Go 와 C++ TBB 의 병렬처리 비교

박동하, 문봉교
동국대학교 컴퓨터공학과
e-mail: luncliff@gmail.com, bkmoon@dongguk.edu

Comparison of Go and C++ TBB on Parallel Processing

Dong-Ha Park, Bong-Kyo Moon
Dept. of Computer Science & Engineering, Dongguk University

Abstract

Applying concurrent structure and parallel processing are a common issue for these day's programs. In this research, Dynamic Programming is used to compare the parallel performance of Go language and Intel C++ Thread Building Blocks. The experiment was performed on 4 core machine and its result contains execution time under Simultaneous Multi-Threading environment. Static Optimal Binary Search Tree was used as an example.

From the result, the speed-up of Go was higher than the number of cores, and that of TBB was close to it. TBB performed better in general, but for larger scale, Go was partially faster than the other.

1. Introduction

For nowadays, multi-processor environment became normal and C++ is being used for performance critical area with modernized standards and concurrency/parallel libraries [1]. However, Go language supports easier parallel programming model with its language primitives, [2] gathering its interests constantly. [3]

In 2012, Doug Serfass and Peiyi Tang's research [4] compared the parallel performance of C++ TBB library and Go language. Reviewing it, there are 2 goals for this research. First one is to re-implement with the latest version of both languages, and the other is to compare their performance.

Section 2 summarized related works for this research. Section 3 explains common issues of target problem. Section 4 is about experiment concept and approaches. Section 5 analyzes its results. At last, section 6 concludes the research.

2. Related Works

Ensar Ajkunic et al. [1] compared 5 parallel programming models of C++ libraries. They parallelized matrix multiplication and compared code implementation for each model.

Doug Serfass and Peiyi Tang [4,5] gave the motivation for this research. Peiyi Tang [5] analyzed the performance of general fork-join parallelism for early Go. It's target problem contained processing of static optimal binary search tree, which requires synchronization between sub-problems. Doug Serfass [4] extended it and compared Go with C++ TBB.

Neil Deshpande et al. [8] analyzed the old code of Go runtime scheduler and its mechanism. The codes are now converted from C to Go, but the overall structure is still being retained.

Carl Johnell's research [10] compared Go and Scala with matrix multiplication and chained multiplication problem. In the case, Scala performed better with a few number of actors.

Arch Robinson et al. [13] discuss optimization of C++ TBB for nested parallelism and cache affinity for task processing. It gives an explanation about work stealing mechanism and internal flow of TBB.

3. Research Problem

Task-Level Parallelism

This research applied task-level parallelism. In the scheme, a task is a unit of work, which is an ordered group of instructions. And a processor is a logical entity that executes set of tasks. It can be OS process/thread, or runtime abstraction of languages, such as Erlang process and Goroutine in Go.

However, parallelism requires synchronization of data and operation for correct program order. Because of its complexity, recent parallel programming interfaces provide task scheduling system. It manages tasks' state and maps them to processors.

Processing of OBST

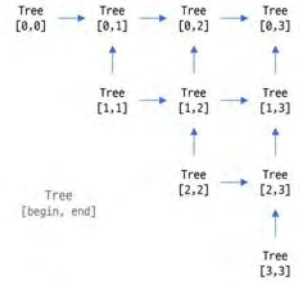
Like previous research, [4, 5] this research uses Static Optimal Binary Search Tree as an example of dynamic programming. Optimal BST is one kind of balanced search tree that is defined recursively like Figure 1. [6] In the figure, E means expected cost and W means total weight, which is sum of all possibilities for tree vertices. Approach for its estimation is separating it to a group of sub-trees, and storing them in the memory location.

$$E = E_L + E_R + W$$
$$E_{i,i-1} = W_{i,i-1} = B_{i-1} \text{ for } 1 \le i \le n+1$$
$$W_{i,j} = W_{i,j-1} + A_j + B_j$$
$$E_{i,j} = \min_{i \le r \le j} (E_{i,r-1} + E_{r+1,j} + W_{i,j}) \text{ for } 1 \le i \le j \le n$$

(Figure 1) Definition of OBST

However, since those sub-trees have data dependency, the algorithm must process smaller sub-trees in prior to larger

trees. It must consider dependency relation, which constructs a directed-acyclic graph in this case. Figure 2 visualized it. After applying task-based design, each sub-tree becomes a task and arrows become the point of synchronization.



(Figure 2) Reduced Graph of Data Dependency

4. Experiment

Experiment and Code

The experiment was performed on a machine with Windows 10 OS (Pro, Build 14393). Its CPU was Intel u7-6700HQ with AMD64 ISA, 4 core and 6MB cache. It supports 8 logical thread contexts with Intel Hyper Threading. The version of Go was 1.7.4 and Intel TBB library was 2017 Up3. C++ code was built with MSVC v14 in Microsoft Visual Studio 2015 Up3.

There were 2 changes in code. First, source embedded constants are removed. The previous code required it for static allocation of OBST. But in this research, memory was allocated dynamically and related factors are provided with command line argument at launching time.

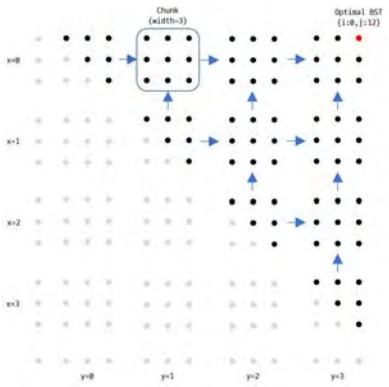
Second, it took account of resource clean-up cost. Since previous design managed space overhead for parallelism at global scope, memory for garbage collection and Goroutine could lead to incorrect time measurement. Like Figure 3, the overhead was managed locally and execution time was measured only for the scope.

```
OBST tree = MakeTree(N);
Timer t;
t.reset();
if (parallel == true){
    Setup();
    ParallelEvaluate(tree, VP)
    CleanUp(); // GC + Scheduling resources
}
else {
    SequentialEvaluate(tree)
}
Duration elapsed = t.pick();
```

(Figure 3) Simplified Code for Performance Evaluation

Concepts

Figure 4 shows the implementation view for the target problem. There are 3 major factors for the program. First, N is problem size. Here, its value is 12. But because of dummy (gray dots), actual memory space usage follows $(N+1)^2$. N were fixed to 2048 and 4096 for the experiment.



(Figure 4) Concepts for the problem

VP determines the total number of chunks. Its value is 4 in the figure, and 10 chunks are created. Since creating tasks for each tree (black dot) is wasteful, trees are chunked to a group like the rounded rectangle. Therefore, with high VP, the scale of each task becomes small and it will result in frequent synchronization (more arrow). This factor varied from 1 to 2048, with exponential growth of 2 (1,2,4,8…)

NP is the number of processors. Since threads are managed by the scheduler, this factor will affect scheduling cost. Because of machine limitation, NP varied from 1 to 8. Execution time for each condition was estimated 5 times in millisecond and averaged.

Approach of Go/TBB

The programming model of Go was designed after Communicating Sequential Processes. With garbage collection, [7] It supports Goroutine as its light-weight processor that are spawned and managed by Go runtime. Its scheduler uses a global lock to map runnable Goroutines thread-safely. [8, 9] And the processors communicate via channel type which is a lock-based queue with a list of readers and writers. Especially, Go code for matrix referenced Carl Johnell's research. [10]

```
func Chunk(
    tree *obst.Tree,
    i int, j int, width int, dep Dependency) {
    // 1. Wait for pre-set...
    dep.Wait()
    // 2. Sequential processing
    for row := i - 1 + width; i <= row; row-- {
        for col := j; col < j+width; col++ {
            root, cost := tree.Calculate(row, col)
            *tree.Root.At(row, col) = root
            *tree.Cost.At(row, col) = cost
        }
    }
    // 3. Notify to post-set...
    dep.Notify()
}
```

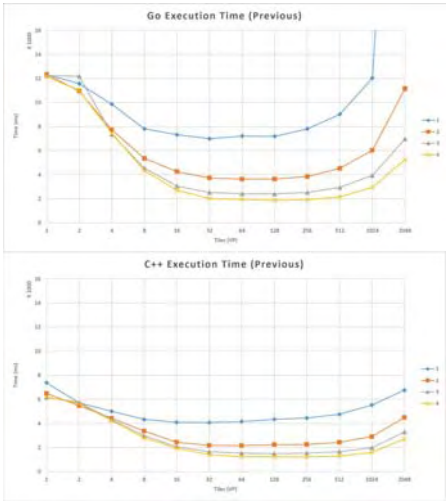
(Figure 5) Go Chunk Processing

Intel's TBB library is thread pool with task scheduling system. [11] It uses OS thread as its processor, and they handle tasks with runtime polymorphism. Tasks' dependency is expressed with atomic reference counter. When its counter becomes 0, the task becomes executable and mapped to a thread [12]. The scheduler's policy is designed to utilize cache efficiently. [13] For work execution, it traverse task

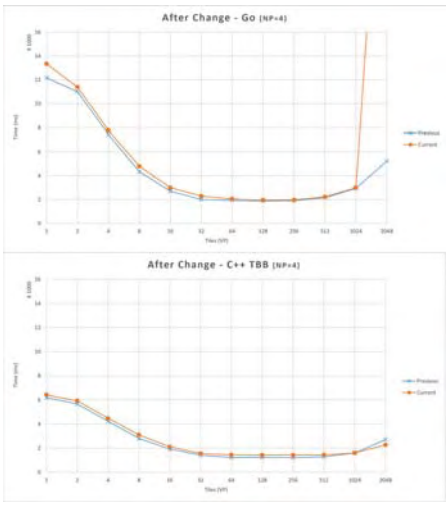
graph in depth-first order so that most recent contents can be reused. For work stealing, breadth-first order is applied to avoid data race.

5. Result Analysis

Figure 6 is execution time of the previous code. Its X-axis is log scale with VP, and Y-axis is elapsed time. Figure 7 summarizes the effect of changed code. It performed slower than previous and lost performance significantly with highest VP.

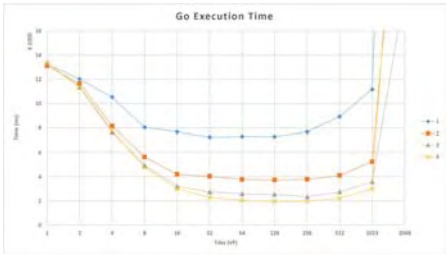


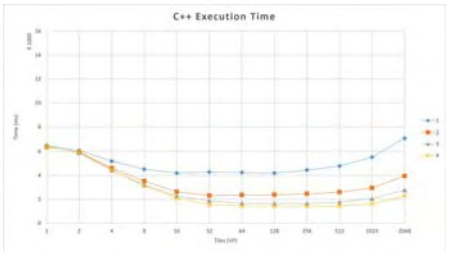
(Figure 6) Previous Code's Execution Time



(Figure 7) Change of code and its effect

In general, TBB performed better than Go and their aspect was quite similar. With fixed N=2048, C++ TBB and Go took 6255.4, 12689.6 ms respectively.





(Figure 8) Execution Time with N=2048

Because of parallel code's chunking policy, the performance increased until VP grows to 128, but decreased after VP=256. When VP=1, the coverage of a single chunk is 2 times larger than that of sequential code. In the case, the elapsed time for TBB was 6488.6 ms (Go: 13266.2 ms). For VP=64, with much lesser coverage, the time for both decreased to 4241.6 (TBB) and 7309.4 (Go). As VP exceeds 256, mode space, scheduling and synchronizations drop performance. Considering the number of chunks follows $O(VP^2)$, their total cost isn't negligible.

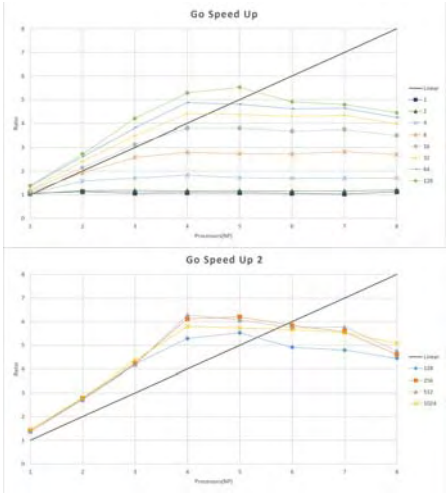
Assuming TBB's reference counter is implemented with atomic integer, its cost depends on architecture. However, Go's channel-based code can trigger scheduling much frequently. Which can be a bottleneck with global lock. Figure 9 shows channel waiting code that can cause 2 times more scheduling if channels are empty.

```
type Dependency struct {
    PreSet  [2]chan int
    PostSet [2]chan int
}
func (rcv *Dependency) Wait() {
    if rcv.PreSet[0] != nil {
        <-rcv.PreSet[0] // Possible Scheduling
    }
    if rcv.PreSet[1] != nil {
        <-rcv.PreSet[1] // Possible Scheduling
    }
}
```

(Figure 9) Go Synchronization Code

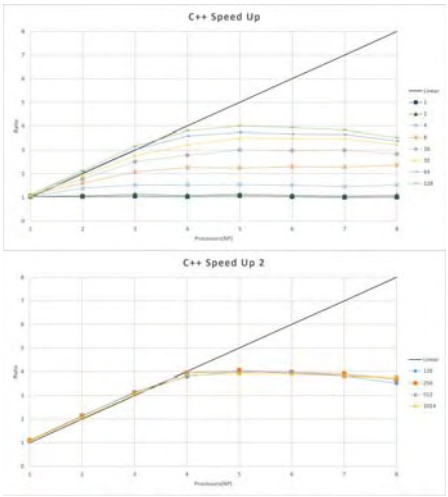
Interestingly, Go's performance often dropped vastly when VP=2048. With this highest VP and NP=1, the execution time was in range of 72140~90605 ms. However, with NP=4, it varied between 4899~66603 ms. Based on profiling [14], the major reason for this phenomenon was Go runtime package. In worst case, it consumed 84% of total CPU time, dropping net processing time to 20.23%. However, when the time was under 5000 ms, it took only 11% with 70.22% of net processing.

Possible origin of it is millions of Goroutines. Since each Goroutine owns segmented stack and guard pages, bad scheduling can make most of them to wait. Eventually, it explodes working set and residence memory of Go application. With poor data locality of high VP, this leads to abnormal stress for virtual memory management to system and therefore poor CPU utilization.



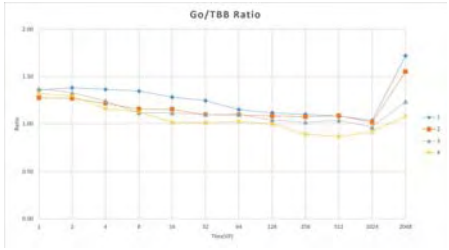
(Figure 10) Go Speed up with N=4096

When N=4096, the execution time increased at least 9 times of that under N=2048. Figure 10 summarizes speedup in the case. As thread context increases, SMT environment resulted in lower performance. Go's speedup was over the linear until NP reaches the maximum number of physical processor. When it achieved best, the performance was 6.28 times faster than sequential processing. Speed up of TBB didn't go over the linear line, and its best speed up ratio was 3.95.



(Figure 11) TBB Speed Up with N=4096

With problem size of N=2048 and NP=4, TBB was always faster than Go. Go/TBB ratio was 1.37 with VP=128. But for larger size(N=4096), Go performed better than TBB under VP=256 and VP=512 condition. The ratio of them was 1.12 and 1.15 respectively.



(Figure 12) Go/TBB Execution Time Ratio with N=4096

6. Conclusion

This research compared parallel performance of TBB and Go. Its target was static OBST with task-level parallelism. Processing codes were written again. And it considered overhead of garbage collection and destruction of scheduler for execution time. Experiment environment provided 4 physical core and 8 logical threads. The speed-up was almost linear for physical cores, but as context increases, the throughput decreased. Additionally, millions of Goroutine caused poor performance with virtual memory and scheduling issue.

With 4 thread, TBB's best speed-up ratio was 4.41, and that of Go was 6.53. When both languages performed best, the TBB was 1.37 times faster. On larger problem size, Go's best performance was higher than TBB with ratio of 1.15.

With the latest language/library version, Go's performance improved greatly and reduced gap of 2012. Even though TBB has been better in general, the result presented that Go language in parallel processing became much competitive.

References

[1] Ensar Ajkunic et al. "A comparison of five parallel programming models for C++" MIPRO, Proceedings of the 35th International Convention. May 2012.
[2] The Go Language Specification
[3] TIOBE Index for Jan 2017
[4] Doug Serfass, Peiyi Tang. "Comparing Parallel Performance of Go and C++ TBB on a direct acyclic task graph using a dynamic programming problem" ACM, Proceedings of the 50th Annual Southeast Regional Conference. March 2012.
[5] Peiyi Tang. "Multi-Core Parallel Programming in Go", Proceedings of the First International Conference on Advanced Computing and Communications, Jan 2010.
[6] Wikipedia, Optimal Binary Search Tree.
[7] Austin Clements, Rick Hudson. "Proposal: Eliminate STW stack re-scanning". https://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md
[8] Neil Deshpande et al. "Analysis of the Go runtime scheduler", Columbia University
[9] The Go Authors, "/src/runtime/proc.go"
[10] Carl Johnell. "Parallel programming in Go and Scala: A performance comparison". Belkinge Institute of Technology.
[11] Intel C++ Thread Building Blocks
[12] Intel TBB: How Task Scheduling Works
[13] Arch Robinson et al. "Optimization via Reflection on Work Stealing in TBB", IEEE International Symposium on Parallel and Distributed Processing, April 2008.
[14] Russ Cox, Shenghou Ma. "Profiling Go Programs" https://blog.golang.org/profiling-go-programs

Acknowledgement

This research was supported by MIPS(Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW(R7116-16-1014) supervised by the IITP(Institute for Information & communications Technology Promotion)