INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CS 622A Semester 2020–2021-I

Assignment 2

GROUP-17

Members
Mayank Bansal
Roll-20111032

Hirak Mondal
Roll-20111022

Professor:
Dr. Mainak Chaudhury
Department of Computer Science

INTRODUCTION

In this assignment we have conducted some studies to understand the reuse and sharing profiles of a set of parallel program. We have instrumented these shared memory parallel programs using PIN and capture the per-thread memory access traces. Next, we have written a couple of analysis codes (outside PIN) to understand the reuse and sharing behavior.

General Instructions for Compiling and Running the Codes

Step 1: Download Pin 3.15 and set up.

Step 2: Paste the Directory named **CS622** that is being provided by us, under "pin-3.15-98253-gb56e429b1-gcc-linux/source/tools/"

Step 3: Set the path of the terminal to this folder using the **cd** command. If your "pin-3.15-98253-gb56e429b1-gcc-linux"file is at the HOME directory then you can use the command →
cd pin-3.15-98253-gb56e429b1-gcc-linux/source/tools/CS622

PART I: Collection of traces

Code provided for this part: A file named **trace.cpp** is provided inside the directory named **"CS622"** to record the total number of machine accesses along with its nature whether read/write as well as the thread number.

Compilation and Run-time Instruction

• Compile the program whose machine access you want to compute. The command for that is → **gcc -O3 -static -pthread filename.c -o filename.**

• Then enter the command → **make obj-intel64/trace.so**

• Then finally run it using →

../../../pin -t obj-intel64/trace.so – ./filename number_of_threads

• To know the line count, we can use → **wc -l trace.out**

Output :It outputs a file named **trace.out** which contains the **thread_id**, **Read or Write information** and **Address of Machine Accesses** and **wc -l trace.out** gives us the total number of machine accesses

TOTAL NUMBER OF MACHINE ACCESS RECORDED

Program Name	Number of Machine Accesses
prog1.c	140525561
prog2.c	2513897
prog3.c	9465464
prog4.c	1064799

PART II: ACCESS DISTANCE ANALYSIS

Code provided for this part: Two files named **trace1.cpp** and **cdf.cpp** is provided inside the directory named **"CS622"**

trace1.cpp produces a **.out** file with only the machine accesses excluding the thread number and Read or Write Information and **cdf.cpp** generates X and Y axis, where the X axis represents distance in log base 10 scale and the Y axis represents the CDF $F(d) = \frac{n}{N}$ as mentioned in the question

Compilation and Run-time Instruction

• Compile the program whose Access Distances Analysis you want to perform. The command for that is → **gcc -O3 -static -pthread filename.c -o filename.**

• Then enter the command → **make obj-intel64/tracel1.so**

• Then finally run it using →

../../../pin -t obj-intel64/trace1.so – ./filename number_of_threads

This will generate a .out file named **trace1.out**, which will be input to our

cdf.cpp file

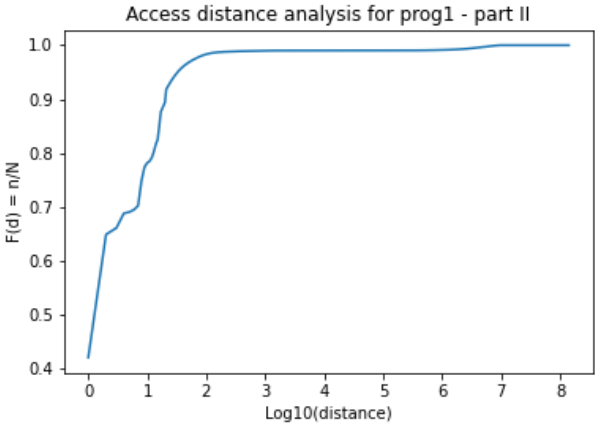
- Compile this cpp file using → **g++ cdf.cpp**

- Run it using → **./a.out**

Output :It outputs a file named **cdf.csv** which contains X and Y axis, where the X axis represents distance in log base 10 scale and the Y axis represents the CDF $F(d) = \frac{n}{N}$ as mentioned in the question

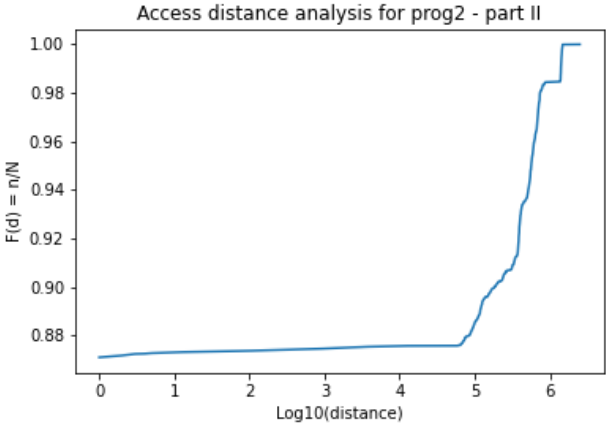
CUMULATIVE DENSITY FUNCTION PLOTS

Program 1



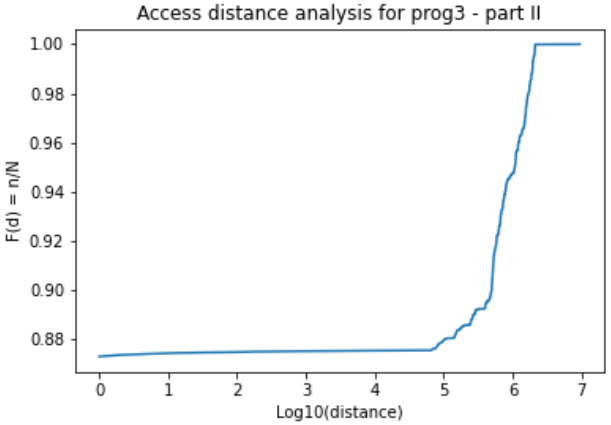
Y-axis: 0.4 to 1.0 with an interval of 0.1 unit
X-axis: 0 to 8 with an interval of 1 unit

Program 2



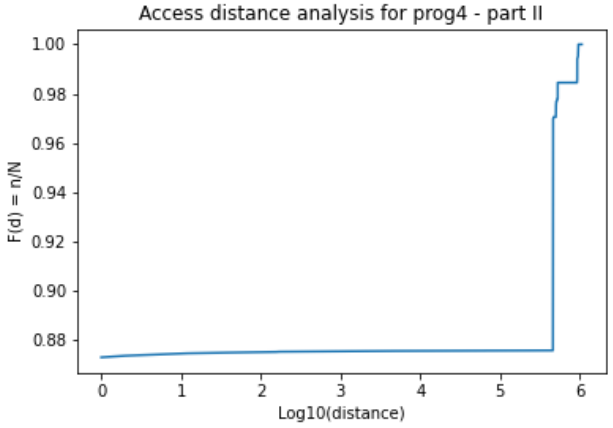
Y-axis: 0.88 to 1.0 with an interval of 0.02 unit
X-axis: 0 to 6 with an interval of 1 unit

Program 3



Y-axis: 0.88 to 1.0 with an interval of 0.02 unit
X-axis: 0 to 7 with an interval of 1 unit

Program 4



Y-axis: 0.88 to 1.0 with an interval of 0.02 unit
X-axis: 0 to 6 with an interval of 1 unit

Explanation

Notice that our graphs doesn't start from 0 in Y axis, but it starts from the cdf obtained for a distance of 1, (i.e. 0 on the log-to-the-base-ten scale on the distance axis (x-axis)). Even for a small distance like 1 (which is represented as 0 in log-to-the-base-ten scale on the distance axis), we are getting a very high Cumulative Density of around 0.88 for prog2.cpp,prog3.cpp and prog4.cpp after which the CDF curve remains more or less the same till around 5.5 or so in X axis. Which means that the memory accesses sharing the same block and having an access distance of 1 is MOST common than having other access distances. There is a rise in the curve after around 5.5 or so in the x axis, which means that after an access distance of 1(which is the most common) the more common access distances are again starting from around $10^{5.5}$.For prog1.cpp the case is slightly different, for prog1 the most frequent access distances are from 1 to 10^2(0 to 2 in log-to-the-base-ten scale on the distance axis), as we can see in the graph that there is a steep rise in cdf from 0 to 2(for x axis) after which it flattens and continues more or less at a same level through out the graph, increasing the slope at a very minute level with respect to the X axis.

From this we know that, most of the machine accesses have very less access distance and thus if we can introduce a cache they will all be hits since it will exploit the

temporal locality and spatial locality thus reducing the run-time and increasing the overall performance. This is the idea that we will be implementing in part 3, by introducing a cache.

Part III: Access distance filtered by LRU cache

In this question we have modeled a **single-level 2 MB 16-way cache** with **64-byte** block size and **LRU replacement policy**. We have passed each trace through the cache and collect the trace of accesses that miss in the cache. Then we have reported the number of hits and misses for each trace and also prepared the cumulative density function of the miss trace for each of the programs.

Code provided for this part: Two main files named **cache.py** and **cdf3.cpp** is provided inside the directory named **CS622** Along with cache.py we have provided 2 more files named **node.py** and **utility.py** which are being called by **cache.py** itself. **cache.py** produces a **.out** file namde **trace_misses.out** with documented misses on which the **cdf3.cpp** is run which generates X and Y axis, where the X axis represents distance in log base 10 scale and the Y axis represents the CDF $F(d) = \frac{n}{N}$ as mentioned in the question

Compilation and Run-time Instruction

• Run the python file **cache.py**, using this command → **python3 cache.py**

This will calculate the total number of hits and misses from the **trace.out** file, which we have generated in question 1. Along with this it will also produce a **trace_misses.out** file containing all the misses

• Then compile the cdf3.cpp using the command → **g++ cdf3.cpp**

• And then run it using → **./a.out**

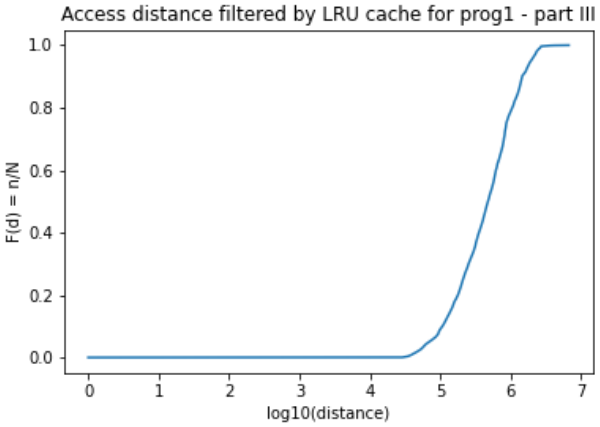
This will generate **cdf3.csv** file which contains all the X and Y axis, where the X axis represents distance in log base 10 scale and the Y axis represents the CDF $F(d) = \frac{n}{N}$ as mentioned in the question

Total Number of Hits and Misses

Program Name	Hits	Miss
prog1.c	133873599	6651950
prog2.c	2360839	185225
prog3.c	9010711	586612
prog4.c	933107	131674

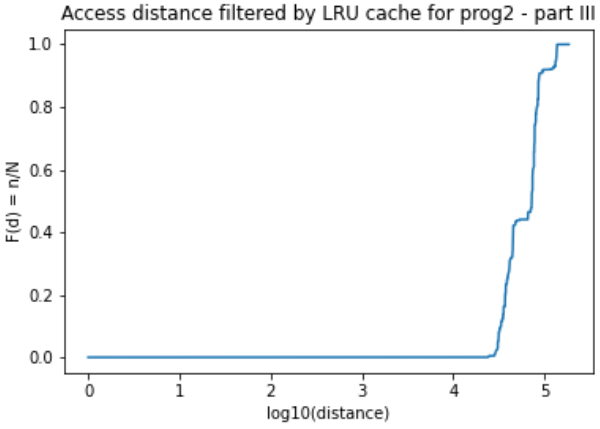
CUMULATIVE DENSITY FUNCTION PLOTS

Program 1



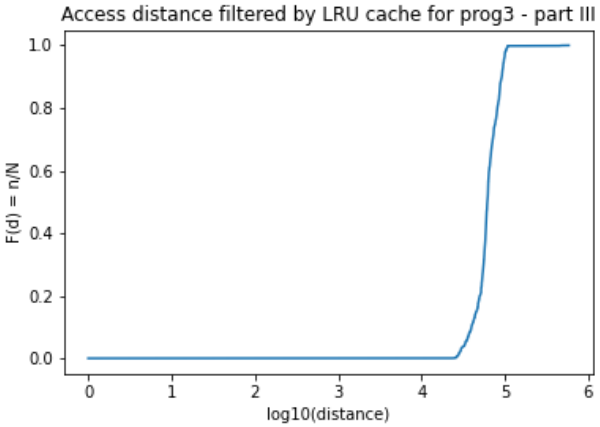
Y-axis: 0.0 to 1.0 with an interval of 0.2 unit
X-axis: 0 to 7 with an interval of 1 unit

Program 2



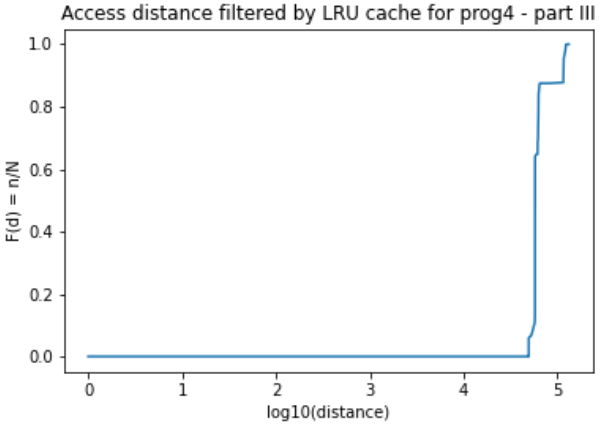
Y-axis: 0.0 to 1.0 with an interval of 0.2 unit
X-axis: 0 to 5 with an interval of 1 unit

Program 3



Y-axis: 0.0 to 1.0 with an interval of 0.2 unit
X-axis: 0 to 6 with an interval of 1 unit

Program 4



Y-axis: 0.0 to 1.0 with an interval of 0.2 unit
X-axis: 0 to 5 with an interval of 1 unit

Difference between Cumulative Density Function Before and After The Cache

Notice that the graph in question 3 is starting from 0 in Y axis, unlike 0.88 for prog2,3,4 in question 2 and 0.4 for prog1 in question 2. We are reaching Y=0.88 in the graphs of question 3 at a very later stage when X is around 4.5 or so. The explanation for which is given below

Explanation

This question has a direct link with question 2. In question 2 we have observed how memory accesses sharing the same block are accessed frequently and most of them posses an access distance of 1. From it, we have got the idea that if we introduce a cache, all those frequent memory accesses which are sharing the same block will be turned to HITS. This is what we have implemented in question 3 and in the graphs we have plotted the CDF of Misses. Each and every graph share more or less the same characteristics. We can see that for a distance less than 10^4 (i.e. 4 on the log-to-the-base-ten scale on the distance axis (x-axis)) we are getting absolutely no misses. This means that we are exploiting the **temporal locality** as well as the **spatial locality** and thus all these accesses having distance less than what mentioned above is turning out to be a HIT. After that only, from a distance of $10^{4.5}$ (i.e. 4.5 on the log-to-the-base-ten scale on the distance axis (x-axis)) or so, we are experiencing misses. This can be mainly due to 2 reasons, either the machine accesses in those regions of code accesses entirely a new block, or the blocks that they are trying to access is now removed from the cache. But overall if we want to look, we can see that we have turned all the frequent accesses to the same blocks to HITS.

How Graphs For Question 2 and Question 3 are Plotted

We have provided 2 python files named **Graph2.py** and **Graph3.py** inside the directory **CS622**. "Graph2.py"takes **cdf.csv** as input and "Graph3.py"takes **cdf3.csv** as input and produces graphs for question 2 and 3 respectively.

Note that **pandas**, **numpy**, **matplotlib** must be installed to run the Graph codes and to be able to visualize the Graphs.

Command to run is → **python3 filename.py**

Part IV: Sharing profile

In this part, we have made use of the thread id to derive the sharing profile. We have written a program to compute the number of memory blocks that are private, or shared by two threads, or shared by three threads, ..., or shared by eight threads. The sum of these eight are equal to the total number of memory blocks for each trace. We have reported these eight numbers for each trace in a table. The LRU cache of PART III has been removed in this part.Note that a memory block is said to be private if it is accessed by exactly one thread in the entire trace. A memory block is said to be shared by exactly n threads if it is accessed by exactly n distinct threads in the trace.

Code provided for this part: A main file named **sharing_profile.py** is provided inside the directory named **CS622** along with **sharing_profile.py** we have provided 1 more file named **utility_sharing_profile.py** which is being called by **sharing_profile.py** itself.

Compilation and Run-time Instruction

• Run the python file **sharing_profile.py**, using this command →
python3 sharing_profile.py

This will calculate the total number of memory blocks, number of private memory blocks and the number of memory blocks shared by 2 to 8 threads

OBSERVATIONS

Analysis For prog1.c

Total number of memory blocks in file is → **721404**
Number of private memory blocks are → **426**

Number of Threads	Number of Machine Blocks Shared
Two	70
Three	1872
Four	32455
Five	143250
Six	244970
Seven	173832
Eight	124529

Analysis For prog2.c

Total number of memory blocks in file is → **66039**
Number of private memory blocks are → **420**

Number of Threads	Number of Machine Blocks Shared
Two	8262
Three	16384
Four	40957
Five	4
Six	0
Seven	1
Eight	11

Analysis For prog3.c

Total number of memory blocks in file is → **66036**
Number of private memory blocks are → **425**

Number of Threads	Number of Machine Blocks Shared
Two	63
Three	0
Four	0
Five	0
Six	0
Seven	1
Eight	65547

Analysis For prog4.c

Total number of memory blocks in file is → **66039**
Number of private memory blocks are → **8611**

Number of Threads	Number of Machine Blocks Shared
Two	57409
Three	6
Four	0
Five	0
Six	0
Seven	1
Eight	12