DV1567: Performance Optimization

Lab Assignment

Name	Monica Gattupalli
Student Email-Id	moga20@student.bth.se
National ID	991130-5002
Group Number	23

Task 1: Information about the SUT

Introduction

In this task our goal is to get the more information about the system under test (SUT). Ubuntu is the system which is under test. The information of "CPU", "memory", "USB controllers" etc.

1)Linux commands used to get information about SUT.

Commands:

- **Sudo lshw:** This command is used to get the hardware details of the system like number of bits, size of the memory, product name, CPU capacity, different network configuration. This command is only executed when you use sudo command (that means only the root user can execute the command).

Figure 1: output of "sudo lshw".

- **Lscpu:** This command is used to get the information about the architecture, CPU op-mode, address sizes, no. of CPUs, hypervisor vendor, virtualization type, model, model name, thread, core, stepping.

Figure 2: output of "lscpu"

- **Free:** this command is used to get the details of RAM, used, free, total amount of memory available memory and swap memory.

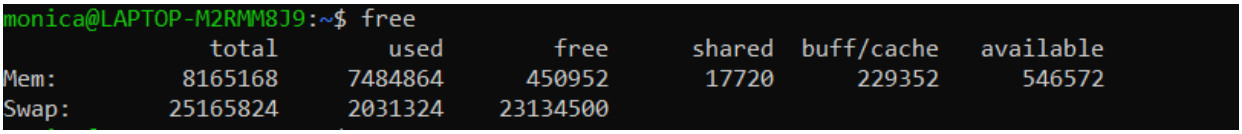
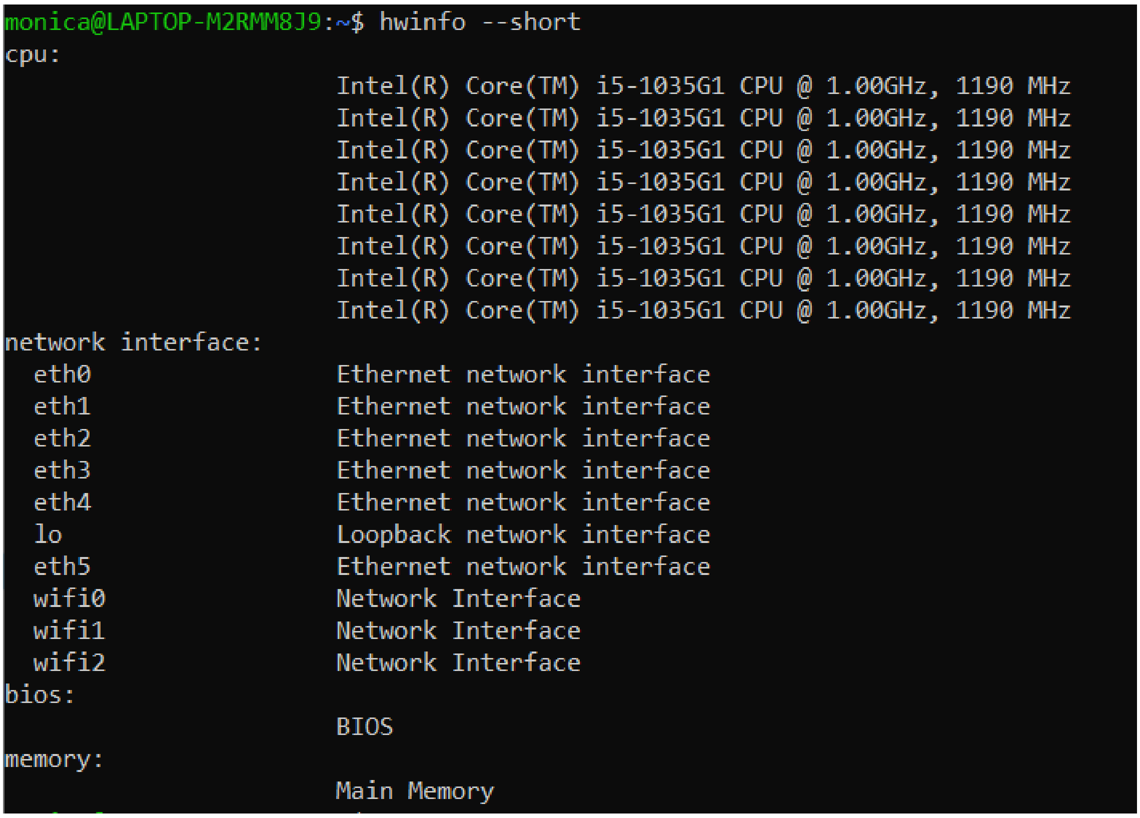


Figure 3: output of "free"

- **hwinfo –short:** this command is used to get the details of CPUs, network interface, memory, bios.



```
monica@LAPTOP-M2RMM8J9:~$ hwinfo --short
cpu:
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
                       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz, 1190 MHz
network interface:
  eth0                 Ethernet network interface
  eth1                 Ethernet network interface
  eth2                 Ethernet network interface
  eth3                 Ethernet network interface
  eth4                 Ethernet network interface
  lo                   Loopback network interface
  eth5                 Ethernet network interface
  wifi0                Network Interface
  wifi1                Network Interface
  wifi2                Network Interface
bios:
                       BIOS
memory:
                       Main Memory
```

Figure 4: output of “hwinfo --short”

2) Characteristics of better performance and scalability

- CPU (Control Processing Unit).
- GPU (Ram), Coolant
- Operating system [Linux operating system has better CI/CD]
- Storage
- Cache memory
- Network speed [16-20 mbps, with GPU 4GB].

3) Description about your SUT (system under test)

The below table represents the complete technical information of SUT

Model Name	Intel® Core™ i5-1035G1 CPU @ 1.00GHz
Architecture	X86_64
CPU op – mode(s)	32 – bits, 64 – bits
Address sizes	36 bits physical, 48 bits virtual
CPU(s)	8
Hypervisor vendor	Windows Subsystem for Linux
Virtualization type	Container
Memory	7973 MB

Task 2(a): General performance monitoring

In this task our goal is to use the command “sar”, “top”, “ps” to monitor the system performance, the state of the processes running on the system.

Sar (System Activity Report)	This command is used to get the information about the CPU, memory, disk, system load, network. In sar command, data collection should be enabled(enabled=true) to get the information. later on, the system should be restarted.
Top (Table of Processes)	This command is used to get the information of CPU and memory utilization.
Ps (Process Status)	This command is used to get the information about the current processes.

1)Performance metrics/counter that can use to detect memory leak

- Check the Ram usage
- Check total amount of memory used.
- Number of page read/write per second.

1.1) Commands and related flag to detect the memory leaks

- **Sar -r:** This sar command is used to get the information related to memory in the below output we can observe

```
monica@LAPTOP-M2RMM8J9:~$ sar -r
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/16/22        _x86_64_        (8 CPU)

20:02:49     LINUX RESTART      (8 CPU)
```

Figure 1: output of "sar -r command".

- **Sar -r 3:** In this command "3" indicates seconds, we can observe that for every 3 seconds the sar -r command is executed infinity until "Ctrl+Z" is pressed.

```
monica@LAPTOP-M2RMM8J9:~$ sar -r 3
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/16/22        _x86_64_        (8 CPU)

20:03:19    kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
20:03:22      1402612         0   6526080     79.93     34032    188576   3450064     10.35    167556    157876         0
20:03:25      1401788         0   6526904     79.94     34032    188576   3450064     10.35    167556    157876         0
20:03:28      1386680         0   6542012     80.12     34032    188576   3450064     10.35    167556    157876         0
20:03:31      1379704         0   6548988     80.21     34032    188576   3450064     10.35    167556    157876         0
20:03:34      1370284         0   6558408     80.32     34032    188576   3450064     10.35    167556    157876         0
20:03:37      1409792         0   6518900     79.84     34032    188576   3450064     10.35    167556    157876         0
^Z
[2]+  Stopped                 sar -r 3
```

Figure 2: output of "sar -r 3 command".

- **Sar -A:** This command gives the complete information of the system in the below picture we can observe.

```
monica@LAPTOP-M2RMM8J9:~$ sar -A
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/07/22        _x86_64_        (8 CPU)

19:05:37     LINUX RESTART      (8 CPU)
monica@LAPTOP-M2RMM8J9:~$ sar -r
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/07/22        _x86_64_        (8 CPU)

19:05:37     LINUX RESTART      (8 CPU)
```

Figure 3: output of "sar -A command".

- **Top:** This command gives the information of the memory and CPU. When you execute this command, the output is display on an interactive shell

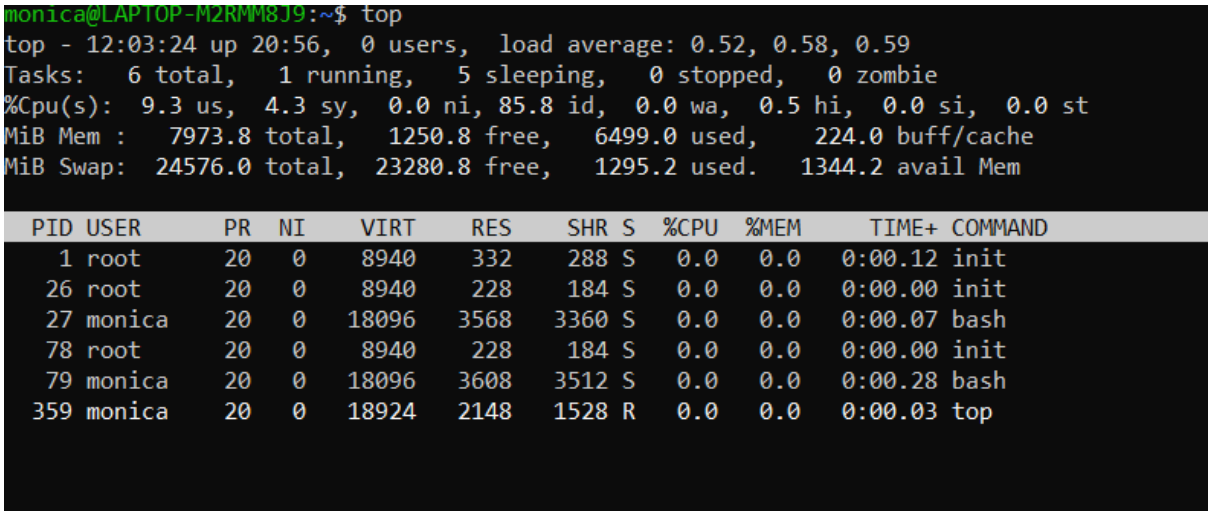


Figure 4: output of "top command".

2)Performance metrics/counter that are used to detect CPU utilization

- Processor time
- User time
- Privilege time
- Interrupt time

2.1) commands and flag related to CPU utilization

- **Sar -u:** This command gives the details of amount of CPU usage of all the CPUs in the system. In the below figure we can observe some of the features of CPU which are related to system under test.

Linux kernel version	Linux 4.4.0-19041-Microsoft.
Host name	LAPTOP-M2RMM8J9
Date	09/16/2022
System Architecture	_X86_64_
No. of. CPU	8 CPU

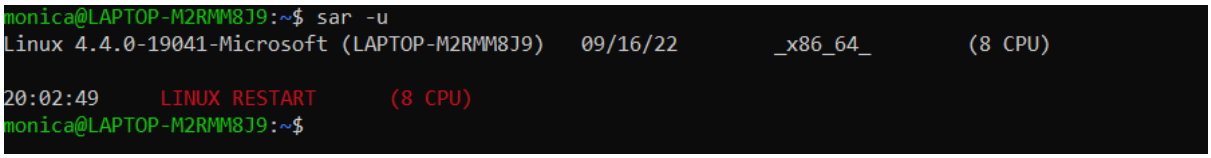


Figure 5: output of "sar -u output"

- **Sar -u 5:** In this command "5" indicates seconds, we can observe that for every 5 seconds the sar -r command is executed infinity until "Ctrl+Z" is pressed. In the below Figure6 we can observe the time, % of users and all the CPU are used, we can also Average usage of CPU's, %of users.

```
monica@LAPTOP-M2RMM8J9:~$ sar -u 5
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/07/22        _x86_64_        (8 CPU)

20:04:42        CPU     %user     %nice   %system   %iowait    %steal     %idle
20:04:47        all     19.40      0.00      3.16      0.00      0.00     77.43
20:04:52        all     24.98      0.00      4.27      0.00      0.00     70.76
20:04:57        all     20.65      0.00      6.14      0.00      0.00     73.21
20:05:02        all     18.61      0.00      4.28      0.00      0.00     77.11
20:05:07        all     21.61      0.00      6.13      0.00      0.00     72.26
20:05:12        all     20.13      0.00      4.24      0.00      0.00     75.63
20:05:17        all     19.62      0.00      4.87      0.00      0.00     75.52
20:05:22        all     25.03      0.00      4.89      0.00      0.00     70.08
20:05:27        all     24.08      0.00      5.67      0.00      0.00     70.25
20:05:32        all     23.50      0.00      3.98      0.00      0.00     72.52
20:05:37        all     23.73      0.00      4.21      0.00      0.00     72.06
20:05:42        all     23.28      0.00      3.20      0.00      0.00     73.52
20:05:47        all     23.08      0.00      5.48      0.00      0.00     71.44
20:05:52        all     26.01      0.00      4.51      0.00      0.00     69.48
20:05:57        all     21.86      0.00      6.32      0.00      0.00     71.82
20:06:02        all     21.79      0.00      4.32      0.00      0.00     73.89
20:06:07        all     36.75      0.00     18.08      0.00      0.00     45.17
20:06:12        all     38.79      0.00     20.24      0.00      0.00     40.97
20:06:17        all     23.91      0.00      6.67      0.00      0.00     69.43
20:06:22        all     29.27      0.00      7.55      0.00      0.00     63.19
^C

Average:        all     24.31      0.00      6.41      0.00      0.00     69.28
```

Figure 6: output of "sar -u 5 command".

3)performance metrics/counter that can be used to detect the disk throughput.

- Current disk queue length and Average disk queue length
- Idle length
- Average disk sec/read and Average disk sec/write.
- Disk reads/sec and Disk writes/sec.

3.1) commands and flags related to disk throughput.

- **Sar -b:** This command is used to get the information related to Input/Output statistics.

```
monica@LAPTOP-M2RMM8J9:~$ sar -b
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/07/22        _x86_64_        (8 CPU)

19:05:37      LINUX RESTART      (8 CPU)
```

Figure7: output of "sar -b command"

- **Sar -b 5:** In this command "5" indicates seconds, we can observe that for every 5 seconds the sar -r command is executed infinity until "Ctrl+Z" is pressed. In the below Figure8 we can observe the transaction per second, read and write transaction per second, byte read/ byte writes per second.

```
monica@LAPTOP-M2RMM8J9:~$ sar -b 5
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/17/22        _x86_64_        (8 CPU)

20:45:38          tps      rtps      wtps      dtps   bread/s   bwrtn/s   bdscd/s
20:45:43         0.00      0.00      0.00      0.00      0.00      0.00      0.00
20:45:48         0.00      0.00      0.00      0.00      0.00      0.00      0.00
20:45:53         0.00      0.00      0.00      0.00      0.00      0.00      0.00
20:45:58         0.00      0.00      0.00      0.00      0.00      0.00      0.00
20:46:03         0.00      0.00      0.00      0.00      0.00      0.00      0.00
20:46:08         0.00      0.00      0.00      0.00      0.00      0.00      0.00
20:46:13         0.00      0.00      0.00      0.00      0.00      0.00      0.00
^Z
[2]+  Stopped                 sar -b 5
monica@LAPTOP-M2RMM8J9:~$
```

Figure8: output of "sar -b 5 command"

4) Sample performance counter for each 2 seconds for all the 3 seconds.

- **sar 2:** This command is used to get the details for every 2 seconds.

```
monica@LAPTOP-M2RMM8J9:~$ sar 2
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/08/22        _x86_64_        (8 CPU)

11:12:16        CPU     %user     %nice   %system   %iowait    %steal     %idle
11:12:18        all     13.97      0.00      2.50      0.00      0.00     83.53
11:12:20        all      2.49      0.00      2.06      0.00      0.00     95.45
11:12:22        all      2.82      0.00      2.63      0.00      0.00     94.55
11:12:24        all      3.36      0.00      2.86      0.00      0.00     93.78
11:12:26        all     15.11      0.00      3.73      0.00      0.00     81.16
11:12:28        all      3.43      0.00      3.18      0.00      0.00     93.39
11:12:30        all      3.05      0.00      1.62      0.00      0.00     95.34
11:12:32        all      3.54      0.00      3.30      0.00      0.00     93.16
^C

Average:        all      5.97      0.00      2.73      0.00      0.00     91.29
```

Figure9: output of "sar 2 command".

- **Sar -u 2:** This command gives the details of CPU utilization for every 2 seconds until it is stopped.

```
monica@LAPTOP-M2RMM8J9:~$ sar -u 2
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/16/22        _x86_64_        (8 CPU)

20:20:43        CPU     %user     %nice   %system   %iowait    %steal     %idle
20:20:45        all      6.80      0.00      3.18      0.00      0.00     90.01
20:20:47        all     10.86      0.00      3.70      0.00      0.00     85.43
20:20:49        all     11.10      0.00      2.76      0.00      0.00     86.14
20:20:51        all      7.94      0.00      2.63      0.00      0.00     89.43
20:20:53        all      8.49      0.00      1.56      0.00      0.00     89.94
20:20:55        all      7.84      0.00      3.24      0.00      0.00     88.92
20:20:57        all      7.81      0.00      3.35      0.00      0.00     88.84
20:20:59        all      8.69      0.00      4.41      0.00      0.00     86.90
20:21:01        all      6.18      0.00      3.12      0.00      0.00     90.70
20:21:03        all      6.64      0.00      2.44      0.00      0.00     90.92
20:21:05        all     11.02      0.00      5.54      0.00      0.00     83.44
^Z
[5]+  Stopped                 sar -u 2
monica@LAPTOP-M2RMM8J9:~$
```

Figure 10: output of "sar -u 5 command".

- **top -d 2:** This command is used to get the details for every 2 seconds.

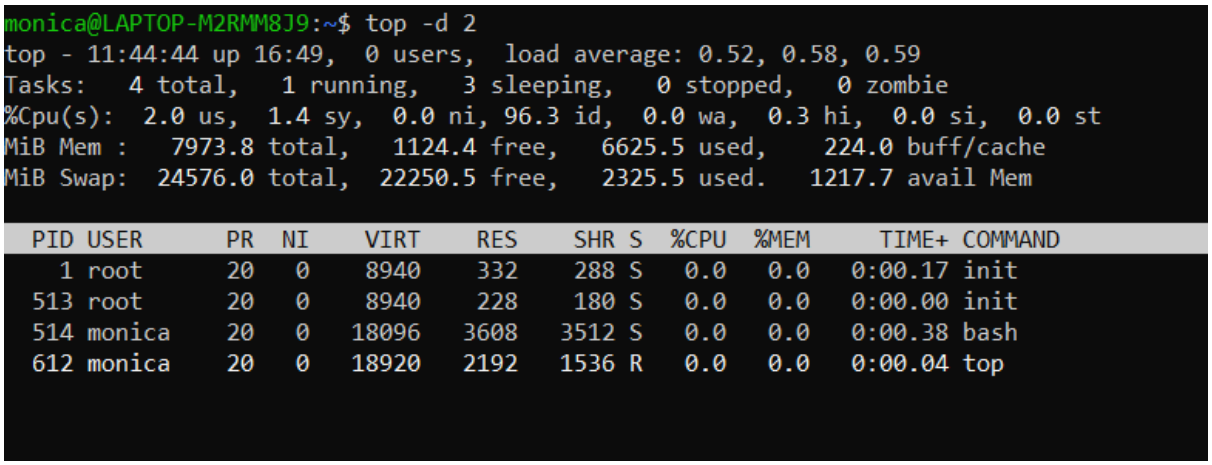


Figure 11: output of "top -d 2 command".

- **Watch -n 2 top:** Watch command is used to execute the command at regular intervals and -n should be given as flag to get the number of seconds and top is given as last parameter.

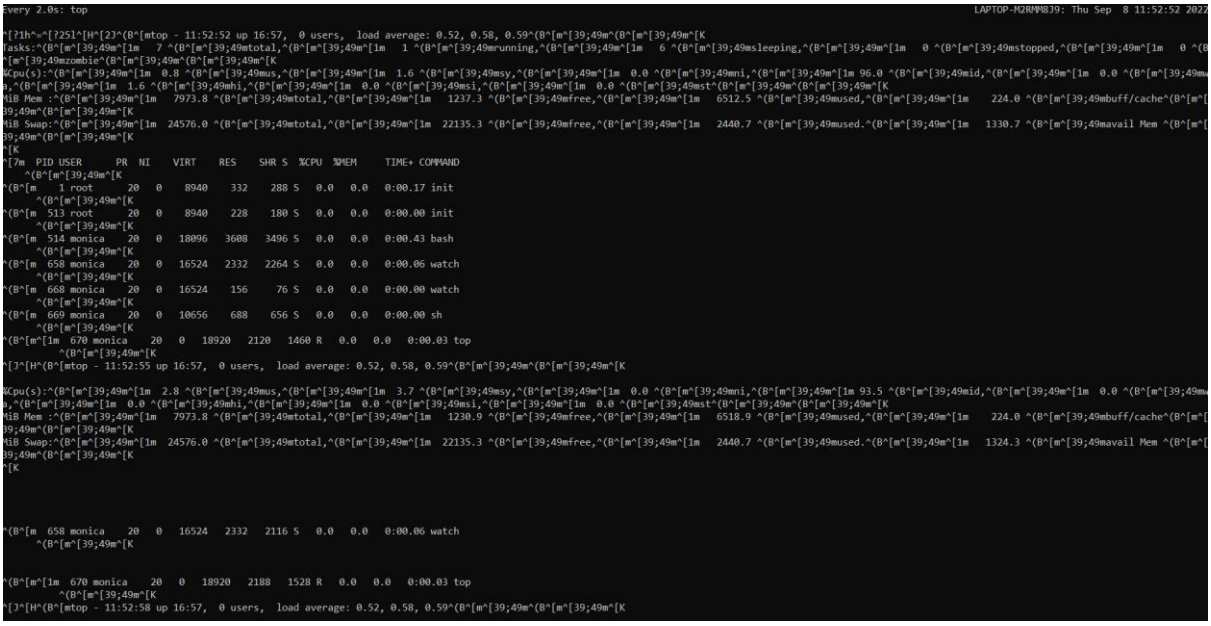


Figure 12: output of "Watch -n 2 top command".

- **Watch -n 2 ps:** Watch command is used to execute the command at regular intervals and -n should be given as flag to get the number of seconds and ps is given as last parameter.

```
Every 2.0s: ps

  PID TTY          TIME CMD
  514 tty1     00:00:00 bash
  692 tty1     00:00:00 watch
  705 tty1     00:00:00 watch
  706 tty1     00:00:00 sh
  707 tty1     00:00:00 ps
```

Figure13: output of “watch -n 2 command”.

5) Automatically stop the sampling after 3 minutes.

- **Sar 1 180:** this command will the output of the sar command from 1 second to 180 second (3 minutes).

```
monica@LAPTOP-M2RMM8J9:~$ sar 1 180
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/08/22        _x86_64_        (8 CPU)

11:16:00        CPU     %user     %nice   %system   %iowait    %steal     %idle
11:16:01        all      7.20      0.00      5.58      0.00      0.00     87.22
11:16:02        all      5.20      0.00      9.41      0.00      0.00     85.40
11:16:03        all      5.72      0.00      5.85      0.00      0.00     88.43
11:16:04        all      5.60      0.00      4.98      0.00      0.00     89.41
11:16:05        all     12.05      0.00     17.02      0.00      0.00     70.93
11:16:06        all     12.66      0.00     15.01      0.00      0.00     72.33
11:16:07        all      5.57      0.00      7.18      0.00      0.00     87.25
11:16:08        all      9.63      0.00      8.64      0.00      0.00     81.73
11:16:09        all      5.21      0.00      7.69      0.00      0.00     87.10
11:16:10        all      5.38      0.00      8.12      0.00      0.00     86.50
11:16:11        all     13.75      0.00      9.88      0.00      0.00     76.38
11:16:12        all      8.47      0.00      7.97      0.00      0.00     83.56
11:16:13        all      6.27      0.00      6.03      0.00      0.00     87.70
```

Figure 14: output of “sar 1 180” start at 11:16:00.

```
11:18:53        all      2.62      0.00      1.62      0.00      0.00     95.77
11:18:54        all      1.63      0.00      4.51      0.00      0.00     93.86
11:18:55        all      2.49      0.00      2.24      0.00      0.00     95.27
11:18:56        all      2.88      0.00      3.75      0.00      0.00     93.38
11:18:57        all      5.59      0.00      3.23      0.00      0.00     91.18
11:18:58        all      1.37      0.00      1.37      0.00      0.00     97.25
11:18:59        all      6.25      0.00      2.12      0.00      0.00     91.62
11:19:00        all     10.15      0.00      8.29      0.00      0.00     81.56
Average:        all      7.71      0.00      6.49      0.00      0.00     85.81
```

Figure 15: output of “sar 1 180” ends at 11:19:00.

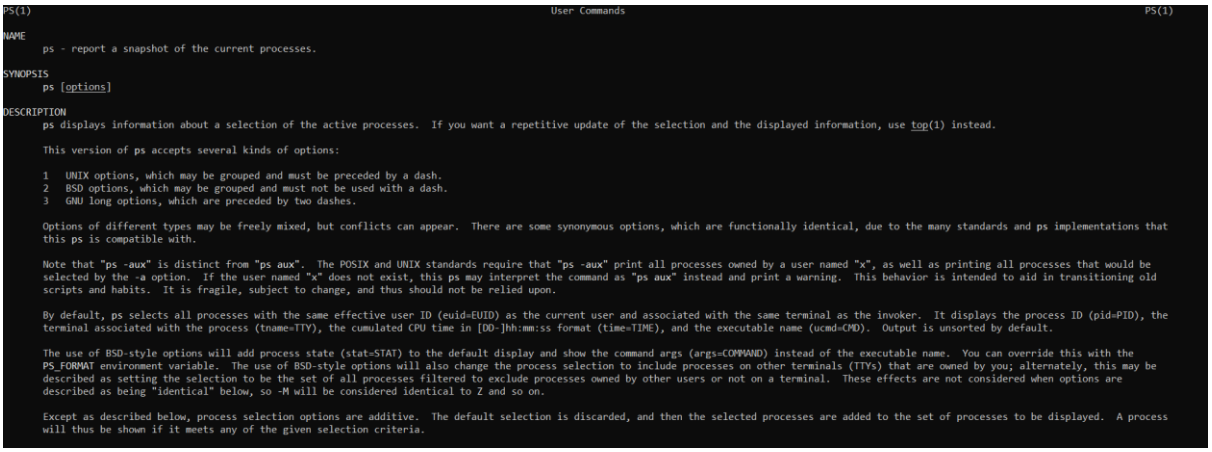
- **top -d 1 -n 180:** This command gives the output of top command for 3 minutes. After 3 minutes the execution is completed without any external command.

```
Tasks:  22 total,   1 running,   3 sleeping,  13 stopped,   5 zombie
%Cpu(s):  6.2 us,  2.0 sy,  0.0 ni, 91.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7973.8 total,   2248.1 free,   5501.7 used,    224.0 buff/cache
MiB Swap:  24576.0 total,  24238.5 free,    337.5 used.   2341.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0    8940    328    284 S   0.0   0.0   0:00.06 init
   11 root      20   0    8940    224    180 S   0.0   0.0   0:00.01 init
   12 monica    20   0   18096   3604   3496 S   0.0   0.0   0:00.23 bash
  113 monica    20   0   29592   7040   6788 T   0.0   0.1   0:00.17 vi
  134 monica    20   0   13892    964    884 T   0.0   0.0   0:00.04 sar
  135 monica    20   0   15344   1204    944 T   0.0   0.0   0:00.00 sadc
  137 monica    20   0   18920   1996   1552 T   0.0   0.0   0:00.03 top
  138 monica    20   0   15988   1768   1588 T   0.0   0.0   0:00.04 watch
  145 monica    20   0       0      0      0 Z   0.0   0.0   0:00.00 watch
  146 monica    20   0   10656    688    656 T   0.0   0.0   0:00.00 sh
  147 monica    20   0       0      0      0 Z   0.0   0.0   0:00.04 top
  148 monica    20   0   15480    964    920 T   0.0   0.0   0:00.01 timeout
  149 monica    20   0   16780   2580   2264 T   0.0   0.0   0:00.04 watch
  177 monica    20   0   16780    180     96 T   0.0   0.0   0:00.00 watch
  178 monica    20   0   10656    688    656 T   0.0   0.0   0:00.00 sh
  179 monica    20   0       0      0      0 Z   0.0   0.0   0:00.01 top
  180 monica    20   0   15480    964    920 T   0.0   0.0   0:00.00 timeout
  181 monica    20   0   16524   2340   1972 T   0.0   0.0   0:00.07 watch
  206 monica    20   0       0      0      0 Z   0.0   0.0   0:00.00 watch
  207 monica    20   0   10656    688    656 T   0.0   0.0   0:00.01 sh
  208 monica    20   0       0      0      0 Z   0.0   0.0   0:00.01 top
  209 monica    20   0   18920   2212   1536 R   0.0   0.0   0:00.60 top
```

Figure 16: output of “top -d 1 -n 180” command

- Ps command don’t have any flag to get the result for 3 minutes. From the man ps command we can observe that ps command doesn’t have any flags.



```
PS(1)                                   User Commands                                   PS(1)

NAME
       ps - report a snapshot of the current processes.

SYNOPSIS
       ps [options]

DESCRIPTION
       ps displays information about a selection of the active processes.  If you want a repetitive update of the selection and the displayed information, use top(1) instead.

       This version of ps accepts several kinds of options:

       1   UNIX options, which may be grouped and must be preceded by a dash.
       2   BSD options, which may be grouped and must not be used with a dash.
       3   GNU long options, which are preceded by two dashes.

       Options of different types may be freely mixed, but conflicts can appear.  There are some synonymous options, which are functionally identical, due to the many standards and ps implementations that
       this ps is compatible with.

       Note that "ps -aux" is distinct from "ps aux".  The POSIX and UNIX standards require that "ps -aux" print all processes owned by a user named "x", as well as printing all processes that would be
       selected by the -a option.  If the user named "x" does not exist, this ps may interpret the command as "ps aux" instead and print a warning.  This behavior is intended to aid in transitioning old
       scripts and habits.  It is fragile, subject to change, and thus should not be relied upon.

       By default, ps selects all processes with the same effective user ID (euid=EUID) as the current user and associated with the same terminal as the invoker.  It displays the process ID (pid=PID), the
       terminal associated with the process (tname=TTY), the cumulated CPU time in [DD-]hh:mm:ss format (time=TIME), and the executable name (ucmd=CMD).  Output is unsorted by default.

       The use of BSD-style options will add process state (stat=STAT) to the default display and show the command args (args=COMMAND) instead of the executable name.  You can override this with the
       PS_FORMAT environment variable.  The use of BSD-style options will also change the process selection to include processes on other terminals (TTYs) that are owned by you; alternately, this may be
       described as setting the selection to be the set of all processes filtered to exclude processes owned by other users or not on a terminal.  These effects are not considered when options are
       described as being "identical" below, so -M will be considered identical to Z and so on.

       Except as described below, process selection options are additive.  The default selection is discarded, and then the selected processes are added to the set of processes to be displayed.  A process
       will thus be shown if it meets any of the given selection criteria.
```

Figure 17: output of “man ps” command

6) saving the sample data of a file and loading it again.

- **top -d 2 > top.txt:** In this command the output of the “top -d 2” command is stored to top .txt. The “>” symbols help in storing the data of the particular command.
- **Cat top.txt**: The cat command is used to display the data in the “top.txt” file.

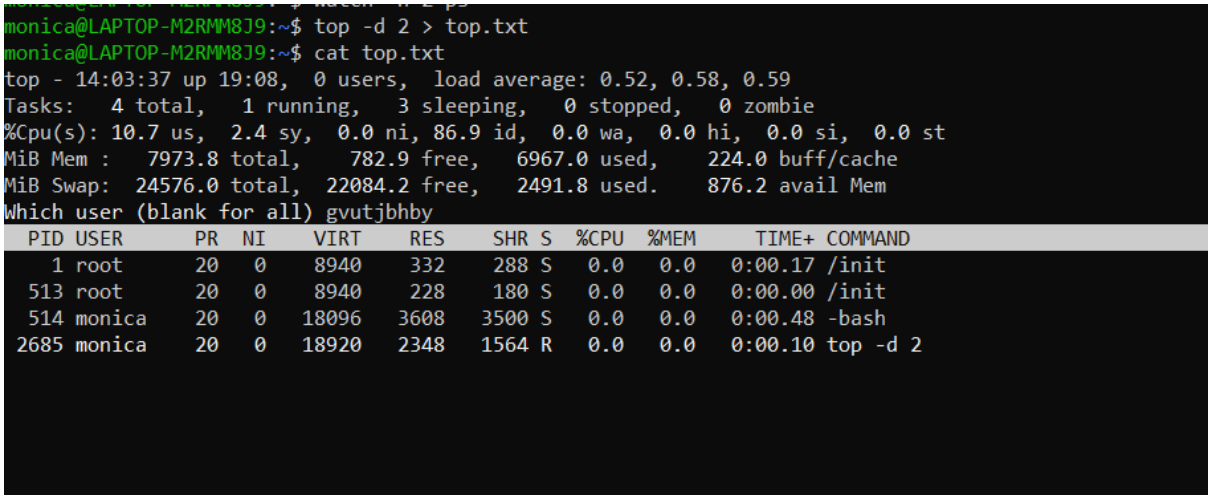


Figure 18: output of “cat top.txt” command

- **Sar 2 > sar.txt:** In this command the output of the “sar 2” command is stored in “sar.txt”.
- **Cat sar.txt:** cat command is used to get the information of “sar.txt”.

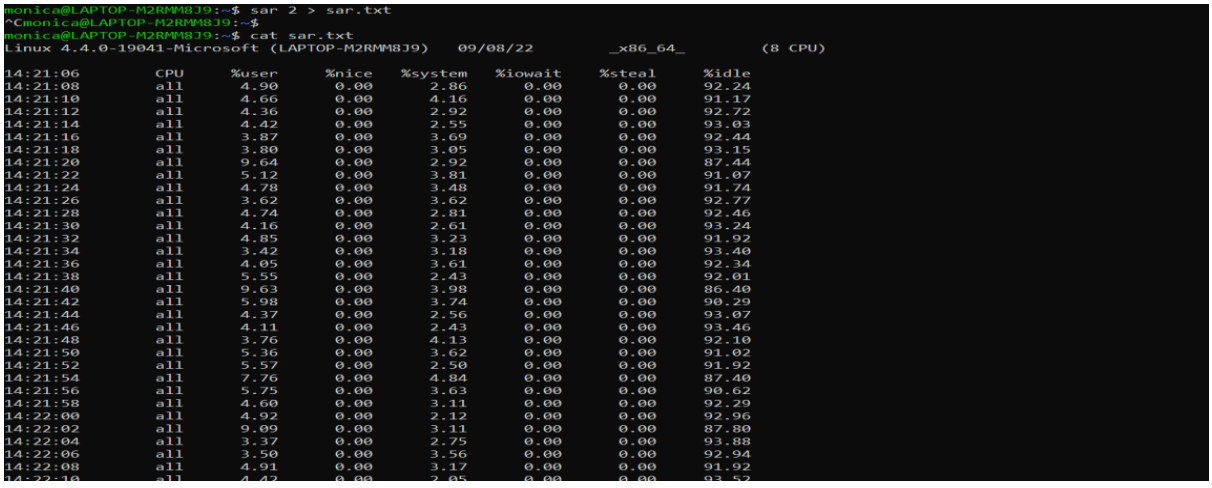


Figure 19: output of “cat sar.txt” command

- **Ps > ps.txt:** The output of the “ps” command is stored in “ps.txt”
- **Cat ps .txt:** Cat command is used to get the information stored in the file “ps.txt”.

```
monica@LAPTOP-M2RMM8J9:~$ ps >ps.txt
monica@LAPTOP-M2RMM8J9:~$ cat ps.txt
  PID TTY          TIME CMD
   12 tty1     00:00:00 bash
  113 tty1     00:00:00 vi
  134 tty1     00:00:00 sar
  135 tty1     00:00:00 sadc
  137 tty1     00:00:00 top
  138 tty1     00:00:00 watch
  145 tty1     00:00:00 watch <defunct>
  146 tty1     00:00:00 sh
  147 tty1     00:00:00 top <defunct>
  148 tty1     00:00:00 timeout
  149 tty1     00:00:00 watch
  177 tty1     00:00:00 watch
  178 tty1     00:00:00 sh
  179 tty1     00:00:00 top <defunct>
  180 tty1     00:00:00 timeout
  181 tty1     00:00:00 watch
  206 tty1     00:00:00 watch <defunct>
  207 tty1     00:00:00 sh
  208 tty1     00:00:00 top <defunct>
  210 tty1     00:00:00 ps
monica@LAPTOP-M2RMM8J9:~$
```

Figure 20: output of “cat ps.txt” command.

7)Simple statistics for a specific processor

- **Sar -p (CPUId-n):** This command is used to get the information(statistics) of the specific processes. In this task **“sar -p 0”** and **“sar -p 1”.**

```
monica@LAPTOP-M2RMM8J9:~$ sar -p 0
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/18/22        _x86_64_        (8 CPU)

12:58:37        CPU     %user     %nice   %system   %iowait    %steal     %idle
12:58:37        all      1.23      0.00      2.14      0.00      0.00     96.63
monica@LAPTOP-M2RMM8J9:~$ sar -p 1
Linux 4.4.0-19041-Microsoft (LAPTOP-M2RMM8J9)   09/18/22        _x86_64_        (8 CPU)

12:58:43        CPU     %user     %nice   %system   %iowait    %steal     %idle
12:58:44        all      1.12      0.00      0.75      0.00      0.00     98.13
12:58:45        all      1.62      0.00      5.98      0.00      0.00     92.40
12:58:46        all      3.13      0.00      3.13      0.00      0.00     93.74
12:58:47        all      1.01      0.00      0.38      0.00      0.00     98.61
12:58:48        all      0.38      0.00      0.63      0.00      0.00     99.00
12:58:49        all      0.50      0.00      1.50      0.00      0.00     98.00
12:58:50        all      1.61      0.00      2.72      0.00      0.00     95.67
12:58:51        all      0.00      0.00      0.63      0.00      0.00     99.37
12:58:52        all      0.38      0.00      0.00      0.00      0.00     99.62
^Z
[1]+  Stopped                 sar -p 1
```

Figure 21: output of “sar -p 0” and “sar -p 1” command.

8)sample statistics for a process/thread or group of process/threads.

- **top -H |grep sar -u 2|ps:** In this command pipe is used so that the output of **"sar -u 2"** is given as the input to **"ps"** command.

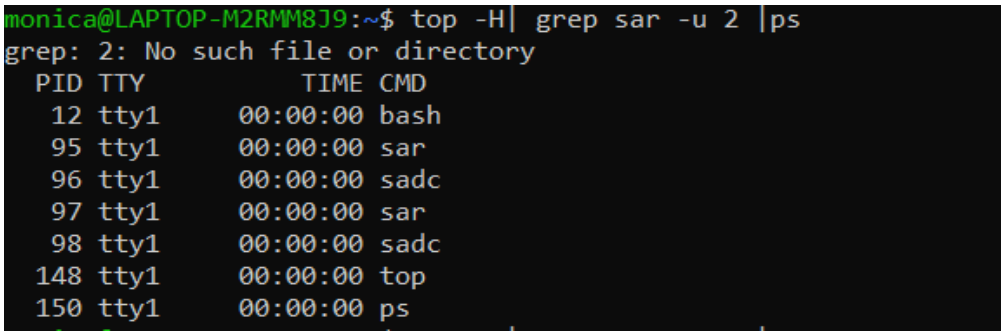


Figure 22: output of "top -H|grep sar -u 2|ps" command.

- **top -H|grep sar -u 2|sar -r 2:** In this command pipe is used to give the output of **"sar -u 2"** as input of **"sar -r 2"** command.

Figure 23: output of "top -H| grep sar -u 2|sar -r 2" command.

Task -2(b) Performance monitor of C++ programs:

1. Execute and monitor the applications with test_rounds set to 32, present your result.

To achieve the aim of the task three C++ programming files are given they are log_analyzer, matrix_multiplication and file_reader. To execute and monitor the C++ programs 2 tools are used those are "valgrind" and "kcachegrind".

Log_analyzer:

In the file log_analyzer three different variations are given they are "analyze_log", "analyze_charbuf", "analyze_pm_hash". For all these variations Valgrind and kcachegrind is executed to monitor the results.

- **Analyze_log**: To achieve the task, we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

  Step1: g++ -std=c++17 -g analyze_log.cpp -o analyze_log.

  Step-2: valgrind –tool=callgrind ./ analyze_log 32.
  Step-3: Kcachegrind callgrind.out.2292

```
dv1567@dv1567-VirtualBox:~/Downloads/code/log_analyzer$ valgrind --tool=callgrind ./analyze_log
32
==2292== Callgrind, a call-graph generating cache profiler
==2292== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2292== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2292== Command: ./analyze_log 32
==2292==
==2292== For interactive control, run 'callgrind_control -h'.
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
```

Figure 24: valgrind output of analyze_log

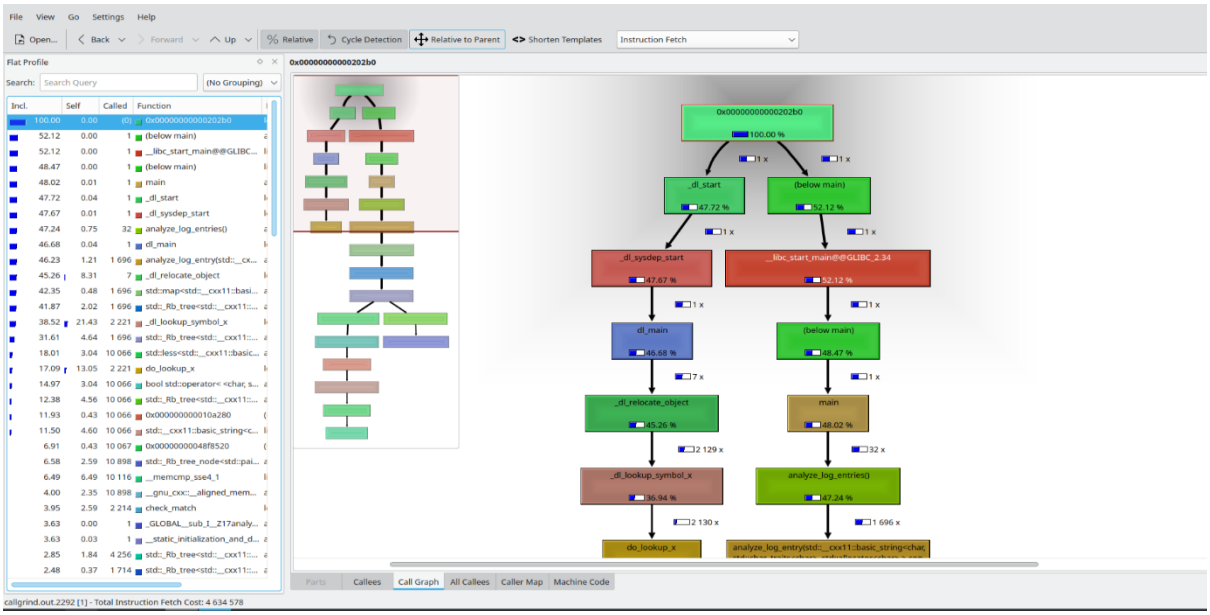


Figure 25: Kcachegrind output of analyze_log

- **Analyze_charbuf:** To achieve the task , we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

Step1: g++ -std=c++17 -g analyze_charbuf.cpp -o analyze_charbuf.
Step-2: valgrind –tool=callgrind ./ analyze_charbuf 32.
Step-3: Kcachegrind callgrind.out.2304

```
dv1567@dv1567-VirtualBox:~/Downloads/code/log_analyzer$ valgrind --tool=callgrind ./analyze_char
buf 32
==2304== Callgrind, a call-graph generating cache profiler
==2304== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2304== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2304== Command: ./analyze_charbuf 32
==2304==
==2304== For interactive control, run 'callgrind_control -h'.
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
351
```

Figure 26: valgrind output of analyze_chabuf.

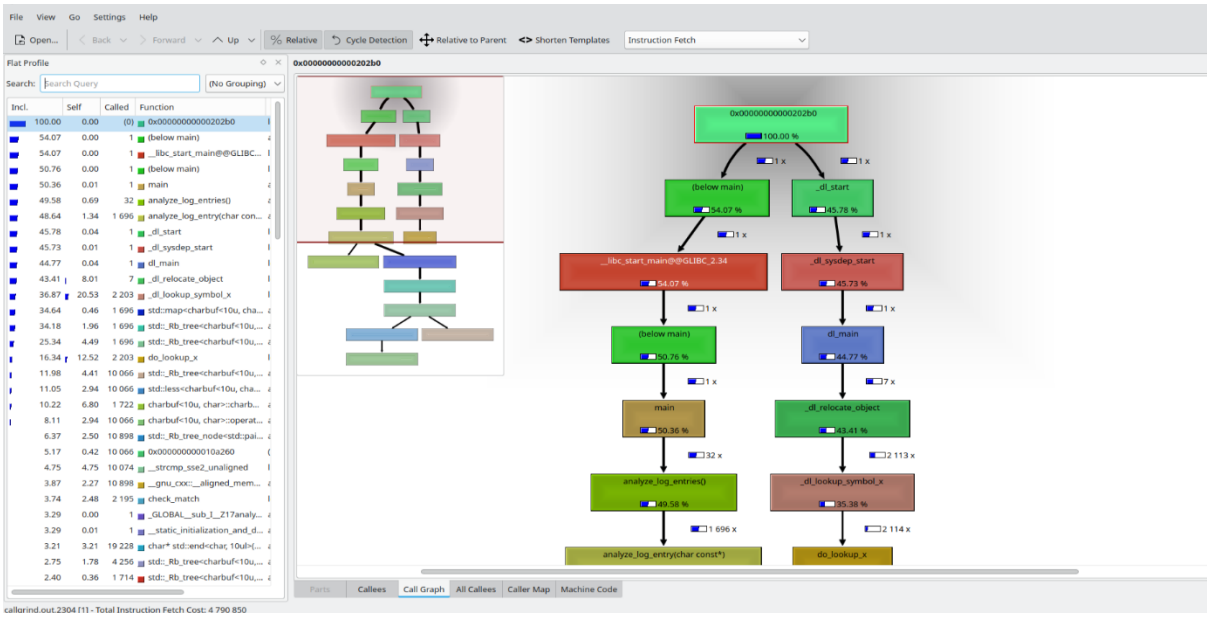


Figure 27: Kcachegrind of analyze_charbuf.

- **Analyze_pm_hash:** To achieve the task , we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

  Step1: g++ -std=c++17 -g analyze_pm_hash.cpp -o analyze_pm_hash.
  Step-2: valgrind –tool=callgrind ./ analyze_log 32.
  Step-3: Kcachegrind callgrind.out.2316

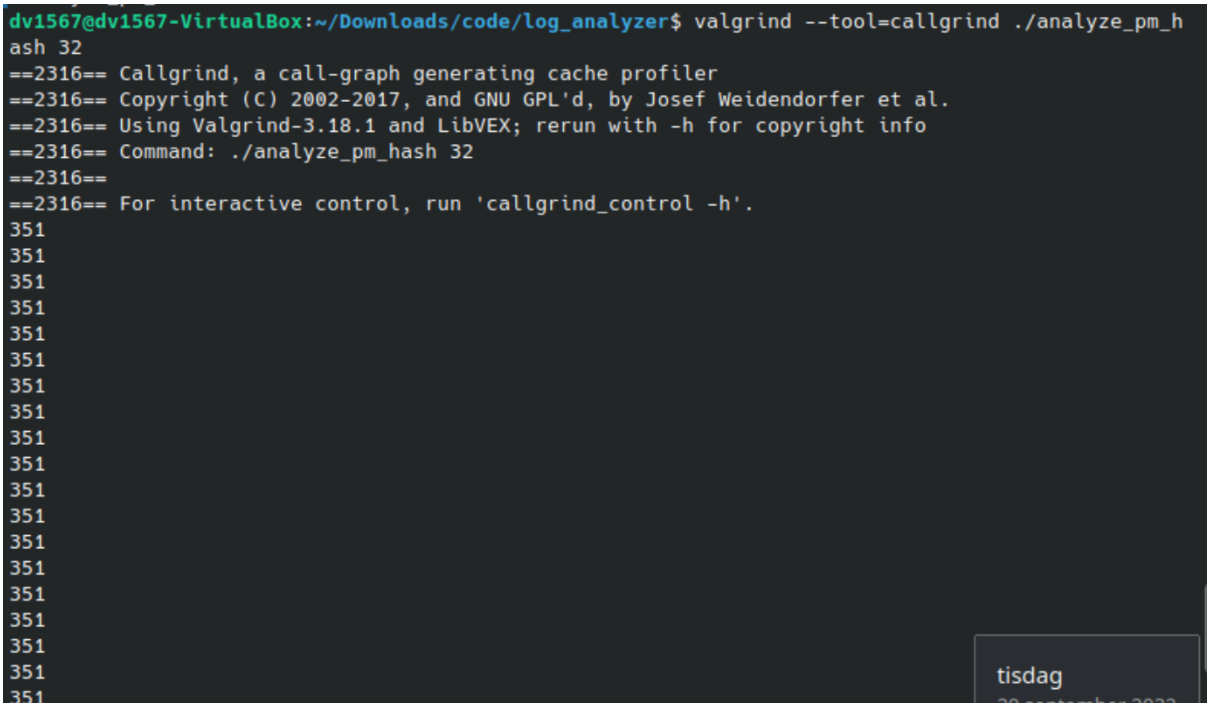


Figure 28: valgrind output of analyze_pm_hash.

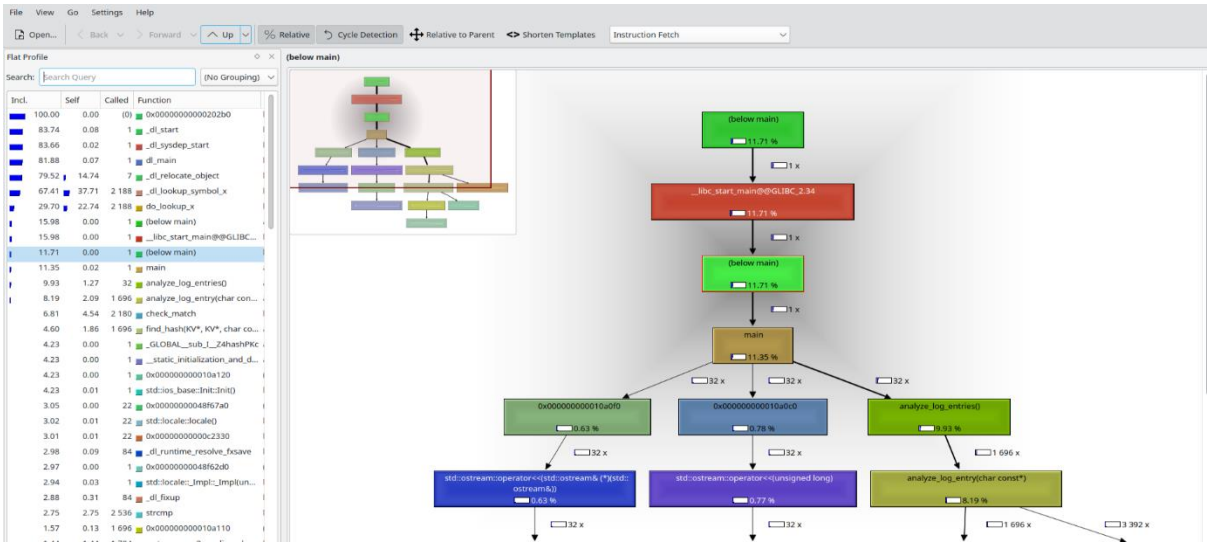


Figure 29: kcachegrind output of analyze_pm_hash.

- **File_reader**: In the file log_analyzer three different variations are given they are file_reader, file_reader_read, file_reader_parsimonious, file_reader_sgetn. For all these variations Valgrind and kcachegrind is runned to execute and monitor the results.

  File_reader: To achieve the task, we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an "callgrind.out.pid" file is generated, then by using this output the kcachegrind output is generated.
  Step1: g++ -std=c++17 -g file_reader.cpp -o file_reader.
  Step-2: valgrind –tool=callgrind ./ file_reader 32.
  Step-3: Kcachegrind callgrind.out.2034

```
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$ valgrind --tool=callgrind ./file_reader 3
2
==2034== Callgrind, a call-graph generating cache profiler
==2034== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2034== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2034== Command: ./file_reader 32
==2034==
==2034== For interactive control, run 'callgrind_control -h'.
==2034== brk segment overflow in thread #1: can't grow to 0x4870000
==2034== (see section Limitations in user manual)
==2034== NOTE: further instances of this message will not be shown
==2034==
==2034== Events    : Ir
==2034== Collected : 138482077
==2034==
==2034== I   refs:      138,482,077
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$
```

Figure 30: valgrind output of file_reader.

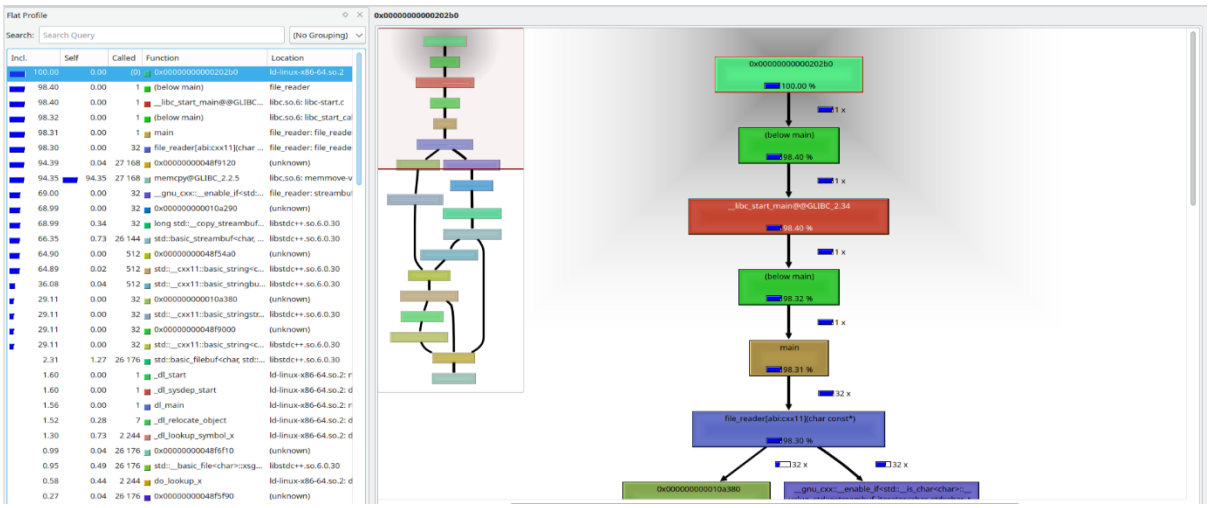


Figure 31: kcachegrind of file_reader

- **File_reader_read:** To achieve the task , we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

  Step1: g++ -std=c++17 -g file_reader_read.cpp -o file_reader_read.

  Step-2: valgrind –tool=callgrind ./ file_reader_read 32.

  Step-3: Kcachegrind callgrind.out.2146

```
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$ valgrind --tool=callgrind ./file_reader_r
ead 32
==2146== Callgrind, a call-graph generating cache profiler
==2146== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2146== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2146== Command: ./file_reader_read 32
==2146==
==2146== For interactive control, run 'callgrind_control -h'.
==2146==
==2146== Events    : Ir
==2146== Collected : 3131684
==2146==
==2146== I   refs:      3,131,684
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$
```

Battery at 76%
2:32 remaining

Figure 32: valgrind output of file_reader_read

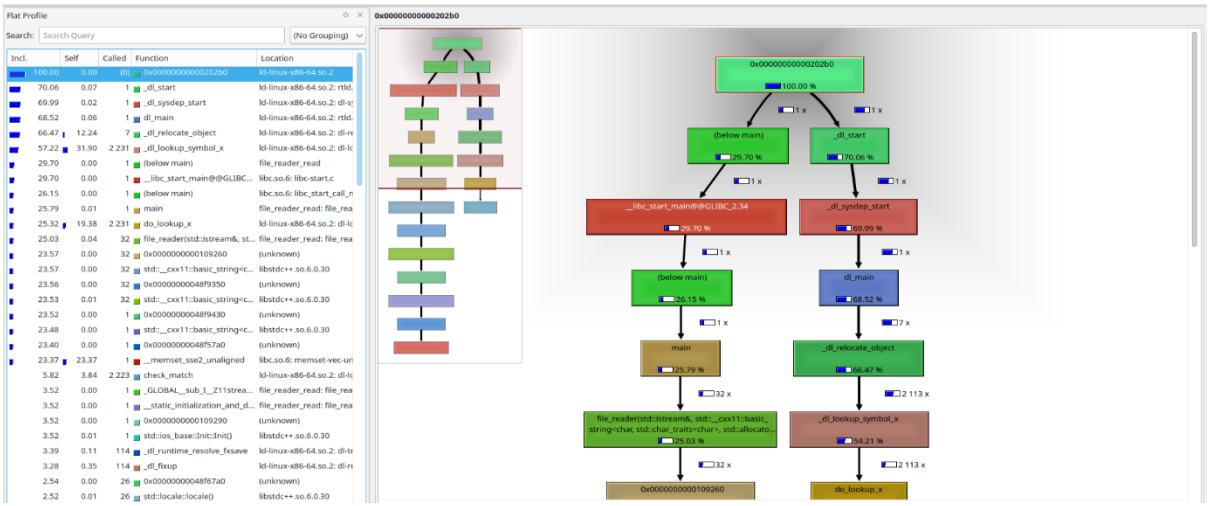


Figure 33: Kcachegrind of file_reader_read

- **File_reader_parsimonious:** To achieve the task, we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

  Step1: g++ -std=c++17 -g file_reader_parsimonious.cpp -o file_reader_parsimonious.

  Step-2: valgrind –tool=callgrind./ file_reader_parsimonious 32.

  Step-3: Kcachegrind callgrind.out.2037

```
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$ valgrind --tool=callgrind ./file_reader_p
arsimonious 32
==2037== Callgrind, a call-graph generating cache profiler
==2037== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2037== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2037== Command: ./file_reader_parsimonious 32
==2037==
==2037== For interactive control, run 'callgrind_control -h'.
==2037==
==2037== Events    : Ir
==2037== Collected : 2949751561
==2037==
==2037== I   refs:      2,949,751,561
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$
```

Figure 34: valgrind of file_reader_parsimonious.

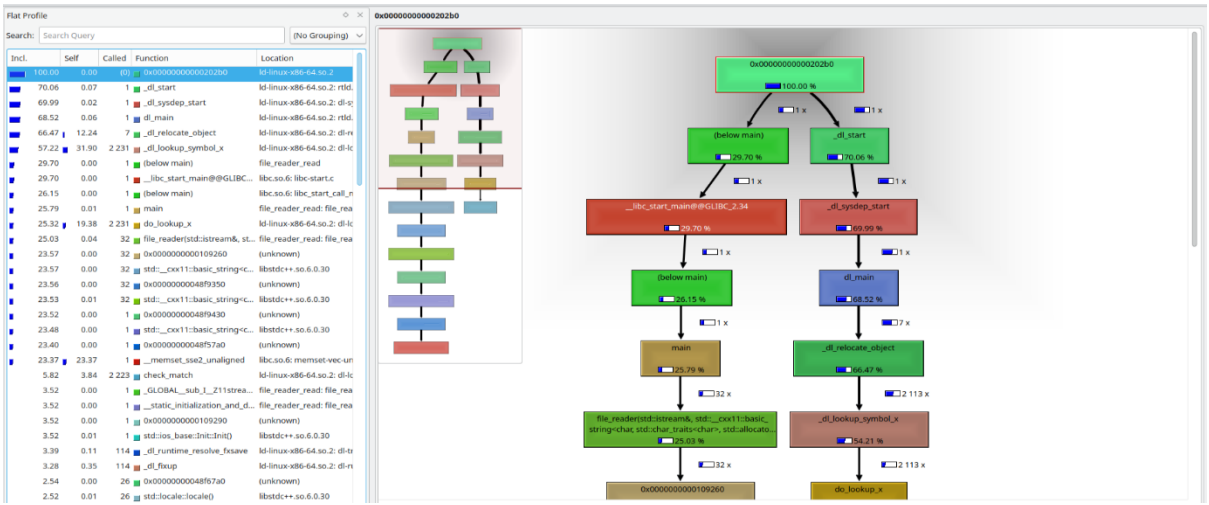


Figure 35: kcachegrind of file_reader_parsimonious

- **File_reader_sgetn:** To achieve the task, we need to traverse to the path where the code recites, then following need to be used to execute. After step-2 an "callgrind.out.pid" file is generated, then by using this output the kcachegrind output is generated.

  Step1: g++ -std=c++17 -g file_reader_sgetn.cpp -o file_reader_sgetn.
  Step-2: valgrind –tool=callgrind./ file_reader_sgetn 32.
  Step-3: Kcachegrind callgrind.out.2131

```
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$ valgrind --tool=callgrind ./file_reader_s
getn 32
==2131== Callgrind, a call-graph generating cache profiler
==2131== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2131== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2131== Command: ./file_reader_sgetn 32
==2131==
==2131== For interactive control, run 'callgrind_control -h'.
==2131==
==2131== Events    : Ir
==2131== Collected : 3130438
==2131==
==2131== I   refs:      3,130,438
dv1567@dv1567-VirtualBox:~/Downloads/code/file_reader$
```

Updates
Updates available

Figure 36: valgrind of file_reader_sgetn

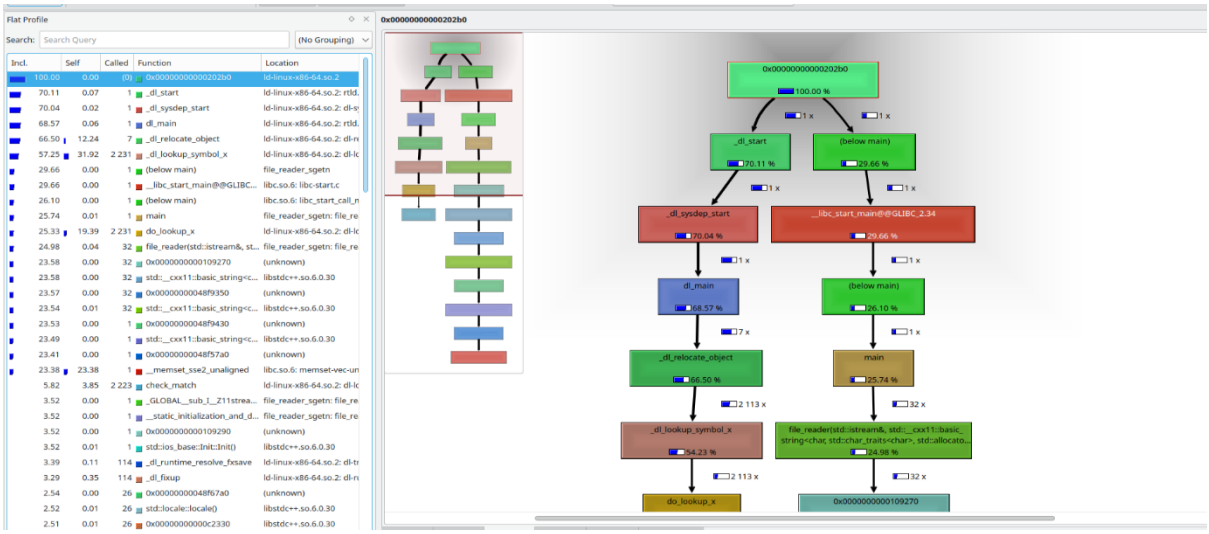


Figure 37: Kcachegrind of file_reader_sgetn

- **Matrix_multiplication**: In the file log_analyzer three different variations are given they are "naïve", "naïve_parallel". For all these variations Valgrind and kcachegrind are executed to monitor the results.

  Naive: To achieve the task, we need to traverse to the path where the code recites then make should be given so that the executable commands are, then following steps need to be done to execute. In the step-2 along with test_rounds we need to give dimensions, after step-2 an "callgrind.out.pid" file is generated, then by using this output the kcachegrind output is generated.

  Naïve with 64 as dimension.

  Step1: g++ -std=c++17 -g -c DNAIVE matrix.cpp -o matrix_naive.o
  Step-2: valgrind –tool=callgrind./ naive 64 32.
  Step-3: Kcachegrind callgrind.out.1779

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive 64 32
==1779== Callgrind, a call-graph generating cache profiler
==1779== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==1779== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1779== Command: ./naive 64 32
==1779==
==1779== For interactive control, run 'callgrind_control -h'.
==1779==
==1779== Events    : Ir
==1779== Collected : 768438237
==1779==
==1779== I   refs:      768,438,237
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 38: valgrind output of naïve 64 32.

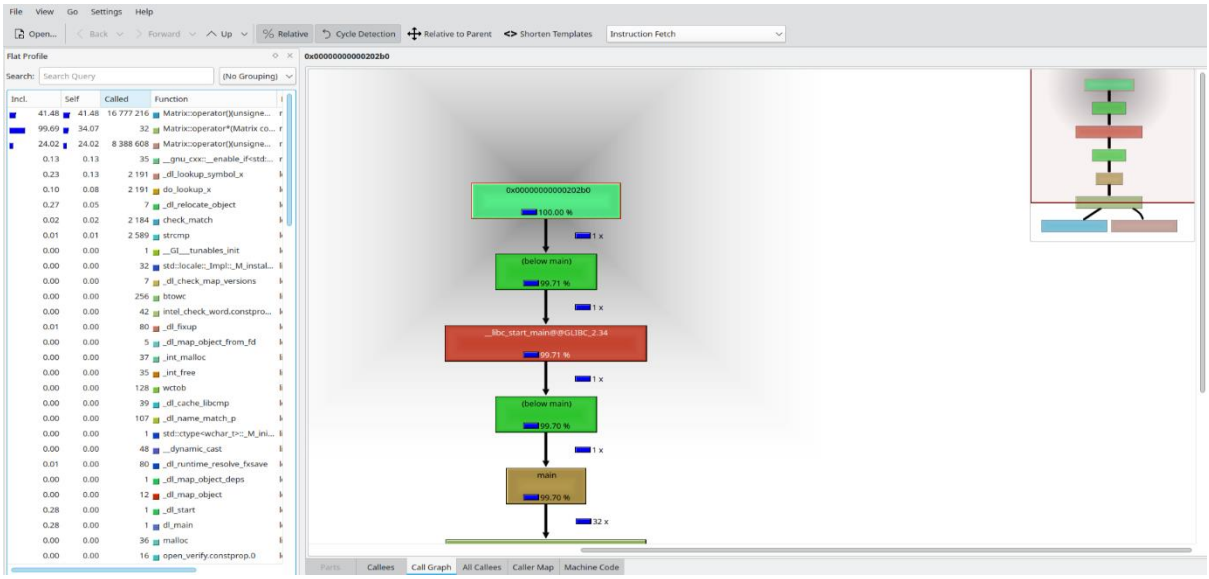


Figure 39: kcachegrind of naïve 64 32

Naïve with 128 as dimension.

Step1: g++ -std=c++17 -g -c DNAIVE matrix.cpp -o matrix_naive.o

Step-2: valgrind –tool=callgrind./ naive 128 32

Step-3: Kcachegrind callgrind.out.1815

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive 128  32
==1815== Callgrind, a call-graph generating cache profiler
==1815== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==1815== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1815== Command: ./naive 128 32
==1815==
==1815== For interactive control, run 'callgrind_control -h'.
==1815==
==1815== Events    : Ir
==1815== Collected : 6120130617
==1815==
==1815== I   refs:      6,120,130,617
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 40: valgrind output of naïve 128 32

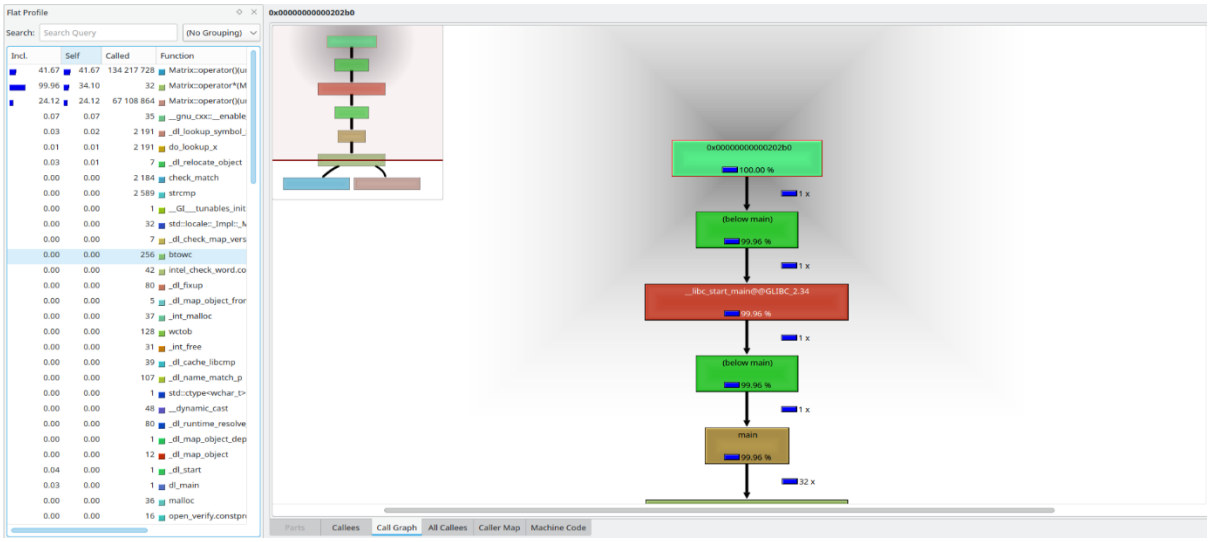


Figure 41: Kcachegrind output of naïve 128 32

Naïve with 256 as dimension.

Step1: g++ -std=c++17 -g -c DNAIVE matrix.cpp -o matrix_naive.o

Step-2: valgrind –tool=callgrind./ naive 64 32.

Step-3: Kcachegrind callgrind.out.1767.

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive 256 32
==1767== Callgrind, a call-graph generating cache profiler
==1767== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==1767== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1767== Command: ./naive 256 32
==1767==
==1767== For interactive control, run 'callgrind_control -h'.
==1767==
==1767== Events    : Ir
==1767== Collected : 48901020016
==1767==
==1767== I   refs:      48,901,020,016
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 42: valgrind output of naïve 256 32

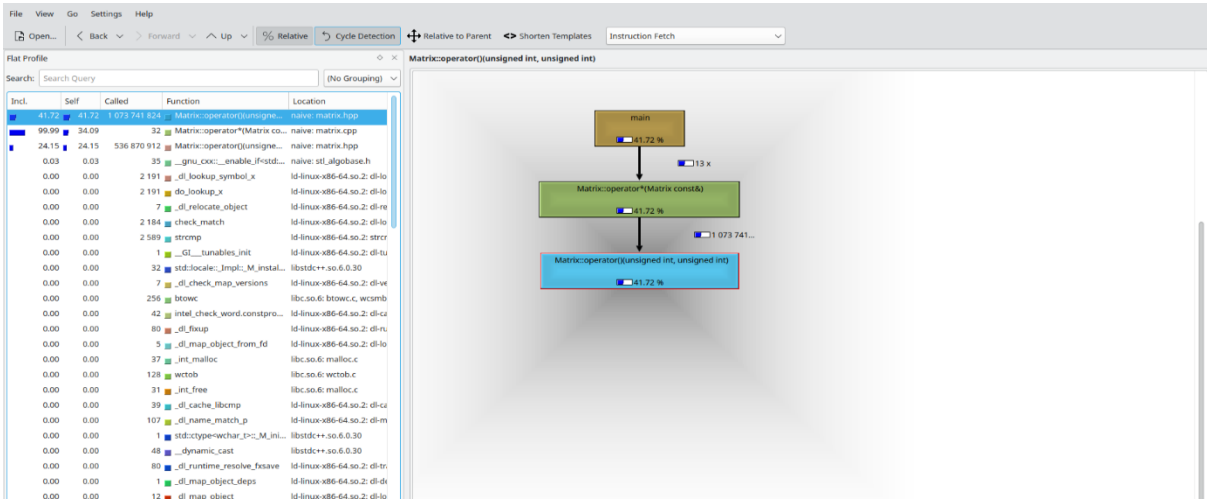


Figure 43: Kcachegrind output of naïve 256 32

Naïve with 512 as dimension.

Step1: g++ -std=c++17 -g -c DNAIVE matrix.cpp -o matrix_naive.o

Step-2: valgrind –tool=callgrind./ naive 64 32.

Step-3: Kcachegrind callgrind.out. 12203

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive 512 32
==12203== Callgrind, a call-graph generating cache profiler
==12203== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==12203== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==12203== Command: ./naive 512 32
==12203==
==12203== For interactive control, run 'callgrind_control -h'.
==12203==
==12203== Events    : Ir
==12203== Collected : 391017855344
==12203==
==12203== I   refs:      391,017,855,344
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 44: valgrind output of naïve 512 32

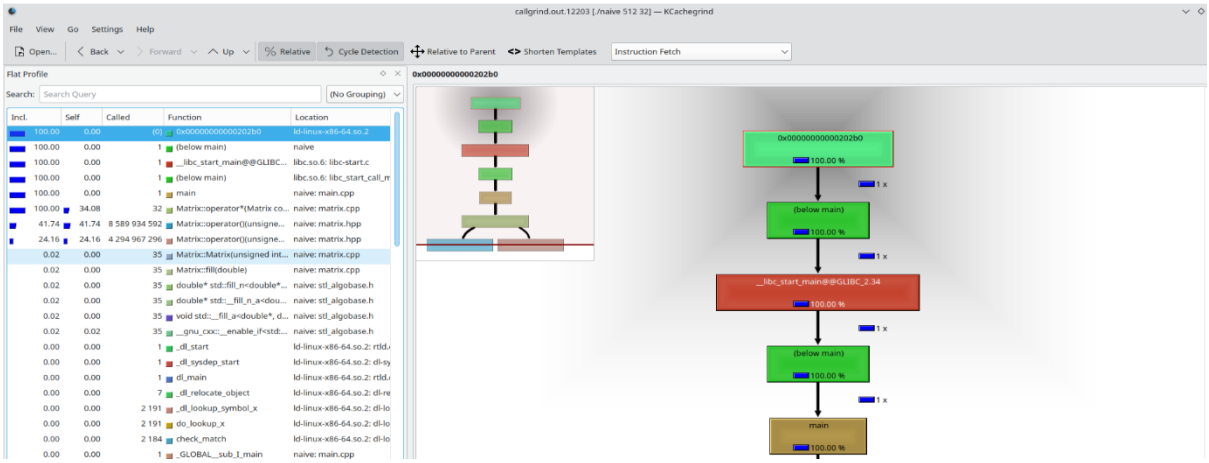


Figure 45: kcachegrind output of naïve 512 32.

Naïve with 1024 as dimension.

Step1: g++ -std=c++17 -g -c DNAIVE matrix.cpp -o matrix_naive.o

Step-2: valgrind –tool=callgrind./ naive 64 32.

Step-3: Kcachegrind callgrind.out.

- **Strassen_parallel:** To achieve the task, we need to traverse to the path where the code recites then make should be given so that the executable commands are, then following steps need to be done to execute. In the step-2 along with test_rounds we need to give dimensions, after step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

  Strassen_parallel with 64 as dimension.

  Step1: g++ -std=c++17 -g -c DSTRASSEN_PARALLEL matrix.cpp -o matrix_strassen_parallel. o

  Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32.

  Step-3: Kcachegrind callgrind.out.14201

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./strassen_parallel 64 32
==14201== Callgrind, a call-graph generating cache profiler
==14201== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==14201== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14201== Command: ./strassen_parallel 64 32
==14201==
==14201== For interactive control, run 'callgrind_control -h'.
==14201==
==14201== Events    : Ir
==14201== Collected : 725330770
==14201==
==14201== I   refs:      725,330,770
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 46: Valgrind output of Strassen_parallel 64 32

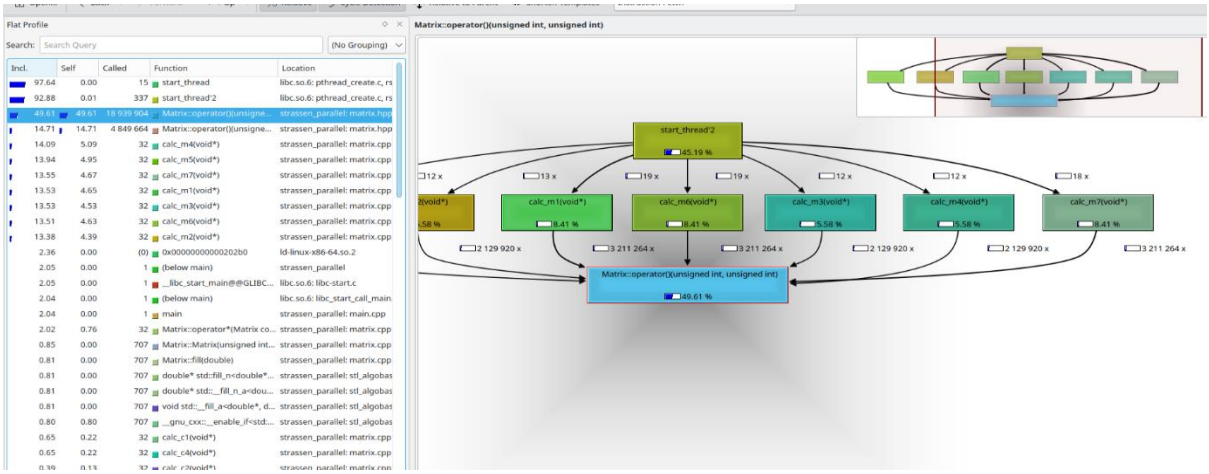


Figure 47: Kcachegrind output of Strassen_parallel 64 32.

Strassen_parallel with 128 as dimension.

Step1: : g++ -std=c++17 -g -c DSTRASSEN_PARALLEL matrix.cpp -o matrix_strassen_parallel. o

Step-2: valgrind –tool=callgrind./ naïve_parallel 128 32.

Step-3: Kcachegrind callgrind.out.14576

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./strassen_parallel 128 32
==14576== Callgrind, a call-graph generating cache profiler
==14576== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==14576== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14576== Command: ./strassen_parallel 128 32
==14576==
==14576== For interactive control, run 'callgrind_control -h'.
==14576==
==14576== Events    : Ir
==14576== Collected : 7820590138
==14576==
==14576== I   refs:      7,820,590,138
```

Figure 48: Valgrind output of Strassen_parallel 128 32

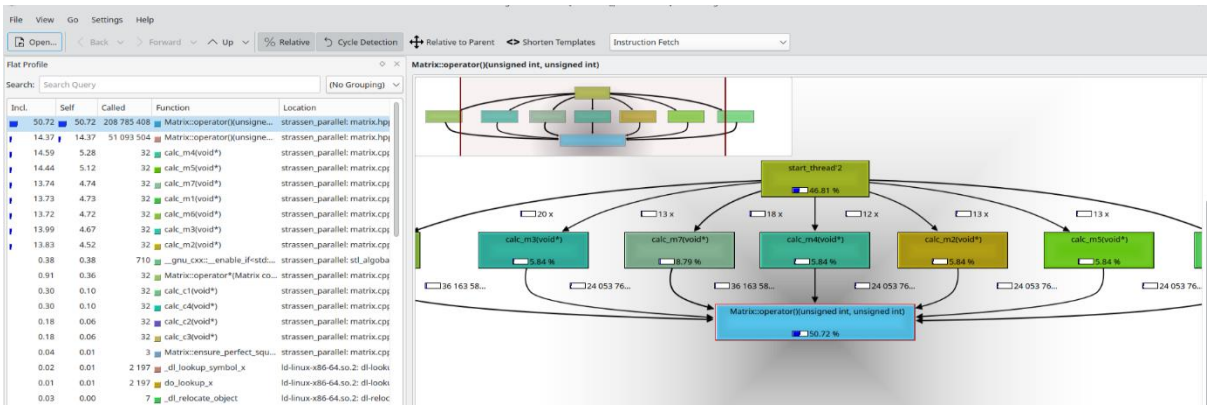


Figure 49: Kcachegrind output of Strassen_parallel 128 32

Strassen_parallel with 256 as dimension.

Step1: : g++ -std=c++17 -g -c DSTRASSEN_PARALLEL matrix.cpp -o matrix_strassen_parallel. o

Step-2: valgrind –tool=callgrind./ naïve_parallel 256 32.

Step-3: Kcachegrind callgrind.out.1741

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./strassen_parallel 256 32
==1741== Callgrind, a call-graph generating cache profiler
==1741== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==1741== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1741== Command: ./strassen_parallel 256 32
==1741==
==1741== For interactive control, run 'callgrind_control -h'.
==1741==
==1741== Events    : Ir
==1741== Collected : 43106942411
==1741==
==1741== I   refs:      43,106,942,411
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 50: valgrind of Strassen_parallel 256 32

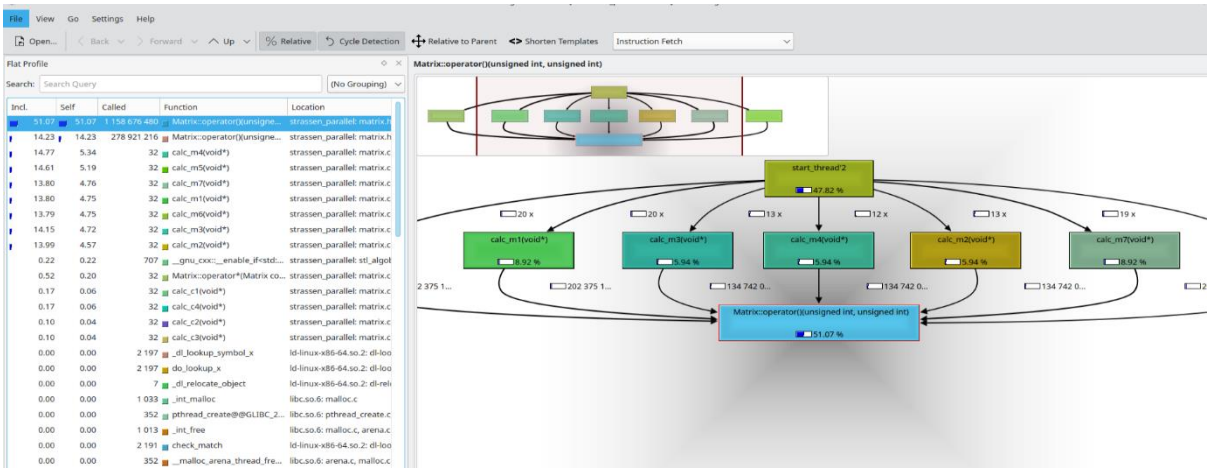


Figure 51: Kcachegrind of Strassen_parallel 256 32

Strassen_parallel with 512 as dimension.

Step1: : g++ -std=c++17 -g -c DSTRASSEN_PARALLEL matrix.cpp -o matrix_strassen_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 512 32.

Step-3: Kcachegrind callgrind.out.1779

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./strassen_parallel 512 32
==2122== Callgrind, a call-graph generating cache profiler
==2122== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2122== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2122== Command: ./strassen_parallel 512 32
==2122==
==2122== For interactive control, run 'callgrind_control -h'.
==2122== brk segment overflow in thread #1: can't grow to 0x4869000
==2122== (see section Limitations in user manual)
==2122== NOTE: further instances of this message will not be shown
==2122==
==2122== Events    : Ir
==2122== Collected : 373595138463
==2122==
==2122== I   refs:      373,595,138,463
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 52: valgrind of Strassen_parallel 512 32

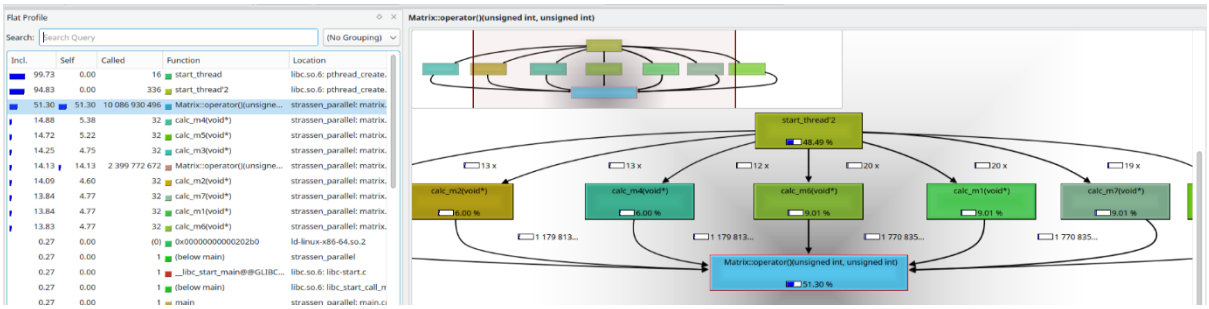


Figure 53: kcachegrind of Strassen_parallel 512 32.

Strassen_parallel with 1024 as dimension.

Step1: g++ -std=c++17 -g -c DSTRASSEN_PARALLEL matrix.cpp -o matrix_strassen_parallel. o

Step-2: valgrind –tool=callgrind. / naïve_parallel 1024 4

Step-3: Kcachegrind callgrind.out.2733

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./strassen_parallel 1024 4
==2733== Callgrind, a call-graph generating cache profiler
==2733== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2733== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2733== Command: ./strassen_parallel 1024 4
==2733==
==2733== For interactive control, run 'callgrind_control -h'.
==2733== brk segment overflow in thread #1: can't grow to 0x486f000
==2733== (see section Limitations in user manual)
==2733== NOTE: further instances of this message will not be shown
==2733==
==2733== Events    : Ir
==2733== Collected : 338690842500
==2733==
==2733== I   refs:      338,690,842,500
```

Figure 54: valgrind output of Strassen_parallel 1024 4

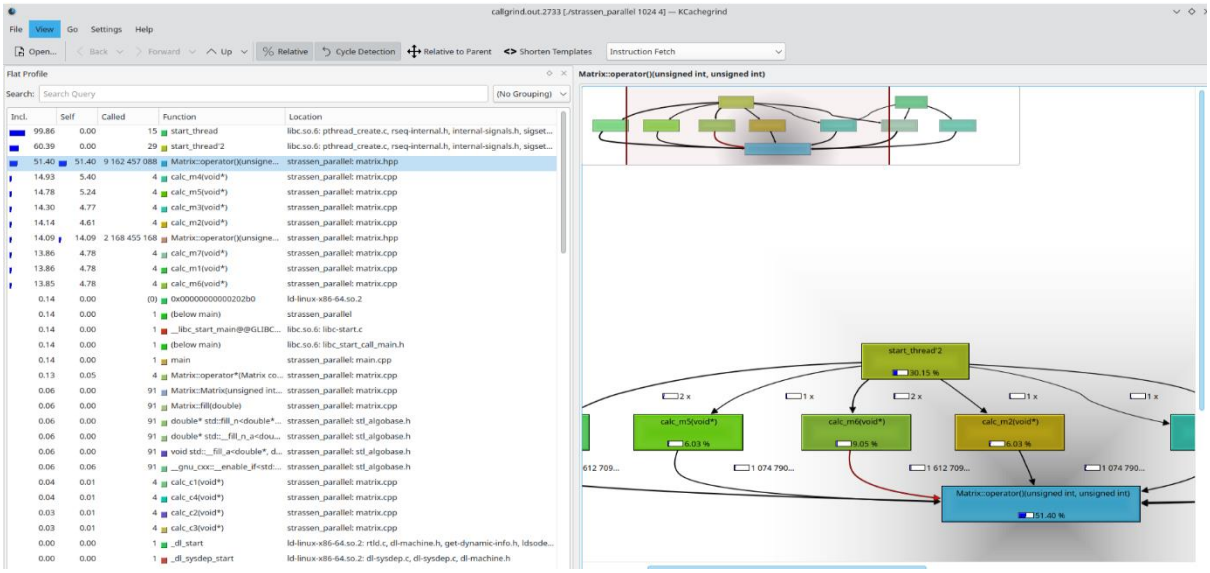


Figure 55: kcachegrind output of Strassen_parallel 1024 4

Naïve_Parallel: To achieve the task, we need to traverse to the path where the code recites then make should be given so that the executable commands are, then following steps need to be done to execute. In the step-2 along with test_rounds and no.of threads we need to give dimensions, after step-2 an “callgrind.out.pid” file is generated, then by using this output the kcachegrind output is generated.

Naïve_parallel with 64 as dimension 1 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32 1.

Step-3: Kcachegrind callgrind.out.2953

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 64 32 1
==2953== Callgrind, a call-graph generating cache profiler
==2953== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2953== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2953== Command: ./naive_parallel 64 32 1
==2953==
==2953== For interactive control, run 'callgrind_control -h'.
==2953==
==2953== Events    : Ir
==2953== Collected : 776359096
==2953==
==2953== I   refs:      776,359,096
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 56: valgrind output of naïve_parallel 64 32 1

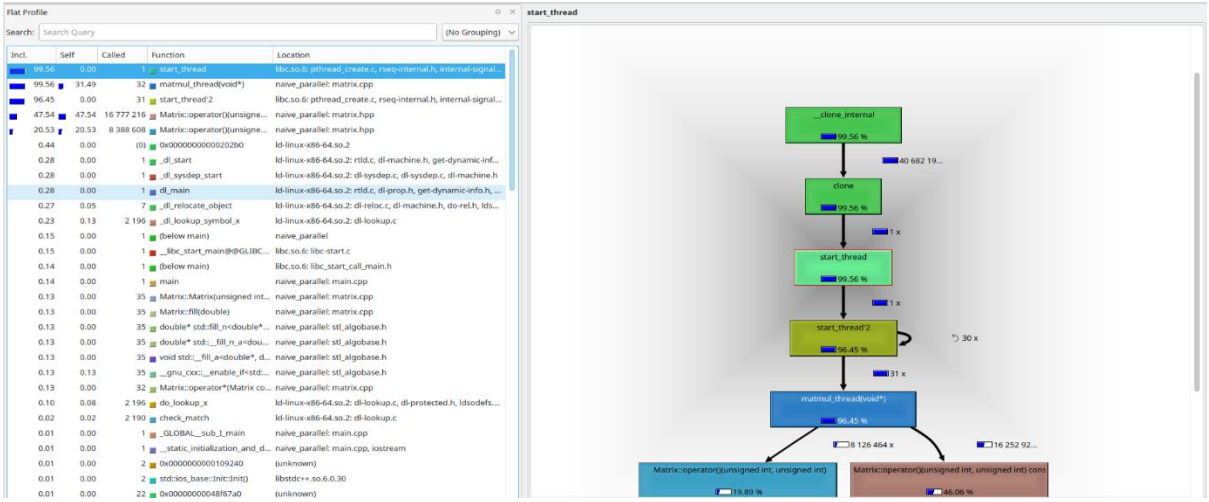


Figure 57: kcachegrind output of naïve_parallel 64 32 1

Naïve_parallel with 64 as dimension 2 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32 2.

Step-3: Kcachegrind callgrind.out.3010

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 64 32 2
==3010== Callgrind, a call-graph generating cache profiler
==3010== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3010== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3010== Command: ./naive_parallel 64 32 2
==3010==
==3010== For interactive control, run 'callgrind_control -h'.
==3010==
==3010== Events    : Ir
==3010== Collected : 776411780
==3010==
==3010== I   refs:      776,411,780
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 58: valgrind output of naïve_parallel 64 32 2

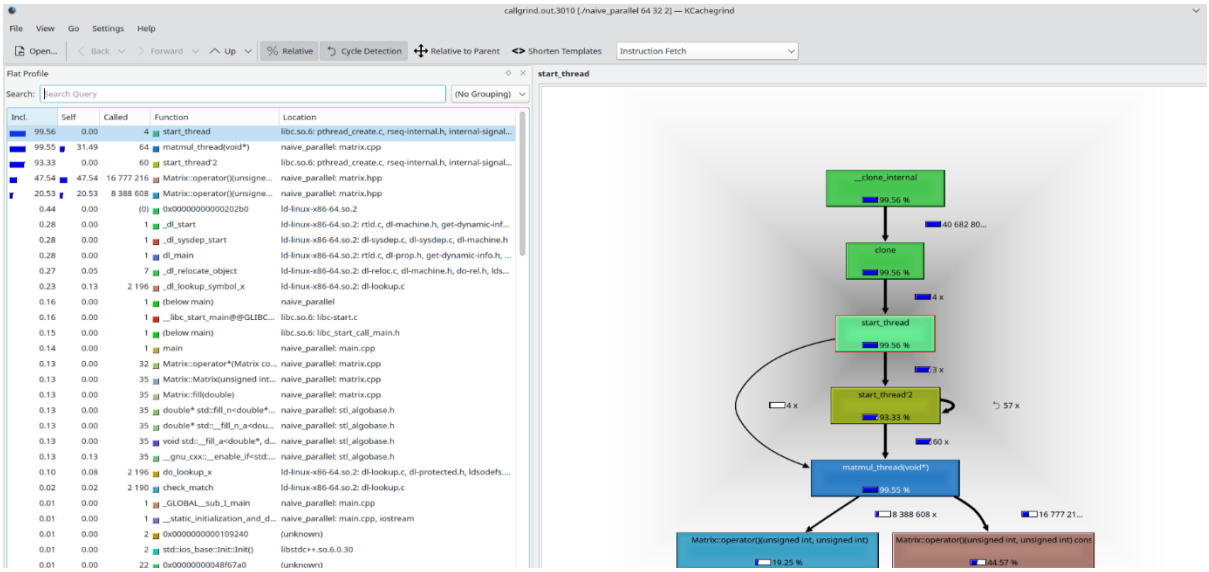


Figure 59: kcachegrind output of naïve_parallel 64 32 2

Naïve_parallel with 64 as dimension 4 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o.

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32 4.

Step-3: Kcachegrind callgrind.out.3088

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 64 32 4
==3088== Callgrind, a call-graph generating cache profiler
==3088== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3088== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3088== Command: ./naive_parallel 64 32 4
==3088==
==3088== For interactive control, run 'callgrind_control -h'.
==3088==
==3088== Events    : Ir
==3088== Collected : 776518114
==3088==
==3088== I   refs:      776,518,114
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 60: valgrind output of naïve_parallel 64 32 4

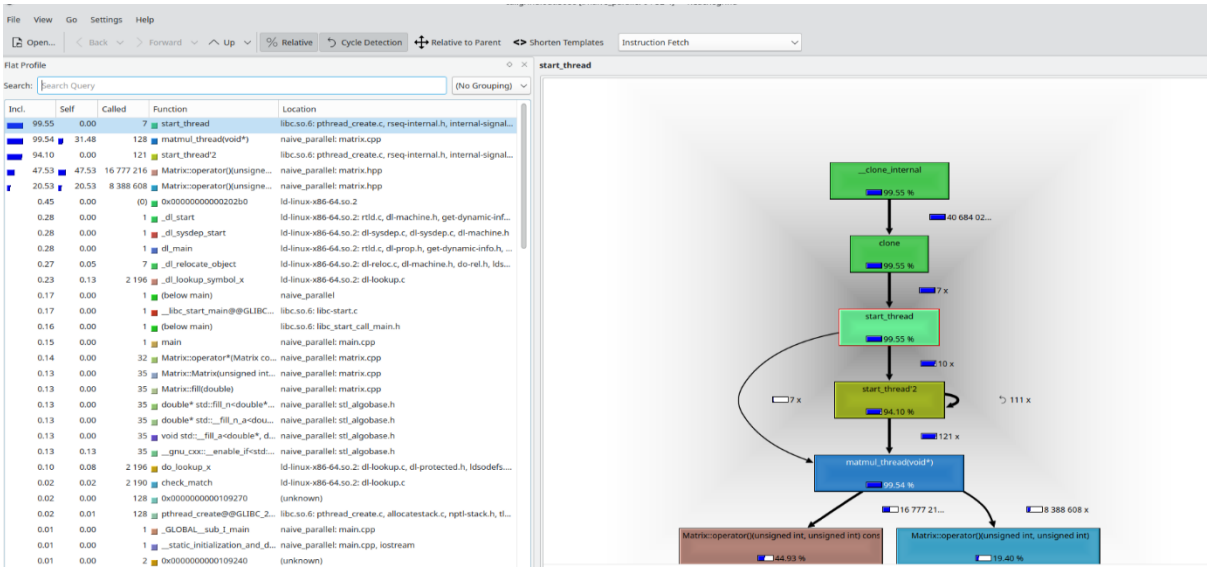


Figure 61: kcachegrind output of naïve_parallel 64 32 4

Naïve_parallel with 64 as dimension 8 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32 8.

Step-3: Kcachegrind callgrind.out.3230

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 64 32 8
==3230== Callgrind, a call-graph generating cache profiler
==3230== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3230== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3230== Command: ./naive_parallel 64 32 8
==3230==
==3230== For interactive control, run 'callgrind_control -h'.
==3230==
==3230== Events    : Ir
==3230== Collected : 776787801
==3230==
==3230== I   refs:      776,787,801
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 62 : Valgrind output of naïve_parallel 64 32 8

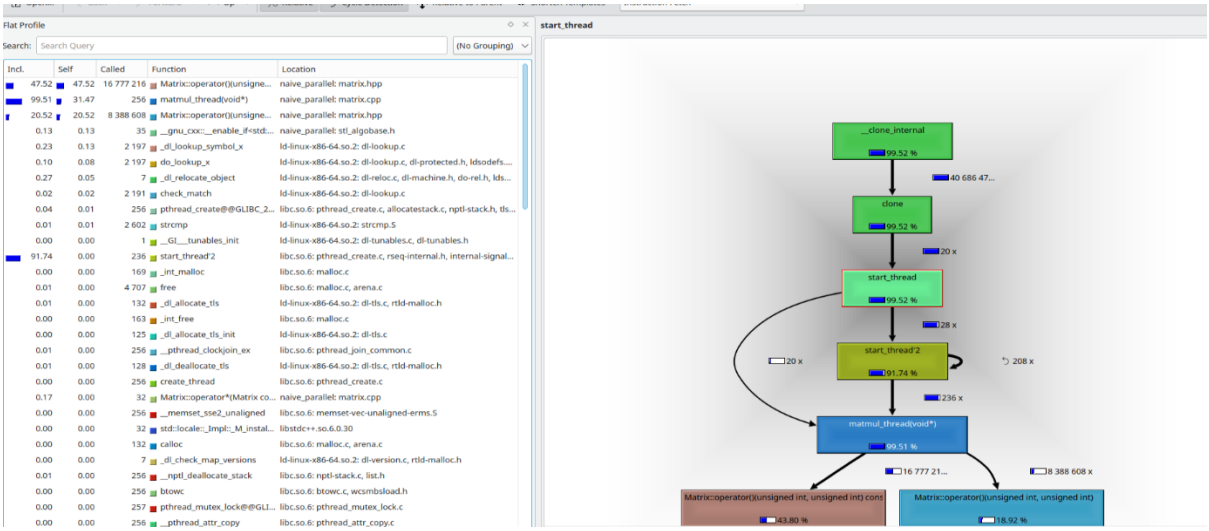


Figure 63:kcachegrind output of naïve_parallel 64 32 8

Naïve_parallel with 64 as dimension 16 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32 16.

Step-3: Kcachegrind callgrind.out.3499

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 64 32 16
==3499== Callgrind, a call-graph generating cache profiler
==3499== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3499== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3499== Command: ./naive_parallel 64 32 16
==3499==
==3499== For interactive control, run 'callgrind_control -h'.
==3499==
==3499== Events    : Ir
==3499== Collected : 777333235
==3499==
==3499== I   refs:      777,333,235
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 64: Valgrind output of naïve_parallel 64 32 16

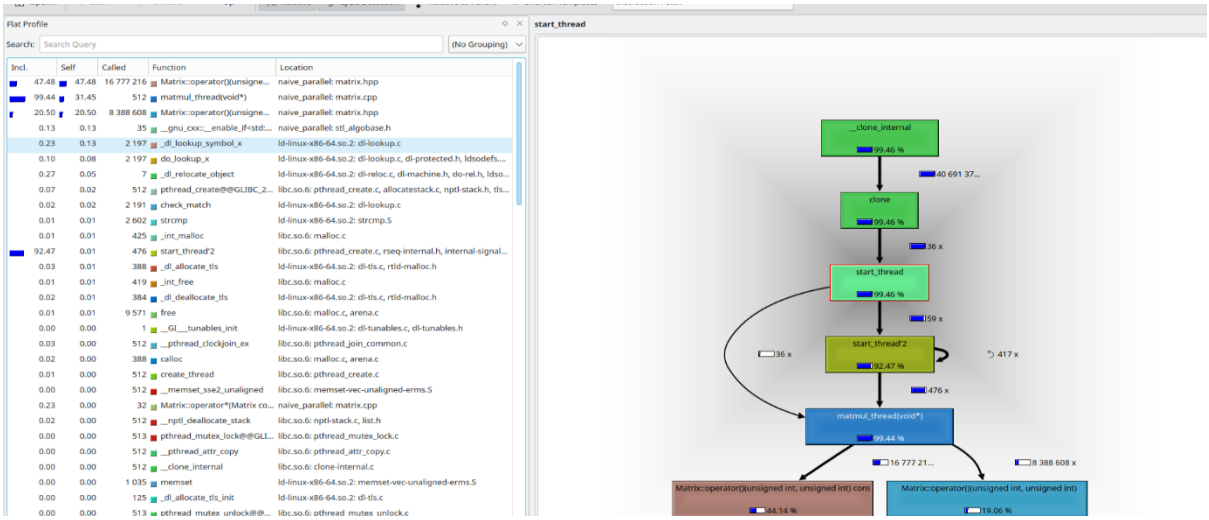


Figure 65: kcachegrind output of naïve_parallel 64 32 16

Naïve_parallel with 64 as dimension 32 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32 32.

Step-3: Kcachegrind callgrind.out.4027

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 64 32 32
==4027== Callgrind, a call-graph generating cache profiler
==4027== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==4027== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4027== Command: ./naive_parallel 64 32 32
==4027==
==4027== For interactive control, run 'callgrind_control -h'.
==4027==
==4027== Events    : Ir
==4027== Collected : 778416027
==4027==
==4027== I   refs:      778,416,027
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 66: Valgrind output of naïve_parallel 64 32 32

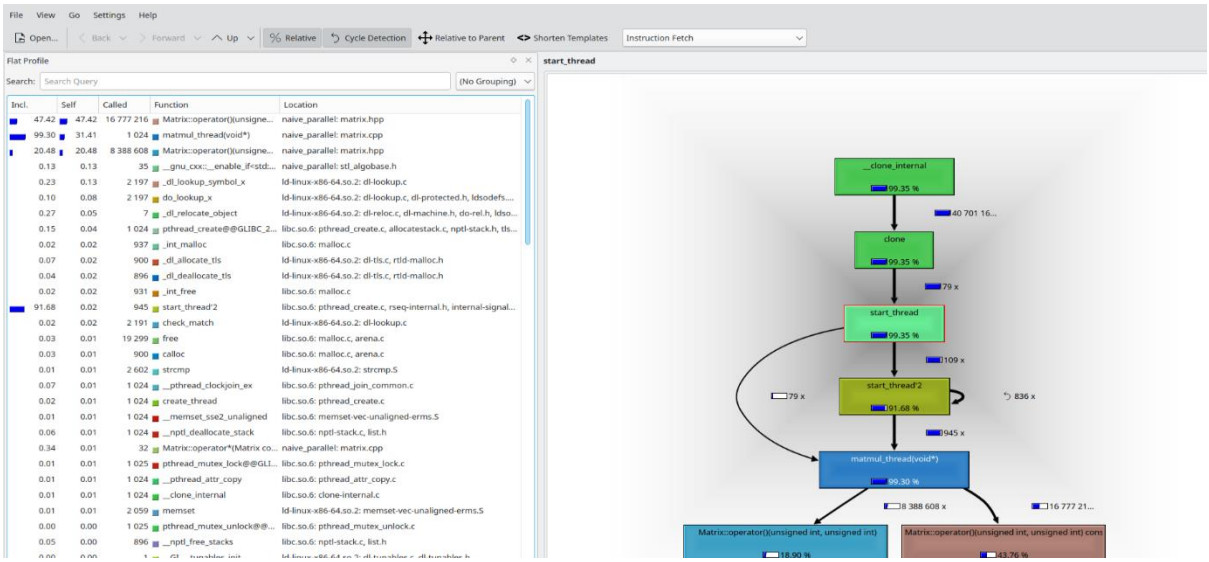


Figure 67: kcachegrind output of naïve_parallel 64 32 32.

Naïve_parallel with 128 as dimension 1 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 128 32 1.

Step-3: Kcachegrind callgrind.out.7374

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 128  32 1
==7349== Callgrind, a call-graph generating cache profiler
==7349== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7349== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7349== Command: ./naive_parallel 128 32 1
==7349==
==7349== For interactive control, run 'callgrind_control -h'.
==7349==
==7349== Events    : Ir
==7349== Collected : 6185191003
==7349==
==7349== I   refs:      6,185,191,003
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 68: valgrind output of naïve_parallel 128 32 1

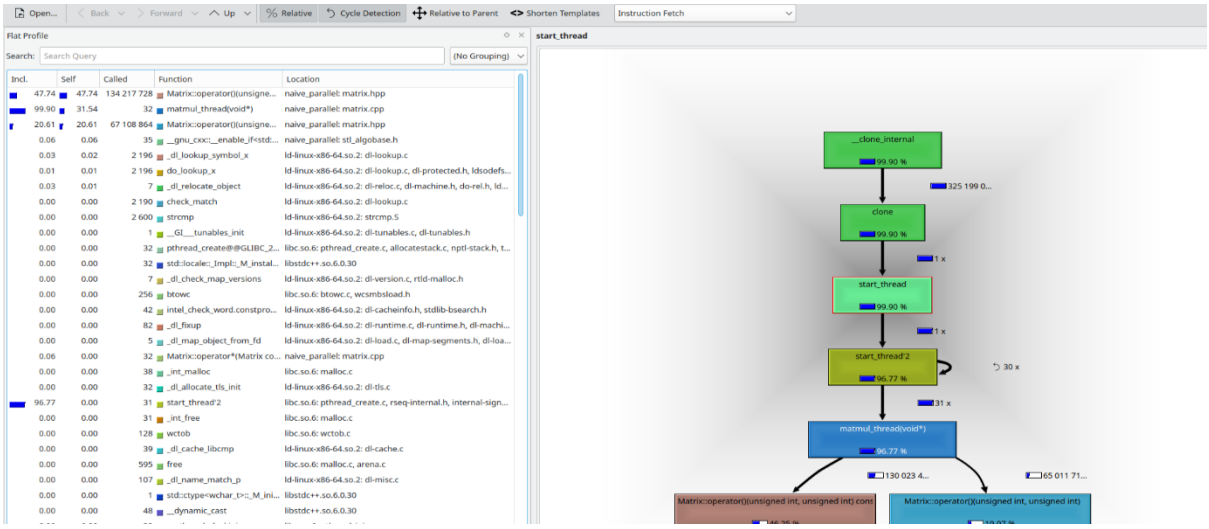


Figure 69: kcachegrind output of naïve_parallel 128 32 1

Naïve_parallel with 128 as dimension 2 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 128 32 2.

Step-3: Kcachegrind callgrind.out.7271

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 128  32 2
==7271== Callgrind, a call-graph generating cache profiler
==7271== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7271== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7271== Command: ./naive_parallel 128 32 2
==7271==
==7271== For interactive control, run 'callgrind_control -h'.
==7271==
==7271== Events    : Ir
==7271== Collected : 6185243941
==7271==
==7271== I   refs:      6,185,243,941
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 70: valgrind output of naïve_parallel 128 32 2

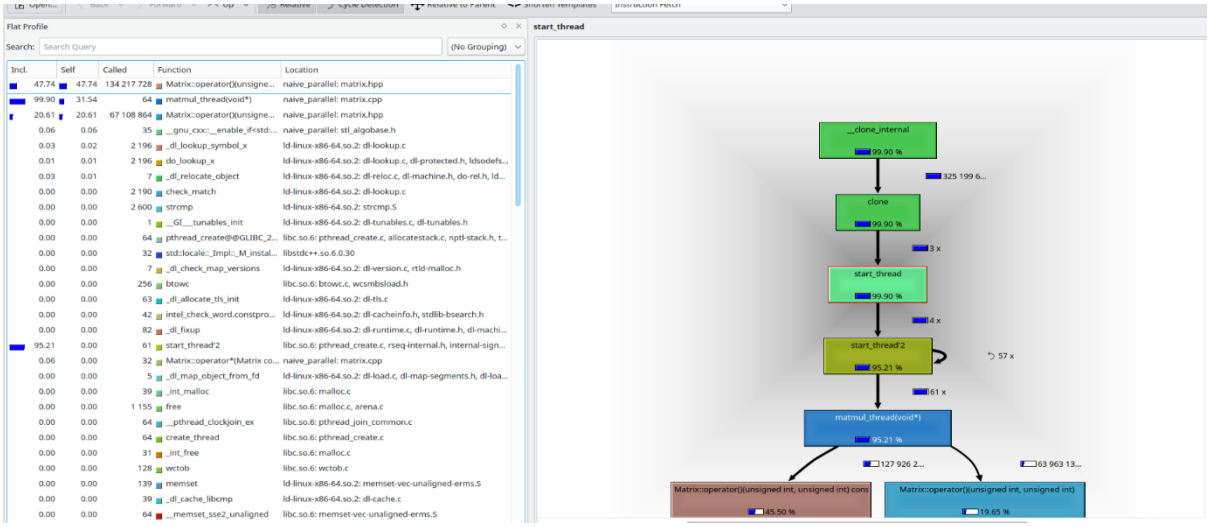


Figure 71: kcachegrind output of naïve_parallel 128 32 2

Naïve_parallel with 128 as dimension 4 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 128 32 4.

Step-3: Kcachegrind callgrind.out.7120

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 128  32 4
==7120== Callgrind, a call-graph generating cache profiler
==7120== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7120== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7120== Command: ./naive_parallel 128 32 4
==7120==
==7120== For interactive control, run 'callgrind_control -h'.
==7120==
==7120== Events    : Ir
==7120== Collected : 6185349956
==7120==
==7120== I   refs:      6,185,349,956
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 72: valgrind output of naïve_parallel 128 32 4

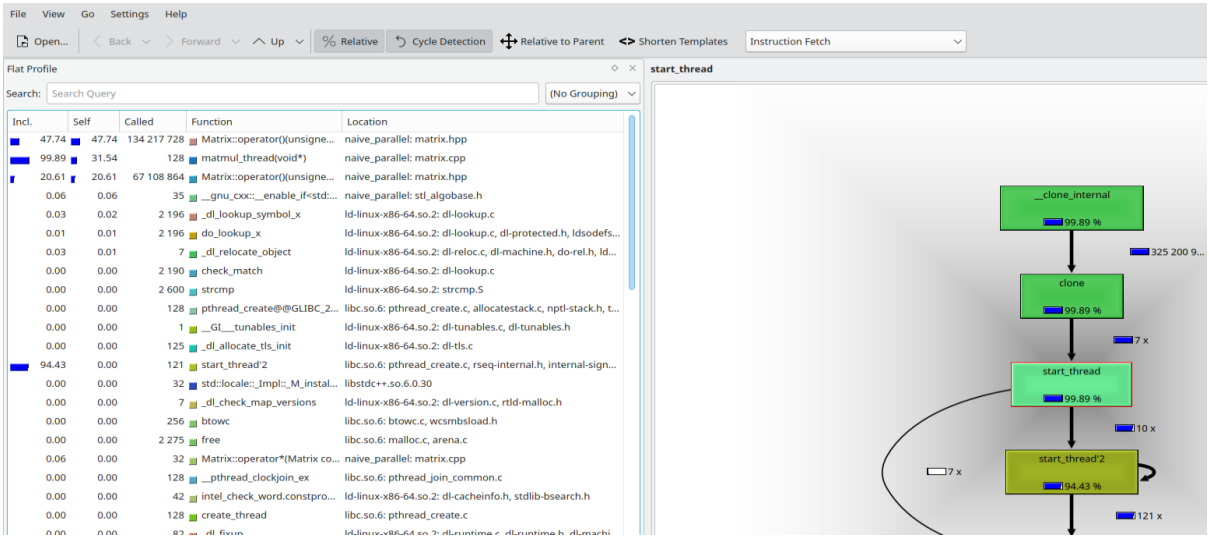


Figure 73: kcachegrind output of naïve_parallel 128 32 4

Naïve_parallel with 128 as dimension 8 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind. / naïve_parallel 128 32 8.

Step-3: Kcachegrind callgrind.out.6864

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 128  32 8
==6849== Callgrind, a call-graph generating cache profiler
==6849== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==6849== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6849== Command: ./naive_parallel 128 32 8
==6849==
==6849== For interactive control, run 'callgrind_control -h'.
==6849==
==6849== Events    : Ir
==6849== Collected : 6185626333
==6849==
==6849== I   refs:      6,185,626,333
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 74: valgrind output of naïve_parallel 128 32 8

Naïve_parallel with 128 as dimension 16 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 128 32 16.

Step-3: Kcachegrind callgrind.out.6104

```
CallGraphView::dotExited: QProcess 0x56031b581eb0
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 128  32 16
==6104== Callgrind, a call-graph generating cache profiler
==6104== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==6104== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6104== Command: ./naive_parallel 128 32 16
==6104==
==6104== For interactive control, run 'callgrind_control -h'.
==6104==
==6104== Events    : Ir
==6104== Collected : 6186178443
==6104==
==6104== I   refs:      6,186,178,443
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 75: valgrind output of naïve_parallel 128 32 16

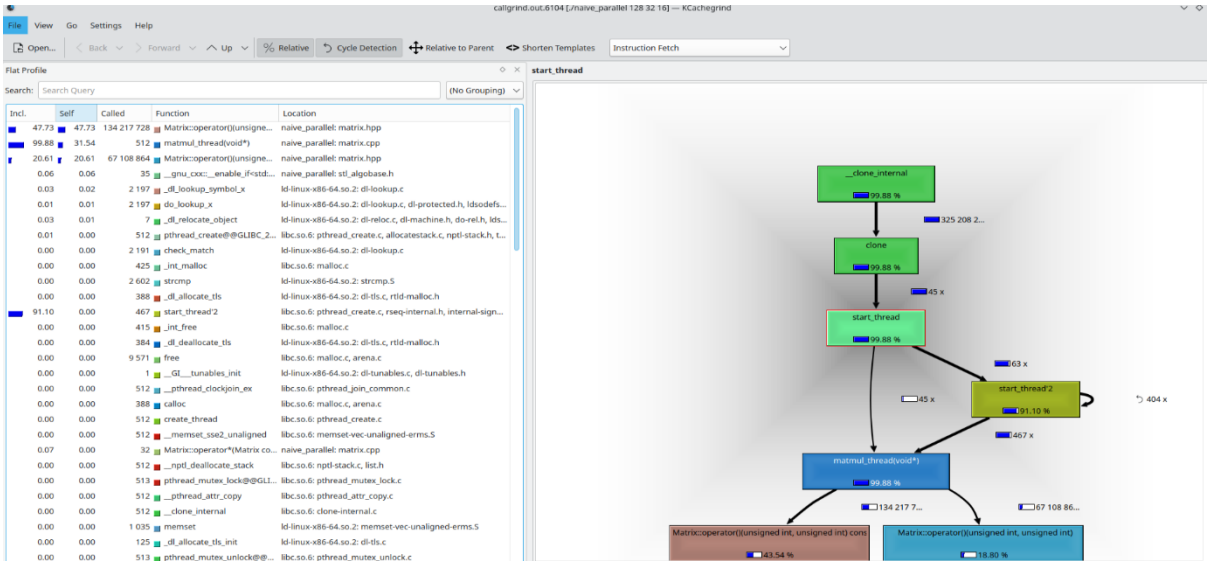


Figure 76: kcachegrind output of naïve_parallel 128 32 16

Naïve_parallel with 128 as dimension 32 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 128 32 32.

Step-3: Kcachegrind callgrind.out.5066

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 128  32 32
==5066== Callgrind, a call-graph generating cache profiler
==5066== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==5066== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5066== Command: ./naive_parallel 128 32 32
==5066==
==5066== For interactive control, run 'callgrind_control -h'.
==5066==
==5066== Events    : Ir
==5066== Collected : 6187255883
==5066==
==5066== I   refs:      6,187,255,883
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 77: valgrind output of naïve_parallel 128 32 32

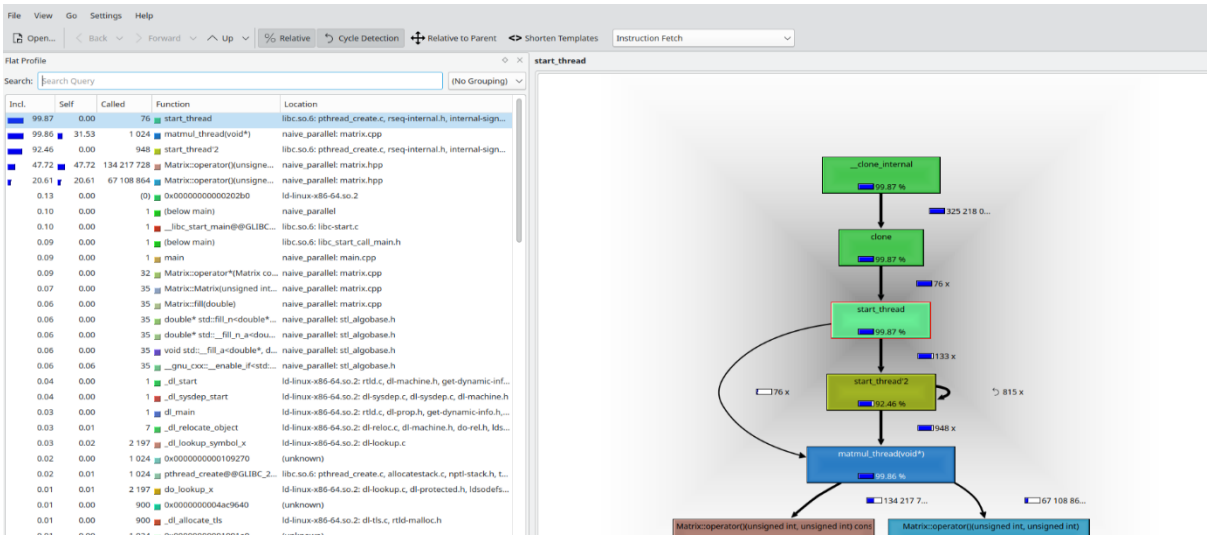


Figure 78:kcachegrind output of naïve_parallel 128 32 32

Naïve_parallel with 256 as dimension 1 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 256 32 1.

Step-3: Kcachegrind callgrind.out.7402

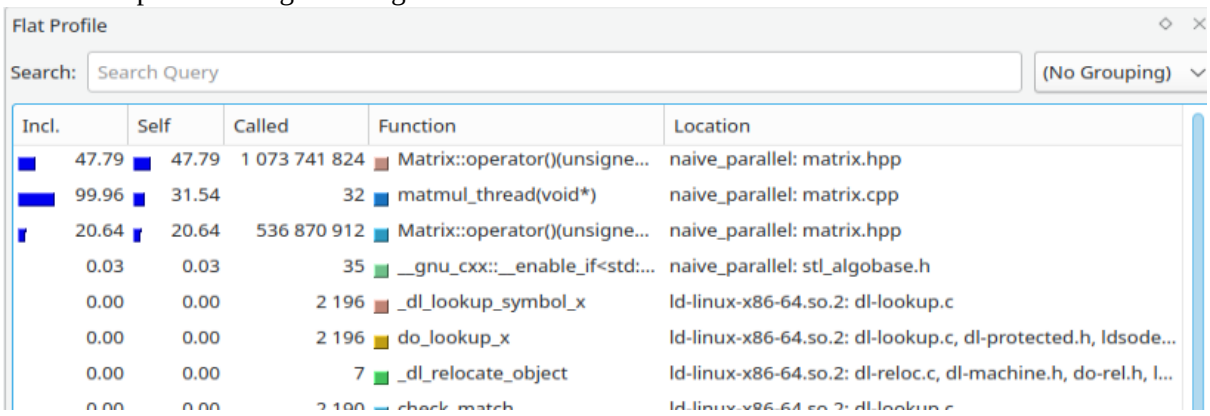


Figure 79: Valgrind output of naïve_parallel 256 32 1

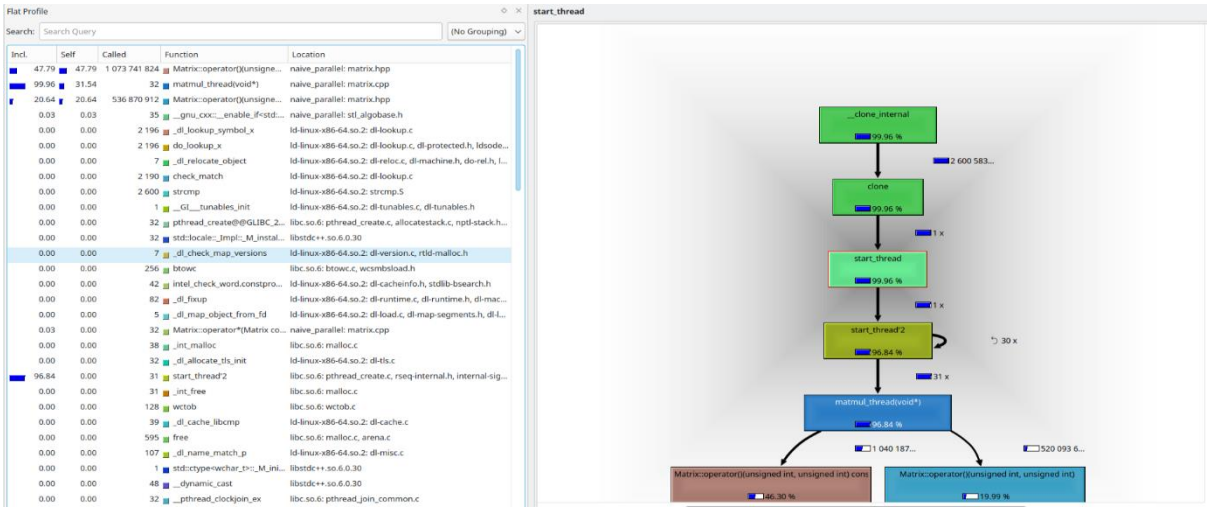


Figure 80: Kcachegrind output of naïve_parallel 256 32 1

Naïve_parallel with 256 as dimension 2 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 256 32 2.

Step-3: Kcachegrind callgrind.out.7478

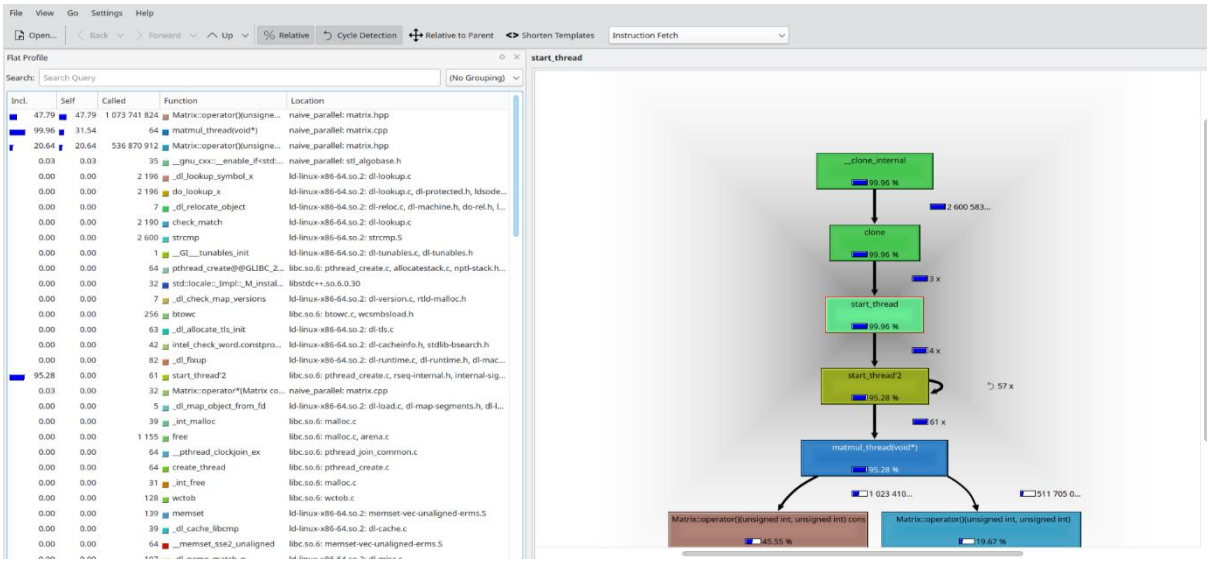


Figure 81: kcachegrind output of naïve_parallel 256 32 2

Naïve_parallel with 256 as dimension 4 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 64 32.

Step-3: Kcachegrind callgrind.out.7562

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 256  32 4
==7562== Callgrind, a call-graph generating cache profiler
==7562== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7562== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7562== Command: ./naive_parallel 256 32 4
==7562==
==7562== For interactive control, run 'callgrind_control -h'.
==7562==
==7562== Events    : Ir
==7562== Collected : 49429694702
==7562==
==7562== I   refs:      49,429,694,702
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 82: valgrind output of naïve_parallel 256 342 4

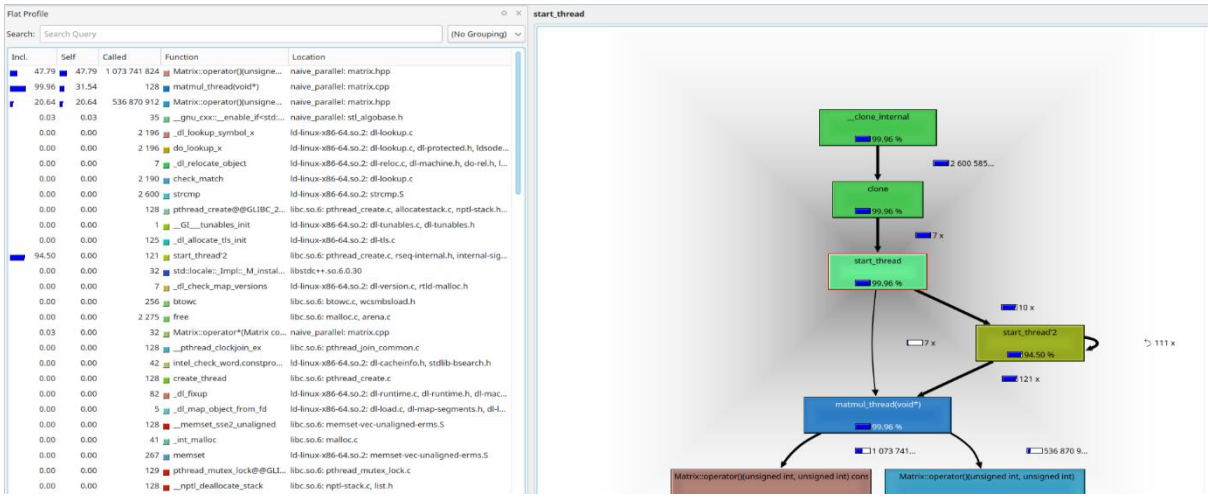


Figure 83: kcachegrind output of naïve_parallel 256 32 4

Naïve_parallel with 256 as dimension 8 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 256 32 8.

Step-3: Kcachegrind callgrind.out.1928

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 256 32 8
==1928== Callgrind, a call-graph generating cache profiler
==1928== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==1928== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1928== Command: ./naive_parallel 256 32 8
==1928==
==1928== For interactive control, run 'callgrind_control -h'.
==1928==
==1928== Events    : Ir
==1928== Collected : 49429970159
==1928==
==1928== I   refs:      49,429,970,159
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 84: valgrind output of naïve_parallel 256 32 8

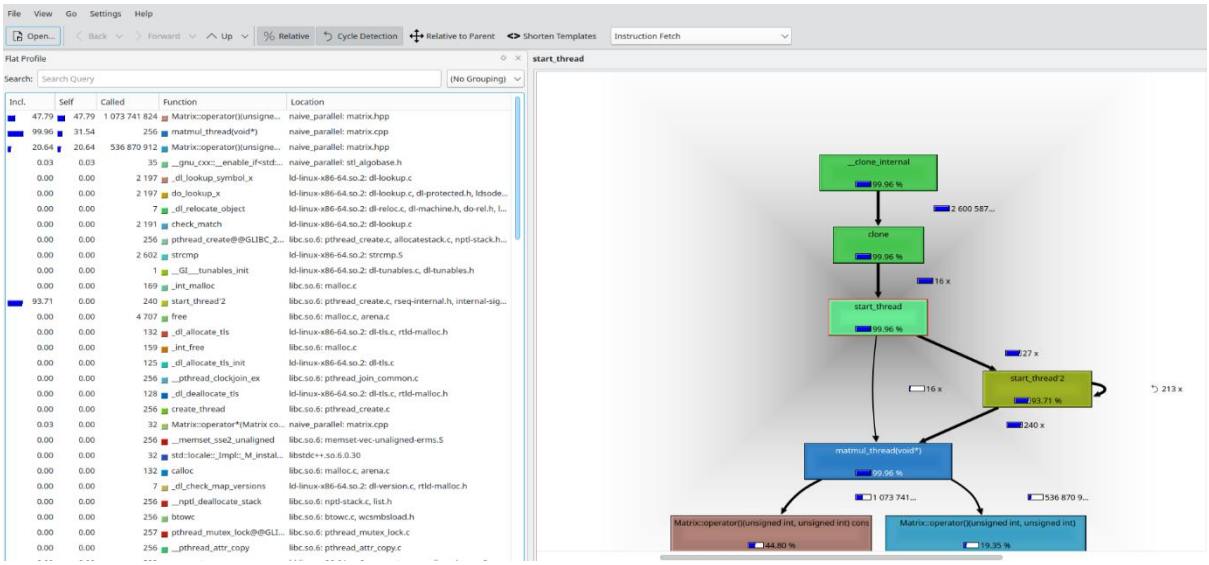


Figure 85: kcachegrind output of naïve_parallel 256 32 8

Naïve_parallel with 256 as dimension 16 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 256 32 16.

Step-3: Kcachegrind callgrind.out.2210

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 256 32 16
==2210== Callgrind, a call-graph generating cache profiler
==2210== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2210== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2210== Command: ./naive_parallel 256 32 16
==2210==
==2210== For interactive control, run 'callgrind_control -h'.
==2210==
==2210== Events    : Ir
==2210== Collected : 49430524978
==2210==
==2210== I   refs:      49,430,524,978
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 86: valgrind output of naïve_parallel 0f 256 32 16

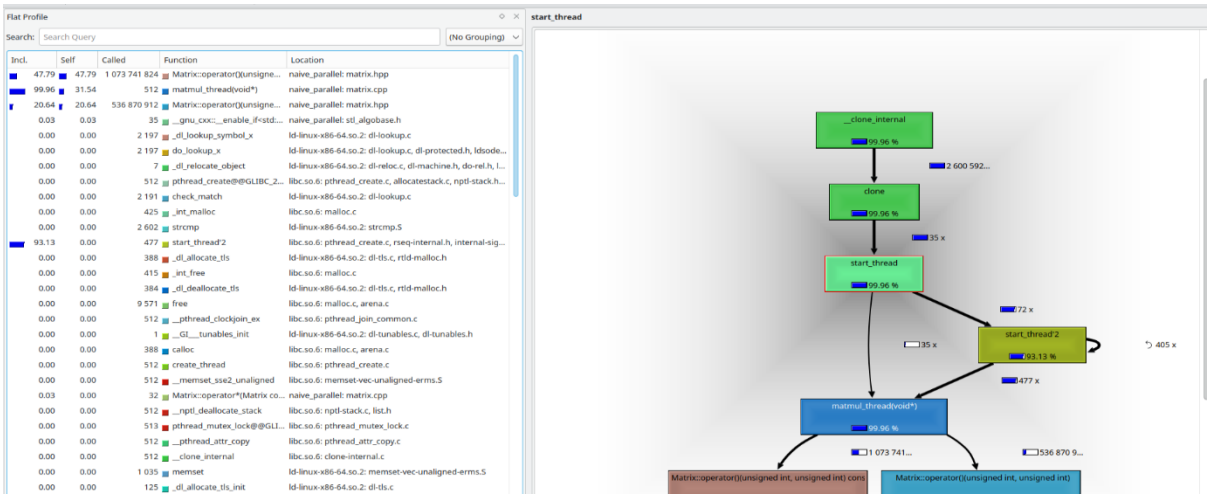


Figure 87: kcachegrind output of naïve_parallel of 256 32 16

Naïve_parallel with 256 as dimension 32 as thread.

Step1: g++ -std=c++17 -g -c DNAIVE_PARALLEL matrix.cpp -o matrix_naive_parallel.o

Step-2: valgrind –tool=callgrind./ naïve_parallel 256 32 32.

Step-3: Kcachegrind callgrind.out.2742

```
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$ valgrind --tool=callgrind ./naive_parallel 256 32 32
==2742== Callgrind, a call-graph generating cache profiler
==2742== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2742== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2742== Command: ./naive_parallel 256 32 32
==2742==
==2742== For interactive control, run 'callgrind_control -h'.
==2742==
==2742== Events    : Ir
==2742== Collected : 49431606371
==2742==
==2742== I   refs:      49,431,606,371
dv1567@dv1567-VirtualBox:~/Downloads/code/matrix_multiplication$
```

Figure 88:valgrind output of 256 32 32

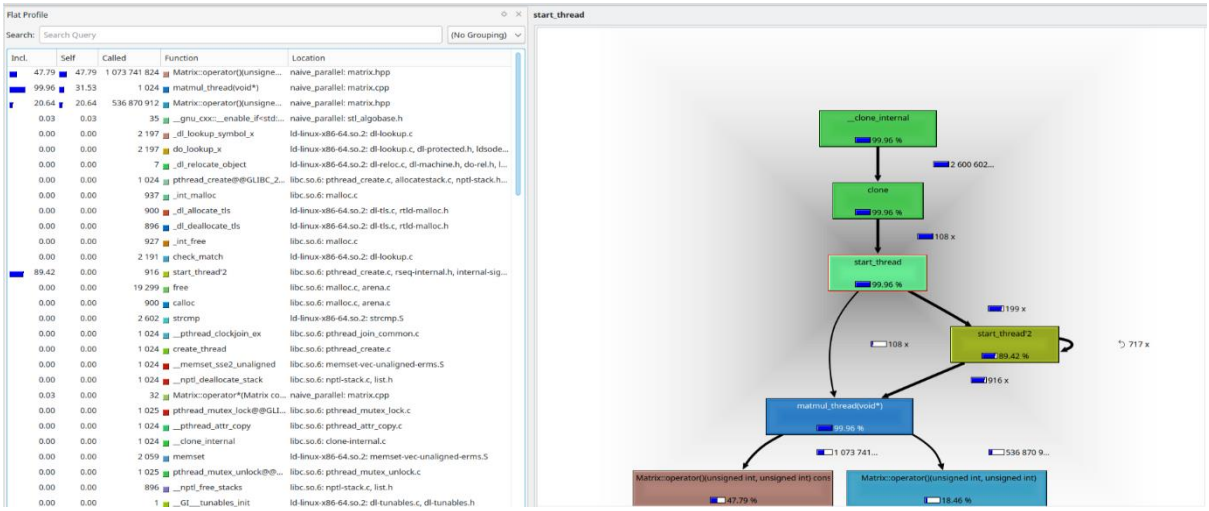


Figure 89: kcachegrind output of naïve_parallel 256 32 32

2) Present your hypothesis about what the hotspot of the application is.

To locate the application's hotspot, run kcachegrind with the application's "process id." The kcachegrind output is split into three sections: "flat profile," "upper pane," and "lower pane." "Inclusive time", "self", "called", "function", and "location" can be seen in the flat profile. We can see "Types", "Callers", "All callers", "Caller map", and "Source code" in the upper pane. "Machine code" and "call graph" can be found in the lower pane. We must concentrate on "self," "called," "Source code," and "Call graph" throughout the Kcachegrind output.

Self: Deal with the amount of time spent by the specific function in that program. If we spend more time on the self, we can expect it to be the place where we can optimize.

Called: This property deals with how frequently a function is called and is displayed in numerical values.

Source Code: Display the source code of the selected function in the self; if we chose an internal function related to Linux, we will not see any code under the source code section.

Call graph: It is a graphical/tree representation of the entire source code that includes the inclusive time of a specific function, with the darkness of the arrow marks indicating the function's importance.

File_reader:

File_reader Hypothesis: After running Kcachegrind, we can see that the highest self in the program is associated with an internal Linux system function that cannot be changed to improve program performance. So I thought about the functions that are related to the program, and then I noticed the "file reader[abi::cxx11]" function, which could be considered the file reader application's hotspot.

File_reader_parsimonious Hypothesis: The Kcachegrind is first executed, and then the self is selected in flat profile to arrange according to self. After that, we can see that in the program, the highest self is for the “iterator's” operation, and in the function, all of the first six self are occupied by the iterator's operation, so we can consider this the application's hotspot.

File_reader_sgetn Hypothesis: Initially, Kcachegrind is run, and then the self is chosen in flat profile so that it can be arranged according to self. After that, we can see that in the program, the highest self is linked to an internal Linux system function that cannot be changed in order to improve the program's performance, and we can also see that the entire flat profile is occupied by the internal Linux functions. We can only see "file reader sgetn(std::istreams) - file reader sgetn: file reader sgetn.cpp" in 2 places, thus those 2 locations can be regarded as the application's hotspots.

File_reader_read Hypothesis: First, Kcachegrind is run, and then the self is chosen in flat profile so that it can be arranged according to self. After that, we can see that in the program, the highest self is linked to an internal Linux system function that cannot be changed to improve program performance, and we can also see that all of flat profile is taken up by internal Linux functions. We can only see "file reader read(std::istreams) - file reader read: file reader read.cpp" in 2 places, thus those 2 locations can be regarded as the application's hotspots.

Matrix_multiplication:

Naïve Hypothesis:

When the naive program is performed with multiple dimensions, we can see the change in performance and the hotspot. This file's naive version is executed using different dimensions such "64, 128, 256, 512" with 32 test runs.

“Matrix::operator()(unsigned int, unsigned int)const” and “Matrix::operator()(unsigned int, unsigned int)” are the two functions that have the highest self for all variations of Naive after running kcachegrind, hence these two functions may be considered the application's hotspot.

Naïve _Parallel Hypothesis:

When the naïve_parallel program is performed with multiple dimensions, and multiple threads we can see the change in performance and the hotspot. This file's naive _parallel version is executed using different dimensions such "64, 128, 256" and different “1,2, 4, 8, 16, 32” with 32 test runs.

“Matrix::operator()(unsigned int, unsigned int)const” and “Matrix::operator()(unsigned int, unsigned int)” are the two functions that have the highest self for all variations of Naive after running kcachegrind, hence these two functions may be considered the application's hotspot.

Strassen_Parallel Hypothesis:

When the Strassen_parallel program is performed with multiple dimensions, we can see the change in performance and the hotspot. This file's naive version is executed using different dimensions such "64, 128, 256, 512" with 32 test runs.

“Matrix::operator()(unsigned int, unsigned int)const” and “Matrix::operator()(unsigned int, unsigned int)” are the two functions that have the highest self for all variations of Naive after running kcachegrind, hence these two functions may be considered the application's hotspot.

Log_analyzer:

Analyze_log hypothesis: Initially, Kcachegrind is run, and then the self is chosen in flat profile so that it can be arranged according to self. After that, we can see that in the program, the highest self is linked

to an internal Linux system function that cannot be changed in order to improve the program's performance, so in my view the function which are related to the application and have highest self can be considered as hotspot, so the functions “iterator” or “map” can be considered as the hotspots of the application.

Analyze_charbuf hypothesis: Initially, Kcachegrind is run, and then the self is chosen in flat profile so that it can be arranged according to self. After that, we can see that in the program, the highest self is linked to an internal Linux system function that cannot be changed in order to improve the program's performance, so in my view the function which are related to the application and have highest self can be considered as hotspot, so the function related “charbuf <10u” in the location “analyze_charbuf charbuf.hpp” can be considered as the hotspots of the application.

Analyze_hm_push hypothesis: Initially, Kcachegrind is run, and then the self is chosen in flat profile so that it can be arranged according to self. After that, we can see that in the program, the highest self is linked to an internal Linux system function that cannot be changed in order to improve the program's performance, so in my view the function which are related to the application and have highest self can be considered as hotspot, so the function “log_entry” in the location “ analyze_log_entry(char const” can be considered as the hotspots of the application.

3)Monitor and Motivate the hotspots of the Applications.

To motivate the hotspot of the applications all the applications flat profile is taken into consideration.

File_reader

File_reader: from the figure90 we can observe that “file_reader[abi:cxx11] char” has “32” calls with “98.30” with inclusive and “0.00” with self even though it has 0.00 self but it is considered as the hotspot because it is opening the same file 32 times.

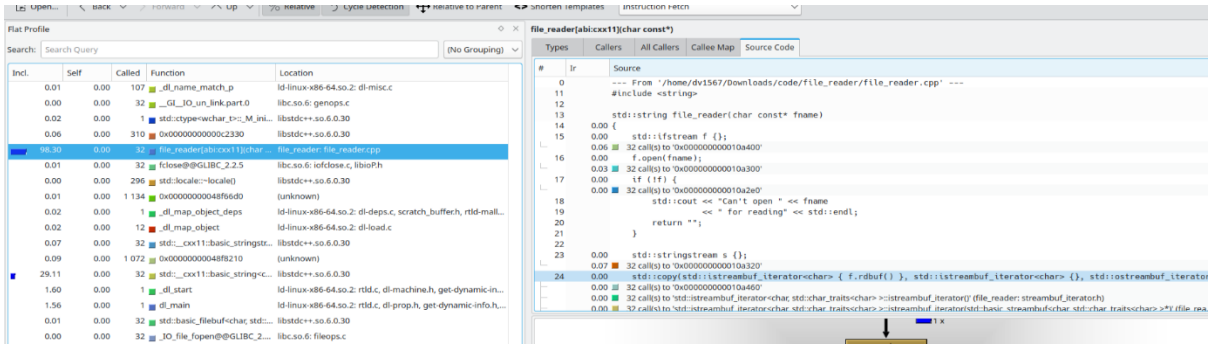


Figure90: hotspot of file_reader.

File_reader_read:

From the below figure 91 we can observe that the functions “stream_size()(std::stream)” and “ file_reader(std::istream &std_)” have 0.05 and 0.04 self respectively which are considered as the hotspot of the application.

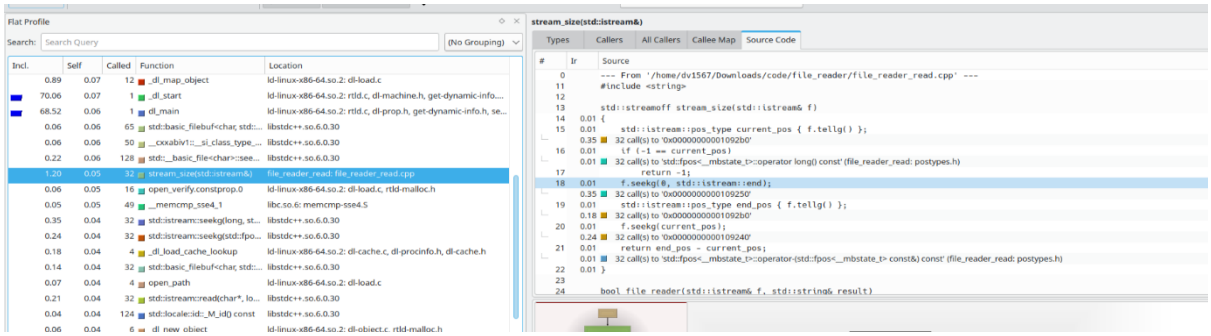


Figure91: hotspot of file_reader_read

File_reader_sgetn:

From the below figure 92 we can observe that the functions “stream_size()(std::stream)” and “ file_reader(std::istream &std_)” have 0.05 and 0.04 self respectively which are considered as the hotspot of the application.

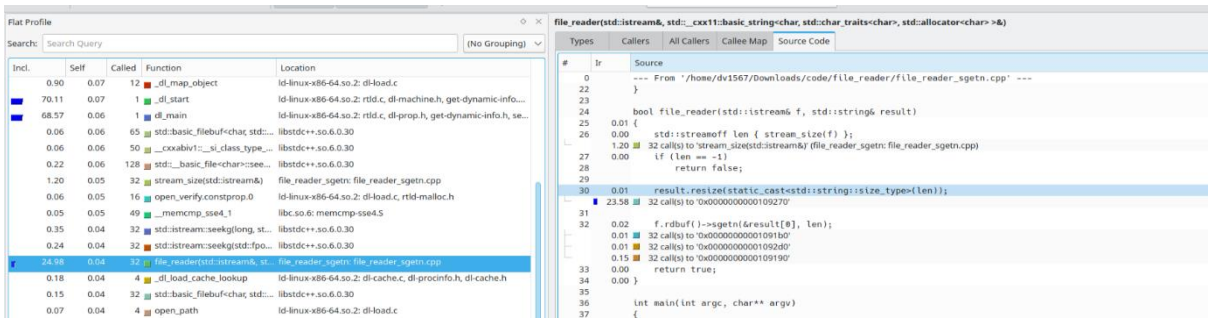


Figure 92: hotspot of file_reader_sgetn

File_reader_parsimonious:

From the figure 93 we can observer that the functions “std:istreambuf_iterator<char,std:: char>” with 25.05 self and 40152332 number of called so, this is the hotspot of the application.

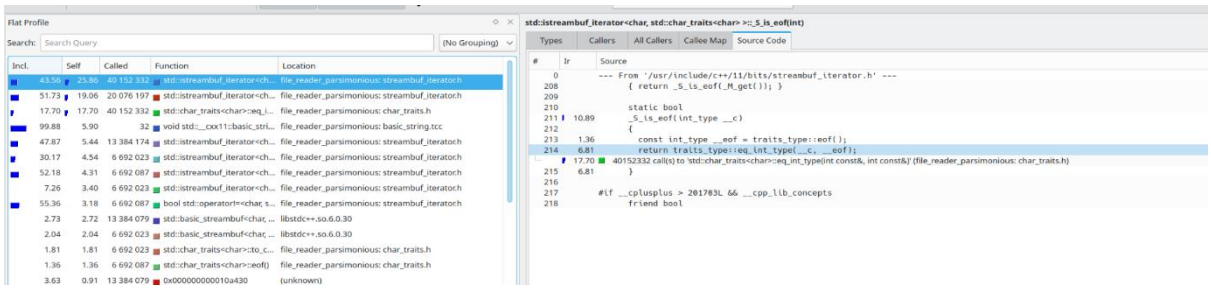


Figure 93: hotspot of file_reader_parsimonious

Matrix_multiplication

Naïve: For all the dimensions of naïve application the time spend on the functions “matrix::operator()(unsigned int& unsigned int)const” and “matrix::operator()(unsigned int &unsigned int)” is more and the number of calls are also mores so these function are considered as the hotspot of the application. This can be observed from Table -1 and Figure94.

The below table will represent the time spend on the functions by different dimension of naïve application.

Application name	Dimensions	Funtion-1	Function-2
Naïve	64 32	41.48	24.02
	128 32	41.67	24.12
	256 32	41.72	24.15
	512 32	41.72	24.16

Table 1: amount of time spends by different dimension in Naïve

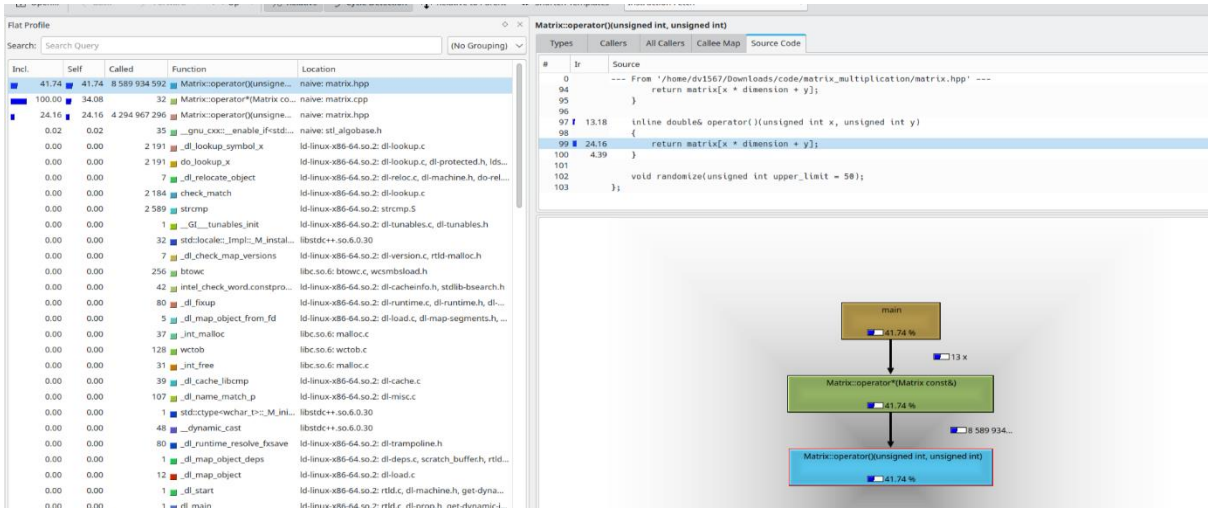


Figure 94: hotspot of naïve application.

Strassen_parallel: For all the dimensions of Strassen_parallel application the time spend on the functions “matrix::operator()(unsigned int& unsigned int)const” and “matrix::operator()(unsigned int &unsigned int)” is more and the number of calls are also mores so these function are considered as the hotspot of the application. This can be observed from Table -2 and Figure 95.

The below table will represent the time spend on the functions by different dimension of Strassen_parallel application.

Application name	Dimensions	Funtion-1	Function-2
Strassen_parallel	1024 4	51.40	14.09
	64 32	49.61	14.71
	128 32	50.72	14.37
	256 32	51.07	14.23
	512 32	51.03	14.13

Table 2: amount of spend by different dimension of Strassen_parallel

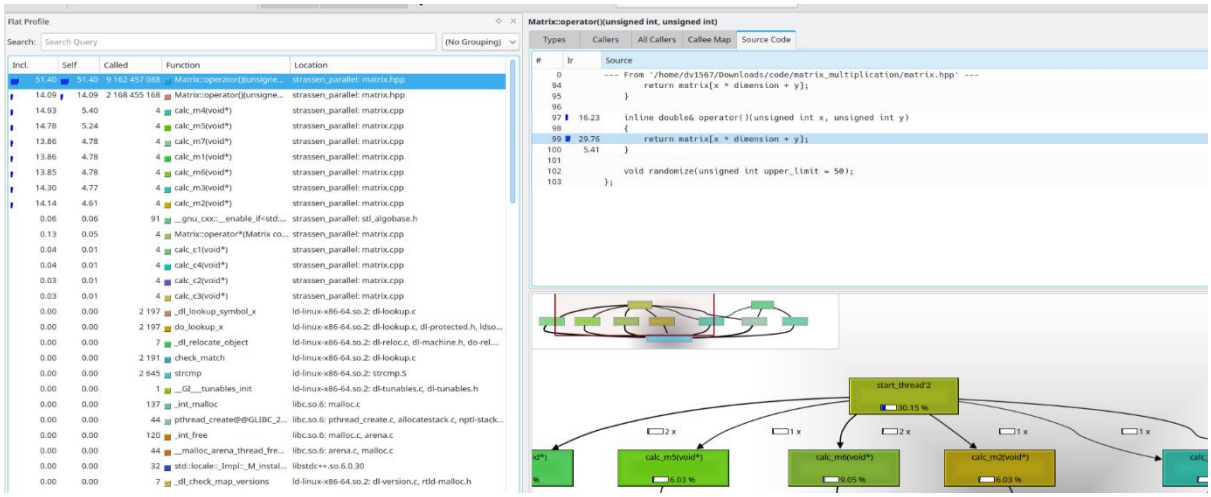


Figure 95: hotspot of the Strassen_parallel application

Naïve_parallel: For all the dimensions and threads of naïve_parallel application the time spend on the functions “matrix :: operator()(unsigned int& unsigned int)const” and “matrix::operator()(unsigned int &unsigned int)” is more and the number of calls are also more so these function are considered as the hotspot of the application. This can be observed from Table -3, Table-4 , Table -5 and Figure 96.

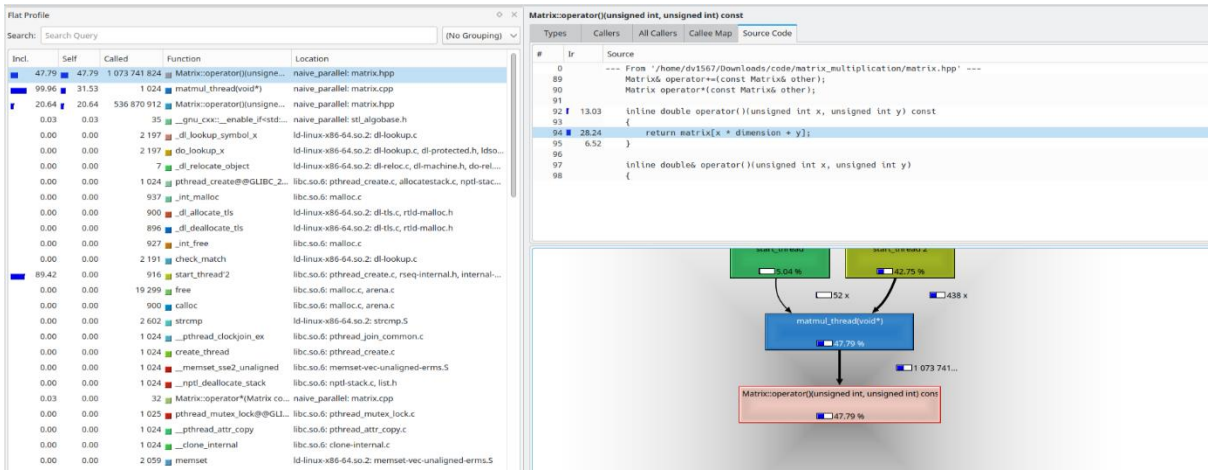


Figure 96: hotspot of naïve_parallel application

Application name	Dimensions	Funtion-1	Function-2
Naïve_Parallel	64 32 1	47.54	20.53
	64 32 2	47.54	20.53
	64 32 4	47.53	20.53
	64 32 8	47.52	20.52
	64 32 16	47.48	20.50
	64 32 32	47.42	20.48

Table 3: amount of time spend by naïve_parallel with 64 dimension.

Application name	Dimensions	Funtion-1	Function-2
Naïve_Parallel	128 32 1	47.74	20.61
	128 32 2	47.74	20.61
	128 32 4	47.74	20.61
	12832 8	47.74	20.61
	128 32 16	47.73	20.61
	128 32 32	47.72	20.61

Table 4: Amount of time spend by naïve_parallel with 128 dimension.

Application name	Dimensions	Funtion-1	Function-2
Naïve_Parallel	256 32 1	47.79	20.64
	256 32 2	47.79	20.64
	256 32 4	47.79	20.64
	256 32 8	47.79	20.64
	256 32 16	47.79	20.64
	256 32 32	47.79	20.64

Table 5: Amount of time spend by naïve_parallel with 256 dimension.

Log _Analyzer:

Analyze_log: From the below figure 97 we can observe that the functions “iterators” and “map” have 1.41 and 0.48 is self, 5952 and 1696 are the called respectively which are considered as the hotspot of the application.

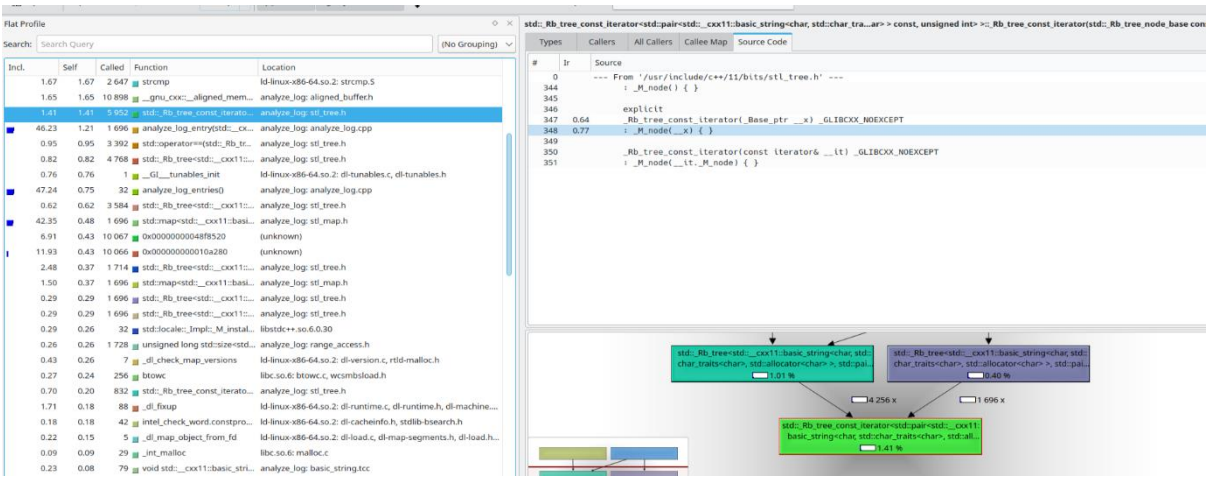


Figure 97: hotspot of the Analyze_log application.

Analyze_charbuf: From the below figure 98 we can observe that the functions “charbuf” have 6.80 self-time and 1722 called, when we see code, it is for loop which are considered as the hotspot of the application.

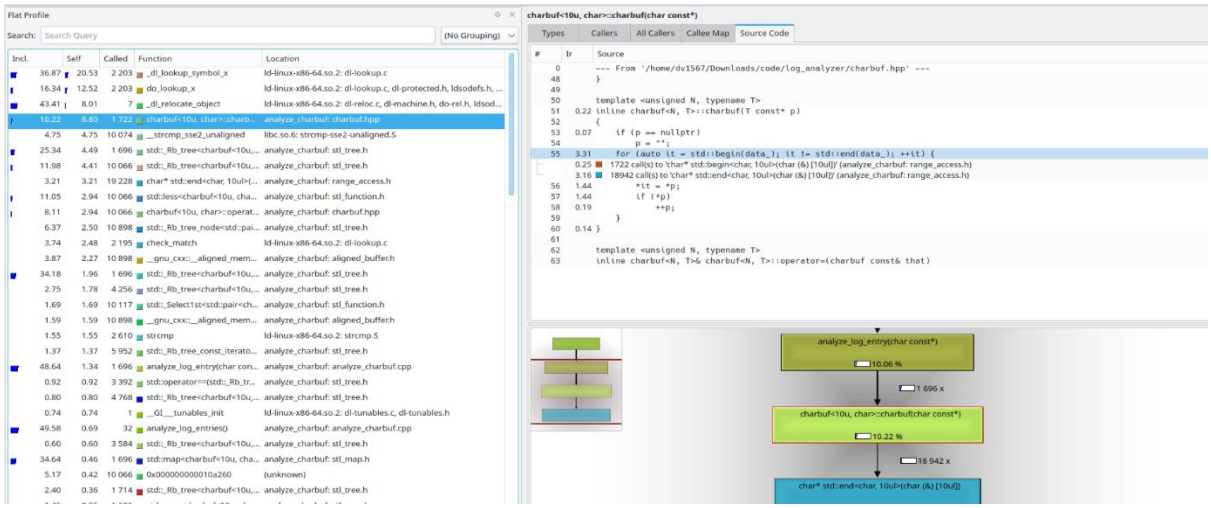


Figure 98: hotspot of the Analyze_charbuf

Analyze_pm_hash: From the below figure 99 we can observe that the functions “analyze_log_entry” have 2.09 self-time and 1696 called which is considered as the hotspot of the application.

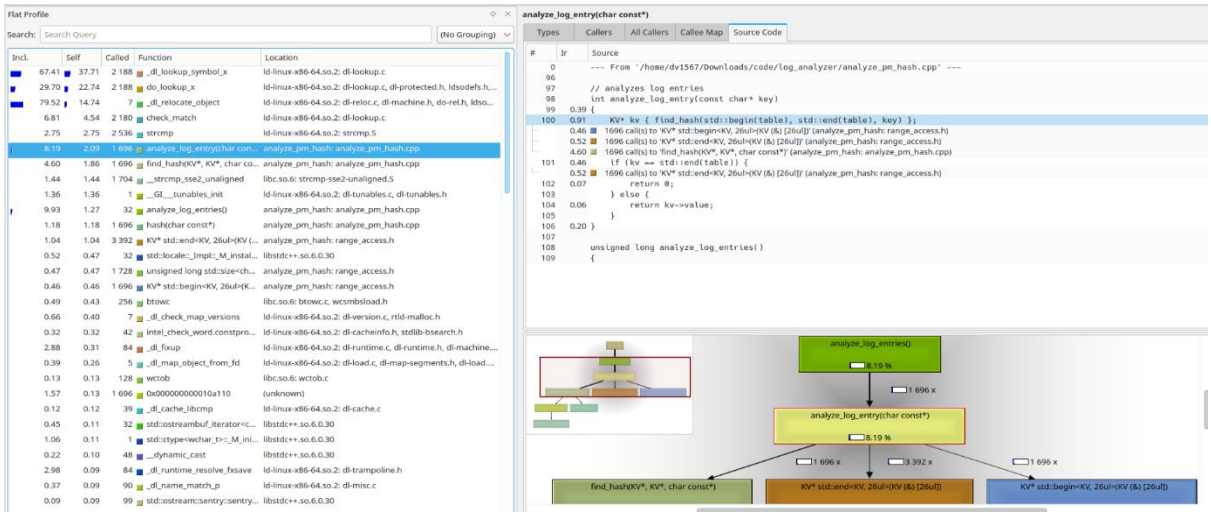


Figure 99: hotspot of the Analyze_pm_hash.

4.variation Comparison and Hypothesis of the variations

File_reader

The file_reader has 4 variations “file_reader”, “file_reader_read”, “file_reader_sgetn”, “file_reader_parsimonious”. All the 4 files are executed with 32 test runs and the different hotspots are found for each application.

From the analysis of 2nd question and from the figures 90,91,92,93 and from the table 6 I think file_reader is worst in case of performance because 32 times the same file need to be opened. File_reader_sgetn is better in terms of performance and CPU utilizations. Even the time of execution is calculated manually and from the table 7 we can observer the time of executions.

Log_analyzer:

The Analyzer _log has 3 variation “analyze_log”, “analyze_charbuf”, “analyze_pm_hash”. All the files are executed with 32 test runs and different hotspots are found for the application.

After the analysis from the question-2 and from the figure 97, 98,99 we can observe that both analyze_pm_hash and analyze_charbuf has one function under optimization category but I think analyze_pm_hash has less time in self that is 2.09 and a smaller number of called that is 1696 that can be observed from table- and even manually the time of execution is calculated in that case also we can say analyze_pm_hash has least time of execution. Which can be seen from table-. I think worst performance can be analyze_log it can also be seen from the table-8 and table-9.

Matrix_multiplication:

The matrix_multiplication has 3 variation “naïve”, “naïve_parallel”, “Strassen_parallel”. All the files are executed with different dimensions and the threads and the hotspot is found.

After the analysis from the question-2 and from the figure 94, 95,96 we can observe that “matrix::operator ()(unsigned int, unsigned int)const” is the hotspot for all the functions. From the tables 1, 2,3 4, 5 we can observer the different in amount of time spend by the same function.

I think “Naïve application” have better performance even with highest dimension also and Strassen parallel has worst performance even with least dimension also. From the analysis of the tables, we can also say that as the dimension increasing the performance decrease (inversely proportional).

5. Monitoring and motivating the Hypothesis.

File_reader:

Variations	Test runs	functions	Self	called
File_reader	32	file_reader[abi:cxx11]	0.0	32
File_reader_read	32	stream_size()(std::stream), file_reader(std::istream &std_	0.05 and 0.04	
File_reader_sgetn	32	stream_size()(std::stream), file_reader(std::istream &std_	0.05 and 0.04	32
File_reader_parsimonious	32	std:istreambuf_iterator<char,std:: char>	25.02	40152332

Table 6: Comparison of file_reader variations

Variations	Execution time
File_reader	0.91
File_reader_read	0.44
File_reader_sgetn	0.38
File_reader_parsimonious	0.69

Table 7: Execution times of the variations

Analyze_log:

Variation	Test_runs	functions	self	called
Analyze_log	32	Iterators, map	1.41, 0.48	5952, 1696
Analyze_charbuf	32	charbuf	6.80	1722
Analyze_pm_hash	32	analyze_log_entry	2.09	1696

Table 8: Comparison of Analyze_log variations

Variation	Execution time
Analyze_log	0.41
Analyze_charbuf	0.58
Analyze_pm_hash	0.38

Table-9: Execution times of variations

Matrix_multiplication:

Variations	Parameters	functions	self	called
Naïve	64 32	Matrtix::operator()(unsinged int & unsigned int)const	41.48	16777216
	512 32		41.72	8589943592
Naïve_parallel	64 32 32		47.42	16777216
	128 32 32		47.72	134217728
	256 32 32		47.79	1073741842
Strassen_parallel	64 32		49.06	18930904
	1024 4		51.40	9162457088

Table 10: Comparison of matrix_multiplication variations