Pre-Analyses Dependency Scheduling with Multiple Threads

Nichlas Korgaard Møller
Supervisor: Bent Thomsen

June 12, 2016

AALBORG UNIVERSITET
STUDENTERRAPPORT

Title:

Pre-Analyses Dependency Scheduling with Multiple Threads

Project period:

P10, Spring Semester 2016

Project Group:

dpt109f16

or nmalle11@student.aau.dk

or nkm182@hotmail.com

Authors:

Nichlas Korgaard Møller

Supervisor:

Bent Thomsen

Total Pages: - 132 pages

Appendix: - 42 pages included in the 132 pages and a folder containing code.

Completion date: - 12-06-2016

Synopsis:

This report contain information on how to utilize multiple cores from the CPU with a different abstraction than threads. The abstraction created was tasks. The tasks can be seen as a to do list, where a previous task has to be finished before the next task, these tasks may be run concurrently if they do not affect one another. The report succeeded in creating a dependency scheduler with comparable performance or better than the existing solutions for managing threads. The report conducts multiple micro benchmarks and a real application test with a spreadsheet, where it succeeds in speeding up the spreadsheet.

Contents

1 Introduction

1.1 Introduction

The technology of the multi-/many-core era, where programmers have to handle the cores with the abstraction of threads is still containing many problems. The computation may have to be predictable and end up with consistent results in the end of the program. To ensure predictable code, *monitors*, *semaphores*, *locks*, *observer pattern*, *future and promises* had to be implemented[13, Chapter 18-19][11]. New model designs with multi-/many-core in mind have to be created[11]. Multitasking introduces problems such as *deadlocks*[24, P. 425-427], *livelocks*, *racing conditions*[28], Amdahl's law[8]. As software developer it becomes difficult to use the multi-/many-cores, as the hardware gets more complex, caches on the multi-/many-core can be implemented in different sizes, access times to RAM vary and the number of cores vary with different speeds between chips.[11]

With Open Multi-Processing (OpenMP)[3] there are numerous of ways to use multiple threads or cores. OpenMP is an open API which is portable to personal computers up to super computers. On a super computer it can be used together with Message Passing Interface (MPI). Alternatively OpenMP's extensions can be used for non-shared memory systems. The OpenMP language is a languages that extends support for use of multiple

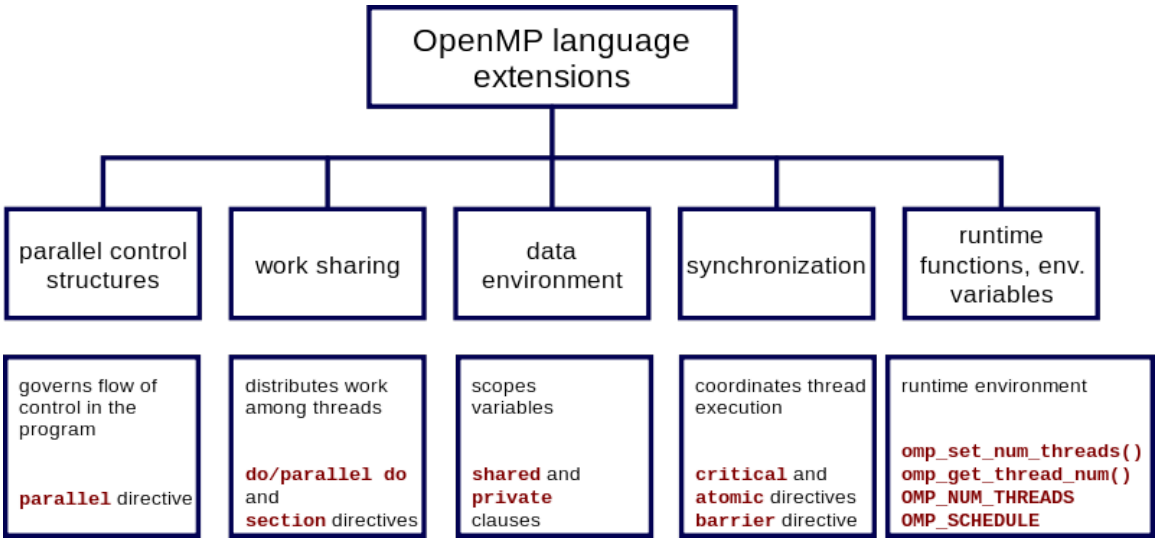

Figure 1.1: This figure has been taken from [29] and illustrates the extensions of OpenMP.

threads, figure 1.1. OpenMP has been written for C, C++ and Fortran. It allows different

control structures to use the cores in parallel such as parallel for, or user defined threads making the user enable to write their own parallel code. OpenMP has synchronization clauses which enables the user to define critical code which only will run specific code parts serial instead of parallel. It can be changed to always have a specific number of threads, or control its internal scheduling of loop iterations. MPI libraries[6] have been created to be used on clusters or for general multiple thread applications. These MPI libraries are used to solve general purpose parallel programming solution with no dependency. The most common MPI libraries are found in Fortran, C and in C++, most MPI libraries are open sourced. The concept of MPI is to create synchronization and communication between a set of processors. These *sets* are mapped to different computers/servers/nodes which are connected together. These MPI libraries are meant to split up the program and run the split up parts on the different processors, send messages back and forth, for either results sharing or new programs to execute and in the end synchronize the result[6].

Today GPUs can be used to calculate parallel data, but the GPUs are hard to program to, and might not always be reliable as it gives unexpected results[1].

There are functional languages that exploits parallel structures or loops to become parallel, with either implemented GPU use or use of multiple threads. Where they shift the focus of the programmer to step from a single threaded ideology to a more multi-threaded focus. Harlan is written with a Scheme syntax that calculates various results on the GPU and transfers the memory[5]. Brahma for F# can calculate results on the GPU[7]. C++ AMP developed by Microsoft which features both the use of multiple threads and interface with the GPU and with DirectX[16]. There are libraries like Paralution[9] which has multiple solutions in a library for multiple threads(OpenMP), for OpenCL, for CUDA created in C++ and FORTRAN.

There are general solutions from Microsoft with their scheduler[22] which features *task stealing*, *thread injection*, this has been implemented in *.NET 4.0*.

A program can only be parallel to a certain degree, the multi-core chips hardware structure will affect the results. This report will include functions from *.NET*[19], these functions will be used for thread handling. *.NET*'s scheduler does not take dependency into accord, the user will have to implement it by themselves. In the 3.0 version of *.NET* they implemented *thread pool*[23] which uses the existing threads in the windows system, with this *thread pool* they also implemented a way to stop a thread with *Waitone* to wait until it have been set with *AutoResetEvent* [15]. These functionalities has been implemented on the Linux *.NET* version where the threads will be created on start up. The idea is to use

the available hardware, as the CPU is often not fully used, as the process of programming multi-threaded programs can be difficult.

All of these languages and libraries have something in common, they do not take care of dependency between tasks at runtime, it is left for the programmer to ensure.Most of them solve specific problems like parallelism of loops. They do not use the idea that multiple single methods could be run concurrently at the same time if they do not depend on shared data. This report investigate some of the multi-thread problems with another perspective than the usual. As the perspective of this report will be how to change the optics of the programmer to be what depends upon this piece of code. Therefore the programmer has to change their train of thought to the following: figure 1.2, 1.3, and 1.4

A. Execute A.
$B \succ A$. Wait for A then execute B.

Figure 1.2: A does not have a dependency. B depends on A. A can be executed. After A has completed its task, B can then be execute.

A. Execute A.
B. Execute B.

Figure 1.3: A does not have a dependency. B does not have have a dependency. Both can be executed concurrently.

A. Execute A.
$B \succ A$. Wait for A then execute B.
C. Execute C.

Figure 1.4: A does not have a dependency. B depends on A. C does not have a dependency. A and C can concurrently be executed. After A has completed its task, B can then be executed.

A application is to use the preliminary data from UPPAAL[10] and execute it on functions corresponding to the UPPAAL system names. Which in this case could be system names A,B,C they are the same template system in UPPAAL. Another system which has to be

implemented in UPPAAL is a system representing the CPU with multiple cores. The cores can either be busy or available. The system names A,B, and C could ex correspond to method names that has been created in a program. What makes the systems different is that the dependency should be implemented by the user as well as how it works should be implemented in UPPAAL. In one such implementation of UPPAAL the dependency can be implemented with bool arrays for the three systems. The three systems has then been programmed to check if they have dependencies in their bool array. If a system does not have a dependency and a core is available to execute the thread, it can then execute the thread. If the system has a dependency it has to check if the system, that it is dependant on, has been executed then it can execute if there is an available core. With this UPPAAL can be used to verify if it executes correctly in a written program.

Another application could be to use spreadsheets, as the spreadsheet can be divided into dependencies. Such as a cell in a spreadsheet ex $A1$ can be dependant on another cell in the spreadsheet $A2$. If $A1$ is dependant on $A2$ then $A2$ should be executed before $A1$ can be executed. With this idea it becomes possible to make an analysis on the spreadsheets dependencies. Which will enable the possibility to execute multiple calculations concurrently.

1.2 Summary

This report will start with an analysis of the current problem, as we lack a library in general which will handle multiple threads with different abstraction than threads. It will contain a chapter in design of the dependency scheduler and how it can be used together with UPPAAL. Which implementations the user can do themselves to use a virtual override, which features and flaws the design contains. As well as a diagram showing what the scheduler do and how it works. The report will contain information about the different functions within the scheduler, with examples of how they are used with simple implementation, there will also be an example of a solution Microsoft has created. The report will feature micro benchmarks on its performances. The report will have a real application test spreadsheet benchmark which will be compared to the original solution using a single thread. Lastly the report will contain a discussion/conclusion and a future works. The appendix will contain the information gathered from the benchmarks in tables, and the library code.

1.3 Background

Microsoft's *.NET* [19] contains partial parallel solutions of the common algorithms. It contain *LINQ* to database coding, different task libraries, *Async* methods, *Thread handling*.

.NET strives to be as general in its solution as it possible can be.

Microsoft have created a *task scheduler* [22], which requires to define the number of threads available for this scheduler. It does not have any dependency implemented and will execute as soon as the main thread has started the task. It features *task stealing*, *thread injection/retirement* for maximum throughput and *task inlining*.

Microsoft have created a *thread pool* [23], which accesses the available threads in the system. The utilization of the threads will save time, as only existing threads will be used, designers can create more threads if necessary.

Microsoft have created a *lazy class* known from functional languages [18], which only evaluates when required. This is used together with Microsoft's scheduler [22] to handle multiple threads.

Microsoft have multiple delegates one of them is *Action* [14], this can be used as a *Thunk* from functional languages. The *Action*(*Thunk*) is a small dummy argument which is used as a delay to calculate specific segments of code. *Action* can be used with multiple parameters in a delegate to pass variables to a function or a calculation.

Microsoft have a *Reflection* method [21] which can call code based on a *class type* and on a *method name*. The *Reflection* method can be given multiple parameters to be used in the called methods.

Microsoft have implemented a way to easy use multiple threads [20] in a for loop with *Parallel.For*. Which splits the data among the available cores.

Microsoft C# features *Generics* [17], which can be used instead of *upcasting* and *down-casting*. This is the better solution in C# compared to casting.

Microsoft released *AutoResetEvent* [15] together with the *Thread Pool*. This can be used as a bit operator to indicate the thread can continue after *Thread.Sleep* has been used, or when a thread has to wait until another thread awaken the thread that has used *Thread.Sleep*.

Microsoft have created the library C++ AMP[16] which uses *DirectX* to interface with the GPU thus removes the need to go through either CUDA or OpenCL. AMP can be used with multiple threads on the CPU.

Larsen, Kim G., Paul Pettersson, and Wang Yi [10] is a tool paper on UPPAAL which was created to solve problems with models. These models can be used to verify different reachability questions such as: If there will be a deadlock or if it reaches a specific state.

Sestoft, Peter[25] has developed a spreadsheet in C#, where the user can define their own functions and most of standard operations known from spreadsheets are implemented.

Sestoft, Peter[26] has written a paper on how to measure the speed of software, or determine if a new solution is faster than an old one. This paper gives multiple advices on how to perform tests and how better tests are made.

Sestoft, Peter[27] has written a book "Spreadsheet Implementation Technology: Basics and Extensions" which contains informations on the spreadsheet[25]. It contains information on, what a spreadsheet is, the referencing other spreadsheet uses for cells, architecture, implementation of the spreadsheet, the functions, support graphs and much more information.

LibreOffice [4] is another spreadsheet with fully implemented OpenCL use. It uses the CPU, APU or the GPU whichever has the most computational power.

Libreoffice spreadsheets[12] is the source that [2] used for its tests, as well as this report uses to compare against.

1.4 Related Work

Hill, Mark D., and Michael R. Marty[8] summarizes Amdahl's Law and how critical it is for the multi-core era. They also investigate different multi-core chips and compare them on graphs for which chips theoretically can get the best results.

Lee, Edward A. [11] states, there exist alternatives to threads such as the observer pattern. He argues that much of the difficulty is a consequence of the abstraction for concurrency which threads bring. The consequences was concluded as experts had written a multi-threaded program, to figure out four years later that it could cause a deadlock. As the paper concludes the use of threads is not ideal, as "But nontrivial multi-threaded programs are incomprehensible to humans."[11].

Alglave, Jade, et al[1] investigate GPU concurrency by studying concurrent behaviour of deployed GPUs, where the current specifications of languages and hardware are inconclusive. To back their claims is a litmus test which they have created by using a low-level language for the GPU to check the hardware for problems. The Litmus tests weak behaviour and ideas for GPU programming, which are supported by official tutorials, as false. This paper was created to prove there was an error in vendor documentations and asking for a clearer documentation.

Holk, Eric, et al[5] is the language/tool Harlan which uses the language Scheme to do GPU calculations. This language features automatic memory transfer from and to the GPU.

PARALUTION [9] is a tool created for various sparse iterative solvers for multi/many core CPU and GPU devices. These solutions are created in C++ and Fortran.

Message Passing Interface Forum has published several documentations on Message Passing Interface(MPI)[6] where a lot of different libraries and standards is based upon.

OpenMP Architecture Review Board[3] is a collection of companies which update and use Open Multi-Processing. They have multiple documents, examples, introductions, books, and wikipedia[29] on how to use OpenMP and how OpenMP is used. As well sources and documentation on its implementation.

Padua, David has written Encyclopedia of Parallel Computing[24, P. 425-427], which contains the description of a deadlock, how to detect a deadlock and how to avoid or prevent deadlocks.

Wheeler, David has written Secure programmer: Prevent race conditions[28], which explains race conditions and their weaknesses, with most claims being security problems. Whereas many different race conditions can be in software.

Mark Michaelis has written Essential C#[13], which contains a lot of information about C# in general for *.NET* library before and with *.NET 4.0*. It contains some of the more normal ways to do concurrent programming with threads, with the use of *monitors*, *locks* and *semaphores*.

Grigorev, Semyon [7] has created BRAHMA a tool to use GPU programming in F#, it uses part F# syntax and part OpenCL syntax.

AMD[2] is an article about the speed up gained in spreadsheets with the use of the APU compared to a single core CPU.

2 Analysis

This chapter will summarize the problems with multitasking on multi-core chips.

2.1 Problem statement

It is difficult to write and debug multiple-/many-core programs as they can have unexpected errors. The unexpected errors, problems, considerations:

- *Racing condition*: Where multiple threads manipulate the same data and read/write does not work as expected. An example can be both threads count up once for every time they access the data. But only one thread count will be counted up because they both read at the same time, thus not being aware of other thread.
- *Deadlock*: Is example: When the two threads can no longer move forward because they lack a resource the other thread manages, thus creating a stalemate between two or more threads.
- *Livelock*: Is in the same category as deadlock. The Livelock is created by the use of multiple threads managing the same data but never being able to settle as they change the shared data, making the threads change the shared data over and over.
- *Debugging*: In languages without an integrated development environment (IDE) as Fortran, C and C++ it is not easy to figure out where the code went wrong, or if the threads intertwined with the same data.
- *Thread creation*: when should a new thread be created, how many threads can the many/multi-core handle, is there a limitation in data size, input/output restriction. When are the threads terminated if terminated.
- *Handle the threads*: when should *monitors*, *semaphores*, *mutex*, *locks*, *atomic locks*, *fence*, *blocks* be used and how are they best used, eventually is there performance issues of using them.
- *Threads to core ratio*: does a thread use an entire core, can a core have multiple threads, how does many threads affect the core, is more threads better or worse.
- *Readability of the program*: When does a new thread start and what does it work on while the other threads or the main thread is working.

- *Cache usage*: How does the cache work best on multiple cores as the L1 cache is local, the L2 cache is shared between two cores, and then L3 cache shared between all cores.
- CPU hardware:
  - How does Intel's Hyper Threading affect the threads performance and when is it activated/deactivated.
  - How does AMD's Turbo Core affect threads in general and when is it activated/deactivated.
  - Can the software influence Hyper Threading and Turbo Core or are they controlled by the hardware and BIOS (Basic Input/Output system also known as motherboard software)
  - Power saving functions, do they affect the results or the core performance.
  - Architecture differences, that affect the computation.
  - Differences in instruction sets.
  - Differences in speed Ghz/Mhz on the processors, number of cycles.

There will always be the pros and cons of whether to use a single thread versus using multiple threads, and various studies throughout time speaking for and against doing one or another. If programs are time critical it can use multiple threads for a better result. There are many libraries which tries to solve the general problems, mainly in Fortran, C and C++ such as MPI libraries. None of these considers dependency as of any importance as the dependency is created by the user and has to be solved by the user. There is a lack of a general solutions for local multiple core programs which uses two or more cores. There is a lack of solutions that uses dependency. For example one solution that requires dependency could be spreadsheets, where the user defines that they require a calculation with a specific cell. That specific cell can depend on calculations from other cells and so forth. However, the dependency can be converted to other forms of programs where one thing is dependant to execute before another.

The solution this report want to create is a dependency scheduler that handles most of the above mentioned problems, or lessens the burden of the above problems.

3 Design

This chapter is how the problems can be solved and the design of the solution.

3.1 The Dependency Scheduler process

The solution which is contemplated, is a Dependency Scheduler. Which handles all the threads that the CPU can manage. This will be achieved through *.NET's thread pool*, which can get the minimum number of threads corresponding to the number of threads the used CPU has. The *thread pool* is used to keep track of the threads, this is managed by *.NET*, and they are all created from system start up, so no additional time has to be used for creating new threads.

The dependency scheduler has to be created as minimalistic as possible to not affect threads too much or make any busy waiting. *.NET* have a lot of functions and among the functions is *WaitOne* which can be used to prevent busy waiting while the thread sleeps.

The dependency scheduler primarily uses *Reflection*. However, *Reflection* is not easy to figure out how to use. Therefore the dependency scheduler required another primary method that is simpler to use. The simpler method that the dependency scheduler will use is *Action*. *Action* was also chosen because it is faster than *Reflection* which have a large overhead.

The naming convention of the methods strived to be as convenient as possible or as understandable as possible. In *Reflection*'s case the methods had to be understandable rather than convenient. The method names had to be different because of the function *params* which enables a method to take as many parameters as required. In the similar case for *Action* it could keep the same name, as it does not use *params*.

The scheduler is a dependency scheduler it requires a dependency system based on integers, which requires the name it is given to be an integer and the dependencies is required to be a list of integers. For a better abstraction it is recommended to use *enum* for the names of the methods and convert the methods to integers.

The scheduler will then handle the dependency if there is any if not it will execute the task right away. The scheduler is finished at the time that all tasks has finished and removed from its internal list.

3.2 The Algorithm

The Dependency Algorithm is started up at the time a task has been added. When a task is added a new thread is created to handle the scheduler. The scheduler then proceed to first create a *AutoResetEvent* which is a wait handler for itself. Then it proceeds to look at the list where the tasks are inserted. It starts from the first task to check if it is worked on, if not it will check if it got any dependencies. If it does not have a dependency it will start a new thread for the task and afterwards mark the bool for the *TaskItem* that it is being worked on to indicate that it should not start this item again. If the *TaskItem* have dependencies it will check the list if there is any of the dependencies which it needs to fulfil before it can execute. After it has traversed all the tasks that it could start up, it will start to look through the list if any of the tasks has finished, if a task has finished it will be removed from the list. If the dependency scheduler is running and is removing tasks while the main thread is adding tasks, the system will go into an exception. This can be avoided while adding tasks with the method *AddingTaskLock*, when there are no more tasks which needs to be added, *AddingTaskUnlock* has to be used else the dependency scheduler will endlessly wait and never awaken the main thread. The tasks will indicate they have finished when one of the worker threads has finished running its task and written back to *TaskItem* that it is done. The scheduler will check if the list is empty if it is empty it will proceed to awaken itself and the main thread. Then finish the scheduler and the scheduler thread. If the list is not empty it will proceed to wait until a thread has finished or until a new task has been added. When it has been re-awoken it will call itself with recursion and start anew from the top, unless the list is empty where it will finish the thread.

The implementation of tasks has been implemented with list from C# and the dependencies has been added as a list, which is used as an access list to the number of tasks that is required to finish the different dependencies.

List is empty = null
AddTask(Action, 5) An action and the name five.
List = 0,0,0,0,1
AddTask(Action, 5) again same action, same name.
List = 0,0,0,0,2
AddTask(OtherAction, 5) different action, same name.
List = 0,0,0,0,3
AddTask(NewAction, 7, Dependencies(5)) This task's name is seven, and it depends upon the name five.
List = 0,0,0,0,3,0,1
The newly added task cannot be executed before the first three tasks has been executed with the name five, resulting in:
List = 0,0,0,0,0,0,1
Then the last added task can be executed.
List = 0,0,0,0,0,0,0

Figure 3.1: This illustrates how the dependency list works.

The explanation of dependency can be seen at figure 3.1.

All of the threads are background threads the programmer has to use *WaitForTasks* that will stop waiting till all tasks has finished.

For another abstraction of the algorithm take a look at figure 3.2 and 3.3. Where it starts in *Start Main Thread Execution* which is entered with the main thread to add tasks to the scheduler. The other method of adding the tasks is to start up the scheduler in the case it sleeps or it has not been initiated. After the first *AddTask* it will start up the Scheduler *Scheduler Thread Start* where it will traverse the list and add all the possible tasks to threads, to afterwards traverse tasks again to remove them in case any have finished. When the list is empty it will finish the scheduler thread, and no additional threads but the main thread will be remaining.

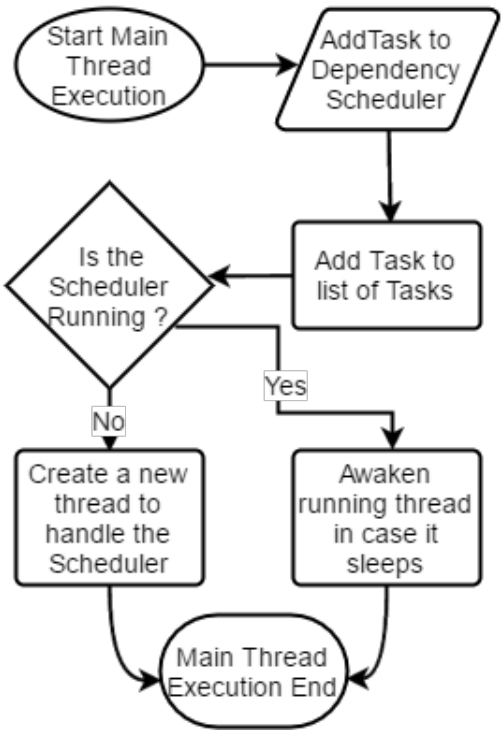

Figure 3.2: A flow diagram over the main thread *AddTask* interaction.

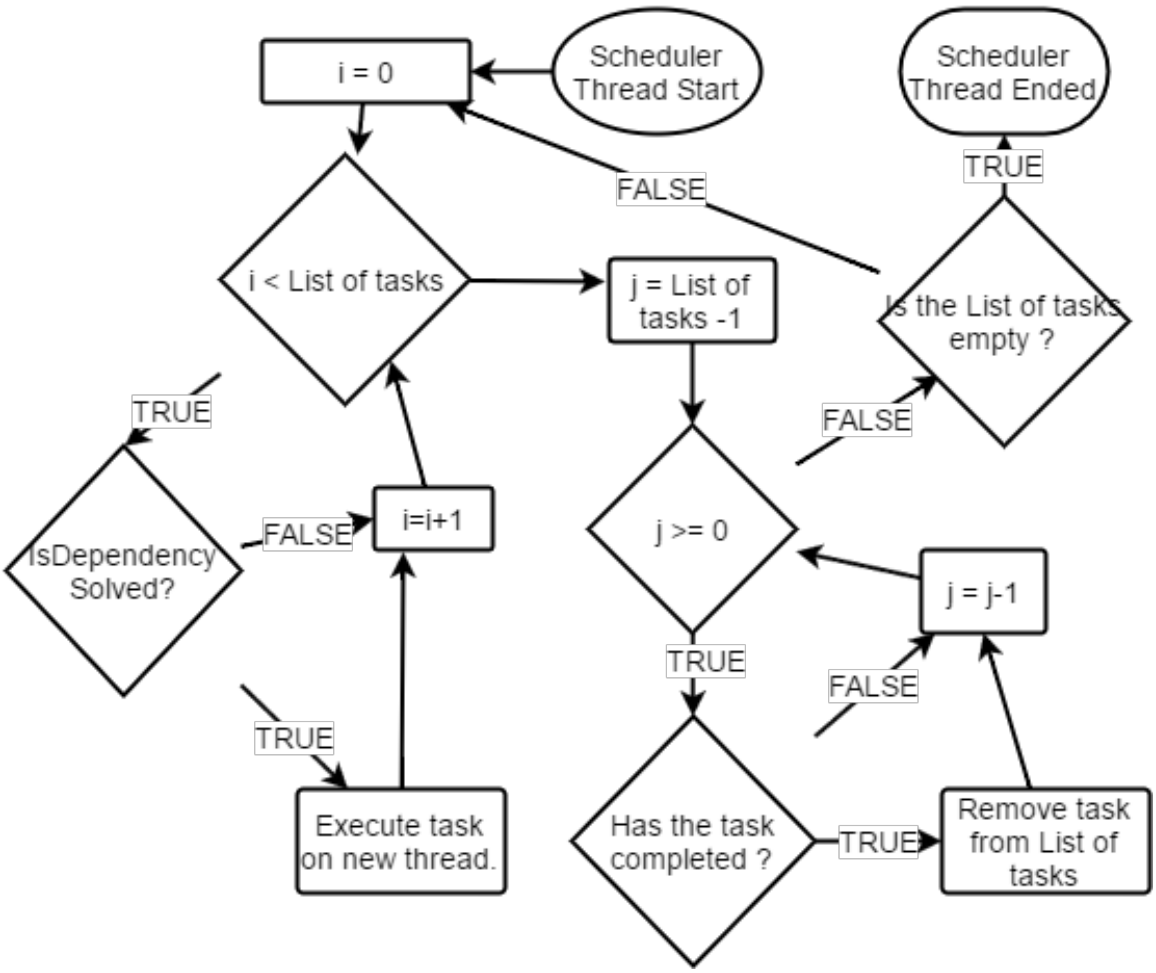

Figure 3.3: A simplified flow diagram over the internal dependency scheduler.

3.3 Architecture of the Dependency Scheduler

The consideration made was from the observations of task groups. Where in some optimal task groups one person will delegate works tasks to available team members. This ideology can resemble concurrent programming where the main thread delegates the task to other threads, decided by the programmer. Then the idea came why not create a scheduler that does the programmers job of deciding when a thread is created and used. This way multitasking will increase in abstraction and remove a part of the programmers problem.

The dependency scheduler require several elements:

- *Thread pool*: A thread pool which handles all the threads. The handling of threads includes, to create the threads, give information on how many logical threads the processor has and when the threads are terminated.

- *Tasks*: A task that contains information on what the threads have to execute.
- *Task List*: A task list that contains the created tasks.
- *Add Tasks*: There is required a way to add the tasks to the task list before a task can be used.
- *Scheduler*: A scheduler that provides an algorithm on how the threads will use the tasks from the task list.
- *Wait For Tasks*: It is required to get information on when the tasks has ended.
- *Remove Tasks from Task List*: To get information on when the tasks has finished, the task list can be emptied.
- *Stop/Start the Scheduler*: The scheduler should not use more computational power than required so a way to stop and start the scheduler is required.
- *Dependency*: The dependency require a way for a task to come before another task or multiple tasks.

3.4 Parser for UPPAAL

In UPPAAL there are many ways to solve the analysis part of multiple tasks, this section will introduce one way to create such a system. Figure 3.6 represents the average function, which might have dependencies and it takes a specific amount of time which y represents. Figure 3.8 represents the physical cores of the CPU or the logical threads, which each can execute a task such as figure 3.6. For the two systems to work together they need checks seen at figure 3.7 and 3.9. This code interact with the global declaration seen at figure 3.4. With these components a dependency system is created in UPPAAL. UPPAAL can proceed to run simulation to see if it ends in a deadlock. Another possibility is to use the verification where a query can be added, this query will be checked in every possible situation to be checked if it is true.With this you can ensure that the tasks you have analysed do not end up in a deadlock.

The parser for UPPAAL is a simple implementation where, it corresponds to the names given in the global declaration in UPPAAL. Such as for example figure: 3.5.

```
const int Cores = 6;
typedef int[0,Cores-1] core_t;
chan Done[core_t];
const int Functions = 6;
typedef int[0,Functions-1] function_t;
int IDPass;
bool inUse = false;
bool valuePassed = false;
bool Finished[Functions]={false,false,false,false,false,false};
urgent chan Work[core_t];
bool depend[Functions][Functions] =
{
{false,false,false,false,false,false}, //SomeFunction
{true,false,false,false,false,false}, //SomeFunction2
{false,false,false,false,false,false}, // HelloWorld
{false,false,false,false,false,false}, // HelloWorld2
{true,true,false,false,false,false}, // SomeFunctionWorld
{true,true,false,false,true,false} // SomeFunctionWorld2
};
```

Figure 3.4: The Declaration(Erklæring) contains the Dependency.

```
CPU(const core_t t) = CPUCore(t);
SomeFunction = Function(0);
SomeFunction2 = Function(1);
HelloWorld = Function(2);
HelloWorld2 = Function(3);
SomeFunctionWorld = Function(4);
SomeFunctionWorld2 = Function(5);

system CPU,SomeFunction,SomeFunction2,HelloWorld,HelloWorld2,
SomeFunctionWorld,SomeFunctionWorld2;
```

Figure 3.5: The System Declaration(SystemErklæring) contains the different systems from UPPAAL which can be given any name.

As seen on 3.5 a function can be named this way. With the line "*system CPU, SomeFunction, SomeFunction2, HelloWorld, HelloWorld2, SomeFunctionWorld, SomeFunctionWorld2,*" can be copied into C# as a string after removing "system" and "CPU," and the semicolon.

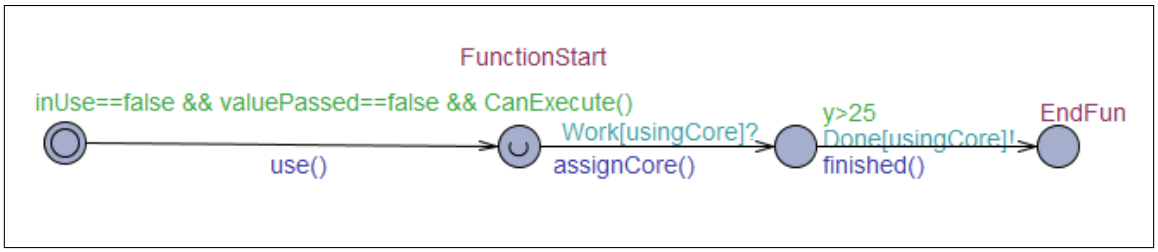

Figure 3.6: This is a representation of an average function.

```
clock y;
function_t f;
int usingCore;
void use() {if(valuePassed == false) {usingCore = IDPass; valuePassed = true;}}
void assignCore() {inUse = true; y=0;}
void finished() {Finished[id] = true;}
bool CanExecute(){
  return forall (i : function_t) ((depend[id][i] && Finished[i]) ||
 !depend[id][i]);
}
```

Figure 3.7: The System Declaration(SystemErklæring) for the average function on how it works.

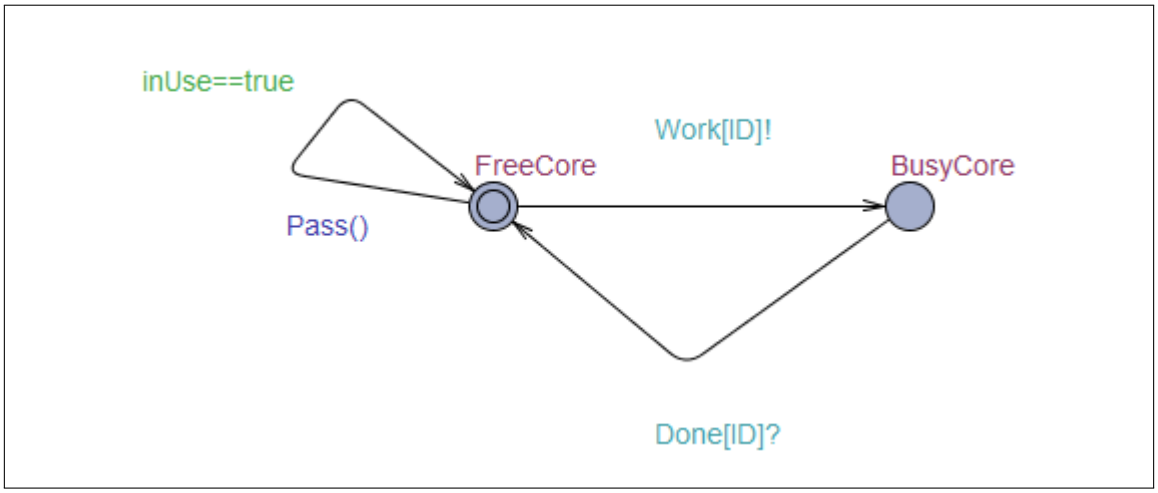

Figure 3.8: This is a representation of the cores or logical threads of the CPU.

```
core_t c;
int local;
int i;
void Pass() {
        inUse = false;
        IDPass = ID;
        valuePassed = false;
}
```

Figure 3.9: The System Declaration(SystemErklæring) for the CPU core/thread and its code on how it works.

For the dependency of the above string look at figure 3.4 which contains the dependency. For using dependency in C# it should be changed to jagged edges array: "*bool[][] depend = new bool[6][]*" "6" corresponding to the number of functions and in front of every line it require a "*new bool[]*" before "*{* ". Now the start parameters has been prepared, the algorithm can be executed at listing 3.1 and the functions it uses are found at 3.2.

Listing 3.1: This code is using a parser for UPPAAL.

```
1  DependencyTaskParser DTP = new DependencyTaskParser(DS
       );
2  UPPAAL U = new UPPAAL();
3  string uppaalMethods = "SomeFunction,SomeFunction2,
       HelloWorld,HelloWorld2,SomeFunctionWorld,
       SomeFunctionWorld2";
4  bool[][] dependUppaal = new bool[6][]
5  {
6      new bool[] {false,false,false,false,false,false},
           //SomeFunction
7      new bool[] {true,false,false,false,false,false},
           //SomeFunction2
8      new bool[] {false,false,false,false,false,false},
           // HelloWorld
9      new bool[] {false,false,false,false,false,false},
           // HelloWorld2
10     new bool[] {true,true,false,false,false,false}, //
           SomeFunctionWorld
11     new bool[] {true,true,false,false,true,false} //
           SomeFunctionWorld2
12 };
```

```
13              DTP.GiveAnalysis(typeof(UPPAAL), U, uppaalMethods,
                    dependUppaal);
14              DS.WaitForTasks();
```

Listing 3.2: These are the UPPAAL functions used.

```
1     class UPPAAL
2     {
3         int a = 0;
4         public void SomeFunction()
5         {
6             a+=250;
7         }
8         public void SomeFunction2()
9         {
10            a += 50;
11        }
12        public void HelloWorld()
13        {
14            Console.WriteLine("Hello World");
15        }
16        public void HelloWorld2()
17        {
18            Console.WriteLine("Hello Worlds");
19        }
20        public void SomeFunctionWorld()
21        {
22            Console.WriteLine("Hello World" + a);
23            a += 20;
24        }
25        public void SomeFunctionWorld2()
26        {
27            Console.WriteLine("Hello World" + a);
28        }
29    }
```

Listing3.1 will always result in "Hello World300" and "Hello World320" is written in that order and "Hello World" and "Hello Worlds" can come in a random order before, after or in between.

3.5 Writing a Parser for the Scheduler

The parser is a user written function which uses the scheduler. The parser is a interpreter from one type of input to another input readable for the scheduler. For every type of different input, a new parser is required. Another function a parser could have, could be to merge tasks together. An example can be seen at listing 3.3.

Listing 3.3: These are user parser function

```
1   class TaskParser
2   {
3       int _name,_row,_col;
4       List<int> _dependency = new List<int>();
5       string _method;
6       Sheet _sheet;
7       public TaskParser(int name, List<int> dependency, string
            methodName)
8       {
9           _name = name;
10          _dependency = dependency;
11          _method = methodName;
12      }
13      public int GetName() { return _name; }
14      public string GetMethod() { return _method; }
15      public List<int> GetDependency() { return _dependency; }
16  }
17
18  class DependencyTaskParser
19  {
20      //Using the Dependency Scheduler
21      DependencyScheduler DS;
22      //Creates an offset for the different names
23      int _offset = 1000, _resetNumber;
24
25      public DependencyTaskParser(DependencyScheduler
            DependencyScheduler)
26      {
27          DS = DependencyScheduler;
28          DS.IncreaseDependencyList(_offset);
29      }
30
31      public DependencyTaskParser(DependencyScheduler
            DependencyScheduler,int offset)
```

```
32          {
33              DS = DependencyScheduler;
34              _offset = offset;
35              _resetNumber = offset;
36              DS.IncreaseDependencyList(_offset);
37          }
38
39          //Reset will continue to increase however, this allows it
                to be reset.
40          public void ResetOffset()
41          {
42              _offset = _resetNumber;
43          }
44
45          /// <summary>
46          /// Give the Classtype from where the Tasks are from, the
                class they operate on, the list of systems from UPPAAL
                 where you have renamed to the function names,
                dependency that you created in UPPAAL
47          /// </summary>
48          public void GiveAnalysis(Type classType, object initClass,
                 string methodNames, bool[][] dependency)
49          {
50              DS.AddingTaskLock();
51              string[] splitter = methodNames.Split(',');
52              int startoffset = _offset;
53
54              List<TaskParser> ArrayTaskParser = new List<TaskParser
                    >();
55              bool[] accessList = new bool[splitter.Length];
56              DS.IncreaseDependencyList(_offset + splitter.Length);
57              for (int i = 0; i < splitter.Length; i++)
58              {
59                  List<int> dependencyList = new List<int>();
60                  for (int j = 0; j < splitter.Length; j++)
61                  {
62                      if (dependency[i][j])
63                      {
64                          dependencyList.Add(j + startoffset);
65                      }
66                  }
67
68                  if (dependencyList.Count <= 0)
```

```
69                  {
70                      DS.AddTaskNamed(classType, RemoveWhitespace(
                            splitter[i]), initClass, _offset + i);
71                      accessList[i] = true;
72                  }
73                  else
74                  {
75                      ArrayTaskParser.Add(new TaskParser(_offset + i
                            , dependencyList, RemoveWhitespace(
                            splitter[i])));
76                      accessList[i] = false;
77                  }
78
79              }
80              TasksPartTwo(ArrayTaskParser,classType,initClass,
                    accessList);
81              _offset += splitter.Length;
82              DS.AddingTaskUnlock();
83          }
84
85          /// <summary>
86          /// This method is to be sure that the dependency is
                correctly placed so there is no risk of the dependant
                to run before the function it depends on.
87          /// This is able to run while the Scheduler is running and
                 the cleaning is disabled.
88          /// </summary>
89          private void TasksPartTwo(List<TaskParser> ArrayTaskParser
                , Type classType, object initClass, bool[] accessList)
90          {
91              bool canRun = true;
92              List<int> currentList;
93              for (int i = 0; i < ArrayTaskParser.Count; i++)
94              {
95                  canRun = true;
96                  currentList = ArrayTaskParser[i].GetDependency();
97                  for (int j = 0; j < currentList.Count; j++)
98                  {
99
100                     if (!accessList[currentList[j]-_offset])
101                     {
102                         canRun = false;
103                     }
```

```
104                 }
105                 if (canRun)
106                 {
107                     DS.AddTaskNamedAndDependencies(classType,
                            ArrayTaskParser[i].GetMethod(), initClass,
                             ArrayTaskParser[i].GetName(),
                            ArrayTaskParser[i].GetDependency());
108                     accessList[ArrayTaskParser[i].GetName() -
                            _offset] = true;
109                     ArrayTaskParser.RemoveAt(i);
110                     i--;
111                 }
112             }
113             if (ArrayTaskParser.Count > 0)
114                 TasksPartTwo(ArrayTaskParser, classType, initClass
                        , accessList);
115         }
116
117         private static string RemoveWhitespace(string str)
118         {
119             return string.Join("", str.Split(default(string[]),
                    StringSplitOptions.RemoveEmptyEntries));
120         }
121     }
```

The first class *TaskParser* is to temporary maintain data that has not been added as a task yet. *GiveAnalysis* requires the *classtype* where the methods should be called from. The *Initclass* is if there is an object it has to use to access the data if the methods are non static. MethodNames are the row from System Declaration, which will be split up on run time. To ensure regardless of number of tasks it should work with *DS.AddingTaskLock*, *DS* in this case is the Dependency Scheduler which is required to create the tasks. Then the methods are split up into separate strings to afterwards create the access list which determines which functions has been used, as the methods are required to be inserted into the correct order with their dependencies. If not the tasks can risk not seeing the dependencies and just execute the task. The dependencies are determined and changed into a list corresponding to the schedulers list. Afterwards the size of the *dependencylist* is increased. If there was no dependency the task is inserted into the scheduler right away. If not it will be inserted into a list with the *TaskParser*. In the next part it will check if the dependency is executed if the dependency is executed will add the task and remove the task from the list, if not it will continue through the list until the next execution-able element has been found. If it has

a dependency to itself or a dependency that has a dependency back and forth they will end in a deadlock.

4 Implementation

4.1 Dependency Scheduler Implementation Method Details

This will serve as a reference on how it works as well as how it is used. The full library code can be seen at section 8.2 listing 8.1

4.1.1 Technology Used

This program uses technology from *.NET 3.0* which introduced *ThreadPool* and *AutoResetEvent* - it is recommended to get the latest version of *.NET*. The system works on Windows and Linux, Mac is currently untested. This *Dependency Scheduler* also uses *Reflection* to call functions. All the threads are background threads taken from the usual system tasks in Windows, on Linux it is started with the program and limits the amount of threads.

4.1.2 Initialization

When the *Dependency Scheduler* is being initialized it can be initialized with an empty constructor which gets access to the threads existing in the *.NET* framework. This constructor also accesses *OptimalTaskSize()* which is used to determine the maximum number of threads the parallel solutions should use which is the number of threads the CPU has. Beyond this it has multiple settings which is meant to be used after the object is initialized.

Public void SetMaxThreads(int)

set the threadpool to use a maximum amount that it may use based on a number the designer gives it.

Public void SetMaxThreads()

sets the threadpool to the maximum amount from .NET, which can be assigned.

Public void SetMinThreads()

sets the threadpool to the minimum amount from .NET, which can be assigned which normally would be the amount of physical CPU threads.

Public void SetMinThreads(int)

is currently untested as thread amount can be set below the number of logical threads the CPU has. However the maximum number of threads is the more interesting part as it will use as many threads a possible.

Public void SetThreadsForMyCPU()

is the function that that sets the available threads to be only the physical threads plus one thread for the *Dependency Scheduler* itself.

4.1.3 Additional Helper Functions

Public void WriteNumberofThreads()

will write to a console how many threads the system has currently assigned to the program.

Public void IncreaseDependencyList(int size)

is used to increase the *Dependency Scheduler's* dependency list as the list is the size based on the number of dependencies, if the normal *addtask* functions are used this will be used automatically.

Public void AddingTaskLock()

this will be used if there is many tasks that has to be added, using this will insure that the scheduler does not remove finished jobs which can give access problems when two threads access the same list.

Public void AddingTaskUnlock()

this is used when the tasks have been added and the scheduler safely can remove finished jobs from the list, this will also restart the scheduler if it went to sleep.

Public void WaitForTasks()

when all tasks are done the main thread will wake up again after this has been used. Always remember to *AddingTaskUnlock()* if you locked the tasks as then the program will lock up.

Public list<int> Dependencies(params int[] dependencies)

can be used to add the dependencies.

4.1.4 User Functions

Public void AddTask(Type classType, string methodName, object InitClass, params object[] objs)

because the functions uses *Reflection* it requires the *classType* and the *methodName*, if there is an object required it should be passed with *InitClass*, if its a static function it should take a *null* parameter, *params object[]* takes all the information the function needs if the function needs any parameter. This is a task that will run as soon as possible as it does not depend on anything and nothing depends on this task.

Public void AddTaskNamed(Type classType, string methodName, objec InitClasst, int name, params object[] objs)

same as *AddTask* can be named which means other tasks can depend on this task and will be run as soon as possible.

Public void AddTaskNamedAndDependencies(Type classType, string methodName, object InitClass, int, list<int> dependencies, params object[] objs)

same as *AddTaskNamed* but another task can both depend on this task and this task can depend on as many tasks as the designer wants it to.

Public void AddTaskDependenciesNoName(Type classType, string methodName, object InitClass, list<int> dependencies, params object[] objs)

this is the same as *AddTaskNamedAndDependencies* with Named excluded as some tasks only depend on some information but nothing after it will depend on this task.

Public void ParallelForTask(Type classType, string methodName, object InitClass, int from, int to, params object[] objs)

this is the same as *AddTask* it will depend upon a method structured with two integers, for the initial integer to count upto the other integer is used to for loop. This is made to most efficiently use the threads of the CPU and to do the least amount of calls through *Reflection*.

Public void ParallelForTaskNamed(Type classType, string methodName, object InitClass, int from, int to, int name, params object[] objs)

the same as for *AddTaskNamed* where it depends on a specific structure as *ParallelForTask()*

Public void ParallelForTaskNamedAndDependencies(Type classType, string methodName, object InitClass, int from, int to, int name, list<int> dependencies, params object[] objs)

the same as for *AddTaskNamedAndDependencies* where it depends on a specific structure as *ParallelForTask()*

Public void ParallelForTaskDependenciesNoName(Type classType, string methodName, object InitClass, int from, int to, list<int> dependencies, params object[] objs)

the same as for *AddTaskDependenciesNoName* where it depends on a specific structure as *ParallelForTask()*

Public void AddTask(Action action)

This function takes an action and will execute the action, this is like a thunk. This task will run as soon as possible as it does not depend on anything and nothing depends on this task.

Public void AddTask(Action action, int name)

This function takes a action and a integer for the name. This task will run as soon as possible as it does not depend on anything and others might depend on this task.

Public void AddTask(Action action, int name, list<int> dependencies)

This function takes a action, a integer and a list of integers that is the dependencies. This task will run as soon as the dependencies has been fulfilled as others might depend on this task.

Public void AddTask(Action action list<int> dependencies)

This function takes an action and a list of integers that is the dependencies. This task will run as soon as the dependencies has been fulfilled, nothing will depend on this task.

4.1.5 TaskItem

This is an object the internal *Dependency Scheduler* uses. It contains the information for the *MethodInfo*. The information it contains is:

- If the task has been started.
- If the task has finished.
- Do the task have a name.

- What dependencies does the task have if any.
- Does it require a special object to access a method.
- All the required parameters.

Where each constructor in the object is for the minimal information or maximum amount of information the *Dependency Scheduler* needs. This can also just contain an *Action* used as a *Thunk*. This contain the same but without the special object and the extra parameters.

4.1.6 Internal Working of the Dependency Scheduler

Private void InitScheduler()

this will start up the *Dependency Scheduler* if it has not been started yet, if it has been started this will awake the scheduler in case it sleeps. All task methods will call this function. The unlock method will call this function in case there is no thread running.

Private void ThreadScheduler(Object threadContext)

Is called upon from *InitScheduler()* where it will start up and initialize a waitHandle for the *WaitForTasks()* which will be used to wake up the main thread. After this it will call *SchedulerWork()* which will handle the tasks. After it has been run it will check if the work list is empty if it is it will wake up the main thread and the scheduler thread. If the tasklist is not empty the scheduler will begin to wait. if the tasklist is empty after the wait it will stop the scheduler, else it will continue to run.

Private void SchedulerWork()

will check if there is any jobs. If there is a job it will traverse the list and check for dependencies, if the dependencies has been fulfilled or there is none it will call upon a thread from the threadpool with the task *ThreadInvoker(Object threadContext)*. All available jobs will be occupied, then it will start to remove the finished jobs to empty out the list. If it removed something from the list is will start the *SchedulerWork()* again else it will go back to wait.

Private void ThreadInvoker(Object threadContext)

this will call *ThreadExecution(TaskItem)* to execute the code, it will then write to the *TaskItem* and will awake the scheduler thread to remove this job or look if a new job is ready.

Public Virtual void ThreadExecution(TaskItem TI)

this will take the information from *TaskItem* to call the *Reflection* or the *Action(thunk)*. This can be overwritten by the programmer if another behaviour is wanted with the use of TaskItem.

4.1.7 Parser

This currently has one function:*GiveAnalysis(Type classType, object initClass, string methodNames, bool[][] dependency)* GiveAnalysis is based upon information you give UPPAAL. The parser has *ParserItem* instead of *TaskItem* to create the tasks in the correct order. The Parser takes the *Dependency Scheduler* as an object which can give information to the scheduler.

Public void GiveAnalysis(Type classType, object initClass, string methodNames, bool[][] dependency)

First it will take a string of information from the methods which will be run. This will be split into a numbers of functions. If there is a bool[] for the method it will look into it to determine if there is a dependency if there is it will be converted to a list<int> containing the number of the functions which it depends upon. It will then add the task to the scheduler if it does not depend on anything. If it depend on something all its information will be put into a list. It will then traverse the list and if what it depends on has been given to the Scheduler, it will give the next piece of information and remove the current item from the list. This will continue until it has added all tasks.

4.2 Examples

This section will illustrate small examples of how the code can be used. They will all share the same code that they will use for illustration purposes.

Listing 4.1: These are some simple methods and a enum they share for the examples.

```
1     public enum MethodNames
2     {
3         Default = 0,
4         HelloWorld,
5         DoSomething,
6         UseParameter
7     };
8
9     public class Methods
```

```
10      {
11          public static void HelloWorld()
12          {
13              Console.WriteLine("Hello World!");
14          }
15
16          public static void DoSomething()
17          {
18              int i = 5 * 20;
19              Console.WriteLine("i is 5*20 equal: {0}", i);
20          }
21
22          public static void Default()
23          {
24              Console.WriteLine("This function is named default");
25          }
26
27          public void UseParameter(string s)
28          {
29              Console.WriteLine(s);
30          }
31      }
```

Listing 4.2: These are the start up to use the dependency Scheduler and to use the methods UseParameter.

```
1               DependencyScheduler DS = new DependencyScheduler();
2               Methods M = new Methods();
```

As a standard the library DLL also has to be included into References(Library) to use the dependency scheduler, and *Using Dependency_Scheduler* at the directives. For the *Reflection* solution it will also be necessary to add *Using System.Reflection* to the directives.

4.2.1 Dependency Scheduler Reflection

This is a part of the standard solution which uses *reflection*.

Listing 4.3: This code is an example of Reflection being used from the library.

```
1               DS.AddingTaskLock();//Lock so the tasks cannot be
                    removed from the task list
2               DS.AddTask(typeof(Methods), "HelloWorld", null); //Has
                     no name
```

```
3           DS.AddTask(typeof(Methods), "DoSomething", null); //
                Has no name
4           DS.AddTask(typeof(Methods), "Default", null); //Has no
                 name
5           DS.AddTask(typeof(Methods), "UseParameter", M,"Write
                Me"); //Has no name
6           DS.AddingTaskUnlock();//Unlock so the tasks can be
                removed from the task list again.
7           DS.WaitForTasks(); //Wait for the above tasks to
                finish.
8           //With Dependencies
9           Console.WriteLine("Now With Dependencies");
10          DS.AddingTaskLock();//Lock so the tasks cannot be
                removed from the task list
11          DS.AddTaskNamed(typeof(Methods), "HelloWorld", null, (
                int)MethodNames.HelloWorld); //Adding a task with
                a name
12          DS.AddTaskNamedAndDependencies(typeof(Methods), "
                DoSomething", null, (int)MethodNames.DoSomething,
                DS.Dependencies((int)MethodNames.HelloWorld)); //
                Adding a task with a name and a dependency
13          DS.AddTaskDependenciesNoName(typeof(Methods), "Default
                ", null, DS.Dependencies((int)MethodNames.
                DoSomething, (int)MethodNames.HelloWorld)); //
                adding a task with no name but with a dependency
14          DS.AddTaskDependenciesNoName(typeof(Methods), "
                UseParameter", M, DS.Dependencies((int)MethodNames
                .HelloWorld), "Write Me"); //adding a task with no
                 name but with a dependency
15          DS.AddingTaskUnlock();//Unlock so the tasks can be
                removed from the task list again.
16          DS.WaitForTasks(); //Wait for the above tasks to
                finish.
```

For *DS.AddTask* in listing 4.3 it requires the class it will call the method from, then the name of the method that should be called the name could be ”HelloWorld" as in listing 4.3. If it is a static method its object should be *null*, whereas if it should use an object like *Methods* then the object should be passed as in listing 4.3 line 4 which is the *Methods* created in listing 4.2 line 2. If the task requires additional parameter they can just be added, one after another like *Console.WriteLine*. For a concrete example take a look on listing 4.3 line 4, it uses the class *Methods* (listing 4.1 and its method *UseParameter*, this is not a

static method that is why we need the object *Methods* listing 4.2 which will be passed to the reflection class and at last the parameter or parameters which are needed for the method.

The first Line is the task lock added to ensure there will not be any problems internally in the dependency scheduler. The next 4 lines are 4 methods with no dependency. Then on line 6 the lock is unlocked making the scheduler able to delete tasks again. On line 7 the main thread will wait until all tasks have been completed. For the rest of the reflection program we use dependency. Line 11 gets a name *HelloWorld* that the scheduler will also use internally. Line 12 requires that *HelloWorld* has been run before *DoSomething* can be executed. Line 13 No one is dependant on *Default* but it depends on both *HelloWorld* and *DoSomething*. Line 14 No one is dependant on *UseParameter* but it depends on *HelloWorld* and can be run after *HelloWorld* has executed.

For simplicity the next examples are structured in the exact same way as this one.

4.2.2 Dependency Scheduler Action

This is a part of the standard solution which uses *action (thunk)*.

Listing 4.4: This code is an example of Action being used from the library.

```
1  Action A1 = delegate () { Methods.HelloWorld(); }; //
       Actions the scheduler later will call.
2  Action A2 = delegate () { Methods.DoSomething(); };
3  Action A3 = delegate () { Methods.Default(); };
4  Action A4 = delegate () { M.UseParameter("Write Me");
       };
5  DS.AddingTaskLock();//Lock so the tasks cannot be
       removed from the task list
6  DS.AddTask(A1); //Adding a task with no requirement
7  DS.AddTask(A2); //Adding a task with no requirement
8  DS.AddTask(A3); //Adding a task with no requirement
9  DS.AddTask(A4); //Adding a task with no requirement
10 DS.AddingTaskUnlock();//Unlock so the tasks can be
       removed from the task list again.
11 DS.WaitForTasks(); //Wait for the above tasks to
       finish.
12 //With Dependencies
13 Console.WriteLine("Now With Dependencies");
14 DS.AddingTaskLock();//Lock so the tasks cannot be
       removed from the task list
15 DS.AddTask(A1, (int)MethodNames.HelloWorld); //Adding
       a task with a name
```

```
16              DS.AddTask(A2, (int)MethodNames.DoSomething, DS.
                    Dependencies((int)MethodNames.HelloWorld)); //
                    Adding a task with a name and a dependency
17              DS.AddTask(A3, DS.Dependencies((int)MethodNames.
                    DoSomething, (int)MethodNames.HelloWorld)); //
                    Adding a task with no name but with dependencies
18              DS.AddTask(A4, DS.Dependencies((int)MethodNames.
                    HelloWorld)); //adding a task with no name but
                    with a dependency
19              DS.AddingTaskUnlock();//Unlock so the tasks can be
                    removed from the task list again.
20              DS.WaitForTasks(); //Wait for the above tasks to
                    finish.
```

In listing 4.4 it uses action instead of Reflection as shown in listing 4.3. On line 1-4 the *Actions* gets defined. On line 5 the task lock is added, to ensure there will not be any problems internally in the dependency scheduler. On line 6-9 tasks gets added without a name, thus executing the tasks concurrent. On line 10 the task lock is unlocked, making the scheduler able to remove completed tasks. On line 11 the main thread will start to wait until all the tasks has completed and removed from the schedulers list. On line 15-18 the tasks get added again but with names, and dependencies. Line 15 the task does not have any dependencies. Line 16 the task is dependant on line 15. Line 17 is dependant on both line 15 and line 16. Line 18 is only dependant on line 15.

4.2.3 Dependency Scheduler Virtual

This is how it works, the idea is to use it with a *enum* name for the different functions like illustrated in listing 4.1 where it contains an *enum* called *MethodNames*.

Listing 4.5: This code is an example of Virtual Override being used on the library.

```
1   class DSC : DependencyScheduler
2   {
3       public override void ThreadExecution(TaskItem TI)
4       {
5           if (TI.getName() == (int)MethodNames.HelloWorld)
6               Methods.HelloWorld();
7           else if (TI.getName() == (int)MethodNames.DoSomething)
8               Methods.DoSomething();
9           else if (TI.getName() == (int)MethodNames.UseParameter
                )
10          {
```

```
11                  Methods M = (Methods)TI.getObj();
12                  M.UseParameter((string)TI.getObjs()[0]);
13              }
14              else
15                  Methods.Default();
16          }
17      }
18
19      class Program
20      {
21          static void Main(string[] args)
22          {
23              DSC DSC = new DSC();
24              Methods M = new Methods();
25              DSC.AddingTaskLock();//Lock so the tasks cannot be
                    removed from the task list
26              DSC.AddTask(null, (int)MethodNames.HelloWorld); //
                    Adding by using the name.
27              DSC.AddTask(null, (int)MethodNames.DoSomething); //
                    Adding by using the name.
28              DSC.AddTask(null); //Adding the default task with no
                    name.
29              DSC.AddTaskNamed(typeof(Methods), "", M,(int)
                    MethodNames.UseParameter, "Write This"); //Adding
                    the task with dummy arguments, and with additional
                     information required from the function.
30              DSC.AddingTaskUnlock();//Unlock so the tasks can be
                    removed from the task list again.
31              DSC.WaitForTasks();//Wait for the above tasks to
                    finish.
32              Console.WriteLine("Now With Dependencies");
33              DSC.AddingTaskLock();//Lock so the tasks cannot be
                    removed from the task list
34              DSC.AddTask(null, (int)MethodNames.HelloWorld); //
                    Adding by using the name.
35              DSC.AddTask(null, (int)MethodNames.DoSomething, DSC.
                    Dependencies((int)MethodNames.HelloWorld)); //
                    Adding by using the name and a dependency.
36              DSC.AddTask(null, DSC.Dependencies((int)MethodNames.
                    DoSomething, (int)MethodNames.HelloWorld)); //
                    Adding the default task with no name but with
                    dependency.
```

```
37              DSC.AddTaskNamedAndDependencies(typeof(Methods), "", M
                    , (int)MethodNames.UseParameter, DSC.Dependencies
                    ((int)MethodNames.HelloWorld), "Write This"); //
                    Adding by using the name and a dependency.
38              DSC.AddingTaskUnlock();//Unlock so the tasks can be
                    removed from the task list again.
39              DSC.WaitForTasks();//Wait for the above tasks to
                    finish.
40          }
41      }
```

As the former examples this will execute in the same order, however, this is different as you can override the thread execution method and make direct calls instead of indirect calls through *Reflection* and *Action*. As shown on line 28 in listing 4.5, by using the normal reflection method, objects and parameters can be passed to the *ThreadExecution*, which in turn can then be run.

4.2.4 Microsoft Scheduler

This is the standard solution from Microsoft, where the user has to decide upon them selves how the dependency should be solved.

Listing 4.6: This code is an example of Microsoft Scheduler. Taken from [22]

```
1               LimitedConcurrencyLevelTaskScheduler lcts = new
                    LimitedConcurrencyLevelTaskScheduler(12);
2               TaskFactory factory = new TaskFactory(lcts);
3               Methods M = new Methods();
4               var taskrun = factory.StartNew(() =>
5               {
6                   Methods.HelloWorld();
7               });
8               var taskrun2 = factory.StartNew(() =>
9               {
10                  Methods.DoSomething();
11              });
12              var taskrun3 = factory.StartNew(() =>
13              {
14                  Methods.Default();
15              });
16              var taskrun4 = factory.StartNew(() =>
17              {
18                  M.UseParameter("Write Me");
```

```
19              });
20              taskrun.Wait();
21              taskrun2.Wait();
22              taskrun3.Wait();
23              taskrun4.Wait();
24              //Now with Dependency
25              Console.WriteLine("Now With dependency");
26              var taskrun5 = factory.StartNew(() =>
27              {
28                  Methods.HelloWorld();
29              });
30              taskrun5.Wait();
31              var taskrun6 = factory.StartNew(() =>
32              {
33                  Methods.DoSomething();
34              });
35              var taskrun8 = factory.StartNew(() =>
36              {
37                  M.UseParameter("Write Me");
38              });
39              taskrun6.Wait();
40              taskrun8.Wait();
41              var taskrun7 = factory.StartNew(() =>
42              {
43                  Methods.Default();
44              });
45              taskrun7.Wait();
46
47          }
48      }
49      /// <summary>
50      /// Provides a task scheduler that ensures a maximum
            concurrency level while
51      /// running on top of the ThreadPool.
52      /// </summary>
53      public class LimitedConcurrencyLevelTaskScheduler :
            TaskScheduler
54      {
55          /// <summary>Whether the current thread is processing work
                items.</summary>
56          [ThreadStatic]
57          private static bool _currentThreadIsProcessingItems;
58          /// <summary>The list of tasks to be executed.</summary>
```

```
59          private readonly LinkedList<Task> _tasks = new LinkedList<
                Task>(); // protected by lock(_tasks)
60          /// <summary>The maximum concurrency level allowed by this
                scheduler.</summary>
61          private readonly int _maxDegreeOfParallelism;
62          /// <summary>Whether the scheduler is currently processing
                work items.</summary>
63          private int _delegatesQueuedOrRunning = 0; // protected by
                lock(_tasks)
64
65          /// <summary>
66          /// Initializes an instance of the
                LimitedConcurrencyLevelTaskScheduler class with the
67          /// specified degree of parallelism.
68          /// </summary>
69          /// <param name="maxDegreeOfParallelism">The maximum
                degree of parallelism provided by this scheduler.</
                param>
70          public LimitedConcurrencyLevelTaskScheduler(int
                maxDegreeOfParallelism)
71          {
72              if (maxDegreeOfParallelism < 1) throw new
                    ArgumentOutOfRangeException("
                    maxDegreeOfParallelism");
73              _maxDegreeOfParallelism = maxDegreeOfParallelism;
74          }
75
76          /// <summary>Queues a task to the scheduler.</summary>
77          /// <param name="task">The task to be queued.</param>
78          protected sealed override void QueueTask(Task task)
79          {
80              // Add the task to the list of tasks to be processed.
                     If there aren't enough
81              // delegates currently queued or running to process
                    tasks, schedule another.
82              lock (_tasks)
83              {
84                  _tasks.AddLast(task);
85                  if (_delegatesQueuedOrRunning <
                        _maxDegreeOfParallelism)
86                  {
87                      ++_delegatesQueuedOrRunning;
88                      NotifyThreadPoolOfPendingWork();
```

```
89                  }
90              }
91          }
92
93          /// <summary>
94          /// Informs the ThreadPool that there's work to be
                executed for this scheduler.
95          /// </summary>
96          private void NotifyThreadPoolOfPendingWork()
97          {
98              ThreadPool.UnsafeQueueUserWorkItem(_ =>
99              {
100                 // Note that the current thread is now processing
                        work items.
101                 // This is necessary to enable inlining of tasks
                        into this thread.
102                 _currentThreadIsProcessingItems = true;
103                 try
104                 {
105                     // Process all available items in the queue.
106                     while (true)
107                     {
108                         Task item;
109                         lock (_tasks)
110                         {
111                             // When there are no more items to be
                                    processed,
112                             // note that we're done processing,
                                    and get out.
113                             if (_tasks.Count == 0)
114                             {
115                                 --_delegatesQueuedOrRunning;
116                                 break;
117                             }
118
119                             // Get the next item from the queue
120                             item = _tasks.First.Value;
121                             _tasks.RemoveFirst();
122                         }
123
124                         // Execute the task we pulled out of the
                                queue
125                         base.TryExecuteTask(item);
```

```
126                 }
127             }
128             // We're done processing items on the current
                    thread
129             finally { _currentThreadIsProcessingItems = false;
                    }
130         }, null);
131     }
132 
133     /// <summary>Attempts to execute the specified task on the
            current thread.</summary>
134     /// <param name="task">The task to be executed.</param>
135     /// <param name="taskWasPreviouslyQueued"></param>
136     /// <returns>Whether the task could be executed on the
            current thread.</returns>
137     protected sealed override bool TryExecuteTaskInline(Task
            task, bool taskWasPreviouslyQueued)
138     {
139         // If this thread isn't already processing a task, we
                don't support inlining
140         if (!_currentThreadIsProcessingItems) return false;
141 
142         // If the task was previously queued, remove it from
                the queue
143         if (taskWasPreviouslyQueued) TryDequeue(task);
144 
145         // Try to run the task.
146         return base.TryExecuteTask(task);
147     }
148 
149     /// <summary>Attempts to remove a previously scheduled
            task from the scheduler.</summary>
150     /// <param name="task">The task to be removed.</param>
151     /// <returns>Whether the task could be found and removed
            .</returns>
152     protected sealed override bool TryDequeue(Task task)
153     {
154         lock (_tasks) return _tasks.Remove(task);
155     }
156 
157     /// <summary>Gets the maximum concurrency level supported
            by this scheduler.</summary>
```

```
158         public sealed override int MaximumConcurrencyLevel { get {
                return _maxDegreeOfParallelism; } }
159
160         /// <summary>Gets an enumerable of the tasks currently
                scheduled on this scheduler.</summary>
161         /// <returns>An enumerable of the tasks currently
                scheduled.</returns>
162         protected sealed override IEnumerable<Task>
                GetScheduledTasks()
163         {
164             bool lockTaken = false;
165             try
166             {
167                 Monitor.TryEnter(_tasks, ref lockTaken);
168                 if (lockTaken) return _tasks.ToArray();
169                 else throw new NotSupportedException();
170             }
171             finally
172             {
173                 if (lockTaken) Monitor.Exit(_tasks);
174             }
175         }
176     }
```

Microsoft Scheduler[22] includes code which has to be used for multiple threads. Microsoft Scheduler does not have any kind of dependency which is the reason why taskrun8 has been placed before taskrun7 as this did not depend upon taskrun7 to finish. *TaskScheduler* itself is only an abstract class which requires the implementation done by *LimitedConcurrencyLevelTaskScheduler*. The implementation can be found at [22].

4.3 Implementation of Scheduler into Spreadsheet

This is the dependency scheduler implemented into spreadsheet [25]. The solution that was implemented used the idea of turning the support graph around to the dependency graph. The new dependency graph could be used right away, but it was not efficient as too many tasks was created for the dependency scheduler to properly handle it. However, the first part of the solution was solved, the tasks was concurrent and independent. Then a new simpler dependency graph could be made with all the tasks that could be run concurrent and was grouped together, the same for the next task depending on, the tasks not depending on anything. Thus the tasks could be split up between the processors without too much

overhead.

The original architecture of the spreadsheet can be found at figure: 4.1. The changed architecture of the spreadsheet can be found at figure: 4.2. The dependency scheduler was changed into a singleton making all the sheets use the same dependency scheduler. The spreadsheet traverses all non-null cells and if a cell *IsNumber* it will not be included in the dependency scheduler. *IsNumber* is a bool that is true when the cell is a *NumberCell*. The support graph gathers information from all cells with calculations then make the information into a dependency graph and then analyses the results. The analysis is used in the dependency scheduler which calculates all the cells from the analysis.

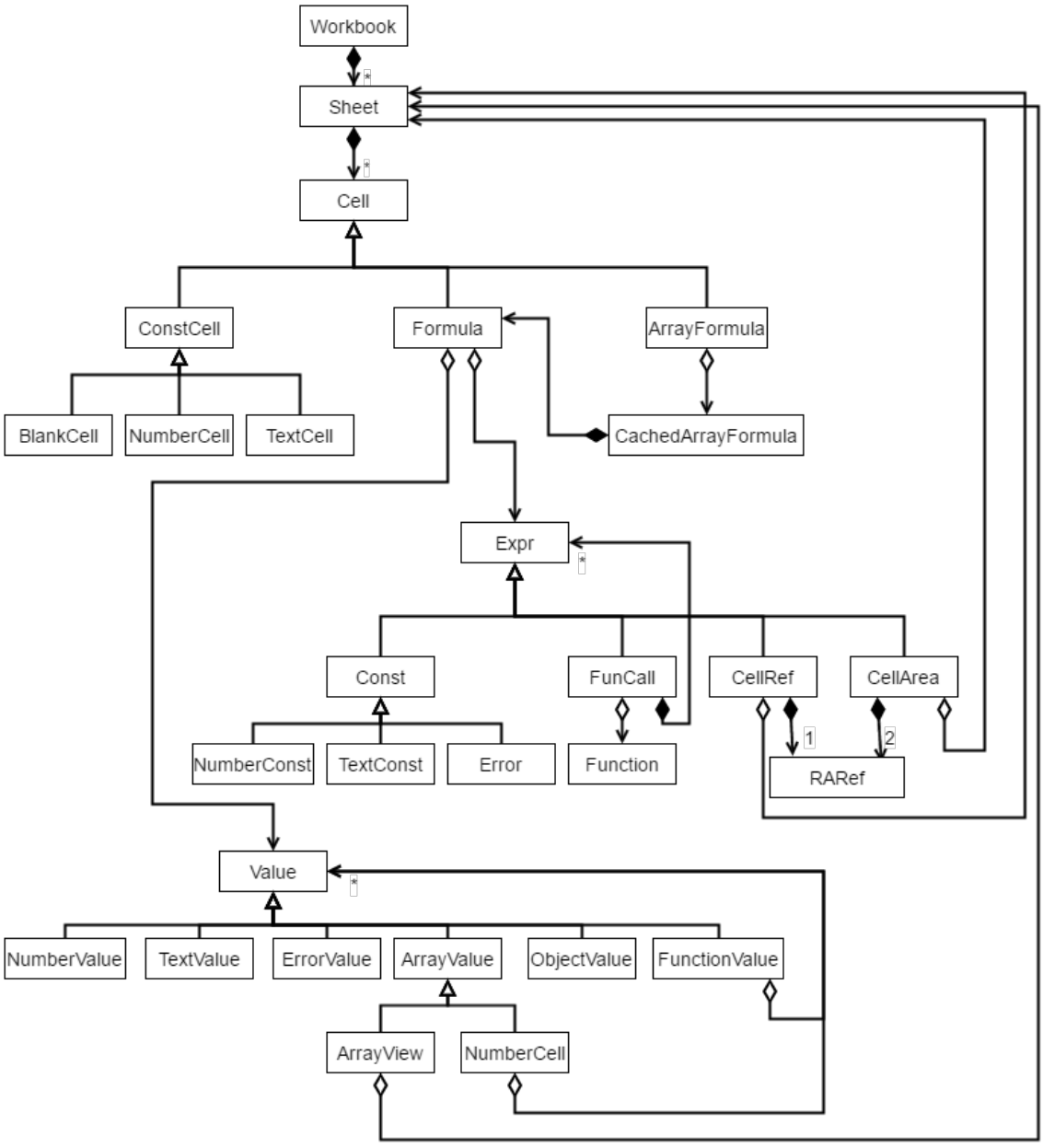

Figure 4.1: Original spreadsheet architecture replicated from [27, p.30]

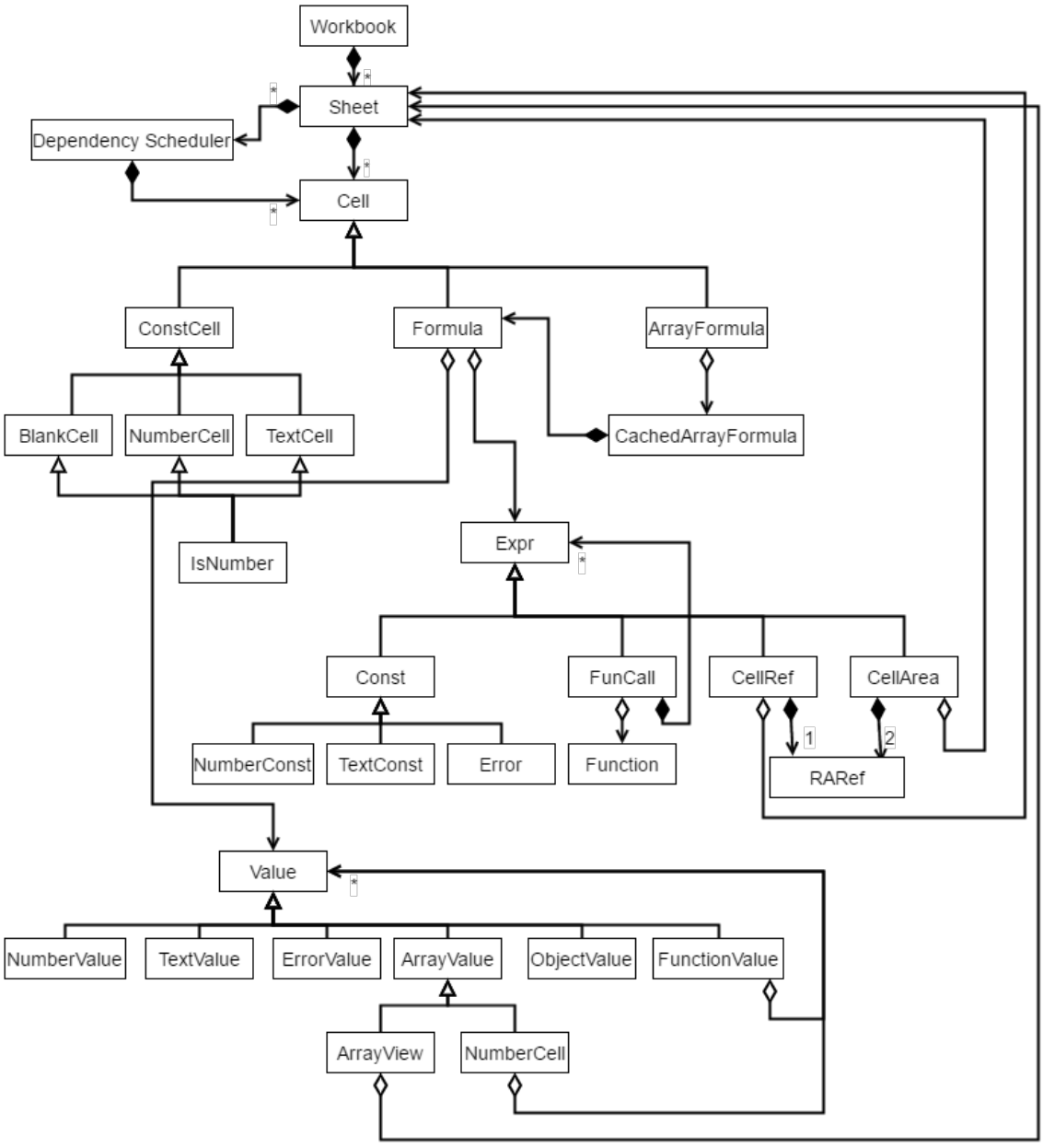

Figure 4.2: This is the changed architecture of the original spreadsheet architecture replicated from [27, p.30]

Listing 4.7: This code is just the things we save or use shortly in the analysis

```
1     private List<SupportSet> SupportSets;
2     private List<SupportSet> TempList;
3     private List<List<SupportSet>> FinishedSets;
4     private bool IsOverlapping = false, isCell = true;
5     private int[][] IDMatrix;
6     private List<int>[][] DependencyMatrix, TempDependencyMatrix;
```

```
7     private bool[][] DependencyMatrixCalc,
          TempDependencyMatrixCalc;
8     private struct xyCoordinate { public int col;public int row;};
9     private List<xyCoordinate> partTaskList = new List<
          xyCoordinate>();
10    private List<List<xyCoordinate>> FullTaskList = new List<List<
          xyCoordinate>>();
11    private int modifier = 4, MinimumNumberofTasks = 10;
12    public bool PreAnalysisFinished = false;
```

Listing 4.8: This code is the pre-analysis, with the traversal of the non-empty cells, with a new implementation to get the support set, to afterwards convert them to a dependencies. It has been chosen to use a lot of RAM for the analysis instead of extra processing power.

```
1     public void PreAnalysis()
2     {
3         int counter = 0;
4         IDMatrix = new int[Cols][];
5         DependencyMatrix = new List<int>[Cols][];
6         TempDependencyMatrix = new List<int>[Cols][];
7         DependencyMatrixCalc = new bool[Cols][];
8         TempDependencyMatrixCalc = new bool[Cols][];
9         for (int i = 0; i < Cols; i++)
10        {
11            IDMatrix[i] = new int[Rows];
12            DependencyMatrix[i] = new List<int>[Rows];
13            TempDependencyMatrix[i] = new List<int>[Rows];
14            DependencyMatrixCalc[i] = new bool[Rows];
15            TempDependencyMatrixCalc[i] = new bool[Rows];
16            for (int j = 0; j < Rows; j++)
17            {
18                IDMatrix[i][j] = counter;
19                DependencyMatrix[i][j] = new List<int>();
20                TempDependencyMatrix[i][j] = new List<int>();
21                DependencyMatrixCalc[i][j] = false;
22                TempDependencyMatrixCalc[i][j] = false;
23                counter++;
24            }
25        }
26
27        Console.WriteLine("Cols {0}, Rows {1}", Cols, Rows);
28        DependencyScheduler_Singleton DSS =
              DependencyScheduler_Singleton.Instance;
```

```
29
30          SupportSet SSTemp = null;
31          SupportCell SC = null;
32          SupportArea SA = null;
33          cells.Forall((col, row, cell) =>
34          {
35              //If the cell is a number no reason to use power on
                    confirming it is a number when the spreadsheet
                    already have done that.
36              if (!cell.isNumber())
37              {
38                  SSTemp = cell.GetSupportSet();
39                  DependencyMatrixCalc[col][row] = true;
40                  if (SSTemp != null)
41                  {
42                      List<SupportRange> ranges = SSTemp.GetRanges()
                            ;
43                      foreach (SupportRange SR in ranges)
44                      {
45                          if (SR.Count == 1)
46                          {
47                              SC = (SupportCell)SR;
48                              DependencyMatrix[SC.col][SC.row].Add(
                                    IDMatrix[col][row]);
49                          }
50                          else
51                          {
52                              SA = (SupportArea)SR;
53                              for (int i = SA.colInt.min; i <= SA.
                                    colInt.max; i++)
54                                  for (int j = SA.rowInt.min; j <=
                                        SA.rowInt.max; j++)
55                                      DependencyMatrix[i][j].Add(
                                            IDMatrix[col][row]);
56                          }
57                      }
58                  }
59              }
60          });
61          //AnalysisComplete now Set threads through the task.
62          xyCoordinate tempXY;
63          FullTaskList.Clear();
64          partTaskList.Clear();
```

```
65          do
66          {
67              //The if is here for because of the first time.
68              if (partTaskList.Count > 0)
69                  FullTaskList.Add(new List<xyCoordinate>(
                        partTaskList));
70              partTaskList.Clear();
71              for (int i = 0; i < Cols; i++)
72              {
73                  for (int j = 0; j < Rows; j++)
74                  {
75                      if (DependencyMatrixCalc[i][j])
76                      {
77                          if (DependencyMatrix[i][j].Count == 0 ||
                                DependencyMatrix[i][j].Count ==
                                TempDependencyMatrix[i][j].Count)
78                          {
79                              tempXY.col = i;
80                              tempXY.row = j;
81                              partTaskList.Add(tempXY);
82                              TempDependencyMatrixCalc[i][j] = true;
83                              DependencyMatrixCalc[i][j] = false;
84                          }
85                      }
86                  }
87              }
88              //Cleanup dependencyGraphs
89              for (int i = 0; i < Cols; i++)
90              {
91                  for (int j = 0; j < Rows; j++)
92                  {
93                      if (TempDependencyMatrixCalc[i][j])
94                      {
95                          SSTemp = cells[i, j].GetSupportSet();
96                          if (SSTemp != null)
97                          {
98                              List<SupportRange> ranges = SSTemp.
                                    GetRanges();
99                              foreach (SupportRange SR in ranges)
100                             {
101                                 if (SR.Count == 1)
102                                 {
103                                     SC = (SupportCell)SR;
```

```
104                                     TempDependencyMatrix[SC.col][
                                            SC.row].Add(IDMatrix[i][j
                                            ]);
105                                 }
106                                 else
107                                 {
108                                     SA = (SupportArea)SR;
109                                     for (int k = SA.colInt.min; k
                                            <= SA.colInt.max; k++)
110                                      for (int l = SA.rowInt.min
                                             ; l <= SA.rowInt.max;
                                             l++)
111                                       TempDependencyMatrix[k
                                              ][l].Add(IDMatrix[
                                              i][j]);
112                                 }
113                             }
114                         }
115                         TempDependencyMatrixCalc[i][j] = false;
116                     }
117                 }
118             }
119         } while (partTaskList.Count > 0);
120         PreAnalysisFinished = true;
121     }
```

Listing 4.9: This code is the actual split up of tasks that will happen aften the analysis, if the analysis has been run it will no longer run the analysis.

```
1     public void RecalculateFull() {
2
3       if (!PreAnalysisFinished)
4           PreAnalysis();
5
6       DSS.AddingTaskLock();
7       List<Action<int, int>> ActionsList = new List<Action<int,
           int>>();
8       List<int> ActionsListA = new List<int>();
9       List<int> ActionsListB = new List<int>();
10      int SplitSize = 0, count = 0;
11
12      for (int i = 0; i < FullTaskList.Count; i++ )
13      {
```

```
14          ActionsList.Clear();
15          ActionsListA.Clear();
16          ActionsListB.Clear();
17          if(FullTaskList[i].Count < MinimumNumberofTasks)//DSS.
                GetNumberofThreads())
18          {
19
20              for(int j = 0; j < FullTaskList[i].Count; j++)
21              {
22                  Action<int, int> action = delegate(int k, int l)
23                  {
24                      cells[k, l].Eval(this, k, l);
25                  };
26                  ActionsList.Add(action);
27                  ActionsListA.Add(FullTaskList[i][j].col);
28                  ActionsListB.Add(FullTaskList[i][j].row);
29
30              }
31              if(i == 0)
32                  DSS.AddTask(ActionsList, i + 1, DSS.Dependencies
                        (i), ActionsListA, ActionsListB);
33              else
34                DSS.AddTask(ActionsList, i+1, DSS.Dependencies(i),
                      ActionsListA, ActionsListB);
35          }
36          else
37          {
38              SplitSize = FullTaskList[i].Count / (DSS.
                    GetNumberofThreads() * modifier);
39              count = 0;
40              for (int j = 0; j < FullTaskList[i].Count; j++)
41              {
42                  Action<int, int> action = delegate(int k, int l)
43                  {
44                      cells[k, l].Eval(this, k, l);
45                  };
46                  ActionsList.Add(action);
47                  ActionsListA.Add(FullTaskList[i][j].col);
48                  ActionsListB.Add(FullTaskList[i][j].row);
49                  count++;
50                  if(count >= SplitSize)
51                  {
52                      if (i == 0)
```

```
53                          DSS.AddTask(ActionsList, i + 1, DSS.
                                Dependencies(i), ActionsListA,
                                ActionsListB);
54                      else
55                          DSS.AddTask(ActionsList, i+1, DSS.
                                Dependencies(i), ActionsListA,
                                ActionsListB);
56                      ActionsList.Clear();
57                      ActionsListA.Clear();
58                      ActionsListB.Clear();
59                      count = 0;
60                  }
61              }
62              if (i == 0 && ActionsList.Count > 0)
63                  DSS.AddTask(ActionsList, i + 1, DSS.Dependencies
                        (i), ActionsListA, ActionsListB);
64              else if(ActionsList.Count > 0)
65                  DSS.AddTask(ActionsList, i + 1, DSS.Dependencies
                        (i), ActionsListA, ActionsListB);
66          }
67      }
68
69      DSS.AddingTaskUnlock();
70      DSS.WaitForTasks();
71  }
```

For this spreadsheet the dependency scheduler was modified to be able to take a list of Action<int,int> with input of lists of corresponding integers. The dependency scheduler was changed into a singleton, thus it was not recreated on every single sheet but shared between every sheet. In listing 4.8 it makes the analysis of the spreadsheet with the build in support graph. This support graph was changed into a dependency graph, the dependencies was compared to see if they could run concurrent. A ID matrix was created to give a unique integer for each cell. As well as two bool multiple arrays was added, to not recalculate the same cells multiple times as the analysis iterates over the spreadsheet to change the support graphs into dependencies. The analysis takes a lot of RAM as it duplicates dependencies in lists twice, as it is faster than removing the information from the newly created dependencies, making this just a comparison of the number of dependencies in each list. In listing 4.9 If there is enough tasks it will split up the tasks in new dependencies to the number of logical threads times four on the computer, making the CPU use 100% of its capacity to calculate the spreadsheet, but if there is too few tasks it will calculate them on a

single thread. The logical threads times four has been found to be the ideal number through multiple tests.

4.4 Micro Benchmarks Implementation

This is all the micro benchmarks where every single function has been implemented as a primary part of the dependency scheduler, making the access of a few tasks differently, compared to the final solution for the Dependency Scheduler.

Listing 4.10: This code is all the different tests conducted to determine how the scheduler will work most efficiently. The tests are inspired by [26] specific on the Timer and on Mark1-3.

```
1            Tester T = new Tester();
2            int size = 100000000;
3
4            T.Initarray(size);
5            DependencyScheduler DS = new DependencyScheduler();
6            DS.WriteNumberofThreads();
7
8            Timer t = new Timer();
9
10           DS.AddingTaskLock();
11           DS.ParallelForTask(typeof(Tester), "OperateonDataFor",
                 T, 0, size);
12           DS.AddingTaskUnlock();
13           DS.WaitForTasks();
14           double runningTime = t.CheckMs();
15           T.Initarray(size);
16           Timer t2 = new Timer();
17           for (int j = 0; j < size; j++)
18           {
19               T.OperateonData(j);
20           }
21           double runningTime2 = t2.CheckMs();
22           T.Initarray(size);
23
24           Timer t3 = new Timer();
25           Parallel.For(0, size-1, k =>
26           { T.OperateonData(k); }
27           );
28           double runningTime3 = t3.CheckMs();
29           T.Initarray(size);
30
```

```
31          Timer t4 = new Timer();
32          DS.AddingTaskLock();
33          DS.ParallelForTaskReflection(typeof(Tester), "
                OperateonData", T, 0, size);
34          DS.AddingTaskUnlock();
35          DS.WaitForTasks();
36          double runningTime4 = t4.CheckMs();
37          T.Initarray(size);
38
39          Action<object> function1 = delegate (object a) { T.
                OperateonData((int)a); };
40
41          Timer t5 = new Timer();
42          DS.AddingTaskLock();
43          DS.ParallelForTaskReflection(function1, 0, size);
44          DS.AddingTaskUnlock();
45          DS.WaitForTasks();
46          double runningTime5 = t5.CheckMs();
47          T.Initarray(size);
48
49          DependChange DC = new DependChange();
50          Timer t6 = new Timer();
51          DC.AddingTaskLock();
52          DC.ParallelForTaskReflection(typeof(Tester), "
                OperateonData", T, 0, size);
53          DC.AddingTaskUnlock();
54          DC.WaitForTasks();
55          double runningTime6 = t6.CheckMs();
56          T.Initarray(size);
57
58          Action<object, object> function2 = delegate (object a,
                 object b) { T.OperateonDataFor((int)a, (int)b);
                };
59
60          Timer t8 = new Timer();
61          DS.AddingTaskLock();
62          DS.ParallelForTaskReflection(function2, 0, size);
63          DS.AddingTaskUnlock();
64          DS.WaitForTasks();
65          double runningTime8 = t8.CheckMs();
66          T.Initarray(size);
67
```

```
68              Action action1 = delegate () { T.OperateonDataFor(0, (
                    size / 12) * 1); };
69              Action action2 = delegate () { T.OperateonDataFor((
                    size / 12) * 1, (size / 12) * 2); };
70              Action action3 = delegate () { T.OperateonDataFor((
                    size / 12) * 2, (size / 12) * 3); };
71              Action action4 = delegate () { T.OperateonDataFor((
                    size / 12) * 3, (size / 12) * 4); };
72              Action action5 = delegate () { T.OperateonDataFor((
                    size / 12) * 4, (size / 12) * 5); };
73              Action action6 = delegate () { T.OperateonDataFor((
                    size / 12) * 5, (size / 12) * 6); };
74              Action action7 = delegate () { T.OperateonDataFor((
                    size / 12) * 6, (size / 12) * 7); };
75              Action action8 = delegate () { T.OperateonDataFor((
                    size / 12) * 7, (size / 12) * 8); };
76              Action action9 = delegate () { T.OperateonDataFor((
                    size / 12) * 8, (size / 12) * 9); };
77              Action action10 = delegate () { T.OperateonDataFor((
                    size / 12) * 9, (size / 12) * 10); };
78              Action action11 = delegate () { T.OperateonDataFor((
                    size / 12) * 10, (size / 12) * 11); };
79              Action action12 = delegate () { T.OperateonDataFor((
                    size / 12) * 11, size); };
80
81              Timer t9 = new Timer();
82              DS.AddingTaskLock();
83              DS.AddLambda(action1);
84              DS.AddLambda(action2);
85              DS.AddLambda(action3);
86              DS.AddLambda(action4);
87              DS.AddLambda(action5);
88              DS.AddLambda(action6);
89              DS.AddLambda(action7);
90              DS.AddLambda(action8);
91              DS.AddLambda(action9);
92              DS.AddLambda(action10);
93              DS.AddLambda(action11);
94              DS.AddLambda(action12);
95              DS.AddingTaskUnlock();
96              DS.WaitForTasks();
97              double runningTime9 = t9.CheckMs();
98              T.Initarray(size);
```

```
99
100             Lazy<int> lazy1 = new Lazy<int>(() => T.
                    OperateonDataFor2(0, (size / 12) * 1));
101             Lazy<int> lazy2 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 1, (size / 12) *
                    2));
102             Lazy<int> lazy3 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 2, (size / 12) *
                    3));
103             Lazy<int> lazy4 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 3, (size / 12) *
                    4));
104             Lazy<int> lazy5 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 4, (size / 12) *
                    5));
105             Lazy<int> lazy6 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 5, (size / 12) *
                    6));
106             Lazy<int> lazy7 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 6, (size / 12) *
                    7));
107             Lazy<int> lazy8 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 7, (size / 12) *
                    8));
108             Lazy<int> lazy9 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 8, (size / 12) *
                    9));
109             Lazy<int> lazy10 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 9, (size / 12) *
                    10));
110             Lazy<int> lazy11 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 10, (size / 12) *
                    11));
111             Lazy<int> lazy12 = new Lazy<int>(() => T.
                    OperateonDataFor2((size / 12) * 11, size));
112
113             Timer t10 = new Timer();
114             DS.AddingTaskLock();
115             DS.AddLazy(lazy1);
116             DS.AddLazy(lazy2);
117             DS.AddLazy(lazy3);
118             DS.AddLazy(lazy4);
119             DS.AddLazy(lazy5);
```

```
120             DS.AddLazy(lazy6);
121             DS.AddLazy(lazy7);
122             DS.AddLazy(lazy8);
123             DS.AddLazy(lazy9);
124             DS.AddLazy(lazy10);
125             DS.AddLazy(lazy11);
126             DS.AddLazy(lazy12);
127             DS.AddingTaskUnlock();
128             DS.WaitForTasks();
129             double runningTime10 = t10.CheckMs();
130             T.Initarray(size);
131
132             LimitedConcurrencyLevelTaskScheduler lcts = new
                    LimitedConcurrencyLevelTaskScheduler(12);
133             TaskFactory factory = new TaskFactory(lcts);
134             Timer t11 = new Timer();
135
136             var taskrun = factory.StartNew(() =>
137             {
138                 T.OperateonDataFor(0, (size / 12) * 1);
139             });
140             var taskrun2 = factory.StartNew(() =>
141             {
142                 T.OperateonDataFor((size / 12) * 1, (size / 12) *
                        2);
143             });
144             var taskrun3 = factory.StartNew(() =>
145             {
146                 T.OperateonDataFor((size / 12) * 2, (size / 12) *
                        3);
147             });
148             var taskrun4 = factory.StartNew(() =>
149             {
150                 T.OperateonDataFor((size / 12) * 3, (size / 12) *
                        4);
151             });
152             var taskrun5 = factory.StartNew(() =>
153             {
154                 T.OperateonDataFor((size / 12) * 4, (size / 12) *
                        5);
155             });
156             var taskrun6 = factory.StartNew(() =>
157             {
```

```
158             T.OperateonDataFor((size / 12) * 5, (size / 12) *
                    6);
159         });
160         var taskrun7 = factory.StartNew(() =>
161         {
162             T.OperateonDataFor((size / 12) * 6, (size / 12) *
                    7);
163         });
164         var taskrun8 = factory.StartNew(() =>
165         {
166             T.OperateonDataFor((size / 12) * 7, (size / 12) *
                    8);
167         });
168         var taskrun9 = factory.StartNew(() =>
169         {
170             T.OperateonDataFor((size / 12) * 8, (size / 12) *
                    9);
171         });
172         var taskrun10 = factory.StartNew(() =>
173         {
174             T.OperateonDataFor((size / 12) * 9, (size / 12) *
                    10);
175         });
176         var taskrun11 = factory.StartNew(() =>
177         {
178             T.OperateonDataFor((size / 12) * 10, (size / 12) *
                     11);
179         });
180         var taskrun12 = factory.StartNew(() =>
181         {
182             T.OperateonDataFor((size / 12) * 11, size);
183         });
184         taskrun.Wait();
185         taskrun2.Wait();
186         taskrun3.Wait();
187         taskrun4.Wait();
188         taskrun5.Wait();
189         taskrun6.Wait();
190         taskrun7.Wait();
191         taskrun8.Wait();
192         taskrun9.Wait();
193         taskrun10.Wait();
194         taskrun11.Wait();
```

```
195             taskrun12.Wait();
196             double runningTime11 = t11.CheckMs();
197             T.Initarray(size);
198
199             bool[][] depend = new bool[12][]
200 {
201   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
202   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
203   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
204   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
205   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
206   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
207   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
208   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
209   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
210   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
211   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */ ,
212   new bool[] {false, false, false, false, false, false, false,
        false, false, false, false, false } /* No Dependency for
        test */
213 };
```

```
214             string methods = "OperateonDataForUPPAAL1,
                    OperateonDataForUPPAAL2, OperateonDataForUPPAAL3,
                    OperateonDataForUPPAAL4, OperateonDataForUPPAAL5,
                    OperateonDataForUPPAAL6, OperateonDataForUPPAAL7,
                    OperateonDataForUPPAAL8, OperateonDataForUPPAAL9,
                    OperateonDataForUPPAAL10, OperateonDataForUPPAAL11
                    , OperateonDataForUPPAAL12";
215
216             DependencyTaskParser DTP = new DependencyTaskParser(DS
                    );
217             Timer t12 = new Timer();
218             DTP.GiveAnalysis(typeof(Tester), T, methods, depend);
219             DS.WaitForTasks();
220             double runningTime12 = t12.CheckMs();
221             T.Initarray(size);
222
223             Thread thread = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL1));
224             Thread thread2 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL2));
225             Thread thread3 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL3));
226             Thread thread4 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL4));
227             Thread thread5 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL5));
228             Thread thread6 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL6));
229             Thread thread7 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL7));
230             Thread thread8 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL8));
231             Thread thread9 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL9));
232             Thread thread10 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL10));
233             Thread thread11 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL11));
234             Thread thread12 = new Thread(new ThreadStart(T.
                    OperateonDataForUPPAAL12));
235             Timer t13 = new Timer();
236             thread.Start();
237             thread2.Start();
```

```
238             thread3.Start();
239             thread4.Start();
240             thread5.Start();
241             thread6.Start();
242             thread7.Start();
243             thread8.Start();
244             thread9.Start();
245             thread10.Start();
246             thread11.Start();
247             thread12.Start();
248             thread.Join();
249             thread2.Join();
250             thread3.Join();
251             thread4.Join();
252             thread5.Join();
253             thread6.Join();
254             thread7.Join();
255             thread8.Join();
256             thread9.Join();
257             thread10.Join();
258             thread11.Join();
259             thread12.Join();
260             double runningTime13 = t13.CheckMs();
261
262             Console.WriteLine("Running Time For scheduler {0} ms,
                   singleThread {1} ms, parallel.for {2} ms,
                   Scheduler Reflection Parallel.for {3} ms,
                   Scheduler Lambda {4} ms, Scheduler Virtual Direct
                   Call {5} ms, Lambda OneCall Per thread {6} ms,
                   Lambda Delay per Thread {7} ms, Lazy per Thread
                   {8} ms, Microsoft Scheduler {9} ms, UPPAAL Parser
                   {10} ms, Threads with no scheduler {11} ms",
                   runningTime, runningTime2, runningTime3,
                   runningTime4, runningTime5, runningTime6,
                   runningTime8, runningTime9, runningTime10,
                   runningTime11, runningTime12, runningTime13);
```

In listing 4.10 there is 10 different tests conducted. From line 8-14 the Standard *Reflection* solution is used, which splits up the tasks to the number of logical threads on the CPU. Line 16-21 is the *Single Thread* regular calculation. Line 24-28 is the *Parallel.For* solution from Microsoft. Line 31-36 is the *Reflection.For* solution which makes every single operation to a thread called through *Reflection*. Line 39-46 is the *Lambda.For* solution which uses

a *Action<object>* and makes a thread for every single operation. Line 49-55 is the *VirtualOverride* which splits up as the Standard *Reflection* solution. Line 58-65 is the *Lambda Standard* which uses *Action<object,object>* with boxing and unboxing but only as many method calls as VirtualOverride or Standard *Reflection*. Line 68-97 is the *Action(Thunk)* which uses *Action* where the programmer has to predefine what it has to execute. Line 100-129 is the *Lazy Evaluation* which uses *Lazy<int>* which requires a return result, but as the results was written to an object it returned zero. Line 132-196 is the *Microsoft Scheduler* which has to be split up, and wait for each task, unless they are added to a list. Line 199-220 is the *UPPAAL Parser*, which uses a predetermined split up data as the lambda uses. Line 223-260 is the *Threads No Scheduler* solution which uses plain threads to do the same job as the other tasks. The threads should have minimal overhead as they are created just before the timer.

All the above tasks was executed independent from one another, without any dependency to determine current speed of the scheduler compared to Microsoft Scheduler.

5 Benchmark

This chapter is about the micro benchmark and the benchmark that have been conducted. All the tables for micro benchmarks can be found at 8.1.

5.1 The Systems

In this report the most relevant component will be the processor. Some of the tests also use RAM access, to save results temporary or for the duration of the program. GPU is shown as LibreOffice chooses the piece of hardware with most computational power, in most cases it will pick the APU's that both the Laptops have.

System Name	Processor	Amount of RAM	DDR3 or DDR4 Frequency	GPU
Desktop	i7-5930K	32 GB	DDR4 2666Mhz	Club3D 49 290x RoyalAce 4GB VRAM
Laptop 1	i7-6700HQ	16 GB	DDR4 2133Mhz	Geforce GTX 970m
Laptop 2	i5-3230M	8 GB	DDR3 1600 Mhz	Geforce GT 635M

Table 5.1: The systems that have been tested on.

5.2 Micro Benchmarks

There was conducted four micro benchmarks:5.1,5.2,5.3,5.4. These tests will be run through 10 times for each of the different tests. The larger amount of tests will give better estimations of the average result. The tests conducted are to figure out what the best implementations of the dependency scheduler would be, all the implementations can be found at section 4.4:

- *Standard*: The use of *Reflection* with the scheduler, with the standard method being to split the task up into the amount of threads available from the processor.
- *Single Thread*: The regular calculation of the results without the scheduler.
- *Parallel.For*: The function from Microsoft that it used as a normal for loop but with parallelism.
- *Reflection.For*: Using the scheduler to add every calculation as its own task.

- *Lambda.For*: Using the scheduler with Lambda.For where every calculation has its own task.
- *VirtualOverride*: Using virtual Override on the schedulers ThreadExecution and make direct calls instead of action or reflection calls.
- *Lambda Standard*: Using Lambda as the standard solution with only creating the optimal number of tasks.
- *Action (Thunk)*: Using Action through the scheduler to calculate the results, split up to a static amount of tasks.
- *Lazy Evaluation*: Using Lazy evaluation together with the scheduler, with a static amount of tasks.
- *Microsoft Scheduler*: The comparable solution from Microsoft, with static amount of threads and tasks.
- *UPPAAL Parser*: This is the UPPAAL Parser, which uses *Reflection*.
- *Threads No Scheduler*: Is the solution that is with created threads and started with the timer, this should be close to most optimal solution with no overhead.

This test is conducted with cache use, it is created to give an idea on how well it runs with different techniques and how does it compare to an existing solution.

Listing 5.1: This is micro benchmark code for double writing to RAM. The tests are inspired by [26] specific on the multiply test.

```
1         public void OperateonDataFor(int low, int high)
2         {
3             for (int i = low; i < high; i++)
4             {
5                 testarray[i] = 1.1 * (double)(i & 0xFF);
6                 testarray[i] = testarray[i] * testarray[i] *
                      testarray[i] * testarray[i] * testarray[i] *
7                     testarray[i] * testarray[i] * testarray[i] *
                          testarray[i] * testarray[i] *
8                     testarray[i] * testarray[i] * testarray[i] *
                          testarray[i] * testarray[i] *
9                     testarray[i] * testarray[i] * testarray[i] *
                          testarray[i] * testarray[i];
10            }
```

```
11          }
```

Listing 5.2: This is micro benchmark code for double without writing to RAM. The tests are inspired by [26] specific on the multiply test.

```
1           public void OperateonDataFor(int low, int high)
2           {
3               for (int i = low; i < high; i++)
4               {
5                   double testarray = 1.1 * (double)(i & 0xFF);
6                   testarray = testarray * testarray * testarray *
                        testarray * testarray *
7                       testarray * testarray * testarray * testarray
                            * testarray *
8                       testarray * testarray * testarray * testarray
                            * testarray *
9                       testarray * testarray * testarray * testarray
                            * testarray;
10              }
11          }
```

Listing 5.3: This is micro benchmark code for int writing to RAM. The tests are inspired by [26] specific on the multiply test.

```
1           public void OperateonDataFor(int low, int high)
2           {
3               for (int i = low; i < high; i++)
4               {
5                   testarray[i] = 1 * (int)(i & 0xFF);
6                   testarray[i] = testarray[i] * testarray[i] *
                        testarray[i] * testarray[i] * testarray[i] *
7                       testarray[i] * testarray[i] * testarray[i] *
                            testarray[i] * testarray[i] *
8                       testarray[i] * testarray[i] * testarray[i] *
                            testarray[i] * testarray[i] *
9                       testarray[i] * testarray[i] * testarray[i] *
                            testarray[i] * testarray[i];
10              }
11          }
```

Listing 5.4: This is micro benchmark code for int without writing to RAM. The tests are inspired by [26] specific on the multiply test.

```
1         public void OperateonDataFor(int low, int high)
2         {
3             for (int i = low; i < high; i++)
4             {
5                 int testarray = 1 * (int)(i & 0xFF);
6                 testarray = testarray * testarray * testarray *
                      testarray * testarray *
7                     testarray * testarray * testarray * testarray
                          * testarray *
8                     testarray * testarray * testarray * testarray
                          * testarray *
9                     testarray * testarray * testarray * testarray
                          * testarray;
10            }
11        }
```

5.3 Micro Benchmarks Graphs

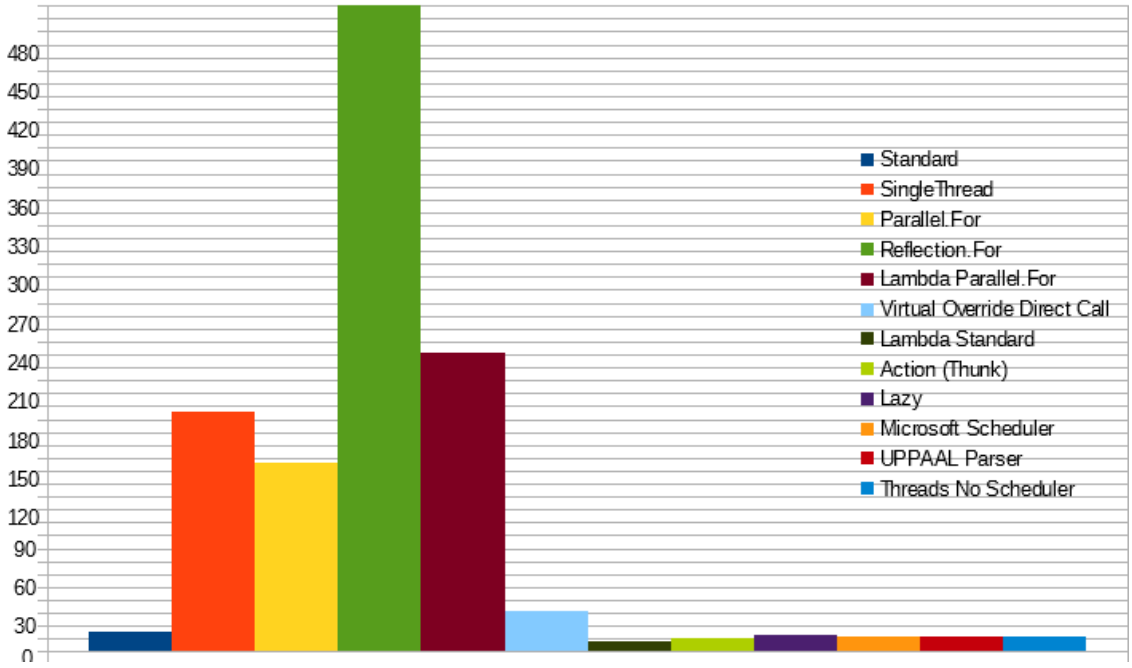

Figure 5.1: Double not written to RAM, listing5.2 all the numbers are measured in milliseconds. From table 8.13. Laptop 1 5.1. Lower is better.

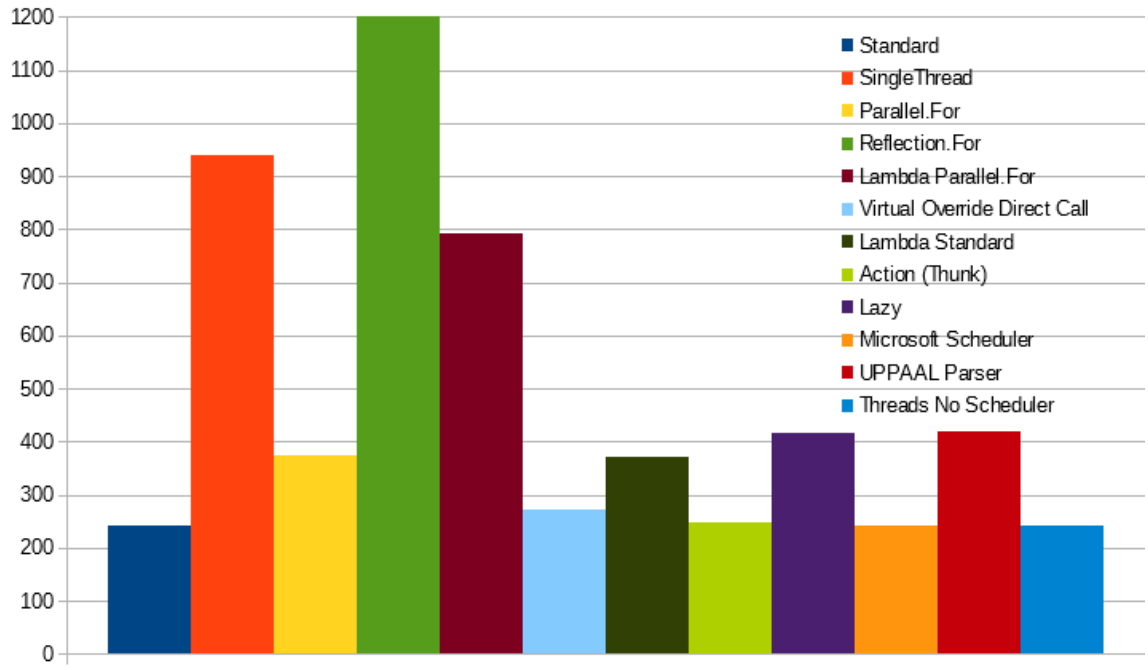

Figure 5.2: Double written to RAM, listing5.1 all the numbers are measured in milliseconds. From table 8.14. Laptop 1 5.1. Lower is better.

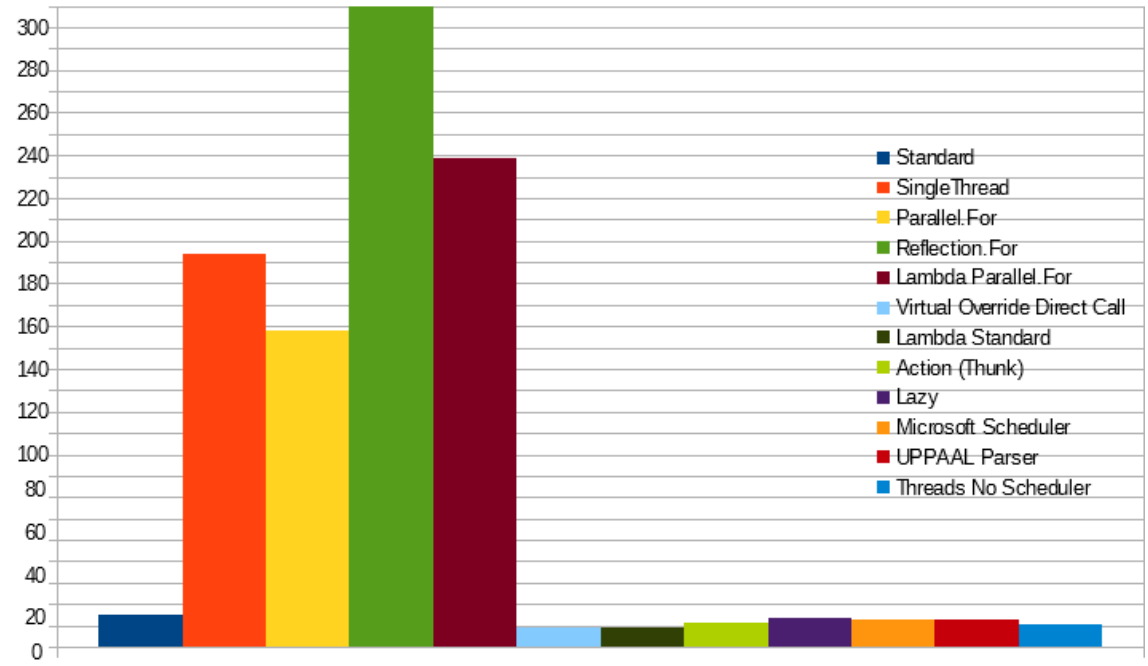

Figure 5.3: Integer not written to RAM, listing5.4 all the numbers are measured in milliseconds. From table 8.15. Laptop 1 5.1. Lower is better.

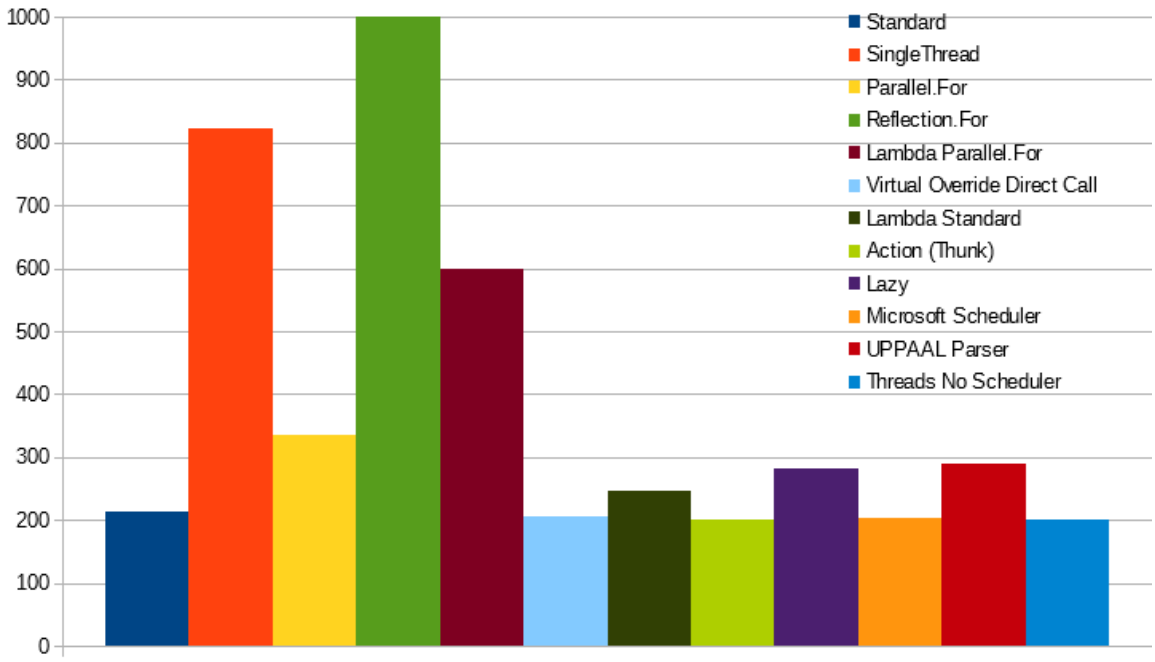

Figure 5.4: Integer written to RAM, listing5.3 all the numbers are measured in milliseconds. From table 8.16. Laptop 1 5.1. Lower is better.

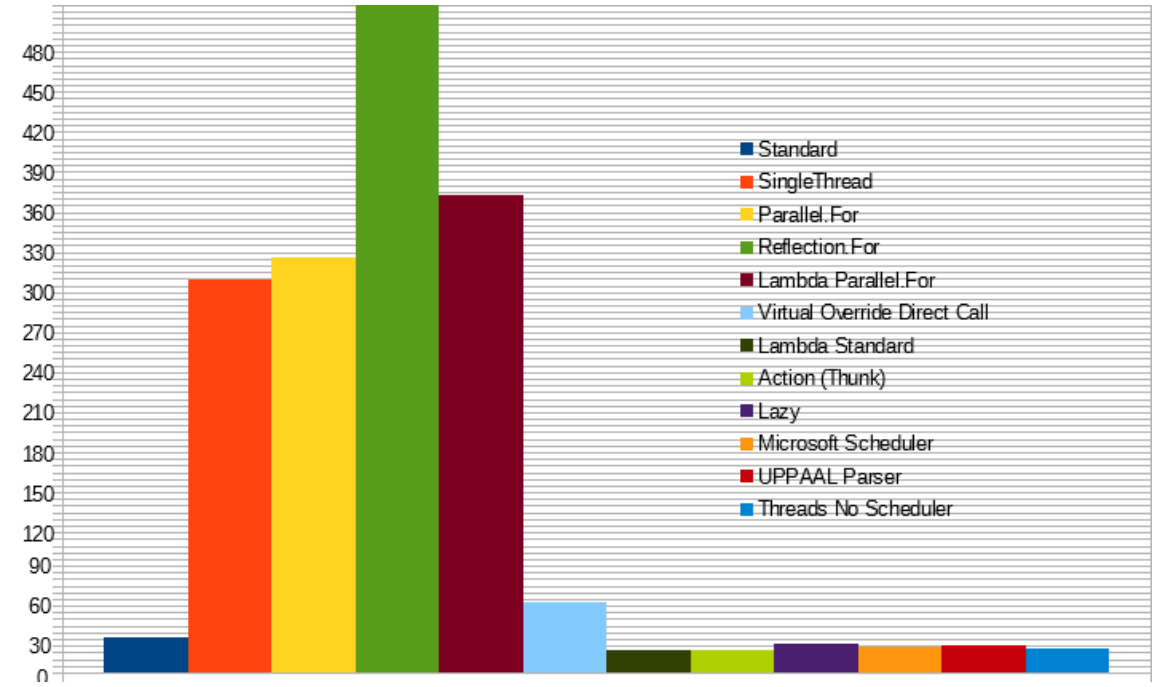

Figure 5.5: Double not written to RAM, listing5.2 all the numbers are measured in milliseconds. From table 8.21. Laptop 2 5.1. Lower is better.

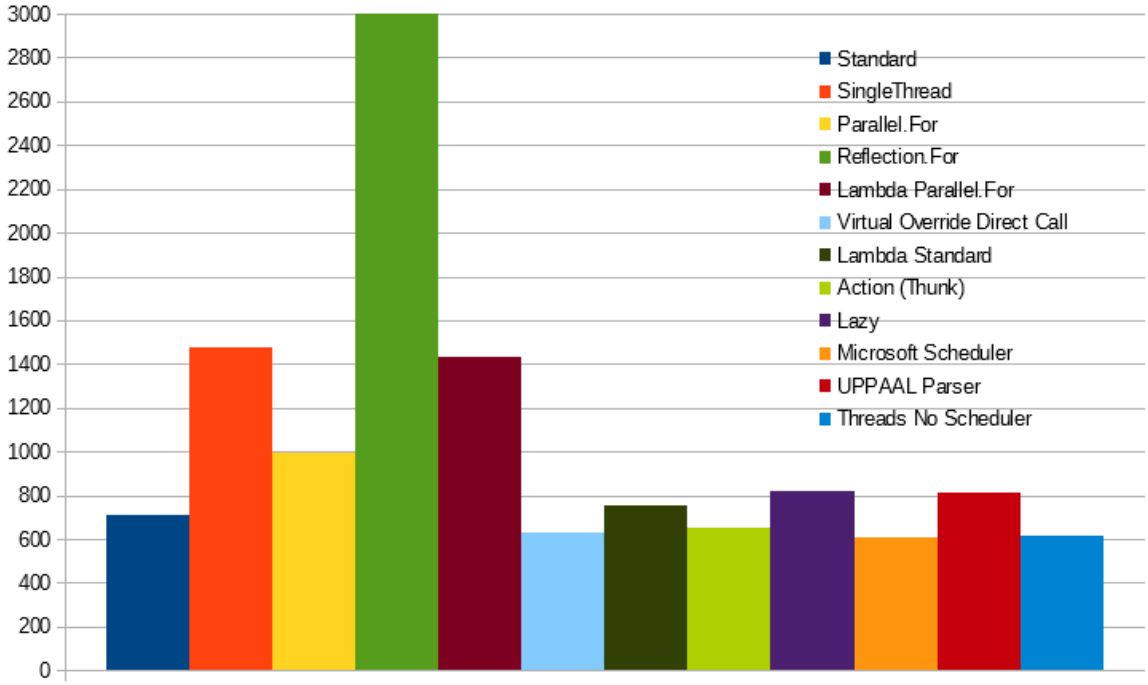

Figure 5.6: Double written to RAM, listing5.1 all the numbers are measured in milliseconds. From table 8.22. Laptop 2 5.1. Lower is better.

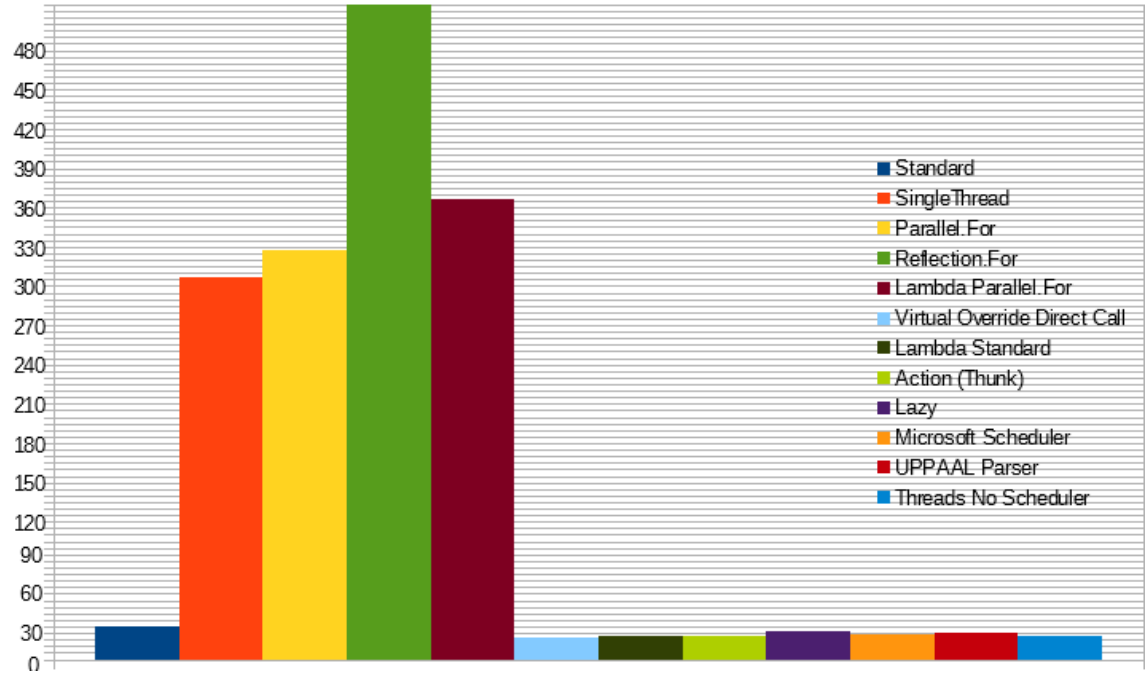

Figure 5.7: Integer not written to RAM, listing5.4 all the numbers are measured in milliseconds. From table 8.23. Laptop 2 5.1. Lower is better.

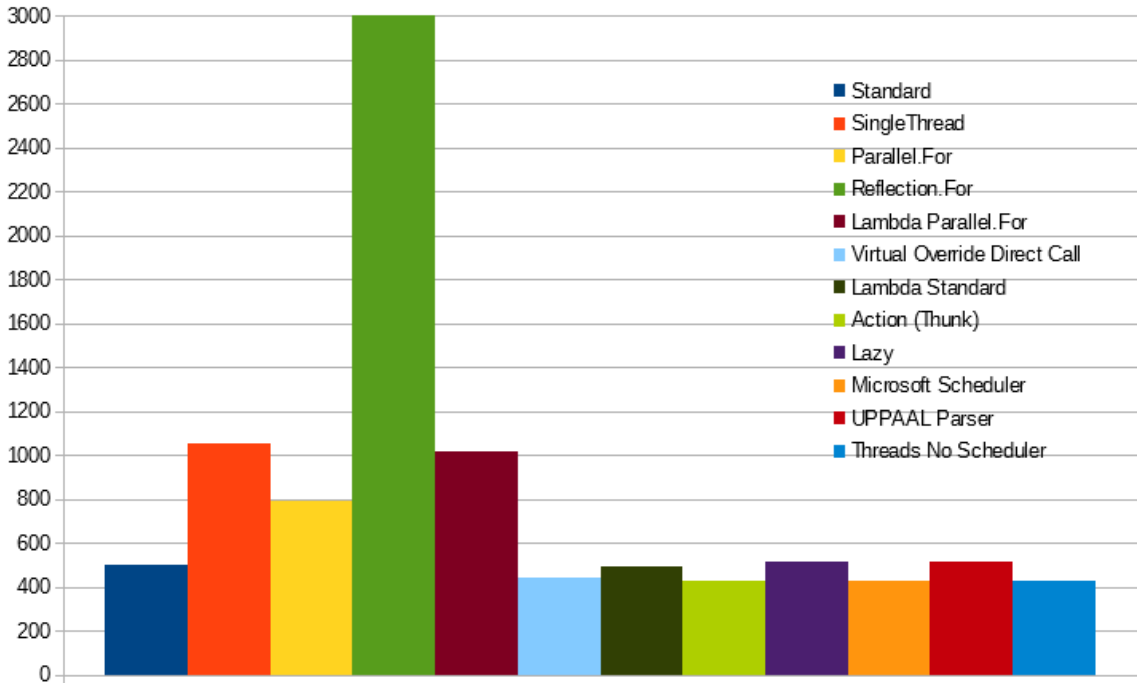

Figure 5.8: Integer written to RAM, listing5.3 all the numbers are measured in milliseconds. From table 8.24. Laptop 2 5.1. Lower is better.

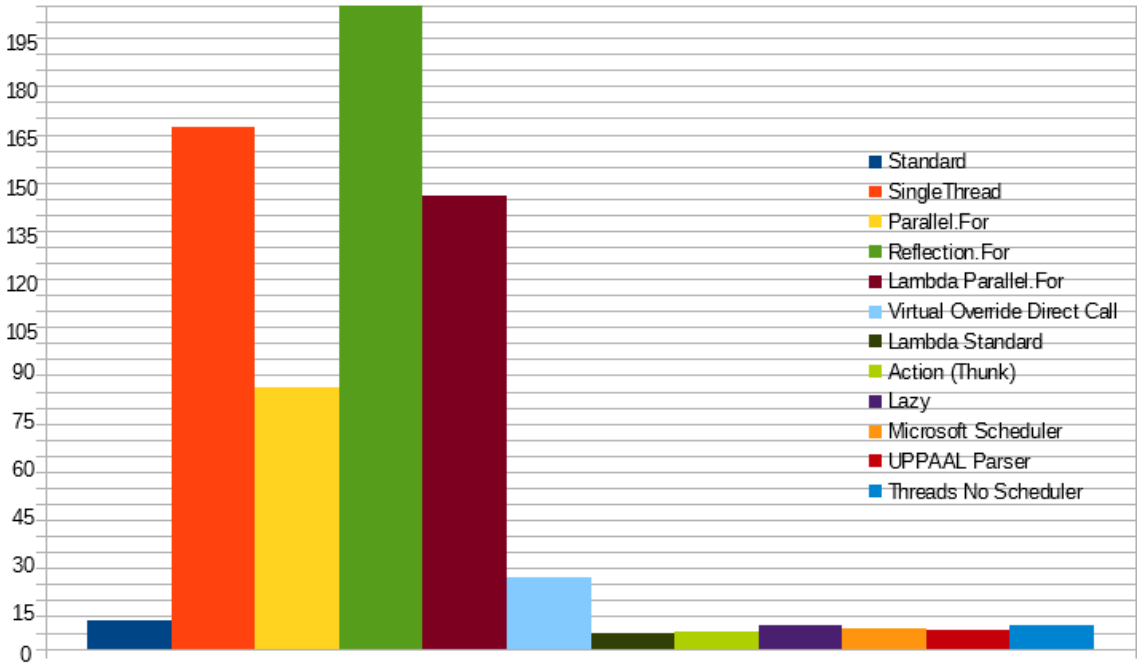

Figure 5.9: Double not written to RAM, listing5.2 all the numbers are measured in milliseconds. From table 8.5. Desktop 5.1. Lower is better.

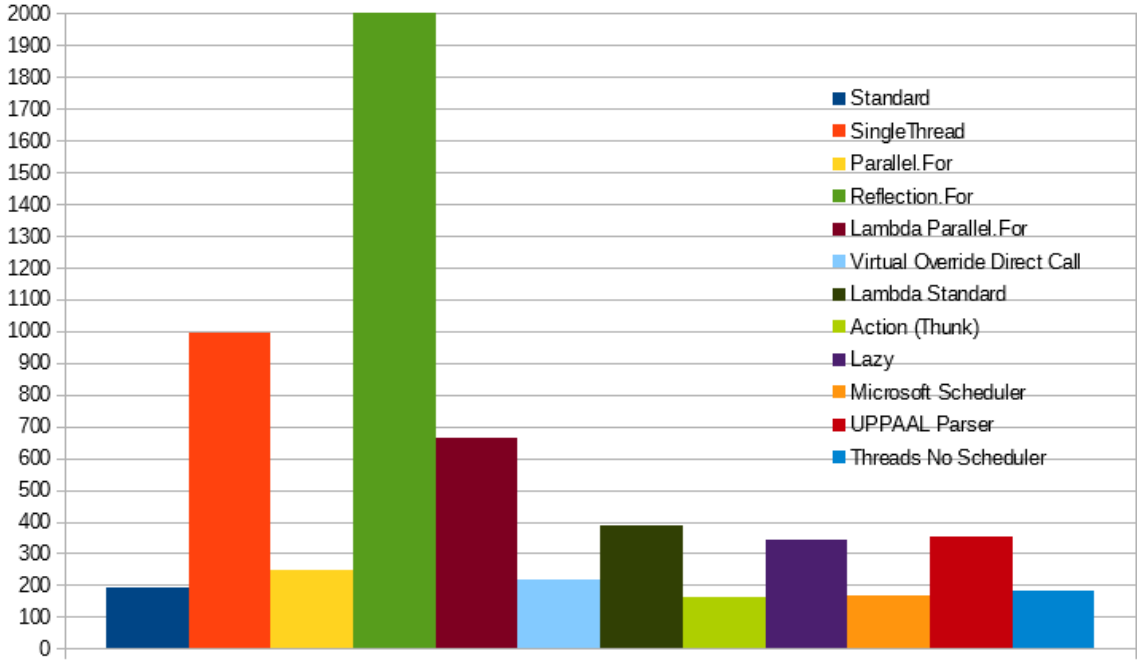

Figure 5.10: Double written to RAM, listing5.1 all the numbers are measured in milliseconds. From table 8.6. Desktop 5.1. Lower is better.

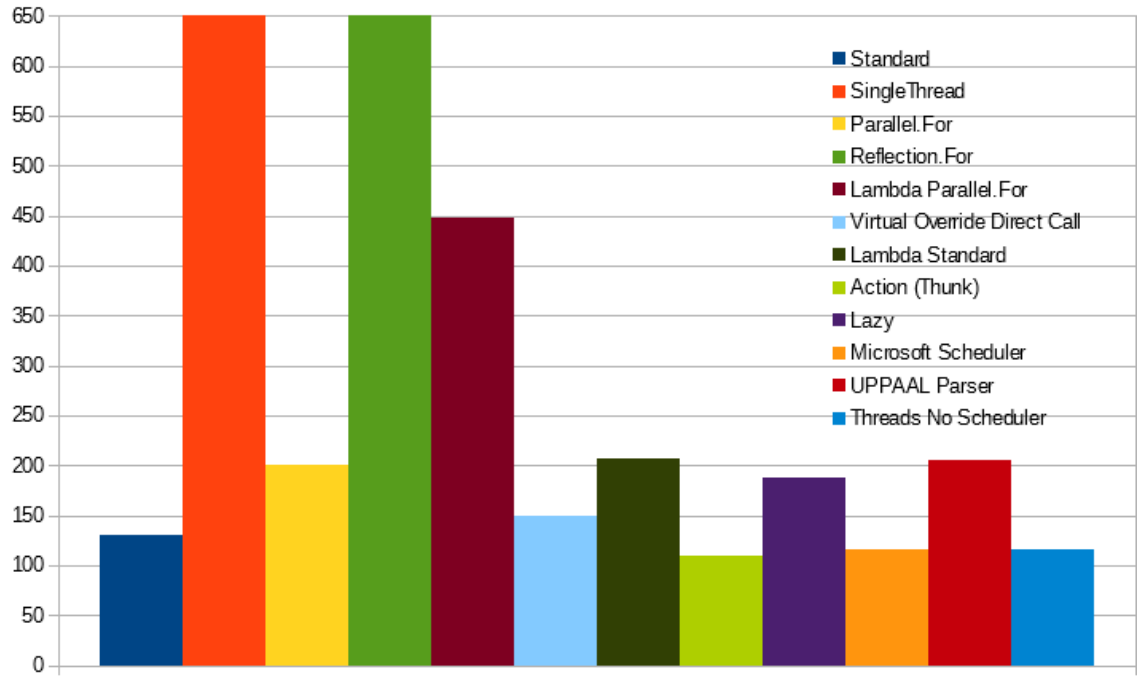

Figure 5.11: Integer not written to RAM, listing5.3 all the numbers are measured in milliseconds. From table 8.8. Desktop 5.1. Lower is better.

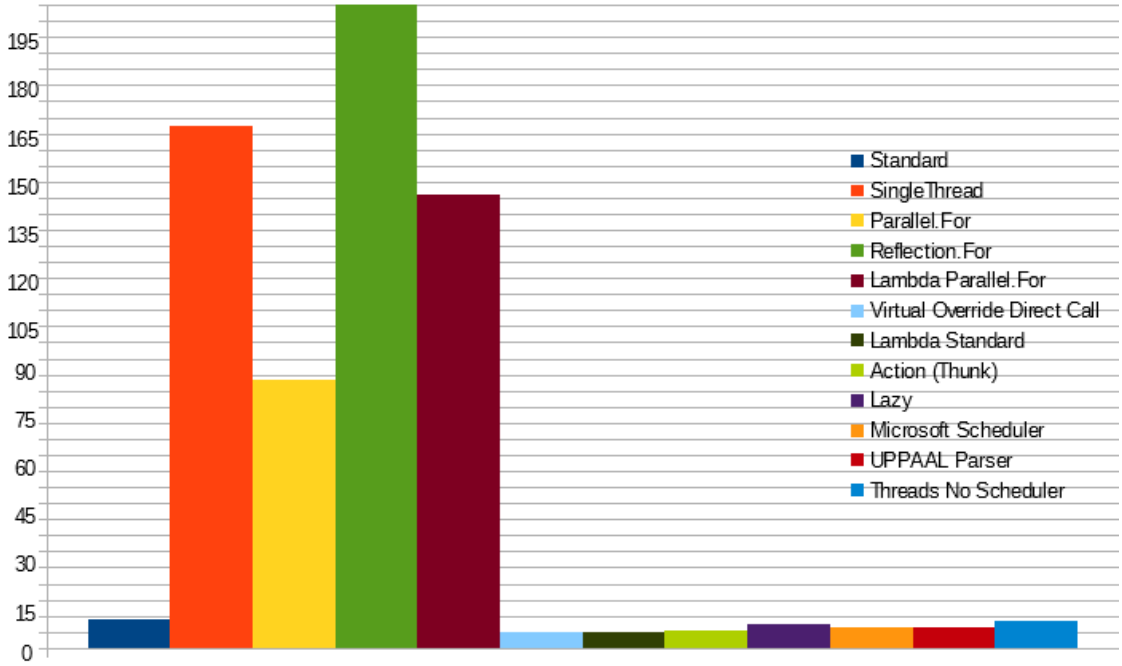

Figure 5.12: Integer written to RAM, listing5.4 all the numbers are measured in milliseconds. From table 8.7. Desktop 5.1. Lower is better.

5.4 Spreadsheet Benchmarks

The benchmark is based on a spreadsheet where tasks have been split up to the amount of cores available. The spreadsheets used in this section is from [12]. The parallel solution will create an analysis of the spreadsheet, which will afterwards be used for the recalculations. All the parallel tests are conducted with the finished analysis, then use the finished analysis in the dependency scheduler. LibreOffice version 5.1.3, has been used. The LibreOffice version 5.1.3 has the acceleration implemented from [2] which determines what piece of hardware is the fastest and then use it on the spreadsheet.

5.4.1 Spreadsheet Building Design

The spreadsheet used in this subsection is *Building Design* converted to XMLSS standard for [25] to be able to read it. The parallel solution requires a lot of RAM which may effect systems with 8GB RAM or less. The spreadsheet is in the area of architecture with constructing buildings with energy conservations as stated in the benchmark [12].

Listing 5.5: This is the spreadsheet test *Building Design* from the original program for Desktop 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 13.463,40 ms
```

Listing 5.6: This is the spreadsheet test *Building Design* from the parallel solution, after the analysis has finished for Desktop 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 2.668,10 ms
```

Listing 5.7: This is the spreadsheet test *Building Design* from the original program for Laptop 1 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 15,369.50 ms
```

Listing 5.8: This is the spreadsheet test *Building Design* from the parallel solution, after the analysis has finished for Laptop 1 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 4.653,00 ms
```

Listing 5.9: This is the spreadsheet test *Building Design* from the original program for Laptop 2 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 20.690,20 ms
```

Listing 5.10: This is the spreadsheet test *Building Design* from the parallel solution, after the analysis has finished for Laptop 2 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 14.604,50 ms
```

5.4.2 Spreadsheet Ground Water Daily

The spreadsheet used in this subsection is *Ground Water Daily* converted to XMLSS standard for [25] to be able to read it. The parallel solution requires a lot of RAM which may effect systems with 12GB RAM or less. The spreadsheet is in the statistical domain. It can be used to analyse ground water data, as stated in the benchmark [12].

Listing 5.11: This is the spreadsheet test *Ground Water Daily* from the original program for Desktop 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 38.327,90 ms
```

Listing 5.12: This is the spreadsheet test *Ground Water Daily* from the parallel solution, after the analysis has finished for Desktop 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 7.142,70 ms
```

Listing 5.13: This is the spreadsheet test *Ground Water Daily* from the original program for Laptop 1 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 45.996,50 ms
```

Listing 5.14: This is the spreadsheet test *Ground Water Daily* from the parallel solution, after the analysis has finished for Laptop 1 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 10.562,30 ms
```

Listing 5.15: This is the spreadsheet test *Ground Water Daily* from the original program for Laptop 2 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 54.208,20 ms
```

Listing 5.16: This is the spreadsheet test *Ground Water Daily* from the parallel solution, after the analysis has finished for Laptop 2 5.1

```
1 === Benchmark workbook called:
2 [Workbook full recalculation] Average of the 10 runs: 34.305,30 ms
```

5.4.3 Combined LibreOffice Results

Listing 5.17: This is the spreadsheet test on both *Building Design* and *Ground Water Daily* from LibreOffice for Desktop 5.1

```
1 Building Design: 68.39 ms, 63.78 ms, 122.40 ms - avg 84.86 ms
2 Ground Water:16181.39 ms, 15845.72 ms, 15852.80 ms - avg 15959,97
      ms
```

Listing 5.18: This is the spreadsheet test on both *Building Design* and *Ground Water Daily* from LibreOffice for Laptop 1 5.1

```
1 Building Design: 214.97 ms, 206.12 ms, 206.77 ms - avg 209.29 ms
```

```
2 Ground Water: 17921.10 ms, 17274.84 ms, 17759.94 ms - avg 17651.96
      ms
```

Listing 5.19: This is the spreadsheet test on both *Building Design* and *Ground Water Daily* from LibreOffice for Laptop 2 5.1

```
1 Building Design: 1338.52 ms, 1156.21 ms, 1178.11 ms - avg 1224,28
      ms
2 Ground Water: 33233.99 ms, 32965.25 ms, 33351.79 ms - avg 33183.68
      ms
```

5.5 Spreadsheet Benchmark Graphs

5.5.1 Spreadsheet Building Design

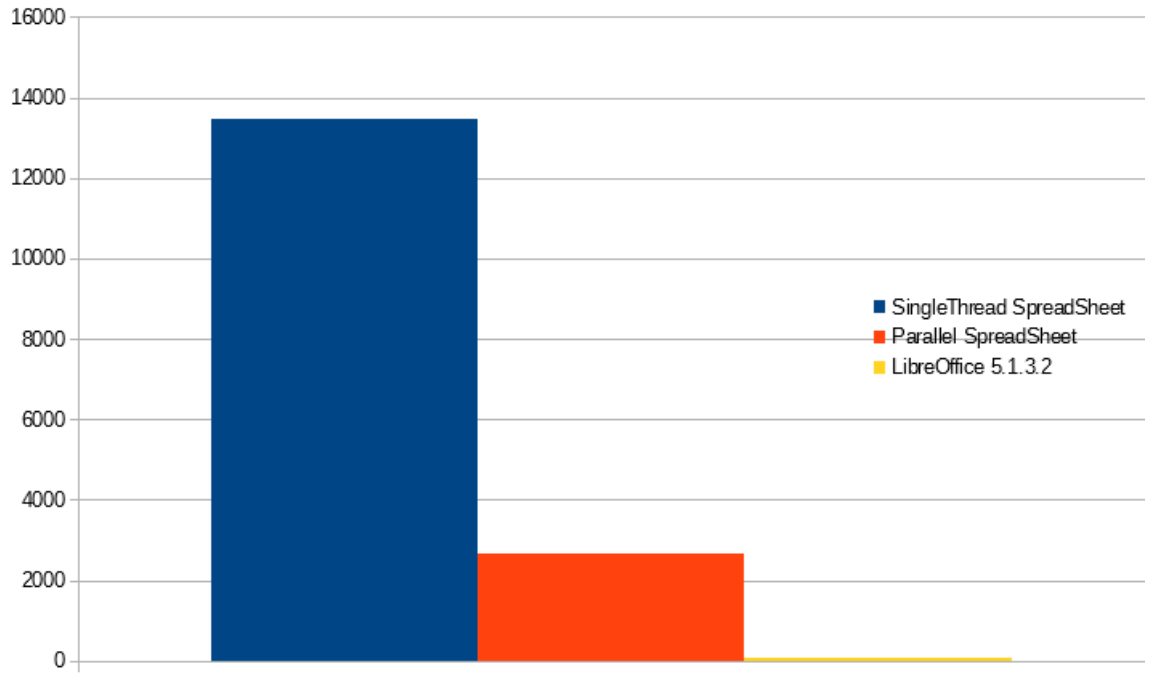

Figure 5.13: Spreadsheet data all the numbers are measured in milliseconds. Data listing 5.5,5.6,5.17, Desktop 5.1. Lower is better.

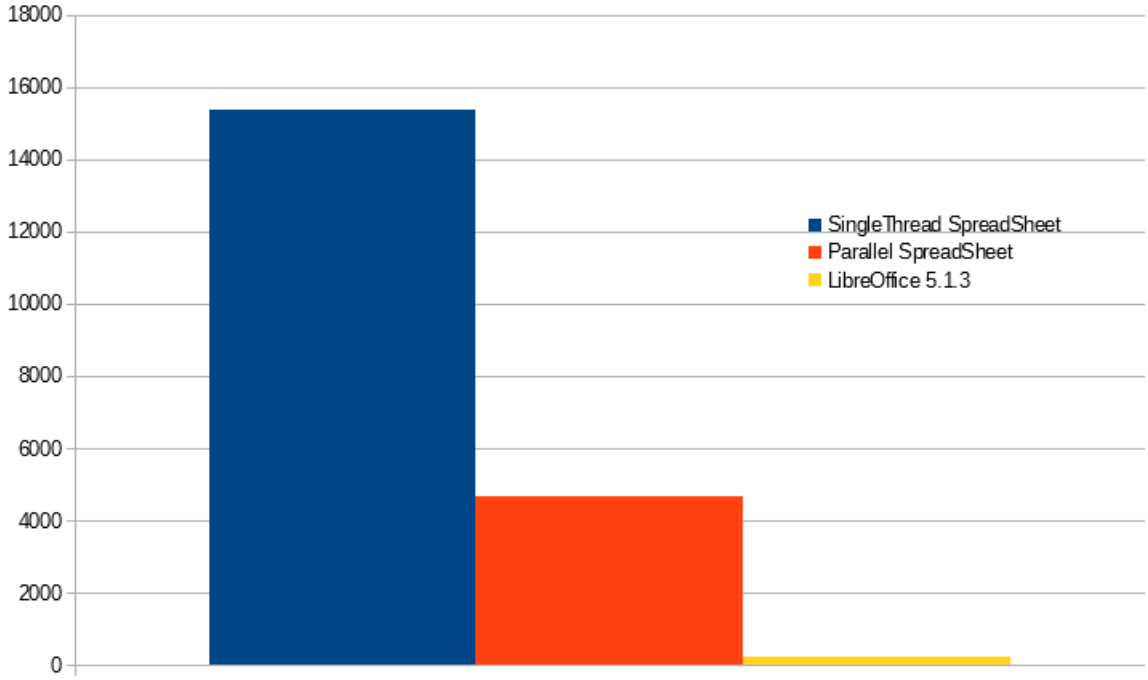

Figure 5.14: Spreadsheet data all the numbers are measured in milliseconds. Data listing 5.7,5.8,5.18, Laptop 1 5.1. Lower is better.

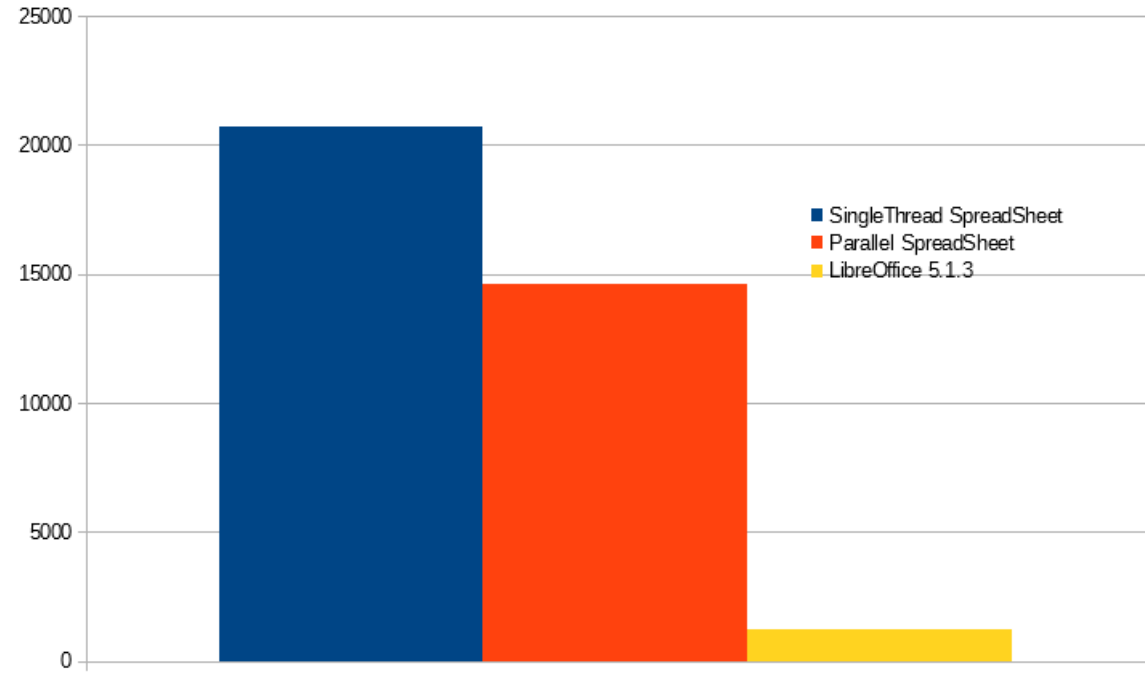

Figure 5.15: Spreadsheet data all the numbers are measured in milliseconds. Data listing 5.9,5.10,5.19, Laptop 2 5.1. Lower is better.

5.5.2 Spreadsheet Ground Water Daily

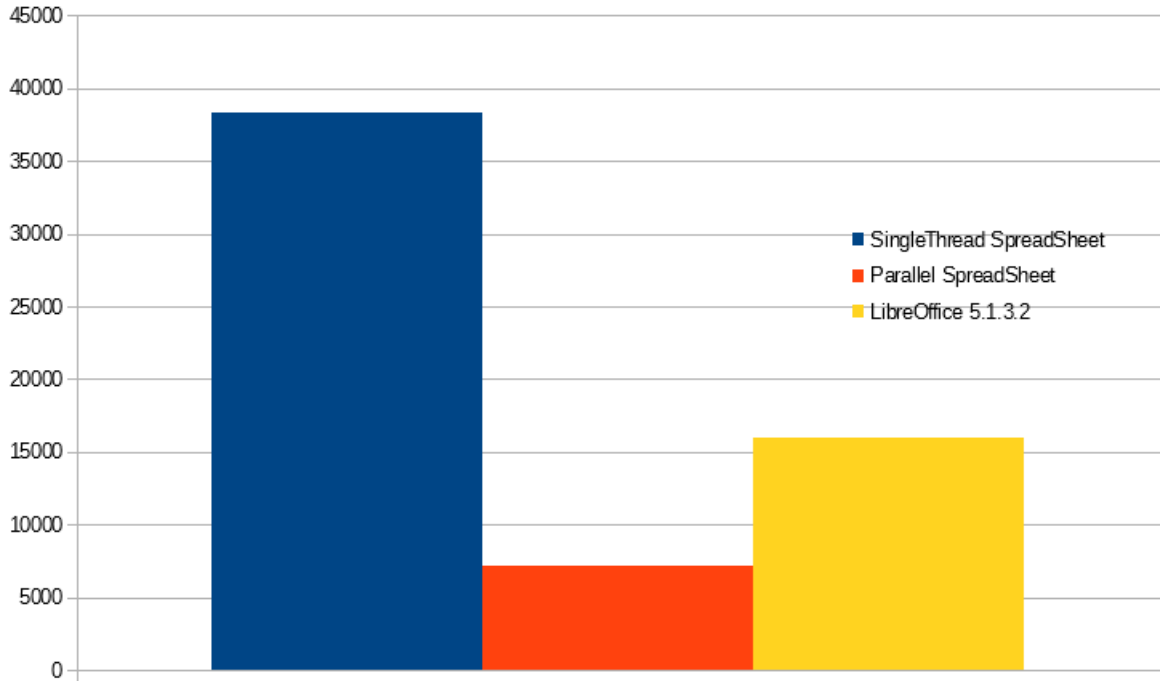

Figure 5.16: Spreadsheet data all the numbers are measured in milliseconds. Data listing 5.11,5.12,5.17, Desktop 5.1. Lower is better.

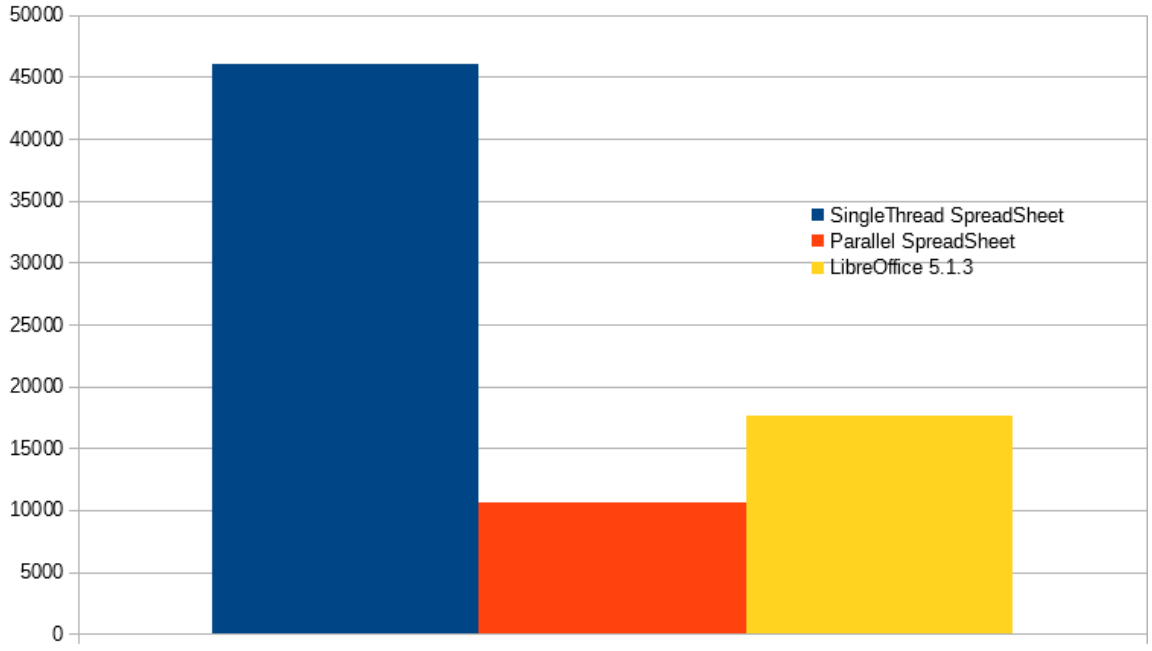

Figure 5.17: Spreadsheet data all the numbers are measured in milliseconds. Data listing 5.13,5.14,5.18, Laptop 1 5.1. Lower is better.

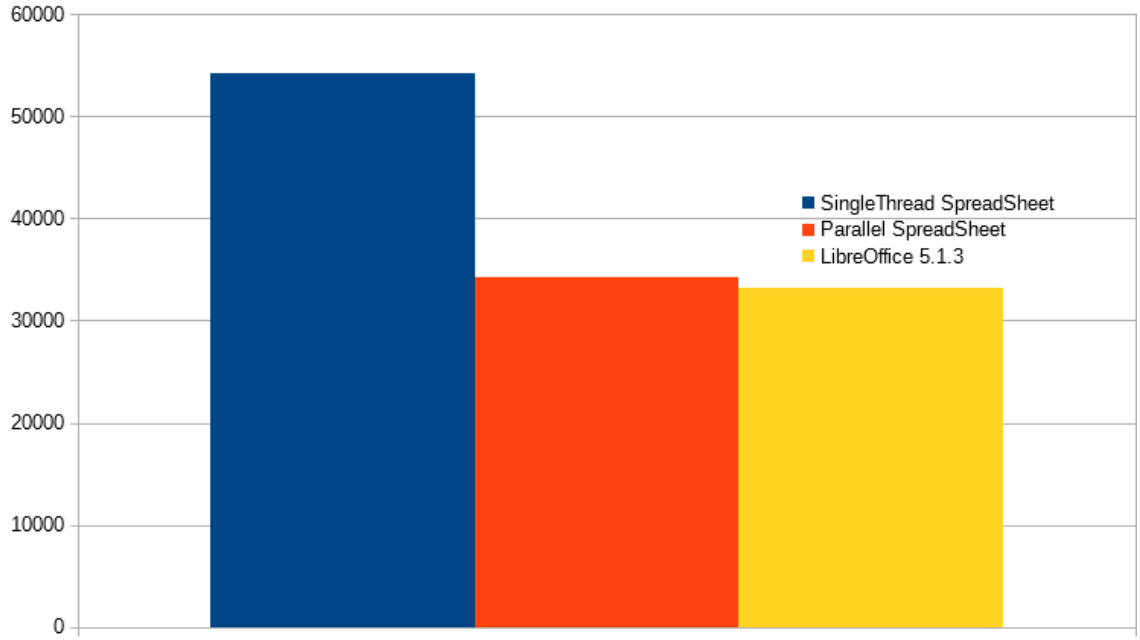

Figure 5.18: Spreadsheet data all the numbers are measured in milliseconds. Data listing 5.15,5.16,5.19, Laptop 2 5.1. Lower is better.

6 Discussion

This chapter will be about the general discussion on the previous sections.

6.1 General Discussion of the Sections

Contains some discussion about each section.

6.1.1 Design

The design of the dependency scheduler can be suited to many needs. The *Dependency Scheduler* is intended as an unchangeable library. The *Dependency Scheduler* could be improved upon with the following points:

- It could be open sourced.
- It could have generic implemented instead of casting.
- Fully implement *Action* with up to 19 parameters.
- A dynamic item class (TaskItem) which could be used by the scheduler and changed by the user. Thus making virtual override faster by removing boxing and unboxing action from the current solution with Object.

The design would be easier to reconstruct after a multitude of tests and figuring out what it might lack. The first issue with the dependency scheduler was the way to handle threads, the decision ended up being the thread pool implemented from Microsoft .NET library, as the other solution was to handle the threads in the scheduler. Handling the threads in the scheduler could be possible as they needs to be assigned to a custom made thread pool which a newly created thread could handle, where each thread had a start and stop for when it could perform a task. The threads handling could be handled in a way where no unnecessary actions should be made. In theory it will have worked the same way as the current scheduler, with a small risk of being less effective on a larger amount of tasks. As well as finding the CPU information would be required to get the dynamic amount of threads, from computer to computer. The next problem was the implementation of waiting on tasks, the *Thread.Sleep* function was first used, it gave some okay results on specific encounters and had to be tweaked for the amount of milliseconds it slept. The Scheduler

ended up with using *WaitOne* as it is dynamic and works best in each case. Another problem with waiting in the main thread was to make the main thread do nothing, to not consume computational power, thus a wait handle was found instead of the previous sleep and a check statement, which is handled by the scheduler. The scheduler itself needed a way to not consume more computational power than required. Which means that after it has traversed all tasks and all jobs have been started, it will go to sleep like the main thread, it will then be awoken by a task thread or the main thread with a new task before it starts all over again. Unfortunately the dependency scheduler have some overhead which makes a lot of small tasks inefficient, but the overhead is required to properly handle dependency. There was an alternative idea for dependency system by using string names instead of integers. The problem with this approach is it would have resulted in a very large overhead. The current system uses the list as an access list without traversing the list. One of the solutions for a lot of parallel data is a *parallel.for* solution with *Reflection* or with *Action*, which decrease number of tasks and increases task size, the most optimal thread size found was using the number of logical threads available from the CPU, which it utilizes automatically in the *Parallel.for Reflection* solution.

6.1.2 Implementation

As mentioned earlier *generic* could be implemented, but this would require a lot of extra programming to make it *generic*. The implementation seems to be sufficient at the given time, with only small room for improvements for the current solution. The spreadsheet implementation could be implemented in the *Workbook*, but it was chosen to be in the *Sheet* as it had direct access to the cells, figure 4.2. The analysis is possible with less RAM, but as the projects main focus was not how to create efficient analyses it was deemed acceptable. Another speculation could be to use some additional ram to store every split up to the scheduler, eliminating the small single threaded part left after the pre-analysis, making the program speed up a little extra.

6.1.3 Benchmarks

Micro Benchmarks

The benchmark section shows interesting results with the CPU. The micro benchmarks are mostly used to determine if a process way is faster or slower than the other processes. The Desktop 5.1, where the micro benchmark solution has different victors for figure 5.9 *Lambda Standard* which is 0.1 ms faster than the Action. However, the *Lambda Standard* is an Action<object,object>. The fastest solution *Lambda Standard* is 32.4 times faster than

SingleThread, when doubles are not written to RAM. For figure 5.10 the fastest is *Action (Thunk)* where *Microsoft Scheduler* is 7.3 ms behind. The fastest solution *Action (Thunk)* is 6.17 times faster than *SingleThread*, while writing doubles to RAM. For figure 5.12 the fastest is *Lambda Standard*, with *Virtual Override* 0.1 ms behind. The fastest solution *Lambda Standard* is 32.44 times faster than *SingleThread*, when integers are not written to RAM. For figure 5.11 *Action (Thunk)* is the fastest with *Threads No Scheduler* 12.8 ms behind. The fastest solution *Action (Thunk)* is 7.12 times faster than *SingleThread*, while writing integers to RAM.

For the Laptop 1 5.1 the micro benchmarks fluctuates with different results as the fastest. For figure 5.1 *Lambda Standard* is the fastest and behind it *Action(Thunk)* 2.4 ms behind. The fastest solution *Lambda Standard* is 22.37 times faster than *SingleThread*, when doubles are not written to RAM. For figure 5.2 *Threads No Scheduler* is the fastest with *Microsoft Scheduler* behind it with 2 ms. The fastest solution *Threads No Scheduler* is 3.91 times faster than SingleThread, while writing doubles to RAM. For figure 5.3 *Lambda Standard* is the fastest with *Virtual Override* just behind with 0.2 ms. The fastest solution *Lambda Standard* is 20.18 times faster than *SingleThread*, when integers are not written to RAM. For figure 5.4 *Action(Thunk)* is the fastest and behind it *Threads No Scheduler* 0.2 ms later. The fastest solution *Action Thunk* is 4.11 times faster than *SingleThread*, while writing integers to RAM.

For the Laptop 2 5.1 the micro benchmarks yet again fluctuates with different results as the fastest. For figure 5.5 *Lambda standard* and *Action (Thunk)* have the same time, with *Threads No Scheduler* 0.2 ms behind. The fastest solution *Lambda Standard* and *Action (Thunk)* is 17.23 times faster than *SingleThread*, when doubles are not written to RAM. For figure 5.6 *Microsoft Scheduler* is the fastest, with *Threads No Scheduler* 4.3 ms behind. The fastest solution *Microsoft Scheduler* is 2.41 times faster than *SingleThread*, while writing doubles to RAM. For figure 5.7 *Virtual Override* is the fastest with *Lambda Standard* behind 0.1 ms. The fastest solution *Virtual Override* is 17.18 times faster than *SingleThread*, when integers are not written to RAM. For figure 5.8 *Threads No Scheduler* is the fastest with *Microsoft Scheduler* right behind with 3.7 ms. The fastest solution *Threads No Scheduler* is 2.48 times faster than *SingleThread*, while writing integers to RAM.

The conclusion on the micro benchmarks is that, the scheduler is highly efficient in most cases. These results indicate the most efficient solutions are Action (Thunk), Lambda Standard and Virtual Override. The implementation of all the different methods might affect the speed of the scheduler as *TaskItem* contains more information, as well as more checks to figure out which functions are currently used.

Real Application Benchmark

The spreadsheet used is from [12] where the spreadsheet used is *Building Design* converted to XMLSS standard for [25] to be able to read it. The benchmarks figure: 5.13,5.14,5.15 gives good results. The parallel solution on figure 5.13 is 5.05 times faster than the original solution. However, LibreOffice was much faster, it was 158.65 times faster than the original solution. The parallel solution on figure 5.14 is 3.30 times faster than the original solution. Once again LibreOffice was faster, it was 73.44 times faster than the original solution. The parallel solution on figure 5.15 is 1.42 times faster than the original solution. LibreOffice solution is still better on the *Building Design* spreadsheet with 16.90 times faster than the original solution. These results are blazingly fast both for LibreOffice and for the dependency scheduler, even when the dependency scheduler does not use APU or GPU and as the spreadsheet contains 937303 cells of which 39519 are unique. Even with a simple analysis of dependency the results has improved fivefold on a hexa-core processor, three and a half fold on a quad core processor and one and a half fold on a duo core processor. Based on the micro benchmarks if there were no dependency the highest possible results would be 6.17 times faster on a hexa-core processor instead of 5.05 times faster from figure 5.13, 3.91 times faster on a quad cores processor instead of 3.30 times faster from figure 5.14 and 2.41 times faster on a duo core processor instead of 1.42 times faster from figure 5.15. These results are quite impressive as the Spreadsheet[25] already was very fast in the first place.

The other spreadsheet used is *Ground Water Daily* converted to XMLSS standard. For the figure 5.16 the parallel solution is 5.37 times faster than the original solution. The parallel solution is 2.23 times faster than LibreOffice solution. For the figure 5.17 the parallel solution is 4.35 times faster than the original solution. The parallel solution is 1.67 times faster than LibreOffice solution. For the figure 5.18 the parallel solution is 1.58 times faster than the original solution. The LibreOffice solution is 1.03 times faster than the parallel solution. The results gathered shows that the CPU's flexibility in its concurrency can give a large boost in performance, even when the GPU has more computational power than the CPU. The comparison gives an interesting idea that the CPU's strength is flexibility whereas the GPU's strength lies in brute force calculation. Nevertheless the pure CPU performance is quite excellent even in comparison to the GPU's throughput.

6.1.4 Alternative thought process to Dependency Scheduler

Without the abstraction the dependency scheduler provide, it would have been difficult to create a concurrent solution. As the first thing that could have been done was creating mon-

itors or semaphores around the cells calculation. However, for this to work properly either the thread should start at the last points of the support graph, where a thread has to be created for every end to minimize overhead. Or another possibility would be to create a thread for every cell after the monitors or semaphores has been implemented, this would give a lot of overhead and a lot of threads waiting for a result from a previous cell. Another problem that could arise was if a part of the formula could be calculated, but the thread started is waiting for a result from another thread. The thread waiting for a result in another thread might happen even if the support graph is used to find the end points. To sum up, when using the abstraction threads with the ideas of locks, monitors, semaphores, it becomes difficult to understand the abstraction, as well as how to minimize overhead, or use correct amount of threads.

6.2 Pros and Cons

The dependency scheduler solves various problems if used correctly. It can solve condition racing, deadlocks, livelocks, but if used in a brute force way it will continue to have the same multitasking problems as usual. It uses C# which can observe other threads than the main thread making it easier to debug than on Fortran, C or C++. The dependency scheduler takes care of everything about thread creation, thread handling, and interleaving. The dependency scheduler can with the use of multiple threads grant a little better use of the cache for single core programs. The dependency scheduler increases readability but this has not been tested among programmers, there cannot be drawn a definite conclusion. The user created parsers should diversify what the dependency scheduler should be able to do as the dependency scheduler should be as minimalistic as possible. The dependency scheduler lack MPI which makes it unable to work on clusters with multiple processors. It still deadlocks if a task have dependency to itself. If it should deadlock it will not consume processing power while this is the case, only for the tasks that can be run without the deadlock. If the dependency scheduler gets too many small tasks, the scheduler will end up creating an overhead. The dependency scheduler gives another abstraction to threads, which can help solve some programming problems. The dependency scheduler is easier to debug as it can use visual studio or mono, which in debug enables the programmer to observe whatever thread the programmer wishes.

6.3 Remarks

All in all the implementation seems highly efficient, as it is comparable with Microsoft Scheduler. The interesting thing about the scheduler is the dependency which makes it

possible to give another abstract than making a lot of waits, locks, creating new threads, interleaving of threads.

7 Conclusion and Future Work

7.1 Conclusion

The report solved the problem with using multiple threads. The abstraction of threads can be difficult, but the abstraction was changed to tasks with a dependency. With the dependency scheduler it changed the focus from multitasking problems to dependency between tasks. If the different abstractions the dependency scheduler gives, is understood and mastered then a lot of the general multitasking problems can be solved. The results from the micro benchmark suggest that the dependency scheduler is comparable to the existing solution Microsoft Scheduler, or regular thread spawning, which does not take dependency into consideration. The results is from various micro benchmarks using integers and double calculation, locally on chip and RAM and cache. The micro benchmark gives an idea on which implementations is the fastest on the dependency scheduler as well as how does it compare to the existing solutions in performance. The results contains tests on two spreadsheets with a parallel implementation with a simple analysis on run time, which gave positive results for all tested processors. The results were then compared to LibreOffice newer version which includes OpenCL implementation to calculate the cells on the spreadsheet. In the first benchmarks *Building Design* LibreOffice is a lot faster. However, in the *Ground Water Daily* spreadsheet the parallel solution with only the processor, is faster. The spreadsheet tests might indicate that the processor has been underestimated in its power to do concurrent tasks, whereas more calculative power from the APU or GPU is best used if many of the same actions are used repeatedly. Thus it can be concluded that the dependency scheduler has a favourable performance and gives a much needed abstraction from threads as well as the regular multitasking problems. The report uses C# for its solution thus making debugging easier than on Fortran, C and C++ which is regularly used for parallel solutions. The results gotten from the dependency scheduler in the report is blazingly fast as it almost get the most performance out of the processor in a spreadsheet. The spreadsheet was used as the real application benchmark, which gave impressive results with a hexa-core processor speeding up a spreadsheet fivefold.

7.2 Future Works

One of these directions could be to make it use C# *Generic*, which would make the virtual override better. *Action* could be fully implemented with up to 19 different constants. It could become a template for a possible way to handle multiple threads. It could be stepping stone for it to become a scheduler behind the scenes that should be implemented in a programming language and be solved at compile time, so no further time should be used on runtime to split up the tasks, granting a better concurrent language. The report might inspire other abstraction for threads, which might be easier to perceive than threads. The dependency scheduler could be used in critical systems where performance is important, as well as the program may not end up in a deadlock. As the dependency scheduler is quite dynamic it could be used in regular application, even if the applications have low concurrency, as the dependency scheduler might speed up small parts of the program where concurrency is present. Further studies to implement a OpenMP and a MPI system into the dependency scheduler to handle distributed systems. Use the dependency scheduler as an inspiration to create a reactive system which handles tasks differently.

Bibliography

[1] Jade Alglave, Mark Batty, Alastair F Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. Gpu concurrency: weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 577–591. ACM, 2015.

[2] AMD. Collaboration and open source at amd: Libreoffice. `http://developer.amd.com/community/blog/2015/07/15/collaboration-and-open-source-at-amd-libreoffice/`.

[3] OpenMP Architecture Review Board. The openmpő api specification for parallel programming. `http://openmp.org/wp/`.

[4] LibreOffice Community. Libreoffice. `https://www.libreoffice.org/download/libreoffice-fresh/`.

[5] Holk et al. harlan. https://github.com/eholk/harlan. Last checked 14-12-2015.

[6] Message Passing Interface Forum. Mpi documents. `http://www.mpi-forum.org/docs/`.

[7] Semyon Grigorev. Brahma.fsharp. https://sites.google.com/site/semathsrprojects/home/brahma-fsharp.

[8] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, (7):33–38, 2008.

[9] PARALUTION Labs. Paralution. `http://www.paralution.com/`.

[10] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[11] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[12] LibreOffice. Benchmark. `https://gerrit.libreoffice.org/gitweb?p=benchmark.git;a=tree`.

[13] Mark Michaelis. *Essential C# 4.0 (3rd Edition) (Microsoft Windows Development Series)*. Addison-Wesley Professional, 2010.

[14] Microsoft. Action<t> delegate. https://msdn.microsoft.com/en-us/library/018hxwa8(v=vs.110).aspx.

[15] Microsoft. Autoresetevent class. https://msdn.microsoft.com/en-us/library/system.threading.autoresetevent(v=vs.110).aspx.

[16] Microsoft. C++ amp (c++ accelerated massive parallelism). https://msdn.microsoft.com/da-dk/library/hh265137.aspx. Last Checked 14-12-2015.

[17] Microsoft. Generics (c# programming guide). https://msdn.microsoft.com/en-us/library/512aeb7t.aspx.

[18] Microsoft. Lazy<t> class. https://msdn.microsoft.com/en-us/library/dd642331(v=vs.110).aspx.

[19] Microsoft. .net framework. https://www.microsoft.com/net/default.aspx.

[20] Microsoft. Parallel.for method. https://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.for(v=vs.110).aspx.

[21] Microsoft. Reflection (c# and visual basic). https://msdn.microsoft.com/en-us/library/ms173183.aspx.

[22] Microsoft. Taskscheduler class. https://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler(v=vs.110).aspx.

[23] Microsoft. Threadpool class. https://msdn.microsoft.com/en-us/library/system.threading.threadpool(v=vs.110).aspx?

[24] David Padua. *Encyclopedia of parallel computing*, volume 4. Springer Science & Business Media, 2011.

[25] Peter Sestoft. Corecalc and funcalc spreadsheet technology in c#. http://www.itu.dk/people/sestoft/funcalc/.

[26] Peter Sestoft. Microbenchmarks in java and c#. *Electronic Proceedings in Theoretical Computer Science*, 2013.

[27] Peter Sestoft. *Spreadsheet Implementation Technology: Basics and Extensions (MIT Press)*. The MIT Press, 2014.

[28] David Wheeler. Secure programmer: Prevent race conditions. *IBM. WWW document {cited 18 March, 2007 from http://www-128.ibm. com/developerworks/linux/library/l-sprace. html }*, 2008.

[29] referenced by `http://openmp.org/` Wikipedia. Openmp. `https://en.wikipedia.org/wiki/OpenMP`.

8 Appendix

8.1 Tables

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	11	10	8	8	9	8	9	9	8	9
Single	162	162	162	162	162	162	162	162	162	162
Parallel.For	80	80	88	82	82	80	81	80	81	80
Reflection.For	5229	5229	5219	5247	5228	5209	5190	5222	4589	5198
Lambda.For	146	144	144	144	145	145	145	144	107	147
VirtualOverride	25	26	18	25	23	26	22	25	14	18
Lambda Standard	5	5	5	5	5	5	5	5	5	5
Action (Thunk)	5	5	5	5	5	6	5	5	5	5
Lazy	7	9	7	7	8	7	8	7	7	7
Microsoft Scheduler	6	6	6	6	7	6	6	6	6	6
UPPAAL Parser	6	6	6	6	6	6	6	6	6	6
Threads No Scheduler	11	10	9	6	7	6	6	5	6	7

Table 8.1: All the results are measured in milliseconds. This test is double calculation without writing it to the RAM. Desktop 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	181	185	213	188	184	178	184	206	187	192
Single	997	993	994	994	994	995	998	996	995	994
Parallel.For	243	251	251	254	238	237	243	270	258	235
Reflection.For	5523	5452	5503	5456	5440	5436	5475	5284	5465	5486
Lambda.For	674	682	796	667	652	659	643	520	664	698
VirtualOverride	239	236	213	232	210	207	214	172	224	227
Lambda Standard	390	395	357	392	388	387	396	385	388	381
Action (Thunk)	159	158	164	157	157	175	157	170	157	158
Lazy	370	335	354	332	346	335	331	354	333	332
Microsoft Scheduler	159	161	159	185	160	160	160	160	196	185
UPPAAL Parser	337	356	377	348	367	355	355	360	337	334
Threads No Scheduler	197	180	157	214	157	201	199	169	153	199

Table 8.2: All the results are measured in miliseconds. This test is double calculation with writing it to the RAM. Desktop 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	9	9	8	11	9	9	8	8	8	8
Single	162	162	162	164	162	162	162	162	162	162
Parallel.For	91	82	86	81	80	80	81	81	91	81
Reflection.For	5192	5084	4785	5175	5191	5216	5193	5191	5262	5237
Lambda.For	146	146	107	144	144	145	147	144	143	145
VirtualOverride	5	5	5	6	5	5	5	5	5	5
Lambda Standard	5	5	5	5	5	5	5	5	5	5
Action (Thunk)	5	5	5	6	5	5	5	5	5	6
Lazy	9	7	7	7	7	8	7	7	7	7
Microsoft Scheduler	6	6	9	6	6	6	6	7	6	6
UPPAAL Parser	6	6	6	6	6	6	6	7	7	6
Threads No Scheduler	11	12	5	8	8	12	6	9	5	5

Table 8.3: All the results are measured in milliseconds. This test is int calculation without writing it to the RAM. Desktop 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	122	139	124	131	128	118	142	140	133	121
Single	789	784	784	774	783	783	782	777	784	783
Parallel.For	208	200	189	204	201	194	196	202	206	199
Reflection.For	5371	5354	4380	5393	5308	5333	5276	5288	4973	5275
Lambda.For	480	474	356	455	435	472	474	462	393	481
VirtualOverride	169	134	114	164	160	162	158	160	113	163
Lambda Standard	220	229	197	214	203	128	203	234	199	236
Action (Thunk)	128	102	130	102	119	101	106	104	107	99
Lazy	199	191	191	214	190	103	191	208	192	192
Microsoft Scheduler	148	103	104	103	148	103	138	102	104	103
UPPAAL Parser	214	228	213	196	209	185	208	193	215	189
Threads No Scheduler	103	145	102	140	101	103	104	102	103	149

Table 8.4: All the results are measured in miliseconds. This test is int calculation with writing it to the RAM. Desktop 5.1.

Standard	8,9 avg ms
SingleThread	162 avg ms
Parallel.For	81,4 avg ms
Reflection.For	5156 avg ms
Lambda.For	141,1 avg ms
VirtualOverride	22,2 avg ms
Lambda Standard	5 avg ms
Action (Thunk)	5,1 avg ms
Lazy Evaluation	7,4 avg ms
Microsoft Scheduler	6,1 avg ms
UPPAAL Parser	6 avg ms
Threads No Scheduler	7,3 avg ms

Table 8.5: The average of the tests are measured in milliseconds. This is for double without writing it to the RAM. Desktop 5.1.

Standard	189,8 avg ms
SingleThread	995 avg ms
Parallel.For	248 avg ms
Reflection.For	5452 avg ms
Lambda.For	665,5 avg ms
VirtualOverride	217,4 avg ms
Lambda Standard	385,9 avg ms
Action (Thunk)	161,2 avg ms
Lazy Evaluation	342,2 avg ms
Microsoft Scheduler	168,5 avg ms
UPPAAL Parser	352,6 avg ms
Threads No Scheduler	182,6 avg ms

Table 8.6: The average of the tests are measured in milliseconds. This is for double with writing it to the RAM. Desktop 5.1.

Standard	8,7 avg ms
SingleThread	162,2 avg ms
Parallel.For	83,4 avg ms
Reflection.For	5152,6 avg ms
Lambda.For	141,1 avg ms
VirtualOverride	5,1 avg ms
Lambda Standard	5 avg ms
Action (Thunk)	5,2 avg ms
Lazy Evaluation	7,3 avg ms
Microsoft Scheduler	6,4 avg ms
UPPAAL Parser	6,2 avg ms
Threads No Scheduler	8,1 avg ms

Table 8.7: The average of the tests are measured in milliseconds. This is for int without writing it to the RAM. Desktop 5.1.

Standard	129,8 avg ms
SingleThread	782,3 avg ms
Parallel.For	199,9 avg ms
Reflection.For	5195,1 avg ms
Lambda.For	448,2 avg ms
VirtualOverride	149,7 avg ms
Lambda Standard	206,3 avg ms
Action (Thunk)	109,8 avg ms
Lazy Evaluation	187,1 avg ms
Microsoft Scheduler	115,6 avg ms
UPPAAL Parser	205 avg ms
Threads No Scheduler	115,2 avg ms

Table 8.8: The average of the tests are measured in milliseconds. This is for int with writing it to the RAM. Desktop 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	13	14	14	14	16	14	16	15	18	16
Single	183	188	189	186	181	188	183	186	189	184
Parallel.For	144	146	155	144	147	144	143	144	147	144
Reflection.For	6625	8025	6436	8018	7765	6425	7856	7986	7977	7911
Lambda.For	191	261	192	255	194	194	252	258	259	255
VirtualOverride	26	35	26	33	26	26	36	35	36	35
Lambda Standard	8	8	8	9	8	8	8	8	9	9
Action (Thunk)	8	11	11	11	11	11	11	11	11	11
Lazy	13	13	12	13	14	14	11	11	13	14
Microsoft Scheduler	10	14	10	10	10	10	12	12	12	12
UPPAAL Parser	11	12	10	12	12	12	12	12	12	10
Threads No Scheduler	11	11	10	8	20	11	9	11	10	17

Table 8.9: All the results are measured in milliseconds. This test is double calculation without writing it to the RAM. Laptop 1 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	248	240	250	237	240	246	250	233	250	232
Single	947	954	926	932	931	928	916	950	951	957
Parallel.For	390	372	367	367	371	364	368	377	388	373
Reflection.For	7345	8518	7436	8552	8445	7388	8367	7343	8404	8545
Lambda.For	679	840	695	849	809	728	918	711	824	866
VirtualOverride	230	289	230	281	294	231	298	227	323	300
Lambda Standard	358	406	384	378	367	355	376	387	352	353
Action (Thunk)	236	302	234	246	242	247	244	229	242	245
Lazy	411	428	448	402	460	386	405	410	402	421
Microsoft Scheduler	232	234	230	236	262	238	257	238	262	234
UPPAAL Parser	409	416	428	419	430	405	425	415	424	422
Threads No Scheduler	248	253	253	234	241	233	248	236	228	229

Table 8.10: All the results are measured in miliseconds. This test is double calculation with writing it to the RAM. Laptop 1 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	13	16	16	15	14	16	14	17	15	13
Single	182	181	189	186	183	183	182	184	184	182
Parallel.For	154	146	144	155	143	145	145	149	157	144
Reflection.For	7977	7687	6432	7742	7878	7986	6596	7988	7946	6789
Lambda.For	250	186	191	256	254	251	193	255	253	195
VirtualOverride	12	9	9	9	9	9	9	9	9	9
Lambda Standard	8	16	8	9	9	8	8	8	8	9
Action (Thunk)	11	11	11	11	11	11	11	10	11	11
Lazy	14	11	13	13	14	14	13	14	13	13
Microsoft Scheduler	30	12	10	12	9	10	12	9	12	12
UPPAAL Parser	12	19	13	12	12	13	12	12	12	12
Threads No Scheduler	11	11	11	11	11	11	9	10	10	11

Table 8.11: All the results are measured in milliseconds. This test is int calculation without writing it to the RAM. Laptop 1 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	215	219	198	220	209	204	218	225	214	203
Single	866	834	842	822	809	832	790	810	835	797
Parallel.For	325	336	362	342	338	327	315	337	338	327
Reflection.For	6771	6857	7916	7616	8133	8265	6981	7359	7891	8203
Lambda.For	529	543	628	662	638	633	504	661	536	664
VirtualOverride	185	183	229	249	240	214	181	180	182	224
Lambda Standard	262	254	254	168	243	242	244	278	271	244
Action (Thunk)	188	191	200	216	224	215	202	186	196	185
Lazy	283	274	291	217	293	295	301	289	280	302
Microsoft Scheduler	195	210	187	214	208	208	185	208	209	209
UPPAAL Parser	275	308	305	291	275	303	278	275	303	275
Threads No Scheduler	181	207	200	207	217	216	181	198	217	181

Table 8.12: All the results are measured in miliseconds. This test is int calculation with writing it to the RAM. Desktop Laptop 1 5.1.

Standard	15 avg ms
SingleThread	185,7 avg ms
Parallel.For	145,8 avg ms
Reflection.For	7502,4 avg ms
Lambda.For	231,1 avg ms
VirtualOverride	31,4 avg ms
Lambda Standard	8,3 avg ms
Action (Thunk)	10,7 avg ms
Lazy Evaluation	12,8 avg ms
Microsoft Scheduler	11,2 avg ms
UPPAAL Parser	11,5 avg ms
Threads No Scheduler	11,8 avg ms

Table 8.13: The average of the tests are measured in milliseconds. This is for double without writing it to the RAM. Laptop 1 5.1.

Standard	242,6 avg ms
SingleThread	939,2 avg ms
Parallel.For	373,7 avg ms
Reflection.For	8034,3 avg ms
Lambda.For	791,9 avg ms
VirtualOverride	270,3 avg ms
Lambda Standard	371,6 avg ms
Action (Thunk)	246,7 avg ms
Lazy Evaluation	417,3 avg ms
Microsoft Scheduler	242,3 avg ms
UPPAAL Parser	419,3 avg ms
Threads No Scheduler	240,3 avg ms

Table 8.14: The average of the tests are measured in milliseconds. This is for double with writing it to the RAM. Laptop 1 5.1.

Standard	14,9 avg ms
SingleThread	183,6 avg ms
Parallel.For	148,2 avg ms
Reflection.For	7502,1 avg ms
Lambda.For	228,4 avg ms
VirtualOverride	9,3 avg ms
Lambda Standard	9,1 avg ms
Action (Thunk)	10,9 avg ms
Lazy Evaluation	13,2 avg ms
Microsoft Scheduler	12,8 avg ms
UPPAAL Parser	12,9 avg ms
Threads No Scheduler	10,6 avg ms

Table 8.15: The average of the tests are measured in milliseconds. This is for int without writing it to the RAM. Laptop 1 5.1.

Standard	212,5 avg ms
SingleThread	823,7 avg ms
Parallel.For	334,7 avg ms
Reflection.For	7599,2 avg ms
Lambda.For	599,8 avg ms
VirtualOverride	206,7 avg ms
Lambda Standard	246 avg ms
Action (Thunk)	200,3 avg ms
Lazy Evaluation	282,5 avg ms
Microsoft Scheduler	203,3 avg ms
UPPAAL Parser	288,8 avg ms
Threads No Scheduler	200,5 avg ms

Table 8.16: The average of the tests are measured in milliseconds. This is for int with writing it to the RAM. Laptop 1 5.1.

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	29	32	23	24	28	28	25	23	26	23
Single	296	311	293	294	291	296	290	292	291	293
Parallel.For	313	320	318	310	306	304	304	323	307	302
Reflection.For	15113	15145	14912	14877	14869	15123	14871	14875	14862	14881
Lambda.For	420	352	345	351	354	355	355	345	351	350
VirtualOverride	53	52	50	54	54	52	50	56	54	53
Lambda Standard	18	17	17	17	17	17	17	17	17	17
Action (Thunk)	18	17	17	17	17	17	17	17	17	17
Lazy	21	21	21	21	21	21	21	21	23	21
Microsoft Scheduler	20	19	19	19	19	19	19	19	19	19
UPPAAL Parser	20	20	20	20	20	20	20	20	20	20
Threads No Scheduler	18	17	17	17	17	17	19	17	17	17

Table 8.17: All the results are measured in milliseconds. This test is double calculation without writing it to the RAM. Laptop 2 5.1

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	706	798	705	739	735	707	683	676	733	646
Single	1485	1432	1408	1632	1536	1440	1423	1440	1495	1436
Parallel.For	1087	973	973	990	999	987	977	985	982	997
Reflection.For	15839	15782	15733	15958	15717	15477	15401	15536	15664	15499
Lambda.For	1543	1395	1421	1432	1433	1409	1427	1427	1416	1413
VirtualOverride	636	624	627	625	651	626	625	619	621	626
Lambda Standard	760	751	756	760	761	755	762	756	753	753
Action (Thunk)	685	660	645	671	650	634	626	648	684	636
Lazy	836	813	814	840	827	802	840	820	825	806
Microsoft Scheduler	643	612	592	623	600	608	594	594	609	630
UPPAAL Parser	798	831	835	851	800	786	810	807	809	805
Threads No Scheduler	668	586	588	591	641	605	632	598	632	607

Table 8.18: All the results are measured in miliseconds. This test is double calculation with writing it to the RAM. Laptop 2 5.1

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	25	25	26	25	23	25	24	28	27	24
Single	296	289	290	294	292	296	290	288	290	295
Parallel.For	308	311	314	311	311	316	303	314	312	324
Reflection.For	14922	14909	14907	14874	14857	15006	14943	14915	14840	15420
Lambda.For	352	352	353	351	349	351	351	351	348	353
VirtualOverride	17	17	17	17	17	17	17	17	17	17
Lambda Standard	17	17	17	17	17	17	17	17	18	17
Action (Thunk)	18	18	18	18	18	18	18	18	18	18
Lazy	21	21	21	21	21	21	21	22	21	21
Microsoft Scheduler	19	19	19	19	19	20	20	20	20	19
UPPAAL Parser	20	20	20	20	20	20	21	20	20	20
Threads No Scheduler	17	17	17	17	17	17	17	17	19	17

Table 8.19: All the results are measured in milliseconds. This test is int calculation without writing it to the RAM. Laptop 2 5.1

Test name	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8	test 9	test 10
Standard	459	493	482	484	470	475	574	589	482	474
Single	1060	1044	1024	1014	1029	1026	1127	1125	1028	1020
Parallel.For	848	778	795	787	788	779	777	797	776	800
Reflection.For	15414	15359	15196	15224	15146	16260	17154	15734	15160	15127
Lambda.For	998	969	975	984	978	1121	1239	982	964	962
VirtualOverride	440	447	440	442	441	440	444	446	441	441
Lambda Standard	478	505	489	500	483	500	479	482	484	485
Action (Thunk)	425	430	425	407	418	430	442	418	416	425
Lazy	505	509	506	521	508	522	546	515	503	504
Microsoft Scheduler	494	485	397	405	413	427	412	422	420	398
UPPAAL Parser	523	423	526	509	510	537	536	522	521	514
Threads No Scheduler	418	515	397	420	432	428	412	402	409	403

Table 8.20: All the results are measured in miliseconds. This test is int calculation with writing it to the RAM. Laptop 2 5.1

Standard	26,1 avg ms
SingleThread	294,7 avg ms
Parallel.For	310,7 avg ms
Reflection.For	14952,8 avg ms
Lambda.For	357,8 avg ms
VirtualOverride	52,8 avg ms
Lambda Standard	17,1 avg ms
Action (Thunk)	17,1 avg ms
Lazy Evaluation	21,2 avg ms
Microsoft Scheduler	19,1 avg ms
UPPAAL Parser	20 avg ms
Threads No Scheduler	17,3 avg ms

Table 8.21: The average of the tests are measured in milliseconds. This is for double without writing it to the RAM. Laptop 2 5.1

Standard	712,8 avg ms
SingleThread	1472,7 avg ms
Parallel.For	995 avg ms
Reflection.For	15660,6 avg ms
Lambda.For	1431,6 avg ms
VirtualOverride	628 avg ms
Lambda Standard	756,7 avg ms
Action (Thunk)	653,9 avg ms
Lazy Evaluation	822,3 avg ms
Microsoft Scheduler	610,5 avg ms
UPPAAL Parser	813,2 avg ms
Threads No Scheduler	614,8 avg ms

Table 8.22: The average of the tests are measured in milliseconds. This is for double with writing it to the RAM. Laptop 2 5.1

Standard	25,2 avg ms
SingleThread	292 avg ms
Parallel.For	312,4 avg ms
Reflection.For	14959,3 avg ms
Lambda.For	351,1 avg ms
VirtualOverride	17 avg ms
Lambda Standard	17,1 avg ms
Action (Thunk)	18 avg ms
Lazy Evaluation	21,1 avg ms
Microsoft Scheduler	19,4 avg ms
UPPAAL Parser	20,1 avg ms
Threads No Scheduler	17,2 avg ms

Table 8.23: The average of the tests are measured in milliseconds. This is for int without writing it to the RAM. Laptop 2 5.1

Standard	498,2 avg ms
SingleThread	1049,7 avg ms
Parallel.For	792,5 avg ms
Reflection.For	15577,4 avg ms
Lambda.For	1017,2 avg ms
VirtualOverride	442,2 avg ms
Lambda Standard	488,5 avg ms
Action (Thunk)	423,6 avg ms
Lazy Evaluation	513,9 avg ms
Microsoft Scheduler	427,3 avg ms
UPPAAL Parser	512,1 avg ms
Threads No Scheduler	423,6 avg ms

Table 8.24: The average of the tests are measured in milliseconds. This is for int with writing it to the RAM. Laptop 2 5.1

8.2 Code

Listing 8.1: This code is the library code of the Dependency Scheduler.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Threading.Tasks;
5  using System.Threading;
6  using System.Reflection;
7
8  namespace Dependency_Task_Scheduling
9  {
10     class TaskItem
11     {
12         private MethodInfo _methodInfo;
13         private bool _working = false, _done = false, _named,
               _actionbool;
14         private int _name;
15         private List<int> _dependant = new List<int>();
16         private object _operatingObj = null;
17         private object[] _objs = null;
18         private Action _action;
19
20         public TaskItem(MethodInfo methodInfo, object[] objs)
21         {
22             _methodInfo = methodInfo;
23             _named = false;
24             _objs = objs;
25         }
26         public TaskItem(MethodInfo methodInfo, int name, object[]
               objs)
27         {
28             _methodInfo = methodInfo;
29             _name = name;
30             _named = true;
31             _objs = objs;
32         }
33         public TaskItem(MethodInfo methodInfo, int name, List<int>
               dependant, object[] objs)
34         {
35             _methodInfo = methodInfo;
36             _name = name;
```

```
37              _dependant.AddRange(dependant);
38              _named = true;
39              _objs = objs;
40          }
41          public TaskItem(MethodInfo methodInfo, List<int> dependant
                , object[] objs)
42          {
43              _methodInfo = methodInfo;
44              _dependant.AddRange(dependant);
45              _named = false;
46              _objs = objs;
47          }
48          public TaskItem(MethodInfo methodInfo, object operatingObj
                , object[] objs)
49          {
50              _methodInfo = methodInfo;
51              _named = false;
52              _operatingObj = operatingObj;
53              _objs = objs;
54          }
55          public TaskItem(MethodInfo methodInfo, int name, object
                operatingObj, object[] objs)
56          {
57              _methodInfo = methodInfo;
58              _name = name;
59              _named = true;
60              _operatingObj = operatingObj;
61              _objs = objs;
62          }
63          public TaskItem(MethodInfo methodInfo, int name, List<int>
                 dependant, object operatingObj, object[] objs)
64          {
65              _methodInfo = methodInfo;
66              _name = name;
67              _dependant.AddRange(dependant);
68              _named = true;
69              _operatingObj = operatingObj;
70              _objs = objs;
71          }
72          public TaskItem(MethodInfo methodInfo, List<int> dependant
                , object operatingObj, object[] objs)
73          {
74              _methodInfo = methodInfo;
```

```
75              _dependant.AddRange(dependant);
76              _named = false;
77              _operatingObj = operatingObj;
78              _objs = objs;
79          }
80          public TaskItem(Action action)
81          {
82              _action = action;
83              _named = false;
84              _actionbool = true;
85          }
86          public TaskItem(Action action, int name)
87          {
88              _action = action;
89              _actionbool = true;
90              _name = name;
91              _named = true;
92          }
93          public TaskItem(Action action, int name, List<int>
                dependant)
94          {
95              _action = action;
96              _actionbool = true;
97              _name = name;
98              _named = true;
99              _dependant.AddRange(dependant);
100         }
101         public TaskItem(Action action, List<int> dependant)
102         {
103             _action = action;
104             _actionbool = true;
105             _named = false;
106             _dependant.AddRange(dependant);
107         }
108
109         //Corrospond to a Enumname
110         public int getName() { return _name; }
111         //Corrospond to a Enumname
112         public List<int> getDependencies() { return _dependant; }
113         public MethodInfo MethodInfo() { return _methodInfo; }
114         public bool IsWorking() { return _working; }
115         public bool isNamed() { return _named; }
116         public bool Work()
```

```
117         {
118             _working = true;
119             return _working;
120         }
121         public void Finished() { _done = true; }
122         public bool isDone() { return _done; }
123         public object getObj() { return _operatingObj; }
124         public object[] getObjs() { return _objs; }
125         public Action getAction() { return _action; }
126         public bool isAction() { return _actionbool; }
127     }
128
129     class DependencyScheduler
130     {
131         private int workerThreads, completionPortThreads,
                optimalThreadSize;
132         private bool runningScheduler = false, canRun = true,
                removedItem=false, addingTasks = false;
133         private AutoResetEvent waitHandle = new AutoResetEvent(
                false), waitMain;
134         private List<TaskItem> workList = new List<TaskItem>();
135         private List<int> DependencyTasks = new List<int>(),
                currentList;
136
137         //Recommanded to use and not handle the number of threads
                by yourself.
138         public DependencyScheduler()
139         {
140             OptimalTaskSize();
141             ThreadPool.GetAvailableThreads(out workerThreads, out
                    completionPortThreads);
142         }
143
144         private void OptimalTaskSize()
145         {
146             ThreadPool.GetMinThreads(out workerThreads, out
                    completionPortThreads);
147             optimalThreadSize = workerThreads;
148         }
149
150         /// <SchedulerWork>
151         /// Assign task to a thread, and remove tasks as they have
                completed
```

```
152         /// </SchedulerWork>
153         private void SchedulerWork()
154         {
155             //Start Job for it to work on linux set a parameter
                    int taskcount instead of workList.count
156             for (int taskNumber = 0; taskNumber < workList.Count;
                    taskNumber++ )
157             {
158                 //Check if thread is already worked on
159                 if (!workList[taskNumber].IsWorking())
160                 {
161                     //Check if there is dependencies if there is
                            check if they are fulfilled
162                     currentList = workList[taskNumber].
                            getDependencies();
163                     if (currentList.Count > 0)
164                     {
165                         canRun = true;
166                         for(int i = 0; i < currentList.Count;i++)
167                         {
168                             if (DependencyTasks[currentList[i]] >
                                    0)
169                                 canRun = false;
170                         }
171                         if (canRun)
172                         {
173                             ThreadPool.QueueUserWorkItem(
                                    ThreadInvoker, workList[taskNumber
                                    ]);
174                             workList[taskNumber].Work();
175                         }
176                     }
177                     else
178                     {
179                         ThreadPool.QueueUserWorkItem(ThreadInvoker
                                , workList[taskNumber]);
180                         workList[taskNumber].Work();
181                     }
182                 }
183             }
184             //Remove job from List
185             if (!addingTasks)
186             {
```

```
187                 removedItem = false;
188                 for (int i = workList.Count - 1; i >= 0; i--)
189                 {
190                     if (workList[i].isDone())
191                     {
192                         if (workList[i].isNamed())
193                         {
194                             DependencyTasks[workList[i].getName()
                                ]--;
195                             removedItem = true;
196                         }
197                         workList.RemoveAt(i);
198 
199                     }
200                 }
201                 //This is required else if the last item was a
                    dependency it will get to be a standstill
202                 if (removedItem)
203                     SchedulerWork();
204             }
205         }
206 
207         //Start the job in a thread.
208         private void ThreadInvoker(Object threadContext)
209         {
210             TaskItem TI = (TaskItem)threadContext;
211             ThreadExecution(TI);
212             TI.Finished();
213             //Signal a thread has finished
214             waitHandle.Set();
215         }
216 
217         //Execute Thread
218         public virtual void ThreadExecution(TaskItem TI)
219         {
220             if (!TI.isAction())
221             {
222                 MethodInfo task = TI.MethodInfo();
223                 //Reflection invoke a method to do on a thread
224                 task.Invoke(TI.getObj(), TI.getObjs());
225             }
226             else
227                 //Action or Thunk
```

```
228                 TI.getAction().Invoke();
229         }
230 
231         //Start the ThreadScheduler in a thread.
232         private void ThreadScheduler(Object threadContext)
233         {
234             waitHandle = new AutoResetEvent(false);
235             //Start the actual SchedulerWork
236             SchedulerWork();
237             //Check if the tasks has finished
238             if(workList.Count == 0)
239             {
240                 waitMain.Set();
241                 waitHandle.Set();
242             }
243             //Wait until new jobs assignment or a thread has
                    finished
244             waitHandle.WaitOne();
245 
246             if (workList.Count > 0)
247                 ThreadScheduler(null);
248             else
249             {
250                 runningScheduler = false;
251             }
252         }
253 
254         //Private InitScheduler
255         private void InitScheduler()
256         {
257             //Start Scheduler if not started
258             if (!runningScheduler)
259             {
260                 runningScheduler = true;
261                 ThreadPool.QueueUserWorkItem(ThreadScheduler, null
                        );
262                 //init the main thread to be able to sleep
263                 waitMain = new AutoResetEvent(false);
264             }
265             else
266             {
267                 //Restart scheduler if sleeping
268                 waitHandle.Set();
```

```
269             }
270         }
271
272         //Increase Dependency List for the for methods
273         public void IncreaseDependencyList(int name)
274         {
275             for (int i = DependencyTasks.Count; i <= name; i++)
276             {
277                 DependencyTasks.Add(0);
278             }
279         }
280
281         //set Amount of Threads
282         public void SetMaxThreads(int numberOfWorkerThreads)
283         {
284             if (numberOfWorkerThreads <= 1)
285                 return;
286             ThreadPool.GetMaxThreads(out workerThreads, out
                    completionPortThreads);
287             ThreadPool.SetMaxThreads(numberOfWorkerThreads,
                    numberOfWorkerThreads);
288             workerThreads = numberOfWorkerThreads;
289         }
290
291         //set Amount of Threads
292         public void SetMinThreads(int numberOfWorkerThreads)
293         {
294             if (numberOfWorkerThreads <= 1)
295                 return;
296             ThreadPool.SetMinThreads(numberOfWorkerThreads,
                    numberOfWorkerThreads);
297             workerThreads = numberOfWorkerThreads;
298             completionPortThreads = numberOfWorkerThreads;
299         }
300
301         //set Amount of Threads to corrospond to the number of
                threads on your CPU +1 for scheduler.
302         public void SetThreadsForMyCPU()
303         {
304             SetMinThreads();
305             ThreadPool.SetMinThreads(workerThreads + 1,
                    completionPortThreads + 1);
```

```
306             ThreadPool.SetMaxThreads(workerThreads + 1,
                    completionPortThreads + 1);
307         }
308
309         //Get Maximum amount of threads.
310         public void SetMaxThreads()
311         {
312             ThreadPool.GetMaxThreads(out workerThreads, out
                    completionPortThreads);
313         }
314
315         //Get the minimum threads, which is the number of threads
                your CPU has, however the Scheduler will take one
                thread.
316         public void SetMinThreads()
317         {
318             ThreadPool.GetMinThreads(out workerThreads, out
                    completionPortThreads);
319         }
320
321         //Get information on the last used number of threads
                either minimum or the highest.
322         public void WriteNumberofThreads()
323         {
324             Console.WriteLine("WorkerThreads: {0},
                    CompletionPortThreads: {1}", workerThreads,
                    completionPortThreads);
325         }
326
327         /// <summary>
328         /// This method is used if there is no requirements.
329         /// </summary>
330         public void AddTask(Type classType, string methodName,
                object initClass, params object[] objs)
331         {
332             MethodInfo Task = classType.GetMethod(methodName);
333             TaskItem TI = new TaskItem(Task, initClass, objs);
334             workList.Add(TI);
335             InitScheduler();
336         }
337
338         /// <summary>
```

```
339         /// This method is used if something else requires this
                one to be finished first.
340         /// </summary>
341         public void AddTaskNamed(Type classType, string methodName
                , object initClass,int name, params object[] objs)
342         {
343             MethodInfo Task = classType.GetMethod(methodName);
344             IncreaseDependencyList(name);
345             DependencyTasks[name]++;
346             TaskItem TI = new TaskItem(Task,name, initClass,objs);
347             workList.Add(TI);
348             InitScheduler();
349         }
350 
351         /// <summary>
352         /// This Methods will specify the task name and what it
                depends upon.
353         /// </summary>
354         public void AddTaskNamedAndDependencies(Type classType,
                string methodName, object initClass, int name, List<
                int> dependencies, params object[] objs)
355         {
356             MethodInfo Task = classType.GetMethod(methodName);
357             IncreaseDependencyList(name);
358             DependencyTasks[name]++;
359             workList.Add(new TaskItem(Task, name, dependencies,
                    initClass, objs));
360             InitScheduler();
361         }
362 
363         /// <summary>
364         /// This Methods will only look at what it depends upon.
365         /// </summary>
366         public void AddTaskDependenciesNoName(Type classType,
                string methodName, object initClass, List<int>
                dependencies, params object[] objs)
367         {
368             MethodInfo Task = classType.GetMethod(methodName);
369 
370             workList.Add(new TaskItem(Task, dependencies,initClass
                    , objs));
371             InitScheduler();
372         }
```

```
373
374         /// <summary>
375         /// This method split the task into all availble threads
                with no requirement. Before Using make sure to Lock
                then unlock.
376         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
377         /// </summary>
378         public void ParallelForTask(Type classType, string
                methodName, object InitClass, int from, int to, params
                 object[] objs)
379         {
380             MethodInfo Task = classType.GetMethod(methodName);
381             //TaskItem TI = new TaskItem(Task, InitClass);
382             //TI.setParallel();
383             int max = to - from, threads = optimalThreadSize,
                    setUpperBound, setLowerBound, listIndex = objs.
                    Length + 2;
384             object[] objstemp;
385             objstemp = new object[listIndex];
386             for (int i = 0; i < objs.Length; i++)
387             {
388                 objstemp[i] = objs[i];
389             }
390             if (max < threads)
391                 threads = max;
392             for (int i = 0; i < threads; i++)
393             {
394                 if (i + 1 == threads)
395                 {
396                     setUpperBound = to;
397                     setLowerBound = from + (max / threads) * i;
398                 }
399                 else
400                 {
401                     setUpperBound = from + (max / threads) * (i +
                            1);
402                     setLowerBound = from + (max / threads) * i;
403                 }
404                 objstemp[listIndex - 2] = setLowerBound;
405                 objstemp[listIndex - 1] = setUpperBound;
406                 TaskItem TI = new TaskItem(Task, InitClass,
                        objstemp);
```

```
407                 workList.Add(TI);
408             }
409             InitScheduler();
410         }
411 
412         /// <summary>
413         /// This method split the task into all available threads
                where all tasks are named and required to be run
                before something else. Before Using make sure to Lock
                then unlock.
414         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
415         /// </summary>
416         public void ParallelForTaskNamed(Type classType, string
                methodName, object InitClass, int from, int to, int
                name, params object[] objs)
417         {
418             MethodInfo Task = classType.GetMethod(methodName);
419             //TaskItem TI = new TaskItem(Task, InitClass);
420             //TI.setParallel();
421             int max = to - from, threads = optimalThreadSize,
                    setUpperBound, setLowerBound, listIndex = objs.
                    Length + 2;
422             object[] objstemp;
423             objstemp = new object[listIndex];
424             IncreaseDependencyList(name);
425             for (int i = 0; i < objs.Length; i++)
426             {
427                 objstemp[i] = objs[i];
428             }
429             if (max < threads)
430                 threads = max;
431             for (int i = 0; i < threads; i++)
432             {
433                 if (i + 1 == threads)
434                 {
435                     setUpperBound = to;
436                     setLowerBound = from + (max / threads) * i;
437                 }
438                 else
439                 {
440                     setUpperBound = from + (max / threads) * (i +
                            1);
```

```
441                     setLowerBound = from + (max / threads) * i;
442                 }
443                 objstemp[listIndex - 2] = setLowerBound;
444                 objstemp[listIndex - 1] = setUpperBound;
445                 DependencyTasks[name]++;
446                 TaskItem TI = new TaskItem(Task, name, InitClass,
                        objstemp);
447                 workList.Add(TI);
448             }
449             InitScheduler();
450         }
451
452         /// <summary>
453         /// This method split the task into all availble threads
                where all tasks are named and requires something else
                to be run before this. Before Using make sure to Lock
                then unlock.
454         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
455         /// </summary>
456         public void ParallelForTaskNamedAndDependencies(Type
                classType, string methodName, object InitClass, int
                from, int to, int name, List<int> dependencies, params
                 object[] objs)
457         {
458             MethodInfo Task = classType.GetMethod(methodName);
459             //TaskItem TI = new TaskItem(Task, InitClass);
460             //TI.setParallel();
461             int max = to - from, threads = optimalThreadSize,
                    setUpperBound, setLowerBound, listIndex = objs.
                    Length + 2;
462             object[] objstemp;
463             objstemp = new object[listIndex];
464             IncreaseDependencyList(name);
465
466             for (int i = 0; i < objs.Length; i++)
467             {
468                 objstemp[i] = objs[i];
469             }
470             if (max < threads)
471                 threads = max;
472             for (int i = 0; i < threads; i++)
473             {
```

```
474                 if (i + 1 == threads)
475                 {
476                     setUpperBound = to;
477                     setLowerBound = from + (max / threads) * i;
478                 }
479                 else
480                 {
481                     setUpperBound = from + (max / threads) * (i +
                        1);
482                     setLowerBound = from + (max / threads) * i;
483                 }
484                 objstemp[listIndex - 2] = setLowerBound;
485                 objstemp[listIndex - 1] = setUpperBound;
486                 DependencyTasks[name]++;
487                 TaskItem TI = new TaskItem(Task, name,
                    dependencies, InitClass, objstemp);
488                 workList.Add(TI);
489             }
490             InitScheduler();
491         }
492
493         /// <summary>
494         /// This method split the task into all availble threads
                this will run when the dependencies has been run.
                Before Using make sure to Lock then unlock.
495         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
496         /// </summary>
497         public void ParallelForTaskDependenciesNoName(Type
                classType, string methodName, object InitClass, int
                from, int to, List<int> dependencies, params object[]
                objs)
498         {
499             MethodInfo Task = classType.GetMethod(methodName);
500             //TaskItem TI = new TaskItem(Task, InitClass);
501             //TI.setParallel();
502             int max = to - from, threads = optimalThreadSize,
                    setUpperBound, setLowerBound, listIndex = objs.
                    Length + 2;
503             object[] objstemp;
504             objstemp = new object[listIndex];
505
506             for (int i = 0; i < objs.Length; i++)
```

```
507             {
508                 objstemp[i] = objs[i];
509             }
510             if (max < threads)
511                 threads = max;
512             for (int i = 0; i < threads; i++)
513             {
514                 if (i + 1 == threads)
515                 {
516                     setUpperBound = to;
517                     setLowerBound = from + (max / threads) * i;
518                 }
519                 else
520                 {
521                     setUpperBound = from + (max / threads) * (i +
                            1);
522                     setLowerBound = from + (max / threads) * i;
523                 }
524                 objstemp[listIndex - 2] = setLowerBound;
525                 objstemp[listIndex - 1] = setUpperBound;
526                 TaskItem TI = new TaskItem(Task, dependencies,
                        InitClass, objstemp);
527                 workList.Add(TI);
528             }
529             InitScheduler();
530         }
531
532         /// <summary>
533         /// Choose yourself how many tasks a method should be
                split into.
534         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
535         /// </summary>
536         public void ParallelForTaskSpecific(Type classType, string
                methodName, object InitClass, int from, int to, int
                numberofTasks, params object[] objs)
537         {
538             MethodInfo Task = classType.GetMethod(methodName);
539             int max = to - from, setUpperBound, setLowerBound,
                    listIndex = objs.Length + 2;
540             object[] objstemp;
541             objstemp = new object[listIndex];
542             for (int i = 0; i < objs.Length; i++)
```

```
543             {
544                 objstemp[i] = objs[i];
545             }
546             for (int i = 0; i < numberofTasks; i++)
547             {
548 
549                 if (i + 1 == numberofTasks)
550                 {
551                     setUpperBound = to;
552                     setLowerBound = from + (max / numberofTasks) *
                            i;
553                 }
554                 else
555                 {
556                     setUpperBound = from + (max / numberofTasks) *
                            (i + 1);
557                     setLowerBound = from + (max / numberofTasks) *
                            i;
558                 }
559                 objstemp[listIndex - 2] = setLowerBound;
560                 objstemp[listIndex - 1] = setUpperBound;
561                 TaskItem TI = new TaskItem(Task, InitClass,
                        objstemp);
562                 workList.Add(TI);
563             }
564             InitScheduler();
565         }
566 
567         /// <summary>
568         /// Choose yourself how many tasks a method should be
                split into and if something has to be dependant upon
                this.
569         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
570         /// </summary>
571         public void ParallelForTaskSpecificNamed(Type classType,
                string methodName, object InitClass, int from, int to,
                 int numberofTasks, int name, params object[] objs)
572         {
573             MethodInfo Task = classType.GetMethod(methodName);
574             int max = to - from, setUpperBound, setLowerBound,
                    listIndex = objs.Length + 2;
575             object[] objstemp;
```

```
576             objstemp = new object[listIndex];
577             IncreaseDependencyList(name);
578             for (int i = 0; i < objs.Length; i++)
579             {
580                 objstemp[i] = objs[i];
581             }
582             for (int i = 0; i < numberofTasks; i++)
583             {
584 
585                 if (i + 1 == numberofTasks)
586                 {
587                     setUpperBound = to;
588                     setLowerBound = from + (max / numberofTasks) *
                            i;
589                 }
590                 else
591                 {
592                     setUpperBound = from + (max / numberofTasks) *
                            (i + 1);
593                     setLowerBound = from + (max / numberofTasks) *
                            i;
594                 }
595                 objstemp[listIndex - 2] = setLowerBound;
596                 objstemp[listIndex - 1] = setUpperBound;
597                 DependencyTasks[name]++;
598                 TaskItem TI = new TaskItem(Task, name, InitClass,
                        objstemp);
599                 workList.Add(TI);
600             }
601             InitScheduler();
602         }
603 
604         /// <summary>
605         /// Choose yourself how many tasks a method should be
                split into and if something has to be dependant upon
                this as well it depends upon something else.
606         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
607         /// </summary>
608         public void ParallelForTaskSpecificNamedAndDependencies(
                Type classType, string methodName, object InitClass,
                int from, int to, int numberofTasks, int name, List<
                int> dependencies, params object[] objs)
```

```
609         {
610             MethodInfo Task = classType.GetMethod(methodName);
611             int max = to - from, setUpperBound, setLowerBound,
                    listIndex = objs.Length + 2;
612             object[] objstemp;
613             objstemp = new object[listIndex];
614             IncreaseDependencyList(name);
615             for (int i = 0; i < objs.Length; i++)
616             {
617                 objstemp[i] = objs[i];
618             }
619             for (int i = 0; i < numberofTasks; i++)
620             {
621
622                 if (i + 1 == numberofTasks)
623                 {
624                     setUpperBound = to;
625                     setLowerBound = from + (max / numberofTasks) *
                            i;
626                 }
627                 else
628                 {
629                     setUpperBound = from + (max / numberofTasks) *
                            (i + 1);
630                     setLowerBound = from + (max / numberofTasks) *
                            i;
631                 }
632                 objstemp[listIndex - 2] = setLowerBound;
633                 objstemp[listIndex - 1] = setUpperBound;
634                 DependencyTasks[name]++;
635                 TaskItem TI = new TaskItem(Task,name,dependencies,
                        InitClass, objstemp);
636                 workList.Add(TI);
637             }
638             InitScheduler();
639         }
640
641         /// <summary>
642         /// Choose yourself how many tasks a method should be
                split into and if it depends upon something without
                anything else depends on this.
643         /// Just remember that the method should use 2 int
                parameters for the for loops as the last parameters.
```

```
644         /// </summary>
645         public void ParallelForTaskSpecificDependenciesNoName(Type
                classType, string methodName, object InitClass, int
               from, int to, int numberofTasks, List<int>
               dependencies, params object[] objs)
646         {
647             MethodInfo Task = classType.GetMethod(methodName);
648             int max = to - from, setUpperBound, setLowerBound,
                   listIndex = objs.Length + 2;
649             object[] objstemp;
650             objstemp = new object[listIndex];
651             for (int i = 0; i < objs.Length; i++)
652             {
653                 objstemp[i] = objs[i];
654             }
655             for (int i = 0; i < numberofTasks; i++)
656             {
657
658                 if (i + 1 == numberofTasks)
659                 {
660                     setUpperBound = to;
661                     setLowerBound = from + (max / numberofTasks) *
                           i;
662                 }
663                 else
664                 {
665                     setUpperBound = from + (max / numberofTasks) *
                           (i + 1);
666                     setLowerBound = from + (max / numberofTasks) *
                           i;
667                 }
668                 objstemp[listIndex - 2] = setLowerBound;
669                 objstemp[listIndex - 1] = setUpperBound;
670                 TaskItem TI = new TaskItem(Task, dependencies,
                       InitClass, objstemp);
671                 workList.Add(TI);
672             }
673             InitScheduler();
674         }
675         /// <summary>
676         /// Add an action Task.
677         /// </summary>
678         public void AddTask(Action action)
```

```
679         {
680             TaskItem TI = new TaskItem(action);
681             workList.Add(TI);
682             InitScheduler();
683         }
684         /// <summary>
685         /// Add an action Task. With a name for the scheduler.
686         /// </summary>
687         public void AddTask(Action action, int name)
688         {
689             IncreaseDependencyList(name);
690             DependencyTasks[name]++;
691             TaskItem TI = new TaskItem(action,name);
692             workList.Add(TI);
693             InitScheduler();
694         }
695         /// <summary>
696         /// Add an action Task. With a name for the scheduler and
                dependency list.
697         /// </summary>
698         public void AddTask(Action action, int name, List<int>
                dependencies)
699         {
700             IncreaseDependencyList(name);
701             DependencyTasks[name]++;
702             TaskItem TI = new TaskItem(action,name,dependencies);
703             workList.Add(TI);
704             InitScheduler();
705         }
706         /// <summary>
707         /// Add an action Task. With a dependency list.
708         /// </summary>
709         public void AddTask(Action action, List<int> dependencies)
710         {
711             TaskItem TI = new TaskItem(action,dependencies);
712             workList.Add(TI);
713             InitScheduler();
714         }
715
716         /// <summary>
717         ///This lock is made so the scheduler will not remove
                elements from the worklist while adding as this can
                give problems.
```

```
718         /// </summary>
719         public void AddingTaskLock()
720         {
721             addingTasks = true;
722         }
723 
724         /// <summary>
725         ///This unlock is made so the scheduler will again remove
                elements from the worklist at any given time a task
                has finished.
726         ///It will also reignite the scheduler if it went to sleep
                .
727         /// </summary>
728         public void AddingTaskUnlock()
729         {
730             addingTasks = false;
731             InitScheduler();
732         }
733 
734         /// <summary>
735         /// MainThread Wait will wakeup when tasks are done, this
                will wait on all tasks
736         /// </summary>
737         public void WaitForTasks()
738         {
739             waitMain.WaitOne();
740         }
741 
742         /// <summary>
743         /// This just makes an int list from the enum you might
                have created, you can give as many ints as you want.
744         /// </summary>
745         public List<int> Dependencies(params int[] dependencies)
746         {
747             List<int> depend = new List<int>();
748             for (int i = 0; i < dependencies.Length; i++)
749                 depend.Add(dependencies[i]);
750             return depend;
751         }
752     }
753 
754     class TaskParser
755     {
```

```
756         int _name;
757         List<int> _dependency = new List<int>();
758         string _method;
759         public TaskParser(int name, List<int> dependency, string
                methodName)
760         {
761             _name = name;
762             _dependency = dependency;
763             _method = methodName;
764         }
765         public int GetName() { return _name; }
766         public string GetMethod() { return _method; }
767         public List<int> GetDependency() { return _dependency; }
768     }
769
770     class DependencyTaskParser
771     {
772         //Using the Dependency Scheduler
773         DependencyScheduler DS;
774         //Creates an offset for the different names
775         int _offset = 1000, _resetNumber;
776
777         public DependencyTaskParser(DependencyScheduler
                DependencyScheduler)
778         {
779             DS = DependencyScheduler;
780             DS.IncreaseDependencyList(_offset);
781         }
782
783         public DependencyTaskParser(DependencyScheduler
                DependencyScheduler,int offset)
784         {
785             DS = DependencyScheduler;
786             _offset = offset;
787             _resetNumber = offset;
788             DS.IncreaseDependencyList(_offset);
789         }
790
791         //Reset will continue to increase however, this allows it
                to be reset.
792         public void ResetOffset()
793         {
794             _offset = _resetNumber;
```

```
795         }
796
797         /// <summary>
798         /// Give the Classtype from where the Tasks are from, the
                class they operate on, the list of systems from UPPAAL
                 where you have renamed to the function names,
                dependency that you created in UPPAAL
799         /// </summary>
800         public void GiveAnalysis(Type classType, object initClass,
                 string methodNames, bool[][] dependency)
801         {
802             DS.AddingTaskLock();
803             string[] splitter = methodNames.Split(',');
804             int startoffset = _offset;
805
806             List<TaskParser> ArrayTaskParser = new List<TaskParser
                    >();
807             bool[] accessList = new bool[splitter.Length];
808             DS.IncreaseDependencyList(_offset + splitter.Length);
809             for (int i = 0; i < splitter.Length; i++)
810             {
811                 List<int> dependencyList = new List<int>();
812                 for (int j = 0; j < splitter.Length; j++)
813                 {
814                     if (dependency[i][j])
815                     {
816                         dependencyList.Add(j + startoffset);
817                     }
818                 }
819
820                 if (dependencyList.Count <= 0)
821                 {
822                     DS.AddTaskNamed(classType, RemoveWhitespace(
                            splitter[i]), initClass, _offset + i);
823                     accessList[i] = true;
824                 }
825                 else
826                 {
827                     ArrayTaskParser.Add(new TaskParser(_offset + i
                            , dependencyList, RemoveWhitespace(
                            splitter[i])));
828                     accessList[i] = false;
829                 }
```

```
830
831             }
832             TasksPartTwo(ArrayTaskParser,classType,initClass,
                    accessList);
833             _offset += splitter.Length;
834             DS.AddingTaskUnlock();
835
836         }
837
838         /// <summary>
839         /// This method is to be sure that the dependency is
                correctly placed so there is no risk of the dependant
                to run before the function it depends on.
840         /// This is able to run while the Scheduler is running and
                 the cleaning is disabled.
841         /// </summary>
842         private void TasksPartTwo(List<TaskParser> ArrayTaskParser
                , Type classType, object initClass, bool[] accessList)
843         {
844             bool canRun = true;
845             List<int> currentList;
846             for (int i = 0; i < ArrayTaskParser.Count; i++)
847             {
848                 canRun = true;
849                 currentList = ArrayTaskParser[i].GetDependency();
850                 for (int j = 0; j < currentList.Count; j++)
851                 {
852
853                     if (!accessList[currentList[j]-_offset])
854                     {
855                         canRun = false;
856                     }
857                 }
858                 if (canRun)
859                 {
860                     DS.AddTaskNamedAndDependencies(classType,
                            ArrayTaskParser[i].GetMethod(), initClass,
                             ArrayTaskParser[i].GetName(),
                            ArrayTaskParser[i].GetDependency());
861                     accessList[ArrayTaskParser[i].GetName() -
                            _offset] = true;
862                     ArrayTaskParser.RemoveAt(i);
863                     i--;
```

```
864                 }
865             }
866             if (ArrayTaskParser.Count > 0)
867                 TasksPartTwo(ArrayTaskParser, classType, initClass
                        , accessList);
868         }
869 
870         private static string RemoveWhitespace(string str)
871         {
872             return string.Join("", str.Split(default(string[]),
                    StringSplitOptions.RemoveEmptyEntries));
873         }
874     }
```