Assignment 3

CSI211—02 Programming Practice, Fall 2011

Due date: 11:59pm, December 6 (Tuesday) 2011.

Introduction

The purpose of this third assignment is to learn about potential performance bottlenecks and performance evaluation of parallel programs. This assignment consists of one example in 4 versions. You must hand in your programs plus a small report as explained in Section "Deliverables and Submission" below.

Environment

You can solve this assignment on any computer that has Linux, GNU GCC, Intel VTune and the Pthreads library installed. A convenient way will be to do the assignment on our server (elc1.cs.yonsei.ac.kr). Our Linux tutorial contains all the information you need to access the server. Our VTune tutorial will tell you how to use the Intel VTune performance analyzer on elc1 or on your own computer. Please note that you should do the *performance evaluation* on the elc1 server. The main reason is that the elc1 server provides 2 quad-core CPUs, which means that you can have up to 8 Pthreads executing in parallel (true parallelism!). If you do this performance evaluation e.g., on a dual-core CPU, then only 2 Pthreads can execute at any one time.
If using VTune on elc1 (command-line interface or GUI X11 forwarding) provides too tedious, you are free to do the VTune performance analysis part on your own multicore computer.

Programming Problem: Counting 3's

In this example we count the occurrences of the number "3" in input data. Input data will be provided in a file according to the following format:

- The first line contains a single number that specifies the number of data items in the file. You can assume that this number is in the range 1 <= N <= 64 000 000 (that is, the input data will consist of no more than 64 million numbers).
- Every subsequent line contains one number (data item). You can assume that numbers are in the range 1 <= N <= 100.

Example Input file (2 occurrences of number 3):

```
4       // 4 data items in file
1       // first data item
3       // second data item
3       // third data item
100     // forth data item
```

One example input file is provided in directory /opt/pp/assignment3/list16M.txt. Such an input file will be passed as a command-line argument to your program. To count the number of 3's in the input data, you should read-in the input data from the file and store it in a one-dimensional array. Because numbers range between 1 and 100, you can use an **`unsigned char`** array (but you should store numbers, and not characters, e.g., the value 33 and not two times the character '3' for an occurrence of number "33" in the input). To explore performance-related pitfalls with parallel programming, we will implement four versions of this program.

Version 1: A sequential version of counting 3's to get the functionality right.

Version 2: A parallel version using a single global variable for counting. We will experience that a single global variable introduces high lock contention and overhead in the memory hierarchy (caches).

Version 3: A parallel version of counting 3's occurrences, where each thread maintains its own private counter variable. As we will see, performance can be improved, but *false sharing* can still have a huge negative impact on program performance.

Version 4: A parallel version of counting 3's that avoids false sharing.

Versions (1) to (4) above follow the lecture on "Performance of Parallel Programs", Slide 18 and onwards.) Our running example in the lecture was on how to sum up numbers in an array; in this example we count the number of 3's in an array.) By carefully walking through a real-world example, you will get hands-on experience on how to avoid potential performance bottlenecks of parallel programs.

Version 1: sequential computation

Version 1 is the sequential version of counting 3s. It consists of three parts: (1) reading the input data into a one-dimensional array, (2) counting the number of 3s in that array, (3) reporting the number of 3s found.

- Usage of the sequential version:

./count_seq <filename>

- Output Format

(occurrences of 3's, followed by a newline (`"\n"`).

- Sample Output

```
[elc1: assignment3]$ cat list.txt
3
2
3
33
6
[elc1: assignment3]$ ./count_seq list.txt
1
[elc1: assignment3]$
```

Note that we count the occurrences of number '3', and not digit '3'!

Program Instrumentation and Performance Measurements

We will measure the wallclock time to evaluate the performance of our program. We determine wallclock times using the **`gettimeofday()`** system call. This system call returns the number of seconds and microseconds that have elapsed since January 1, 1970.

By adding calls to **`gettimeofday()`** to our program, we can determine the program's execution time. Adding code that gathers information during program execution is called **program instrumentation.** To see how our program instrumentation works, assume that the **`main()`** function of your program has the following structure (in pseudo-code). The code highlighted in yellow is the program instrumentation code that has been added to determine program execution times.

```
1  #include <sys/time.h>
2  struct timeval startread, startcalc, finish, readtime,
3                 calctime, overalltime;
4
5  int main(...) {
6    gettimeofday(&startread, NULL);
7    read input data
8    gettimeofday(&startcalc, NULL);
9    count the number of 3s in the input data
10   gettimeofday(&finish, NULL)
11   timersub(&startcalc, &startread, &readtime);
12   timersub(&finish, &startcalc, &calctime);
13   timersub(&finish, &startread, &overalltime);
14   output result
15 }
```

Our program instrumentation follows the measurement scheme introduced in the lectures. We are interested in the wallclock time that elapses between the start of a computation and the end of a computation. In Lines 2 and 3 we declare **`struct timeval`** variables that we will use to record wallclock times. When we pass such a **`struct timeval`** variable as argument to the **`gettimeofday()`** systemcall, the Linux kernel will record the time elapsed since January 1, 1970, in this variable. Such a measurement is called a **time-stamp**. We take time-stamps before reading

the input-data (Line 6), after reading the input data (Line 8) and at the end of counting (Line 10). By taking two time-stamps and subtracting the later one from the earlier one, we can determine the wallclock time that has elapsed between the two time-stamps. The **`timersub()`** system call does just that: it takes two timestamps (argument 1 and 2), and returns the difference in the third argument. You find all the details of the **`gettimeofday()`** system call in the corresponding man-page (i.e., "man gettimeofday"). In lines 11 – 13 we compute the execution times for reading the input data, counting, and reading + counting.

Interpretation: we assume that we cannot parallelize the time to read data from the file (sequential part), and that we can parallelize counting 3s.

Question 1: Using Amdahl's Law on the execution times for I/O and counting, what speedup can you expect for this program?

Version 2: one global counter variable

Version 2 uses one global counter for threads to count the number of 3s. Assume that you write the below pseudo code for your thread function.

```
int Count

count_thread()
{
   for(i in this thread's share of the work)
        if (array[i] == 3) Count++
}
```

Unfortunately, this seemingly straightforward code will not produce the correct result because it contains a race condition in the statement that increments the counter value (the 'Count ++' statement). Therefore, this code requires us to implement mutual exclusion to protect the counter variable against concurrent access.

Note that in the above pseudo-code, the critical section is the 'Count++' statement. To ensure mutual exclusion, you can use a mutex or a binary semaphore. It is quite interesting to compare the performance obtained using a mutex vs. the performance obtained from using a semaphore! (Semaphores are more complicated, so they are less efficient...)

The number of threads that you should use for counting 3s is given to your program as the second command-line argument. You can assume that the required number of threads will be no larger than 64.

- Usage of the parallel version: ./count_par <filename> <number of threads>

Note: the input file format is the same as for Version 1).

- Output Format

same as for Version 1.

- Sample Output

[elc1: assignment3]$ ./count_mutex list.txt 4

1

[elc1: assignment3]$

Performance Measurements:

In this programming assignment, we will evaluate the performance of this program. We will focus on the execution time of the *parallel part only*. In this way, you should measure the execution time in your **`main()`** function from before creating threads until after joining threads (see the below pseudo-code).

```
1  int main(...) {
2    ...
3    read input data
4    gettimeofday(&startcalc, NULL)
5    create all threads
6    join all threads //Threads terminate after completing their job.
7    gettimeofday(&finish, NULL)
8    timersub(&finish, &startcalc, &calctime);
9    ...
10 }
```

Please measure and report the execution times for 1 thread, 2 threads, 4 threads, 8 threads and 16 threads, and compare them to the calculation time of the sequential version. Explain the "speedup" that you gained. Please see Section "Reporting your results" below on how to report your performance data.

Version 3: private counter variables for each thread

Version 3 uses a private counter variable for each thread. Thereby we avoid contention of the lock protecting the global counter variable. Compared to Version 2, threads do not have to queue anymore for every single counter update. Only after a thread finishes its share of the work, it adds its findings to the global counter variable. Consider the below pseudo code.

```
int overall_count = 0;
int private_count[NumberOfThreads] = {0};

count_thread()
{
```

```
        for(i in this thread's share of the work) {
            if (array[i] == 3)
                Private_count[MyThreadID]++
        }
    lock(mutex)
    overall_count += Private_count[MyThreadID];
    unlock(mutex)
}
```

- Input Format

./count_sum [filename] [number of threads]

Note: Input file format and output format are the same as for the previous versions.

Please measure and report the execution times for 1 thread, 2 threads, 4 threads, 8 threads and 16 threads, and compare them to the calculation time of the sequential version. Explain the "speedup" that you gained. Please see Section "Reporting your results" below on how to report your performance data.

Version 4: Counting 3's occurrences using padding

If you check the speedup obtained from Version 3, it is not as fast as expected. The remaining performance bottleneck is due to false sharing of counter variables because of CPU cache lines. If two counters end up in the same cache line, then the cache line needs to be copied between the caches of different CPU cores whenever a thread references his counter. This situation is almost as bad as if we would use a single global counter variable (see the lecture on "Performance of Parallel Programs", the slide on "False Sharing"). The below diagram depicts a single cache line that holds the counter variables of several threads.

Multiple counter variables in a single cache line:

private_count[0]	private_count[1]	private_count[2]	...	...

To solve this problem, we need to use extra space (aka **padding**) to force every counter variable into a separate cache line.

```
struct padded_int
{
    int value; // 4 bytes
```

```
        char padding[60];    // 60 bytes of padding, for 64 byte cache line sizes
} private_count[NumberOfThread];
```

The above pseudo-code shows a C struct that avoids false sharing of counter variables for caches with 64-byte cache lines (see also the figure below.

Multiple counter variables, each in a cache line of its own:

private_count[0]	padding
private_count[1]	padding

.....

- Input Format

./count_padd <filename> <number of threads>

Note: Input file format and output format are the same as for the previous versions.

Please measure and report the execution times for 1 thread, 2 threads, 4 threads, 8 threads and 16 threads, and compare them to the calculation time of the sequential version. Explain the speedup that you gained. Please see Section "Reporting your results" below on how to report your performance data.

Generation of bar-charts

To report performance results, we will use bar-charts. The following example shows how 'gnuplot' can be used to generate bar-charts (gnuplot is installed on the elc1-server).

```
[elc1: assignment3]$ cat graph_script.gnu
set terminal png
set output 'result.png'
set boxwidth 0.5 absolute
set style fill solid
set title "Execution time Version 1 (Demo only with arbitrary data!)"
set noxtics
set ylabel "Time in milliseconds (ms)"
set grid
set yrange [0:1200]
set xrange [-1:6]
```

```
set xtics ("serial" 0, "T=1" 1, "T=2" 2, "T=4" 3,   "T=8" 4, "T=16" 5)
set size 0.7, 0.7
plot "-" title "Execution time" with boxes
470
480
322
500
600
700
e
```

[elc1: assignment3]$ **gnuplot graph_script.gnu**

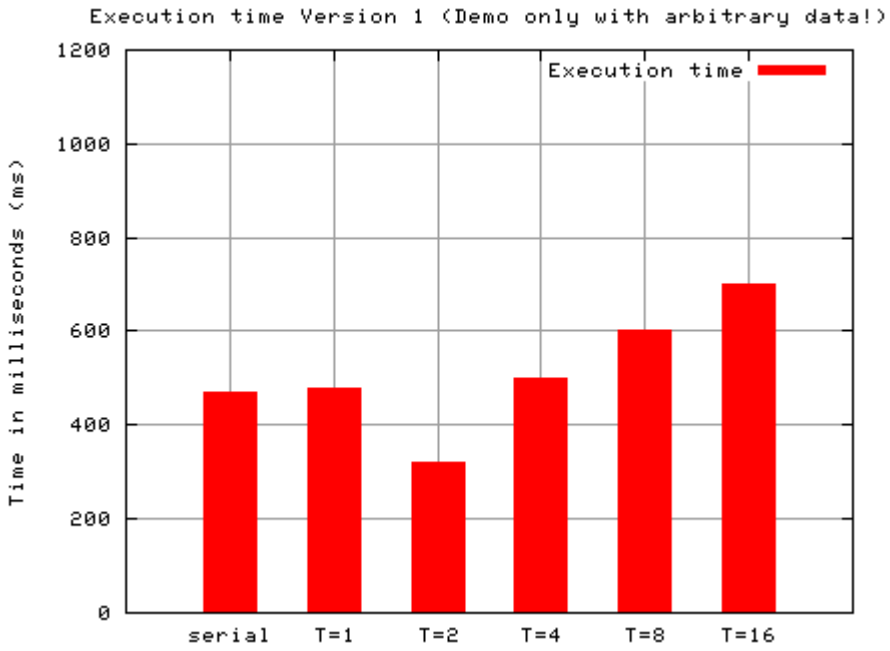

Figure 1

When executing the above script, gnuplot will generate file result.png (see Figure 1 above). Like PPM, PNG is a graphics format that you can view with the Gimp program. You can modify the above script to generate your bar-charts. You find more information on gnuplot at http://www.gnuplot.info/, together with a Windows version (gp426win32.zip).

Note: you find the above gnuplot script in directory /opt/pp/assignment3 on the elc1 server.

Performance Analysis with Intel VTune

After implementing and manually instrumenting your code with calls to gettimeofday(), we now use the Intel VTune performance analyzer to determine performance and locking overhead of your code. To do so, you are asked to run version 2 (count_mutex!) and version 4 of your count3

program with Intel VTune to determine performance bottlenecks and locking overhead.

Question 2: Please determine the performance bottleneck(s) and locking overhead with VTune and **explain** the data provided by VTune in your report (about 0.5 page). Please also provide the CPU-type and number of cores of the computer on which you ran the VTune performance analyses.

Report

You should implement Versions 1 – 4 of the count 3s program and determine program execution times as outlined above. For Version 2, 3 and 4, you should report your measured execution times using a bar chart like the one in Figure 1 above. Please name your chart files "resultV2.png", "resultV3.png" and "resultV4.png" and submit them with your source code (see Section "Deliverables and Submission" below). Please also submit a short README.txt file where you answer Question 1 and 2 outlined above plus the explanations for your program's performance (gettimeofday instrumentation) with Versions 2, 3 and 4.

Marking

We will check the correctness of your programs. In addition, we will check the speedup that you have achieved with Version 4. Programs that fail to achieve sufficient speedup will receive fewer points. Programs that are slower than the sequential version will receive 0 points.

Testing will be done on the elc1 server, so please make sure that your program compiles and executes there. Programs which fail to compile, crash, or produce no or the wrong output will receive no points.

Please note that all assignments are *individual* assignments. You are most welcome to discuss your ideas with your colleagues, but 'code-sharing' or 'code-reuse' is not allowed.

Deliverables and Submission

Please create a directory, e.g., ~/myassignment3, and put the following files in this directory:

- count_seq.c
- count_mutex.c
- count_sum.c
- count_padd.c
- resultV2.png
- resultV3.png
- resultV4.png
- README.txt (your report)

To submit, after copying the above files in your submission directory, you must cd to the submission directory and type 'submit'. This will collect your files and copy them to our

repository.
[you@elc1 ~] cd ~/myassignment3
[you@elc1 myassignment3] submit

Please note: There is no submission unless you issue the 'submit' command! Failing to issue the 'submit' command will result in 0 points. Please note also that if you re-submit, then a previous submission will be overwritten and a new timestamp will be recorded.

If you have questions, don't hesitate to contact the TAs. ☺

Good luck!