INFORMATICS AND COMPUTER SCIENCE,
COMPUTER SYSTEMS AND TECHNOLOGIES

HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

HDU_ITMO Joint Institute

Parallel Computing

Laboratory research 1. "Automatic program parallelization"

陈浩东 Chen Haodong

金一非 Jin Yifei

王灵斌 Wang Lingbin

Table of Contents

1 Description of the task

Laboratory research 1 “Automatic program parallelization” mainly focus on understanding the performance achieved by the automatic program parallelization of different complier. The task should be performed on a Linux operating system with GCC compiler version 4.7.2 or higher on a computer with a multi-core processor. The console program in C completing the task in item 4 is a sort function dealing with an array, and in this report, a comb sort is achieved. After writing the c program, the command:

```
gcc -O3 -Wall -Werror -o lab1-seq lab1.c
```

is executed to compile the program without using automatic parallelization, thus output a executable program $lab1$.

Besides, the command:

```
gcc -O3 -Wall -Werror -floop-parallelize-all -ftree-parallelize-loops=K lab1.c -o lab1-par-K
```

should also be executed by assigning at least 4 different integer values to the variable K in turn, and explain the choice. In this report, 2, 4, 8, and 16 are chosen to assign to the variable. The reason for choosing these values is that the program is running on an ubuntu with 2 cores, therefore, 2 is the most intuitive choice while other variables are multiples of 2. Completing all works above, a result is one non-parallelized program and four parallelized programs. Run the lab1-seq file from the command line, increasing the value of N to the value of $N1$, at which the execution time exceeds 0.01 s. Similarly, find the value of $N = N2$, at which the execution time exceeds 2 s after closing all application programs running on the operating system to avoid affecting the results of the experiments. Now, the experiments can be performed using the found values $N1$ and $N2$. When the experiments is completed, write a report on the work performed, be ready to answer questions on the presentation, and find the algorithm’s computational complexity before and after parallelization, compare the results obtained. Two optional tasks are set to get good and excellent mark.

2 Description of the processor, operating system, and compiler

2.1 Main task:

The main task is 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	2	4 GB
GCC	11.4.0	\	\	\
Processor	Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz	32-bit, 64-bit	\	\

2.2 Optional task:

The optional task is performed on MacOS since some problems are met on installing ICC on virtual machine. Candidates of MacOS are shown in table 2.

In addition, the reason why Solaris Studio compiler mentioned in Optional task #1 is not used for experiment is that there is no suitable operating system for this compiler. After reading the documentation, it is known that Solaris Studio can only be installed on the following operating systems: Solaris 10 1/06 and subsequent Solaris 10 OS updates, SuSE Linux Enterprise Server 11, RedHat Enterprise Linux 5, and Oracle Enterprise Linux 5. But there are only MacOS and ubuntu.

Table 2: candidates of MacOS

Official name	Version number	Bit depth	Number of cores	RAM capacity
MacOS	Ventura 13.4	64-bit	8	32 GB
ICC	2021.10.0	\	\	\
Processor	Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz	32-bit, 64-bit	\	\

3 Program consuming time

In this section, experiments results are put in the tables. $N1$ 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 compiled in parallel and non-parallel ways for values $N = N1, N1+\Delta, N1+2\Delta, N1+3\Delta, \ldots, N2$ to get results.

3.1 Table and graph of gcc compiler:

The following command:

```
gcc -O3 -Wall -Werror -o lab1-seq lab1.c -lm
```

produce a program called $lab1-seq$ in a non-parallel way.

Result of the function $gcc\_seq(N)$ is the time consuming when executing comb sort program in a non-parallel way, the unit of which is ms.

Variable k is the number of threads created according to the command:

```
gcc -O3 -Wall -Werror -floop-parallelize-all -ftree-parallelize-loops=K lab1.c -o lab1-par-K -lm
```

and in this experiment, the parameter K is set to 2, 4, 8, and 16 sequentially. The unit of output of $gcc\_par\_k(N)$ is ms, which represents the time consuming of the program. The results are shown in table 3 and figure 1.

Table 3: table of program consuming time of gcc compiler

N	6000	105400	204800	304200	403600	503000	602400	701800	801200	900600	1000000
$gcc\_seq(N)$	11	210	419	643	833	1059	1320	1500	1753	2125	2128
$gcc\_par\_2(N)$	9	221	427	637	852	1060	1300	1491	1992	2280	2510
$gcc\_par\_4(N)$	16	264	517	775	1018	1260	1532	1767	2023	2283	2546
$gcc\_par\_8(N)$	21	271	516	762	1042	1257	1520	1755	2027	2291	2596
$gcc\_par\_16(N)$	33	272	522	788	1024	1252	1507	1760	2043	2287	2554

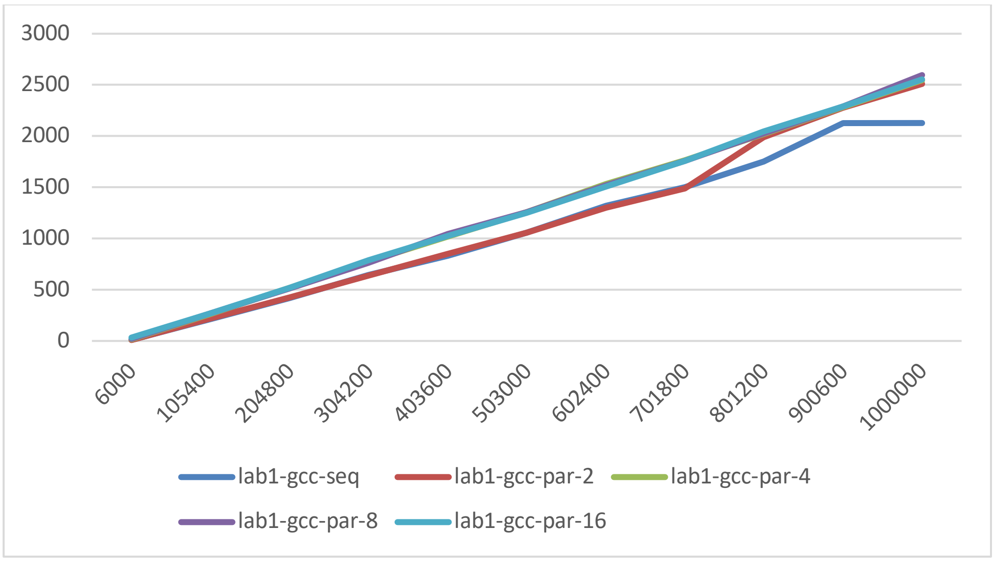

Figure 1: graph of program consuming time of gcc compiler

3.2 Table and graph of icc compiler:

In this section, a different compiler called "intel C++ compiler" is used to compile comb sort c program in a non-parallel way.

In addition, execute the following command:

```
icc -parallel -qopt-report-phase=par -par-num-threads=k  -o lab1-icc-par-k lab1.c
```

to compile $lab1.c$ in a parallel way with k threads, where k should be replaced by 2, 4, 8, 16, and the results are shown in table 4 and figure 2, and the unit of outputs is ms.

Table 4: table of program consuming time of icc compiler

N	6000	105400	204800	304200	403600	503000	602400	701800	801200	900600	1000000
$icc\_seq(N)$	15	220	438	658	886	1113	1350	1578	1850	2085	2399
$icc\_par\_2(N)$	16	219	436	665	963	1158	1443	1643	1872	2100	2377
$icc\_par\_4(N)$	10	227	463	766	1047	1315	1620	1864	2152	2448	2733
$icc\_par\_8(N)$	12	251	514	771	1060	1309	1593	1863	2181	2476	2763
$icc\_par\_16(N)$	12	258	515	771	1067	1315	1606	1867	2168	2471	2751

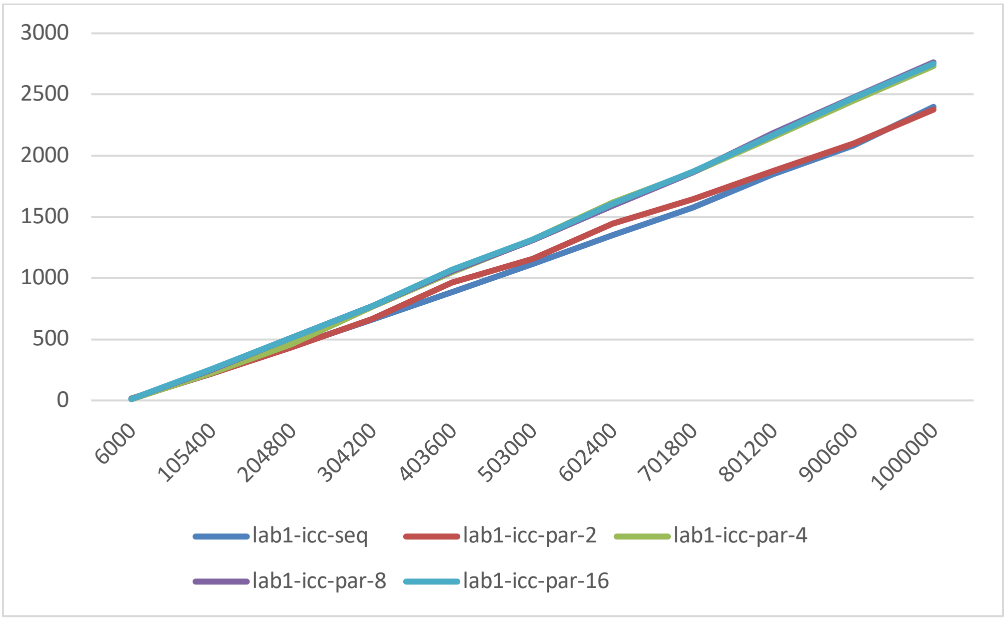

Figure 2: figure of program consuming time of icc compiler

4 Parallel acceleration

Now the acceleration value can be calculated using following formula, since the task of this example is fixed:

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

The results obtained shows that parallel acceleration values do not increase as expected, the reason for this phenomenon might be that the time overhead of creating processes in the program is much greater than the time overhead of performing computational steps within the program. Therefore, when executed with multiple threads, the program's time overhead is much higher than when executed without parallelism.

4.1 Parallel acceleration value of *gcc* compiler:

Figure 3 shows the parallel acceleration values of *gcc* compiler calculated from table 3.

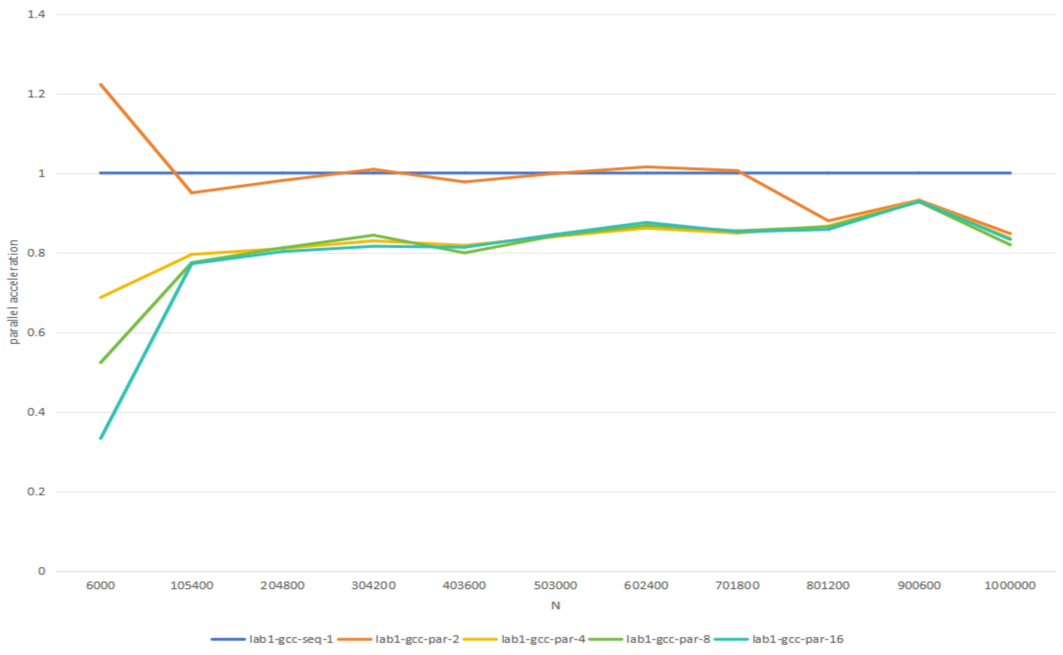

Figure 3: parallel acceleration values of *gcc* compiler

4.2 Parallel acceleration value of *icc* compiler:

Figure 4 shows the parallel acceleration values of *gcc* compiler calculated from table 4.

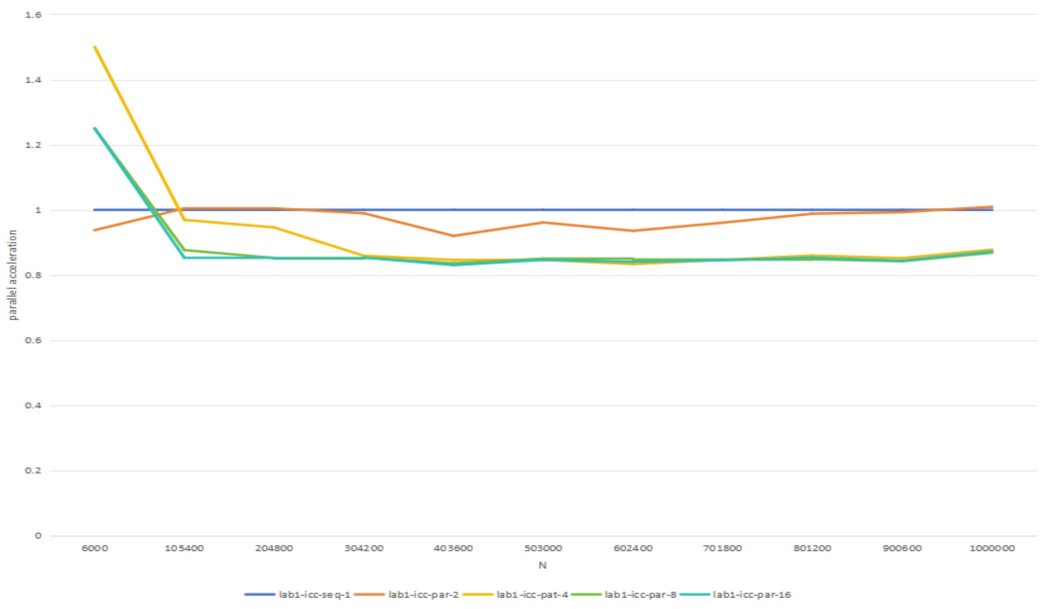

Figure 4: parallel acceleration values of *icc* compiler

5 Conclusion:

5.1 Algorithm' conputational complexity:

It can be concluded from the table 3 and table 4, when the value of N increases, the function value also increases linearly. Therefore, the time complexity of $icc\_seq(N)$ is $O(N)$ or linear time complexity, which is just the same for $icc\_par\_k(N)$, $gcc\_seq(N)$, and $gcc\_par\_k(N)$.

5.2 Analysis:

The results obtained shows that the program running on non-parallel way and parallel way have the similar performance or even worse. The evidence to prove this result can be found in a file called "$lab1.optrpt$", which shows optimization, obtained by intel C++ compiler. It tells that every loop in "$lab1.c$" was not parallelized since there are existences of parallel dependence, and insufficient computational work . Scan every loop in $lab1.c$ and try to suppose the reason why them cannot be automatically parallelized:

1. Two loops in the $main$ function (lab1.c:129,5 and lab1.c:135,3) were not parallelized due to the existence of parallel dependence. This means that there are data dependencies between loop iterations, making it impossible to execute multiple iterations simultaneously. Due to the presence of data dependencies, the compiler could not automatically parallelize these loops.

2. One loop in the $main$ function (lab1.c:149,6) was unrolled during vectorization (Peeled loop) but was not parallelized due to insufficient computational work. This indicates that the amount of computation within the loop is too small to effectively distribute among multiple threads for a performance boost.

3. Another loop in the $main$ function (lab1.c:68,5 and lab1.c:73,5) was vectorized when inlined into another loop but was not parallelized, primarily due to insufficient computational work. Additionally, the compiler pointed out the existence of parallel dependence, which is another reason for not parallelizing it.

4. A loop in the $main$ function (lab1.c:81,2) was also multi-versioned but was not parallelized due to insufficient computational work and the existence of parallel dependence. The compiler attempted different versions of the loop but still could not meet the requirements for parallelization.

5. A loop in the $main$ function (lab1.c:28,9) was labeled as "not a parallelization candidate," indicating that it is not suitable for parallelization. The loop may contain code structures or dependencies that are not conducive to parallel execution.

6. Another loop in the $main$ function (lab1.c:91,5) was vectorized but not parallelized due to insufficient computational work. The loop contains too few computations to effectively leverage the advantages of parallel execution with multiple threads.

7. A loop in the $combSort$ function (lab1.c:14,17) was not parallelized due to being labeled as "not a parallelization candidate." The loop may include code structures or dependencies that are not suitable for parallelization.

8. A loop in the o$utputArray$ function (lab1.c:52,2) was not parallelized due to the existence of parallel dependence. Data dependencies within the loop prevented parallelization by the compiler.

In summary, loops were not parallelized primarily for two reasons: the existence of parallel dependence and insufficient computational work. Parallel dependence means that there are data dependencies between loop iterations, preventing simultaneous execution of multiple iterations. Insufficient computational work indicates that the amount of computation within the loop is too small to effectively benefit from parallel execution with multiple threads. Some loops also contain code structures or dependencies that are not suitable for parallel execution, contributing to their non-parallelization. To improve the program's parallelism, it may be necessary to redesign algorithms or loop structures to reduce dependencies and increase computational work, enabling better utilization of the performance of multi-core processors.