Czech University of Life Science Prague

Faculty of Economics and Management

Department of Information Engineering

Diploma Thesis

Parallelism in Computer Architecture

Author: **Jan Rúčka**

Supervisor: **Ing. David Buchtela, Ph. D.**

© 2012 CULS in Prague

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Department of Information Engineering

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Rúčka Jan

Informatics

Thesis title

Parallelism in Computer Architecture

Objectives of thesis

The aim of thesis is an analysis of parallelism.In the practical part there will be a practical usage,meaning of parallel methods compared with classic methods.

Methodology

The metodology of the thesis is based on the study and analysis of scientific information sources.The practical part is focused on comparison classic and parallel methods of programming. Based on the synthesis of theoretical findings and results of the practical part, the diploma thesis conclusions will be formulated.

Harmonogram zpracování

06/2011-08/2011 Detailed specification of the goals and development of appropriate solving steps
09/2011 Result inspection and 1st credit
09/2011-12/2011 Analysis of scientific information sources and elaboration of the recherche of the topic
01/2012 Result inspection and 2nd credit
01/2012-02/2012 Elaboration of practical part of the thesis
03/2012 Finishing the thesis and submition, 3rd credit

The proposed extent of the thesis

60 - 80 stran

Keywords

software, parallelism, parallel principles, programming

Doporučené zdroje informací

Fountain T.J.: Parallel Computing, Cambridge University Press, United Kingdom, 2006, ISBN 978-05-210-3189-9

The Diploma Thesis Supervisor

Buchtela David, Ing., Ph.D.

Deadline of the diploma thesis submission

březen 2012

prof. Ing. Ivan Vrana, DrSc.	**prof. Ing. Jan Hron, DrSc., dr.h.c.**
Head of the Department	Dean

Prague June 29. 2011

Declaration

I hereby declare that I have worked on my Diploma Thesis titled “Parallelism in Computer Architecture” solely and completely on my own and that I have marked all quotations in the text. The literature and other material which I have used are mentioned in the Bibliography Section of the Thesis

In Prague 5th April 2012 ______________________________

Acknowledgement

I would like to use this opportunity and thank Ing. David Buchtela, Ph. D. for his valuable comments and remarks.

Parallelism in Computer Architecture

Summary

This diploma thesis deals with parallelism above all in personal computers in view of microprocessor and operating system. At the first part are defined important theoretical background about hardware and software where is basic theory about parallel programing. The second part of the work deals with parallelism practically. Firstly there is a simple algorithm where are compared different parallel programing ways. Then there is a sample GUI (Graphical User Interface) application which explains advantages and disadvantages of parallelism.

Keywords: parallelism, parallel algorithm, parallel programming, hyper-threading, multi-core microprocessor, multi-threading, multi-processing, chess algorithm

Paralelismus v architektuře počítače

Souhrn

Tato diplomová práce se zabývá paralelismem především v osobních počítačích z hlediska procesoru a operačního systému. V první části jsou definovány důležité teoretické základy o hardwaru a softwaru, kde je základní teorie o paralelním programování. Druhá část se zabývá paralelismem v praxi. Nejprve jednoduchá aplikace kde jsou porovnány různé paralelní přístupy. Poté ukázková aplikace s GUI (grafické uživatelské rozhraní) na které jsou popsány výhody a nevýhody paralelismu.

Klíčová slova: paralelismus, paralelní algoritmus, paralelní programování, hyper-threading, vícejádrový procesor, multi-threading, multi-processing, šachový algoritmus

Contents:

1 Introduction ... 4
2 Objectives of Thesis and Methodology ... 5
3 Parallelism and Principles ... 6
3.1 Definition of Parallelism ... 6
3.1.1 Multitasking ... 6
3.1.2 Hyper-Threading ... 7
3.1.3 Multicore ... 10
3.2 Using of Parallelism ... 12
3.2.1 Algorithmic Complexity ... 13
3.2.2 Amdahl´s Law ... 14
3.2.3 Practical Use ... 16
3.3 Parallel Programming ... 17
3.3.1 Parallel Algorithms Problems ... 17
3.3.2 Low-Level Parallel Programming ... 18
3.3.3 High-Level Parallel Programming ... 19
4 Practical Usage, Meaning and Comparison of Parallelism ... 21
4.1 Matrix Multiplication ... 21
4.1.1 Methodology ... 21
4.1.2 Single-Threaded C++ ... 22
4.1.3 Multi-Threaded OpenMP ... 23
4.1.4 Multi-Threaded POSIX Thread ... 26
4.1.5 Multi-Threaded Windows Threading ... 29
4.1.6 Multi-Threaded Qt Framework ... 31
4.1.7 Single-Threaded Java ... 33
4.1.8 Multi-Threaded Java ... 35
4.1.9 Compare and Evaluation of Different Implementation ... 37
4.1.10 Disadvantage of Hyper-Threading ... 39
4.2 Chess Application ... 41
4.2.1 Methodology ... 41

4.2.2 Single-threaded Chess Application Using Minimax 42
4.2.3 Multi-threaded Chess Application Using Minimax 49
4.2.4 Multi-threaded AI Chess Application Using Minimax 52
4.2.5 Multi-threaded Chess Application Using Alfabeta 57
4.2.6 Multi-threaded AI Chess Application Using Alfabeta 59
4.2.7 Chess Application Conclusion 61
5 Conclusion 62
6 Bibliography 63
7 Supplements 66
7.1 List of images 66
7.2 List of tables 66
7.3 List of source codes 66
7.4 Attached Files 67
7.5 Complete Results of Matrix Multiplication 68

1 Introduction

Parallelism in computer architecture is used more and more thanks to more modern technologies of production. While earlier was parallelism in form of multicore or multiprocessor systems domain of servers, mainframes or workstations, today is standard even at personal computers.

Parallel programming is more important with using multicore processors at personal computers. Parallel programming usually increases complexity but programs can be several-times quicker and have also other advantages. Meaning of parallelism is getting more important and we can expect that it will appear more and more frequently.

Despite the parallelism today is not too much broadly used, author analyses how can be parallelism used and writes a necessary theoretical background. Author also explains different ways of parallelism on a simple example and then contributions on a sample application.

2 Objectives of Thesis and Methodology

The aim of this diploma thesis is to define what parallelism is. It explains a necessary theoretical background for parallelism. The first part of the background is about importance of hardware features. Afterwards it explains fundamental theory relevant to parallel programming as is Amdahl´s law and parallel programming theoretical ways.

The second part starts with parallel programming of a simple algorithm in several different ways. Specifically it is a matrix multiplication with the standard (trivial) algorithm and evaluates contributions of parallelism (especially speed-up) and differences between implementations. Then there is a sample application with GUI (Graphical User Interface) where are shown contributions of parallelism. The first contribution is responsible interface during intensive computing. The second is the same as in the matrix multiplication part – speed-up. Also there are mentioned disadvantages of parallelism as is difficulty with debugging and to ensuring that application is without errors. Then it mentions importance of a good design and comparing parallelism with other possibilities.

For practical part – the matrix multiplication author has chosen to compare these possibilities of parallelism: OpenMP, POSIX Thread, Windows Threading, Qt framework and Java multi-threading. For the sample application – a chess algorithm is chosen C++ language with Qt framework which is used for multi-threading together with OpenMP standard.

3 Parallelism and Principles

This part of work discusses a theoretical background of parallelism, important elements of hardware and software architecture which allows parallelism. It looks at the basic theoretical background of algorithm and analyse ways of the parallel programming which we divide into the low-level and the high-level.

3.1 Definition of Parallelism

In Personal computer architecture there are a lot of examples of parallelism. In personal computer is a parallel transfer nearly between every part of computer, almost massive parallelism in graphic cards. There are multitasking, which can be described as a virtual or an apparent parallelism, pipelining and superscalar architecture. But the work discusses above all multicore (or in this meaning almost the same – multiprocessor) systems and hyper-threading which can be seen as a multicore system.

3.1.1 Multitasking

From the users point of view there is actually no difference between single-core and multi-core systems. For users it is difference between single-task (DOS-like) and multitasking systems (typically Windows). With single and multi-core systems users can do the same amount of work; system behaviour is in most cases the same. Multitasking is not only operation system matter. Without support of microprocessors is not possible to interrupt process in work, do some another work and return back without damage of data in process. Process saves into TSS (Task State Segment). This is coming with the processor Intel 80286. The processor Intel 80386 is improving these properties; principle remains the same until today. [1] [2]

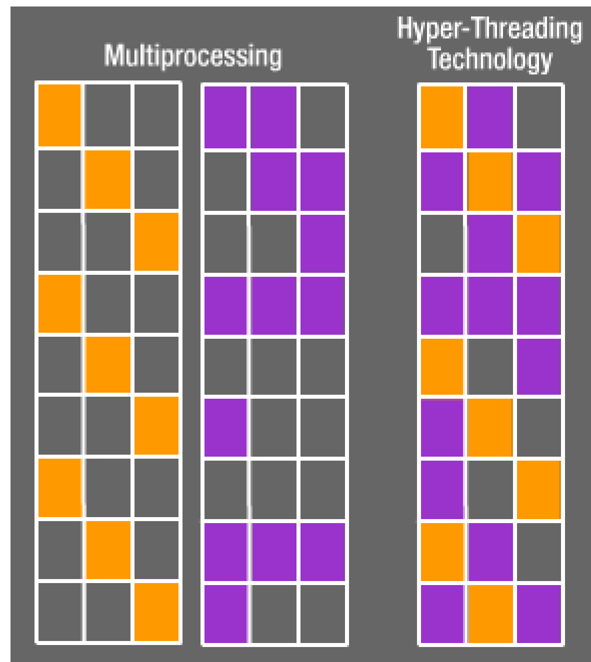
These properties with support of operation system make possible the illusion of doing multiple tasks simultaneously. This is achieved by switching between the running processes many times per second. But this is (on single-core system) not true, progress can do only single task in one moment. With multi-core system there are more processes but also only small finite number in comparison with the illusion of system.

In multitasking system processes are running under protected mode. This should prevent from “freezed” system and prevent from unauthorized access to computer peripherals and resources. These properties have all modern operating system running on x86 (or x64) architecture as Windows NT, Unix-like, MAC OS and so on.

3.1.2 Hyper-Threading

In personal computer this is quite new technology. Firstly introduced with Intel Pentium 4 Processor 3.06GHz core Northwood in year 2002. But in the fact Hyper-Threading (HT) is actually not a new technology it is only Intel implementation of Simultaneous multithreading (SMT). Simultaneous multithreading was firstly researched by IBM in 1968. Simultaneous multithreading or Hyper-Threading allows processor (core of processor) to execute more than one thread in case of Hyper-Threading exactly two. In the case of another Simultaneous multithreading even more – good example is SPARC architecture. This can be a little bit confusing because there are virtual CPU´s. The computer has dual-core processor but thanks Hyper-Threading technology it seems as it has quad-core processor because of two virtual CPUs. [3] [4]

The idea of Hyper-Threading is actually pretty easy processor seldom uses all its resources. If the same amounts of resources are allocated into two (or more) threads, processor probably uses more resources and does more work. Hence processor (in this example processor with Hyper-Threading) is able to execute more instructions and even finish more instructions in one moment because occurrence of so-called bubbles, – places where are not used all resources. Reason for that is typically a data dependency. It can be seen in the *Picture 3.1*.



Picture 3.1 – Ideal function of Hyper-Threading technology [5]

According Intel there is 15 to 30% better performance and first chips had about 5% more transistors to enable this technology. But of course 30% is probably only if there are the ideal or almost the ideal conditions. Reason why is rarely achieved these performances is that processor have some so-called bottleneck. This is place where is not possible to do work simultaneously, in this case is advantage of Hyper-Threading zero. From these statement is quite clear that the best results are achieved especially when two thread use different processor´s resources. [6]

Multiple hardware threads are also particular solution to the problem of memory latency. When one thread has to wait for data returning from a slow memory (RAM i.e.), the other thread can still make some progress. This can improve the utilization of processor (or core) and improve performance. Hyper-Threading can be used as one of the technique particularly solving the memory latency problem, such as for example out-of-order execution and so on.

Picture 3.2 - Ideal function of Hyper-Threading technology [7]

It may seem that Hyper-Threading has only advantages but that is not true. In some cases if threads are using similar resources as big amount of cache memory, performance could be worse. In some cases some operation systems use virtual CPU (Central Processing Unit) completely the same as real CPU (single-core CPU with Hyper-Threading as dual-core CPU). Hence if in system are two threads (two programs), one with the critical importance with high or real-time priority and another with the low importance with low or idle priority, the operation system can think that there are enough resources to execute both threads it the same time. But the real operation system has only single-core processor with Hyper-Threading technology, hence the operation system may do more work in total, but the critical thread (program) is slowed-down. This is probably the biggest disadvantage of Hyper-Threading. In the ideal case there is let say 130% performance of single-core processor without Hyper-Threading, but for the critical thread (program) is available only half of it: 65% which is actually about 35% worse performance. In the worst case it is 100% performance with comparison with processor without Hyper-Threading and for the critical thread (program) only 50% of performance. This is shown in the practical part in next chapters. That is operations system matter and in fact some of operation systems should be able to

handle it for example Solaris which typically uses SPARC architecture processor which can have up to eight threads per core. [8] [9]

3.1.3 Multicore

The first multicore processor for personal computer was Intel Pentium eXtreme Edition 840 in 2005. This was actually the first time when personal computer could fully execute two threads. In case of workstations or servers there were multiprocessors systems long times ago. It is possible to have one processor which may be seem as two (or more) and actually in practise there is not serious difference in performance.

The first processor x86 architecture was released in 1978 and had 5MHz. Today´s processors have about 3GHz what is only in the processor clock speed 600 times increase. Today´s processors have about 3GHz quite long times, in compare with the past. Hence the multicore is another technique how to increase performance on the same frequency. Reason is that it has become harder and harder to improve serial performance. It would be necessary use larger area of silicon to enable executes instructions faster. Also there is problem with amount of power consumption and heat generation.

Picture 3.3 – Single and multicore processors in compare without HeatSpreader [10]

In most of the cases there are not any important differences between multiprocessor and multicore system. In both of it we can execute two (or more) threads at the same time. That provides us theoretically two times more performance. In operation system multicore processor is may seem as multiprocessor system. It is possible to talk about virtual CPU (Central Processing Unit).

There are several differences between multiprocessor and multicore solution. Multicore processors have more shared resources. Typically L2 cache memory (or sometimes also L3 cache memory) transfer rate between processor and rest of the system is the same doesn´t matter how much cores it has. These disadvantages are not in common too critical and don´t slow-down system too much. The advantage is for

example faster communication between cores than between processors. Threads can be synchronized more often without significance loss of performance. Hence it can be profitable to paralyze programs which were not profitable before.

Multicore processor needs almost two times bigger area of silicon which means almost two times bigger price. Multicore is not the cheap way to enhance performance. The biggest disadvantage is that if software is not optimized there is practically not any more performance. Hence today is parallel programming much more important than in the past. [11]

3.2 Using of Parallelism

The first of all it is necessary to define used terms. The first term is thread at which we can think a software thread or a hardware thread. The software thread is some stream of instructions that the processor executes. We can have two threads in one application or in two different applications. The hardware thread is some resource. Operation system has typically many of software threads and only few hardware threads.

Every application has at least one process; every process has at least one thread. Difference between process and thread is following. Main difference is that the process has a state. The state is set of values, which processor has in memory. These values are for example addresses of the currently executing instructions and translation lookaside buffer (TLB). The state uniquely defines the process in time. It is possible to say that multiple applications are just multiple processes. One process is totally independent from other processes what is not true for threads. Each process can run multiple threads. Threads have also some state as processes do, but threads state is just values in register and data in stack.

There are many reasons why programme multiplies threads. Today the main reason is speed; an application with multiple threads could be multiple faster. Another reason could be naturally decomposed, if there is an application executes different mutually independent tasks, which can be divided into multiple threads. Sometimes it can be good to have multiple threads if we want to keep programs staying responsive during intensive computation. If in GUI (Graphical User Interface) starts some long calculation in single threaded program, whole application is “freezed” until the end of the

calculation. If there are two threads one for GUI and another for calculation, program is still responsible. It is not "free" but at the cost of a little slowed-down the calculation (in case of single processor system). This is shown in practical parts in next chapters.

Multiple processes have higher cost of sharing data between them; hence they are typically used for another reason. Processes are mutually independent. This means that each process request its own state, increasing the memory and computing cost. But because processes are independent it is possible to build much more stable applications. If one process dies, does not matter why, other processes can continue. In case of the multiple-threaded application, if one thread fails, probably the entire application fails. Good example for the multiple-processed program is recent change in web browser design. Google´s Chrome browser has for each tab its own process. Hence if one tab fails rest of the browser continues without problem, which is in comparison with other browsers the interest advantage. Because of unconstrained and unpredictable nature of the internet it seems as a good design decision. [9]

3.2.1 Algorithmic Complexity

First of all it is necessary to measure input data. After that it's needed to estimate time complexity. That is number of steps in algorithm. It is not important to be accurate it is fully sufficient rough measures. Steps are for example one operation as +, -, *, /, condition evaluate and so on. It is possible to assume every step is the same despite that every operation needs another time to execute and it is even different with different processors, it is useless try to be accurate.

	n = 10	n = 100	n = 1000	n = 1 000 000
log n	3.3ns	6.7ns	10ns	20ns
√n	3.2ns	10ns	31.6ns	1µs
n	10ns	100ns	1µs	1ms
n*log*n	33ns	664ns	9.9µs	20ms
n^2	100ns	10µs	1ms	16.5 min
n^3	1µs	1ms	1s	31 years
2^n	1µs	$3*10^{14}$years	$3*10^{286}$years	≈∞
n!	3ms	$3*10^{142}$years	≈∞	≈∞

Table 3.1 – Time complexity of different algorithm with presumption 10^9 execute operations for second [12]

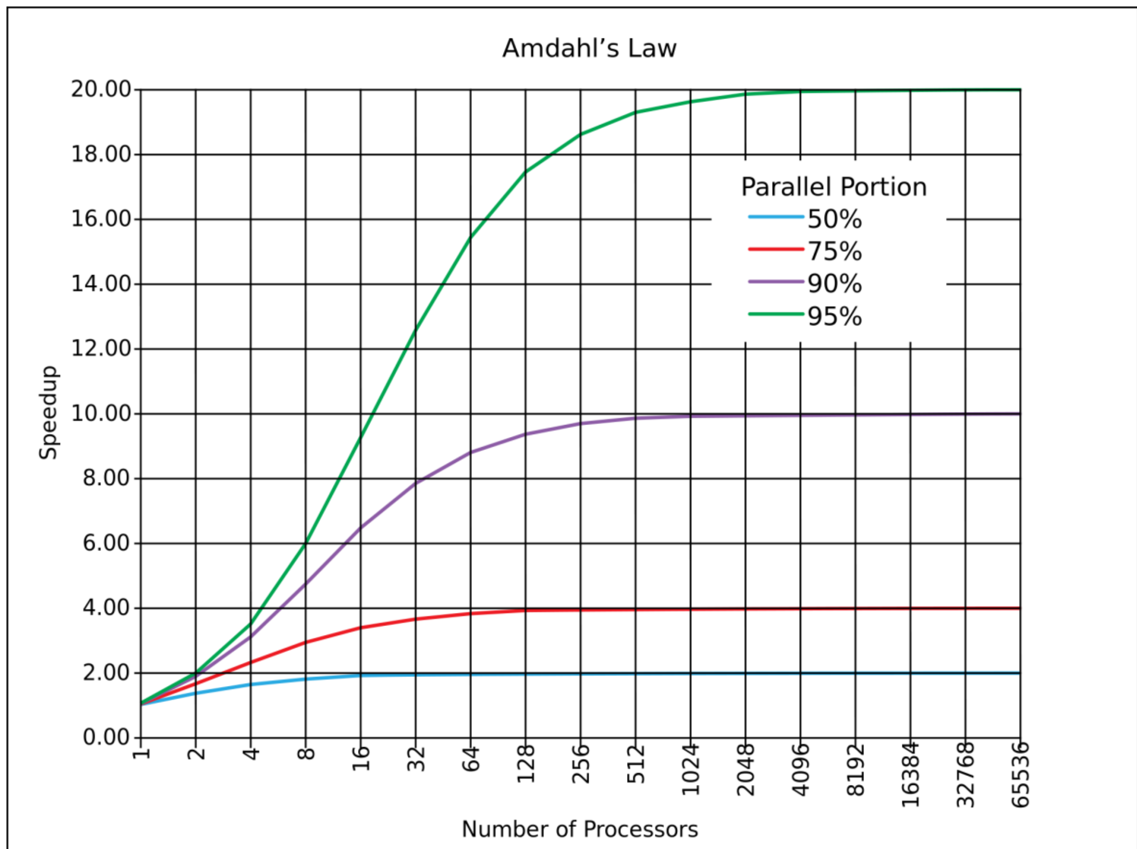
In the *Table 3.1 - Time complexity of different algorithm with presumption 10^9 execute operations for second*, it is possible to see how long it takes algorithm according to the number of elements and time complexity. From this it can be seen constant is not too important. Hence it is possible to commute for example multiplication and sum. In the *Table 3.1* it is clear that it does not matter if there is n^2 or $5n^2$ or $30n^2$, there is no difference. All of these will increase almost with the same speed. Hence if we have two computers and one of them has twice performance it means that a certain algorithm takes half of the time. It is quite clear but in case of polynomial complexity it can happen that is not possible to compute even with one more input in the same time.

Hence if it is necessary to compute algorithm with one more input at the same time, sometimes it is quite easy. Sometimes it can help parallelism but sometimes we are not able to do it with any optimization.[12]

3.2.2 Amdahl´s Law

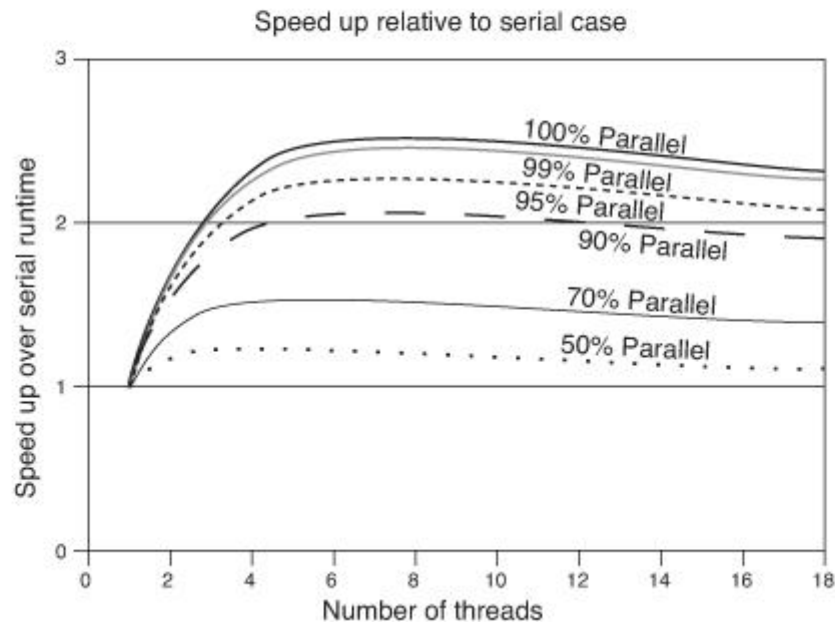
Amdahl´s Law basically says that it is possible to divide algorithm into two parts. One part which is possible parallelized and another part which is not possible parallelized. If the part which is possible parallelized is very small, it is not possible to have a good result with parallelism.

The speed-up of algorithms with using of multiple threads (assuming we have it) is limited by the time which is needed for a serial part. If the serial part takes 20% from the total time and the parallel part 80%, then if it are used two threads, there is 20% + 80% / 2 = 60% of the not optimized time. In case of eight threads it is 20% + 80% / 8 = 30% of the not optimized time. If there is used really big amount of threads the result is not so different because the serial part. In the example 1000 threads: 20 % + 80% / 1000 = 20.08% of the not optimized time. Hence we cannot speed-up under the serial time which can be very significant.



Picture 3.4 – Scaling with different parallel portion according Amdahl´s Law [13]

It is necessary to think over how many threads are suitable to use. If another thread speed-up application insignificantly question is if it is suitable. Used resources in a place where is not meaningful effect can cause that in other places could resources miss. Amdahl´s Law also don´t count with synchronization costs and so on. Hence in reality we can expect more likely the result as is in the *Picture 3.5 - Scaling with different parallel portion in real* than in the *Picture 3.4 – Scaling with different parallel portion according Amdahl´s Law* . [9]



Picture 3.5 - Scaling with different parallel portion in real [14]

3.2.3 Practical Use

There were mentioned a lot of things but what is important is influence into practice. Sometimes it is better don´t do anything about optimization, sometime no optimizations the best way. In some algorithms it does not matter if takes one second or few days. Typically some programs which are used only once and time use with optimization is useless because it is not important to have the result immediately. Good programmer has to know what he wants from programs and according this to decide what it is the best.

On the other side we have applications which need infinite of performance. Typically some models which are simplified because we do not have enough performance. In this case parallel algorithms allow smaller simplified. Parallel algorithms allow using programs on personal computers which were few years ago needed on servers or workstations.

If we use parallelism optimization is one of the key factors. If we ignore optimization there is actually no reason for parallelism. Performance which we obtain by using parallelism we lost because of no usage of optimization. Another thing is efficiency, how it is possible to see in the *Picture 3.5 - Scaling with different parallel portion in real* which is probably a little bit excessive. Every other processor adds less

and less performance. Hence in general with every another processor decreases efficiency of usage of resources.

Optimization has sense only with function which takes meaningful part of time. If it is possible to easily optimize and even parallelize function which takes only 1% of runtime of applications it is probably worthless. It should be optimized functions (and in general such pieces of code) which take meaningful part of runtime. In that case optimization and the parallelization will probably have good sense.

3.3 Parallel Programming

3.3.1 Parallel Algorithms Problems

Parallel algorithms are much more complicated than serial. It is obvious because what is done in serial code that is necessary to do also in parallel code. Every mistake, problem, optimization and so on remains. In addition there is more source code and more possible problems. Problem is also with debugging such applications. Parallel applications don´t have to behave deterministic.

Every time they can behave different depending on which thread is quicker, which thread acts first. This is why it is the problem debugs parallel applications. In serial applications if there is any problem, programmer can run through critical part step by step until he discovers the problem. In parallel application this is often not possible and if it is, it is much more complicated and time consuming.

For example following source code the *Source 3.1 – C++ source code where can occurs data race*, can behave nondeterministic. If there are more threads and every use this function and the "a" variable it is not clear what value will has the "a" variable at the end. We cannot be sure if the variable changes properly. So it is actually not possible to use this function in this way. Another problem is when two or more threads change value of the "a" variable at the same time. It can happen that the "a" variable changes only once. This problem is called data race.

```
void change(int *a)
{
	*a += 4;
}
```

Source 3.1 – C++ source code where can occurs data race

Data races can be hard to find. Previous example is clear but in a complicated program it is not obvious. The problem with finding these kinds of problems is that what it does the parallel programming hard. This kind of error can even occur only sometimes. Some software (for example Valgrind suite or Thread Analyzer in Oracle Solaris Studio) is able to identify data races but unfortunately not all. Avoiding data races is not too hard; it is just using synchronization lock around accesses to that variables.

There are lots of problems in the parallel programming. Some of these problems are often and harder, some of them seldom and easier. What it is important is that there are problems which we don´t find in the serial programming. Hence the parallel programming is harder and more time consuming. This is why we often parallelize only part of code which takes lots from runtime and rest of the application let to serial.

3.3.2 Low-Level Parallel Programming

The low-level parallel programming is most known technique. This technique can be represented with for example POSIX (Portable Operating System Interface) thread, Windows threading, but also with Java multithreading and with QThread class within Qt framework. All these standards are shown in the practical part in next chapters. Between the low-level parallelisms we can also place MPI (Message Passing Interface) and many others.

It is necessary to care about all problems which can happen in parallel programming. It is needed to create and terminate every thread. It is necessary to care about passing data between threads, synchronization, creating lock, sharing data and so on. In opposite in the high-level parallel programming, the programmer don´t have to care about all of these things.

POSIX is specifications for Unix-like operating system. POSIX handles programming interface and in general behave of operating system. Thanks to POSIX are programs portable between the difference versions of Unix-like operating systems. POSIX also ensures the way of the thread implementation. In every Unix-like system should be threading the same if keeps the POSIX standard. POSIX provide all necessary functions for multithreading. POSIX threads implementation exists also for another operation systems for example Pthreads is implementation for windows. [15] [16]

Windows threading is good alternative for POSIX in windows. In Windows API it exists with lots of other functionalities and possibilities also possibility to work with threads. Functionality is very similar with POSIX threads. Most of the functions have above all different names. Between Windows threading and POSIX threads is not very big difference.

Java is very robust programming language. Java contains many features and between them is supporting multithreading. The implementation is quite difference with compare of Windows threading and POSIX threads but basic idea is the same. Creating, terminating threads, using some lock for synchronization; the biggest different is that thread is as almost everything in Java an object. But in practice this is not too important. The main advantage is the same as advantage of Java, threading is also multiplatform.[17]

Qt is framework primary for programming GUI (Graphical User Interface) but it provides much more functionality. One of this functionality is also support of threads. Qt has actually two main classes for threads QThread and QtConcurrent. QThread provide the low-level API (Application Programming Interface) for threads which is in functionality similar as POSIX and windows threading. QtConcurrent in opposite provide the high-level API which is completely different. Because Qt is multiplatform it is possible to write code only once and run everywhere which is support way of portability of C++. [18] [19]

There are many different low-level threaded API, but most of them are quite similar, because necessary functionality is in general the same. Of ′ course it is only in the case of the same architecture (x86 in this case). For example MPI which is used in massive parallel architecture is very different.

3.3.3 High-Level Parallel Programming

The ideal situation of high-level parallel programming is that a programmer creates the serial application and compiler creates the multi-threaded application. This is actually possible. The best-known is probably Intel Compiler. But of′ course it is not ideal. This compiler is able to parallelize only something for example some kinds of loops. The compiler has to be sure that it is “safe” - result serial and parallel computing is the same. The programmer has to create loops in “safe” way or use another approaches. There is

probably doesn´t reach so good result but in the other side it is not necessary to care about it. It only takes a little bit bigger amount of time to compile. [20]

Intel has probably the best compiler for multithreading nowadays. Another way of multithreading is Cilk which is programming language designed for the multithreaded parallel computing. In this language programmer only define how should be his program parallelized. [21]

In the high-level approaches to parallelism it is not necessary to care about creating or terminating threads and sharing data. These mechanisms are solved by the compiler. For the practical part of this work was chosen OpenMP API (Application Programming Interface) specification for the parallel programming. This API provides sets of tools for defining how to create the parallel version of the serial code. It is much easier than the low-level parallel programming because in most cases is enough just defined parallel directives into the serial code. [22]

Qt framework also provides some kind of the high-level API for multithreaded through QtConcurrent namespace. It operates with help of functions which provide parallel way of usual problems. In another way there is parallelize code with help function and it is not necessary to care how it is done. With this it is similar as OpenMP. [23]

There are probably more approaches for the high-level parallel programming. Some lesser used way of the programming are probably more suitable for parallelize for example functional programming languages. But it is not possible to describe all of them. There are no others possibilities than to restrict only on few of them.

4 Practical Usage, Meaning and Comparison of Parallelism

In this part are shown two examples of the serial and the parallel programming and comparison of them. The first is the matrix multiplication which is implemented in the serial and many different styles of the parallel programming. After that it is compared. The second part is complete chess application including AI (Artificial intelligence) and GUI (Graphical User Interface). There is shown possibility of parallelism even for the single-threaded processor and contribution of parallelism in such application.

4.1 Matrix Multiplication

4.1.1 Methodology

For all programs in C++ programming language was used Microsoft Visual Studio 2010 Professional software development tool. For POSIX thread was used Pthreads implementation for MS Windows in version 2.8.0. [16] Qt framework was used under MS Visual studio in version 4.8.0. For Java programs was used Java SE 7u3 and Eclipse software development tool in version 3.7.2.

For testing are used four different computers. First has processor Intel® Core™ i7-2600 and uses MS Windows 7. This processor has four cores and support Hyper-Threading technology which means eight virtual processors. The second one has processor Intel® Core™2 Quad Processor Q9550 and uses MS Windows 7. This processor has four cores. Third has Intel® Pentium Processor P6200 and uses MS Windows 7. This processor has two cores. The last one has Intel® Pentium® 4 Processor 650 and uses MS Windows XP. This processor has only one core but supports Hyper-Threading technology which means two virtual processors.[24]

For performance test was used standard (trivial) multiplication algorithm. There are quicker algorithms but there are not suitable for this works purposes it goal is not the fastest program but comparison. Compared was time needed for compute matrix 1000*1000 with range <-100; 100>. This is executed 24 times and written into file. Two

best and worst values are discarded and from remaining 20 values is created an arithmetic mean. Programs are run with real-time priority. For this is used “start /realtime“ command.

After this comparison of all values together knowing what way is the fastest. Then there is proved the main disadvantage of Hyper-Threading technology, which is explained in the chapter *3.1.2 Hyper-Threading*.

4.1.2 Single-Threaded C++

First of all the matrix multiplication is shown in the serial way. From this implementation see the *Source 4.2 – Matrix multiplication C++ serial implementation* are derived other C++ implementations. It is very simple program. Vector is used because matrix 1000*1000 has one million elements which mean 4MB (4 byte for integer) and it is too much for stack. The “timeGetTime” is function for time measure with accuracy about one millisecond in Windows 7. In Windows XP the accuracy is a little bit less.

```
#include <iostream>
#include <fstream>
#include <Windows.h>
#include <vector>
#define SIZE 1000

int main() {

std::vector<std::vector<int> >MatrixA(SIZE, std::vector<int>(SIZE)),
       MatrixB(SIZE, std::vector<int>(SIZE)),
       MatrixC(SIZE, std::vector<int>(SIZE));

//loop  for more than one result
for(int x=0; x<24; x++)
{
       srand(timeGetTime()); // random number where seed = time

//Fill Matrix A and B with random number, result matrix C with 0
       for (int i=0; i<SIZE; i++)
       {
              for (int j=0; j<SIZE; j++)
              {
                     MatrixA[i][j] = rand() % 201 - 100;
                     MatrixB[i][j] = rand() % 201 - 100;
                     MatrixC[i][j] = 0;
              }
       }

       unsigned long time = timeGetTime(); //start time measure

//Matrix multiplication compute
```

```
        for (int i=0; i<SIZE; i++)
        {
                for (int j=0; j<SIZE; j++)
                {
                        for (int k=0; k<SIZE; k++)
                        MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
                }
        }

        //write result into file
        std::ofstream fout("output", std::ios::app);
        fout << timeGetTime()-time << std::endl;
        fout.close();
}
return 0;
}
```

Source 4.2 – Matrix multiplication C++ serial implementation

The matrixes are defined as vectors. Then it starts the loop which is repeated for multiple results. Then are fulfilled matrixes with random values (result matrix with zeros) computed and results are written into file. With time of the execution of this implementation we will compare others parallel C++ programs. The time of execution can be seen in the *Table 4.2*.

Processor	Average time [ms]
Core i7-2600	6262
Core 2 Quad Q9550	7141
Pentium 4 650	12331
Pentium P6200	14804

Table 4.2 – C++ matrix multiplication time of execution in milliseconds

4.1.3 Multi-Threaded OpenMP

The implementation with OpenMP – API (Application Programming Interface) specification for the parallel programming is similar to the serial one. There is only the OpenMP library included and defines the parallel directives which are only few more lines and code is not too much more complicated, see the *Source 4.3*. There are two possibilities of setting the number of threads. The first one is to let it on OpenMP which creates the suitable number of threads (typically for all accessible virtual processors in system) or set with the "num_threads" directive. How it is possible to see in the *Table 4.3*, both approaches are combined. Threads one, two, four and eight are set by the author and automatically set by OpenMP.

```
#include <omp.h>
#include <iostream>
#include <fstream>
#include <Windows.h>
#include <vector>
#define THREADNUMBER 8 //number of threads
#define SIZE 1000

int main()
{
std::vector<std::vector<int> >MatrixA(SIZE, std::vector<int>(SIZE)),
       MatrixB(SIZE, std::vector<int>(SIZE)),
       MatrixC(SIZE, std::vector<int>(SIZE));

//loop  for more than one result
for(int x=0; x<24; x++)
{
       srand(timeGetTime()); // random number where seed = time

//Fill Matrix A and B with random number, result matrix C with 0
       for (int i=0; i<SIZE; i++)
       {
              for (int j=0; j<SIZE; j++)
              {
                     MatrixA[i][j] = rand() % 201 - 100;
                     MatrixB[i][j] = rand() % 201 - 100;
                     MatrixC[i][j] = 0;
              }
       }

       unsigned long time = timeGetTime(); //start time measure

       //start of parallel computing, definition of parallel work
       #pragma     omp     parallel     shared(MatrixA,     MatrixB,     MatrixC)
/*num_threads(THREADNUMBER)*/
       {
              #pragma omp for schedule(dynamic)
              //Matrix multiplication compute
              for (int i=0; i<SIZE; i++)
              {
                     for (int j=0; j<SIZE; j++)
                     {
                            for (int k=0; k<SIZE; k++)
                            MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
                     }
              }
       }
       //write result into file
       std::ofstream fout("output", std::ios::app);
       fout << timeGetTime()-time << std::endl;
       fout.close();
}
return 0;
}
```

Source 4.3 - Matrix multiplication C++ OpenMP implementation

In the *Table 4.3* are the results. The speed-up means how many times it is quicker. If there is speed-up 120% it means it is 1.2 times quicker than the base value. The results are quite good. For processor Pentium 4 its best result is 114.4% thanks to Hyper-Threading technology. This is actually very good if it is supposed some cost for the multi-threading implementation. This cost can be approximated by OpenMP with set of one-thread which is 84.24%. If it is computed this (114.4% / 84.24% = 135.8%) and it is gets 135.8%. Almost 136% is in the case of Hyper-Threading very good– the question is about the real cost.

For Pentium Processor P6200 the results are normal - 172.62%, for two cores it is not the ideal value but it is still very good in the opposite of serial results. If is supposed some cost for multi-threading as above the result is (172.62% / 83.12% = 207.67%) 207.67% - it would be more than the linear speed-up. But the cost is divided between two cores and it is only theoretical so this result is not accurate.

For Processor Core 2 Quad the results are very good - 361.75% it is not too far from the ideal 400%. There is pretty good difference in the comparison with the serial results. If it is supposed the cost for multithreading (361.75% / 91.10% = 397.11%) the result 397.11% is almost the ideal, but it is only theoretical.

The results for processor Core i7 are actually very surprising. The best results are when there are set four threads and it is 407.42% which means more than the linear speed-up. Another very surprising thing is that the best results are with four threads and not with eight threads. Hyper-Threading here doesn´t show any advantage. It cannot be explained but only guessed. More than the linear speed-up is because every core has its own L1 cache and if every core does smaller part there is lower rate of L1 cache miss. Another possible explanation is that the processor use Turbo Boost technology in some strange way. Possible interpretation why is not the best result with eight threads is that there is some bottle neck when eight threads are computed in the same time. To determine exactly this result it is needed to do more tests. But this is not related to this work.

Pentium P6200	C++ single	C++ OpenMP				
number of threads	x	1	2	4	8	auto
average [ms]	14804	17810	8576	8722	8600	9035
speed-up	100.00%	83.12%	172.62%	169.73%	172.14%	163.85%
Pentium 4 650	**C++ single**	**C++ OpenMP**				
number of threads	x	1	2	4	8	auto
average [ms]	12331	14638	10826	10779	11162	10825
speed-up	100.00%	84.24%	113.90%	114.40%	110.47%	113.91%
Core 2 Quad Q9550	**C++ single**	**C++ OpenMP**				
number of threads	x	1	2	4	8	auto
average [ms]	7141	7839	3928	1974	1998	1976
speed-up	100.00%	91.10%	181.80%	361.75%	357.41%	361.39%
Core i7-2600	**C++ single**	**C++ OpenMP**				
number of threads	x	1	2	4	8	auto
average [ms]	6262	5937	3003	1537	1560	1546
speed-up	100.00%	105.47%	208.52%	407.42%	401.41%	405.05%

Table 4.3 - Result of OpenMP matrix multiplication implementation

4.1.4 Multi-Threaded POSIX Thread

The implementation with POSIX Thread is the low-level parallel programming. The base is the same as in the previous version but only the matrix multiplication how can be seen in the *Source 4.4*. There is attempt to do parallelization simply. There is master thread which doesn´t do any computing only creates and waits for terminate of child threads. Child threads compute its part (the size of part is according number of threads) and then terminate. There aren´t things which do parallel programming really hard as passing data between thread, synchronization, creating lock and so on. But anyway it can be seen that the source code is notable more complicated than the serial version.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <windows.h>
#include <pthread.h>
#define THREADNUMBER 8 //number of threads
#define SIZE 1000

std::vector<std::vector<int> >MatrixA(SIZE, std::vector<int>(SIZE)),
       MatrixB(SIZE, std::vector<int>(SIZE)),
       MatrixC(SIZE, std::vector<int>(SIZE));
int numberCPU;

void* matrixCompute( void * param )
{
       int id = (int)param; //number of thread
       int from = (SIZE * id) / numberCPU; //place where thread should start
       int to = (SIZE * (id+1)) / numberCPU; //place where thread should end

       //chunk of matrix multiplication for thread
       for (int i=from; i<to; i++)
       {
              for (int j=0; j<SIZE; j++)
              {
                     for (int k=0; k<SIZE; k++)
                     MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
              }
       }
       return (void *)id;
}

int main()
{
//loop for more than one result
for(int x=0; x<24; x++)
{
       srand(timeGetTime()); // random number where seed = time

//Fill Matrix A and B with random number, result matrix C with 0
       for (int i=0; i<SIZE; i++)
       {
              for (int j=0; j<SIZE; j++)
              {
                     MatrixA[i][j] = rand() % 201 - 100;
                     MatrixB[i][j] = rand() % 201 - 100;
                     MatrixC[i][j] = 0;
              }
       }
       unsigned long time=timeGetTime(); //start time measure

       SYSTEM_INFO sysinfo;
       GetSystemInfo(&sysinfo );
       numberCPU = sysinfo.dwNumberOfProcessors;
       //numberCPU = THREADNUMBER;

       std::vector<pthread_t>thread(numberCPU);
       std::vector<int> return_value(numberCPU);
```

```
        //creating threads
        for ( int i=0; i<numberCPU; i++ )
        {
                pthread_create(&thread[i], 0, &matrixCompute,(void*)i );
        }

        //ending threads
        for ( int i=0; i<numberCPU; i++ )
        {
                pthread_join( thread[i], (void**)&return_value[i] );
        }

        //write result into file
        std::ofstream fout("output",std::ios::app);
        fout << timeGetTime()-time << std::endl;
        fout.close();
}
return 0;
}
```

Source 4.4 -Matrix multiplication C++ POSIX Thread implementation.

In the *Table 4.4* are results from POSIX Thread implementation. For processor Pentium 4 the result is very good 129.13% it is almost the ideal. Hyper-Threading in this case works surprisingly good. For Pentium Processor P6200 the result is also quite good 188.08%. It is not so far from the ideal.

For Core 2 Quad processor results are very surprising. The best result is 448.27% which is almost about 50% better than linear speed-up. The reason why does it happen could be, as well as in case of OpenMP above, guessed. More than linear speed-up can be because every core has its own L1 cache and if every core does smaller part there is lower rate of L1 cache miss. But this is not important for this work. In one child case the result is 113.30%. This can be because this thread has theoretically whole core for itself. In case of single-threaded program it is different that thread needs only resources for matrix multiplication, not for the whole program.

The result for processor Core i7 is again strange. It seems as Hyper-Threading doesn´t have any effect here. Not even 6% between four and eight threads this is not significant. There can be the same reason as mentioned in the previous chapter *4.1.3 Multi-Threaded OpenMP*.

Pentium P6200	C++ single	C++ POSIX				
number of threads	x	1	2	4	8	auto
average [ms]	14804	16504	8131	8209	8291	7871
speed-up	100.00%	89.70%	182.07%	180.34%	178.56%	188.08%
Pentium 4 650	**C++ single**	**C++ POSIX**				
number of threads	x	1	2	4	8	auto
average [ms]	12331	13156	9549	9571	9608	9571
speed-up	100.00%	93.73%	129.13%	128.84%	128.34%	128.84%
Core 2 Quad Q9550	**C++ single**	**C++ POSIX**				
number of threads	x	1	2	4	8	auto
average [ms]	7141	6303	3161	1593	1601	1595
speed-up	100.00%	113.30%	225.91%	448.27%	446.03%	447.71%
Core i7-2600	**C++ single**	**C++ POSIX**				
number of threads	x	1	2	4	8	auto
average [ms]	6262	6029	3051	1607	1585	1591
speed-up	100.00%	103.86%	205.24%	389.67%	395.08%	393.59%

Table 4.4 - Results of POSIX Thread matrix multiplication implementation

4.1.5 Multi-Threaded Windows Threading

Implementation of Windows threading is in the *Source 4.5*. It is very similar to rhea previous chapter *4.1.4 Multi-Threaded POSIX Thread*. It actually is possible to say almost the same what was mentioned above.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <windows.h>
#include <process.h>
#define THREADNUMBER 8 //number of threads
#define SIZE 1000

std::vector<std::vector<int> >MatrixA(SIZE, std::vector<int>(SIZE)),
       MatrixB(SIZE, std::vector<int>(SIZE)),
       MatrixC(SIZE, std::vector<int>(SIZE));
int numberCPU;

unsigned int__stdcall matrixCompute( void * param )
{
       int id = (int)param; //number of thread
       int from = (SIZE * id) / numberCPU; //place where thread should start
       int to =(SIZE * (id+1)) / numberCPU; //place where thread should end

       //chunk of matrix multiplication for thread
       for (int i=from; i<to; i++)
       {
              for (int j=0; j<SIZE; j++)
              {
```

```
                        for (int k=0; k<SIZE; k++)
                        MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
                }
        }
        return (unsigned int)id;
}

int main()
{
//loop  for more than one result
for(int x=0;x<24;x++)
{
        srand(timeGetTime()); // random number where seed = time

//Fill Matrix A and B with random number, result matrix C with 0
        for (int i=0; i<SIZE; i++)
        {
                for (int j=0; j<SIZE; j++)
                {
                        MatrixA[i][j] = rand() % 201 - 100;
                        MatrixB[i][j] = rand() % 201 - 100;
                        MatrixC[i][j] = 0;
                }
        }
        SYSTEM_INFO sysinfo;
        GetSystemInfo(&sysinfo );
        numberCPU = sysinfo.dwNumberOfProcessors;
        numberCPU = THREADNUMBER;

        unsigned long time=timeGetTime(); //start time measure
        std::vector<HANDLE>thread(numberCPU);

        //creating threads
        for ( int i=0; i<numberCPU; i++ )
        {
                thread[i]  = (HANDLE)_beginthreadex(0, 0, matrixCompute, (void*)i  ,
                0, 0);
        }

        //ending threads
        for ( int i=0; i<numberCPU; i++ )
        {
                WaitForSingleObject( thread[i], INFINITE );
                CloseHandle(thread[i]);
        }

        //write result into file
        std::ofstream fout("output",std::ios::app);
        fout << timeGetTime()-time << std::endl;
        fout.close();
}
return 0;
}
```

Source 4.5 - Matrix multiplication C++ Windows Threading implementation.

The results, which are in the *Table 4.5*, are also almost the same as in the previous chapter *4.1.4 Multi-Threaded POSIX Thread*. There is not any difference needing another explanation.

Pentium P6200	C++ single	C++ Windows Threading				
number of threads	x	1	2	4	8	auto
average [ms]	14804	17711	8537	8540	8678	8600
speedup	100.00%	83.58%	173.41%	173.36%	170.59%	172.14%
Pentium 4 650	**C++ single**	**C++ Windows Threading**				
number of threads	x	1	2	4	8	auto
average [ms]	12331	13147	9574	9559	9618	9571
speedup	100.00%	93.79%	128.80%	128.99%	128.21%	128.83%
Core 2 Quad Q9550	**C++ single**	**C++ Windows Threading**				
number of threads	x	1	2	4	8	auto
average [ms]	7141	6296	3160	1597	1602	1595
speedup	100.00%	113.43%	226.01%	447.26%	445.74%	447.67%
Core i7-2600	**C++ single**	**C++ Windows Threading**				
number of threads	x	1	2	4	8	auto
average [ms]	6262	6101	3052	1540	1584	1584
speedup	100.00%	102.65%	205.17%	406.61%	395.23%	395.28%

Table 4.5 – Result of Windows threading matrix multiplication implementation

4.1.6 Multi-Threaded Qt Framework

Implementation of Qt framework is in the *Source 4.6*. It is quite similar to the previous chapters *4.1.4* Multi-Threaded POSIX Thread and *4.1.5* Multi-Threaded Windows Threading. Information which is mentioned above is valid also for this implementation.

```
#include <qtconcurrentrun.h>
#include <QtCore/QCoreApplication>
#include <fstream>
#include <vector>
#include <windows.h>

#define THREADNUMBER 8 //number of threads
#define SIZE 1000
Using namespace QtConcurrent;

std::vector<std::vector<int> >MatrixA(SIZE, std::vector<int>(SIZE)),
       MatrixB(SIZE, std::vector<int>(SIZE)),
       MatrixC(SIZE, std::vector<int>(SIZE));
int numberCPU;

void matrixCompute(int id)
{
       int from = (SIZE * id) / numberCPU; //place where thread should start
```

```
        int to =(SIZE * (id+1)) / numberCPU; //place where thread should end

        //chunk of matrix multiplication for thread
        for (int i=from; i<to; i++)
        {
                for (int j=0; j<SIZE; j++)
                {
                        for (int k=0; k<SIZE; k++)
                        MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
                }
        }
}

int main(int argc, char **argv)
{
        //loop  for more than one result
        for(int x=0; x<24; x++)
        {
                srand(timeGetTime()); // random number where seed = time

//Fill Matrix A and B with random number, result matrix C with 0
                for (int i=0; i<SIZE; i++)
                {
                        for (int j=0; j<SIZE; j++)
                        {
                                MatrixA[i][j] = rand() % 201 - 100;
                                MatrixB[i][j] = rand() % 201 - 100;
                                MatrixC[i][j] = 0;
                        }
                }
                unsigned long time=timeGetTime(); //start time measure

                //set number of threads
                numberCPU = QThread::idealThreadCount();
                //numberCPU = THREADNUMBER;

                std::vector<QFuture<void>>thread(numberCPU);
                //creating threads
                for ( int i=0; i<numberCPU; i++ )
                {
                        thread[i] = run(matrixCompute, int(i));
                }

                //ending threads
                for ( int i=0; i<numberCPU; i++ )
                {
                        thread[i].waitForFinished();
                }
                //write result into file
                std::ofstream fout("output",std::ios::app);
                fout << timeGetTime()-time << std::endl;
                fout.close();
        }
}
```

Source 4.6 - Matrix multiplication C++ Qt framework implementation

The results are very similar as in the previous chapters *4.1.4* Multi-Threaded POSIX Thread and *4.1.5 Multi-Threaded Windows Threading*. There is not any important difference which isn´t explained above.

Pentium P6200	C++ single	C++ Qt				
number of threads	**x**	**1**	**2**	**4**	**8**	**auto**
average [ms]	14804	15786	7737	8424	7953	8035
speedup	100.00%	93.78%	191.33%	175.73%	186.14%	184.23%
Pentium 4 650	**C++ single**	**C++ Qt**				
number of threads	**x**	**1**	**2**	**4**	**8**	**auto**
average [ms]	12331	13104	9578	9584	9585	9590
speedup	100.00%	94.10%	128.74%	128.66%	128.65%	128.58%
Core 2 Quad Q9550	**C++ single**	**C++ Qt**				
number of threads	**x**	**1**	**2**	**4**	**8**	**auto**
average [ms]	7141	6274	3148	1595	1591	1590
speedup	100.00%	113.82%	226.81%	447.64%	448.74%	449.09%
Core i7-2600	**C++ single**	**C++ Qt**				
number of threads	**x**	**1**	**2**	**4**	**8**	**auto**
average [ms]	6262	6230	3117	1576	1593	1594
speedup	100.00%	100.52%	200.92%	397.45%	393.06%	392.81%

Table 4.6 - Result of QThread matrix multiplication implementation

4.1.7 Single-Threaded Java

Single-threaded Java implementation – the source code is in the *Source 4.7 - Matrix multiplication Java serial implementation* Implementation is quite similar to C++ implementation. Variances between these two implementations are because of different concepts of these languages. There is used the function "System.nanoTime()" for time measure which is accurate in nanoseconds.

```
import java.util.Random;
import java.io.*;

public class MM {
       static final intSIZE = 1000;

       public static void main(String[] args)
    {
             int MatrixA[][] = new int[SIZE][SIZE];
             int MatrixB[][] = new int[SIZE][SIZE];
             int MatrixC[][] = new int[SIZE][SIZE];

             //loop  for more than one result
             for(int x=0;x<24;x++)
             {
             //Fill Matrix A and B with random number, result matrix C with 0
                    Random randomGenerator = new Random();
                    for (int i=0; i<SIZE; i++)
                    {
                           for (int j=0; j<SIZE; j++)
                           {
                           MatrixA[i][j] = randomGenerator.nextInt(201) - 100;
                           MatrixB[i][j] = randomGenerator.nextInt(201) - 100;
                           MatrixC[i][j] = 0;
                           }
                    }
                    //start time measure
                    long startTime = System.nanoTime();

                    //Matrix multiplication compute
                    for (int i=0; i<SIZE; i++)
                    {
                           for (int j=0; j<SIZE; j++)
                           {
                                  for (int k=0; k<SIZE; k++)
                           MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
                           }
                    }
             //write result into file
                    BufferedWriter out;
                    try{
                    FileWriter fstream = new FileWriter("output", true);
                    out = new BufferedWriter(fstream);
                    out.write(System.nanoTime() - startTime+"\n");
                    out.close();
                      }
                    catch (Exception e)
                      {//Catch exception if any
                           System.err.println("Error: " + e.getMessage());
                      }
             }
    }
}
```

Source 4.7 - Matrix multiplication Java serial implementation

In next part of the work is time comparison of runtime of this implementation with parallel implementation. The time of execution is in the *Table 4.7*.

Processor	Average time [ms]
Core i7-2600	6627
Core 2 Quad Q9550	10674
Pentium 4 650	18487
Pentium P6200	21223

Table 4.7 – Java matrix multiplication time of execution in milliseconds

4.1.8 Multi-Threaded Java

Parallel implementation in Java is the low-level way of parallel programming – shown in the *Source 4.8*. The base of matrix multiplication is the same as in the serial algorithm. There are master and child threads. The implementation is not special with comparison with other implementation in C++.

```
import java.util.Random;
import java.io.*;

public cclass ExtendThread extends Thread {
	@Override
	Public void run()
	{
	int  id  =  Integer.valueOf(Thread.currentThread().getName());  //number
of thread
	int from = (SIZE * id) / numberCPU; //place where thread should start
	int to =(SIZE * (id+1)) / numberCPU; //place where thread should end

	//chunk of matrix multiplication for thread
	for (int i=from; i<to; i++)
	    {
	for (int j=0; j<SIZE; j++) {
	      for (int k=0; k<SIZE; k++)
MatrixC[i][j] += MatrixA[i][k] * MatrixB[k][j];
	        }
	  }
    }
static final int SIZE = 1000;
static final int THREADNUMBER = 8;
static int numberCPU = THREADNUMBER;

static int MatrixA[][] = new int[SIZE][SIZE];
static int MatrixB[][] = new int[SIZE][SIZE];
static int MatrixC[][] = new int[SIZE][SIZE];

	public static void main(String[] args)
	{
	for(int x=0;x<24;x++)
	{
```

```
		//get number of available processors
		Runtime runtime = Runtime.getRuntime();
		numberCPU = runtime.availableProcessors();

	//Fill Matrix A and B with random number, result matrix C with 0
		Random randomGenerator = new Random();
		for (int i=0; i<SIZE; i++)
		{
			for (int j=0; j<SIZE; j++)
			{
				MatrixA[i][j] = randomGenerator.nextInt(201) - 100;
				MatrixB[i][j] = randomGenerator.nextInt(201) - 100;
				MatrixC[i][j] = 0;
			}
		}
		//start time measure
		long startTime = System.nanoTime();

		  ExtendThread ThreadArray[] = new ExtendThread[numberCPU];
	//creating threads
	for ( int i=0; i<numberCPU; i++ )
		  {
		ThreadArray[i] = new ExtendThread();
		ThreadArray[i].setName(Integer.toString(i));
		ThreadArray[i].start();
		  }

	//ending threads
	for ( int i=0; i<numberCPU; i++ )
		  {
		try
		{
					ThreadArray[i].join();
				} catch (InterruptedException e)
				{//Catch exception if any
					e.printStackTrace();
				}
		  }
	//write result into file
			BufferedWriter out;
			try
			{
				FileWriter fstream= new FileWriter("output", true);
				out = new BufferedWriter(fstream);
				out.write(System.nanoTime() - startTime+"\n");
				out.close();
			}catch (Exception e)
			{//Catch exception if any
				System.err.println("Error: " + e.getMessage());
			}

	}
	}
}
```

Source 4.8 - Matrix multiplication Java multi-threaded implementation

In the *Table 4.8* are the results of Java multi-threaded implementation. The results are very good. Even with one child thread there can be seen some speed-up. This can be because this thread has theoretically whole core for itself. The thread has only resources for matrix multiplication in case of using one core by single-threaded program. In case of Core i7 can be seen that hyper-threading technology doesn't have expected effect again.

Pentium P6200	Java single	Java Multi-Threading				
number of threads	x	1	2	4	8	auto
average [ms]	21223	20241	9919	9783	9548	9713
speedup	100.00%	104.85%	213.96%	216.93%	222.28%	218.51%
Pentium 4 650	**Java single**	**Java Multi-Threading**				
number of threads	x	1	2	4	8	auto
average [ms]	18487	17614	13134	13145	13164	13146
speedup	100.00%	104.96%	140.76%	140.64%	140.44%	140.63%
Core 2 Quad Q9550	**Java single**	**Java Multi-Threading**				
number of threads	x	1	2	4	8	auto
average [ms]	10674	10038	5041	2684	2593	2669
speedup	100.00%	106.39%	211.75%	397.70%	411.61%	399.92%
Core i7-2600	**Java single**	**Java Multi-Threading**				
number of threads	x	1	2	4	8	auto
average [ms]	6627	6258	3133	1659	1630	1623
speedup	100.00%	105.89%	211.54%	399.51%	406.65%	408.31%

Table 4.8 - Results of Java multi-threaded matrix multiplication implementation

4.1.9 Compare and Evaluation of Different Implementation

Firstly it is focused on implementation itself. It can actually split into two categories the low-level and the high-level which is explained in the chapter *3.3 Parallel Programming*. Actually from this simple algorithm it is not possible to say in which way it is easier or harder to programme. There are not used all of the possibilities of parallel programming. Another thing is that author doesn´t have enough experiences with each way of implementation. Some of them were easier to understand for the author but this is not accurate evaluation and it is after all more about which is easier to learn than which is easier to use. There are lots of ways of code evaluation: complexity, maintenance and so on, but it is not too much significant especially for this case. Hence

in the high-level parallel programming in this case only OpenMP is easier to programme than in the low-level - in this case all others.

Another way of comparison is performance - in this case C++ and Java programs are split-up. Because even small change of implementation has influence on performance, hence the comparison is not significant. For example if C++ used instead vector another store the performance it would be different. Also between processors are different results. The results are shown in the *Table 4.9* for each processor and implementation is there only the best value for transparency.

Pentium P6200	C++ S	OpenMP	Posix	Windows	Qt	Java-S	Java-M
threads	x	2	auto	2	2	x	8
average [ms]	14804	8576	7871	8537	7737	21223	9548
speedup	100.00%	172.62%	188.08%	173.41%	191.34%	100.00%	222.28%
Pentium 4 650	**C++ S**	**OpenMP**	**Posix**	**Windows**	**Qt**	**Java-S**	**Java-M**
threads	x	4	2	4	2	x	2
average [ms]	12331	10779	9549	9559.4	9578	18487	13134
speedup	100.00%	114.40%	129.13%	128.99%	128.74%	100.00%	140.76%
Core 2 Quad	**C++ S**	**OpenMP**	**Posix**	**Windows**	**Qt**	**Java-S**	**Java-M**
threads	x	4	4	auto	auto	x	8
average [ms]	7141	1974	1593	1595	1590	10674	2593
speedup	100.00%	361.75%	448.27%	447.71%	449.12%	100.00%	411.65%
Core i7-2600	**C++ S**	**OpenMP**	**Posix**	**Windows**	**Qt**	**Java-S**	**Java-M**
threads	x	4	8	4	4	x	auto
average [ms]	6262	1537	1585	1540	1576	6627	1623
speedup	100.00%	407.42%	395.08%	406.62%	397.34%	100.00%	408.32%

Table 4.9 – Compare of results of different implementations

For Pentium processor P6200 has OpenMP and Windows threading a little bit less performance than POSIX thread and Qt framework. For Windows threading it is quite strange. It is probably because for every processor are different the ideal instructions. The error of measurement is not likely because two best and worst measurements are discarded. Computer runs only Windows threading program with real-time priority. In Java is strange that the best results are with eight child threads but speed-up is excellent.

For Pentium 4 650 we it is possible to see that Hyper-Threading technology works surprisingly fine except OpenMP which works rather on the average. But in total are the results very good in benefit of Hyper-Threading.

For Core 2 Quad processor are the results more surprising. The best result is 449.12% which is almost about 50% better than linear speed-up. How it is possible is mentioned in the chapter *4.1.4 Multi-Threaded POSIX Thread*. Again the worst result come with OpenMP. For Java is the result also very good.

For Core i7-2600 processor are the results quite strange. The expected speed-up from Hyper-Threading just does not appear. Also as only one has the best result with OpenMP and with comparison of other processors has much better results in Java. This is probably newer generation of processor. It is better optimized for this kind of executions. By comparing this processor with older Core 2 Quad the results are significantly better only in single-threaded algorithm, OpenMP and Java. In single-threaded and OpenMP it is matter of higher frequency (3.4GHz vs. 2.83GHz). In other C++ multi-threaded programs it is practically the same through the higher frequency. Only the Java is significantly better.

4.1.10 Disadvantage of Hyper-Threading

Hyper-Threading technology is described in the chapter *3.1.2 Hyper-Threading*. In this chapter is proved its main disadvantage. Shortly in some cases low priority processes can take processor time from high priority processes. For those there are needed two programs – one with high priority, for these purposes are used the matrix multiplication programs and another with low priority. As the low priority program was created simple program which creates eight threads with infinite loop, as can be seen in the *Source 4.9*.

```
#include <omp.h>
#define THREADNUMBER 8 //number of threads

int main()
{
        //start of parallel computing
        #pragma omp parallel num_threads(THREADNUMBER)
        {
                #pragma omp for
                //infinite loop
                for (int i=0; i<100; i++)
                {
                        for (int j=0; j<100; j++)
                        {
                                j--;
                        }
                }
        }
}
```

Source 4.9 – Infinite loop for fully workloaded processor

In the *Table 4.10* is listed comparison of the same processors – once time with Hyper-Threading on and once time with Hyper-Threading off. For these purposes are chosen the values where is disadvantage most significant. In Core i7 processor disadvantage of Hyper-Threading sometimes does not appear in cases when author expected.

Pentium 4 650*	**C++ S**	**OpenMP**	**Posix**	**Windows**	**Qt**	**Java-S**	**Java-M**
threads	**x**	**1**	**1**	**1**	**1**	**x**	**1**
average	14783	19984	17008	16927	16927	26727	25156
speedup	100.00%	73.97%	86.92%	87.33%	87.33%	100.00%	106,25%
Pentium 4 650**	**C++ S**	**OpenMP**	**Posix**	**Windows**	**Qt**	**Java-S**	**Java-M**
threads	**x**	**1**	**1**	**1**	**1**	**x**	**1**
average	12265	14468	13128	13208	13217	18248	17535
speedup	100.00%	84.77%	93.43%	92.86%	92.80%	100.00%	104,07%
Core i7-2600*		**C++ S**	**OpenMP**	**OpenMP**	**OpenMP**	**Java-S**	**Java-M**
threads		**x**	**1**	**2**	**4**	**x**	**1**
average		6090	7055	3553	2348	9358	8930
speedup		100.00%	86.31%	171.39%	259.35%	100.00%	104.79%
Core i7-2600**		**C++ S**	**OpenMP**	**OpenMP**	**OpenMP**	**Java-S**	**Java-M**
threads		**x**	**1**	**2**	**4**	**x**	**1**
average		6249	5948	2999	1514	6572	6245
speedup		100.00%	105.07%	208.40%	412.81%	100.00%	105.25%
*Hyper-Threading on		**Hyper-Threading off			-both with infinite loop		

Table 4.10 - Disadvantage of HT, comparison with the same processor with HT off

For Pentium 4 processor there is in every situation worst result. The best result is in the case of single-threaded C++ (12265 / 14783 = 82.97%) which is only 82.97% of performance – comparing to the same situation with Hyper-Threading off. The worst result is in the case of single-threaded Java (18248 / 26727 = 68.28%) which is only 68.28% of performance with Hyper-Threading off. Disadvantage of Hyper-Threading is in this case proved.

The situation in case of processor Core i7 is not so clear. In some expected situations its disadvantage of Hyper-Threading does not appear. But in cases which are in the *Table 4.10* is the disadvantage proved. This processor disadvantages were shown less often and above all it cannot be entirely sure that it is because of Hyper-Threading. In many cases when is expected to appear disadvantage of Hyper-Threading were the results opposite. There are lots of others results therefore all of them are in supplements.

4.2 Chess Application

4.2.1 Methodology

The chess application is written in C++ programming language with Qt framework in version 4.8.0. As software development tool is used Microsoft Visual Studio 2010 Professional. For testing is used processor Intel® Core™2 Quad Processor Q9550 in MS Windows 7 environment.

Author decided for GUI (Graphical User Interface) chess application with AI (Artificial Intelligence) because he considers that as good example. Parallelism is good solution to provide still responsible GUI. The chess application is good example of such application which needs almost infinite performance (nowadays unavailable for any computer). It doesn´t matter which algorithm is used. The application is only demonstrative. Hence author will not try to do the best chess application, but demonstrates some possibilities of parallelism.

First of all is introduced the application in serial way. There are shown disadvantages and possible solutions. With the help of multi-threading is edited the application to stays responsible during intensive processing (with using Qt framework way of parallelism). After that is used parallelism for AI (Artificial Intelligence) or chess brain (with using OpenMP way of parallelism). This is compared with another

algorithm, which is also parallelized. In the next step are compared and evaluated all solutions.

To measure the time is used the mean of fifteen moves. Moves are the same for every test. In some settings there are given first fifteen moves which are used in every setting.

4.2.2 Single-threaded Chess Application Using Minimax

Minimax is one of the easiest algorithms which can be used for chess AI (Artificial Intelligence) but it is also one of the weakest algorithms for chess AI. For demonstration of chess brain it is more than satisfactory. Minimax algorithm is just searching in the tree of moves into a given depth. Firstly it is needed to know how good is given position. The basic evaluation of pieces in chess is quite easy – it is only needed to count all the pieces. For better evaluation is necessary to compute positions of figures. Every chess brain has some evaluation function.

In average we have 35 moves from position [25] which means that minimax has to search through 35^n position (and every of them evaluate) where n is the depth of searching. When it searches into the depth of four (which are four half-moves or just two moves) it is $35^4 = 1\,500\,625$ of evaluations of positions and every another half-move 35-times more. How long it takes depend on implementation. That is why it is this algorithm so weak because it is too slow.

```
int chessAI::minimax(int depth)
{
	int realRange, player, price;
	int best = -9990 - depth;
	int realMoveFrom[MOVESIZE];
	int realMoveTo[MOVESIZE];

	//choose which one play white/black
	player = playerChoose();

	//return evaluation of position
	if (depth <=0) return positionValue(player);

	//move generator
	realRange = possibleMovement(player, realMoveFrom, realMoveTo);

	if (doMove(1,1) == 2) return best; //check for checkMate
	if (doMove(1,1) == 3) return 0; //check for stateMate

	//try every move
	for (int i = 0; i < realRange; i++)
	{
		int doMoveReturn;
		doMoveReturn = doMove(realMoveFrom[i],realMoveTo[i]);
		if (doMoveReturn > 10) promotion(doMoveReturn, 5);
		price = -minimax(depth-1);
		lastMove--;

		if (price > best)
		{
			best = price;
		}
	}

return best;
}
```

Source 4.10 – Author´s implementation of minimax for chess artificial intelligence

In the *Source 4.10* can be seen author´s implementation of minimax for the chess brain. It is quite simple - only several lines of code. But much harder is to implement other necessary functions like evaluation, move function and so on. The best variable has very low initializing value. It is for the checkmate situation which has to have the very low or very high evaluation. The “realRange” variable is index for the “realMoveFrom” and the “realMoveTo” arrays. If the function “doMove” return more than 10 - it is position of promotion. At this case it does return the queen for simplicity because the other figures are used only very rarely. Then algorithm searches for the best evaluation of position until it finds it and writes it into the “best” variable.

```
int chessAI::AIMovement(int *from, int *to, int depth)
{
	int realRange, bestInRange;
	int player, price;
	int best = -10000; //We can be sure, we find better
	int realMoveFrom[MOVESIZE];
	int realMoveTo[MOVESIZE];

	//resize vector
		if (lastMove > 44)
		{
			chessboard.push_back(std::vector<int>(120));
			chessboard.push_back(std::vector<int>(120));
		}

	//choose which one play white/black
	player = playerChoose();

	//move generator
	realRange = possibleMovement(player, realMoveFrom, realMoveTo);

	//try every move
	for (int i = 0; i < realRange; i++)
	{
		int doMoveReturn;
		doMoveReturn = doMove(realMoveFrom[i],realMoveTo[i]);
		if (doMoveReturn > 10) promotion(doMoveReturn, 5);
		price = -minimax(depth);
		lastMove--;

		if (price > best)
		{
			best = price;
			*from = realMoveFrom[i];
			*to = realMoveTo[i];
		}
	}
	return best;
}
```

Source 4.11 – Author´s implementation of chess artificial intelligence

In the *Source 4.11* is the rest of the implementation of the chess artificial intelligence. It is actually similar to the minimax function. It resizes a vector variable because in constructor of class it is defined in size 50*120. 50 is number of moves. The vector "lastMove" is resized when is bigger than 44 because the same vector is used for computing function inside the minimax which computes few moves forward. 120 stands for chessboard which is implemented as 10*12 as is possible to see in the *Source 4.12*. This implementation is simpler because it is not necessary to take care about the boundary of chessboard. The move forward is +10 instead +8 (and multiples as 16, 24,

32 and so on) which is better for readability. Rest of the function is almost the same. Only when it finds better price it saves the move into “*from” and “*to” pointers.

```
// Chessboard representation 10*12

//100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
//100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
//100,  -4,  -2,  -3,  -5,  -6,  -3,  -2,  -4, 100,
//100,  -1,  -1,  -1,  -1,  -1,  -1,  -1,  -1, 100,
//100,   0,   0,   0,   0,   0,   0,   0,   0, 100,
//100,   0,   0,   0,   0,   0,   0,   0,   0, 100,
//100,   0,   0,   0,   0,   0,   0,   0,   0, 100,
//100,   0,   0,   0,   0,   0,   0,   0,   0, 100,
//100,   1,   1,   1,   1,   1,   1,   1,   1, 100,
//100,   4,   2,   3,   5,   6,   3,   2,   4, 100,
//100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
//100, 100, 100, 100, 100, 100, 100, 100, 100, 100

//100 is square where is not possible to go
//1 = pawn, 2 = knight, 3 = bishop, 4 = rook, 5 = queen, 6 = king
//positive are white, negative are black
```

Source 4.12 – Implementation of chessboard 10*12

Now it is shows how is called the “AIMovement” function (*Source 4.11*) from GUI (Graphical User Interface) class. It is possible to see this in the *Source 4.13*. The “movesWidget” is text field where are print-outs from the application. It prints-out are the time how long the artificial intelligence takes the “thinking” and evaluation – which is the “best” value. After the computation of the “best” value it uses the “doMove” function to finish the move. From this function it gets the result which is printed out. If the result is more than 10 it returns position of promotion and computer always automatically promotes pawn onto queen (in case of user, it is possible to choose). After that the pieces are moved into the present state. Then program continues in calling function which finishes very quickly which means the continuation of event-loop.

```
void MainWindow::StartAI()
{
	int from, to, result;

	unsigned long time  =timeGetTime(); //start time measure
		result = AI->AIMovement(&from, &to, depth);
	movesWidget->append("Thinking  took  me:  "+QString::number(timeGetTime()-
		time)+"ms"); //write time

	movesWidget->append("Evaluation: "+QString::number(result));

	result = AI->doMove(from, to); //write move into array
	if (result == -1) movesWidget->append("White now!");
	if (result == 1) movesWidget->append("Black now!");
	if (result == 2) movesWidget->append("CheckMate!");
	if (result == 3) movesWidget->append("StateMate!");

	//automatic promotion onto queen
	if (result > 10)
	{
		movesWidget->append("Promotion!");
		AI->promotion(result,5);
		if (result > 50)
			movesWidget->append("White now!");
		else
			movesWidget->append("Black now!");
	}

	repaintFigures();
}
```

Source 4.13 – Call of the AIMovement function (Source 4.11) from GUI class

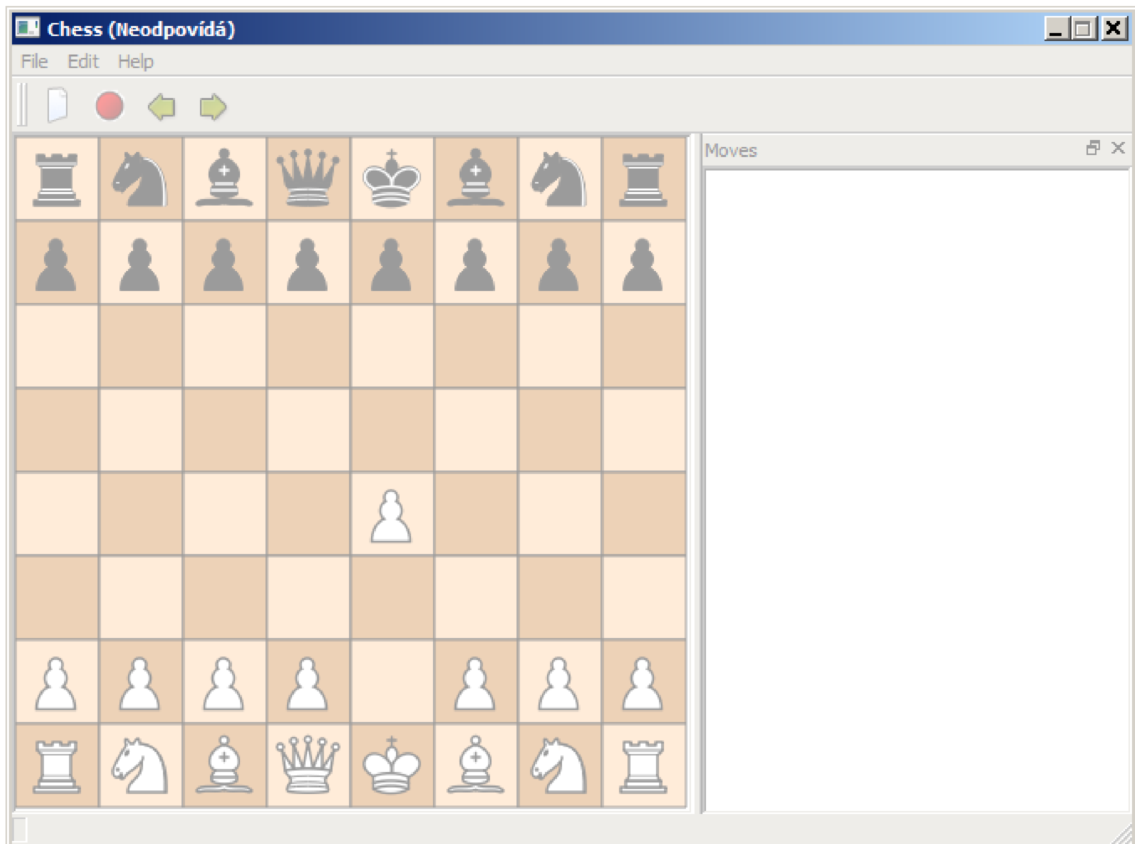
From the implementation it is possible to see, that if the chess artificial intelligence computes, the graphics user interface does not respond. This problem depends on how quick artificial intelligence is. If it lasts longer than let say 200 milliseconds user can probably notice it. When it lasts longer than one second it starts to be a problem. More than five seconds is almost the critical error. The time of computing can be seen in the *Table 4.11*. When the depth is equal to one, the average time is 81 milliseconds. When the depth is equal to two, the situation is quite bad- it lasts 2.93 seconds. With depth three and more is the situation critical.

depth:	1	2	3	4
time (ms):	81	2930	99964	around 35-times more
time (sec):	0.081	2.93	99.964	around 35-times more
time (min):	0.00135	0.0488	1.67	around 35-times more

Table 4.11 – Time of single-threaded minimax computing

In the *Picture 4.6* is shown non responding program. It happens when program does not respond for some time back to the operation system. The graphics user interface is even

so unresponsive that it does not repaint itself. It is overlaid as in the *Picture 4.7*. In the case that user doesn't know that AI is computing the moves, he'll probably try to exit the program by standard way then non-standard way. If user would know that the program after computing behaved correctly, he would wait. But it is certainly not user-friendly behaviour.

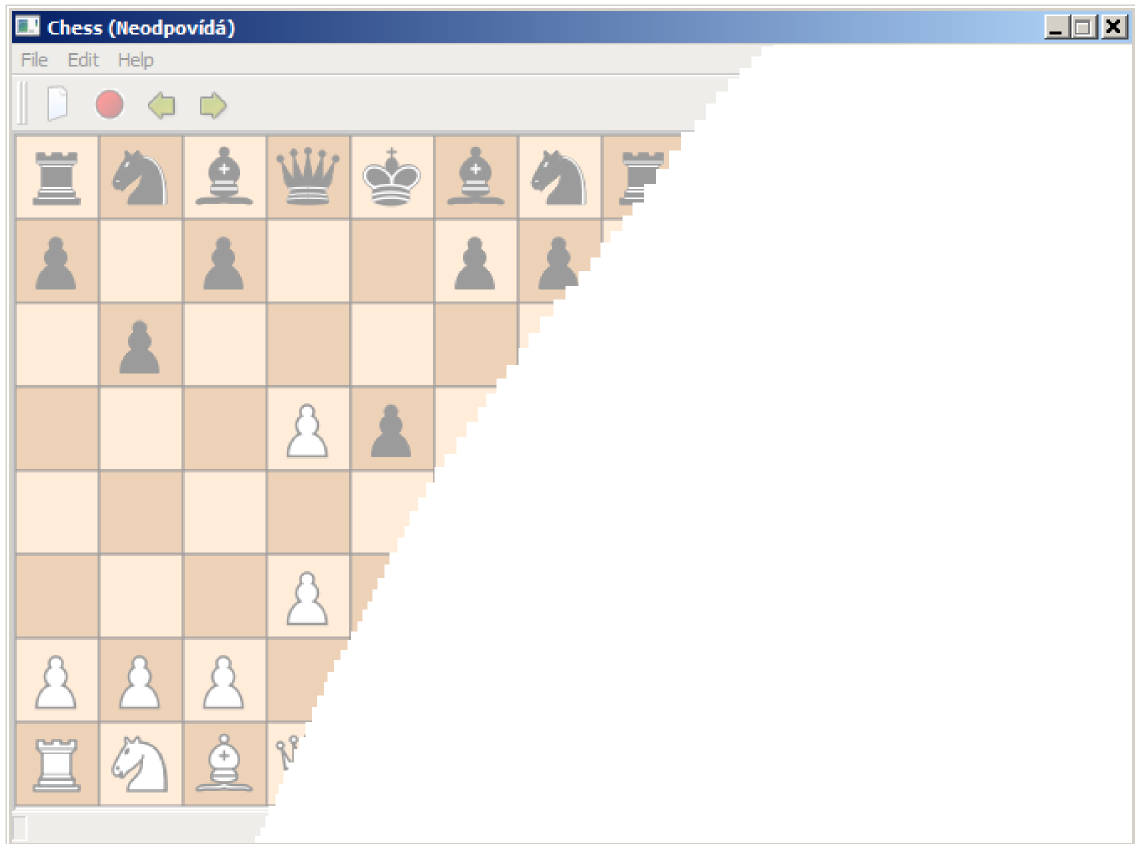


Picture 4.6 – Chess program is in "not responding" state

There are two basic solutions for this kind of problem. The first is changing of artificial intelligence algorithm. This algorithm interrupts the computing to check the event loop. It is done in every few milliseconds. It is necessary to change the algorithm. In this case it can be done but it is not really good solution because it reduces reusability. There are also cases when is not possible to change the algorithm or it is very time consuming.

In most of cases it is better to have one more thread for the event loop. In this case the graphical user interface is still responding and it is not needed to edit the artificial intelligence algorithm. Even if we have only one processor (or one-core) system, interface is still responsible. Switching between many threads is task of operation

system. This solution is shown in the next chapter *4.2.3 Multi-threaded Chess Application Using Minimax*.



Picture 4.7 - Chess program which is "not responding" and was partially overlaid

How good aritificial intelligence as opponent is could be very interesting question. If the depth of minimax is zero it is very weak opponent. In this case AI just takes the best piece which it can - without taking care of player´s movement. Artificial intelligence is able to sacrifice a queen for a pawn. Even the player who has just learned chess rules probably wins. When the depth of one is set AI examines the player´s response movements. This is enough difficulty for absolute beginners. With higher depths the algorithm is stronger but there are two or three missing features which can do the really strong chess brain - database of chess opening and database of chess ending games or special algorithm for ending games. With these features and depths three or four AI could be strong enough against the advanced players.

4.2.3 Multi-threaded Chess Application Using Minimax

In this chapter is shown how to solve the problem with “not responding“ GUI (Graphical User Interface) during intensive processing. With using Qt framework it is quite easy job how can be seen in the *Source 4.14*. It uses the same “StartAI“ function which is in the *Source 4.13* except the last call of the function “repaintFigures” which is now included in the “multiAI” function. First of all it tests the flag “AICompute”. If it is true the function cannot be executed again. If not, it executes the function. The first of all the “AICompute“ flag is set. Then is called the “repaintFigures“ function (after player´s move). After this starts parallelization process. It creates “thread“ which starts with the function with non-return value (void). Save thread is necessary because it accesses it on the next line. Function “run” (in QTConcurrent namespace) starts thread in the function “StartAI()“ with parent “this“ pointer. The while-loop is running till the end of the “StartAI()“ function. In the loop it calls the “sleep“ which is quite important. If it doesn’t call the loop it consumes all the performance of one processor (or core). With attribute one which means one millisecond is the consumption of performance on the test system insignificant. If it would be significant it can be set more than one millisecond. The “qApp->processEvents()“ function processes all events which occurred. After this it changes the “AICompute“ flag back and repaints the figures.

```
void MainWindow::multiAI()
{
	if (AICompute)
	{
		movesWidget->append("Sorry, I´m thinking wait.");
		return;
	}

	AICompute = true;
	repaintFigures();

	//compute on another thread!
	QFuture<void> thread = QtConcurrent::run(this, &MainWindow::StartAI();
	while (thread.isRunning())
	{
		Sleep(1);
		qApp->processEvents();
	}

	AICompute = false;
	repaintFigures();
}
```

Source 4.14 – Graphical user interface in another thread

But this is not all what is needed to do. Functions as “back” or “forward” aren´t probably implemented to work in multithread environment in another words functions aren´t thread-safe. The simplest way is the same procedure as in the Source 4.14 – it uses the flag. When the flag is on then the function prints-out the caution and stops the processing. This way is not too much user-friendly but despite it much more user-friendly than having a “not responding” program. There could be compared three different implementations of the “moveBack” function as in the Source 4.15, Source 4.16 and *Source 4.17*. In the *Source 4.15* is implementation from previous chapter *0* The chess application is written in C++ programming language with Qt framework in version 4.8.0. As software development tool is used Microsoft Visual Studio 2010 Professional. For testing is used processor Intel® Core™2 Quad Processor Q9550 in MS Windows 7 environment.

Author decided for GUI (Graphical User Interface) chess application with AI (Artificial Intelligence) because he considers that as good example. Parallelism is good solution to provide still responsible GUI. The chess application is good example of such application which needs almost infinite performance (nowadays unavailable for any computer). It doesn´t matter which algorithm is used. The application is only demonstrative. Hence author will not try to do the best chess application, but demonstrates some possibilities of parallelism.

First of all is introduced the application in serial way. There are shown disadvantages and possible solutions. With the help of multi-threading is edited the application to stays responsible during intensive processing (with using Qt framework way of parallelism). After that is used parallelism for AI (Artificial Intelligence) or chess brain (with using OpenMP way of parallelism). This is compared with another algorithm, which is also parallelized. In the next step are compared and evaluated all solutions.

To measure the time is used the mean of fifteen moves. Moves are the same for every test. In some settings there are given first fifteen moves which are used in every setting.

Single-threaded Chess Application Using Minimax. As is obvious, “AI” is instance of the artificial intelligence class in program. In the *Source 4.16* is the simplest implementation of the thread-safe function which is used in author´s implementation. In

the *Source 4.17* it is the ideal implementation. It stops computing and then shifts movement in a safe way. The “stopCompute()” function could be also in the “moveShift” function. But what is important in this way of implementation – it allows the user to move back even if artificial intelligence computes – which can take few minutes.

```
void MainWindow::moveBack()
{
	AI->moveShift(-1);
	repaintFigures();
}
```

Source 4.15 – The single-threaded “moveBack()” function from the chapter 0

```
void MainWindow::moveBack()
{
	if (AICompute)
	{
		movesWidget->append("Sorry, I´m thinking wait.");
		return;
	}
	AI->moveShift(-1);
	repaintFigures();
}
```

Source 4.16 – Multi-threaded implementation of the “moveBack()” function in this chapter

```
void MainWindow::moveBack()
{
	if (AICompute)
		AI->stopCompute();

	AI->moveShift(-1);
	repaintFigures();
}
```

Source 4.17 – The ideal multi-threaded implementation of the “moveBack()” function

In author´s program most of the non-threaded-safe functions are edited in the *Source 4.16* way for simplification. The program doesn´t have ambition to be the best chess program or have the best graphical user interface. That is why is chosen the simplest way, the program is only for demonstration.

Picture 4.8 - Chess program is responsible even during intensive processing.

The program is strong just like single-threaded implementation. The time for computation is a little bit different (with one-core little bit slower, with multi-core little bit faster) but insignificantly. Because the program, when is under intensive processing, is not in the "not responding" state, it can eventually use the minimax with depth three. It waits for move 1.67 minutes (It is possible to see in the *Table 4.11 – Time of single-threaded minimax computing*) is quite long time but it is still possible if program is responding at least. Then the program is stronger than single-threaded because it is usable with minimax depth three.

4.2.4 Multi-threaded AI Chess Application Using Minimax

In previous chapters was added one thread to allow the program to response during an intensive processing. In this chapter it comes out with program from the previous chapter. Here it paralyzes the minimax algorithm. This part uses most of the processor time. Other parts it is not useful to paralyze because they take only insignificant amount of time. It seems very simple and it is shown in the chapter *4.1 Matrix Multiplication*

and there is very similar situation – again the for-loop which has to be paralyzed (see the *Source 4.11 – Author's implementation of chess artificial*). There is obvious how important is the suitable design. If is known that this algorithm has to be parallelized (in OpenMP way) it is done in a suitable way – debugged and then is easily added parallel directive. Or it is programme in a parallel way from the start (in case of another parallel way).

In the *Source 4.11* almost every function has to use some kind of global variable (in this case member variable) because in these functions are missing arguments for the proper work. In some cases the reason is because of bad programming habits. In some cases it is proper way of programming. When in the most of functions is used the same variable, the proper way is to create it as a member (in case of the class) or a global variable. In case of the class is also the proper way to create the member variable if it needs to store the value after the function finishes.

```
int chessAI::AIMovement(int *from, int *to, int depth)
{
	int realRange, bestInRange;
	int player, price;
	int best = -10000; //We can be sure, we find better
	int realMoveFrom[MOVESIZE];
	int realMoveTo[MOVESIZE];

	//resize vector
		if (lastMove > 44)
		{
			chessboard.push_back(std::vector<int>(120));
			chessboard.push_back(std::vector<int>(120));
		}

	//choose which one play white/black
	if ((lastMove % 2) == 0)
		player = 1;
	else
		player = -1;

	//move generator
	realRange = possibleMovement(player, realMoveFrom, realMoveTo);

	//create parallel variables
	int lastMoveM = lastMove;
	std::vector<std::vector<int> >chessboardM(50, std::vector<int>(120));
	for(int i =44; i<lastMove; i++)
		chessboardM.push_back(std::vector<int>(120));

	chessboardM = chessboard;

#pragma  omp  parallel  for  firstprivate(realMoveFrom,  realMoveTo,  lastMoveM,\
chessboardM, depth, price) schedule(dynamic)
```

```
	for (int i = 0; i < realRange; i++)
	{
		int doMoveReturn;
		doMoveReturn = doMove(realMoveFrom[i], realMoveTo[i], &chessboardM,
			&lastMoveM);

		if  (doMoveReturn  >  10)  promotion(doMoveReturn,  5,  &chessboardM,
			lastMoveM);

		price = -minimax(depth, &lastMoveM, &chessboardM);
		lastMoveM--;

		#pragma omp critical
		{
			if (price > best)
			{
				best = price;
				*from = realMoveFrom[i];
				*to = realMoveTo[i];
			}
		}
	}
	return best;
}
```

Source 4.18 - Author´s implementation of chess multi-threads artificial intelligence

The parallel version in the *Source 4.18* is similar to the serial version in the *Source 4.11*. OpenMP API (Application Programming Interface) doesn´t allow to using member variables in a parallel part as private variables. Because of this fact it is necessary to transform member variables into local "parallel" variables. Then it is needed to change every function which is called from the parallel part. Even functions which are called from called function and so on. The missing argument has to be added (instead using member variables use argument of functions) and functions have to be changed. Pointer or references are used instead of member variables. It's important to create the parallel variables "lastMoveM" and vector "chessboardM". Sometimes "chessboardM" need to be resized, before it is copied to it the original variable values.

Then it starts the parallel part which begins with pragma directive which says to compiler how to compile. The directive "for" is used for the next for-loop. After this there are used directives "firstprivate" and "private". It ensures that enumerated variables are not shared. The "first" before "private" ensures that in the beginning variables have the same value (in another case there will be undefined value). Then is used the directive "schedule(dynamic)" which ensures that every thread does only one iteration and asks master thread for the new job. This is a little bit slower than a static

way but in this case when it is used, it would be slower thanks the fact that not all iterations take the same time. If we don´t want to one thread work and another finish and don´t do anything we have to use the dynamic way. Because there will not be finished thread without work it will be faster.

For the loop it is almost the same – the same function only with another arguments which were mentioned above, until it reaches the “critical” directive – next region can execute only one thread at one moment. This is very important because without usage of this directive it can happen that the one thread compares its ‘’price’’ with the ‘’best’’ variable. But before it is the “best” variable changed the next thread compares its “price” with the “best” variable. In that case the first thread has the higher “best” variable it causes an error in compute. It can make a movement which is not best.

There is one more difference with the serial solution. The serial solution is completely deterministic. In the same position it does every time the same move. There is not any random factor. For user it is actually not advantage. He gets the same answer every time. In the same position continues the same movement. But in parallel solution it cannot be assured which will be chosen because if two moves have the same evaluation it is not ensured which one of them is computed and executed first.

In this example can be seen why parallel programming is so tricky. If the “price” variable is defined as shared there is very low probability that something will happen, error occurs. The program can be run many times and there don´t have to happen anything wrong. Error can occur in one of thousands cases. When it happens there is no possibility to replicate it. There can be lots of errors as this potential one and it is hard to notice it even in a case of a very detailed testing. This is also reason why in applications, where don´t have to be errors, is better to use the serial solution.

Single-thread Chess Application Using Minimax:				
depth:	**1**	**2**	**3**	**4**
time (ms):	81	2930	99964	around 35-times more
time (sec):	0.081	2.93	99.964	around 35-times more
time (min):	0.00135	0.0488	1.67	around 35-times more
Multi-thread AI Chess Application Using Minimax:				
depth:	**1**	**2**	**3**	**4**
time (ms):	28	805	24749	around 35-times more
time (sec):	0.028	0.805	24.749	around 35-times more
time (min):	0.0005	0.0134	0.412	around 35-times more
speed-up:	**292.1%**	**364.10%**	**403.9%**	similar to depth 3

Table 4.12 – Speed-up between serial and parallel implementation of minimax

In the *Table 4.12 – Speed-up between serial and parallel implementation of minimax* is possible to see different depending on a time execution and speed-up. For speed-up is valid the same what is above in the chapter *4.1.3 Multi-Threaded OpenMP*; if is speed-up 120% it means that parallel version is 1.2-times quicker and so on.

Depth one is the smallest speed-up – 292.1%. With the minimax's depth three is the biggest 403.9% which means more than the linear speed-up which is the very good result. How is possible to achieve more than the linear speed-up is explained in the previous chapter *4.1 Matrix Multiplication*. The reason is that with depth one the time of execution is only 28 respectively 81 milliseconds. Hence there is big influence on the cost for parallelization. The most of costs are the same. It doesn´t matter if there is depth one or higher. That is why with bigger depths come better results.

At this case program with the minimax's depth three and more is a little bit more user-friendly – user waits for move approximately less than 25 seconds and it is alright in opposite of almost 100 seconds. But power of the artificial intelligence is the same – it is not possible to calculate the higher depth than three in a reasonable time. In expecting the more powerful it would be probably quite disappointing. But this result was expected because algorithmic complexity is n^{35}. It is clear from chapter *3.2.1 Algorithmic Complexity*.

4.2.5 Multi-threaded Chess Application Using Alfabeta

In the previous chapter *4.2.4 Multi-threaded AI Chess Application Using Minimax* is parallelized the minimax algorithm. This chapter comes out with the program from the previous chapter *4.2.3 Multi-threaded Chess Application Using Minimax*. In this chapter is used another algorithm: Alfabeta which cuts out the bad moves and doesn´t compute the whole three as the minimax algorithm.

```
int chessAI::alfabeta(int depth, int alfa, int beta)
{
	int realRange, player, price;
	int best = -9990 - depth;
	int realMoveFrom[MOVESIZE];
	int realMoveTo[MOVESIZE];

	player = playerChoose();

	//return evaluation of position
	if (depth <=0) return positionValue(player);

	realRange = possibleMovement(player, realMoveFrom, realMoveTo);

	if (doMove(1,1) == 2) return best; //check for checkMate
	if (doMove(1,1) == 3) return 0; // checkfor stateMate

	for (int i = 0; i < realRange; i++)
	{
		int doMoveReturn;
		doMoveReturn = doMove(realMoveFrom[i],realMoveTo[i]);
		if (doMoveReturn > 10) promotion(doMoveReturn, 5);
		price = -alfabeta(depth-1, -beta, -alfa);
		lastMove--;

		if (price > alfa)
		{
			alfa = price;
			if (price >= beta)
			{
				return beta;
			}
		}
	}

return alfa;
}
```

Source 4.19 - Author´s implementation of alfabeta for chess artificial intelligence

In the *Source 4.19* is author´s implementation of alfabeta algorithm for the chess artificial intelligence. In comparing this implementation with the minimax implementation (in the *Source 4.10*) it is almost the same. The main difference is in inner condition “price >= beta” which actually cuts out all the bad moves.

```
int chessAI::AIMovement(int *from, int *to, int depth)
{
	int realRange, bestInRange;
	int player, price;
	int best = -10000; //We can be sure, we find better
	int realMoveFrom[MOVESIZE];
	int realMoveTo[MOVESIZE];

	//resize vector
		if (lastMove > 44)
		{
			chessboard.push_back(std::vector<int>(120));
			chessboard.push_back(std::vector<int>(120));
		}

	//choose which one play white/black
	player = playerChoose();

	//move generator
	realRange = possibleMovement(player, realMoveFrom, realMoveTo);

	//trying of movement
	for (int i = 0; i < realRange; i++)
	{
		int doMoveReturn, minimaxReturn;
		doMoveReturn = doMove(realMoveFrom[i],realMoveTo[i]);
		if (doMoveReturn > 10) promotion(doMoveReturn, 5);
		//price = -minimax(depth); for compare
		price = -alfabeta(depth, -10000, 10000);
		lastMove--;

		if (price > best)
		{
			best = price;
			*from = realMoveFrom[i];
			*to = realMoveTo[i];
		}
	}
	return best;
}
```

Source 4.20 - Author´s implementation of chess artificial intelligence with alfabeta

In the *Source 4.20* is author´s implementation of chess artificial intelligence with alfabeta algorithm. In comparing this with implementation with minimax algorithm (in the *Source 4.11*) there is only one difference. The alfabeta function is called instead of the minimax function. The alfabeta values are rewritten. The changes are certainly simpler than parallelized algorithm.

Single-thread Chess Application Using Minimax:				
depth:	1	2	3	4
time (ms):	81	2930	99964	around 35-times more
time (sec):	0.081	2.93	99.964	around 35-times more
time (min):	0.00135	0.0488	1.67	around 35-times more
Multi-thread AI Chess Application Using Minimax:				
depth:	1	2	3	4
time (ms):	28	805	24749	around 35-times more
time (sec):	0.028	0.805	24.749	around 35-times more
time (min):	0.0005	0.0134	0.412	around 35-times more
speed-up:	292.1%	364.1%	403.9%	similar to depth 3
Multi-thread Chess Application Using Alfabeta:				
depth:	1	2	3	4
time (ms):	79	1726	21608	around 12-times more*
time (sec):	0.079	1.726	21.608	around 12-times more*
time (min):	0.00132	0.0288	0.360	around 12-times more*
speed-up:	102.4%	169.7%	462.6%	1337.4%*
*Presumed values				

Table 4.13 - Speed-up between minimax and alfabeta

In the *Table 4.13* is speed-up between alfabeta implementation and implementation in previous chapters. The alfabeta algorithm is faster in higher depths comparing to the minimax algorithm. With depth one the result is almost the same. Speed-up is only 102.4%. But in the depth three is result even better than in the paralyzed minimax 462.6%. This solution is better because time critical are bigger depths. The proposed values are derived from the next chapter *4.2.6 Multi-threaded AI Chess Application Using Alfabeta* where is expected 4-times quicker execution.

4.2.6 Multi-threaded AI Chess Application Using Alfabeta

This chapter compares parallel solution of alfabeta algorithm. The source code is quite clear from the *Source 4.18 - Author´s implementation of chess multi-threads artificial* and the *Source 4.19 - Author´s implementation of alfabeta for chess artificial* . With these two previous solutions is this one very simple.

Single-thread Chess Application Using Minimax:				
depth:	**1**	**2**	**3**	**4**
time (ms):	81	2930	99964	around 35-times more
time (sec):	0.081	2.93	99.964	around 35-times more
time (min):	0.00135	0.0488	1.67	around 35-times more
Multi-thread AI Chess Application Using Minimax:				
depth:	**1**	**2**	**3**	**4**
time (ms):	28	805	24749	around 35-times more
time (sec):	0.028	0.805	24.749	around 35-times more
time (min):	0.0005	0.0134	0.412	around 35-times more
speed-up:	**292.1%**	**364.1%**	**403.9%**	similar to depth 3
Multi-thread Chess Application Using Alfabeta:				
depth:	**1**	**2**	**3**	**4**
time (ms):	79	1726	21608	around 12-times more*
time (sec):	0.079	1.726	21.608	around 12-times more*
time (min):	0.00132	0.0288	0.360	around 12-times more*
speed-up:	**102.4%**	**169.7%**	**462.6%**	1337.4%*
Multi-thread AI Chess Application Using Alfabeta:				
depth:	**1**	**2**	**3**	**4**
time (ms):	27	484	5512	64771
time (sec):	0.027	0.484	5.512	64.771
time (min):	0.00045	0.00807	0.0919	1.080
speed-up:	**297,80%**	**605.0%**	**1813.4%**	5401.7%*
*Presumed values				

Table 4.14 - Speed-up between minimax and alfabeta in serial and parallel version

In the *Table 4.14* is comparison of the minimax and the alfabeta algorithm in the serial and the parallel version. Finally it is stronger algorithm because paralyzed alfabeta algorithm is able to search in the depth four quicker than the serial minimax into the depth three. The speed-up in the depth four 5401.7% is only proposed. It is computed with presumption that minimax is 35-times slower in computations in the depth four than in the depth three. The chess program is the most user-friendly (in time of answer) from all showed.

If we would want stronger artificial intelligence in similar amount of time we need 12-times more compute performance or better algorithm. If it uses any of today´s personal computers it is not have 12-times more of compute performance so the only solution is better algorithm. This solution it is possible to divide into two possibilities:

another quicker algorithm or another quicker implementation. The really strong chess artificial intelligence combines both approaches.

4.2.7 Chess Application Conclusion

The chess application confirms few possibilities of parallel programming. The first is responsibility during the intensive computing. If this feature is not in program and program executes too many computing (is in “not responding” state) user will probably try to end the program in non-standard ways. Previous chapters prove that parallelism really helps with solving this problem.

The design of applications is very important and in case of parallel applications (especially when is used the low-level way of parallel programming) even more. In the case of an improper design it is harder to add parallelism. Debugging of parallel application is much harder than serial one. In parallel application can be hidden errors which are not almost possible to reproduce.

In case of the suitable algorithm is not problem to paralyze it. But it can happen that few-times more performance is not significant because of the algorithmic complexity (see *3.2.1 Algorithmic Complexity*). Sometime it is better solution to use another algorithm or combine both of these styles.

5 Conclusion

In this diploma thesis is defined what parallelism is. It explains a necessary theoretical background. The first part of the background is about important hardware features which allow parallelism in personal computers. Then are explained important parallel theories for this work. After this are theoretically analysed ways of parallel programming. It describes fundamental problems and divides the parallel programming into the low-level and the high-level.

The practical part of diploma thesis is divided into two parts. The first part deals with a matrix multiplication and the second part deals with a GUI (Graphical User Interface) chess application. The matrix multiplication is implemented in several different ways. These ways are compared together. There is confirmed a possible linear speed-up and also a disadvantage of Hyper-Threading technology which is not often mentioned. In this part is also confirmed that high-level ways of parallelism are easier to implement than low-level. But in the same way as low-level programming languages, low-level parallel ways have their place.

In the chess application author uses parallelism in two different ways. The first one parallelism was used for a responsible interface during intensive computing and the second one for the speed-up. Both usages of parallelism are shown on a real program. Author explains the importance of a good design and harder situation of debugging parallel application compared to the serial one. There is also confirmed that parallelism is not only one of the ways of programming and sometimes other ways bring a better solution.

Author fulfilled all goals, explained, showed and confirmed all proposed possibility of parallelism. The main contribution of work is a practical demonstration and the contribution of parallelism with comparison of the serial way. These contributions are supported with the necessary theoretical background.

6 Bibliography

[1] "Architecture of Intel 80286," [Online]. Available: http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT-KANPUR/microcontrollers/micro/ui/Course_home4_32.htm. [Accessed 8 11 2011].

[2] "Intel 80386 - A 32-bit Microprocessor with Memory Paging Facility," [Online]. Available: http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT-KANPUR/microcontrollers/micro/lecture33/lec33_1.htm. [Accessed 8 11 2011].

[3] "Intel® Pentium® 4 Processor supporting HT Technology 3.06 GHz, 512K Cache, 533 MHz FSB," [Online]. Available: http://ark.intel.com/products/27499. [Accessed 10 11 2011].

[4] "SPARC international, Inc. - Technical Documents," [Online]. Available: http://www.sparc.org/specificationsDocuments.html. [Accessed 18 November 2011].

[5] "Intel," [Online]. Available: http://software.intel.com/file/3914. [Accessed 15 November 2011].

[6] L. Chao, "Intel Technology Journal," 14 February 2002. [Online]. Available: http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf. [Accessed 10 11 2011].

[7] "Hyper-Threading detailně," 22 November 2002. [Online]. Available: http://www.pctuning.cz/ilustrace/HyperThreading/ideal%%20HT.gif. [Accessed 1521 January 2010].

[8] "Oracle Solaris," [Online]. Available: http://www.oracle.com/us/products/servers-storage/solaris/overview/index.html. [Accessed 25 November 2011].

[9] D. Gove, Multicore Application Programming, Crawfordsville: Pearson Education, Inc., 2010.

[10] "Svět hardware," [Online]. Available: http://www.svethardware.cz/sh/media.nsf/v/F8E28DBA4EA6C1DCC1256FE600

6F88. [Accessed 30 11 2011].

[11] D. V., Architektura a programování paralelních systémů, Brno: VUTIUM, 2004.

[12] P. Jakub Černý, “Základní grafové algoritmy,” [Online]. Available: http://kam.mff.cuni.cz/~kuba/ka/. [Accessed 2 12 2011].

[13] Daniel, 2 May 2009. [Online]. Available: http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/AmdahlsLaw.svg/2000px-AmdahlsLaw.svg.png. [Accessed 5 December 2011].

[14] D. Gove, “Multicore Application Programming: Identifying Opportunities for Parallelism,” 9 November 2010. [Online]. Available: http://www.informit.com/content/images/chap3_9780321711373/elementLinks/gove_3-12.jpg. [Accessed 5 December 2011].

[15] “POSIX.1, POSIX.1b and POSIX.1c,” LynuxWorks™, Inc., [Online]. Available: http://www.lynuxworks.com/products/posix/posix2.php3. [Accessed 10 Decenmer 2011].

[16] R. Johnson, “POSIX Threads (pthreads) for win32,” [Online]. Available: http://sourceware.org/pthreads-win32/. [Accessed 10 December 2011].

[17] “Class Thread,” [Online]. Available: http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html. [Accessed 14 December 2011].

[18] http://qt-project.org/doc/qt-4.8/qthread.html, “QThread Class Reference,” [Online]. Available: http://qt-project.org/doc/qt-4.8/qthread.html. [Accessed 14 December 2011].

[19] “Thread Support in Qt,” Qt Project Hosting, [Online]. Available: http://qt-project.org/doc/qt-4.8/threads.html. [Accessed 14 December 2011].

[20] “Automatic Parallelization with Intel® Compilers,” [Online]. Available: http://software.intel.com/file/39655. [Accessed 2011 December 21].

[21] J. Reinders, “Parallelism as a First Class Citizen in C and C++, the time has come,” 9 August 2011. [Online]. Available: http://software.intel.com/en-us/blogs/2011/08/09/parallelism-as-a-first-class-citizen-in-c-and-c-the-time-has-come/. [Accessed 21 December 2011].

[22] B. Barney, "OpenMP," [Online]. Available: https://computing.llnl.gov/tutorials/openMP/. [Accessed 21 December 2011].

[23] "Concurrent Programming," [Online]. Available: http://qt-project.org/doc/qt-4.8/threads-qtconcurrent.html. [Accessed 21 December 2011].

[24] "Compare Intel® Products," [Online]. Available: http://ark.intel.com/compare/52213,33924,50176,27482. [Accessed 1 January 2012].

[25] L. V. Allis, Searching for Solutions in Games and Arti, University of Limburg in Maastricht, 1994.

[26] "Qt library 4.8 | Documentation | Qt Developer Network," [Online]. Available: http://qt-project.org/doc/qt-4.8/. [Accessed 2012].

[27] J. Blanchette and M. Summerfield, C++ GUI Programming with Qt 4, Second Edition, New Jersey: Prentice Hall, 2008.

[28] S. Prata, Mistrovství v C++, Praha: Computer Press, 2007.

[29] J. L. Bradley L. Jones, Naučte se C++ za 21 dní, Praha: Computer Press, 2007.

[30] H.-P. Messmer and K. Dembowski, Velká kniha hardware, Brno: CP Books, 2005.

7 Supplements

7.1 List of images

Picture 3.1 – Ideal function of Hyper-Threading technology [5] 8
Picture 3.2 - Ideal function of Hyper-Threading technology [7] 9
Picture 3.3 – Single and multicore processors in compare without HeatSpreader [10].... 11
Picture 3.4 – Scaling with different parallel portion according Amdahl´s Law [13] 15
Picture 3.5 - Scaling with different parallel portion in real [14] 16
Picture 4.6 – Chess program is in “not responding” state 47
Picture 4.7 - Chess program which is “not responding” and was partially overlaid 48
Picture 4.8 - Chess program is responsible even during intensive processing. 52

7.2 List of tables

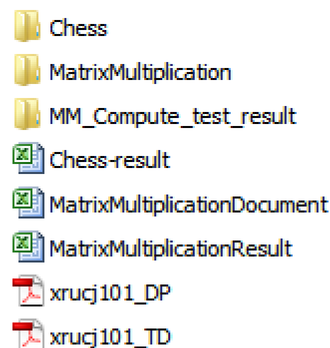
Table 3.1 – Time complexity of different algorithm with presumption 10^9 execute operations for second [12] 13
Table 4.2 – C++ matrix multiplication time of execution in milliseconds 23
Table 4.3 - Result of OpenMP matrix multiplication implementation 26
Table 4.4 - Results of POSIX Thread matrix multiplication implementation 29
Table 4.5 – Result of Windows threading matrix multiplication implementation 31
Table 4.6 - Result of QThread matrix multiplication implementation 33
Table 4.7 – Java matrix multiplication time of execution in milliseconds 35
Table 4.8 - Results of Java multi-threaded matrix multiplication implementation 37
Table 4.9 – Compare of results of different implementations 38
Table 4.10 - Disadvantage of HT, comparison with the same processor with HT off 40
Table 4.11 – Time of single-threaded minimax computing 46
Table 4.12 – Speed-up between serial and parallel implementation of minimax 56
Table 4.13 - Speed-up between minimax and alfabeta 59
Table 4.14 - Speed-up between minimax and alfabeta in serial and parallel version 60

7.3 List of source codes

Source 3.1 – C++ source code where can occurs data race 17
Source 4.2 – Matrix multiplication C++ serial implementation 23
Source 4.3 - Matrix multiplication C++ OpenMP implementation 24
Source 4.4 -Matrix multiplication C++ POSIX Thread implementation. 28

Source 4.5 - Matrix multiplication C++ Windows Threading implementation.................... 30
Source 4.6 - Matrix multiplication C++ Qt framework implementation............................... 32
Source 4.7 - Matrix multiplication Java serial implementation 34
Source 4.8 - Matrix multiplication Java multi-threaded implementation............................ 36
Source 4.9 – Infinite loop for fully workloaded processor.. 40
Source 4.10 – Author´s implementation of minimax for chess artificial intelligence 43
Source 4.11 – Author´s implementation of chess artificial intelligence............................ 44
Source 4.12 – Implementation of chessboard 10*12 45
Source 4.13 – Call of the AIMovement function (Source 4.11) from GUI class 46
Source 4.14 – Graphical user interface in another thread 49
Source 4.15 – The single-threaded “moveBack()” function from the chapter 0................. 51
Source 4.16 – Multi-threaded implementation of the “moveBack()” function in this chapter .. 51
Source 4.17 – The ideal multi-threaded implementation of the “moveBack()” function ... 51
Source 4.18 - Author´s implementation of chess multi-threads artificial intelligence 54
Source 4.19 - Author´s implementation of alfabeta for chess artificial intelligence 57
Source 4.20 - Author´s implementation of chess artificial intelligence with alfabeta 58

7.4 Attached Files



In the “Chess” directory are all executable instances of the chess program with necessary dynamic libraries. In the “MatrixMultiplication” directory are all executable instances of matrix multiplication programs with the source code and necessary libraries. In “MM_Compute_test_result” are raw output data.

7.5 Complete Results of Matrix Multiplication

Pentium 4 650	**serial**	**multi-threaded**				
		1	**2**	**4**	**8**	**x**
1) OpenMP average:	12265.6	14550.8	14455.5	14491.3	14485.85	14544.6
1) OpenMP speed-up:	100.00%	118.63%	117.85%	118.15%	118.10%	118.58%
1) POSIX average:	12265.6	13089.15	13096.85	13101.5	13098.55	13090.5
1) POSIX speed-up:	100.00%	106.71%	106.78%	106.81%	106.79%	106.73%
1) Qt average:	12265.6	13071.15	13038.25	13051.6	13057.25	13068.7
1) Qt speed-up:	100.00%	106.57%	106.30%	106.41%	106.45%	106.55%
1) Window average:	12265.6	13089.8	13092.9	13103.05	13103.85	13095.25
1) Window speed-up:	100.00%	106.72%	106.74%	106.83%	106.83%	106.76%
1) Java average:	18321.85	17530.9	17527.5	17511.25	17535.55	17540.7
1) Java speed-up:	100.00%	104.51%	104.53%	104.63%	104.48%	104.45%
2) OpenMP average:	12331.15	14637.5	10825.8	10779.05	11162.25	10824.95
2) OpenMP speed-up:	100.00%	118.70%	87.79%	87.41%	90.52%	87.79%
2) POSIX average:	12331.15	13156.3	9549.2	9571.15	9607.95	9571.05
2) POSIX speed-up:	100.00%	106.69%	77.44%	77.62%	77.92%	77.62%
2) Qt average:	12331.15	13104.05	9578	9584.15	9585.05	9589.8
2) Qt speed-up:	100.00%	106.27%	77.67%	77.72%	77.73%	77.77%
2) Window average:	12331.15	13146.8	9573.5	9559.4	9617.9	9571.25
2) Window speed-up:	100.00%	106.61%	77.64%	77.52%	78.00%	77.62%
2) Java average:	18487.15	17613.75	13133.65	13145.25	13164.15	13145.6
2) Java speed-up:	100.00%	104.96%	140.76%	140.64%	140.44%	140.63%
3) OpenMP average:	12264.85	14468.05	14392.85	14412.55	14421.9	14461.75
3) OpenMP speed-up:	100.00%	117.96%	117.35%	117.51%	117.59%	117.91%
3) POSIX average:	12264.85	13128	13353.05	13739.1	14025.7	13108.75
3) POSIX speed-up:	100.00%	107.04%	108.87%	112.02%	114.36%	106.88%
3) Qt average:	12264.85	13216.5	12961	12982.15	12986.6	13000
3) Qt speed-up:	100.00%	107.76%	105.68%	105.85%	105.88%	105.99%
3) Window average:	12264.85	13207.7	13360.15	13752.55	14087.3	13166.35
3) Window speed-up:	100.00%	107.69%	108.93%	112.13%	114.86%	107.35%
3) Java average:	18248.4	17535.35	17753.9	17899.75	18311.05	17594.55
3) Java speed-up:	100.00%	104.07%	102.79%	101.95%	99.66%	103.72%
4) OpenMP average:	14782.8	19983.75	10872.65	10752.9	11070.9	10815
4) OpenMP speed-up:	100.00%	135.18%	73.55%	72.74%	74.89%	73.16%
4) POSIX average:	14782.8	17007.9	9735.1	9897.7	9917.25	9596.85
4) POSIX speed-up:	100.00%	115.05%	65.85%	66.95%	67.09%	64.92%
4) Qt average:	14782.8	16927.45	9603.9	9644	9595.3	9598.35
4) Qt speed-up:	100.00%	114.51%	64.97%	65.24%	64.91%	64.93%
4) Window average:	14782.8	16926.65	9600.95	9946.15	9867.15	9617.2
4) Window speed-up:	100.00%	114.50%	64.95%	67.28%	66.75%	65.06%
4) Java average:	26727.15	25155.9	13178	13298.6	13585.05	13228.35
4) Java speed-up:	100.00%	106.25%	202.82%	200.98%	196.74%	202.04%
1) without Hyper-Threading (HT), 2) with HT, 3) infinite loop, 4) infinite loop with HT						

Core i7-2600	serial	multi-threaded				
		1	2	4	8	x
1) OpenMP average:	6257.9	6014.4	3003.85	1507	1602.05	1498.65
1) OpenMP speed-up:	100.00%	96.11%	48.00%	24.08%	25.60%	23.95%
1) POSIX average:	6257.9	6181.9	3056.65	1516.75	1529.95	1528.65
1) POSIX speed-up:	100.00%	98.79%	48.84%	24.24%	24.45%	24.43%
1) Qt average:	6257.9	6239.9	3099.65	1563.55	1581.4	1564.65
1) Qt speed-up:	100.00%	99.71%	49.53%	24.99%	25.27%	25.00%
1) Window average:	6257.9	6105	3026.7	1519.75	1531.95	1531.15
1) Window speed-up:	100.00%	97.56%	48.37%	24.29%	24.48%	24.47%
1) Java average:	6602.35	6339.2	3134.95	1568.55	1567.55	1568.5
1) Java speed-up:	100.00%	104.15%	210.60%	420.92%	421.19%	420.93%
2) OpenMP average:	6261.5	5937.25	3003.2	1536.65	1559.65	1545.6
2) OpenMP speed-up:	100.00%	94.82%	47.96%	24.54%	24.91%	24.68%
2) POSIX average:	6261.5	6029.2	3051.45	1606.5	1585.2	1590.95
2) POSIX speed-up:	100.00%	96.29%	48.73%	25.66%	25.32%	25.41%
2) Qt average:	6261.5	6229.8	3116.6	1575.55	1593.15	1594.15
2) Qt speed-up:	100.00%	99.49%	49.77%	25.16%	25.44%	25.46%
2) Window average:	6261.5	6100.6	3052.15	1540.05	1584.4	1584.2
2) Window speed-up:	100.00%	97.43%	48.74%	24.60%	25.30%	25.30%
2) Java average:	100.00%	102.64%	205.15%	406.58%	395.20%	395.25%
2) Java speed-up:	105.84%	99.95%	50.03%	26.49%	26.03%	25.92%
3) OpenMP average:	6249.1	5947.6	2998.6	1513.8	1602.15	1512
3) OpenMP speed-up:	100.00%	95.18%	47.98%	24.22%	25.64%	24.20%
3) POSIX average:	6249.1	6097.55	3025.25	3131.95	2369.95	3107.25
3) POSIX speed-up:	100.00%	97.57%	48.41%	50.12%	37.92%	49.72%
3) Qt average:	6249.1	6228.1	3125.05	1564.45	1565.35	1584.35
3) Qt speed-up:	100.00%	99.66%	50.01%	25.03%	25.05%	25.35%
3) Window average:	6249.1	6096.2	3042.2	3311	3049.7	3109.2
3) Window speed-up:	100.00%	97.55%	48.68%	52.98%	48.80%	49.75%
3) Java average:	6572.45	6244.65	5172.35	4453	3891.35	3511.25
3) Java speed-up:	100.00%	105.25%	127.07%	147.60%	168.90%	187.18%
4) OpenMP average:	6089.7	7055.45	3553.15	2348.1	1570.6	1572.65
4) OpenMP speed-up:	100.00%	115.86%	58.35%	38.56%	25.79%	25.82%
4) POSIX average:	6089.7	6191.8	3142.8	2310.9	2272.2	2327.35
4) POSIX speed-up:	100.00%	101.68%	51.61%	37.95%	37.31%	38.22%
4) Qt average:	6089.7	6241	3162.95	1586.6	1624.05	1586.25
4) Qt speed-up:	100.00%	102.48%	51.94%	26.05%	26.67%	26.05%
4) Window average:	6089.7	6211.6	3102.1	2087.25	2353.25	2353.55
4) Window speed-up:	100.00%	102.00%	50.94%	34.28%	38.64%	38.65%
4) Java average:	9357.6	8930.05	4531.8	2748	3528.55	3456.05
4) Java speed-up:	100.00%	104.79%	206.49%	340.52%	265.20%	270.76%
1) without Hyper-Threading (HT), 2) with HT, 3) infinite loop, 4) infinite loop with HT						

Pentium P6200	serial	multi-threaded				
		1	2	4	8	x
1) OpenMP average:	14803.95	17810.3	8576.35	8721.85	8599.8	9035.25
1) OpenMP speed-up:	100.00%	120.31%	57.93%	58.92%	58.09%	61.03%
1) POSIX average:	14803.95	16503.5	8130.85	8208.8	8291.25	7871.15
1) POSIX speed-up:	100.00%	111.48%	54.92%	55.45%	56.01%	53.17%
1) Qt average:	14803.95	15785.6	7737.35	8424.35	7953	8035.45
1) Qt speed-up:	100.00%	106.63%	52.27%	56.91%	53.72%	54.28%
1) Window average:	14803.95	17711.4	8537.05	8539.5	8678.1	8600.2
1) Window speed-up:	100.00%	119.64%	57.67%	57.68%	58.62%	58.09%
1) Java average:	21222.65	20241.4	9918.85	9783.1	9547.55	9712.65
1) Java speed-up:	100.00%	104.85%	213.96%	216.93%	222.28%	218.51%
2) OpenMP average:	16295.45	17830.4	9119.95	8819	9163.35	9056.8
2) OpenMP speed-up:	100.00%	109.42%	55.97%	54.12%	56.23%	55.58%
2) POSIX average:	16295.45	17066.6	8376.6	8513.65	8661.2	8425.4
2) POSIX speed-up:	100.00%	104.73%	51.40%	52.25%	53.15%	51.70%
2) Qt average:	16295.45	16232.4	8170.8	7833.5	8136.3	8090.35
2) Qt speed-up:	100.00%	99.61%	50.14%	48.07%	49.93%	49.65%
2) Window average:	16295.45	16807.1	8419.2	8661.5	8748.7	8814.7
2) Window speed-up:	100.00%	103.14%	51.67%	53.15%	53.69%	54.09%
2) Java average:	21109.05	20335.25	10161.05	11666.4	10604.55	10226.05
2) Java speed-up:	100.00%	103.81%	207.74%	180.94%	199.06%	206.42%
Core 2 Quad Q9550	serial	multi-threaded				
		1	2	4	8	x
1) OpenMP average:	7140.55	7838.6	3927.55	1974.25	1997.65	1976.2
1) OpenMP speed-up:	100.00%	109.78%	55.00%	27.65%	27.98%	27.68%
1) POSIX average:	7140.55	6303.25	3161.1	1593.05	1601	1594.95
1) POSIX speed-up:	100.00%	88.27%	44.27%	22.31%	22.42%	22.34%
1) Qt average:	7140.55	6273.8	3148.4	1595.25	1591.35	1590.1
1) Qt speed-up:	100.00%	87.86%	44.09%	22.34%	22.29%	22.27%
1) Window average:	7140.55	6295.55	3159.6	1596.6	1602.05	1595.15
1) Window speed-up:	100.00%	88.17%	44.25%	22.36%	22.44%	22.34%
1) Java average:	10674.15	10032.7	5040.9	2684	2593.25	2669.1
1) Java speed-up:	100.00%	106.39%	211.75%	397.70%	411.61%	399.92%
2) OpenMP average:	7156.15	7794.95	3953.05	2043.9	1991.75	2256.55
2) OpenMP speed-up:	100.00%	108.93%	55.24%	28.56%	27.83%	31.53%
2) POSIX average:	7156.15	6333.6	3203.1	3052.1	2401.85	3160.7
2) POSIX speed-up:	100.00%	88.51%	44.76%	42.65%	33.56%	44.17%
2) Qt average:	7156.15	6273.55	3175.35	1631.95	1710.1	1659.8
2) Qt speed-up:	100.00%	87.67%	44.37%	22.80%	23.90%	23.19%
2) Window average:	7156.15	6307.65	3214.2	3050.25	2501.4	2883.5
2) Window speed-up:	100.00%	88.14%	44.92%	42.62%	34.95%	40.29%
2) Java average:	10575.55	10030.35	5211.85	4769.9	4116.25	4965.45
2) Java speed-up:	100.00%	105.44%	202.91%	221.71%	256.92%	212.98%
1) standard setting, 2) with infinite loop						