedup (-O3 Baseline)
-O3	5.42	1

-O2	4.50	1.2
-Os	3.90	1.4
Gcov (-g -fprofile-arcs -ftest-coverage)	2.05	2.6
Gprof (-g -pg)	1.81	3.0
-g (Debug)	1.79	3.0

Question 3

O3 is the slowest since it includes all optimizations in O2 and adds additional optimizations including loop unrolling and more inlining.

Question 4

Debug (-g) is the fastest since it does not do any optimizations and only includes debug information.

Question 5

Gprof is faster because it only inserts code for generating call graph information and to interrupt the program. Gcov, however inserts code to analyze how many times each line is executed, which loops and which if/else statements are more important within functions etc

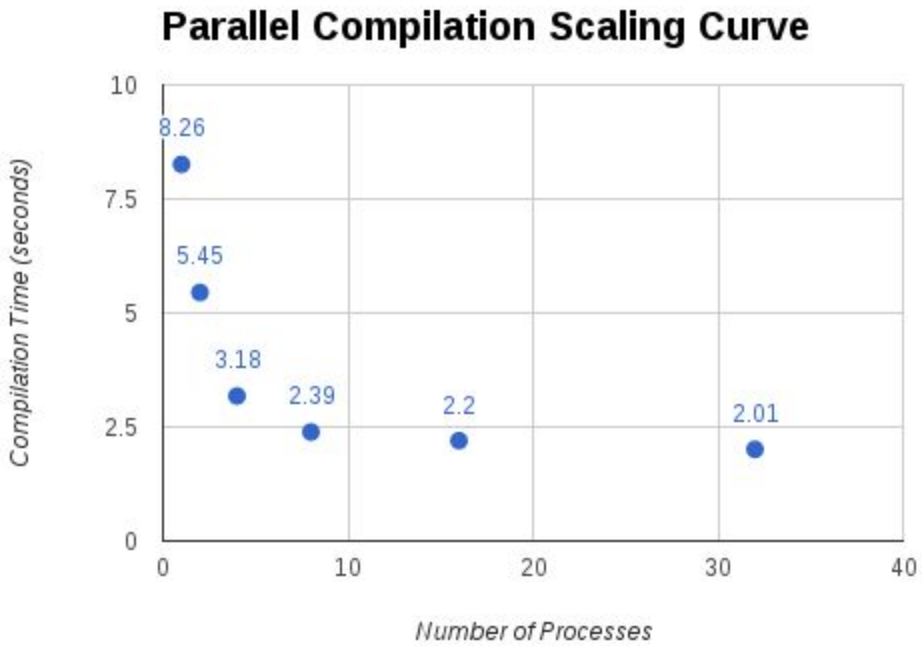
Section 3.4 Measuring Parallel Compilation Time

Question 6

e.g make -j 4

Number of Processes	Flag	Elapsed Time (seconds)	Speedup
1	-O3	8.26	1
2	-O3	5.45	1.5
4	-O3	3.18	2.6
8	-O3	2.39	3.5

16	-O3	2.20	3.8
32	-O3	2.01	4.1



As we double the number of processes in make, the elapsed time decreases by a few seconds at the beginning. However, later on as we use more than 8 processes the elapsed time stays relatively close and does not decrease as much.

3.5 Measuring Program Size

Question 7

Compiler Options	Size (Bytes)	Relative size increase
Gcov (-g -fprofile-arcs -ftest-coverage)	1014640	3.6
Gprof (-g -pg)	751184	2.7
-G	747184	2.7
-O3	379648	1.3

-O2	333952	1.2
-Os	281528	1

Question 8

Os is the smallest in size since the compiler does not do any optimizations that increase the size of the program and rather optimizes to reduce code size.

Question 9

Gcov is the largest in size since it instruments the code to add instrumentation within functions, such as in loops and if/else conditions etc. This will require more symbols etc causing the program to be bigger.

Question 10

Gprof is smaller since instrumentation code is inserted into the program code to interrupt the program at set intervals to approximate the time spent per function, cpu usage, call graph etc while gcov provides greater detail such as determining how many times lines are executed, which loops more imp etc and this leads to more greater overhead.

3.6 Measuring Performance

Run using:
Change opt flag
make clean
/usr/bin/time ./vpr iir1.map4.latren.net k4-n10.xml place.out route.out -nodisp -place_only -seed 0

Question 11

Compiler Options	Runtime Time (seconds)	Speedup
Gprof (-g -pg)	3.52	2.8
Gcov (-g -fprofile-arcs -ftest-coverage)	2.99	2.4
-g (Debug)	2.91	2.3
-Os	1.47	1.2

-O2	1.33	1.1
-O3	1.25	1

Question 12

The slowest is gprof because, the program is periodically interrupted during runtime and therefore the time program will include the time for these interrupts.

Question 13

The fastest is -O3 since it includes all optimizations enabled in O2 as well as additional optimizations including function inlining, loop unrolling etc resulting in a smaller runtime.

Question 14

Gcov is faster than gprof at runtime since the program is not periodically interrupted and just has instrumentation added in. Gprof gets interrupted over certain fixed intervals.

3.7 Profiling with gprof

Question 15

Top 5 functions (give function name and percentage execution time).

How - http://www.thegeekstuff.com/2012/08/gprof-tutorial/
They said - gprof test_gprof gmon.out > analysis.txt

Do

1. ./vpr iir1.map4.latren.net k4-n10.xml place.out route.out -nodisp -place_only -seed 0
2. gprof ./vpr gmon.out > analysis.txt

Version	Flag	Top function	2nd function	3rd function	4th function	5th function
-g	-g -pg	Comp_delta_td_cost	Comp_td_point_to_point_d	Find_affected_nets	Get_non_updateable_	Try_swap

		(17.79%)	elay (15.77%) (0.47 s)	(12.42 %)	bb (11.07%)	(10.74%)
-O2	-O2-pg	Try_swap (44.78%)	Get_non_updateable_bb (16.42 %)	Comp_td_point_to_point_delay (10.07%)	Get_net_cost (7.46%)	Get_seg_start (7.46%)
-O3	-O3-pg	Try_swap (55.12%)	Comp_td_point_to_point_delay (20.47%) (0.26 s)	Label_wire_muxes (7.87%)	Update_bb (4.72%)	Get_bb_from_scratch (4.72%)

Question 16

In the -O3 version, try_swap is the function with the greatest percentage execution time (55.12%). In the -g version, it's percentage execution time is 10.74%.

Try_swap has a higher CPU percentage in -O3 than -g, since other functions have minimal or no effect on the CPU percentage calculation as they have been optimized greatly.

Optimizations like function inlining and loop unrolling have been used by the compiler. The functions are optimized so much (possibly inlined) that they are not even detected in -O3. These include comp_delta_td_cost, find_affected_nets, get_non_updateable_bb, get_net_cost

Question 17

The number two ranked function in -O3 and -g is Comp_td_point_to_point_delay.
For -g, self seconds is 0.47s, whereas for -O3 it is 0.26s.

The compiler did not inline comp_td_point_to_point_delay, because it is not called as frequently as the other functions that were inlined.

The compiler did not do the same transformations within comp_td_point_to_point_delay as before there are no function calls for the compiler to implement function inlining to those functions. Also there are no loops here for the compiler to implement loop unrolling.

3.8 Inspecting Assembly

Question 18

Flag	Function	Instruction Count	Reduction Ratio
-g	update_bb	8792 - 8242 + 1 = 551	-g/-O3 = 551/214 =2.6
-O3	update_bb	377-164+1 = 214	

Question 19

Flag	Function	Self seconds	Speedup
-g	update_bb	0.14	-g/-O3 = 0.14/0.06 = 2.3
-O3	update_bb	0.06	

The speedup compared to the size reduction is almost the same with around a 0.3 difference. As you have lesser number of instructions, the function can be executed faster with lesser CPU cycles and so less self seconds.

3.9 Profiling with gcov

Question 20

Number-one function from the -O3 version is Try_swap (place.c: line 262).

Loops:

Optimization Priority	Times line executed	Branch Taken	Original Source File Line	Source code

1	17,855,734	95%	1377	for(k = 0; k < num_nets_affected; k++)
2	13,649,026	97%	1512	for(k = 0; k < num_nets_affected; k++)
3	4,349,990	80%	1279	for(i = 0; i < num_types; i++)
4	4,206,708	91%	1473	for(k = 0; k < num_nets_affected; k++)
5	#####	Never executed	1312	while(block[b_from].type == IO_TYPE)

Question 21 (Bonus)

The original user time of the program using -O3 using the time program is 1.45.
After inlining the comp_td_point_to_point_delay function, the user time decreases to 1.18s.
Speed up => 1.45/1.18 = 1.23

This speedup is due to the elimination of function call and return overhead.

Old

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 64.87      0.72     0.72   869998     0.00     0.00  try_swap
 19.37      0.94     0.22 25424969     0.00     0.00  comp_td_point_to_point_delay
  7.21      1.02     0.08    68548     0.00     0.00  label_wire_muxes
  2.25      1.04     0.03  3008047     0.00     0.00  update_bb
  1.35      1.06     0.02  3022684     0.00     0.00  my_irand
  0.90      1.07     0.01   315200     0.00     0.00  get_bb_from_scratch
  0.90      1.08     0.01    65868     0.00     0.00  get_unidir_track_to_chan_seg
  0.90      1.09     0.01    55176     0.00     0.00  get_track_to_tracks
```

```
-----------------------------------------------
                0.72    0.27  869998/869998      try_place [1]
[3]     88.7    0.72    0.27  869998         try_swap [3]
                0.22    0.00 25424969/25424969     comp_td_point_to_point_delay [4]
                0.03    0.00 3008047/3008047     update_bb [11]
                0.01    0.00 3022546/3022684     my_irand [12]
                0.01    0.00  315164/315200      get_bb_from_scratch [13]
                0.00    0.00  638248/638248      my_frand [22]
                0.00    0.00       4/377666      my_malloc [23]
-----------------------------------------------


Placement. Cost: 1  bb_cost: 34.3475  td_cost: 5.82187e-08  delay_cost: 4.82928e-07.
1.45user 0.00system 0:01.49elapsed 97%CPU (0avgtext+0avgdata 7652maxresident)k
0inputs+304outputs (0major+1596minor)pagefaults 0swaps
```

New with inlining comp_td_point_to_point_delay

```
Placement. Cost: 1  bb_cost: 34.3475  td_cost: 5.82187e-08  delay_cost: 4.82928e-07.
1.18user 0.00system 0:01.21elapsed 97%CPU (0avgtext+0avgdata 7720maxresident)k


Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.51      0.95     0.95   869998     0.00     0.00  try_swap
 11.02      1.08     0.13    68548     0.00     0.00  label_wire_muxes
  1.69      1.10     0.02      124     0.00     0.00  load_net_slack
  0.85      1.11     0.01  3008047     0.00     0.00  update_bb
  0.85      1.12     0.01   315200     0.00     0.00  get_bb_from_scratch
  0.85      1.13     0.01    65868     0.00     0.00  get_unidir_track_to_chan_seg
  0.85      1.14     0.01    15972     0.00     0.00  get_track_to_ipins
  0.85      1.15     0.01        1     0.01     0.01  check_rr_graph


-----------------------------------------------
                0.95    0.02  869998/869998      try_place [1]
[3]     82.2    0.95    0.02  869998         try_swap [3]
                0.01    0.00 3008047/3008047     update_bb [13]
                0.01    0.00  315164/315200      get_bb_from_scratch [14]
                0.00    0.00 3022546/3022684     my_irand [21]
                0.00    0.00  638248/638248      my_frand [22]
                0.00    0.00       4/377666      my_malloc [23]
-----------------------------------------------
```

Extra notes

Functions that have been inlined do not show up in the gprof output.

The fidelity of the self-seconds column isn't very high, and as a result, there can be many non-inlined functions that show up in the gprof output with 0 self seconds despite being called and executing for a finite period of time.