INFORMATICS AND COMPUTER SCIENCE,
COMPUTER SYSTEMS AND TECHNOLOGIES

HDU_ITMO Joint Institute

Parallel Computing

Laboratory research #2.

"Study of the parallel libraries for C-programs effectiveness"

陈浩东 Chen Haodong

金一非 Jin Yifei

王灵斌 Wang Lingbin

Table of Contents

1 Description of the processor, operating system, and compiler

According to the given task description, the main task and optional task are performed on the virtual machine whose candidates are shown in the table 1 below.

Table 1: candidates of the virtual machine

Official name	Version number	Bit depth	Number of cores	RAM capacity
Ubuntu	22.04	64	4	2 GB
GCC	11.4.0	\	\	\
Processor	Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz	32-bit, 64-bit	\	\

2 Description of the parallel library configuration features

2.1 Description of installation steps:

First, download the Framewave project for the corresponding Linux operating system from the official website "https://sourceforge.net/projects/framewave/". Then, read the README file and follow the tutorial to complete the installation steps. It's important to note that it is needed to add the following command to the $.bash\_rc$ file.

```
export LD_LIBRARY_PATH={ABSOLUTEPATH}/FW/lib:$LD_LIBRARY_PATH
```

2.2 Description of library:

According to the official manual, Framewave consists of the following libraries:

- The Base Library functions are essential for primary tasks such as memory allocation and functions that manage the performance of other library functions.
- The Image Processing Library functions perform a variety of tasks related to image processing.
- The JPEG Library functions perform a variety of tasks related to Joint Photographic Experts Group image manipulation
- The Signal Processing Library functions perform a variety of tasks related to signal processing.
- The Video Library functions perform video manipulation, encoding and decoding.

In this task, functions in Base Library and Signal Processing Library are implemented to yield maximum performance since the computations in tasks are mainly related to vector operations.

2.3 Description of configuration options:

Commands below are used to compile the code of lab2.

```
gcc -O3 -m64 -c -I/home/parallels/Desktop/lab2/FW lab2.c
```

2. gcc -O3 -m64 -L/home/parallels/Desktop/lab2/FW/lib lab2.o -lfwSignal -lfwBase

where:

- $-O3$ is an optimization level option. It instructs the compiler to perform more optimizations to improve the execution speed of the program. $O3$ represents the highest level of optimization.
- $-m64$ indicates generating code for a 64-bit architecture. This means that the produced object code is intended for 64-bit systems.
- $-c$ indicates compiling the source file only, without linking. This will produce an object file named lab2.o.
- $-I$ specifies the search path for header files. When the source file lab2.c includes other headers, the compiler will look for them in this path.
- $lab2.c$ is the source file to be compiled.
- $-L$ specifies the search path for the linker to find library files. The compiler/linker will look for library files in this path or other standard library paths.
- $lab2.o$ is the file to be linked.
- $-lfwSignal$ and $-lfwBase$ tell the linker to link with two libraries: libfwSignal and libfwBase.

3 Description of scripts

To automatically test experiments, a few scripts are written. Makefile is used to compile lab2.c code. lab2.sh is used to test threads vary from 1 to 7. Lab2-optional.sh should work with lab2-optional.c after cpu-supervisor.sh, which is the script to supervise CPU utilization. Besides, all scripts and codes are submitted in the zip file.

4 Experiments results

In this section, experiments results are shown in tables and graphs. $N1$ is set to be 6000, $N2$ is set to be 1000000, the value of Δ: $\Delta = \frac{N2 - N1}{10} = 99400$ for experiments. Create bash scripts to run the programs on different thread counts for values $N = N1, N1 + \Delta, N1 + 2\Delta, N1 + 3\Delta, \dots, N2$ to get results and compare them with the results in lab 1. In addition, investigation of parallel acceleration for different threads larger than core numbers are performed to estimate virtualization overhead when creating a large number of threads for optional task. For optional task, we expend the value of threads to 7 to estimate virtualization overhead when creating a large number of threads. Results of the main task and optional task will be presented in the following section 3.1 and 3.2, respectively. However, at first, as demonstrated in Table 2, it is evident that the outcomes for X remain consistent between Laboratory 1 and Laboratory 2.

Table 2: Proof of X's consistency

$Lab1\_X$	2.220679	-0.543258	-0.287317	2.287252	…
$Lab2\_X$	2.220679	-0.543258	-0.287317	2.287252	…

4.1 Parallel acceleration of main task:

Since the core number on the experimental stand is 4, the experiment runs the program on 1, 2, 3, and 4 threads respectively. Moreover, 5, 6, 7 threads are tested for optional task. Function $gcc\_fw\_M(N)$ represents the consuming time in unit ms of each experiment, where M is the number of threads and N is the size of array. Function gcc_lab1 represents the result in lab1, and it is used to be compared with other results obtained from lab 2 since we conclude in lab 1 that the automatic parallelization of compiler does not make any parallelization for code of lab1. The comparison results are shown in table 3 and figure 2.

Table 3: consuming time

N	6000	105400	204800	304200	403600	503000	602400	701800	801200	900600	1000000
$gcc\_lab1(N)$	11	210	419	643	833	1059	1320	1500	1753	2125	2485
$gcc\_fw\_1(N)$	22	289	536	805	1079	1330	1937	2320	2668	2508	2760
$gcc\_fw\_2(N)$	17	228	436	652	876	1060	1320	1547	1750	1969	2221
$gcc\_fw\_3(N)$	19	223	428	628	823	1009	1234	1404	1641	1859	2136
$gcc\_fw\_4(N)$	16	218	395	589	796	992	1184	1375	1549	1775	1979
$gcc\_fw\_5(N)$	17	217	391	580	781	976	1186	1322	1561	1780	1987
$gcc\_fw\_6(N)$	16	219	393	589	773	969	1167	1360	1585	1739	1996
$gcc\_fw\_7(N)$	17	218	394	592	784	970	1169	1365	1568	1783	1967

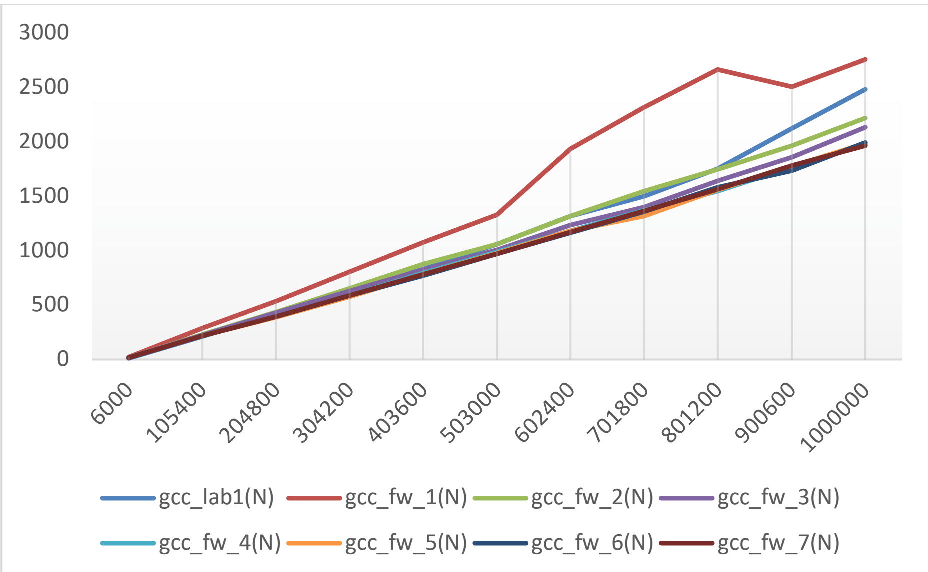

Figure 1: consuming time

Now the calculation of parallel acceleration and parallel efficiency can be performed. The acceleration value can be calculated using following formula (1), since the task of this example is fixed:

$$S(p)|_{w=const} = \frac{t(1)}{t(p)} \tag{1}$$

where:

- $S(p)$ is the acceleration speed.
- $t(1)$ represents the time it takes for the program to run on a single processor.
- $t(p)$ is the time it takes for the program to run on p threads.

The parallel acceleration is shown in table 4 and Figure 2.

Table 4: parallel acceleration

N	6000	105400	204800	304200	403600	503000	602400	701800	801200	900600	1000000
$S(1)$	1	1	1	1	1	1	1	1	1	1	1
$S(2)$	1.29	1.27	1.23	1.23	1.23	1.25	1.25	1.32	1.36	1.13	1.25
$S(3)$	1.15	1.29	1.25	1.28	1.31	1.32	1.38	1.41	1.43	1.17	1.29
$S(4)$	1.38	1.33	1.36	1.37	1.36	1.34	1.41	1.50	1.50	1.27	1.39
$S(5)$	1.29	1.33	1.37	1.39	1.38	1.36	1.47	1.49	1.49	1.26	1.39
$S(6)$	1.38	1.32	1.36	1.37	1.39	1.37	1.42	1.46	1.53	1.26	1.39
$S(7)$	1.29	1.33	1.36	1.36	1.38	1.37	1.42	1.48	1.49	1.28	1.40

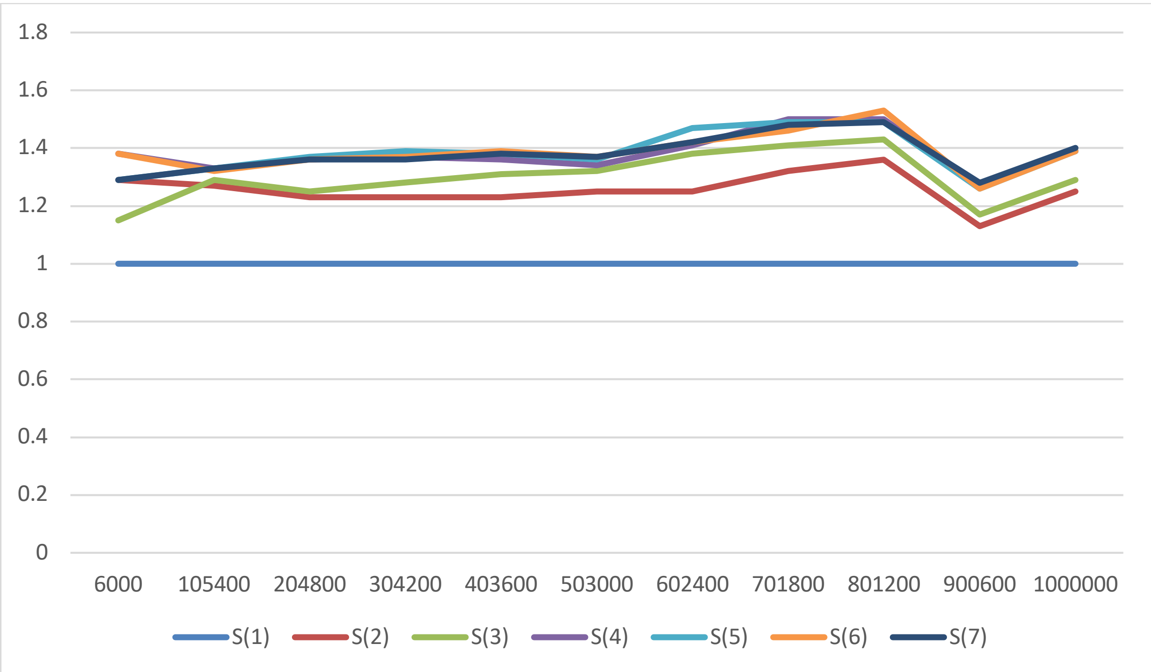

Figure 2: parallel acceleration

4.2 Parallel efficiency of optional task:

Now the parallel efficiency can be derived from Table 4 using formula (2).

$$E(p) = \frac{S(p)}{p} \tag{2}$$

Where $S(p)$ is derived directly from parallel acceleration and p is the number of processors. The results of parallel efficiency are shown in table 5 and figure 3.

Table 5: parallel efficiency

N	6000	105400	204800	304200	403600	503000	602400	701800	801200	900600	1000000
$E(1)$	1	1	1	1	1	1	1	1	1	1	1
$E(2)$	0.65	0.64	0.62	0.62	0.62	0.63	0.63	0.66	0.68	0.57	0.63
$E(3)$	0.38	0.43	0.42	0.43	0.44	0.44	0.46	0.47	0.48	0.39	0.43
$E(4)$	0.35	0.33	0.34	0.34	0.34	0.34	0.35	0.38	0.38	0.32	0.35
$E(5)$	0.26	0.27	0.27	0.28	0.28	0.27	0.29	0.30	0.30	0.25	0.28
$E(6)$	0.23	0.22	0.23	0.23	0.23	0.23	0.24	0.24	0.26	0.21	0.23
$E(7)$	0.18	0.19	0.19	0.19	0.20	0.20	0.20	0.21	0.21	0.18	0.20

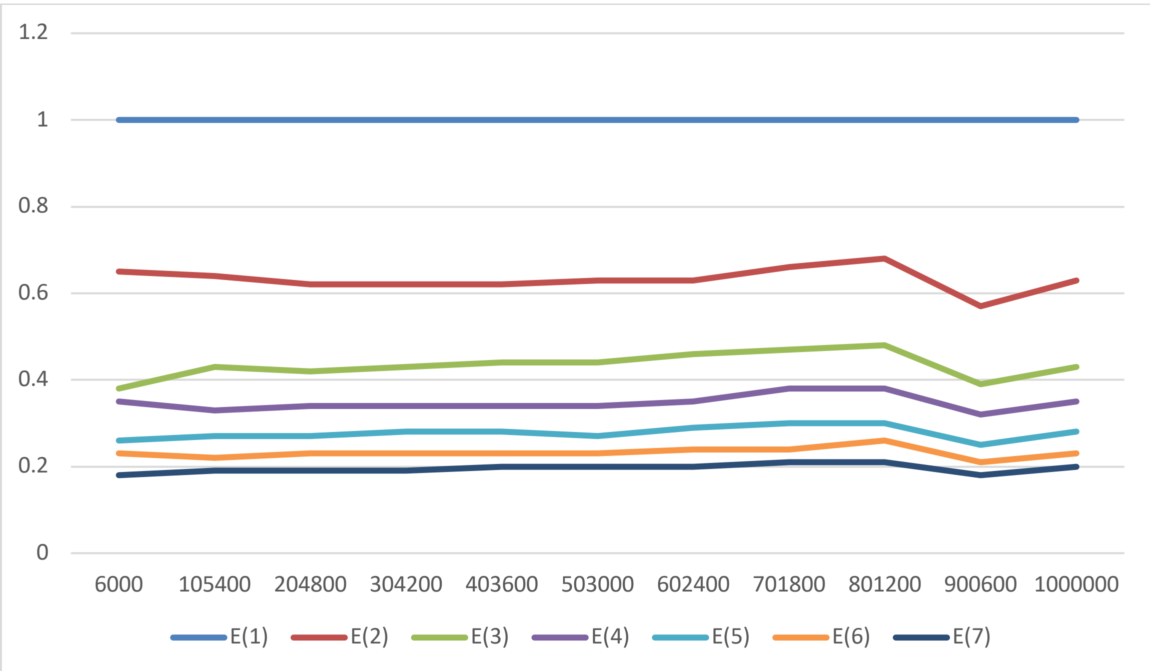

Figure 3: parallel efficiency

4.3 Evidence of parallelizaion:

To illustrate that the program is really parallelized, a test is performed and results are shown in figure 4. Left part of the figure is a CPU supervisor which shows CPU utilization, and the right part of the figure outputs start time and end time of each experiment. In this way, evidence of parallelization is shown, e.g. lab2-fw-4 test begins at time 07:15:54 and ends at 07:15:55, and all CPUs utilizations increase at this period of time.

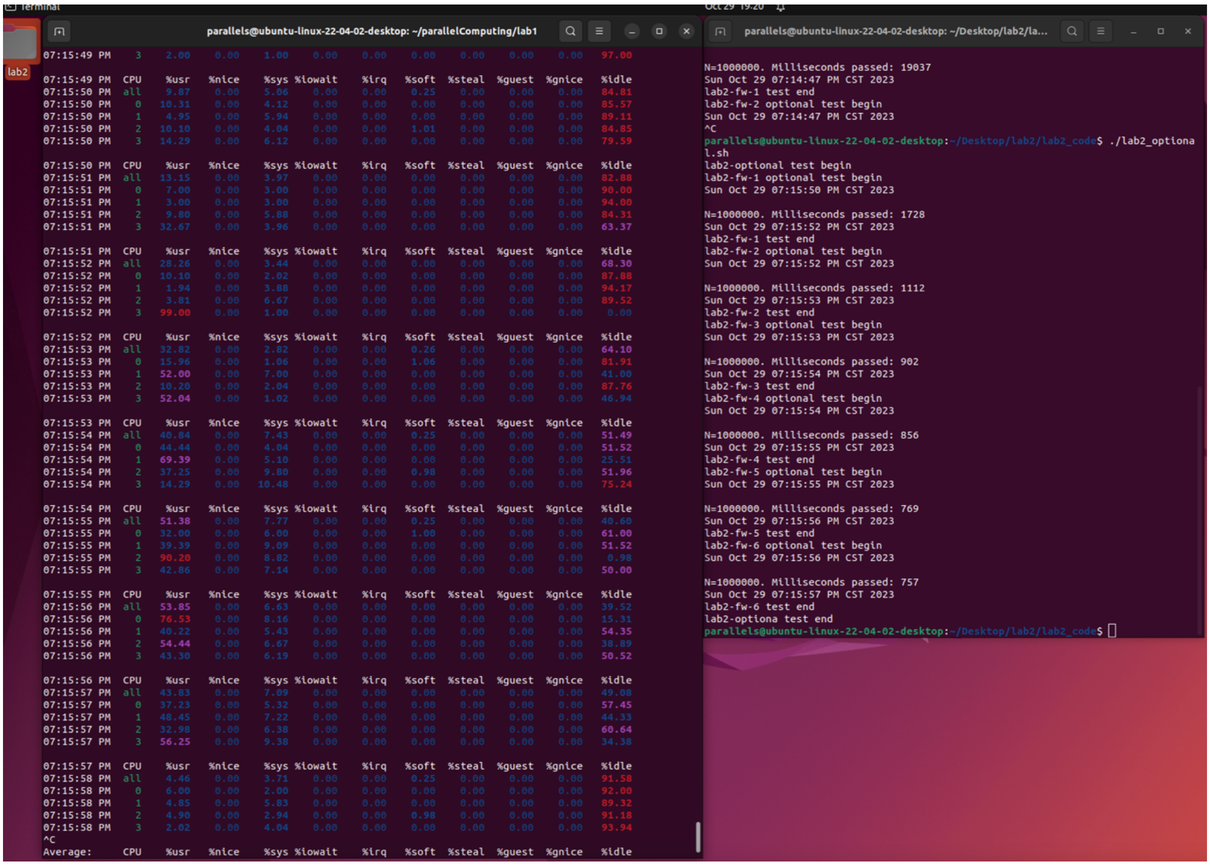

Figure 4: evidence of parallelization

5 Conclusion

5.1 Comments on consuming time:

From the figure 1, we can conclude that, when using function in Framewave, the program runs for a longer time. It is supposed that the time taken to call the Framewave library function is longer than the time taken to call the C standard library function. However, when the program runs on multi-threads, time taken decreases to around 2000 ms due to parallel computing.

5.2 Comments on parallel acceleration:

From table 4, it is conclude that values of parallel acceleration increases when the threads vary from 1 to 4, but maintain around the same when the threads are larger than 4. Exploring the program, it is found that when the thread number is set to be larger than existing cores, there will be no more threads be created than the number of cores, for example, when the number of CPUs is 4, a maximum of four threads are created.

5.3 Comments on parallel efficiency:

Table 5 showcases the parallel efficiency of a primary task. From the table, the following observations can be concluded, when the number of threads is 1, i.e., E(1), the parallel efficiency is consistently 1. This is expected since there's only one thread executing the task, with no parallel overhead. As the number of threads increases, from E(2) to E(7), we can observe a gradual decline in parallel efficiency. This could be due to synchronization overhead among threads, data dependencies, or other such reasons. Lastly, although increasing the number of threads results in a drop in efficiency, for some task sizes, adding more threads still might improve the overall performance of the task. This requires a careful consideration based on the specific application scenario and hardware configurations.

5.4 Comments on code:

To make a good understanding of the experiment results, we should explore the functions used in lab2.

1. \`$outputDoubleArray$\` and \`$outputArray$\` functions: These two functions simply iterate through the arrays and output elements. There is no need for parallelization as each element's processing is independent and does not involve complex calculations or data dependencies.

2. \`$outputSum$\` function: This function calculates the sum of array elements and is not directly parallelized. It uses the \`$fwsSum\_64f$\` function to calculate the sum, which does not have explicit MT (MultiThreaded) support.

3. \`$hyperbolicCosinePlusOne$\` function: This function computes the hyperbolic cosine function for the array and adds one to the result. It involves element-wise operations and can be parallelized. The functions \`$fwsCosh\_64f\_A53$\` and \`$fwsAddC\_64f\_I$\` internally implement parallelization, as they have MT support.

4. \`$squareRootAfterMultByE$\` function: This function includes multiple operations, such as computing the sum of partial elements, copying arrays, multiplying by the base of the natural logarithm E, and calculating square roots. Some of these operations can be parallelized, such as computing the sum of partial elements and copying arrays.

5. \`$selectLarger$\` function: This function compares the corresponding elements of two arrays and stores the larger value in the second array. It involves element-wise comparison and can benefit from parallel loops or vectorization instructions to accelerate the comparison process.

6. \`$swap$\` function: This function performs a simple value swap operation and does not require parallelization, as it only involves the exchange of two variables, making it an atomic operation.

7. \`$fwsMulC\_64f\_I$\` function: This function has inherent parallelism, as it is supported by multiple technologies, including MT (Multi Threaded), enabling parallelization.

8. \`$fwsSqrt\_64f\_A53$\` function: Similar to point 7, this function also benefits from parallelization due to its support for multiple technologies, including MT.

The key to significantly improving code compilation speed is to leverage functions with MT (MultiThreaded) support. Therefore, when writing code, we can refer to the "Frameewave" documentation to check if specific functions have the MT attribute. By strategically incorporating these functions, we can expedite the compilation speed of our code. However, every program has a non-parallelizable section that can't be optimized, which is the reason for prolonged execution times. One solution is to design specialized functions for these non-parallel sections, which might involve more advanced compilation methods.