Green

Fredrik Öberg
fobe@kth.se

Examiner: Prof. Johan Montelius

KTH Royal Institute of Technology, Sweden

January 10, 2021

Abstract

This paper examines how parallelism works in a computer system, how it could be implemented and what benefits could be achieved by using it. This is done through performance tests between a native thread library and a, for the purpose of this paper, self-implemented one. The results of the tests indicate that parallelism is not a tool for all situations. Parallelism could, indeed, increase the performance by a large factor, but could also make a system perform worse if used incorrectly. The results also indicates that a non-optimal parallelism implementation could worsen the performance vastly compared to a more optimized one.

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 overall performance of a computer system. These workings are investigated by creating and implementing a library for parallel execution 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 and so on. In parallel computing, multiple instructions are executed simultaneously. 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. In that address space each thread has their own stack but share a common heap and thus can access the same data easily. Since a multi-threaded program has more than one point of execution, each thread needs 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 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. This also means that green threads normally can be started

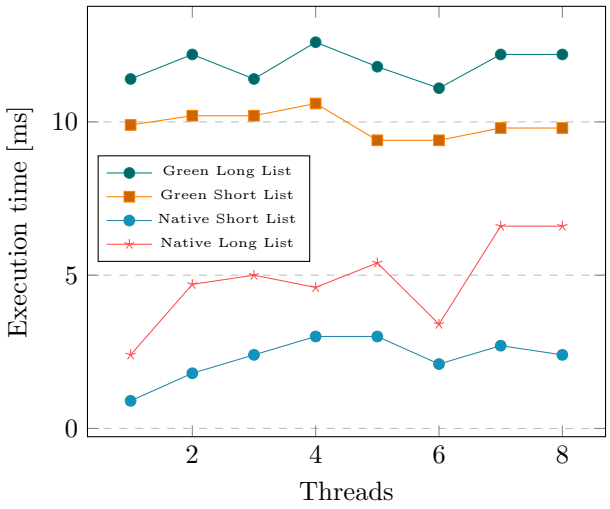

Figure 1: Single Lock List

much faster in some situations since they can be executed from within a process in user mode and do not need to call the OS via a costly system call. Another difference is that on multi-core processors, many native thread implementations automatically assign work to multiple processors where green thread implementations normally cannot. This makes up the main problem with green threads since the emulation of multi-threading they provide results in that they cannot normally use the benefit of multiple cores. 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 than intended. The tests performed in this paper is set out to see if this theoretical behavior can be observed in practice.

3 Set Up

To compare the performance between the implemented green and the native thread library 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 by order of size 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 that number instead. These tests where done for 5000 iterations in 10 consecutive executions where the average execution times where documented. The tests were performed on an eight-core central processing unit(CPU) with the testing application having one and up to eighth threads working concurrently on the linked list.

Two versions of the program were developed; one with a single lock for the entire linked list which the threads were to acquire before being able to insert

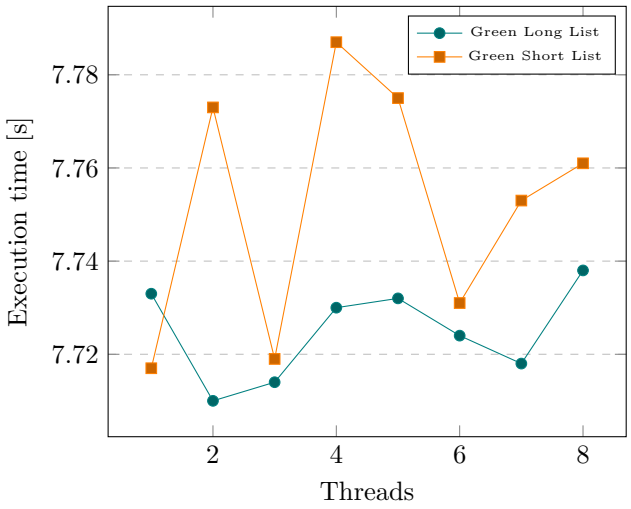

Figure 2: Green Multiple Lock List

an element. In the other version each element had one lock. This means that the threads were to acquire and hold two locks for it to iterate through the list: one of the previous and one for the current for the thread to search safely for the correct position of the threads generated number. This was so that multiple threads could work simultaneously on the list to see if there was any significant difference between the two thread libraries when multiple threads could work concurrently inserting and removing elements. 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 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 it should be if the list is larger.

4 Results

First is the result of the test with a single lock for the entire linked list(see Figure 1). It shows that the green library is quite consistent in its performance regardless of how many threads are being used. The native library performs better over all but seem to perform slightly 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 an easy-to-read way. The green result shows that the performance on the short list was slightly worse overall, but also more inconsistent(see figure 2). 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(see figure 3). Both shows an increase in performance with more threads added. Do note that the result on the multiple lock lists is in seconds

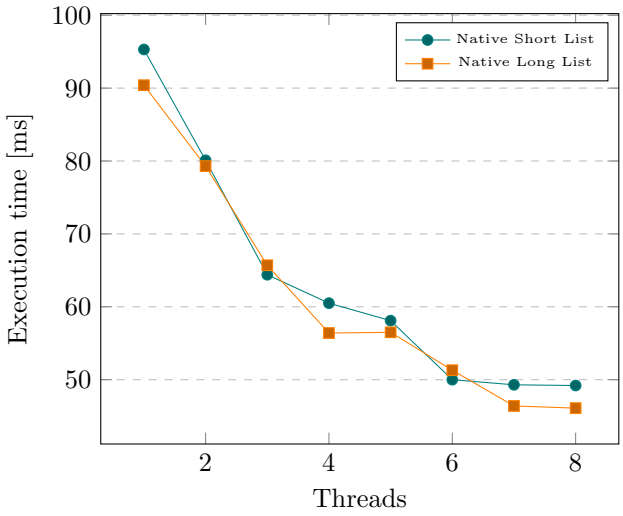

Figure 3: Native Multiple Lock List

for the green and milliseconds on the native version, hence the split into two figures.

5 Discussion

The first thought after the result that the green library performs worse than the native one. Beforehand it seemed likely that the green would at least perform comparable at least when there was only one thread doing the work, but this is not the case in these test results. The execution pattern on the single lock list could indicate that the difference in performance could have something to do with code optimization since the execution times are relatively close to each other(see figure 1), at least compared to the results on the multi-locked list(see figures 2 and 3). 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 impact the performance that much. The inconsistent result (see figure 2) could be an anomaly since there is such a small comparable difference of some hundredth of a second. The pattern of the native implementation performing slightly worse with more threads seem also reasonable(see figure 1). That is because there is only on thread executing on the list regardless of how many CPU concurrently being used. That means that the benefit of parallelism should be wasted, and the cost of the system calls made to switch between threads should make the performance decline. A tendency we can see.

The result on the test with a list with multiple locks was – at a first glance – astounding. The pattern of the native library performing better than the green

and even better with more threads being used makes sense. 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 leading to better performance(see figure 3). But that the green version performs worse on a factor of 100 at least initially surprised the author. It can be stated 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. After given it some thought though, could the result possibly be with the test routine. That the threads need to make one allocation and a deallocation for every iteration on the list should lead to thousands of more locking and unlocking procedures compared to the single lock version. If those procedures in the green library are only a few percent worse in execution time, these thousands of more procedures might add up to the vastly difference in execution time. It is and interesting hypothesis and it makes at least some sense but to conclude that this is the reason for the much worse performance for the green threads, more and deeper testing need to be performed.

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 indicated 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 to be – could vastly outperform a lesser one. So, the lesson is: learn how a tool works, how to use it and keep it sharp and you could get more work done with the same investment in resources.