Pthreads and OpenMP

A comparison

Henrik Swahn

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Contact Information:
Author(s):
Name: Henrik Swahn
Email: hesw91@gmail.com/h_swahn@hotmail.com

University Advisor:
Name: Daniel Häggander
Department: Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Internet : www.bth,se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Blabla

Table of Contents

1 Introduction

1.1 Background

Ever since the 1970's when the microprocessor that we know today was developed and launched increase in it's performance been achieved by increasing the clock speed. All the way until early 2000 the increase in clock speed was statically, every two years the number of transistors that would fit on a chip would double. This is more commonly known as Moore's Law[1].

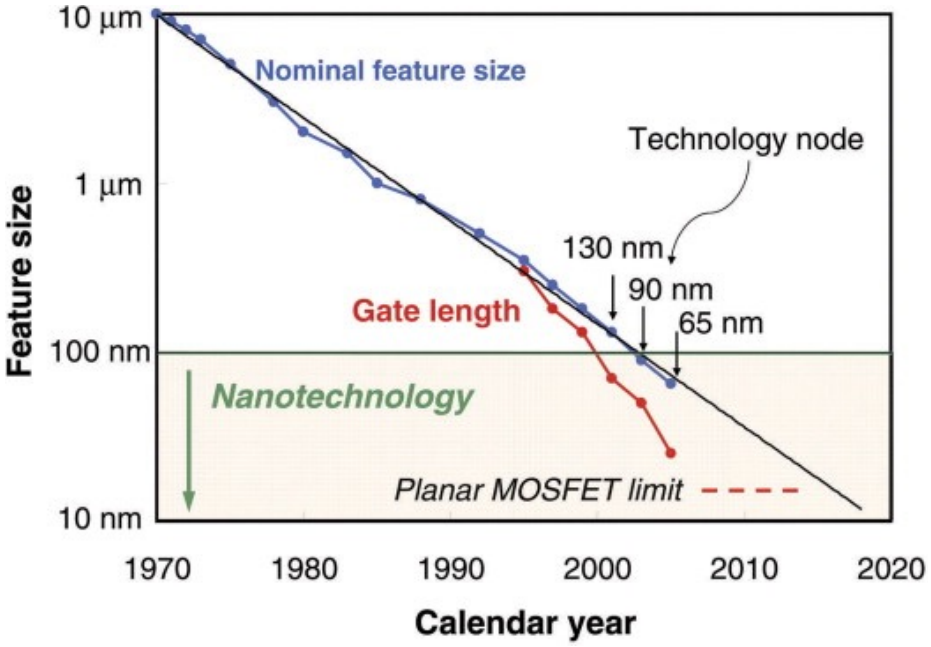

Figure 1 - Transistor size decrease

When increase no longer could be achieved by just increasing the number of transistor on a chip and the clock speed due to high power consumption and heat other means to increase performance had to be developed. One technique that is used to counter the problem is multi-core processors[2].

Today it's very common with a multi-core processors in most computers, laptops, work stations, servers, phones and tablets. Adding just the cores does not means high performance, in some cases it can actually mean worse performance because the individual cores may not have as high clock speed as a single core processor[3]. It's is absolutely vital that the programs running on the hardware utilize the cores that the processor offer in order to get the performance boost. A multithreaded piece of software can run on a single-core processor if the processor supports context-switching but that won't be the focus of this paper, all testing environments will have a multi-core processor.

Developing software that can take advantage of multiple cores can be done in many different programming languages. If the languages does not have a standard library that supports threads there is often a third party library that can be added to allow usage of threads. In this paper C will be used as main languages during the experiments. First reason for choosing C is that C is and has been a very popular programming language for a long time and is used in many different areas of development[4]. Second reason for using C is that on Linux with GCC compiler it has support for the two libraries i chose to investigate without having to download any external libraries. The threading techniques the will be examined and tested in the paper is POSIX Threads and OpenMP.

1.2 Technology

1.2.1 C

C is a general-purpose programming language developed in the 1970's by Dennis M. Ritchie. Ever since it's development C has had a central role in the UNIX environment, it started with becoming the main programming language when the UNIX operating system was re-written[8]. The language is still used today, we can just look at the Linux kernel which has around ~97% of it's code base in C[7].

C has many features that appeals to programmers, some of them are:

- C offers low-level features. It is possible to program Assembly in C and access low level features of the computer[9].
- C is portable, if not using any OS specific libraries C should work on any computer with no modification required[9].
- It offers different datatypes such as char, int, long, float, double etc.[10].
- C is procedural i.e it has support for functions[9].
- C has support for dynamic memory allocation and de-allocation using pointers.

1.2.2 Threads

A thread is a way to make a program execute two or more things in parallel. A thread consist of it's own program counter, it's own stack and a copy of the registers of the CPU, but it shares other things like the code that is executing, heap and some data structures. The main goal with threads is to enhance the program performance and help to efficiently structure a program to performance multiple task at the same time[11].

An example is a web server.

If we have a web server that is not multithreaded and is listening on a certain address a port for a connection. When a connection from a client is detected the web server starts to service the request. When the web server is handling the request the web server will be blocked. This means that if another clients tries to connect to the web server he will either be refused or he has to wait, none of the alternatives is something to aim for even if the wait time is small.

Instead if the web server was multithreaded what would happen is, when a request is detected the web server would immediately spin up a new thread and han the request to the thread. Now the thread can handle the request and the server can go back to listen for new connections. By using multithreading we have eliminated the wait time for clients.

1.2.2.1 Alternative to threads

An important note here is that a comparison can not be made directly between thread and processes because a process contains one or multiple threads. But what can be done instead of creating threads and making parts of the program run on them a new process can be spawned instead. Then the new process could spawn new processes or threads. A big difference though is that a process can have multiple threads, not the other way around.

Both threads and processes are independent sequence of executions of the program. The process has it's own PID and it's own memory space, meaning it does not share anything unless told so via specific interfaces. Processes are supported by the UNIX operating system and is included in the standard library for C. To spawn a new process in C the function fork is used. When fork is invoked it spawns the new process with a exact copy of the program the parent process is running. 0 is returned to the new process but the to the parent the new process PID is returned from fork. This makes it possible to differentiate the behavior of the program depending of which process executing it, the parent or a child.

1.2.3 POSIX Threads

POSIX Threads is a common threading model when working in C or C++. It is a standardized model to work with threads and was created by IEEE. There has been other threading models available but none has been as well defined as POSIX Threads and that's mainly why it's so popular today. POSIX Threads can be use to parallelize task of a program in order to increase the performance of a program. The POSIX standard states that threads must share a number of things, for example the threads must share:

- Process ID
- Parant Process ID
- Open FD's (File Descriptors)

There are a some additional ones but it's not necessary, the goal was to point out that the standard states that some rules the must be fulfilled.

POSIX Threads offers a header file and a library the the programmer includes at compilation time of his program. POSIX Threads offers a lot of functions to create, manipulate, synchronize and end threads in a program. POSIX Threads is build with functions in mind and to start threads using POSIX Threads a function is fed to the function pthreads_create. The function that was fed to pthreads_create is now the entry point for the newly created thread and where it will begin it's execution[5].

1.2.4 OpenMP

OpenMP is a model for parallel programming in shared-memory system and was developed in the 1990's by SGI. OpenMP is available for three different programming languages, Fortran, C and C++. OpenMP consist of a number of compiler directives that can be used in the source code[6]. These directives need some functions to run and those functions is specified in a library that must be included at compile time. So how it works is that the directives in the source codes describe which functions should be run from the library, then the compiler knows it needs to call these functions from the OpenMP library in order to make it work.

That means that the compiler must have support for OpenMP, if it does not have support for OpenMP it wont know what to do with the directives. This wont crash the application, that is a sub goal of OpenMP. It should be portable between different systems, even systems that does not support it. Of course these system wont have the increased performance from parallel computing but they should still be able to run the program[12].

1.3 Research focus

The main focus for the research is the two models under investigation, mainly the performance they offer against the effort required to create a program with them. The interesting here is that the two models take a very different approach to allow parallel computing. POSIX threads acts as and extension of the language that uses the library while OpenMP depend of that the compiler has support for it. Either way, parallelism is achieved and the important aspect is if there is any performance differences.

The idea is to test this strictly scientifically on a number of common algorithms, Dijkstras algorithm, Quicksort, Matrix multiplication. There will not be test on more real life example such as web server etc. The whole point is to see how the models differentiate in performance, not test what they can be used to build.

Another aspect that will be investigated is the amount of effort that is required to be able to achieve the increased performance using the models. The effort will be the amount of lines required, reason for using the number of lines as measurement unit is that it gives a more accurate result because i have more experience with POSIX Threads going into this project than with OpenMP so using time would not give and accurate estimate.

1.4 Problem to solve

Today multithreaded computers are everywhere so it's important that the software utilizes this. POSIX threads and OpenMP has been around for some time and are both well tested techniques to parallel computing. But POSIX Threads is more common than OpenMP, so is there a reason for that?
I want to investigate the models and the performance that can be achieved with them.

This information can hopefully help developers or companies when they set out to build a system. When they design their system and realize that they are in need of parallelism they should do some research in how different models perform, if they scale good and if they are easy to work with and maintain. I want to help here with important information on how POSIX Threads and OpenMP perform in comparison to one another.

This can hopefully help developers and companies make the right decision early in the development process where it easy and cheap to make changes.

1.5 Method

To investigate the performance of the models a series of test will be performed on common algorithms that can be found in many different applications. Testing algorithms:

- Quicksort
- Matrix Multiplication
- Dijkstras Algorithm

For each of the three algorithms three programs will be written. One sequential, and two using parallelism. The sequential one is used as a point of reference to measure how much the models increase the performance from a sequential execution. Each of the program will be developed in C programming language.

So measures will be taken on the sequential version of the programs, then on the POSIX version and OpenMP version. Many measures will be taken on different input set to se how the parallel version scale depending on the input set's size. Then a comparison between the models will be made to be able to make a prediction on which models perform best under what circumstances.

The last thing that will be taken into an account is the implementation effort. Both the POSIX version and OpenMP version of the programs will be paralleling the sequential version of the algorithms and the amount of effort that require is the final measurement. The number of lines required to achieve the required parallelism will be counted and taken into an account when looking at how the models perform and if the should be recommended.

1.6 Expected Outcome

1.7 Thesis Overview

2 Related Work

asd

3 Method

asd

4 Results

asd

5 Analysis

asd

6 Conclusion and Future Work

asd

References

1. Moore's law: the future of Si microelectronics, Scott E. Thompson, Srivatsan Parthasarathy, June 2006 http://ac.els-cdn.com/S1369702106715395/1-s2.0-S1369702106715395-main.pdf?_tid=2c23d742-dee4-11e5-a061-00000aab0f6b&acdnat=1456750907_9c115ce84ccc9526d08983102e58c78c
2. Balaji Venu. Multi-core processors - An overview. http://arxiv.org/pdf/1110.3535.pdf
3. The Free Lunch is Over, A Fundamental turn towards Concurrency in Software, Herb Sutter http://mondrian.die.udec.cl/~mmedina/Clases/ProgPar/Sutter%20-%20The%20Free%20Lunch%20is%20Over.pdf
4. Tiobe Software. 2016. Top programming language of 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
5. Linux Manual. 2015. PThreads, POSIX Threads Manual. http://man7.org/linux/man-pages/man7/pthreads.7.html
6. OpenMP Architecture Review Board. 2013. OpenMP API Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
7. torvalds/linux https://github.com/torvalds/linux
8. History of C Programming language, 2014, Myriam Grandchamp, Saraswathi Kaja, Sourav Chandra, Oscar Serrano Serrano http://www.peoi.org/Courses/Coursesen/cprog/fram1.html
9. Features of C Programming Language www.c4learn.com/c-programming/c-features
10. C - Data Type, Tutorialspoint www.tutorialspoint.com/cprogramming/c_data_types.html
11. PThreads Programming: A POSIX Standard for Better Multiprocessing, Bradford Nichols,Dick Buttlar,Jacqueline Farrell, 1996, pages.1-26 https://books.google.se/books?hl=sv&lr=&id=oMtCFSnvwmoC&oi=fnd&pg=PR5&dq=Pthreads&ots=QNaTaoulU7&sig=PgVf_phIX3gxNo7OtBpvowCqVyE&redir_esc=y#v=onepage&q&f=false
12. Parallel Programming in OpenMP, Rohit Chandra Leonardo Dagum Dave Kohr Dror Maydan Jeff McDonald Ramesh Menon, 2001, ISBN 1-55860-671-8