Available online at www.sciencedirect.com

SciVerse ScienceDirect

Systems Engineering Procedia 1 (2011) 457–464

Procedia
Systems Engineering

Meta-Heuristics in Multi-Core Environments

Ragnar Magnús Ragnarsson[a], Hlynur Stefánsson[a], Eyjólfur Ingi Ásgeirsson[a*]

[a]*Reykjavík University, Menntavegur 1, 101 Reykjavík, Iceland*

Abstract

CPU manufactures have been adding more cores to CPU's instead of only focusing on increasing the speed. There are many interesting questions regarding implementation and performance of algorithms using the new CPU's. We look at three well known meta-heuristic algorithms with different ratios between independent calculations and shared memory usage and analyze the benefits of using multi-core CPU's compared to single core CPU's. By programming specifically for multi-core processors, the performance of meta-heuristic algorithms can be improved without much effort, but the improvement is heavily dependent on the system architecture and the ratio between independent calculations and shared memory usage.

Keywords: Meta-heuristics, Multi-core architecture, Parallel implementation.

1. Introduction

Many real life problems are large NP-hard problems and thus for nontrivial instances it is almost impossible to find the best solution in a reasonable amount of time. Meta-heuristic algorithms are often used to get a good solution in a decent amount of time but they do not guarantee an optimal solution. Meta-heuristic algorithms have become ever more popular during the recent years in part due to the computer revolution [1].

Meta-heuristic algorithms are fairly easy to implement in a single core environment, i.e. one CPU (Central Processing Unit) using integrated cache memory and random access memory (RAM). The cache memory is integrated in the CPU and is faster than the RAM memory which is connected to the CPU through a bus (a high speed channel) which is much slower than the CPU. Though meta-heuristic algorithms are easy to implement on single core computers they often run rather slow, especially on personal computers, though modern PCs are more powerful than many supercomputers through the years. To get better performance more focus has been put into parallel computing. Parallel computing is where a problem is broken up into smaller entities and two or more entities are being worked on at the same time. In parallel computing there are two prevailing inter-processor communication methods; shared and distributed memory models [2]. In shared memory systems a collection of homogenous processors share the same main memory through busses (or crossbar for more than four processors). Symmetric multi-processors, SMP, are an example of a shared memory model and many consider that chip-level multi core processors belong to this model. Distributed memory systems are on the other hand comprised of more than one heterogeneous stand-alone machines, each with its own central processor unit and memory set, connected together through a high speed network. Good examples of distributed memory systems are Beowulf clusters [3].

* Corresponding author. Tel.: +354-599-6385; fax: +354-599-6201.
E-mail address: eyjo@ru.is

The two main APIs (Application Programming Interfaces) for the shared memory model are POSIX threads [4] and OpenMP [5], while for the distributed model the MPI (Message Passing Interface) is by far the most commonly used [6]. Both the shared memory and the distributed memory models have their advantages and disadvantages. When porting a sequential code to a shared memory model, parallelizable code needs to be identified and written again in APIs such as OpenMP. This can introduce complications such as race conditions, deadlocks or other problems often accompanied with shared memory models. It is even more complicated to write code for distributed memory models, since it usually involves writing efficient algorithms to divide tasks among processor and memory sets and then synchronizing the results. The distributed model is often limited by slow network speeds and thus each task needs to be large enough to overcome the network latency. On the other hand, distributed systems are more scalable; adding more processors to such a system often simply involves adding another node (computer) to the current environment whereas doing so on shared memory systems increases the bus traffic and slows down memory access. In some cases it even requires expensive and complex hardware changes maybe involving investing in a completely new system with room for more processors.

In 1965 Gordon Moore wrote an article for Electronics magazine titled "Cramming more components onto integrated circuits" [7] where he predicted, often called Moore's Law, that the number of transistors on a chip would double each year into the foreseeing future. His prediction has, more or less, come true over the years. Around the year 2005 chip makers were having trouble with increasing the performance of chips; the small size of the transistors were causing heat and power issues. The answer was adding more cores on the same die, which gives a significantly better performance while only increasing power usage by a small percentage. This essentially threw the problem of ever increasing speed on to the software engineers but so far not many programmers have been utilizing this technology to any extent. In recent years, tools and support for parallel programming have been added to many software languages, such as Java, C++ and C#, giving programmers better opportunities to use the full power of the modern CPU without spending too much time in training and/or refactoring [8][9]. Since the burden of increasing speed is on software vendors, operating systems have to be scalable and use multi core technologies, today many operating systems are not optimized for multi core CPUs which can have an effect when writing multi core algorithm implementations [10].

There are not many meta-heuristics papers focusing on multi core architectures. Bui, Nguyen and Rizzo Jr. [11] showed results where the running times of an Ant Colony algorithm improved by around 40% with a dual core system and by a factor of up to 6 on a eight core system. In this paper we will take a look at how to increase performance with multi core architectures and try to identify potential pitfalls along the way.

1.1. Algorithm Selection and Implementation

We selected three different meta-heuristic algorithms along with three different problems. Our selection criteria was to select algorithms with different ratios of independent calculations to shared memory usage, to be able to analyze the effect of this ratio on the possible improvement gained by using multi core architecture instead of a single core implementation. Simulated Annealing with random restarts was selected mainly because the lack of shared memory usage while both Ant Colony and Tabu Search have features, such as shared memory structures, that are interesting to analyze in multi core environments. Ant Colony uses shared memory but the running time of the algorithm is still dominated by independent calculations while Tabu Search uses shared memory constantly during its search.

Although multi core processors have been mainstream for quite some time, not many software developers are programming to use their full power. In recent years companies like Intel, Microsoft and the Java open community have been creating tools to make it easier for the everyday programmer to start programming for the multi core CPUs [12][13][14].

The .NET4 platform from Microsoft comes with a set of tools for parallel programming, in this paper the .NET4 platform is used to program both the single- and multi core implementations for all the algorithms.

2. Experiments

To analyze the effect of the number of cores and the effect of system two computers were used, computer A and computer B. Computer A runs a dual core AMD CPU and computer B runs a quad core Intel CPU. Computer B is