MULTI-CORE PATTERN

Author: Volodymyr Bendiuga
vba10001@student.mdh.se

Carried out at: Etteplan Industry AB
Advisor at MDH: Frank Lüders
Advisor at Etteplan: Björn Andersson
Examiner: Ivica Crnkovic

December 5, 2012
A lot of research has been done in area of multicore systems. And yet it is not enough. Legacy code is not ready yet to be put on to multicore hardware. It takes huge amount of time to develop appropriate methods and practices for proper parallelization of legacy software systems, and also enormous amount of time to perform actual parallelization.

This thesis work solves one of many problems in multicore world. Throughout 6 months time pattern and synchronization technique has been developed to target parallel execution in true multicore environment. Two compilers have been tested for parallelization and optimization: Microsoft Visual studio 2011 and GNU GCC/G++ together with QT Creator, on both Windows and Linux platforms. As a theoretical part of the thesis, some research has been done in area of parallelizing compilers currently being developed. Unfortunately tests could not be performed due to expensive license for compilers, such as Intel C/C++ compiler.

Pattern together with synchronization techniques has been developed and tested for three different platforms: Windows, Linux and VxWorks. Synchronization technique is quite unique itself, as it can solve any kind of synchronization problems, from very simple ones, to most complex. Although it is intended for complex problems, as there are other patterns that are more optimized for simple synchronization problems.

More specifically synchronization technique solves problems in situations where there are many parallel objects and many shared resources, and each object tries to access many resources at once.

No existing solutions that target this problem have been found, and therefore this has become an objective for this thesis work.
ACKNOWLEDGEMENT

This thesis came into existence throughout 6 months of work. Different ideas, thoughts and suggestions have been considered. Many of them have been implemented and tested and others thrown away due to different reasons.

The result of this thesis has been born thanks to Frank Lüders – university supervisor, Björn Andersson and Dag Lindahl – supervisors from Etteplan. I want to express thanks to all of them, for their support, ideas and suggestions. Also I want to express thanks to everyone who directly or indirectly positively influenced me during work.

Special thanks I want to express to my girlfriend Khrystyna, who has been remotely supporting me all the time during my work on thesis.

Västerås, December 2012
Volodymyr Bendiuga
## NOMENCLATURE

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCURRENT</td>
<td>Name of the design pattern that has been developed within this thesis</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit, central processor, processor</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit, graphics processor, vector processor</td>
</tr>
<tr>
<td>STARVATION</td>
<td>a process when processing unit is waiting for long lasting memory access (computer memory is much slower than CPUs)</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture, architecture specific to modern GPUs developed by Nvidia</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction Single Data</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple Instructions Single Data</td>
</tr>
<tr>
<td>MIMS</td>
<td>Multiple Instructions Multiple Data</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter Process Communication</td>
</tr>
<tr>
<td>Od</td>
<td>MS Visual studio compiler switch for disabling optimization</td>
</tr>
<tr>
<td>Ox</td>
<td>MS Visual studio compiler switch for full optimization</td>
</tr>
<tr>
<td>O2</td>
<td>MS Visual studio compiler switch for optimization for speed</td>
</tr>
</tbody>
</table>
## CONTENTS

### Chapter 1: INTRODUCTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 ADDRESSED PROBLEM</td>
<td>1</td>
</tr>
<tr>
<td>1.2 CONTRIBUTION</td>
<td>1</td>
</tr>
<tr>
<td>1.3 METHODOLOGY</td>
<td>2</td>
</tr>
<tr>
<td>1.4 OUTLINE</td>
<td>2</td>
</tr>
<tr>
<td>1.5 OBJECTIVE</td>
<td>2</td>
</tr>
<tr>
<td>1.6 PROBLEM FORMULATION</td>
<td>3</td>
</tr>
<tr>
<td>1.7 LIMITATIONS</td>
<td>6</td>
</tr>
</tbody>
</table>

### Chapter 2: OVERVIEW OF MULTICORE PLATFORMS & TECHNOLOGIES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 TYPES OF MULTICORE PROCESSORS</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 General Knowledge</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 General Purpose Multicore Processors</td>
<td>9</td>
</tr>
<tr>
<td>2.1.3 Graphical Multicore Processors with general Purpose Processing Capabilities</td>
<td>9</td>
</tr>
<tr>
<td>2.2 PARALLEL COMPUTER ARCHITECTURES</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 SISD – Single Instruction Single Data</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 SIMD – Single Instruction Multiple Data</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 MISD – Multiple Instruction Single Data</td>
<td>11</td>
</tr>
<tr>
<td>2.2.4 MIMD – Multiple Instruction Multiple Data</td>
<td>12</td>
</tr>
<tr>
<td>2.3 TYPES OF MULTIPROCESSING MODELS</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 AMP – Asymmetric Multiprocessing</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 SMP – Symmetric Multiprocessing</td>
<td>13</td>
</tr>
<tr>
<td>2.4 MEMORY ARCHITECTURE SUITABLE FOR MULTIPROCESSING</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1 UMA – Uniform Memory Architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.4.2 NUMA - Non Uniform Memory Architecture</td>
<td>15</td>
</tr>
<tr>
<td>2.5 PARALLEL PROGRAMMING MODELS</td>
<td>16</td>
</tr>
<tr>
<td>2.5.1 Data-Based Parallel Model</td>
<td>16</td>
</tr>
<tr>
<td>2.5.2 Task-Based Parallel Model</td>
<td>16</td>
</tr>
<tr>
<td>2.6 PARALLEL TECHNOLOGIES</td>
<td>17</td>
</tr>
<tr>
<td>2.6.1 TPL – Task Parallel Library</td>
<td>17</td>
</tr>
<tr>
<td>2.6.2 MPI</td>
<td>18</td>
</tr>
<tr>
<td>2.6.3 Pthreads</td>
<td>18</td>
</tr>
<tr>
<td>2.6.4 OpenCL</td>
<td>18</td>
</tr>
<tr>
<td>2.6.5 Cilk++</td>
<td>19</td>
</tr>
<tr>
<td>2.6.6 OpenMP</td>
<td>19</td>
</tr>
<tr>
<td>2.6.7 AMD ATI Stream</td>
<td>19</td>
</tr>
<tr>
<td>2.6.8 NVIDIA CUDA</td>
<td>20</td>
</tr>
<tr>
<td>2.7 SCHEDULING POLICIES FOR MULTICORE SYSTEMS</td>
<td>20</td>
</tr>
<tr>
<td>2.7.1 Global</td>
<td>20</td>
</tr>
<tr>
<td>2.7.2 Partitioned</td>
<td>20</td>
</tr>
<tr>
<td>2.8 MULTICORE SYNCHRONIZATION PROTOCOLS</td>
<td>21</td>
</tr>
<tr>
<td>2.9 SYNCHRONIZATION PRIMITIVES FOR MULTICORE SYSTEMS</td>
<td>21</td>
</tr>
</tbody>
</table>
Chapter 1
INTRODUCTION

1.1  ADDRESSED PROBLEM

Design Pattern Concurrent together with synchronization technique addresses big set of problems that can be often met in real world. As an example, problems where there are many parallel objects and many shared resources and each object is willing to access a set of shared resources concurrently, special synchronization technique is needed. Unfortunately there is no specific technique which would be able to solve this problem. At least, none have been found within this master thesis.

The problem described above is huge, but it shows just a part of much bigger problem, where for instance could happen that concurrent objects need to change resources that they use to other ones. This case requires specific synchronization logics.

Another example could be that during program execution there are multiple objects and multiple resources where each object accesses set of resources. Somewhere at runtime new objects could come and ask for resources they need. This case should be also handled somehow.

1.2  CONTRIBUTION

With the part of the work that I have done related to design pattern Concurrent, I have contributed to a set of existing design patterns for multicore systems and also to a set of synchronization techniques that are used to synchronize tasks running on different cores.

Contribution is quite huge as no similar existing patterns and synchronization techniques have been found within the time spent on this thesis.

The problem that is targeted, or a set of problems, to be more precise, is not targeted by any other existing synchronization techniques, which have been discovered during master thesis, which gives a good value to developed solution.

Concurrent design pattern can be used in future as a technique to help parallelize sequential applications and of course to build new ones.

Uniqueness of contribution lies in the synchronization algorithm, which is capable of handling multiple concurrent objects that access concurrently set of shared resources. Algorithm avoids all types of synchronization problems that can happen, including deadlocks and starvation.

Design pattern, which is a wrapper class for algorithm, was created mainly for usability reasons. Algorithm is embedded in design pattern, and provides interface to objects.
Programmers can use design pattern in existing applications as well as new applications. Small changes are required in existing applications in order to embed design pattern.

1.3 METHODOLOGY

In order to find solution to problems specified in problem formulation I will make a research firstly.

As for the design pattern and synchronization technique, I will look up for existing patterns in order to find out all or most design patterns and problems that they target.

Research must be conducted to find out information about existing compilers and their ability to optimize and parallelize code.

Synchronization problems that this thesis is targeting are very complex; therefore I will try to develop synchronization technique by small and understandable parts. Each small part will be targeting specific problem related to general synchronization.

I will compose complete synchronization technique from many small pieces, when they all will be implemented.

Tests will be performed on different platforms for different implementations in order to ensure that algorithm and pattern work well.

1.4 OUTLINE

The following is a short description of what you will find in the next chapters:

- **Chapter 1**: Remaining part of this chapter gives an overview of objective for this thesis work and also explains problem formulation with some examples.
- **Chapter 2**: In this chapter there is an overview of multicore platforms and technologies that are quite popular and widely used throughout the world.
- **Chapter 3**: Detailed information about results of this thesis work can be found in this chapter.
- **Chapter 4**: Strong and weak points of Concurrent design pattern are discussed.
- **Chapter 5**: Conclusion is drawn in this chapter summarizing achieved results. Possible improvements to the pattern and also testing on proper multicore hardware are discussed.

1.5 OBJECTIVE

The following is an objective of this thesis:

- **Is it necessary to adapt applications manually when running them on multicore processor or does compiler do the job for you?**
  
  a. Make sequential application which should contain some algorithms (one or more) that can theoretically be parallelized;
  
  b. Compile application in: Windows, using Microsoft Visual Studio; Linux, using GNU Compiler for C/C++;
  
  c. Profile application in order to figure out its performance and heavy parts; note and visualize results;
d. Manually parallelize the same sequential application according to good practice, using some parallel specification (one or more, could be OpenMP, Pthreads, MPI);  
e. Compile application in Windows and Linux using MS Visual Studio and GNU Compiler for C/C++ respectively, with relevant specifications (OpenMP, Pthreads, MPI, etc);  
f. Profile application to see possible speed ups, usage of CPU; note and visualize results;  
g. Run both sequential and parallel applications on machines with different numbers of cores; note execution time; represent, using charts, difference in execution times and utilization of cores;  

- If sequential applications need to be adapted for multicore processors, how is it done in the easiest/fastest/best way?  
  a. Provide description of current status of research and development of parallelizing compilers (there are a couple of them, being developed in USA, and NEDO Advanced Parallelizing Compiler project, being developed in Japan by Waseda University in cooperation with Fujitsu, Hitachi and other huge Japanese companies);  
  b. Describe the best practices of parallelization; reason about different parallel platforms, technologies, specifications; describe possible ways of parallelization;  
- Develop new design pattern for synchronization / adjust Etteplan’s existing pattern, if possible.  
- General overview of the information regarding programming multi core systems.  

Developing a design pattern for synchronization is a main task of this thesis work, which requires lots of research on the beginning, and also development. This is what most of the time will be spent on. Investigation of compilers and auto parallelization is the second important task.

1.6 PROBLEM FORMULATION  
Multi core world is not very new to us. Most of desktop, laptop and server computers contain multicore hardware for already a couple of years. And year by year, number of cores increases. But the problem is that applications, that were built for single core processors do not take advantage of underlying multicore hardware. In some cases, especially in embedded world, those applications cannot even run on parallel hardware, mainly because of synchronization issues.  

Here the problem forks, making two:  
1. Multithreaded applications written for single core processors might fail while running on multicore processors.
2. Sequential applications, containing enormous amount of lines of code written must be somehow adapted, i.e. parallelized in order to run on multicore hardware.

These are of course very big issues that this thesis will not solve completely. But some small part can definitely be targeted and solved.

It is well known that there is ongoing development of parallelizing compilers. Many companies strive to make their compilers capable of automatic parallelization. But, unfortunately, they have come to the conclusion that right now it is extremely difficult to implement it. Compilers become better and smarter, but they are very far from what we want them to be.

Means of synchronization that we have now like mutexes, semaphores, critical sections are relatively quite low level and help us to solve some smaller problems easily. But problem arises when structure of software system gets complicated, with many independent software parts or components and shared resources. Then it is difficult to track all resources and objects, and programmers might forget to either lock or release semaphores. Even though there are many existing solutions to some common synchronization problems, programmers still tend to make many mistakes related to synchronization.

Synchronization patterns targeting particular cases have been developed. They are a bit easier to use and can help solve more complex problems with less efforts and less errors. Another problem is that there are not enough of such design patterns, and they do not cover all problems.

From general knowledge provided above, we can formulate concrete problems that thesis work is going to target. It might sound a bit unusual, but this thesis targets not just one problem, but a few different ones, related to testing, development and research.

Problems are as following:

- How good compilers are in terms of parallelization? To what extent they can parallelize applications? Is there anything they cannot parallelize for some reasons?
- How good compiler optimization is comparing to manual optimization? How good compiler optimization is comparing to manual parallelization? What is the difference between Linux and Windows compilers in terms of optimization and parallelization? Which platform gives better results and under what conditions?
- Currently there is no synchronization technique described or pattern developed that solves synchronization problem with many concurrent objects and many shared resources, where each object is trying to access many resources at a time in a way it avoids starvation and deadlock situations. Moreover, the number of concurrent objects may vary, and the number of shared resources may vary also. During lifecycle of application, new concurrent objects might come and ask for access to resources, or new resources might arrive. One more very important moment is that objects might want to change the amount of resources they access, or shift to other resources. All this depicts quite complicated synchronization problem. But not only. Solution might differ from case to case, depending on the number of resources and parallel objects. Therefore design pattern is also needed to simplify the use of such synchronization technique and increase maintainability, portability, usage, testability, reliability and other important attributes. Design pattern should be easy to understand for average programmer.
Solutions to common IPC problem called Dining Philosophers solves situation where there is predefined number of concurrent objects and number of resources should correspond to the number of concurrent objects, or philosophers. Each philosopher can access only 2 shared resources that are nearby him, and no other. Philosopher can never change number of resources he accesses.

The solution to above problem of Dining Philosophers is quite limited and therefore cannot be used massively as other synchronization techniques and patterns can.

New synchronization technique and design pattern must be developed to target problem number 3.

Here are some examples of problem scenarios that have no ready solutions.

![Figure 1. Problem scenario, example 1](image)

In this case there are 5 concurrent objects and 8 shared resources. Object1 to Object 4 each access 4 different resources at once, and object 0 wants to access all 8 resources at a time. Now imagine a situation where all this objects run each on its own core, truly in parallel and continuously asking for resources, performing some calculations on them, and releasing them.

Figure 1 depicts an example of problem scenario that does not have general solution and consequently must be solved every time it is met, and solution to it will vary every time, as number of objects and resources will vary also.

Another example of problem scenario, depicted in Figure 2 below has 7 objects and 6 resources. Objects 0 to 5 each accesses two neighboring resources at a time, just as same as in dining philosophers problem, and object 6 accesses all 6 resources at a time. Again let’s imagine that all of them run truly in parallel and continuously asking for resources. It becomes obvious that object 6 will never get a chance to run. This can be explained by the
fact that there will always be a few objects running simultaneously at a time, not allowing object 6 to access resources.

Figure 2. Problem scenario, example 2

No general solution exists for this kind of problem. And there are much more of such problem scenarios that compose a major problem for world of parallel computation.

Some common synchronization technique must be developed to target most or all problem scenarios, remaining understandable and relatively easy to use.

1.7 LIMITATIONS

Limitations for testing compilers:

- Only Microsoft Visual Studio 2011 and GNU GCC/G++ compiler for C/C++ together with QT Creator will be tested
- Windows and Linux are the only platforms for testing
• Tests are going to be done on desktop and laptop computers with dual core processors, and also computers connected through the network

**Limitations for pattern and synchronization technique:**

• Implementation is going to be done in C++ and usage is most suitable for C/C++
• Intended to be used in object oriented environment
• No appropriate multicore hardware for testing pattern together with synchronization technique
• Tests will be performed only on Windows, Linux and Tornado for VxWorks (Sequential execution)
Chapter 2

OVERVIEW OF MULTICORE PLATFORMS & TECHNOLOGIES

Multi core world is quite rich on different parallel technologies, types of processors, and programming languages with support for parallel programming and so forth. Consequently before going into planning or programming one should investigate about those technologies, and decide which one is better to use for particular case.

This section contains general information about some popular parallel technologies, different types of multi core processors, multiprocessing models, languages, etc.

2.1 TYPES OF MULTICORE PROCESSORS

There are different types of multicore processors. Generally speaking, we can divide all processors in two groups:

- General purpose processors or CPUs
- Graphics Processors or GPUs

Nowadays there is an intersection between both groups. There are particular tasks that can be done quite well by both groups of processors. But in the past two groups were completely separated. One was intended for general purpose computation whereas the other one for graphical computations.

Data-intensive computations can be done by both groups of processors, although GPUs tend to be much faster in this kind of computation then CPUs. CPUs are better when a lot of control and logics is needed [14].

2.1.1 General Knowledge

Processing Units are composed of a number of parts, and some programmers are even confused by some of them. Here is some short processor terminology:

- Processor Package – is a circuit board containing one or more dies and a number of cores. Processor package is intended to be set up on to processor socket [15].
- Processor Die – single piece of silicon, covered with metallic or ceramic cover. Under each die there can be a couple of cores and cache memories. One package can have one or more dies. All cores inside die share last level cache [15].
- Processor Core (Physical) – execution unit. There can be a couple of them in a single package. In uniprocessor system this is the only execution unit. Each core has its first level cache. And in case it resides under same die with other cores, they share last level cache as well [15].

- Processor Core (Logical) – some companies like Intel managed to develop Hyperthreading technology that allows having two logical cores inside of one physical core. Therefore quad core CPU has 8 logical cores [15].

### 2.1.2 General Purpose Multicore Processors

This group is the most popular and incorporates all CPUs that can be found in all desktop systems, laptops, embedded systems and servers. These CPUs perform regular everyday work. They are suitable for control and data computations. They are also capable of processing graphics.

Normally systems are composed of two types of processors. They also include graphics processing units to process all graphics related tasks, in addition to main processing unit.

CPUs can be single core or multicore. Right now up to 8 cores available on a single package.

### 2.1.3 Graphical Multicore Processors with general Purpose Processing Capabilities

GPUs have recently become extremely powerful in graphics and data computations. They are composed now of multiple cores. Number of cores varies from 8 to 512 on a single chip, and even more coming. Therefore in GPU world, CPUs are called “Many Core” instead of “Multi Core” [14].

Execution units, i.e. cores on CPUS and GPUs differ. GPU cores are capable of processing graphics and also general purpose computation, especially data intensive. GPU performance is much higher than that of CPU [14].

As usually computing systems are composed of both CPU and GPU, it is worth sometimes to build heterogeneous system, which performs computations on both CPU and GPU, depending on the type of computation. Those kinds of systems are called parallel heterogeneous systems [14].

### 2.2 Parallel Computer Architectures

According to Flynn’s taxonomy [9], there are 4 types of computer architectures:

- Single Instruction Single Data
- Single Instruction Multiple Data
- Multiple Instruction Single Data
- Multiple Instruction Multiple Data
2.2.1 SISD – Single Instruction Single Data

The simplest type of computer architecture is SISD architecture, where there is only one processing element which operates one single data element at one period of time. If there are many data to be processed, machine with SISD architecture will run it sequentially [9].

![Figure 3. SISD architecture according to Flynn’s taxonomy](image)

The problem with architecture shown in Figure 3 is that it cannot handle concurrent execution of instructions, because it has only one processing element.

2.2.2 SIMD – Single Instruction Multiple Data

SIMD computer contains multiple processing elements, each of which is capable of handling one particular type of instruction. This type of architecture exploits data parallelism [9, 14].

Each processing element of SIMD computer accesses unique memory location, and executes assigned instruction [14].

SIMD type of machine has row of advantages, as it can access multiple memory locations at once, and execute particular operation only once, but on each processing element, which is very efficient [14].
All types of application that operate on big amount of data can benefit from SIMD computer [9]. As a disadvantage of SIMD, is that it can execute only single instruction, as shown in Figure 4.

### 2.2.3 MISD – Multiple Instruction Single Data

MISD type of machine, Figure 5, exists only in theory, not in practice. MISD defines machine with multiple processing elements capable of handling different instructions at one time [9].

All MISD processing elements access single data stream with mutual exclusion, making this machine not very efficient.

MISD type is a subtype of MIMD, but MIMD is much more efficient as it can handle multiple streams of data and keep all processing elements busy, taking the most of it [9].
2.2.4 MIMD – Multiple Instruction Multiple Data

MIMD computer architecture is one of the most complex architectures for parallel computing machines. Similarly to SIMD, it has multiple processing elements. But the difference is that it can handle multiple independent and concurrent instructions on multiple independent data at any time [9].

MIMD architecture depicted below on Figure 6 is used by modern hardware manufacturers, like Intel and AMD. MIMD can have different types of memory architecture: shared or distributed, and different number of cores, varying from 2, on desktop computers or laptops, to a couple of hundreds, on computer clusters [9].

![MIMD architecture](image)

Figure 6. MIMD architecture according to Flynn’s taxonomy

MIMD architecture suits best for task parallelism, as each processing element is given different instruction, which is independent, and some data to be processed by this element. Few processing elements might intersect sometimes when accessing same data. Then appropriate synchronization technique should be applied explicitly by programmer, or implicitly by the run time system of particular framework or language [9].

SPMD – stands for Single Program Multiple Data [14] and is a type of computer architecture, which is quite similar to MIMD. SPMD can be executed on general purpose CPUs, unlike SIMD, which require vector processors (GPUs). In SPMD, processes execute program at any point, whereas SIMD processors work at one time.

2.3 TYPES OF MULTIPROCESSING MODELS

2.3.1 AMP – Asymmetric Multiprocessing

Asymmetric multiprocessing, as depicted on Figure 7, is an early type of multiprocessing in machines with two CPUs. First computers had only one processing unit, which was responsible for managing operating system kernel and services, and of
course user applications. Every time it had to switch from one process to another, depending on the priority of ones [9, 6, 16].

All operating systems and all user software were developed to be processed by only one CPU. With the passage of time, applications and operating systems became more demanding, requiring more CPU time to be processed, or even additional CPU.

Later it was decided to have one more CPU on computing machines to make them faster. But then a big problem rose, a problem of multiprocessing and synchronization between those CPUs. Operating systems and software were not ready for this kind of changes.

A little later, a first solution was found. Having two cores on the system, one becomes responsible for operating system processes, the other one for user level applications. This type of multiprocessing is called asymmetric multiprocessing [9].

![Figure 7. Asymmetric Multiprocessing Model](image)

### 2.3.2 SMP – Symmetric Multiprocessing

Symmetric multiprocessing machine is a type multiprocessor machine which has two or more cores, each of which is capable of handling operating system level processes as well as user level processes, refer to figure 8 [9, 17, 6].

This type of processing is hard to achieve, because the more cores you have, the better synchronization mechanisms you need.

The main synchronization rule in this case is if a CPU wants to access any data from memory, or to execute some task, it has to make sure that no one else is having access to it now. If no, than it is free to use it; if yes, then it should probably switch to another task, if one is made available by scheduler.
All CPUs are interconnected by a bus or crossbar switch, according to SMP, which gives access to the global memory and I/O. This type of architecture has some problems, if there are more than 8 CPUs.

Another solution for systems with more than 8 CPUs lies in NUMA memory architecture, which is discussed later in the document [17].

2.4 MEMORY ARCHITECTURE SUITABLE FOR MULTIPROCESSING

2.4.1 UMA – Uniform Memory Architecture

UMA is another type of computer memory organization, shown on figure 9, where all memory is shared among processors. UMA memory is divided into blocks, each of which contains some unique data, for instance: one block of memory holds an operating system kernel and all related OS data; another block is allocated for user applications and data; one more block could be allocated for video memory, if the graphics processor is integrated [9, 16].

To make things more clear, UMA architecture allows unified access to different blocks of memory, each of which serves different purpose.
2.4.2 NUMA - Non Uniform Memory Architecture

NUMA is a design of computer memory for processors that run concurrently, see example on figure 10. Computers with only one CPU can have unlimited access to any memory location at any time. It is very well known that in a computer with more than one CPU a problem of accessing common memory arises. If two or more CPUs at one quantum of time try to access memory, only one will be granted access, others will be blocked or suspended, waiting until other processor finished work with memory [9, 17]. (When two or more processors access the same memory location at one time, memory becomes corrupt; such situations should be avoided).
Similar problem arises with shared memory. If more than one processor tries to access same memory location, only one will be granted access, others will be suspended, until memory is released.

NUMA solves first problem in a way that it provides for each CPU separate memory space, in accordance to CPU’s location. To be more precise, it assigns the closest to CPU memory address space, so that the access time is much faster. The memory which has been assigned to particular CPU is called local memory, according to this CPU. For all other processors, this memory is called non-local memory [9].

So in this case, with local memory, there is no more starvation for processors; they don’t have to wait any more for memory to be released by other processor. They all can be now busy doing their job.

When it comes to shared memory, NUMA solves it by moving shared data between processors’ memory spaces, which makes it complicated and not much efficient, but, despite this fact, this is the way to keep all CPUs away from starvation.

2.5 PARALLEL PROGRAMMING MODELS

There are two known types of parallel programming models: Data-Based and Task-Based. They differ from each other by the way they achieve parallelism and by the purpose of parallelism.

2.5.1 Data-Based Parallel Model

Data-Parallel programming model is a type of computation where sequence of instructions applied to multiple blocks or elements of memory. If to imagine a modern streaming graphics processor, which has a couple of hundreds of cores, each capable of handling separate computation, then it becomes more clear. Having a big set of data that has to be processed, it can be loaded into memory of processor and each memory block or element of data set can be accessed at the same time by a separate thread of execution, or separate stream [14, 9]. Afterwards, synchronization takes place.

With a single core CPU, the same could be achieved sequentially accessing elements of data one by one, which would be time consuming.

Data parallelism has focus on distributing data across multiple lightweight threads [14], and each of those threads performs some atomic operation on its own element of data. This execution model is specific to SIMD architecture, which stands for Single Instruction Multiple Data, meaning that one instruction is executed on multiple data elements by multiple threads.

In this case, threads do not mean operating system threads of control. These are lightweight threads [14].

2.5.2 Task-Based Parallel Model

Task-based parallelism is also known as control parallelism. It is different from data parallelism in a way that each of its threads of control is much more powerful and is suitable for executing long lasting tasks that require a lot of control decisions, not just for accessing an element of data, as lightweight data parallelism threads do. Threads
of control do not necessarily operate on the same data, though this intersection happens quite often [14, 9].

All threads of control are completely independent and usually each of them is executed on a separate core. They have their own allocated memory and they can be in few states: running, suspended, stopped or ready.

A simple example could be as following:

There are two tasks to be executed on a machine with one processor, one execution unit. So what happens is the processor decides which task to run first and launches it, executes it for one millisecond, if it is not completed its job yet, it suspends it and launches another task, giving to it one millisecond of time as well. Right after, it checks again if the job has been completed, and if not, it suspends this task, and switches to previous task. This process goes continuously, until there are available tasks.

So as can be seen from the example, processor has to switch between the tasks all the time, scheduling only one quantum of time for each.

If to imagine a machine with two execution units or two processors, those two tasks can be launched and executed simultaneously, each on separate processor.

It is worth mentioning one more time the difference between task parallelism threads and data parallelism threads:

- Task parallelism threads are capable of handling whole applications, which require lots of memory space, execution control, independence from other threads [17]
- Data parallelism threads are capable of accessing independent memory blocks simultaneously, and then synchronize execution and perform operations on all of them [14]

2.6 PARALLEL TECHNOLOGIES

2.6.1 TPL – Task Parallel Library

Microsoft Task Parallel Library is a high level, multi language construct that allows running different tasks concurrently, taking care of all low level synchronization. TPL makes it easy to achieve task parallelism as well as data parallelism [18].

TPL is defined within .NET Framework 4, and is very easy and efficient way of parallelization of new/existing applications. TPL makes all necessary actions implicitly to have all separate tasks run concurrently, on all available processors.

To achieve task parallelism, Parallel.Invoke method is used [18], where tasks are passed as parameters. In this case programmer does not have to think about how to synchronize access to shared resources that independent tasks use, or where and when to use barrier synchronization.

To achieve data parallelism, TPL has built-in parallel for and foreach loops to substitute sequential for and foreach [18].
2.6.2 MPI

MPI is an Application Programming Interface defined for C/C++ and FORTRAN [10]. MPI has become a standard for programming parallel and distributed systems. It differs from other multiprocessing technologies in a way that it uses message passing between nodes instead of direct access, which is very time consuming.

MPI is the most commonly used library for parallel, and especially distributed computing, where clusters might be far away from each other. MPI parallel model lies perfectly on distributed memory system. MPI standard provides communication between processes, in terms of message passing, and synchronization between them.

MPI program consists of multiple parallel concurrent processes. All processes are created once, defining parallel part of application [9]. When the program is running it is not possible to create additional processes or to destroy existing ones. Each MPI process runs in its own address space. There are no shared variables between processes.

Each process in MPI program executes its job and then sends appropriate message to another process, which is a receiver. MPI suits best for data parallelism model.

2.6.3 Pthreads

Pthreads stands for POSIX Threads, meaning it is a POSIX standard for creating and manipulating with threads. Pthreads was implemented as a library for Unix-like operating systems. Though, windows implementation, as a third party product, named Pthreads-w32, also exists [9].

Pthreads API defined for C language, and is available in pthread.h header file. Pthreads library contains many functions that can be categorized in the following way [9]:

- Thread management
- Conditional variables
- Mutexes
- Synchronization using reader/writer locks and barriers.

2.6.4 Open CL

Open CL stands for Open Computing Language [13, 12], is a standard for general purpose parallel programming across heterogeneous systems (Central Processing Units and Graphics Processing Units).

Open CL was developed by consortium of major hardware and software manufacturers: Apple, Intel, AMD, ARM, IBM, NVIDIA and some others [13, 12]. Therefore it is a cross platform API, providing low level access to different hardware, making it possible to get the maximum performance from one.

Open CL allows executing general purpose code on graphics processing units, which are nowadays multi core boards, with hundreds of cores. It is important to mention that
Open CL supports both data and task-based parallelism, allowing getting the most of task parallelism from CPUs and data parallelism from GPUs [13, 12].

Open CL is an extension to C programming language [13, 12], which makes it easy to integrate with even existing applications.

Open CL can be used for wide range of applications, including embedded software and High Performance Computing software.

As a disadvantage of Open CL is that in order to get the maximum performance from particular hardware, separate implementation should be done, targeting that specific hardware, though, general implementation will run on all possible Open CL hardware, but with difference in performance.

2.6.5 Cilk++

Cilk++ or Intel Cilk Plus is an extension to C/C++ programming languages which enables programmers to write parallel code in C/C++ applications. Cilk++ supports both data parallelism, as its commands are suitable for parallelization of for loops, as well as task parallelism [9].

Cilk Plus provides minimal extension, only three keywords, for achieving parallelism.

Cilk Plus should be used when:

- There is a need to mix serial and parallel code
- Programmer wants to use native programming (no run-time libraries)
- There is a need to get higher performance in data parallelism

2.6.6 OpenMP

Open MP is a multi platform library for shared memory multiprocessing [9]. It supports C/C++ and FORTRAN. It best scales on the computers with shared memory, though it does not run on graphics processing units. It can run on both desktop computers and on supercomputers. It also supports hybrid model, where it can be combined with MPI.

Open MP works in the way that it has a master thread, which is given a task to compute anything. Then it decides how much other slave threads it will need to solve particular task, and creates those slaves, assigning to each part of work. Then it sets up a synchronization point, where it waits for the result from each of the slaves [9].

Workload division between slave threads is done implicitly, so that the programmer does not care about this.

Open MP best suits for data parallelism, but also supports task parallelism. The good side of Open MP is that it is just a set of compiler directives, which can be injected into existing sequential code to make it run in parallel.

2.6.7 AMD ATI Stream

ATI Stream technology is very similar to Nvidia CUDA, which will be introduced in section 2.6.8. On a hardware level, it is a multi core GPU, where each core is named streaming processor. Similarly to CUDA, ATI Stream uses data
parallelism as a first class parallelism, but also allows for task based parallelism, making heterogeneous ecosystem.

ATI Stream provides extension to C language and can be easily integrated into existing applications. ATI streaming technology uses SPMD architecture for achieving data parallelism.

With this streaming technology, application can run sequentially or in parallel on AMD CPU, getting the most of task parallelism, and portions of program that need to process data, can switch to AMD ATI GPU.

AMD’s streaming technology provides all necessary libraries and run time system to make programmers life easier, allowing one to concentrate on the algorithm, rather that low level hardware complexity.

2.6.8 NVIDIA CUDA

Nvidia’s Compute Unified Device Architecture [14, 19, 20] is an API and a run-time system for parallel programming, mainly to achieve data parallelism, by exploiting GPU resources of NVIDIA cards.

CUDA is an extension to C, partially C++ and FORTRAN programming languages. It uses SPMD parallel model to achieve parallelism. The fact that CUDA executes on Graphics Processing Unit does not mean that it can process code related to graphics. CUDA makes it possible to run general purpose code on GPU. Since GPUs are now multi core systems, they suit well for data parallelism [14, 19].

CUDA functions are named kernels. They are similar to normal functions, but with some more CUDA keywords and they run on graphics processor. CUDA function is executed by many threads, each of which has its own ID [14, 19, 20].

SPMD model used by CUDA, similarly to SIMD, allows execution of same instruction on multiple data within one program.

Using CUDA, task parallelism can be achieved by normal means of operating system, and data parallelism by means of CUDA. An application, where data and task parallelism is combined, gives the most performance.

2.7 SCHEDULING POLICIES FOR MULTICORE SYSTEMS

There are two main types of scheduling for multi-core systems: global and partitioned [7].

2.7.1 Global

Global type of scheduler stores all tasks in a single queue based on their priorities [7]. Scheduler takes advantage of all cores available in the system. It can schedule task on any available core. Task migration is allowed, meaning that a task suspended on one core can be resumed on another core [7].

2.7.2 Partitioned
Partitioned scheduler assigns tasks to cores. It makes a number of queues, depending on the number of cores available in the system, and assigns available tasks to particular queue, depending on some attributes of a task [7]. One example could be that two tasks reside in one queue because they both use same shared resource. To be more precise here, each queue is assigned to some core in a system.

2.8 MULTICORE SYNCHRONIZATION PROTOCOLS

Synchronization Protocols are synchronization features of real time operating systems. They help to synchronize tasks that use common resources and meanwhile avoid all common problems, such as deadlocks, starvation, priority inheritance, etc [7, 3, 4, 5].

There are few types of synchronization protocols, developed for both uniprocessor and multiprocessor systems. And more are being developed.

The most common protocols are:

- MPCP - Multiprocessor Priority Ceiling Protocol
- MSRP - Multiprocessor Stack-Based Resource allocation Protocol
- FMLP – Flexible Multiprocessor Locking Protocol
- MHSP – Multiprocessor Hierarchical Synchronization Protocol

These protocols were developed for a particular reason, which says that mutexes, critical sections and semaphores are just building blocks of synchronization. But they do not solve synchronization in complex systems in an easy way. Therefore, use of synchronization protocols or other methods is required.

2.9 SYNCHRONIZATION PRIMITIVES FOR MULTICORE SYSTEMS

There are many different types of synchronization mechanisms for parallel tasks, each of which has its own advantages and disadvantages. It is hard to say which one of those is the best, because there are so many different situations that can occur, and there is no single solution for solving them.

The most popular and commonly used means of synchronization are [16, 17, 9]:

- Mutexes, Critical Sections
- Semaphores
- Events, Timers
- Monitors

Each of them serves one purpose – synchronization, but all of them provide some unique way of doing it.

Unfortunately, these are just synchronization primitives, which are hard to use in complicated software systems. Programmers often forget to release locks, or just get lost in complicated structures and not release locks, etc. While development, these
primitives must be embedded into software systems, which makes code less readable and understandable.

Therefore, specific patterns, that implement logics with these primitives, should be used. Patterns discussed more in detail in next chapters.

2.9.1 Mutexes

Mutex is a special type of variable, which controls access to shared resource. It can be in two possible states [16]:

- Locked – meaning that some process has already acquired it and using shared resource
- Unlocked – means that it can be acquired and access to shared resource is open

In UNIX based systems mutexes are user level objects [16], whereas in Windows they are kernel level objects [17]. The difference here is that user level synchronization techniques perform faster than kernel level. This caused by time consuming operating system switch from user to kernel level.

2.9.2 Semaphores

Semaphore is one of the most powerful mean of synchronization. It is a special variable that is used for resource counting. It differs in implementation from platform to platform, but still the purpose is the same.

On UNIX systems there are two operations on semaphores: “sleep” and “wakeup”. First operation puts process to sleep, if it tries to acquire semaphore which is already acquired, and second one – wakes up process that is waiting in the queue in order to get access to shared resource. Sleep operation firstly check if the mutex value is greater than 0, and if it is, it decrements it. If mutex value is equal to 0, it puts process to sleep [16].

It is good to use semaphores when there are a limited number of resources, shared by processes. On UNIX systems usually at least 3 semaphore variables are used: one for number of available shared resources, one for number of used shared resources, and one for mutual exclusion, which allows only one process at a time to use shared resource.

Each process that uses block of shared resource, decrements semaphore which is responsible for number of available resources, and increments one that is responsible for number of used resources. The mutual exclusion semaphore can only be 0, or 1, so that it allows only one process to get access to shared resource at a time [16, 9].

Mutex operations are atomic, meaning that they are indivisible, and cannot be interrupted during execution. Once they started, they have to finish their work. Process cannot be put to sleep if it performs mutex operation. This ensures proper synchronization.

2.9.3 Events and Timers

Events and timers are used for synchronization of branch execution [17]. Whenever a situation occurs that one process requires information from another one,
there is a need to synchronize execution. This can be explained by the fact that
processes are independent and know nothing about each other, including the time
when they finish execution.

So if a process that requires information needs result produced by process
producer, it checks for a particular event, created by producer. Whenever producer
finishes work, it makes event signaled, because initially it is in non signaled state, so
that it is visible to external processes, and now they can get to know that this process
finished execution and produced its result.

Timers here are used by processes which require information. They specify
how much time they are going to wait for other processes to execute. It might be some
certain period of time, or just infinite.

2.9.4 Monitors

A monitor is a higher level of synchronization primitives used in concurrent
programming [16, 9]. Internally they use mutexes and semaphores to achieve
synchronization, but externally they are easier to program in comparison with
semaphores.

A monitor is a programming construct containing data structures and methods
[16]. Any thread can use monitor, but only one at a time, satisfying mutual exclusion
rule.

If a process, for instance, trying to execute monitor procedure, when this
monitor is being occupied by other process, the first process will be suspended, until
initial process leaves monitor. It is internal feature of a monitor to check whether any
process can execute monitor’s methods.

The problem with monitors is that not every operating system and
programming language provides it. It is available for instance in Java, but not in C++
[16].

2.9.5 Critical Sections

Critical sections or also called Critical Regions are a mechanism to avoid race
conditions, which may occur in situations where two or more processes are trying to
get access to shared resource [17, 9]. Critical Sections might vary internally depending
on operating systems. In Microsoft Windows OS, critical sections are user level
objects, meaning they are extremely fast and efficient, whereas mutexes, which
provide similar functionality, are kernel objects, and that’s why perform slower, in
comparison to critical sections [17]. The reason is that when using mutexes, Windows
has to switch from user level mode to kernel level mode, which is very costly.

So critical region is a region that protects shared resource, no matter what it is.
Critical region allows only one process to use it at one time. Process should leave
region after finishing working with shared resource.
2.10 DESIGN PATTERNS AROUND SYNCHRONIZATION

Design Pattern for synchronization – is a way to use synchronization primitives in a relatively easy and understandable way, making code reusable, maintainable, extendable and less error prone [1, 2]. Design patterns are usually constructed as higher level mechanisms for synchronization, which makes them easier to use comparing to synchronization primitives.

There are many design patterns related to many problems, such as synchronization, resource sharing, resource acquisition, concurrency patterns, event patterns, initialization patterns, etc [1, 2]. Unfortunately discussion of those patterns goes beyond this master thesis, and therefore we will have a short look at few of them, related to synchronization.

2.10.1 Scoped Locking

Scoped locking – is a way to use mutexes, semaphores or critical sections in a safer way [8, 1, 21]. Scoped locking pattern automatically takes the lock when it is called. Programmer does not have to care about relinquishing of the lock, as it will be done automatically when control passes scope where lock has been taken.

Implementation of scoped locking is very easy and straightforward. Separate class should be created with constructor and destructor. In constructor, we pass lock by reference and acquire it [8, 1, 21]. In destructor we just release lock [8, 1, 21]. This means that we just need to create object that incorporates scoped locking pattern, in a place, where we want to acquire lock. Later during execution lock will be automatically relinquished. Otherwise, destructor can be called manually.

2.10.2 Strategized Locking

Strategized Locking is a pattern that allows use of different synchronization strategies [8, 1]. Each synchronization strategy might use different types of locks: mutexes semaphores, etc. It is quite inconvenient to have separate implementations for each strategy.

Strategized locking makes this situation easier, introducing common way for each strategy.

2.10.3 Reader/Writer Locking

Reader/Writer Locking pattern is widely used in many different software systems, due to its popularity [9, 16, 21]. And, also, different implementations of it exist.

A typical scenario for this pattern is when there are multiple reader tasks and one or more writer tasks. Each of them time to time refers to some shared data to perform either reading or writing. Hence, proper locking should be used, allowing all tasks function correctly.

Reader/Writer locking pattern solves this above described scenario. Pattern uses two levels of locks. First level locks intended for all reader tasks, and second level locks are intended for writer tasks. If a second level lock is acquired, no reader can perform reading.
Chapter 3

RESULTS

3.1 COMPILER TESTING FOR PARALLELIZATION AND OPTIMIZATION

This part of thesis includes results from testing Microsoft Visual Studio 2011 and GNU GCC/G++ with QT Creator for parallelization and/or optimization of sequentially written code.

3.1.1 Test Applications

Test application is a matrix multiplication program. It multiplies 2 square matrices and produces the result. Matrix dimensions can be set by the user. Application outputs time spent on calculation.

![Figure 11. Test application “matrix Multiplication” for windows](image)

A couple of test applications have been created. An example of Windows application is shown on Figure 11. There are a few written in normal sequential manner, without taking much care about calculation code. This is done intentionally, in order to see if compilers are able to optimize/parallelize this kind of “dirty” written code. Others are parallel, manually parallelized using MPI, which is described in 2.6.2.
Matrix Multiplication (Sequential, naive) – sequential version of matrix multiplication application. It has very basic implementation of multiplication. It has been used 2 times with different compiler switches: Od and Ox, as depicted on picture 12.

Matrix Multiplication (Sequential, manual optimization) – sequential version of matrix multiplication application. This version is manually optimized. It uses registers, tiling algorithm and multiplication performs in reverse order, in order to achieve better memory usage and therefore better execution time. It has been used 2 times with Od and Ox compiler switches, as shown on picture 12.

Matrix Multiplication (Parallel, manual parallelization) – parallel version of matrix multiplication application. This version is manually parallelized using MPI technology, which will allow us to run code locally on multicore processor and in distributed manner on computers connected via network. 2 different compiler switches have been used: Od and Ox. For details see picture 12.

3.1.2 Test Platforms

Tests have been performed on different computers with different operating systems. The following are types of computers that have been used during tests:

- Dell Latitude D630
  - Operating System: Microsoft® Windows® 7 Enterprise 64 bit OS
  - Processor: Intel® Core™ 2 Duo T7700 2,4 GHz
  - IDE: Microsoft® Visual Studio® 2010 Premium

- Dell Latitude D630
  - Operating System: Linux Ubuntu 11.10 64 bit OS
  - Processor: Intel® Core™ 2 Duo T7700 2,4 GHz
  - IDE: NOKIA QT Creator 2.4.1 based on QT 4.7.4 (64 )
  - Compiler: GNU gcc/g++ (mpicc/mpicxx)

- Dell Optilex 745
  - Operating System: Linux OpenSUSE 11.1 i586
  - Processor: Intel® Core™ 2 Duo 6700 2,66 GHz
  - Compiler: GNU gcc/g++ (mpicc/mpicxx)
3.1.3 Test Results for Windows

Test results are divided into two groups. One group has only one chart, which is a comparison between all types of applications under different conditions on different platforms. Another group has many charts. Each chart represents particular type of application and condition under which it has been running.

![Comparison Chart (Windows)](chart.png)

Figure 12. Comparison of execution time between different types of matrix multiplication applications in Windows

Comparison on Figure 12 shows that MPI implementation together with compiler optimization gives the most speed up to the computation, when running on a single computer with dual core processor. Manual optimization together with automatic compiler optimization gives almost the same result. Although MPI version would run much faster if there were more cores available. In that case version with manual and compiler optimization would lose completely. As to MPI without any optimizations, it shows good result, but it loses 20 seconds to MPI with optimization.

Figure 12 gives us a clear picture of which version is better. And consequently from those tests we can make a conclusion that manual parallelization is still the winner.

The following are charts for particular application types. Each chart depicts time of execution for each dimension that has been set, type of application and compiler switch, if any and implementation technology, if any.
• **Application:** Matrix Multiplication  
  o **Type:** Sequential  
  o **Implementation technology:** None  
  o **Optimization Option:** /Od (Disabled)  
  o **Matrix info:** Square Matrices  
  o **Computer:** Dell Latitude D630

![Matrix Multiplication (/Od)](image)

Figure 13. Matrix Multiplication with /Od compiler switch

• **Application:** Matrix Multiplication  
  o **Type:** Sequential  
  o **Implementation technology:** None  
  o **Optimization Option:** /Ox (Full Optimization)  
  o **Matrix info:** Square Matrices  
  o **Computer:** Dell Latitude D630
Figure 14. Matrix Multiplication with /Ox compiler switch

- **Application:** Matrix Multiplication
  - **Type:** Parallel
  - **Implementation technology:** MPI
  - **Optimization Option:** None
  - **Matrix info:** Square Matrices
  - **Number of threads:** 3
  - **Computer:** Dell Latitude D630

Figure 16. Matrix Multiplication with /Od switch and MPI implementation
- **Application**: Matrix Multiplication
  - **Type**: Parallel
  - **Implementation technology**: MPI
  - **Optimization Option**: /O2 (Maximize Speed)
  - **Matrix info**: Square Matrices
  - **Number of threads**: 3
  - **Computer**: Dell Latitude D630

![Matrix Multiplication (/O2:MPI)](image_url)

Figure 17. Matrix Multiplication with /O2 switch and MPI implementation

- **Application**: Matrix Multiplication
  - **Type**: Sequential
  - **Implementation technology**: None
  - **Optimization Option**: /Od, Manual Optimization
  - **Matrix info**: Square Matrices
  - **Computer**: Dell Latitude D630
**Figure 18.** Matrix Multiplication with /Od switch and manual optimization

- **Application:** Matrix Multiplication
  - **Type:** Sequential
  - **Implementation technology:** None
  - **Optimization Option:** /Ox, Manual Optimization
  - **Matrix info:** Square Matrices
  - **Computer:** Dell Latitude D630

**Figure 19.** Matrix Multiplication with /Ox switch and manual optimization
3.1.4 Test Results for Linux

Test results for Linux are divided analogically to results for Windows.

Figure 20. Comparison of execution time between different types of matrix multiplication applications in Linux

As for tests on Linux, each implementation tends to run a bit faster than on Windows. In this test, manual optimization wins, showing fantastic result, only 14.9 seconds for 2200 * 2200 elements, see Figure 20. No implementation can compete with this one. Although parallel implementation still has more advantages, but not in this case. It (parallel implementation) would take the most speed up on any multi core hardware, as it is evident from number of external projects, studies and tests.

The following are charts for particular application types. Each chart depicts time of execution for each dimension that has been set, type of application and implementation technology if any.

- **Application**: Matrix Multiplication
  - **Type**: Sequential
  - **Implementation technology**: None
  - **Optimization Option**: None
  - **Matrix info**: Square Matrices
  - **Computer**: Dell Latitude D630
Figure 21. Matrix Multiplication (naive)

- **Application**: Matrix Multiplication
  - **Type**: Parallel
  - **Implementation technology**: MPI
  - **Optimization Option**: None
  - **Matrix info**: Square Matrices
  - **Number of threads**: 4
  - **Computer**: Dell Latitude D630
Figure 22. Matrix Multiplication with MPI

- **Application:** Matrix Multiplication
  - **Type:** Parallel
  - **Implementation technology:** MPI
  - **Optimization Option:** None
  - **Matrix info:** Square Matrices
  - **Number of threads:** 6 (3 dual core processors connected via network)
  - **Computer:** Dell Optiplex 745

![Matrix Multiplication MPI (Distributed)](image)

Figure 23. Matrix Multiplication with MPI (distributed)

- **Application:** Matrix Multiplication
  - **Type:** Sequential
  - **Implementation technology:** None
  - **Optimization Option:** None
  - **Matrix info:** Square Matrices
  - **Computer:** Dell Latitude D630
Figure 24. Matrix Multiplication with manual optimization
The process of planning new applications or customization of existing application for multi core systems requires at least 3 steps [9]:

- Decomposition
- Connections
- Synchronization

Though, there might be some more steps to be considered, especially those ones related to the very beginning phase of planning, where architect needs to identify what kind of parallelism is needed; or one related to customization of existing programs, which is profiling.

### 3.2.1 Identification of parallelism

As it was mentioned in the preceding chapters of this document, there are two types of parallelism: data and task parallelism. They are used for different purposes, but in spite of this sometimes they might intersect. The most common situation is when inside independent parallel task one needs to have data parallelism.

Therefore, on the first stages of application planning, architect should figure out what kind of parallelism is required [9]. Answer on this question can be found in project requirements.

### 3.2.2 Profiling

Profiling – is a step which is more related to customization of existing application, where one should identify the performance of particular function or the whole application [9].

There are many profiling tools that can help to identify the heaviest parts of application. One should only load application into profiling tool, and launch execution.

For parallel application, profiler can show how much part of the code work collaboratively and on what. For sequential application, with the help of profiler one can figure out heavy loops in the code, which signal for possible parallelization of that particular part of code.

Profiling tools are mostly used in data parallelism. But despite this fact they can also help in task parallel applications. For instance, they can show how many times or how often particular process was using shared resource, keeping other processes sleeping.

### 3.2.3 System decomposition

Decomposition is a process of breaking down the system into independent parts [9]. Software decomposition is sometimes also named WBS – which stands for Work Breakdown Structure [9], where word work means also task. Work decomposition identifies what should those decomposed parts do.

One of the biggest problems of parallel programming – is to identify natural decomposition of works (tasks), to be solved programmatically. There is no simple
and straightforward way to identify WBS. Because software development process is a process of translation of ideas, principles, formulas, rules into a set of instructions, readable for processing unit, which can execute them. All mentioned above is included into a process of software modeling. Modeling process opens natural decomposition of tasks. The more model in understood, the more natural decomposition will be.

The main purpose is to identify parallelism and decomposition with the help of modeling. If natural parallelism cannot be identified while modeling, it should not be applied.

Two or more tasks can be considered parallel, if they can run independently. In the places of intersection of parallel tasks, appropriate synchronization should be applied, to avoid starvation, or wasteful use of resources.

3.2.4 Identification of connections

After the process of decomposition another big problem arises, a problem of identification of connections between independent tasks [9], which also includes problem of how tightly these tasks are bound.

Now when the system is decomposed, independent tasks might be assigned to different processing units on one parallel machine, or they might run on a distributed machine, physically separated and connected by network.

There are many questions that should be answered during this identification stage:

- Should independent tasks use shared memory?
- How do independent tasks get to know about whether other tasks started or finished execution?
- Which task should start work first?

Answer on these and many more questions should be found during this stage. It is also important to know, that all parallel tasks are independent, but they work within one program, and they all eventually have one aim, meaning, that at somewhere at one point they should be synchronized. If two independent tasks do not require synchronization, this is the fact that they do not compose one application.

3.2.5 Synchronization

Synchronization is the third biggest problem of creating applications for multi core systems. If decomposition identifies independent tasks, then synchronization identifies how often when and where they should synchronize execution. There are also many questions that should be answered during synchronization stage:

- Which tasks have access to shared resource?
- Who has the right to get access first to particular resource?
- If some tasks finished execution much earlier that others, what should they do?
- When particular tasks should be synchronized?

Answers on these and many more questions should be found during synchronization stage of parallel software development.
3.3 PARALLELIZING COMPILERS

3.3.1 DEVELOPMENT OF PARALLELIZING COMPILERS

Advanced parallelizing compiler technology is an R&D project that is conducted mainly in Japan by JIPDEC in collaboration with Fujitsu, Hitachi, National Institute of Advanced Industrial Science and Technology, Waseda University, Tokyo Institute of Technology and The Toho University also take part in the project [10].

One of the main problems as visioned by the project is adoption of parallel processing as a core technology on Personal Computer to High Performance Computer.

Object of development is an Advanced Parallelizing Compiler together with performance evaluation technology for parallelizing compilers.

One of major goals is to double the performance of applications. As the number of processing units is increasing, gap between maximum hardware performance and effective performance widens, as mentioned in [10].

Currently there is ongoing research about automatic loop parallelization. There are few different problems, for example data dependency analysis, or loop restructuring. University of Illinois together with Stanford University and University of Maryland are trying to solve mentioned problems.

As paper [10] describes, technology has reached a state where loops are difficult to parallelize. This can be explained by the fact that loop order plays a great role in loop parallelization and previously this knowledge was avoided by programmers due to popularity of single core processing chips.

3.3.2 LIMITATIONS OF PARALLELIZING COMPILERS

As described by Mafijul [11], 10% of loops can be automatically parallelized by Intel C/C++ compiler. The remaining part of theoretically parallelizable code remains not parallelized due to many reasons.

Research shows that in general auto parallelizable loops cover only 12.5% of total execution time of application.

Research also shows that major reason behind inability to parallelize code is unknown loop trip count and of course data dependencies. On average 37.5% of loops cannot be auto parallelized because of the mentioned reasons.

If in general the number of parallelizable loops is 100% - only 22% can be auto parallelized, which is not sufficient.

3.4 DESIGN PATTERN AND SYNCHRONIZATION TECHNIQUE

Design pattern named “Concurrent” has been developed, together with synchronization technique to target the most complex synchronization situations in relatively easy way.

Concurrent design pattern has been developed for 3 platforms: Windows, Linux and VxWorks.
3.4.1 Design Pattern “Concurrent”

Concurrent is a C++ class that plays a role of an abstract class, although it is not. Instance of this class can be created, but it will not make any sense. This class is intended for derivation. Concurrent should always play a role of parent class.

What is so special about this class? It contains synchronization logics that is implemented inside it and provided as public interface to external world.

Synchronization provided by Concurrent class is encouraged to be used in particular cases, where objects that are running concurrently want to access many resources at once in parallel. In this case, every object should be derived from Concurrent class, as described below on Figure 25.

![Figure 25. Class diagram of Concurrent design pattern and example of inheritance](image)

As one of the existing examples, solution to Dining Philosophers problem partially solves similar type of problem. But solution to Dining Philosophers problem and problem itself is quite limited and relatively simple, comparing to problem scenarios presented in problem formulation section.

Generally, any case with multiple objects and multiple resources can be solved with Concurrent design pattern. One more good point is that pattern can also handle simple
synchronization cases; such as for example synchronize 3 objects that want to get access to one shared resource. The last problem can be solved in different ways and quite easily and there are existing solutions for that. Concurrent design pattern can also solve this type of synchronization. The only disadvantage in this case is that it might take some more time, comparing to existing solutions that specifically target that case. But still the algorithm has proved its ability to solve almost any kind of synchronization problem.

Concurrent design pattern is quite unique. No analogical synchronization techniques and patterns have been found during the time frame allocated for this thesis work. Such algorithms might exist, but none have been found.

Detailed information about pattern usage is described in 3.4.3.

3.4.2 Areas of application

Pattern exists in 3 versions, for Linux, VxWorks and Windows, so that it can be used in any of mentioned operating systems. “Concurrent” is not dependent on types of threads, tasks or processes. It can be used in many different ways, giving flexibility for programmers and architects when designing a parallel software system.

Problem scenarios mentioned in problem formulation section are a good example of area of application. Basically any situation that has many objects that run in parallel and many resources that are being accesses by those objects can be handled by pattern.

Here are some more examples of scenarios where pattern can be applied.

In scenario depicted in Figure 26 there are 6 concurrent objects and 8 resources. Each object tries to access some amount of shared resources. Each object is free to access any resource he wants. In the example above, some objects access 2 resources, some 3, 4 and even 5 resources at a time. And this makes this example quite complicated. Scenario in Figure 13 can easily be solved by Concurrent design pattern.
Scenario on Figure 27 shows one more complicated resource division between objects. Concurrent design pattern can solve this situation in an easy and maintainable way.

### 3.4.3 Usage of design pattern

The public interface of Concurrent consists of the following functionality:

- **Init()** – This function initializes some global structures, such as creates binary semaphore for object that calls it, for instance. That semaphore is a part of synchronization mechanism. It takes no parameters. Initialization of global structures is done through references to global structures that reside in Concurrent class. init() must be called for every object before it starts concurrent execution, i.e. starts accessing shared resources. If some objects already access shared resources and new object comes, it can safely call init() and start accessing resources it wants. Function should be called only once for each object.

- **Deinit()** - This function is a contrary to init() - it deinitializes global structures. It takes no parameters. deinit() must be called before object terminates in order to properly deinitialize all structures previously initialized by init(). deinit() should be called only once at the end of lifecycle of the object.

- **Mark_resource(int res_id)** – Function marks specified resource. After it is called, specified resource will be registered in objects specific table and will take part in synchronization operations. It should be called for every resource object wants to use. If object does not use some resources for a long time,
unmark_resources() should be called in order to improve performance of synchronization operation. If in this case unmark_resources() is not called, synchronization will take longer time.

- **Unmark_resource(int res_id)** – Function unmarks resources that are not used by object for a long time. If object is not accessing resources he marked before for some long time, it might slow down performance of synchronization for other objects who currently take part in synchronization. Although this function should not be called if object is not using resources for few seconds. It should also be called at the end of the lifecycle of the object, before it terminates.

- **Register_resource(int res_id, std::map<int, int> )** – This function is used to register shared resource, if resource is a part of an object who calls this function. It can be an attribute of a class, for example. All resources should be registered before any object wants to access them. Resources can be registered during runtime of application. Function ensures mutual exclusion before doing any registration operations. Global shared resources, that are not part of any class, should be registered globally. This can be done in two ways: First - call this function from any object passing necessary parameters, just the same as you would do if resource was part of a class; Second - do manual initialization:
  
  o m_rTablesOfUsers[ res_id ] = table_of_users; - add to global table of users map containing two integers, in this case it is table_of_users. It should be empty; res_id should be unique id for resource you are registering;
  
  o m_rTableOfFlags[ res_id ] = FREE; - initialize unique resource id with initial status FREE. As FREE is not a global definition, it can not be used outside any class, therefore integer 3 is used instead. It is equivalent to FREE.

  If manual registration is done at runtime, code should be wrapped with semaphore acquisition and relinquishing.

- **Unregister_resource(int res_id)** – Function unregisters previously registered shared resource. Unregistration of any shared resource can be done if resource is not needed any more and if no object is using it. If there is at least one object using resource, it should not be unregistered, otherwise behavior is unpredictable. Unregistration usually is done when object is terminating. Programmer must make sure that no objects are using resource, before unregistering it. Unregistration can be also done manually in the following way:
  
  o m_rTablesOfUsers[ res_id ].clear(); // clear table of users
  
  o m_rTablesOfUsers.erase( m_rTablesOfUsers.find( res_id ) ); // erase table of users from map
  
  o m_rTableOfFlags.erase( m_rTableOfFlags.find( res_id ) ); // erase resource from table of flags

  Important thing to keep in mind is runtime. If unregistration is done at runtime, above operations should be wrapped with acquisition of global semaphore.

- **Acquire_resources()** – Funktion contains algorithm for resource acquisition. It can be called if all resource registration and initializations have been made. Function takes no parameters as it knows about the situation by global shared structures. It should be called just before resource is accessed. Function can be
called anywhere programmer wants. It can be called directly, or it can reside inside of a public interface function that accesses resources, or in any other way.

- **Release_resources()** – function containing main relinquishing algorithm; it releases resources that have been acquired before;

The following pieces of code show examples of how pattern can be used.

```cpp
#include "Concurrent.h" // include header of the base class
#include "Shared.h"

class Active_1 : public IRunnable, public Concurrent // inherit synchronization properties of base concurrent class
{
public:
    Active_1( HANDLE hmutex, std::map<int, HANDLE>& event_table, std::map<int, HANDLE>& access_table,
              std::map<int, std::map<int, int>>& tables_of_users, std::map<int, int>& count_table,
              std::map<int, std::map<int, int>>& table_of_flags,
              std::map<int, int>& missed_wakeups, int id );

    ~Active_1( void );

    void Active_1::Run()
    {
        long cnt = 0;
        while( cnt != 10 ) // run this example certain amount of cycles
        {
            do_other_work();
            if( acquire_resources() != 1 ) // inherited synchronization function that acquires needed resource
                printf("Problems with resource acquisition in Obj1\n");
            use_shared_resources();
            if( release_resources() != 1 ) // inherited synchronization function that releases resource that has been acquired
                printf("Problems with resource release in Obj1\n");
            update_cnt();
            cnt++;
        }
    }
}
```

The above piece of code shows the use of functions acquire_resources() and release_resources(). These functions and some others have been inherited from Concurrent class.

The two synchronization functions wrap use_shared_resources() function, making resource access safe, deadlock and starvation free.
Question that remains unknown is how algorithm knows who to synchronize and with what. All this information lies in global tables. The following is a description of those tables:

- `extern HANDLE g_hMutex;` - global semaphore that is used in the algorithm for synchronization purpose.
- `extern std::map<int, HANDLE> g_eventTable;` - global event table containing object IDs and binary semaphores for every object. In Windows version table contains event objects.
- `extern std::map<int, std::vector<int>> g_accessTable;` - global table containing object IDs and vector of resource IDs that object is accessing.
- `extern std::map<int, std::map<int, int>> g_tablesOfUsers;` - global table of users containing resource IDs and tables of users as separate maps for each resource. Each table of users contains object ID and its status. Status values are described later in this section.
- `extern std::map<int, int> g_countTable;` - global count table containing object IDs and number of accesses that object has made to a set of resources.
- `extern std::map<int, int> g_table_of_flags;` - global table of flags containing resource IDs and their state. State variable related to resources is explained later in this section.
- `extern std::map<int, int> g_missed_wakeups;` - global table containing Object IDs and integer variable representing number of missed wake ups. Missed wake up is when one object is trying to wake up another object but cannot, then it increases that objects wake up count variable.

**Status variables for objects:**

- **PASSIVE** – indicates that object is not interested in the resource yet
- **ACCESSING** – indicates that object is currently using resources
- **WAITING** – indicates that object is waiting for the resources to be free to get access to them

**Status variable for resources:**

- **FREE** – indicates that resource is free and can be accessed
- **BUSY** – indicates that resource is currently being accessed by some object

All mentioned parameters should be passed to a class derived from concurrent class. Initially all structures are empty. They can be created by the programmer and passed to constructors of objects. Initialization of structures is happening behind the scene, when calling functions of concurrent class.

Generally pattern can be used in many different ways. There is no predefined way of how to use it. One is free to implement its own program structure. The only important thing is to use derivation from Concurrent pattern properly as depicted in this chapter.
Chapter 4

DISCUSSION

This chapter describes strong and weak points of design pattern “Concurrent” that has been developed within thesis work.

4.1 STRONG POINTS OF “CONCURRENT” DESIGN PATTERN

Concurrent design pattern solves complex synchronization situations in relatively easy and understandable way, allowing programmers to concentrate on the problem, rather than synchronization. Concurrent also makes synchronization common for all objects, which makes it reusable and the code maintainable.

The following are strong points of the pattern:

- No knowledge about synchronization, deadlocks, starvation and CPU resource usage is required from programmers
- No implementation of synchronization code is explicitly involved in the logics of classes
- Classes remain maintainable and reusable
- Easy integration with any C++ classes with minimal code modifications
- Algorithm is capable of acquisition of any number of resources at a time

4.2 WEAK POINTS OF “CONCURRENT” DESIGN PATTERN

Inherently it was decided to make pattern compatible with C/C++, as these languages are very popular. This is where some weak points come from. Algorithm that is implemented within interface functions of the pattern is quite complicated and involves a lot of logics to avoid all possible problematic situations in parallel execution and therefore pattern needs to have complicated input parameters that help solving synchronization. The following are weak points of the pattern:

- Applicable not to all programming languages
- Only applicable to Object Oriented environment
- Algorithm requires complicated input parameters
5.1 ACHIEVED RESULT

As the work of this master thesis has been divided into three major parts, results have been achieved in every part.

In the first part compilers were tested for parallelization and optimization. Two compilers have been chosen for two different platforms: Microsoft Visual Studio 2011 for Windows and GNU GCC/G++ compiler together with QT Creator IDE on Linux Ubuntu.

As have been found, both compilers do not support auto parallelization. Instead, Visual Studio 2011 has a couple of compiler switches for code optimization, unlike QT Creator that does not have any built-in features for code optimization. But QT uses GNU GCC compiler which has optimization options similar to those in Visual Studio environment.

Results chapter presents charts and diagrams from all tests that have been performed. It shows that manual parallelization tends to be a little faster than compiler optimization. But if using both of them, then we get twice as much speedup. Parallel version of test application with MPI specification is quite fast, although it could be faster if tests were made on better multicore hardware, for instance a quad core CPU. One more big benefit of parallel version is that it will always scale through the number of cores available or number of cores specified, meaning that the better the hardware the better the result.

What is very interesting is that same implementation tends to run faster on Linux platform then on Windows.

There are compilers on the market that can parallelize sequential code. One of those is Intel C/C++ compiler. Research has shown that it can parallelize generally up to 25 percent of code. The rest of the code cannot be parallelized by it due to many reasons; some of those are loop dependencies, unpredictable number of iterations, etc. Investigation of information about parallelizing compilers is part number two of thesis work.

There are other compilers also that are being developed by some major Japanese companies together with universities. Example of such compiler is NEDO Advanced Parallelizing Compiler.

Third part of thesis work is development of design pattern for synchronization. “Concurrent” design pattern has been developed. It contains synchronization technique that solves very common and very problematic scenarios in parallel computing. To be more precise, parallel objects that run concurrently and access many resources at a time can be synchronized. Design pattern itself helps making this synchronization code reusable, maintainable, and easier to understand. It creates a level of abstraction between programmer and synchronization technique.
5.2 IMPROVEMENTS FOR CONCURRENT DESIGN PATTERN

Design pattern "Concurrent" is fully functional. It has been tested under different conditions, although it would be great to do some more tests on different types of multicore hardware.

Following are the things that could be improved:

- Testing on different multicore hardware
  - With different numbers of cores
  - Different hardware vendors
- Synchronization algorithm is quite extensive and could be reviewed for simplifications
- Proper testing on multicore environment should be done for VxWorks

5.3 TESTING ON PROPER MULTICORE HARDWARE

Test applications developed within master thesis has been tested on different dual core computers, including computers connected via network.

The following things could be done:

- Testing on different multicore hardware for more precise results
  - Testing on multicore processors with shared memory
  - Testing on multicore processors distributedly
  - Testing Intel C/C++ compiler for parallelization and optimization

More research about software parallelization techniques could be done also.
REFERENCES


APPENDIX A

CONCURRENT.H FOR WINDOWS

class Concurrent
{
    public:
        Concurrent( HANDLE& hmutex, std::map<int, HANDLE>& event_table, std::map<int, std::vector<int>>& access_table, std::map<int, std::map<int, int>>& count_table, std::map<int, int>& missed_wakeups, int id );
        ~Concurrent (void);

        int init(); // initialise yourself
        int deinit(); // deinitialise yourself
        int mark_resource(int res_id); // mark wanted resource if you need to use it
        int unmark_resource(int res_id); // unmark resource if you dont use it anymore
        int register_resource( int res_id, std::map<int, int> ); // register resource if you have it and if you wish to share it
        int unregister_resource(int res_id ); // unregister resource before exiting
        int acquire_resources(); // function to acquire resources
        int release_resources(); // function to release resources
        int m_nID; // object ID

    private:
        int find_min( std::map<int, int>& inmap ); // function to find minimum access count
        int find_max( std::map<int, int>& inmap ); // function to find maximum missedwakesups count
        HANDLE& m_rhMutex;
        std::map<int, HANDLE>& m_rEventTable; // reference to global event table; one event for each object (key: objectID; value: event)
        std::map<int, std::vector<int>> & m_rAccessTable; // reference to global access table;(key: objectID; value: vector of resource IDs that object accesses)
        std::map<int, std::map<int, int>> & m_rTablesOfUsers; // reference to global table of users; contains tables of users; each table contains: (id: resourceID; key: ObjectID; value: accessType)
        std::map<int, int>& m_rCountTable; // reference to global count table; number of accesses to different shared resources by each object (key: objectID; value: count)
        std::map<int, int>& m_rTableOfFlags; // reference to global table of flags; one flag for each resource; flag can be either BUSY or FREE
        std::map<int, std::map<int, int>> :: iterator m_it_tu; // iterator for m_rTableOfUsers
        std::map<int, int>:: iterator m_it;

        static const int ACCESSING = 1; // state of object when it is currently accessing shared resources
}
static const int PASSIVE = 0; // state of object when it is not interested in any resource
static const int WAITING = 2; // state of object when it is interested in the resources, but they are taken
static const int FREE = 3; // state of resource when it is free and can be taken by the first one who asks for it
static const int BUSY = 4; // state of resource when it is being accessed by some object
};

GLOBAL SHARED STRUCTURES

extern HANDLE g_hMutex; // global mutex, protects all resources
extern std::map<int, HANDLE> g_eventTable; // event table; one event for each object (key: objectID; value: event)
extern std::map<int, std::vector<int>> g_accessTable; // access table; key: objectID; value: vector of resource IDs that object accesses
extern std::map<int, std::map<int, int>> g_tablesOfUsers; // map of tables of users for each resource
extern std::map<int, int> g_countTable; // key: objectID; value: number of accesses to set of shared resources
extern std::map<int, int> g_table_of_flags; // table of flags; one flag for each resource; key: resourceID; value: state of resource
extern std::map<int, int> g_missed_wakeups; // table contains missed wakeups for each object; key: ObjectID; value: number of missed counts
#ifndef CONCURRENT_H
#define CONCURRENT_H

#include "semLib.h"
#include "stdio.h"
#include <vector>
#include <map>

class Concurrent
{
public:
    Concurrent(SEM_ID& hmutex, std::map<int, SEM_ID>& event_table, std::map<int, std::vector<int>>& access_table, std::map<int, std::map<int, int>>& tables_of_users, std::map<int, int>& count_table, std::map<int, int>& table_of_flags, std::map<int, int>& missed_wakeups, int id);
    ~Concurrent(void);

    int init();   // initialise necessary data
    int deinit(); // deinitialise on exit
    int mark_resource(int res_id);   // mark specified resource (only if you want to use it)
    int unmark_resource(int res_id); // unmark resource if you dont use it anymore
    int register_resource(int res_id, std::map<int, int> ); // register resource if you have it and if you wish to share it (global shared resources should be registered globally)
    int unregister_resource(int res_id ); // unregister resource before exiting
    int acquire_resources(); // algorithm for resource acquisition
    int release_resources(); // algorithm for relinquishing resource

protected:
    int m_nID;   // object’s ID

private:
    int find_min( std::map<int, int>& inmap );   // finds minimum access count
    int find_max( std::map<int, int>& inmap );   // finds maximum missedwakeups

    SEM_ID& m_rhMutex; // reference to global mutual-exclusion semaphore
    std::map<int, SEM_ID>& m_rEventTable; // reference to global event table;
    one event for each object (key: objectID; value: binary semaphore)
    std::map<int, std::vector<int>> &m_rAccessTable; // reference to global access table;(key: objectID; value: vector of resource IDs that object accesses)
    std::map<int, std::map<int, int>> &m_rTablesOfUsers; // reference to global table of users; contains tables of users; each table contains: (id: resourceID; key: ObjectID; value: accessType)
    std::map<int, int>& m_rCountTable; // reference to global count table; number of accesses to different shared resources by each object (key: objectID; value: count)
    std::map<int, int>& m_rTableOfFlags; // reference to global table of flags; one flag for each resource; flag can be either BUSY or FREE
    std::map<int, int>& m_rMissedWakesups; // reference to global missedWakeups table; key: objectID; value: cariable containing number of missed wakeups
    std::map<int, std::map<int, int>> ::iterator m_it_tu; // iterator for m_rTableOfusers

    std::map<int, int>::iterator m_it; // iterator for map<int, int>

    static const int PASSIVE = 0;   // state of object when it is not interested in any resource
    static const int ACCESSING = 1; // state of object when it is currently accessing shared resources

};
static const int WAITING = 2; // state of object when it is
interested in the resources, but they are taken

static const int FREE = 3;    // state of resource when it is free and can
be taken by the first one who asks for it

static const int BUSY = 4;    // state of resource when it is being
accesses by some object

#endif CONCURRENT_H
EXPLANATION OF “CONCURRENT”

Concurrent is a design pattern that contains special synchronization algorithm for synchronization of threads/tasks in complex situations. Design pattern is a special way to deal with synchronization on multi-core processors as well as on single core. Pattern is designed to be used for control/task parallelism.

Synchronization algorithm is capable to synchronize any number of threads/tasks that access any amount of shared resources at once, which is a unique feature of the algorithm. Design Pattern Concurrent wraps algorithm, providing convenient API functionality for dealing with synchronization without having to know much about synchronization itself. With Concurrent, synchronization is seen as a black box, where programmer has to know only what to input; output of this black box is proper synchronization and less headache from deadlocks, starvation, inefficient CPU usage, semaphores and so forth. Synchronization with Concurrent pattern can be applied to many situations where mutual exclusion is required. Algorithm manages mutual exclusion in heavily loaded parallel environment.

One of the key features of Concurrent is that it is a dynamic algorithm. It provides dynamic management of parallel objects and shares resources. It can adapt to parallel environment just as programmer wants. During runtime, parallel objects can play with number of resources, i.e. add some more resources to access, or subtract some resources. They can also add new resources at runtime and delete resources. New parallel objects can also arrive and existing objects can leave execution. All API functionality ensure mutual exclusion while allowing multiple objects to access multiple resources at the same time.

There are existing synchronization algorithms that solve concrete synchronization problems. Concurrent contains a general purpose synchronization algorithm that handles many types of synchronization problems, including dining philosophers problem, readers and writers problem and similar. But the most important thing is that it has been designed to solve more complex situations with multiple objects and multiple resources. Therefore, performance of the algorithm might be slower when synchronizing some basic solution, such as dining philosophers problem or some well know simple problems.

Concurrent is a C++ class that implements synchronization logics and other functionality for managing synchronization at run time. Concurrent class should be used as an abstract class, only for derivation, as instances of it will make no sense. Concurrent provides public interface for dealing with synchronization and shared resources. All public methods will be inherited by child classes. Any C++ class can be derived from Concurrent. When derived, new object inherits synchronization functionality from parent class, i.e. Concurrent and becomes capable of taking part in parallel execution.

A simple use case could be as following:

There is an application containing 8 C++ objects and 6 shared resources. 5 objects out of 8 are using shared resources. Moreover, each of them uses different number of resources. The other 3 objects do some other work and do not take part in synchronization. In this case, the 5 objects that use shared resources should be synchronized. Consequently they should be derived from Concurrent class. Other 3 objects should be left as they are. Now 5 objects have
synchronization properties inherited from Concurrent. Lets suppose that all 5 objects have access_resources() function in which they access their set of resources. In this case this function should be wrapped with calls to inherited functions acquire_resources() and release_resources().

Each object can run in a separate task/thread, and each task/thread can be assigned to specific processor core, which would give us truly parallel environment. Scenario can be more complicated. New objects can come during runtime of application and start accessing resources. And that is possible, if using Concurrent class. Objects can also change resources they use to other ones, or just stop using resources whenever they want. All this can be done during runtime. Interface of Concurrent makes it easy to achieve all above mentioned.

*************** IMPORTANT TO KNOW ***************

OBJECT - C++ object, in our case also derived from Concurrent class. This object takes part in concurrent execution and accesses some shared resources.

RESOURCE - any type of variable or structure, that is a subject of concurrent access. Each OBJECT has a set of RESOURCES it accesses. RESOURCE can be global or local.

GLOBAL RESOURCE - it is a resource, i.e. variable or structure, that is defined in global scope.

LOCAL RESOURCE - is a resource, i.e. member variable or member structure of some OBJECT that can be accessed directly or indirectly via interface by other objects.
There are 8 global variables and maps that are important for concurrent execution with Concurrent pattern. They all must be defined in application before concurrent execution starts. These maps contain information about objects and resources, which is needed by Concurrent.

<table>
<thead>
<tr>
<th>Variable/Map</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM_ID g_MSemaphore;</td>
<td>Global mutual-exclusion semaphore used for synchronization. Should be created with semMCreate() function and passed to Concurrent's constructor</td>
</tr>
<tr>
<td>std::map&lt;int, SEM_ID&gt; g_eventTable;</td>
<td>Global event table containing ObjectIDs as keys and binary semaphores as values. It is internally used for synchronization. It will be automatically initialized when init() function is called. It should be passed to Concurrent's constructor. This map is initially empty.</td>
</tr>
<tr>
<td>std::map&lt;int,std::vector&lt;int&gt;&gt; g_accessTable;</td>
<td>Global access table containing objectIDs as keys and vector of resource IDs that object accesses as values. This map can be initialized by calling mark_resource() function and deinitialized by calling unmark_resources(). This map is initially empty.</td>
</tr>
<tr>
<td>std::map&lt;int, std::map&lt;int, int&gt; &gt; g_tablesOfUsers;</td>
<td>Global map containing tables of users. It contains resourceIDs as keys and as values it contains a map which has ObjectIDs as keys and accessType as values. Access type can be one of the three object states declared in Concurrent class. If object provides its own member or members as shared resources, it must first register them by calling register_resource() function. Otherwise, if shared resources are global, this structure should be initialized manually. Note that before manual initialization global semaphore must be acquired!See example below.</td>
</tr>
<tr>
<td>std::map&lt;int, int&gt; g_countTable;</td>
<td>Global count table contains ObjectIDs as keys and count variables as values. This table contains number of accesses to different shared resources by each object. This map will be initialized by calling init() function. Initialization will be dependent on the parameter to init() function.</td>
</tr>
<tr>
<td>std::map&lt;int, int&gt; g_table_of_flags;</td>
<td>Global table of flags containing resourceIDs as keys and flags as values. Flag can be one of two values declared in Concurrent class: FREE and BUSY. Flags signal current state of resource. This map is empty initially. It will be initialized by calling init() function.</td>
</tr>
<tr>
<td>std::map&lt;int, int&gt; g_missed_wakeups;</td>
<td>Global map containing number of missed wakeups for each object. It contains objectIDs as keys</td>
</tr>
</tbody>
</table>
and integer variable as value, which signals the number of missed wake ups. This variable is empty initially. It will be initialized by calling init() function.

||
|---|---|
| **std::map<int, int> g_resource_state;** | Shows if resource is marked by some objects, and by how many (key: resource id; value: number of objects that marked resource). This map is used internally for synchronization purposes. |

**SUMMARY**

To summarize general knowledge about global shared tables (maps) it is important to remind that all variables should be created before execution starts and passed empty to the constructors of objects derived from Concurrent class. Names of all global maps can be defined by users, they do not necessarily need to be named as described above. The most important thing is that they must be passed to constructor of objects derived from Concurrent class.
int init( bool bcount = false );

DESCRIPTION:

This function initializes some global structures, such as creates binary semaphore for object that calls it, for instance. Function takes one boolean parameter: bcount. It will indicate whether object calls init() before concurrent execution, or at a runtime. Initialization of global structures is done through references to global maps that reside in Concurrent class. init() must be called for every object before it starts concurrent execution, i.e. starts accessing shared resources. If new object comes at runtime, it can safely call init() and start its work. Function should be called only once for each object.

PARAMETERS:

bool bcount - boolean variable initialized to false by default. It means object is created before concurrent execution has started, on the beginning. If object comes at runtime, it must call init( true ) and set parameter to true, in order to align it with other objects. Ignoring this might cause starvation problems.

PRECONDITION:

• If called at run time: “true” should be passed as a parameter;
• If called at the beginning, when concurrent execution has not been started (default), nothing should be passed, or “false” - which is by default;

RETURN VALUES:

1 - SUCCESS: function executed correctly
-1 - ERROR: general semaphore problem while acquiring semaphore
-2 - ERROR: invalid mutual exclusion semaphore; resulted from S_objLib_OHJ_ID_ERROR this error can happen while acquiring semaphore or while releasing it
-3 - CRITICAL ERROR: general semaphore problem while releasing semaphore
-4 - ERROR: binary semaphore creation error; function can be called again.

EXAMPLE CODE:

SomeClass Instance = new SomeClass( g_MSemaphore, g_eventTable, g_accessTable, g_tablesOfUsers, g_countTable, g_table_of_flags, g_resource_state, 1 );
Instance->init();
...
// if object has arrived at run time, when other objects are already accessing some resources\n    call init( true ) and set parameter to true
...
    Instance->init( true );

*******************************************************************************

DEINIT
*******************************************************************************

int deinit();

DESCRIPTION:

deinit() - deinitializes global maps. It takes no parameters. deinit() must be called
before object exits, in order to properly deinitialize all maps previously initialized by init().
deinit() should be called only once at the end of lifecycle of the object, or when it has to
terminate.

PRECONDITION:

init() must have been called;

RETURN VALUES:

  1 - SUCCESS: function executed correctly
  -1 - ERROR: general semaphore problem while acquiring semaphore
  -2 - ERROR: invalid mutual exclusion semaphore; resulted from
               S_objLib_OHJ_ID_ERROR this error can happen while acquiring
               semaphore or while releasing it
  -3 - CRITICAL ERROR: general semaphore problem while releasing semaphore

EXAMPLE CODE:

    Instance->init();
    ...
    //access resources concurrently
    ...
    //if now is the end of lifecycle of the object, or it has to terminate, deinit() should be
called
    Instance->deinit();

*******************************************************************************

MARK_RESOURCE
*******************************************************************************

int mark_resource(int res_id);
DESCRIPTION:

This function marks specified resource. After it is called, specified resource will be registered in object's specific table and will take part in synchronization. This function should be called for every resource object wants to use. If object does not use some resources for a long time, unmark_resources() should be called in order to improve performance of synchronization. Object cannot access any shared resource before it calls this function.

PARAMETERS:

\( \text{int res_id} \) - id of resource you want to add to your access table in order to access it later by calling acquire_resource().

PRECONDITION:

No precondition

RETURN VALUES:

1 - SUCCESS: function executed correctly

-1 - ERROR: general semaphore problem while acquiring semaphore

-2 – ERROR: invalid mutual exclusion semaphore; resulted from S_objLib_OHJ_ID_ERROR this error can happen while acquiring semaphore or while releasing it

-3 - CRITICAL ERROR: general semaphore problem while releasing semaphore

-4 – ERROR: resource does not exist

-5 – ERROR: Bad resource id

EXAMPLE CODE:

```
Instance->init( true );
...
int ret = Instance->mark_resource( 5 );
switch(ret)
{
    case 1:
        //SUCCESS, do something
    case -7:
        // resource does not exist, try other resource id
    ....
}
```

*******************************************************************************
int unmark_resource(int res_id);

DESCRIPTION:

Function unmarks resources that are not going to be used by object for some time, or at all. If object is at the end of its lifecycle or is about to terminate it should call this function for every resource it previously marked.

This function should also be called if object is not going to use resources for some period of time. This period is not defined, and should be decided by the programmer. It is worth to know also that calling this function too often might also slow down execution. Programmer should find tradeoff, depending on the system that is being developed.

As an example, if object makes access to resources every 5 seconds, it should not call this function after each access, i.e. after every 5 seconds. But if object makes access to resources every 15-20 seconds or so, it should better call this function after every access. Important to consider the fact that if there are not many objects in the system who are in concurrent execution, call to this function might be skipped. It would be not necessary to call it due to small number of concurrent objects.

NOTE: If object is not accessing resources he marked before for long time and did not unmark, it will slow down performance of synchronization for other objects who currently take part in synchronization.

PARAMETERS:

int res_id - id of resource you want to delete from your access table.

PRECONDITION:

• mark_resource() has been called;

RETURN VALUES:

1 - SUCCESS: function executed correctly
-1 – ERROR: general semaphore problem while acquiring semaphore
-2 - ERROR: invalid mutual exclusion semaphore; resulted from S_objLib_OHJ_ID_ERROR this error can happen while acquiring semaphore or while releasing it
-3 - CRITICAL ERROR: general semaphore problem while releasing semaphore
-4 – ERROR: resource does not exist
-5 – ERROR: Bad resource id

EXAMPLE CODE:
Instance-&gt;access_resources();
...
//if now object is not going to access resources for some reasonable amount of time,\
unmark_resource() should be called
...
Instance-&gt;unmark_resource( 5 );

*******************************************************************************

************ REGISTER_RESOURCE ************

\textbf{int register_resource( int res_id, std::map<int, int> );}\n
\textbf{DESCRIPTION:}\n
This function is used to register shared resource, whether it is global or local. All resources should be registered before objects start accessing them. Resources can be registered at startup or during runtime of application. This function ensures mutual exclusion. Global shared resources, that are not part of any class (not a member), should be registered globally with this function called from any object, or explicitly (see example below).

\textbf{NOTE:} there is no difference in registering local and global resource. For instance, Object2 can register resource that belongs to Object1, and vice versa. Also, one object can register all resources that are in the system. This is legal. Registration of resources is not private, it is global and has global effect, therefore it can be done by any object or explicitly.

\textbf{PARAMETERS:}\n
\texttt{int res_id} - unique id of a resource that is being registered

\texttt{std::map<int, int> - map containing two integers, as key and value. It represents table of users for this specific resource. Map with two integers should be created and passed to this function empty, as a parameter.}\n
\textbf{PRECONDITION:}\n
No precondition

\textbf{RETURN VALUES:}\n
\begin{itemize}
\item \textbf{1 - SUCCESS:} function executed correctly
\item \textbf{-1 - ERROR:} general semaphore problem while acquiring semaphore
\item \textbf{-2 - ERROR:} invalid mutual exclusion semaphore; resulted from \texttt{S_objLib_OHJ_ID_ERROR} this error can happen while acquiring semaphore or while releasing it
\item \textbf{-3 - CRITICAL ERROR:} general semaphore problem while releasing semaphore
\end{itemize}
ERROR: resource does not exist

ERROR: Bad resource id

EXAMPLE CODE:

LOCAL RESOURCE REGISTRATION:

```
Instance->init();
...
std::map<int, int> table_of_users; // table of users for shared resource
```

```
Instance->register_resource( 4, table_of_users ); // lets assign id 4 to shared
```

GLOBAL RESOURCE REGISTRATION CASE 1:

```
int shared_resource = 200; // global shared resource
```

```
Instance->init();
...
std::map<int, int> table_of_users_1; // table of users for global shared resource
```

```
Instance->register_resource( 5, table_of_users_1 ); // assign id 5 to shared \
resource
```

GLOBAL RESOURCE REGISTRATION CASE 2:

```
int shared_resource_2 = 30; // global shared resource
```

```
std::map<int, int> table_of_users_2; // table of users for this resource
```

```
semTake( &g_MSemaphore ); // take global semaphore
```

```
mr_tables_of_users[ 6 ] = table_of_users_2; // assign id 6 to this resource
```

```
mr_table_of_flags[ 6 ] = 3; // make this resource FREE, i.e. assign 3 to its ID default
```

// FREE is an in class identifier, therefor it can not be used externally, but as it \n// corresponds to ... \n// ... integer 3, then integer 3 can be used instead.

```
semGive( &g_MSemaphore ); // relinquish global semaphore
```

From now this resource can be referenced by ID 6;

```
Instance->mark_resource( 6 );
```
UNREGISTER_RESOURCE

`int unregister_resource(int res_id );`

DESCRIPTION:

Function unregisters previously registered shared resource. Unregistration of any shared resource should be done if resource is not needed any more and if no object is using it. If there is at least one object using resource, unregistration will return specific error code. If object that provides resource is about to terminate, it should call this function.

PARAMETERS:

`int res_id` - unique id of a resource that is being unregistered

PRECONDITION:

- `register_resource()` has been called;

RETURN VALUES:

1 - `SUCCESS`: function executed correctly

-1 - `ERROR`: general semaphore problem while acquiring semaphore

-2 - `ERROR`: invalid mutual exclusion semaphore; resulted from S_objLib_OHJ_ID_ERROR this error can happen while acquiring semaphore or while releasing it

-3 - `CRITICAL ERROR`: general semaphore problem while releasing semaphore

-4 - `ERROR`: resource does not exist

-5 - `ERROR`: Bad resource id

EXAMPLE CODE:

LOCAL & GLOBAL RESOURCE UNREGISTRATION:

`Instance->unregister_resource( 6 );`

GLOBAL RESOURCE UNREGISTRATION:

`semTake( &g_MSemaphore );`

`mr_tables_of_users[ 6 ].clear();` // clear table of users

`mr_tables_of_users.erase( mr_tables_of_users.find( 6 ) );` // erase table of users from map

`mr_table_of_flags.erase( mr_table_of_flags.find( 6 ) );` // erase resource from table of flags
semGive( &g_MSemaphore );

**************************************************************

*************** ACQUIRE_RESOURCE ***************

int acquire_resources();

DESCRIPTION:

Function contains algorithm for resource acquisition. It should be called just before resource is accessed, to ensure mutual exclusion.

PRECONDITION:

• mark_resource() has been called;

POSTCONDITION:

• release_resources() must be called immediately after access to resources;

RETURN VALUES:

1 - SUCCESS: resources acquired successfully
-1 - ERROR: general semaphore problem while acquiring semaphore
-2 - ERROR: invalid mutual exclusion semaphore; resulted from S_objLib_OBJ_ID_ERROR this error can happen while acquiring semaphore or while releasing it
-3 - CRITICAL ERROR: general semaphore problem while releasing semaphore
-4 - ERROR: error on semGive(), but rollback has been made; function can be called again, as it is now in the initial state.
-6 - ERROR: error on semTake(), but function can be called again, as it is now in the initial state.

EXAMPLE CODE:

...
...
Instance->acquire_resources();
Instance->access_shared_resources();
Instance->release_resources();

**************************************************************
**RELEASE_RESOURCE**

`int release_resources();`

**DESCRIPTION:**

Function contains algorithm for relinquishing resources. It should be called right after access to shared resources has been done. There are no limitations on this function.

**PRECONDITION:**

- acquire_resources() has been called;

**RETURN VALUES:**

1 - **SUCCESS:** resources released successfully

-1 - **ERROR:** general semaphore problem while acquiring semaphore

-2 - **ERROR:** invalid mutual exclusion semaphore; resulted from S_objLib_OBJ_ID_ERROR this error can happen while acquiring semaphore or while releasing it

-3 - **CRITICAL ERROR:** general semaphore problem while releasing semaphore

**EXAMPLE CODE:**

```c
...
...
Instance->acquire_resources();
Instance->access_shared_resources();
Instance->release_resources();
```