Green

Fredrik Öberg
fobe@kth.se

Examiner: Prof. Johan Montelius

KTH Royal Institute of Technology, Sweden

January 9, 2021

Abstract

Contents

1 Introduction

This paper is written as a part of the examination of the course ID1206 - Operating Systems; itself part of the operations of KTH Royal Institute of Technology. The course provides knowledge of the principles of abstractions concerning computer hardware as well as virtualization of resources and timetabling of assignments and how those principles can be implemented; mainly as regards to program execution, memory management and persistent storage of data. The purpose of this paper is to get a deeper understanding of the workings of parallelism; a type of computation where processes or threads carry out their executions simultaneously in parallel. Large problems could be divided into smaller ones which could be solved from many execution points by the threads with the intent of increasing the over all performance of a computer system. These workings are investigated by creating and implementing a thread library on the application level as well as test and compare its performance to a natively implemented one.

2 Background

Computer software has, traditionally, been written for serial computation where a problem has been constructed through an algorithm and implemented as a serial stream of instructions. Only one instruction could be executed at a time and when that instruction was finished the next one could be executed. In parallel computing, multiple instructions are executed simultaneously and this is accomplished by breaking the problem into smaller independent parts solved either by different processes or what is called a thread. A thread is often defined as the smallest sequence of programmed instructions that can be managed independently by a scheduler, typically a part of the operating system(OS) the thread is running on. The thread could be considered a separate process except that threads of the same process share address space but has there own stack but share heap and thus can access the same data easily. Since a multi-threaded program has more than one point of execution each thread need to have its own program counter value. These values track where the program is fetching instructions from meaning that if there are two or more threads that are running on a single processor, a context switch must take place. This means that when there is a switch from one of the threads to the other the state of this thread is saved and the new threads value is fetched from memory and used instead. These variables, among others, is what makes up a multi-threaded program and how the threads can concurrently operate in a safe and reliable way is what makes up the most of the difficulties concerning parallelism. In this paper a library of green threads where implemented. Green threads emulate multi-threaded environments without relying on any native OS abilities. They are also managed in user instead of kernel space, enabling them to work in environments that do not have a native thread support. Another difference is that on multi-core processors, many native thread implementations automat-

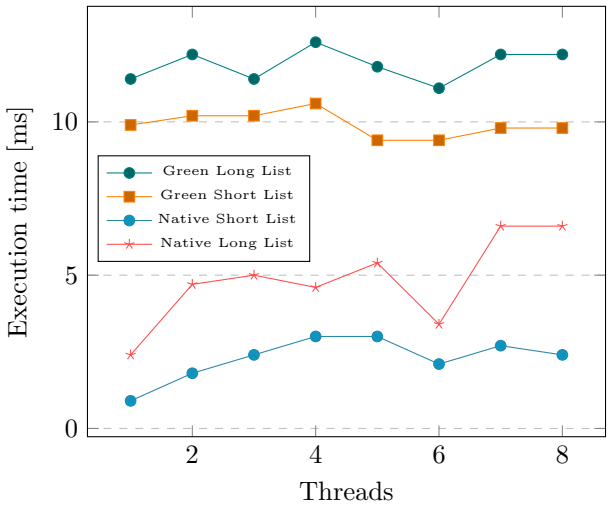

Figure 1: Single Lock List

ically assign work to multiple processors where green thread implementations normally cannot. This means that green threads can be started much faster in some situations since they can be started from within a process in user mode and do not need to call the OS via a costly system call. The problem with green threads however, is that they only simulate multi-threading on platforms that do not necessarily provide that capability; green threads cannot normally use the benefit of multiple cores. This means that when a green thread starts to execute, it blocks the previously running thread as well as all other threads within the process – the purpose of threads and parallelism gets wasted leading most likely to worse performance. The tests performed in this paper is set out to see if this theoretical behaviour can be observed in practice.

3 Set Up

To compare the performance between the implemented green thread library and the native of a Linux Ubuntu OS called "pthread.h", a test application was developed. It was based on the threads inserting a set of randomly generated number into a linked list unless that generated number already was in the list. If that was the case then the thread was supposed to remove it. These tests where done for 5000 iterations in 10 consecutive executions where the average execution times where documented. They were performed on an eight core central processing unit(CPU) with the testing application having one up to eighth threads working concurrently on the linked list.

Two versions of the program was developed; one with a single lock for the entire linked list which the threads were to acquire before being able to insert an element, and one where each element had one lock each. This was so that

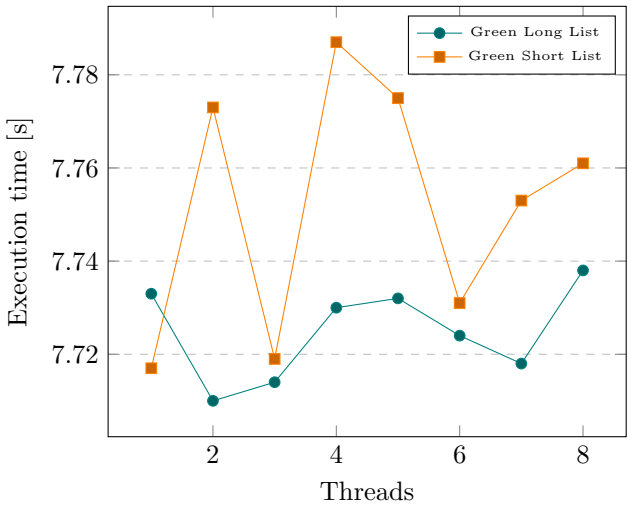

Figure 2: Green Multiple Lock List

multiple threads could work simultaneously on the list and see if there was any significant difference between the two thread libraries. The tests were also done on lists of different length where on the shorter list the numbers could go from zero up to 100 and from zero up to 500 on the longer. This was to see if there was any difference in performance if it was less likely for threads interfere with each other if they got more space in between - something which is likely if the list is larger.

4 Results

The first image is the result of the test with a single lock for the entire linked list. It shows that the green library is quite consistent in its performance regardless of how many threads are being used. The native library perform better over all but seem to performs worse the more threads are being used. The test result on the list where very element got its own lock showed a vastly different result between the green and the native libraries so the result was split up in two images to display it in a meaningful way. The green result shows that the performance on the short list was slightly worse but also mor inconsistent. The native result shows a similar execution time between the short and the longer list versions with a slightly better performance on the longer list. Both of them shows a increase in performance with more threads added. Do note that the result on the multiple lock lists are in seconds for the green and milli seconds on the native version.

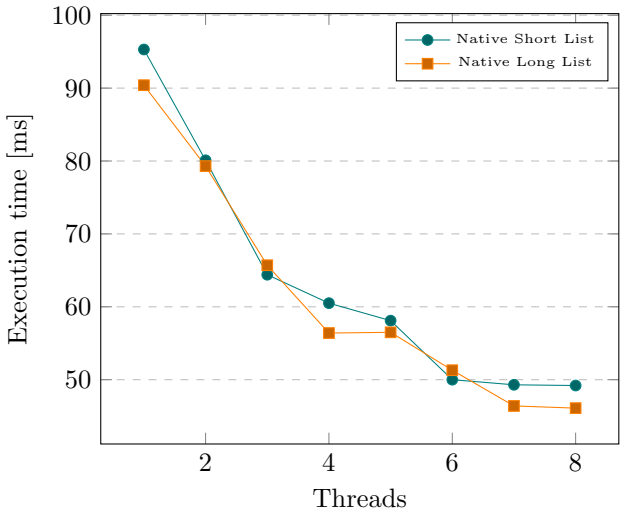

Figure 3: Native Multiple Lock List

5 Discussion

The first thougt after the resul that the green library performs worse than the native one. Before hand it seemed likely that the green would at least perform comparable at least with one thread doing the work. The execution pattern on the single lock list could indicate that it has something to do with code optimization since the execution times are – relatively speaking – close to each other. The pattern that the green library performs similar regardless of the number of threads seems reasonable since there is only one thread executing at the time and with few costly system calls it should not impacat the performance that much. The pattern of the native implementation performing slightly worse with more threads seem also reasonable. That is because there is only on thread executing on the list reagerdless of how many CPU concurrently being used so the benefit of parallelism should be wasted and the cost of the context switch between them should make the performence decline, which is a tendency we can see.

The result on the test with a list with multiple locks is – to some degree – astounding. The pattern of the native library performing better than the green and even better with more threads being used is reasonable. When the threads can work concurrently on the list it is reasonable that more work could be done and the practice of parallelism could shine. But that the green version performs worse on a factor of 100 at least suprised the author. We can state that some of it might be the same optimization issue – as was reasoned with the test on the single lock tests – but the green library should at least be almost as close to the native when running one thread as it was with one lock. The result likely has something to do with the test routine that is explaining the

result. The green locking and unlocking procedure might be culprit – at least a logical one. If those procedures ar only a few persent worse in execution time, the thousands of more of these procedures when the threads traverse the list by fetching and releasing each elemnts lock might add up to the wastly difference in execution time. To conclude that hypothesis, more and deeper testing need to be performed, however.

6 Conclusions

Parallelism used correctly is a powerful tool, indicated by the test results in this paper. When the threads worked concurrently the execution times where halved which indicates that there is a lot performance that could be gained. The result also showed that used incorrectly, the performance could worsen with more threads being used. That is if the environment the threads works in not take advantage of the strengths of parallelism. It also showed that a highly optimized version – which the native library should be considered – could vastly outperform a lesser one. So the lesson is learn how a tools works, how to use it and keep it sharp and you could get more work done if not.