ECE 699 – Technical Report

A Basic Profiling and Performance Assessment Utility for C++ Codes

Praveen Sankaranarayanan

May 2014

ABSTRACT

Profiling code is an essential part of software development. It helps the programmer understand how the code can be expected to perform. Specifically, profiling helps us identify what regions of the code take the most amount of time and evaluate how different optimization strategies can be directed towards these time critical regions. However, profiling large codebases can become unwieldy due to the large amount of timed data that the programmer has to keep track of and process to better understand the underlying picture. In order to reduce programmer effort and provide a simpler interface that helps profile code easier, I have implemented an Application Programming Interface (API) in C++ that could be imported into any codebase. The API keeps track of the different timers invoked by the user and provides elegant reports, which have more information than evident from the timed data. The API has the ability to keep track of nested timers, which helps one understand how functions or regions of code nested within a larger code section contribute to the overall timing. It also provides methods to determine flop rate and bandwidth. The API provides an ability to compare performance changes between multiple builds of the codebase. This technical report describes the features of this API with numerous examples and illustrates how it could be used to profile the Chrono Engine library.

Table of Contents

1. INTRODUCTION **1**

2. FEATURES OF THE *PROFILER* CLASS **2**

2.1 Creating Timers 2

2.2 Tracking Nested Timers 3

2.3 Computing Flop Rate and Bandwidth 4

2.4 Support for multiple timing utilities 5

2.5 Get a specific function's execution time 5

2.7 Compare multiple builds to evaluate optimizations 7

2.8 Error Messages 7

2.9 Reset timer data 8

3. OUTPUT REPORTS **8**

3.1 Hierarchy report 8

3.2 Concise Report 11

3.3 Printing hierarchy of nested timers to file 12

3.4 Comparison Report 13

4. PROFILING CHRONO **14**

5. FUTURE WORK **15**

6. CONCLUSION **16**

1. INTRODUCTION

While it is absolutely essential to profile any large codebase, very little effort has been taken to make the job of the programmer easier. In a typical scenario, the user has to wrap the regions of code that he wishes to time with multiple function calls depending on the timers that are used. For instance, if the programmer wishes to use Linux's *gettimeofday()* function to time a certain section of code, he has to declare an object of the structure this function takes as an argument. The user then wraps the region of code with calls to *gettimeofday()* with the appropriate arguments passed. The elapsed time is finally determined by doing arithmetic calculations on certain members of the structure. To further complicate this process, it is not very easy to remember the syntax of these timing functions, which would indeed require a quick Google search. While doing all these steps might seem viable for a small number of timers, it becomes infeasible with large codebases which require functions to be timed in multiple files.

The primary motivation behind this project was to reduce the amount of effort that the programmer has to put in to profile code, thereby letting him concentrate better on implementing functionality. The idea was to develop an API that could be used as a utility to profile code and which could be easily imported into any codebase by including the necessary header files. The API is built in C++ and greatly simplifies profiling through numerous functions that could be invoked as needed to time different regions of code. Instead of the steps listed in the *gettimeofday()* example, the user just has to wrap the regions of interest with appropriate function calls from the API, which are as simple as *start(<timer_name>)* and *stop(<timer_name>)*. The API provides numerous parameters like execution time, flop rate, bandwidth and performance improvement.

The remainder of the technical report is organized as follows:

- Section 2 describes the features of the *Profiler* class, particularly highlighting the ability to track nested timers, the support for different timing utilities and the ability to compare performance changes between two independent runs of the program. Detailed documentation with examples is provided, explaining how to invoke each function of the class
- Section 3 describes the three different kinds of reports that can be generated – a tree hierarchy, a concise report and a comparison report
- Section 4 illustrates how this API can be used to profile the Chrono Library
- Section 5 describes how this work can be extended in the future
- Section 6 provides a summary of the project and conclusion

2. Features of the *Profiler* Class

All the features of the profiling API are implemented as functions of a C++ class called *Profiler*. This section describes the features of the *Profiler* class and describes how different functions can be used. While the report provides numerous examples to illustrate the features of the class, it also serves as a documentation that could be referred by anyone who wishes to use this library to profile code.

2.1 Creating Timers

The greatest simplification that the *Profiler* class provides is that it eliminates the need to create a separate object of the timer class for each timer that the user wishes to create. All the functions in the class can be invoked using a single instance of the *Profiler* class. The ability to use the same object to invoke multiple timers reduces the programmer effort to remember the different objects that he created to time each section of code. Starting and stopping timers is as simple as the code snippet shown in Fig 1. *timer_obj* is an object of the *Profiler* class.

```
int main(){
	Profiler timer_obj;

	timer_obj.start("A");
		//do something here...
	timer_obj.stop("A");

	timer_obj.start("B");
		//do some other stuff...
	timer_obj.stop("B");
}
```

Fig. 1: Starting and stopping timers

start
Start a timer with the specified timer name. If the timer does not exist, the timer is created.

```
void start(string timerName)
```

Parameters:

timerName – Name of the timer that the user wishes to create or start. Could be any valid string.

stop
Stop the timer with the specified timer name.

```
void stop(string timerName)
```

Parameters:

timerName – Name of the timer that the user wishes to stop. Could be any valid string.

Errors: Timer does not exist.

2.2 Tracking Nested Timers

The *Profiler* class provides the ability to keep track of timers that are nested. This is a very useful functionality as it provides information to the programmer about the contribution of the execution time of the nested timers to a certain parent timer. The class provides information about the percentage of time of a given parent that was not accounted for by its children. This helps the user identify regions of code within a certain function that could have contributed a significant overhead but had been overlooked while timing. Consider the programming scenario in Fig 2.

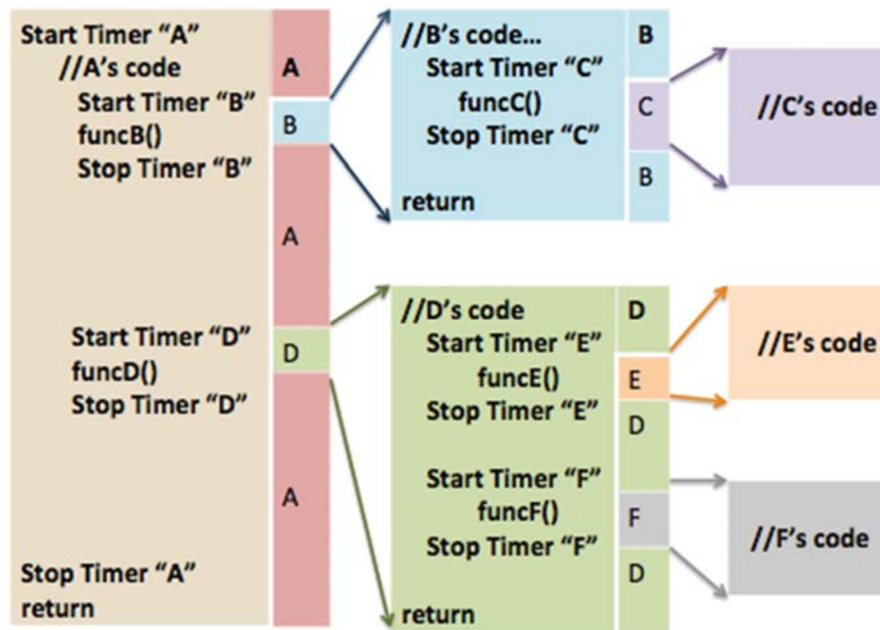

Fig. 2: Typical programming scenario illustrating nested timers

In the example shown above, the main function "A" has the timers "B" and "D" nested within it, which in turn have multiple timers nested within them. Given the fact that such scenarios are pretty common in software development, it would become cumbersome if the programmer has to individually invoke separate timer objects for each function he wishes to time. Further, tracking which timers are nested and how much they contributed to their parent would not be possible. The *Profiler* class automatically keeps track of such data and provides an elegant report that lists the hierarchy of the timers in the form of a tree structure. It also provides the fraction of time that was accounted for by the children to the overall execution time of the parent. The greatest advantage of this functionality is that the programmer does not have to do anything special to track such nested timers. All that the programmer has to do is wrap *start()* and *stop()* calls around the regions he wishes to time, and the class automatically keeps track of different levels of nesting! This is illustrated later with a code snippet in section 3.

2.3 Computing Flop Rate and Bandwidth

The *Profiler* class also provides the ability to determine the flop rate and bandwidth of a certain function or region of code. This is achieved using two functions namely *setFlop()* and *setMemory()*. The user has to provide the number of floating-point operations or the number of memory transactions within the region of interest to these functions to determine the flop rate and bandwidth. Determining the number of floating-point operations is difficult as this is very much dependent on the Instruction Set Architecture (ISA). The user can hence provide an estimated value to determine the flop rate. Determining the number of memory transactions is relatively simpler as this just involves counting the number of loads and stores within the function.

setFlop
Compute the flop rate of the specified timer with the given number of floating-point operations.

```
void setFlop(string timerName,
             double f)
```

Parameters:

timerName – Name of the timer for which the flop rate is to be computed.
f – Number of floating-point operations for the specified timer.

Errors: Timer does not exist

setMemory
Compute the bandwidth of the specified timer with the given number of memory operations.

```
void setMemory(string timerName,
               double m)
```

Parameters:

timerName – Name of the timer for which the bandwidth is to be computed.
m – Number of memory operations for the specified timer.

Errors: Timer does not exist

The code snippet in Fig. 3 shows how these functions can be invoked.

```
timer_obj.setFlop("A",4);
          //Some function with four floating-point operations

timer_obj.setMemory("B",2)
          //An operation that involves a load and a store
```

Fig. 3: Computing flop rate and bandwidth

2.4 Support for multiple timing utilities

In certain situations, the programmer may wish to switch between multiple timing utilities for better accuracy and precision. For instance, some users might prefer using *gettimeofday()* while some others might prefer using the OpenMP timer available as part of the GCC Compiler. The *Profiler* class provides the ability to choose the timer of our choice by just specifying the desired timer as a *constructor argument* while creating an object of the *Profiler* class. The class has support for Linux's *gettimeofday(),* Windows' *QueryPerformanceCounter(),* OpenMP timer and also the conventional system timer *clock().* Fig. 4 shows how one of these four timers can be selected while creating the object.

```
/**
 * Defaults to gettimeofday() on Linux machines
 * Defaults to QueryPerformanceCounter() on Windows
 */
Profiler timer_obj;

//Uses OpenMP timer available in GCC
Profiler timer_obj("OPENMP")

//Uses clock() timer
Profiler timer_obj("CLOCK")
```

Fig. 4: Choosing one of the four available timers

2.5 Get a specific function's execution time

The class provides the ability to retrieve the execution time of any function using the *getTime()* method. The time can be retrieved in either milliseconds or seconds.

getTime

Get the execution time of the specified timer in seconds or milliseconds.

```
double getTime(string timerName,
               string unit)
```

Parameters:

timerName – Name of the timer whose time needs to be retrieved.

unit – Should be either "*sec*" or "*ms*" to retrieve time in seconds and milliseconds respectively.

Errors: If the timer does not exist, the function returns -1.

2.6 Support for Interleaved Timers

The class provides the ability to interleave timers by letting the user specify which timer should be the parent of the newly created timer. This might be useful in scenarios in which the user does not wish to automatically nest timers but instead wants a timer to be placed at a different level of the hierarchy. This interleaving is achieved using an overloaded *start()* function that takes two arguments.

start

Start a timer by placing it under the user specified parent.

```
void start(string timerName,
           string parent)
```

Parameters:

timerName – Name of the timer to create or start.

parent – The name of the parent under which the newly created timer has to be placed. If parent is empty (i.e. *""*), the newly created timer is placed at the same level as the root (i.e. it has no parent).

Error: Parent timer does not exist.

Fig. 5 shows how interleaved timers could be created, with the interleaving between the timers depicted.

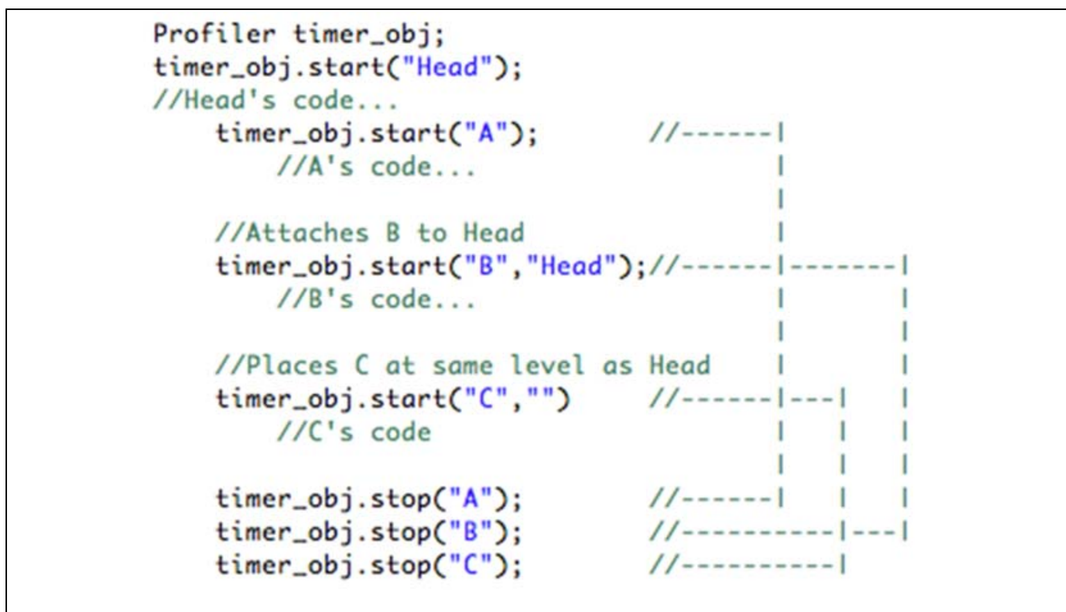

Fig. 5: Interleaved Timers

2.7 Compare multiple builds to evaluate optimizations

At times, the user might do some changes to the code to optimize a certain portion. But this optimization effort might not always have a positive impact on the program. To better evaluate the effects of introducing optimizations, the class provides the ability to compare two different runs of the codebase. It provides information that lets the user know if the optimization had a positive or negative impact on performance. The user can dump the reference data of the run which serves as the base benchmark to a reference file using the *dumpReference()* function described below. The user can then generate a comparison report using the *comparisonReport()* function, which is described in section 3.4.

dumpReference

Dump reference data to a file for comparison with future runs of the program. Dumps the average execution time, average flop rate and average bandwidth for each timer. Average values are used to overcome the impact of varying number of iterations for each timer between two independent runs of the program.

void dumpReference(string file)

Parameters:

file – Name of the file to which the reference data has to be dumped.

The following snippet shows how this function can be invoked to dump reference data to a file.

```
timer_obj.dumpReference("referenceFile.txt");
```

2.8 Error Messages

The class provides certain error messages that help the user avoid common errors. Some of the possible error conditions are:

- Stopping an undefined timer (i.e. a timer that does not exist)
- Requesting timing data for an undefined timer
- Passing an undefined timer as an argument to any of the report generating functions
- Unable to find reference data in the reference file for a certain timer. This could happen if the second run of the program has a timer that was not included in the base run, and hence is not found in the reference file.

There are times at which the error messages can clutter the output reports. To avoid this, the class provides the ability to turn off error messages using the debug flag.

```
timer_obj.debug = false;
```

Error messages are ON by default.

2.9 Reset timer data

It might be useful in certain scenarios to reset the data of all timers. For instance, the program could involve multiple time steps, each of which could invoke all the timers. In that case, it would make sense to reset all the timers between two successive steps. The *reset()* function resets all the parameters i.e. execution time, number of floating-point operations, number of memory operations, flop rate and bandwidth. The function does not take any argument and can be invoked as follows.

```
timer_obj.reset();
```

3. Output Reports

The greatest advantage of the *Profiler* API is that it provides elegant reports that consolidate the data from all the timers. The class has the ability to print three different kinds of reports. Each of these reports provides the execution time, flop rate, bandwidth and percentage of time accounted by nested timers. The three reports are:

- A detailed report that depicts the hierarchy of the nested timers using a tree structure
- A concise report that prints all relevant data as a table to either the standard output or a file
- A comparison report that prints a performance comparison between multiple runs of the program.

The class also has the ability to print the nested timers' hierarchy to a file. This section describes how each of these reports could be generated using numerous examples.

3.1 Hierarchy report

The hierarchy of the nested timers can be printed using the *printReport()* function.

printReport

Prints the hierarchy of the nested timers under the specified timer to standard output.

```
void printReport(string timerName)
```

Parameters:

timerName – Name of the timer whose nested timer hierarchy is to be printed.

There is also an overloaded function with the same name that takes no argument. This function prints the complete hierarchy of all the timers in the program.

Errors: Timer does not exist.

Fig. 6 shows how this function can be invoked.

```
        //Prints the nested timer hierarchy of timer A
        timer_obj.printReport("A");

        //Prints the complete hierarchy of all timers
        timer_obj.printReport();
```

Fig. 6: Invoking the *printReport()* function

Consider the following example.

```
Profiler obj("OPENMP");
    int i;
    for(i=0;i<2;i++){
    obj.start("Head");
    /**master function "Head" that has several
     * functions within it
     */
    sleep(1);
        obj.start("A");
        //function A that has functions B and D as children.
        sleep(1);
            obj.start("B");
            //function B...has C
            sleep(2);
                obj.start("C");
                //function C...
                sleep(1);
                obj.stop("C");
            obj.stop("B");

            obj.start("D");
            //function D...belongs to A
            sleep(3);
            obj.stop("D");

        obj.stop("A");
    obj.stop("Head");
    }

    obj.printReport();
    obj.printReport("A");
    obj.printReport("B");
```

Fig. 7: Example to demonstrate nested timers

Outputs

```
----------------***************
----------------Timer Name: C
----------------Time taken: 2.00013s = 2000.13ms
------------***************
------------Timer Name: B
------------Time taken: 6.00028s = 6000.28ms
------------Nested timers' time: 2.00013
------------Nested timers' percentage: 33.334 %
------------***************
------------Timer Name: D
------------Time taken: 6.00011s = 6000.11ms
--------***************
--------Timer Name: A
--------Time taken: 14.0005s = 14000.5ms
--------Nested timers' time: 12.0004
--------Nested timers' percentage: 85.7139 %
----***************
----Timer Name: Head
----Time taken: 16.0006s = 16000.6ms
----Nested timers' time: 14.0005
----Nested timers' percentage: 87.4998 %
***************
Total time taken: 16.0006
```

Fig. 8: Output for obj.printReport()

```
------------***************
------------Timer Name: C
------------Time taken: 2.00013s = 2000.13ms
--------***************
--------Timer Name: B
--------Time taken: 6.00028s = 6000.28ms
--------Nested timers' time: 2.00013
--------Nested timers' percentage: 33.334 %
--------***************
--------Timer Name: D
--------Time taken: 6.00011s = 6000.11ms
----***************
----Timer Name: A
----Time taken: 14.0005s = 14000.5ms
----Nested timers' time: 12.0004
----Nested timers' percentage: 85.7139 %
```

Fig. 9: Output for obj.printReport("A")

```
--------***************
--------Timer Name: C
--------Time taken: 2.00013s = 2000.13ms
----***************
----Timer Name: B
----Time taken: 6.00028s = 6000.28ms
----Nested timers' time: 2.00013
----Nested timers' percentage: 33.334 %
```

Fig. 10: Output for obj.printReport("B")

3.2 Concise Report

A concise report in the form of a table can be printed to standard output or a file using the *printConcise()* function.

printConcise

Prints a concise report to the standard output or the specified file. The report includes the specified timer and its nested timers.

```
void printConcise(string timerName,
                  string file)
```

Parameters:

timerName – Name of the timer whose data is to be printed. An empty *timerName (i.e. “”)* prints the complete report for all timers.

file – Name of the file to which the report has to be generated. Specifying file as *“STDOUT”* prints the concise report to the standard output.

Error: Timer does not exist.

Fig. 11 shows how this function can be invoked.

```
//prints the complete data of all timers to file 'report.txt'
     obj.printConcise("","report.txt");

//prints the data for B and its nested timers to stdout
     obj.printConsise("B","STDOUT");

//prints the data for A and its nested timers to 'report.txt'
     obj.printConsise("A","report.txt");
```

Fig. 11: Invoking the *printConcise()* function

Using the same example shown in Fig. 7, the output for the three function calls in Fig. 11 would be as shown below.

```
Timer Name        Time(s)      Time(ms)   t_Nested(s)     Nested %   Flop_Rate   Bandwidth
C                2.000110   2000.109911           N/A          N/A         N/A         N/A
B                6.000263   6000.262737      2.000110  33.333706 %         N/A         N/A
D                6.000118   6000.118017           N/A          N/A         N/A         N/A
A               14.000521  14000.520945     12.000381  85.713816 %         N/A         N/A
Head            16.000658  16000.658035     14.000521  87.499657 %         N/A         N/A
16.000658

Timer Name        Time(s)      Time(ms)   t_Nested(s)     Nested %   Flop_Rate   Bandwidth
C                2.000110   2000.109911           N/A          N/A         N/A         N/A
B                6.000263   6000.262737      2.000110  33.333706 %         N/A         N/A
D                6.000118   6000.118017           N/A          N/A         N/A         N/A
A               14.000521  14000.520945     12.000381  85.713816 %         N/A         N/A
```

Fig. 12: Contents of file ‘report.txt’

```
[praveen@mumble-17] (25)$ ./test
Timer Name                  Time(s)        Time(ms)     t_Nested(s)      Nested %    Flop_Rate    Bandwidth
C                          2.000110     2000.109911             N/A           N/A          N/A          N/A
B                          6.000263     6000.262737        2.000110   33.333706 %          N/A          N/A
```

Fig. 13: Contents printed in standard output for obj.printConcise("B","STDOUT")

3.3 Printing hierarchy of nested timers to file

If the user wishes to see the nested hierarchy of timers along with the concise report in a file, the *printNestedTimers()* function can be used.

printNestedTimers

Prints the hierarchy of the nested timers under the specified timer to either the standard output or specified file.

```
void printNestedTimers(string timerName,
                       string file)
```

Parameters:

timerName – Name of the timer whose nested timers have to be printed to file. An empty *timerName (i.e. "")* prints the complete hierarchy of all timers.

file – Name of the file to which the hierarchy is to be printed. Specifying file as *"STDOUT"* prints the hierarchy to the standard output.

Errors: Timer does not exist.

```
//prints the complete hierarchy to file 'nest.txt'
     obj.printNestedTimers("","nest.txt");

//prints the complete report of all timers to the file 'nest.txt'
     obj.printConcise("","nest.txt");
```

Fig. 14: Printing the nested timers to a file before printing a concise report

Considering the same example shown in Fig. 7, the output for the above snippet would look as shown below.

```
----Head
--------A
------------B
----------------C
------------D

Timer Name                  Time(s)        Time(ms)     t_Nested(s)      Nested %    Flop_Rate    Bandwidth
C                          2.000112     2000.112057             N/A           N/A          N/A          N/A
B                          6.000238     6000.237942        2.000112   33.333879 %          N/A          N/A
D                          6.000115     6000.115156             N/A           N/A          N/A          N/A
A                         14.000481    14000.480890       12.000353   85.713864 %          N/A          N/A
Head                      16.000605    16000.604868       14.000481   87.499698 %          N/A          N/A
16.000605
~
```

Fig. 15: Contents of file 'nest.txt'

3.4 Comparison Report

As stated before, the class has the ability to provide a report that compares two different runs of the program. This report can be generated using the *comparisonReport()* function. The reference file for comparison should be dumped using the *dumpReference()* function as explained in section 2.7.

comparisonReport

Prints a comparison report for the specified timer and its nested timers to either the standard output or given file.

```
void comparisonReport( string timerName,
                       string newFile,
                       string refFile)
```

Parameters:

timerName – Name of the timer whose data has to be compared and printed. An empty *timerName (i.e. "")* prints the complete report showing comparison for all timers.

newFile – The file to which the comparison report is to be printed. Specifying file as *"STDOUT"* prints the report to the standard output.

refFile – The reference file which has the data for comparison. This file is generated using the *dumpReference()* function.

Errors:

- Timer does not exist
- Reference file does not exist
- Unable to find reference data for a certain timer in reference file.

For the example shown in Fig. 7, the duration of each timer's *sleep* was changed as shown in the table below.

Timer Name	Initial Run	Second Run
A	7s	6s
B	3s	4s
C	1s	3s
D	3s	1s
Head	8s	8s

Table 1: Changes introduced between two runs for example of Fig. 7

The initial run should have the following line:

```
//dump reference data to file 'ref.txt'
obj.dumpReference("ref.txt");
```

The second run of the program should have the following line to generate the comparison report. The reference file for comparison is ‘ref.txt’

```
//generate comparison report using 'ref.txt' as reference
obj.comparisonReport("","comparison.txt","ref.txt");
```

The output report appears as shown in Fig. 16.

Timer Name	Time(s)	Time(ms)	t_Nested(s)	Nested %	Flop_Rate	Bandwidth	t(change)	FR(change)	BW(change)
C	6.000121	6000.120878	N/A	N/A	N/A	N/A	-199.990744	N/A	N/A
B	8.000233	8000.232935	6.000121	74.999327 %	N/A	N/A	-33.331660	N/A	N/A
D	2.000125	2000.125170	N/A	N/A	N/A	N/A	66.665292	N/A	N/A
A	12.000491	12000.490904	10.000358	83.332909 %	N/A	N/A	14.285392	N/A	N/A
Head	16.000585	16000.585079	12.000491	75.000326 %	N/A	N/A	0.000231	N/A	N/A
16.000585									

Fig. 16: Contents of file ‘comparison.txt’

As shown in the report, the class provides the percentage change in the execution time, flop rate and bandwidth between two different runs of the program. If the flop rate or bandwidth is not computed, the field has the value N/A (Not Available).

4. Profiling Chrono

The API was used to profile the Chrono::Parallel library developed at SBEL. The following hierarchical report was generated by timing several functions in a step.

```
[psankaranara@euler chrono_models]$ ./shaker
Set BPA: 25 12 25
50 24 50 60000
--------****************
--------Timer Name: update
--------Time taken: 0.0708418s = 70.8418ms
--------****************
--------Timer Name: collision
--------Time taken: 2.85801s = 2858.01ms
----------------****************
----------------Timer Name: shurA_normal
----------------Time taken: 0.235997s = 235.997ms
----------------Bandwidth: 7.61171
----------------****************
----------------Timer Name: shurA_reduce
----------------Time taken: 0.628352s = 628.352ms
----------------****************
----------------Timer Name: shurA_sliding
----------------Time taken: 2.32018s = 2320.18ms
----------------Bandwidth: 0.613162
------------****************
------------Timer Name: shurA
------------Time taken: 3.51276s = 3512.76ms
------------Nested timers' time: 3.18453
------------Nested timers' percentage: 90.6561 %
------------****************
------------Timer Name: shurB
------------Time taken: 2.72752s = 2727.52ms
--------****************
--------Timer Name: lcp
--------Time taken: 17.2382s = 17238.2ms
--------Nested timers' time: 6.24028
--------Nested timers' percentage: 36.2003 %
----****************
----Timer Name: step
----Time taken: 20.1671s = 20167.1ms
----Nested timers' time: 20.167
----Nested timers' percentage: 99.9994 %
****************
Total time taken: 20.1671
```

Fig. 17: Nested timing report generated by profiling Chrono Library

An interesting observation from Fig. 17 is that the *shurA* and *shurB* functions accounted only for 36.20 % of the *lcp solver* function. This provides the programmer with the useful information that there might be some other time critical function or region of code within *lcp solver* that was overlooked while profiling. This highlights the advantage of tracking nested timers. The concise report generated for two steps is shown in Fig. 18. The complete hierarchy of all the nested timers is also printed to the file.

```
----step
--------update
--------collision
--------lcp
------------shurA
----------------shurA_normal
----------------shurA_reduce
----------------shurA_sliding
------------shurB
```

Timer Name	Time(s)	Time(ms)	t_Nested(s)	Nested %	Flop_Rate	Bandwidth
update	0.070842	70.841789	N/A	N/A	N/A	N/A
collision	2.858011	2858.011007	N/A	N/A	N/A	N/A
shurA_normal	0.235997	235.996962	N/A	N/A	N/A	7.611714
shurA_reduce	0.628352	628.352165	N/A	N/A	N/A	N/A
shurA_sliding	2.320179	2320.179462	N/A	N/A	N/A	0.613162
shurA	3.512757	3512.757301	3.184529	90.656095 %	N/A	N/A
shurB	2.727521	2727.521181	N/A	N/A	N/A	N/A
lcp	17.238175	17238.175154	6.240278	36.200343 %	N/A	N/A
step	20.167149	20167.149067	20.167028	99.999399 %	N/A	N/A
20.167149						

Timer Name	Time(s)	Time(ms)	t_Nested(s)	Nested %	Flop_Rate	Bandwidth
update	0.076726	76.725721	N/A	N/A	N/A	N/A
collision	2.607432	2607.432127	N/A	N/A	N/A	N/A
shurA_normal	0.231743	231.743097	N/A	N/A	N/A	7.760812
shurA_reduce	0.618612	618.612051	N/A	N/A	N/A	N/A
shurA_sliding	2.377523	2377.523184	N/A	N/A	N/A	0.598116
shurA	3.561449	3561.449051	3.227878	90.633848 %	N/A	N/A
shurB	2.704726	2704.725981	N/A	N/A	N/A	N/A
lcp	16.769811	16769.810915	6.266175	37.365806 %	N/A	N/A
step	19.454019	19454.019070	19.453969	99.999741 %	N/A	N/A
19.454019						

Fig. 18: Concise Report generated by profiling Chrono Library

5. Future Work

This work has a lot of potential to be improved in the future. The API currently has the ability to compare two independent runs of the program. This could be extended to compare multiple runs by dumping the data of each run to the reference file and retrieving the appropriate run that the user wishes to compare. The class can also be extended to include the ability to time GPU kernels. This would require adding support for CUDA events and CUDA timers. As CUDA kernel calls are asynchronous, timing them using just the *start()* and *stop()* functions would be a challenging but extremely useful functionality to implement.

6. Conclusion

This technical report described an API that was developed to help programmers profile code more easily. The features of the *Profiler* class were explained. It was observed that the API greatly simplifies the effort needed by the programmer to track the different timing data and provides useful methods to print elegant reports. The ability to track nested timers provides an added advantage to better analyze code. The functionality to compare performance between independent runs of the program helps the programmer evaluate how different optimization strategies perform. Overall, this API turns out to be a valuable addition to anyone who wishes to profile code with minimal effort.