Increased efficiency of Energy Calculation
Using .NET 4.0

Carl Blommé
Abstract

Increased efficiency of Energy Calculation

Carl Blommé

The purpose of this thesis is to investigate if one is able to make the calculations that are performed in the calculation engine, in a program that calculates energy consumption, faster by using a multi-core based architecture. To do this the new multi-core functionalities in .Net 4.0 were examined, in order to determine if they are usable in the current system. A prototype for the calculation engine was made with parallel programming. An evaluation of this code was made and compared to the previous code in order to be able to determine the gain in parallel programming. The comparison was made with tests using the Concurrency Visualizer of Visual Studio 10 and time measurements were taken on running code. The results that were received from parallelizing this program were that one can clearly see that those parts that got parallelized got a significant speed increase. The amount of time saved from parallelizing the code pales however in comparison to the amount of times the calculation engine takes to work with the database.
# Table of Contents

1 Introduction ........................................................................................................... 1

1.5 Reading guide ..................................................................................................... 2

2 Theory .................................................................................................................... 3

2.1 Parallel Computing ............................................................................................ 3

2.2 .Net 4.0 ................................................................................................................ 5

2.2.1 Parallel.For .................................................................................................... 6

2.2.2 Parallel.ForEach ............................................................................................ 8

2.2.3 Parallel.Invoke ............................................................................................. 10

2.2.4 False sharing ................................................................................................ 10

2.2.5 Global variables & results .......................................................................... 11

2.2.6 PLINQ ........................................................................................................... 14

2.2.7 Parallel.ForEach or PLINQ? ....................................................................... 15

2.3 Debug tools ........................................................................................................ 16

2.3.1 Visual Studio windows ............................................................................... 16

2.3.2 Concurrency Visualizer ............................................................................. 17

3 Empirics ................................................................................................................. 19

4 Analysis .................................................................................................................. 22

4.1 Coding ............................................................................................................... 22

4.2 Results ............................................................................................................... 26

5 Conclusion ............................................................................................................. 29

6 References ............................................................................................................. 30

6.1 Books ................................................................................................................. 30

6.2 Articles .............................................................................................................. 30

6.3 Internet .............................................................................................................. 30

7 Appendix .............................................................................................................. 31

7.1 Dictionary ........................................................................................................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Code</td>
<td>32</td>
</tr>
<tr>
<td>7.2.1</td>
<td>rcCalv5_cons.cs</td>
<td>32</td>
</tr>
<tr>
<td>7.2.2</td>
<td>rcCalv5_classes.cs</td>
<td>57</td>
</tr>
</tbody>
</table>
1 Introduction

Momentum Software AB is a company that makes solutions for real-estate companies in order to help them make their daily work more efficient. Momentum delivers systems for leasing, rent administration, maintenance, working orders, surveying, energy calculation, etc.

When a real-estate company gets their measurements for the energy consumption for their different estates they get stored in a database. On these databases different kinds of statistical calculations are made. For example, they might want to know what energy consumption a block has, or how much energy a certain kind of house consumes.

The calculations that are being performed take their data from a database. This database contains a lot of data, which leads to these calculations also taking a long time. The desire is therefore to improve the calculation engine that is being used.

Currently a change is occurring with the architecture of the processors in most computers. Previously, computers had one core and the goal was to make it as fast as possible, which meant increasing the frequency and the amount of transistors. This leads to a large increase in power consumption in the processor which also makes their temperature rise. (Hennessy J., Patterson D., 2002) There has been a change from frequency scaling towards parallel scaling. This means, that instead of only focusing on increasing the clock frequency of the processors, the focus now lies more in the increasing of the amount of cores the processor have. This results in the single cores being slower but instead you have several cores. This will decrease the heat of the processor. The currently existing programs are adapted to the previous architecture of using one core which makes most programs run slower since they run on only one core in the multi-core processors. To be able to take advantage of the new technology, the programs must be built using code that enables parallel computing.

The purpose of this dissertation is to investigate if one is able to make the calculations that are performed in the calculation engine faster by using a multi-core based architecture. To do this the new multi-core functionalities in .Net 4.0 were examined, in order to determine if they are usable in the current system. The functionalities that were examined in .Net 4.0 are the ones that have been added to make parallel programming a viable tool when programming. The existing code can currently only make serial calculations, which means that it only uses one processor at a time. If several processor cores could be used, the calculations that are being made would get considerably faster. It was investigated whether .Net 4.0 is able to be implemented for the current system, and a
prototype was created and measurements its performance was compared to the previously used code.

An analysis of the current code was made in order to determine if and where parallel programming can be applied. A prototype for that calculation engine was made with parallel programming. An evaluation of this code was made and compared to the previous code in order to be able to determine the gain in parallel programming. The new code was not fully integrated with the current system but was evolving continuously to fit the system. The theory was based on articles and web pages about parallel programming and literature about .Net 4.0.

After examining what is needed for code to be parallelized, an analysis of the current code was made in order to determine where parallel programming could be implemented. Based on this, a prototype was constructed and compared to the previous code. The comparison was made with tests using the Concurrency Visualizer of Visual Studio 10 and time measurements were taken on running code.

1.5 Reading guide

Words that are written with *italics* will have an explanation in the dictionary in the Appendix.

When one comes across a place where text is written within a framed area, like the one below, all the text in that area will be code. All the code that is displayed will be written in C#.

Example of how code will be displayed

If references to the code will be made, these will be written in “this font (consolas)”. Colors represent different *methods* or *libraries*. 
2 Theory

2.1 Parallel Computing

Traditionally, code has been written serially for computers. It has been done this way since a computer only has a processor with one core where one task can be done at a time. Now when multicore processors are being used more often and the same code is running on them, it will actually be slower with the newer processors since the code is not adapted to use threads. (Pitzel, 2007)

Even if the code that is used is adapted for multi-core processors, one would think that it would run twice as fast on a dual core processor then on a single core processor. This is not the case though. All processors need to access data. When the processors do this, they have to do trips to the cache, main memory, disk or any other sort of storage that is used, as can be seen in picture 1. This can be a big cost in terms of access time to the CPU. Because of this, many processors now use hyperthreads. Hyperthreading is when each core in the CPU has two sets of hardware threads that can each use the core. They are scheduled to access the core when the other core is using the different memories. By doing this, it will seem like a CPU has twice the amount of cores that it actually has.

![Picture 1: A map of how the CPU communicates with the memory.](image)

There’s almost always parts in the code that cannot be parallelized, which means that the increase in speed will only affect certain parts of the code as can be seen in picture 3.

If one is able to make an algorithm run twice as fast on two cores than on a single core, there is no guarantee that it will run eight times as fast on eight cores. Many algorithms do not scale linearly, this might be because there is not enough parallelism in the algorithm or hardware accessing begins to dominate the running time. (Ostrovsky, 2010)

The amount of speedup a code using multiple processors can achieve is thereby restricted by the sequential parts of the code. Amdahl’s law states that amount of speedup that can be achieved are defined by:

\[
\frac{1}{(1 - P) + \frac{P}{N}}
\]

Where P is the percentage of the code that can be parallelized and N is the amount of processors. This result in the graph below describing the effect multicore processors can have on the speed depending on the percentage of code that can be parallelized. (Amdahl, 1967)

The dependencies in the code put restrictions on how the code will be able to get parallelized. Take the following example:

We have the function \( f(a, b) \).

Within that function two calculations are being made:
\[
c = \sum_{i=0}^{a} i \text{ and } d = c \times b.
\]

The two calculations cannot be performed at the same time since \( c \) is needed for the second calculation to be able to be executed. If dependencies exist and there is no way to bypass them you need to speed up the separate processes instead. The first calculation could be parallelized since it basically is a loop. The second part is only one calculation so there is no speedup to be made there. So by parallelizing this program you would get the effect shown below.

![Picture 3: A graph over the speedup gained when several cores are used depending on how much of the program that is run in parallel.](image)

Thus, by parallelizing functions it is not necessary that the increase in speed will be big. It all comes down to dependencies and the amount of the code that you are able to parallelize.

As with all threaded codes, there are several problems that can occur. Examples of these problems can be mutual exclusion and the race condition. Mutual exclusion can be described as two threads waiting for each other. Thread A holding the variable \( a \) while waiting to get the variable \( b \) all the while thread \( B \) is holding the variable \( b \) and is waiting to be able to use the variable \( a \). The race condition can occur when threads write and read the same variable in the thread. When this is the
case, one thread $A$ is writing to the variable $c$ while thread $B$ is reading from the same variable. This may produce incorrect data in these calculations. Because of these problems one has to be very careful while writing and reading to shared variables in a multi-threaded environment. If one, in any case, needs to do this, a lock can be a useful tool to ensure that a variable is not used in several threads. This will however make the program run slower since the threads will wait until the lock is removed, hence it is best to avoid it if possible. (wiki, parallel computing)

When working with multi-threading, it is important to observe that thread-safety is guaranteed. So if one is working in a multi-threaded environment, one has to make sure that methods are called in a thread-safe manner. If they are not, they need to be protected with locks or rewritten. (Ostrovsky, 2010)

For a deadlock to occur, four conditions have to be fulfilled. These four conditions are mutual exclusion, hold and wait, no preemption and circular wait. If any of these conditions does not hold, a deadlock is not possible. Thus, in order to ensure that a deadlock will not occur one has to avoid at least one of these conditions. (Toub, 2010)

### 2.2 .Net 4.0

Why does one need to adapt a language to parallel programming in particular? Why not simply use lots of threads instead? This could be a solution to making a program parallel, but creating threads is a resource-intensive process, it might not be the fastest or the most efficient way to complete a task. If one creates a lot of threads, it can actually make the programs run slower, because the threads never get the time to get completed since the operating system quickly changes between them. In order to solve this there has been a thread pool implemented in .Net and the use of this thread pool has been extended in .Net 4. (Toub, 2010)

The idea behind a thread pool is to create a number of threads at process start-up and place them into a “pool”, where they sit and wait for work. When a request is received it awakens a thread from this pool, if a thread is available. Once the thread has completed its service it returns to the pool and awaits more work. (Silberschatz et al, 2005)

Creating and tearing down threads is also a relatively costly action to perform, hence it shall preferably be avoided if possible. Instead of creating and tearing down threads the thread pool in .Net 4 allows threads to return to the pool instead of tearing them down when they are done with their task. This makes working with threads much more efficient. (Toub, 2010)

In concurrent programming, independent operations may be carried out at the same time using threading. When using parallel programming, however, there is a need for the operation to be divided into sub-operations. This is called partitioning. (Toub, 2010)
Because of this, there will be an overhead when using parallel loops. This can result in that it is not worth making simple loops parallel, since the overhead might be bigger than the actual time it took to go through the loop before the parallelizing. (Ostrovsky, 2010) This depends on the complexity of the loop and the amount of iterations that are supposed to be done. In many cases, it will be hard to beforehand know if the loop will be faster when parallelized, so the only way to know is by testing to parallelize the loop and comparing the speed to the previous version of the code. (Toub, 2010)

One thing to be aware of when programming in parallel, is that it is almost impossible to make a whole program run in parallel. Most often it is just parts of the program that can be made parallel. Thus, when one instinctively thinks that one can double or quadruple the speed of a program because one has several cores, this is incorrect. One can, however, double or quadruple the parts that one can convert into parallel code. Hopefully these parts are a big portion of the code. (Toub, 2010)

A big part of the work that is done in applications and algorithms are done by different kinds of loops. Loops are used because it allows the application to execute the same set of instructions over and over. for and foreach are two kinds of loops that are often used. (Toub, 2010) Parallel.For and Parallel.ForEach are conceptually similar to for and foreach and are often the easiest way to take advantage of multi-core processors (Ostrovsky, 2010). These kinds of loops are in many cases very easy to convert into parallel loops (Toub, 2010).

### 2.2.1 Parallel.For

#### Code 1

```csharp
Parallel.For(int from, int to, Action<n> body);
```

The `Parallel.For` method basically accepts three parameters, a lower bound, an upper bound and a delegate to be invoked for each iteration. What one sees in code 1, is the basic parallel loop, but it can be configured for more advanced uses. By applying different parallel options that allows one to use thread-local data in which iterations may run in parallel, loop options can be configured, and the state of the loop can be monitored and manipulated. (Microsoft Corporation, 2010a) By default the `Parallel.For` loop uses as much parallelism it can muster from the thread pool and it invokes the provided body for each iteration. `Parallel.For` also provides more than this:

- **Exception handling**
  
  If one iteration of the loop that is running throws an exception, all of the threads that are participating in the loop will attempt to stop processing as soon as possible.
• **Breaking out of a loop early**
  To stop loops early, you can use functions such as `break` and `stop` to make the loop stop prematurely. Depending on which kind of stop operator one uses, one will get different effects. `ParallelLoopResult` is used to get a return value which can inform if and why a loop was stopped before it was finished.

• **Long ranges**
  Overload support works with both Int32- and Int64-based ranges.

• **Thread-local state**
  Several overloads provide support for thread-local state. This means that the result that each iteration returns can be used in calculations.

• **Configuration options**
  Loop executions can be controlled in multiple ways, for example setting a limit to the number of thread or the number of cores that are allowed to be used by the loop.

• **Nested parallelism**
  The ability to use a `Parallel.For` loop within a `Parallel.For` loop is implemented. This will work since they are coordinated to share the threading resources. The ability to use `Parallel.For` loops concurrently is also implemented.

• **Dynamic thread counts**
  Rather than being statically set, the amount of processors used for a loop is dynamically adjusted in relation to the workload that is straining the computer. This will make the threads involved change over time.

• **Efficient load balancing**
  When it comes to load balancing `Parallel.For` takes a large variety of potential workloads into account. It tries to maximize the efficiency and minimize the overhead. The partitioning that is working in the loop is creating chunks that the different threads work on. The amount of chunks that are created depends on how many iterations that are going to be done. In addition to this, it tries to ensure that most iterations from a thread is focused in the same region in order to provide high cache locality. (Toub, 2010)

Below you can see the difference between a regular `for` loop and a `Parallel.For` loop:
The basic difference between `for` and `Parallel.For` is not that big when the most basic parallel loop is used. The difference does not lie in how the outer part is written. The biggest difference lies in the body. (Toub, 2010)

Since the `Parallel.For` loop does things in parallel one has to code differently if one wants something to be ordered. The code below illustrates this:

```csharp
Parallel.For(0, 10, n =>
{
    Console.WriteLine(n);
});
```

If one wants this code to write 0123456789 in the console, one will have to write it differently since when one runs this code it might as well return 0567893412. (Toub, 2010) So when coding, one has to verify that the loop body delegate does not make any assumptions about the order in which the iterations will be executed (Ostrovsky, 2010).

If one wants to be sure that something is written in order, one has to use some variable to store the result in, as shown in the code below:

```csharp
int[] returnVals = new int[10];
Parallel.For(0, 10, n =>
{
    returnVals[n] = n;
});
```

Now one can be certain that what is stored at the different positions in `returnVals` is the desired result. Since `n` is unique for every loop, the different loops will not try to write to the same place in the memory which otherwise could cause this loop to crash. (Toub, 2010)

### 2.2.2 Parallel.ForEach

When `for` is used as a loop it iterates through numbers that represents a range. If you want to use a more general concept there is the alternative to use `foreach` instead of `for` (Toub, 2010). `Parallel.ForEach` is similar to a `foreach` loop that it iterates over an enumerable data set, but
unlike `foreach`, `Parallel.ForEach` uses multiple threads to evaluate the different invocations of the loop body. These characteristics make `Parallel.ForEach` a broadly useful mechanism for data-parallel programming (Vagata, 2009). Much more complicated iteration patterns can be achieved by using enumerable and one has the ability to iterate through any enumerable data set. `Parallel.ForEach` that is used in parallel programming includes many of the overloads `Parallel.ForEach` provides support for including breaking out of loops early, thread count dynamics and advanced partitioning. (Toub, 2010)

Code 5

```csharp
Parallel.ForEach(Enumerable.Range(lowerBound, upperBound), i => { /* ...body */ });
```

This `Parallel.ForEach` loop is written to work as a regular `Parallel.For` loop. The enumerable will be looped through and in this case it contains numbers from `lowerBound` to `upperBound`. `Parallel.ForEach` is optimized to use when working on data sources that can be indexed, such as lists and arrays. `Parallel.ForEach` works best there because when using indexes the need for locking is decreased. (Toub, 2010)

Since coding in parallel does not automatically ensure that the code is thread-safe, there are a number of things that one need to think of when coding or converting existing code into parallel. The iterations that are done within the loop must be independent. If they are not, one has to make sure that the iterations are safe to execute concurrently. Consider the examples below.

Code 6

```csharp
for (int i = upperBound; i >= 0; i--) { /* ...Body */ }
```

Code 7

```csharp
for (int i = 0; i < upperBound ; i += 3) { /* ...Body */ }
```

Usually when loops are written with downward iteration the loop most often have dependencies. Otherwise there would be no reason to write it that way if both upward and downward iterations would work.

In the code 7 case there is an iteration that has steps bigger then 1. `Parallel.For` does not support this kind of patterns, instead the code has to be written with `Parallel.ForEach` iterating through a list, as shown in code 8.

Code 8

```csharp
int[] list = [0,3,6...,upperBound];
Parallel.ForEach(list, i => { /* ...Body */ });
```

(Toub, 2010)
2.2.3 Parallel.Invoke

When a developer has multiple functions that are independent of each other and are able to be run concurrently, `Parallel.Invoke` can be used. `Parallel.Invoke` provides the same support as `Parallel.ForEach` and `Parallel.ForEach` when it comes to exception handling, synchronization, invocation, scheduling etc. `Parallel.Invoke` is meant to reduce the overhead one gets when using `Parallel.ForEach`. If one only has a few tasks that one wants to run, `Parallel.Invoke` is the better choice. However, at a certain threshold `Parallel.ForEach` becomes more effective to use. In code 9 and 10 one can see examples of how `Parallel.Invoke` is used.

Code 9

```csharp
Parallel.Invoke(
    () => FirstAction(),
    () => SecondAction(),
    ...,
    () => LastAction());
```

Code 10

```csharp
Action[] actionList = {FirstAction(), SecondAction(), ..., LastAction()};
Parallel.Invoke(actionList);
```

As with the previous parallel functions, `Parallel.Invoke` will only be considered complete when all the actions are completed. (Toub, 2010)

2.2.4 False sharing

Even if the code that is written has no errors in it, there are still errors that can occur on hardware level when parallel programming is used. These errors can result in loss of performance. Memory systems use cache lines that are moved around the system as a chunk instead of individual bytes. If several cores are trying to access different bytes in the same cache line there is no sharing conflict but only one will be able to access the line at a time. This results in the equivalent of a lock on hardware level that one might not be able to spot in the code. Below is an example of this using `Parallel.Invoke` and a solution to that specific problem.
Random firstRandom = new Random(), secondRandom = new Random();
int[] firstResult = new int[1000000], secondResult = new int[1000000];
Parallel.Invoke(
    () =>
    {
        for (int i = 0; i < firstResult.Length; i++)
            firstResult[i] = firstRandom.Next();
    },
    () =>
    {
        for (int i = 0; i < secondResult.Length; i++)
            secondResult[i] = secondRandom.Next();
    });

These two random instances will be likely to be located on the same cache line in the memory. One way to solve this is to write it like this:

int[] firstResult, secondResult;
Parallel.Invoke(
    () =>
    {
        Random firstRandom = new Random();
        firstResult = new int[1000000];
        for (int i = 0; i < firstResult.Length; i++)
            firstResult[i] = firstRandom.Next();
    },
    () =>
    {
        Random secondRandom = new Random();
        secondResult = new int[1000000];
        for (int i = 0; i < secondResult.Length; i++)
            secondResult[i] = secondRandom.Next();
    });

When written like this the random instances will most likely be allocated at different cache lines. (Toub, 2010)

2.2.5 Global variables & results

When an output is desired from a loop a variable is often needed to be defined beforehand. In regular sequential code it is easy to do this without risking locks.
To write this code in parallel would, however, not work since the different threads might want to add to the same place in output. However, if the size of output is known there will be no problem making the loop run in parallel.

This loop could be written as this in parallel:

If the amount of computation done in SomeCalculation is significant then the cost of the lock is likely to be negligible. As the size of the job executed in SomeCalculation decreases the overhead of taking a lock becomes more relevant and the block will cause more threads to wait to be able to acquire output. Because of this, new thread-safe collections have been introduced in .Net 4.0. These collections exist in the System.Collections.Concurrent namespace and can be used in a multi-core environment. (Toub, 2010)
**ConcurrentQueue**

ConcurrentQueue is a data structure that provides thread-safe access to FIFO ordered elements. When working with light computations ConcurrentQueue is optimal on two threads: one pure producer and one pure consumer. Queues will not scale well beyond two threads when working with light computations. For scenarios when working with moderate-size work functions or when working with a mixed producer-consumer scenario, one will get a lot better scalability while using ConcurrentQueue then using a regular Queue with locking. (Song et al., 2010)

**ConcurrentStack**

ConcurrentStack is an implementation of the LIFO data structure that provides thread-safe access without the need for external synchronization. In a pure producer scenario ConcurrentStack scales as well as a regular Stack with locking. When dealing with a mixed producer-consumer relationship ConcurrentStack scales a lot better. (Song et al., 2010)

**ConcurrentBag**

The ConcurrentBag does not have a counterpart in the previous .Net frameworks. Items can be added and removed as with the previous collections, but the difference is that they are not kept in a certain order. Because of this one might as well use any of the collections above if one does not care about the order of the data contained in the variable. However, the collections above need some ordering rules and more synchronization which could decrease scalability which is one thing ConcurrentBag does not need. (Song et al., 2010) All the synchronization of the output data structure is handled internally by the ConcurrentBag. (Toub, 2010)

**Code 17**

```csharp
ConcurrentBag<int> output = new ConcurrentBag<int>;
Parallel.ForEach(input, item =>
    {
        int result = SomeCalculation(item);
        output.Add(result);
    });
```

In a mixed producer-consumer environment ConcurrentBag will be superior ConcurrentStack and ConcurrentQueue when order is not needed as is shown in the graph below. (Song et al., 2010)
ConcurrentDictionary

The ConcurrentDictionary is a thread-safe version of a Dictionary. The ConcurrentDictionary is viable to use in a producer-consumer environment, however in a pure consumer scenario a regular Dictionary will run faster since it has lower overheads. If one has a red-heavy scenario with occasional updates that requires completely thread-safe operations, the ConcurrentDictionary is superior. (Song et al., 2010)

2.2.6 PLINQ

PLINQ is the parallel version of LINQ. LINQ is a set of extensions to the .NET Framework that encompass language-integrated query, set, and transform operations (Microsoft Corporation, 2010b). When converting a LINQ query into a PLINQ query the result might not come in the expected order. Just as with the Parallel.For the order of the outcome will be mixed if one imports a set of numbers with PLINQ. (Tan, 2010)

You can see an example of this below.

Code 18

```csharp
var query = Enumerable.Range(0, 10).AsParallel().Select(x => x);
foreach (var item in query)
{
    Console.Write("{0}", item);
}
Console.WriteLine();
```

So this might as well return 0567893412 like the Parallel.For that we mentioned previously.

When using a very simple solution, like the one below, this will be solved. (Tan, 2010)

Code 19

```csharp
var query = Enumerable.Range(0, 10).AsParallel().AsOrdered().Select(x => x);
foreach (var item in query)
{
    Console.Write("{0}", item);
}
```

Picture 4: A comparison of the speed between the different concurrent variables.
An alternative to `AsOrdered` is to use `OrderBy` (Ostrovsky, 2010). This does however only guarantee an ordered output. It does not mean that the order of execution on the delegates is in order (Tan, 2010). `PLINQ` should be used to express computations with an expensive operation applied over a sequence. The queries should be kept simple so that it is easy to understand them. If possible try to break up complex queries so that the cheap but complex part is done externally to `PLINQ`, an example is showed below.

```csharp
var query = Enumerable.Range(0, 100).TakeWhile(x => SomeFunction(x)).AsParallel().Select(x => SomeOtherFunction(x));
foreach (var x in query) Console.WriteLine(x);
```

Here `AsParallel` is placed after the `TakeWhile` operator instead of immediately at the data source. (Ostrovsky, 2010)

### 2.2.7 Parallel.ForEach or PLINQ?

Both `PLINQ` and `Parallel.ForEach` can be used to specify the number of threads that one wants to use in the execution. The difference between these is that the `Parallel.ForEach` is more dynamic. `Parallel.ForEach` uses `ParallelOptions.MaxDegreeOfParallelism` which specifies that at most N threads are needed. Because of this, the number of threads used can be less than N if resources are scares and when more resources become available it will adjust to this. `PLINQ` on the other hand uses `WithDegreeOfParallelism` which specifies exact N number of threads. (Vagata, 2009)

If order preservation is desired it will probably be easier to achieve this by using `PLINQ`. Using the `AsOrdered` operator in `PLINQ` will automatically handle the order preservation. When it comes to order preservation using `Parallel.ForEach` you need to use a predefined `array` as we showed earlier. If the collection used were an `Enumerable` instead of an `array` there would be four ways to implement order preservation.

- The first alternative would be to use `Enumerable.Count` which iterates over the entire collections and returns the number of elements. Then you could allocate the `array` before calling `Parallel.ForEach` and then insert each element in the right position in the `array`.
- The second alternative would be to materialize the original collection before using it.

The first two alternatives would not be good to use if the input data is large.

- The third alternative is to create a hashing data structure as output collection. This would however need to be twice the size of the input data which could drive the performance down if it gets too big.
- The last alternative would be to store the result with the original in data and then create your
own sorting algorithm to use on the output data.

When using *PLINQ* you need simply to ask for order preservation which makes it much easier to use in these cases. (Vagata, 2009)

### 2.3 Debug tools

#### 2.3.1 Visual Studio windows

To debug your code while stepping through it there are three different tools that can be useful. They all have to do with keeping control over the different threads used. These windows can be found in **Debug → Windows** in Visual Studio.

In the **Parallel Tasks** window one can keep track of if threads are blocking each other or are kept waiting for some other task to finish. This is good to use in combination with the **Threads** window. The **Threads** window keep track of where in the code the different threads are executing currently, so if you see that a thread is in deadlock in the **Parallel Tasks** window you can switch to the **Threads** window and quickly found out at what place in the code the deadlock occurs. The **Parallel Tasks** window can also be used to freeze threads or just run single threads to determine the behavior in certain cases.

![Parallel Tasks window](image)

**Picture 5:** The parallel task window showing the state of different threads

![Threads window](image)

**Picture 6:** In the task window you are able to control the threads and see at what place they are at in the code.
2.3.2 Concurrency Visualizer

To be able to track how the program that you are coding is performing, and how the threads are behaving, a tool called Concurrency Visualizer has been created for Visual Studio 2010. The Concurrency Visualizer is only available in the premium and ultimate edition of VS 2010. If you are coding in a multi-threaded environment this can be a very powerful tool. (George, Nagpal, 2010)

One thing to remember when using the Concurrency Visualizer is that it will make the programs run slower because it collects a lot of data while the program is running. Thus, if you want an accurate number of how fast the code is, a timer should be added in the code instead. It is however very useful to use the Concurrency Visualizer in order to compare if the code has improved or not. The Concurrency Visualizer consists of three different tools.

**CPU Utilization**

The CPU Utilization tool provides a snapshot of the logical CPU core utilization during execution of the application that you are creating a profile of.

![CPU Utilization](image)

*Picture 7: An example of the CPU utilization in the Concurrency Visualizer.*

The green part represents the application that you are profiling. (George, Nagpal, 2010)

The CPU Utilization tool is a more detailed version of the *Windows Task Manager* (Ostrovsky, 2010).

**Threads**

To find out more detailed information about the threads and how they are working or why they are in their current state, one can use the Threads view. (Ostrovsky, 2010)
The last part of the Concurrency Visualizer is the core view. It displays which threads that got executed on the different cores and at what time. (George, Nagpal, 2010) The core view can help identify issues with thread migration (Ostrovsky, 2010).

Context switches that also cross from one logical core to another can reduce the performance of your process.

Picture 8: An example of the threads view in the Concurrency Visualizer.

Picture 9: An example of the core view in the Concurrency Visualizer.
3 Empirics

The program that is being investigated for possible speed increases is built to do statistical analyses of different measurements. These measurements come from electricity meters that for example are placed at apartment. The meters are sorted into a tree-like structure shown in picture 10.

![The tree structure of the electricity meters.](image)

Here $A$ represents a block, $B$ and $C$ houses, $D$, $E$, $F$ and $G$ floors and $H$, $I$ and $J$ apartments. If the measurements from $A$ are desired for a certain period of time the calculation engine will go downwards in the tree to the bottom meters and summarize the measurements for that period. It is not certain that the measurements will be for every day. In that case the measurement that is obtained will be divided into days and then used in the calculation.

All the measurements are stored in a database and the calculations that are desired to be done are stored in a queue in the database. The calculation engine fetches the top prioritized from this queue and then starts to process it.

If the assignment that is fetched wants to do a calculation of consumption over a given period of time a certain function is entered. In this function the calculations for consumptions are done over each day.
For (int n = 0; n <= numOfDays; n++)
{
    DateTime startTime = DateTime.Now;
    DateTime dayDate = startDate.AddDays(n);
    Console.WriteLine("{0,4} Calculate clientId={1}, counterId={2}, dayDate='{3:d}'", n, clientId, counterId, dayDate);
    If (!m_deletedCounters.Contains(counterId))
        m_deletedCounters.Add(counterId);
    _calcCons(clientId, cgId, dayDate, touchedCounters);
}

When you enter _calcCons every meter, also called counter, that is used in the calculations for the desired counter will be entered into a list and calculated for this specific day. The calculations of the consumption for each counter are done in the loop below.

foreach (int counterId in o_cgld.CounterIds.Values)
{
    double counterConstant =
    m_collCounters.GetEntityWithId(counterId).Constant;
    var q1 = from cdv in m_collCounterDailyConsumption
             where (cdv.counterId == counterId && cdv.dayDate == dayDate)
             select cdv;
    foreach (CounterDayValue cdv in q1)
    {
        cgcons += (cdv.dayValue * counterConstant);
        CounterWithFacts c = new CounterWithFacts(clientid, counterId);
        c.Cons = (cdv.dayValue * counterConstant);
        counters.Add(c.CounterId, c);
    }
}

These are small selected parts from the given code. The full code can be seen in the appendix.

The program keeps track of all the counters that have been used and for what date their consumption has been calculated. This is to keep track of what is supposed to be updated in the database.

An analysis was made with the Concurrent Visualizer in order to determine how fast the current code was running. This calculation was done on a small span with a single meter.

From picture 10 it is easy to see that most of the work being done is spent procedures and writing to different segments within the database containing information of the power consumption. This since the code that is running for calculations is in idle for most of the time, which is represented by the grey area. The focus will lie on the area within the red rectangle in picture 11, which is zoomed in picture 12.
Picture 11: An analysis made with the Concurrency Visualizer showing the CPU Utilization of the program. In this run the program was subjected to a low load. The red arrow shows the area where the calculations are being made.

Picture 12: A zoomed in view of the red rectangle showed in picture 11.
Since this is the area where the calculations are done it will be the part focused on in the coming visual representations. Picture 13 displays the amount of work and time needed when the program is subjected to a high load. This is achieved by using counters that are high up in the tree structure, like A in picture 10, over a longer period of time.

### 4 Analysis

#### 4.1 Coding

When the code is analyzed thoroughly, some places where the possibility of implementing parallel programing are found. At first glance the easiest part to parallelize of the program would be the small loop that does the computations. This seems suitable because it is only simple operations that are made within the loop and there is no editing of shared variables. The easiest way would be to just change the foreach loops into Parallel.ForEach loops. PLINQ could also be used but since order preservation is not needed Parallel.ForEach will be used instead. Trials were made where both loops were parallel or just one of them.
Parallel.ForEach (o_cgid.CounterIds.Values, counterId =>
{
    double counterConstant =
    m_collCounters.GetEntityWithId(counterId).Constant;
    var q1 = from cdv in m_collCounterDailyConsumption
    where (cdv.counterId == counterId && cdv.dayDate == dayDate)
    select cdv;
    Parallel.ForEach (q1, CounterDayValue =>
    {
        cgcons += (cdv.dayValue * counterConstant);
        CounterWithFacts c = new CounterWithFacts(clientid, counterId);
        c.Cons = (cdv.dayValue * counterConstant);
        counters.Add(c.CounterId, c);
    });
});

Results were measured using a Stopwatch like below.

Stopwatch sw = new Stopwatch();
sw.Start();
// What you want to measure..
Console.WriteLine("{0}", sw.ElapsedMilliseconds);

Below you see the results of the measurements that were received. These results are from how long
each day took to execute in a certain period. The same day and period was used for both the serial
and the parallel runs.

<table>
<thead>
<tr>
<th>Day</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>2,0209</td>
<td>0,2271</td>
<td>0,2225</td>
<td>0,2137</td>
<td>0,2332</td>
<td>0,2122</td>
<td>0,2107</td>
<td>0,2148</td>
<td>0,6957</td>
<td>0,2573</td>
</tr>
<tr>
<td>Parallel</td>
<td>39,6449</td>
<td>0,3834</td>
<td>0,3076</td>
<td>0,2491</td>
<td>0,3147</td>
<td>0,2814</td>
<td>0,2794</td>
<td>0,2568</td>
<td>0,2640</td>
<td>0,2737</td>
</tr>
</tbody>
</table>

As you can see above the overhead for the first iteration in the parallel run is very high and every
iteration runs slower than the serial run. This might be because the calculations done within the loop
are so fast it is not worth parallelizing this loop. To determine whether this was the case or not a try
was made with a higher load. A Thread.Sleep for one millisecond was added in inner loop of the
calculations to serve as an artificial calculation. The result from this is shown below.

<table>
<thead>
<tr>
<th>Day</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>8,4792</td>
<td>10,5529</td>
<td>6,6038</td>
<td>6,7227</td>
<td>6,5156</td>
<td>6,4489</td>
<td>6,8893</td>
<td>9,9182</td>
<td>7,0708</td>
<td>6,416</td>
</tr>
<tr>
<td>Parallel</td>
<td>45,0491</td>
<td>2,5567</td>
<td>2,3634</td>
<td>2,5659</td>
<td>2,3342</td>
<td>3,1340</td>
<td>1,8061</td>
<td>1,6600</td>
<td>2,3952</td>
<td>1,6264</td>
</tr>
</tbody>
</table>

The overhead is still very high for the parallel run, but each day is running a lot faster. Because of
this, we can make the conclusion that this loop is not worth parallelizing, since the load is not big
enough and the amount of iterations done for each day in this loop is not so many that a need for
parallelizing exists. Because of this, focus was moved to the outer big loop. This would have a bigger load but it would bring some complications with it too. The main thing wanted when parallelizing code is that the iterations act independently from each other. This was not the case with the bigger loop which was separated into one day for each iteration. It contained public variables that were added and removed from which could cause problems. If this program would have been programmed from the beginning with the intent of using parallel programming, it might have looked very different. The easiest solution now would be to adjust the current code and hope there would not be too many complications.

An analysis of the code within the loop is being made to find places where possible problems can occur when parallelizing. Calls to any shared resource have to be changed in some way, for example this debug call.

**Code 25**

```csharp
Debug.WriteLine(String.Format("treeDepth={0}, clientId={1}, cgId={2},
dayDate='{3}', cgType={4}"), recurseDepth, clientId, cgIdLevelParent, dayDate, mainCG.cgType.ToString());
```

The debug is a shared resource. Writing to the debug creates a lock which makes the different threads wait for each other. All writings to the debug panel will therefore be removed.

When it comes to the adding and removing, a class called `calcConsCollection` is being used. This collection is to keep track of what counters has been changed and update the value that was used in the last calculation. It is possible that the `calcConsCollection` can be used outside this calculation engine and it might be hard to make it work concurrently. Therefore, the decision was made to move the deletion out of the parallel loop instead of trying to make it work concurrently.

**Code 26**

```csharp
foreach (CounterWithFacts c in mainCG.Counters.Values)
{
    if (!touchedCounters.Contains(c.CounterId))
        touchedCounters.Add(c.CounterId);
    collCalcCons_delete(c.ClientId, c.CounterId, mainCG.cgId, dayDate);
    collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate, c.Cons, CALCCONS_TYPE.CONS_DATA);
    collCalcCons_delete(c.ClientId, c.CounterId, mainCG.cgId, dayDate);
    collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate, c.Distributed_Cons, CALCCONS_TYPE.CONS_DISTRIBUTED_DATA);
    collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate, cCorrection_Cons, CALCCONS_TYPE.CONS_CORRECTION_DATA);
}
```

There is still need to keep track of all these counters within the parallel loop. Thus, in order to keep track of this, a global thread-safe variable needs to be used to solve this, since there will be writing to this variable from all of the parallel iterations that are running. Because there is no need for the
data to be in order, a `ConcurrentBag` is going to be used. The data that needs to be stored in this bag is the counter group and the date for the calculation. To do this, a class was created to be used in this situation. This class is only used for the purpose of keeping track of counters that needs to be updated after the calculations for each day is finished.

Code 27

```csharp
class ForDeletion
{
    public CGWithFacts CG { get; set; }
    public DateTime date { get; set; }
}
```

The list that is used for deletion looks like this and is sent as a reference within the parallel loop.

Code 28

```csharp
ConcurrentBag<ForDeletion> deleteList = new ConcurrentBag<ForDeletion>();
```

The loop that we parallelized now looks like this:

Code 29

```csharp
Parallel.For(0, numOfDays, n =>
{
    DateTime dayDate = startDate.AddDays(n);
    if (!m_deletedCounters.Contains(counterId))
        m_deletedCounters.Add(counterId);
    _calcCons(clientId, cgId, dayDate, touchedCounters, ref deleteList);
});
```

The reason for using `Parallel.For` instead of `PLINQ` is that there is no need to get the result in order and it is an easy transition from the previous `for` loop. The amount of cores used will also vary, so it is good that the code is as flexible as possible.

At some places within the calculation loop dictionaries are used and these are shared globally. Because of the use of these dictionaries, a change was made from `Dictionary` into `ConcurrentDictionary` in order to be able to work in a multi-threaded environment.

The deletion and inserting in the list was moved out of the loop and looked like this:
foreach (ForDeletion mainCG in deleteList)
{
    foreach (CounterWithFacts c in mainCG.CG.Counters.Values)
    {
        if (!touchedCounters.Contains(c.CounterId))
touchedCounters.Add(c.CounterId);
collCalcCons_delete(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date);
    }
}

foreach (ForDeletion mainCG in deleteList)
{
    foreach (CounterWithFacts c in mainCG.CG.Counters.Values)
    {
        collCalcCons_insert(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date, c.Cons, CALCCONS_TYPE.CONS_DATA);
collCalcCons_insert(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date, c.Distributed_Cons, CALCCONS_TYPE.CONS_DISTRIBUTED_DATA);
collCalcCons_insert(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date, c.Correction_Cons, CALCCONS_TYPE.CONS_CORRECTION_DATA);
    }
}

One may question why the part in the red rectangle is there; this is something that is quite puzzling. When runs were executed with the deletion and insertions in the same loop it took several times longer. A lot of time is saved by separating them. This, however, only happens when the code has been run in parallel; without parallelizing it there is no need to separate them. There has been speculation between myself and my supervisor if this is because the memory allocation is performed differently when parallel programming. It should, however, not be the case since trials were made where a conversion from ConcurrentBag to an array were made before the deletion started, in order to ensure that the data were reserved at the same place. This did, however, not affect the outcome, hence this remains an unsolved problem.

4.2 Results

Using the Concurrency Visualizer we can do test runs and examine the outcome of our parallelizing for the code. Some measurements were also made using a Stopwatch, which is a method used to accurately measure elapsed time. It was, however, only done on the calculation part where the parallelizing had been done, not including the deletions. This is to show the improvement of parallelizing code. The run was made with a dual core processor and it shows that the part that got parallelized got a speed improvement that is about twice as fast as without parallelizing, as is shown in picture 14.
When it comes to the Concurrency Visualizer we look at a bigger picture. A processor with 4 cores was used for these runs. The run that was done in serial can be seen in picture 15.

Notice that the Y-axis represents the virtual amount of cores and says 8 since hyper threading is used. The results that were obtained when running the code in parallel with the same configurations can be seen in picture 16.

**Picture 14:** A graph over the speed of a parallelized run compared to a serial run. The X-axis represents the number of days of the current execution and the Y-axis represents how long each of these executions took.

**Picture 15:** A CPU utilization of the program running with a high load.
The improvement speed wise is about 4 times as fast, so it seems that the parallel part scales very well with several cores. You can clearly see that up to 7 logical cores are used in the calculation process. To check if there is anything that causes threads to wait for each other we study the visualization of the threads.

![Thread View](image)

**Picture 16:** CPU Utilization running in parallel with a high load.

**Picture 17:** A thread view of a parallel run. Within the rectangle is the parallel part of the program. Red = Synchronization, Green = Execution, Yellow = Preemption, Blue = Sleep.

If there would be a case where threads were waiting for each other it would be shown as synchronization which is marked in red. In this run there is no synchronization while running in parallel.
5 Conclusion

When looking at the results that were received from parallelizing this program one can clearly see that those parts that got parallelized got a significant speed increase. The scaling depending on the amount of cores is very good and gives a good indication on how much difference parallel programming can do. The scaling that was achieved when a heavy load was used became the previous speed, times the amount of cores the processor had, not counting the hyper threading. As the amount of cores increases, the speed increases will decline, due to Amdahl's law.

It is a waste not to take advantage of the advanced technology that resides in the newer computers. If serial code is just going to be used on a server then it makes more sense in using a single core processor to obtain as good performance as possible.

When it comes to the program at hand, parallelizing code might not be the most important speed increase that can be made. The amount of time saved from parallelizing the code pales in comparison to the amount of times it takes to work with the database, can be seen in picture 11. If possible, this is where the attention should be focused if a big gain in speed is desired.

I do, however, recommend using parallel programming when coding since it is not as complicated as it sounds when you first hear about it and it is not that hard to get used to. Generally, the codes gets structured better when you have to use parallel programming since you have to think a lot beforehand regarding how the code needs to be built. This is because using global variables as a solution is a quick fix, but not the most effective one for example.
6 References

6.1 Books

Amdahl G. (1967), Validity of the single processor approach to achieving large scale computing capabilities, International Business Machines Corporation, Sunnyvale, California

6.2 Articles

George B., Nagpal P. (2010), Optimizing Parallel Applications Using Concurrency Visualizer: A Case Study
Ostrovsky I. (2010), Parallel Programming in .NET 4
Patrick Tan R. (2010), PLINQ’s ordering model
Song C., Omara E., Liddell M. (2010), Thread-safe Collections in .Net Framework 4 and Their Performance Characteristics
Toub S. (2010), Patterns Of Parallel Programming
Vagata P. (2009), When Should I Use Parallel.ForEach? When Should I Use PLINQ?

6.3 Internet

Steve Pitzel (2007), The Multi-Core Dilemma – By Patrick Leonard [www].
Retrieved 2010-07-21
7 Appendix

7.1 Dictionary

- **Array**: An array is a data type that is meant to describe a collection of elements.
- **Body**: The main part of the code.
- **Calculation engine**: This is the C# code that uses the data obtained from the database to do different operations on it.
- **Circular wait**: There is a set of \{T_1, \ldots, T_N\} threads, where \(T_1\) is waiting for a resource held by \(T_2\), \(T_2\) is waiting for a resource held by \(T_3\), and so forth, up through \(T_N\) waiting for a resource held by \(T_1\).
- **Deadlock**: A deadlock is a situation where two or more competing tasks are each waiting for the other to finish, and thus neither ever does.
- **FIFO**: FIFO is short for First In, First Out which describes a principle of a queue processing technique.
- **Hold and wait**: A thread holding a resource may request access to other resources and wait until it gets them.
- **Hyper threading**: Hyper-threading works by duplicating certain sections of the processor. This allows the processor to appear as two "logical" processors to the host operating system. When a task for example makes trips to the memory hyper threading will allow another task to take advantage of the processor.
- **High load**: High load is when the calculation engine is subjected so a process intensive operation.
- **LIFO**: LIFO is short for Last In, First Out which describes a principle of a queue processing technique.
- **LINQ**: LINQ is short for Language Integrated Query. LINQ defines a set of method names, along with translation rules from query expressions similar to the ones used in SQL.
- **Mutual exclusion**: Only a limited number of threads may utilize a resource concurrently.
- **No preemption**: Resources are released only voluntarily by the thread holding the resource.
- **PLINQ**: PLINQ is a parallel use of LINQ.
- **Race condition**: Race condition is a flaw in an electronic system or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events.
- **Thread migration**: Thread migration is when processes that are run on a computer migrate between different cores, or when processes migrate between different computers in a cluster.
- **Windows Task Manager**: Windows Task Manager provides detailed information about computer performance and running applications, processes and CPU usage, commit charge and memory information, network activity and statistics.

### 7.2 Code

#### 7.2.1 rcCalcv5_cons.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Timers;
using System.Threading;
using System.Data.SqlClient;
using System.Data;
using System.Linq;
using Momentum.RC.EntityDataReader;
using System.Diagnostics;
using Momentum.RC.BusinessEntities.CounterObjects;
using Momentum.RC.BusinessEntities.CounterObjects.DivideObjects;
using Momentum.RC.BusinessEntities.TariffClientObjects;
using Momentum.RC.BusinessEntities.TaxesObjects;
using Momentum.RC.BusinessEntities.Calculate;
using Momentum.RC.BusinessEntities.CalculationObjects;
using Momentum.RC.Bll;
using Momentum.RC.Bll.CounterObjects;
using Momentum.RC.Bll.CounterObjects.DivideObjects;
using Momentum.RC.Bll.Calculate;
using Momentum.RC.Bll.CalculationObjects;
using Momentum.RC.Bll.FactObjects;

namespace Momentum.RC.CalculationEngine.algoEval.rcCalc_version5
{
    public partial class rcCalcv5
    {
        private enum CALCCONS_TYPE
        {
            CONS_DATA = 1,
            CONS_DISTRIBUTED_DATA = 2,
            CONS_CORRECTION_DATA = 3
        }

        private int _wait_for_db_mutex_milliseconds = 300000;
        private string _connStr = null;
        private string _dbConnstrName = null;

        private int recurseDepth = 0;
        DateTime _lastCalcTimeStamp;

        // <cgid, <date, <cgid, calcinfo> >
```
Dictionary<int, Dictionary<DateTime, Dictionary<int, config_entry>>> m_dicCGs_DailyConfigurationTree = null;

CounterGroupCollection m_collCGs = null;
CounterCollection m_collCounters = null;

// <cgid, cgid>
Dictionary<int, int> m_cg_in_maintree_path = null;

// <counterid, counterid>
Dictionary<int, int> m_dicCounterDailyConsumption_Processed = null;

CounterDayValueCollection m_collCounterDailyConsumption = null;
calcConsCollection m_collCalcCons = null;

Dictionary<int, int> m_dicCGs_UG = null; // <cgId, ugId>

List<int> m_deletedCounters = null;

/// <summary>
/// Initializes a new instance of the <see cref="rcCalcV5"/> class.
/// </summary>
/// <param name="connStr">The connection string</param>
/// <param name="dbConnstrName">Name of the connection string</param>
/// <param name="wait_for_db_mutex_milliseconds">The wait time for mutex in milliseconds.</param>
public rcCalcV5(string connStr, string dbConnstrName, int wait_for_db_mutex_milliseconds)
{
    _connStr = connStr;
    _dbConnstrName = dbConnstrName;
    _wait_for_db_mutex_milliseconds = wait_for_db_mutex_milliseconds;
}

/// <summary>
/// Gets the connection string.
/// </summary>
/// <returns>Connection string</returns>
public string getConnStr()
{
    return _connStr;
}

private void fill_counterDailyConsumption(ref RCSession _sessionHolder, int clientId, DateTime startDate, DateTime stopDate)
{
    foreach (BusinessEntities.CounterObjects.Counter c in m_collCounters)
    {
        if (!m_dicCounterDailyConsumption_Processed.ContainsKey(c.counterId))
        {
            // Keep track of all counters that have already been loaded/expanded
            m_dicCounterDailyConsumption_Processed.Add(c.counterId, c.counterId);

            // Load all counterReadings for this counterId
            CounterReadingManager mgrCounterReading = new
            CounterReadingManager(_sessionHolder);
            CounterReadingCollection collCRs = mgrCounterReading.GetList(c.counterId);

            CounterReading crPrevious = null;
            bool pastStopDate = false;
            int i = 0;

            var collSorted = from cr in collCRs
            }
orderby cr.readingDate ascending
select cr;

foreach (CounterReading crCurrent in collSorted)
{
    if (i > 0)
    {
        TimeSpan diff =
        crCurrent.readingDate.Subtract(crPrevious.readingDate);
        int numOfDays = diff.Days;
        double dayValue = 0.0;
        double readingConsumption = 0.0;
        if (c.isConsumptionCounter)
        {
            if (crCurrent.readingWrapAround == 1) throw new
            ApplicationException("wrap around not expected on a consumption counter!");
            readingConsumption = crCurrent.readingValue;
            // dayValue = zeroIfNaN(crCurrent.readingValue / numOfDays);
        }
        else
        {
            // Calculate diff between 2 readings (consumption)
            // if (c.CounterReadingDirection ==
            CounterProperty.CounterReadingDirectionEnum.COUNTERDIRECTION_FORWARD)
            if (c.ReadingDirectionType ==
            CounterReadingDirectionType.Forward)
            {
                if (crCurrent.readingWrapAround == 1) throw new
               reading
                int turnaroundValue = 1;
                double d1Tmp = crPrevious.readingValue;
                while (d1Tmp > 1.0)
                {
                    turnaroundValue *= 10;
                    d1Tmp /= 10;
                }
                readingConsumption = (turnaroundValue -
                crPrevious.readingValue) + crCurrent.readingValue;
            }
            else
            {
                readingConsumption = (crCurrent.readingValue -
                crPrevious.readingValue);
            }
        }
    }
    else
    {
        if (crCurrent.readingWrapAround == 1) throw new
        ApplicationException("wrap around not expected on a backward counter!");
        readingConsumption = (crPrevious.readingValue -
        crCurrent.readingValue);
    }
    // dayValue = zeroIfNaN(counter_reading_diff / numOfDays);
    dayValue = zeroIfNaN(readingConsumption / numOfDays);
// if (c.CounterSign ==
CounterProperty.CounterSignEnum.COUNTERSIGN_NEGATIVE)
if (c.SignType == CounterSignType.Negative)
    dayValue = -dayValue; // change sign

DateTime dayDate = crPrevious.readingDate.AddDays(0);
for (int n = 0; n < numOfDays; n++)
{
    if (dayDate >= startDate && dayDate <= stopDate)
    {
        CounterDayValue cdv = new CounterDayValue();
        cdv.clientId = c.clientId;
        cdv.counterId = c.counterId;
        cdv.dayDate = dayDate;
        cdv.dayValue = dayValue;
        m_collCounterDailyConsumption.Add(cdv);
    }
    else if (dayDate > stopDate)
    {
        pastStopDate = true;
        break;
    }

    dayDate = dayDate.AddDays(1); // next day
}

if (pastStopDate) // vi har gått förbi stopDate - vi behöver inte hämta mer data!
    break;

crPrevious = crCurrent;
i++;
}

private class FactDayConsByDateCollection : List<FactDayConsByDate>
{
}

private class FactDayConsByDate
{
    public FactDayConsByDate(DateTime dt, int tk)
    {
        _dt = dt;
        _tk = tk;
    }

    public DateTime _dt = DateTime.MinValue;
    public int _tk = 0;
    public DependentsFactDayConsCollection collFactsPoint = null; // cons_cgKey, cons_counterKey
}

    /// <summary>
    /// Calculates the counter consumption.
    /// </summary>
    /// <param name="conn">The connection</param>
    /// <param name="_sessionHolder">The _session holder</param>
    /// <param name="clientId">The client id</param>
    /// <param name="counterId">The counter id</param>
public bool CalculateCounterConsumption(SqlConnection conn, ref RCSession _sessionHolder, ref TariffSystem _objTariffSystem, ref TariffClientCollection _collTariffClients,
   int clientId, int counterId, DateTime startDate, DateTime stopDate, List<int> touchedCounters, List<Period> listPeriod)
{
    _lastCalcTimeStamp = DateTime.Now;

    // Hämta anläggning alt. fördelare av typen fristående um för mätaren (counterId)
    // dvs hämta den översta nivån som mätaren kan stjäla förbrukning ifrån
    int cgId = getCGforCounter_DB(conn, clientId, counterId);

    // // om mätaren sitter på en fördelare 'um' (cgIdConnected != -1) så byt till den 'riktiga' anläggning som fördelaren 'um' är kopplad mot.
    if (cgId == -1) throw new ApplicationException("Mätaren saknar koppling till anläggning - kan ej beräknas! (Calculate getCGforCounter => cgId == -1)"));

    // CG configuration on a daily basis
    if (m_dicCGs_DailyConfigurationTree != null)
        m_dicCGs_DailyConfigurationTree.Clear();
    m_dicCGs_DailyConfigurationTree = new Dictionary<int, Dictionary<DateTime, Dictionary<int, config_entry>>>();

    // Dictionary holding CG properties that are not time dependant
    if (m_collCGs != null)
        m_collCGs.Clear();
    m_collCGs = new CounterGroupCollection();
    if (m_cg_in_maintree_path != null)
        m_cg_in_maintree_path.Clear();
    m_cg_in_maintree_path = new Dictionary<int, int>();

    // Dictionary holding Counter properties that are not time dependant
    if (m_collCounters != null)
        m_collCounters.Clear();
    m_collCounters = new CounterCollection();
    if (m_deletedCounters != null)
        m_deletedCounters.Clear();
    m_deletedCounters = new List<int>();

    // Load configuration tree (recursive for entire date span) into memory tables
    // Populates config_tree, cg_mem, algo_mem, counter_mem
    Console.Write("\nLoading configuration tree into memory ... 1\n");
    loadConfigTree_DB(conn, ref _sessionHolder, clientId, cgId, startDate, stopDate);
    Console.WriteLine("complete.");

    if (m_collCounterDailyConsumption != null)
        m_collCounterDailyConsumption.Clear();
    m_collCounterDailyConsumption = new CounterDayValueCollection();

    if (m_dicCounterDailyConsumption_Processed != null)
        m_dicCounterDailyConsumption_Processed.Clear();
    m_dicCounterDailyConsumption_Processed = new Dictionary<int, int>();

    return true;
}
fill_counterDailyConsumption(ref _sessionHolder, clientId, startDate, stopDate);
//</Exjobb>
Console.WriteLine(" complete.");

if (m_collCalcCons != null)
    m_collCalcCons.Clear();
m_collCalcCons = new calcConsCollection();

if (m_dicCGs_UG != null)
    m_dicCGs_UG.Clear();
m_dicCGs_UG = new Dictionary<int, int>();

TimeSpan diff = stopDate.Subtract(startDate);
int numOfDays = diff.Days;

//<Exjobb> The loop that is now run in parallel with the deletion
//moved out so that it is done after.
ConcurrentBag<ForDeletion> deleteList = new ConcurrentBag<ForDeletion>();

//We have to use a concurrent list, otherwise we wont be able to use it in a
//parallel loop.
//All writing to the console/debug has to be removed in all the sections that
//is written in parallel since the console is sharded between the threads and
//will make them wait for eachother.
Stopwatch sw = new Stopwatch();

Parallel.For(0, numOfDays, n =>
    //for(int n=0; n<numOfDays;n++)
    {
        DateTime dayDate = startDate.AddDays(n);

        // Cleanup in factDayCons for the CounterId and dayDate
        // This will delete *all* rows for the counter and day out of the
        FactDayCons table.
        // Data for counterId and dayDate will be deleted no matter which cgId
        // the consumption is distributed to.
        if (!m_deletedCounters.Contains(counterId))
            m_deletedCounters.Add(counterId);

        //deleteList added for the removal loop afterword.
        _calcCons(clientId, cgId, dayDate, touchedCounters, ref deleteList);
    });

foreach (ForDeletion mainCG in deleteList)
{
    foreach (CounterWithFacts c in mainCG.CG.Counters.Values)
    {
        if (!touchedCounters.Contains(c.CounterId))
            touchedCounters.Add(c.CounterId);
        collCalcCons_delete(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date);
    }
}
foreach (ForDeletion mainCG in deleteList)
{
    foreach (CounterWithFacts c in mainCG.CG.Counters.Values)
    {
        collCalcCons_insert(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date, c.Cons, CALCCONS_TYPE CONS_DATA);
        collCalcCons_insert(c.ClientId, c.CounterId, mainCG.CG.cgId, mainCG.date, c.Distributed_Cons, CALCCONS_TYPE CONS DISTRIBUTED_DATA);
collCalcCons_insert(c.ClientId, c.CounterId, mainCG.CG.cgId,
mainCG.date, c.Correction_Cons, CALCCONS_TYPECONS_CORRECTION_DATA);
}
}
//</Exjobb>
Console.WriteLine("\tStoring result in database ... 1 ");
deleteFrom_DBCalcCons(ref _sessionHolder);

Console.WriteLine(" 2 ");
insertInto_DBCalcCons_from_collCalcCons_BULK(ref _sessionHolder, clientId);
Console.WriteLine(" 3 ");
Console.ReadLine();
// Cleanup in FactDayCons for the CounterId and interval
// This will delete *all* rows for the counter and period out of the FactDayCons
table.
// Data for counterId and period will be deleted no matter which cgId the
consumption is distributed to.
foreach (int cntrId in m_deletedCounters)
{
    Console.WriteLine(""
    Console.WriteLine("\t delete<{0}> ", cntrId);
    foreach (Period p in listPeriod)
    {
        Console.WriteLine("\t\t period {0} -> {1} ", p.startDate, p.stopDate);
        deleteFrom_DBFactDayCons_byCounter_MUTEX(ref _sessionHolder, clientId,
        cntrId, p.startDate, p.stopDate);
    }
}
Console.WriteLine(""
Console.WriteLine("\t insert ");
foreach (Period p in listPeriod)
{
    Console.WriteLine("\t\t period {0} -> {1} ", p.startDate, p.stopDate);
    insertInto_DBFactDayCons_from_DBCalcCons_period_MUTEX(conn, clientId,
    p.startDate, p.stopDate);
}
Console.WriteLine(" 5 ");
deleteFrom_DBCalcCons(ref _sessionHolder);
Console.WriteLine(" - complete.");
return true;

private Dictionary<int, CGWithFacts> combine_config_with_cons(int clientId,
int cgIdLevelParent, DateTime dayDate, Dictionary<int, config_entry> child_conf_list,
bool useCalcConsOnly)
{
    // build cg dict for this child level using config_tree, cg_props,
counter_props, algo_props, cdv_mem, calc_cons_mem
    // todo
    Dictionary<int, CGWithFacts> cg_dict = new Dictionary<int, CGWithFacts>();

    foreach (config_entry ce in child_conf_list.Values)
    {
        switch (ce._cgttype)
        {
            case config_entry.CGTYPE_ENUM.CG:
                // skipped ... case config_entry.CGTYPE_ENUM.CG_SUB:
                // skipped ... case config_entry.CGTYPE_ENUM.CG_SUB_DISCO:
                if (useCalcConsOnly)
                {
                    // Get reference of correct type to the config_entry
                }
//config_entry_static cms = (config_entry_static)ce;
// load consumption from _dtCalcCons (one row for each
cgid/counterid/valuetype combination)
CGWithFacts cg = null;
if (!cg_dict.ContainsKey(ce._child_cgid)) // cg_CalcAlgo already
added to the cg_dict?
{
    switch (ce._cgtype)
    {
    case config_entry.CGTYPE_ENUM.CG:
        //<Exjobb> Changed to ConcurrentDictionary
        cg = new CGWithFacts(clientId, ce._child_cgid,
        cgIdLevelParent, (int)ce._cgtype, 0, new ConcurrentDictionary<int, CounterWithFacts>(), -1, null);
        //</Exjobb>
        break;
    case config_entry.CGTYPE_ENUM.CG_SUB:
    case config_entry.CGTYPE_ENUM.CG_SUB_DISCO:
        cg = new cg_SubCounter(clientId, ce._child_cgid,
        cgIdLevelParent, 0.0, new Dictionary<int, CounterWithFacts>(), -1, null);
        break;
    }
    cg_dict.Add(cg.cgId, cg);
}
else // already in dictionary - get reference to existing
{
    cg = (CGWithFacts)cg_dict[ce._child_cgid];
    load_cg_from_calc_cons_mem(cg, clientId, ce._child_cgid,
    dayDate);
}
else
{
    CGWithFacts cg = null;
    if (!cg_dict.ContainsKey(ce._child_cgid)) // cg_CalcAlgo already
added to the cg_dict?
    {
        cg = getCG(clientId, ce._child_cgid, dayDate);
        switch (ce._cgtype)
        {
        case config_entry.CGTYPE_ENUM.CG:
            cg_dict.Add(cg.cgId, cg);
            break;
        }
    }
    break;
}

return cg_dict;

#endregion

private void _calcDistribution(int clientId, int cgIdLevelParent, DateTime dayDate,
List<int> touchedCounters)
{


recurseDepth++; if (recurseDepth > 100) {
    recurseDepth = 0; // RESET recurseDepth så att den inte genererar fel igen
    throw new ApplicationException("_calcDistribution recurseDepth > 100");
}

// build mainCG for this child level using config_tree, cg_props, counter_props, algo_props, cdv_mem, calc_cons_mem
CGWithFacts mainCG = GetCGFromCalcCons(clientId, cgIdLevelParent, dayDate);
Debug.WriteLine(String.Format("treeDepth={0}, clientId={1}, cgId={2}, dayDate='{3}', cgType={4}", recurseDepth, clientId, cgIdLevelParent, dayDate, mainCG.cgType.ToString()));

// Join configuration with consumption data for the specified day
Dictionary<DateTime, Dictionary<int, config_entry>> cgParent = m_dicCGs_DailyConfigurationTree[cgIdLevelParent];
Dictionary<int, config_entry> child_conf_list = null;
if (cgParent.ContainsKey(dayDate))
    child_conf_list = cgParent[dayDate];
else
    child_conf_list = new Dictionary<int, config_entry>();

// build cg_dict for this child level using config_tree, cg_props, counter_props, algo_props, cdv_mem, calc_cons_mem
Dictionary<int, CGWithFacts> cg_dict = combine_config_with_cons(clientId, cgIdLevelParent, dayDate, child_conf_list, true);

// We will reset distribution on all dependant CG:s excluding the one we have in the _dicAlgoCGs_Checkpoint
// The one in _dicAlgoCGs_Checkpoint will in a moment be adjusted by _calcAlgoCons
double sum_cons_from_children = 0.0;
foreach (CGWithFacts childCG in cg_dict.Values)
{
    sum_cons_from_children += childCG.Cons;
}
mainCG.Distributed_Cons = 0.0; // reset the distribution
mainCG.Distributed_Cons += sum_cons_from_children;

// Adjust mainCG.Counters
foreach (CounterWithFacts mainCounter in mainCG.Counters.Values)
{
    // Calc main counters current proportion
    double mainCounter_proportion_on_main_current_total = 1.0;
    if (mainCG.Cons != 0) mainCounter_proportion_on_main_current_total = zeroIfNaN(mainCounter.Cons / mainCG.Cons);

    mainCounter.Distributed_Cons = zeroIfNaN(- (mainCG.Distributed_Cons * mainCounter_proportion_on_main_current_total));
}
// Store mainCG as factDayCons
foreach (CounterWithFacts c in mainCG.Counters.Values)
{
    if (!touchedCounters.Contains(c.CounterId))
        touchedCounters.Add(c.CounterId);

    // deleteFrom_dtCalcCons(c.ClientId, c.CounterId, mainCG.cgId, dayDate);
collCalcCons_delete(c.ClientId, c.CounterId, mainCG.cgId, dayDate);
collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate, c.Cons,
CALCCONS_TYPE.CONS_DATA);
collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate,
c.Distributed_Cons, CALCCONS_TYPE.CONS_DISTRIBUTED_DATA);
collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate,
c.Correction_Cons, CALCCONS_TYPE.CONS_CORRECTION_DATA);
}
recurseDepth--;
}
#endregion
#region "CALC_CONS"
//<Exjobb> Added "ref List<ForDeletion> deleteList" as a param.
private void _calcCons(int clientId, int cgIdLevelParent, DateTime dayDate,
List<int> touchedCounters, ref ConcurrentBag<ForDeletion> deleteList)
//</Exjobb>
{
    recurseDepth++; 
    if (recurseDepth > 100)
    {
        recurseDepth = 0; // RESET recurseDepth så att den inte genererar fel igen
        throw new ApplicationException("_calcCons recurseDepth > 100");
    }

    // build mainCG for this child level using config_tree, cg_props, counter_props,
algo_props, cdv_mem, calc_cons_mem
CGWithFacts mainCG = getCG(clientId, cgIdLevelParent, dayDate);

    if (!m_cg_in_maintree_path.ContainsKey(cgIdLevelParent))
        m_cg_in_maintree_path.Add(cgIdLevelParent, cgIdLevelParent); // Add to list
so that we keep track of main tree path of CG:s

    // Join configuration with consumption data for the specified day
    Debug.WriteLine(String.Format("treeDepth={0}, clientId={1}, cgId={2},
dayDate='[3]', cgType={4}",
recurseDepth, clientId, cgIdLevelParent, dayDate,
mainCG.cgType.ToString()));
    //</Exjobb>
CounterGroup cgLevelParent = m_collCGs.GetEntityWithId(cgIdLevelParent);
switch (cgLevelParent.CounterGroupType)
{
    case CounterGroupType.CounterGroup:
    // case CounterGroupType.SubCounterGroup:
    // case CounterGroupType.SubDividerDisconnected:
        foreach (int cntr in cgLevelParent.CounterIds.Values)
        {
            if (!m_deletedCounters.Contains(cntr)) m_deletedCounters.Add(cntr);
        }
        break;
}

    // Join configuration with consumption data for the specified day
Dictionary<DateTime, Dictionary<int, config_entry>> cgParent = m_dicCGs_DailyConfigurationTree[cgIdLevelParent];

Dictionary<int, config_entry> child_conf_list = null;

if(cgParent.ContainsKey(dayDate))
{
    child_conf_list = cgParent[dayDate];
}
else
{
    child_conf_list = new Dictionary<int, config_entry>();
}

// build cg_dict for this child level using config_tree, cg_props,
counter_props, algo_props, cdv_mem, calc_cons_mem
Dictionary<int, CGWithFacts> cg_dict = combine_config_with_cons(clientId, cgIdLevelParent, dayDate, child_conf_list, false);

//<Exjobb> instead of removing everything here we add it to the
//deleteList and remove it after.
deleteList.Add(new ForDeletion { CG = mainCG, date = dayDate });

    //foreach (CounterWithFacts c in mainCG.Counters.Values)
    //{ //
    //    if (!touchedCounters.Contains(c.CounterId))
    //        touchedCounters.Add(c.CounterId);
    //    // deleteFrom_dtCalcCons(c.ClientId, c.CounterId, mainCG.cgId,
    //    dayDate);
    //    // collCalcCons_delete(c.ClientId, c.CounterId, mainCG.cgId, dayDate);
    //    // collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate,
    //    c.Cons, CALCCONS_TYPE.CONS_DATA);
    //}
    //</Exjobb>
    recurseDepth--;

#endregion

private void load_cg_from_calc_cons_mem(CGWithFacts cg, int clientId, int child_cgid, DateTime dayDate)
{
    // string select = String.Format("cgid={0} and daydate='{1}'", child_cgid,
    dayDate);
    var q1 = from ocons in m_collCalcCons
             where (ocons.cgId == child_cgid && ocons.dayDate == dayDate)
            select ocons;
    // foreach (cdv_mem.calcConsRow calc_cons_row in _dtCalcCons.Select(select))
    foreach (calcCons calc_cons_row in q1)
    {
        // Check if this counter has been added to the cg_CalcAlgo.Counters
        CounterWithFacts c = null;
        if (!cg.Counters.ContainsKey(calc_cons_row.counterId))
        {
            c = new CounterWithFacts(clientId, calc_cons_row.counterId);
            //<Exjobb> Due to the change to ConcurrentDictionary AddOrUpdate
            //has to be used instead of add.
```csharp
        cg.Counters.AddOrUpdate(calc_cons_row.counterId, c, (key, oldvalue) =>
        c);
        //</Exjobb>
        } else // already in dictionary - get reference to existing
        {
            c = cg.Counters[calc_cons_row.counterId];
        }
        // Now we have cga and c so we can set Cons, Distributed_Cons,
        Correction_Cons depending on current rows valuetype
        switch ((CALCCONS_TYPE)calc_cons_row.calcValueType)
        {
            case CALCCONS_TYPE CONS_DATA:
                c.Cons += calc_cons_row.dayValue;
                cg.Cons += calc_cons_row.dayValue;
                break;
            case CALCCONS_TYPE CONS_DISTRIBUTED_DATA:
                c.Distributed_Cons += calc_cons_row.dayValue;
                cg.Distributed_Cons += calc_cons_row.dayValue;
                break;
            case CALCCONS_TYPE CONS_CORRECTION_DATA:
                c.Correction_Cons += calc_cons_row.dayValue;
                cg.Correction_Cons += calc_cons_row.dayValue;
                break;
        }
    }
}  

#region "CALC_CORRECTIONS"
private void _calcCorrections(int clientId, int cgId, DateTime dayDate, List<int> touchedCounters)
{
    recurseDepth++;
    if (recurseDepth > 100) throw new ApplicationException("_calcCorrections
    recurseDepth > 100");
    CGWithFacts mainCG = GetCGFromCalcCons(clientId, cgId, dayDate);
    Dictionary<int, CGWithFacts> cg_dict = null;
    // find all cg children connected to cgId
    // Join configuration with consumption data for the specified day
    Dictionary<DateTime, Dictionary<int, config_entry>> cgParent = 
    m_dicCGs_DailyConfigurationTree[cgId];
    Dictionary<int, config_entry> child_conf_list = null;
    if (cgParent.ContainsKey(dayDate))
    {
        child_conf_list = cgParent[dayDate];
    }
    else
    {
        child_conf_list = new Dictionary<int, config_entry>();
    }
    cg_dict = combine_config_with_cons(clientId, cgId, dayDate, child_conf_list,
    true);
    if (mainCG.Cons < 0) Debug.WriteLine(String.Format("mainCG.Cons < 0 at clientId={0}, cgId={1}"));
    // More than 100% distributed?
```
if ((mainCG.Cons >= 0.0) && (Math.Abs(mainCG.Distributed_Cons) > mainCG.Cons))
{
    // Yes create corrections
    // reset the previous corrections
    mainCG.Correction_Cons = 0.0;
    foreach (CounterWithFacts mainCounter in mainCG.Counters.Values)
    {
        mainCounter.Correction_Cons = 0.0;
    }
    // throw new ApplicationException("More than 100% distributed - stop for
    until buf fixed. /PB");
    // distribute a Correction among all involved cg/counter
    double correction_diff_total_for_maincg = Math.Abs(mainCG.Distributed_Cons) - mainCG.Cons; //totalSum - mainCG.Cons; // mainCG.Cons;
    Debug.WriteLine(String.Format("\t Needs correction for {0} units",
        correction_diff_total_for_maincg));
    // Store mainCG as FactDayCons
    foreach (CounterWithFacts c in mainCG.Counters.Values)
    {
        if (!touchedCounters.Contains(c.CounterId))
            touchedCounters.Add(c.CounterId);
        // deleteFrom_dtCalcCons(c.ClientId, c.CounterId, mainCG.cgId, dayDate);
        collCalcCons_delete(c.ClientId, c.CounterId, mainCG.cgId, dayDate, c.Cons, CALCCONS_TYPE.CONS_DATA);
        collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate, c.Distributed_Cons, CALCCONS_TYPE.CONS_DISTRIBUTED_DATA);
        collCalcCons_insert(c.ClientId, c.CounterId, mainCG.cgId, dayDate, c.Correction_Cons, CALCCONS_TYPE.CONS_CORRECTION_DATA);
    }

    // Recursion - traverse down the tree
    foreach (CGWithFacts childCG in cg_dict.Values)
    {
        _calcCorrections(clientId, childCG.cgId, dayDate, touchedCounters);
    }
    recurseDepth--;
using (SqlCommand sqlCmd = new SqlCommand())
{
    sqlCmd.CommandText = "select top 1 cgId, clientId, cgTypeId, [level]";
    sqlCmd.CommandText += String.Format(" from ufn_getConnectedCGs({0}, {1})", clientId, counterId);
    sqlCmd.CommandText += " order by [level] desc";
    sqlCmd.Connection = conn;
    sqlCmd.CommandTimeout = 600;
    SqlDataReader reader = sqlCmd.ExecuteReader();
    if (reader.HasRows)
    {
        reader.Read();
        cgId = reader.GetInt32(0);
        // if (!reader.IsDBNull(1)) cgParentId = reader.GetInt32(1);
    }
    reader.Close();
}

return cgId;

private void loadCgCounters_DB(ref RCSession _sessionHolder, int clientId, int root_cgid, CounterGroup cgProps, CounterCollection collCounters)
{
    loadCGCountersAndProps_DB(ref _sessionHolder, clientId, root_cgid, cgProps, collCounters);
}

private void loadCgProps_DB(ref RCSession _sessionHolder, int clientId, int root_cgid, CounterGroupCollection collCGInfos) //, Dictionary<int, CounterProperty> counter_mem, Dictionary<int, AlgoCheckPoint> algo_mem)
{
    // load all CG properties and store in cg_mem
    if (collCGInfos.GetEntityWithId(root_cgid) == null)
    {
        CounterGroup cg = loadConfig_CGProps_DB(ref _sessionHolder, clientId, root_cgid);
        collCGInfos.Add(cg);
    }
}

/// <summary>
/// Loads the entire configuration tree starting with specified root_cgid and for
/// each day in the time interval [start, stop).
/// After completion the config tree is loaded in memory into
/// _dicCGs_DailyConfigurationTree and also is
/// _dicCGInfos, _dicCounterInfos and _dicAlgoIds dictionaries are populated with
/// properties for cg, counter and algoId.
/// </summary>
/// <param name="conn">The SQL connection</param>
/// <param name="clientId">The client id.</param>
/// <param name="root_cgid">The cgid to start from</param>
/// <param name="startDate">The start date of paeriod</param>
/// <param name="stopDate">The stop date of period</param>
private void loadConfigTree_DB(SqlConnection conn, ref RCSession _sessionHolder, int clientId, int root_cgid, DateTime startDate, DateTime stopDate)
// Will load required properties for specified root_cgid, its counters (if applicable) and any algoId’s
loadCgProps_DB(ref _sessionHolder, clientId, root_cgid, m_collCGs);
CounterGroup cgRoot = m_collCGs.GetEntityWithId(root_cgid);
loadCgCounters_DB(ref _sessionHolder, clientId, root_cgid, cgRoot, m_collCounters);

switch (cgRoot.CounterGroupType)
{
    case CounterGroupType.CounterGroup: // Anläggning
        // case CounterGroupType.SubCounterGroup: // Fördelare med
        undermätare
        // case CounterGroupType.SubDividerDisconnected: // Fristånde fördelare
        med undermätare
        {
            // loadCgAlgoIds_DB(ref _sessionHolder, clientId, root_cgid,
            m_dicAlgoIds);
            break;
        }
    default:
        break;
}

Dictionary<DateTime, Dictionary<int, config_entry>> root = null;
if (!m_dicCGs_DailyConfigurationTree.ContainsKey(root_cgid))
{
    root = new Dictionary<DateTime, Dictionary<int, config_entry>>();
    m_dicCGs_DailyConfigurationTree.Add(root_cgid, root);
}
else
    root = m_dicCGs_DailyConfigurationTree[root_cgid];

// build cg_dict for this child level
switch (cgRoot.CounterGroupType)
{
    case CounterGroupType.CounterGroup: // Anläggning
    case CounterGroupType.SubCounterGroup: // Fördelare med undermätare
    case CounterGroupType.SubDividerDisconnected: // Fristånde fördelare med
    undermätare
    {
        // skipped ... loadConfig_SubCounterChildren_DB(conn, clientId,
        root_cgid, startDate, stopDate, root);
        // skipped ... loadConfig_StaticChildren_DB(conn, clientId,
        root_cgid, startDate, stopDate, root);
        // skipped ... loadConfig_M2Children_DB(conn, clientId, root_cgid,
        startDate, stopDate, root);
        // skipped ... loadConfig_Algochildren_DB(conn, clientId, root_cgid,
        startDate, stopDate, root);
        break;
    }
    default:
        break;
}

// recursion down the tree
// iterate all days in date span
foreach (Dictionary<int, config_entry> day in root.Values )
{
    // iterate all childcgs in each day
foreach (config_entry childcgList in day.Values)
{
    // If not already in config tree as a root - then dive down and add it
    if (!m_dicCGs_DailyConfigurationTree.ContainsKey(childcgList._child_cgid))
    {
        // recursion
        loadConfigTree_DB(conn, ref _sessionHolder, clientId, childcgList._child_cgid, startDate, stopDate);
    }
}

/// <summary>
/// Loads all counters involved within a CG
/// </summary>
/// <param name="conn">The SQL connection</param>
/// <param name="clientId">The client id</param>
/// <param name="cgId">The cgId to investigate</param>
/// <param name="cgProps">The CG properties for cgId</param>
/// <param name="counter_mem">? counter_mem</param>
private void loadCGCountersAndProps_DB(SqlConnection conn, int clientId, int cgId, CGProperty cgProps, Dictionary<int, CounterProperty> counter_mem)
{
    CounterManager mgrCounter = new CounterManager(_sessionHolder);
    CounterCollection collCs = mgrCounter.GetListFromCounterGroup(cgId);
    foreach (BusinessEntities.CounterObjects.Counter c in collCs)
    {
        // BusinessEntities.CounterObjects.Counter = collCounters.GetEntityWithId(c.counterId);
        if (c.parentCGId == null)
        {
            c.parentCGId = cgId;
            c.ParentCounterGroup = cgProps;
        }
        if (collCounters.GetEntityWithId(c.counterId) == null)
            collCounters.Add(c);
        // Also add counterId to the CGs list of counters
        if (!cgProps.CounterIds.ContainsKey(c.counterId))
            cgProps.CounterIds.Add(c.counterId, c.counterId);
    }
}

private CGWithFacts GetCGFromCalcCons(int clientId, int cgId, DateTime dayDate)
{
    // Get cgType out of cg_props
    // int cgtypeId = (int)_dicCGInfos[cgId].cgType;
    CounterGroup o_cgid = m_collCGs.GetEntityWithId(cgId);
    int cgtypeId = (int)o_cgid.CounterGroupType;
    // get connected_cgid out of config_tree / cg_props
    int connected_cgid = (int)o_cgid.ConnectedToCGId;

    int wwp_inherited_cgid = -1;
    DateTime? wwp_inherited_cgid_dateStart = null;
    if (o_cgid.collCGWWPs.Count > 0)
{ foreach (CGWarmWaterPart wwp in o_cgid.collCGWWPs) {
    if (dayDate >= wwp.dateStart && (wwp.dateStop == null || dayDate <= wwp.dateStop)) {
        wwp_inherited_cgId = cgId;
        wwp_inherited_cgId_dateStart = wwp.dateStart;
        break;
    }
    }
}

if (wwp_inherited_cgId == -1 && (CGWithFacts.CGTYPE_ENUM)o_cgid.CounterGroupType == CGWithFacts.CGTYPE_ENUM.CG) wwp_inherited_cgId = 0; // CG kan ej ärva wwp så sätt till 0 (klar, men ingen ärvning finns)
     //<Exjobb> Changed to ConcurrentDictionary
CGWithFacts cg = new CGWithFacts(clientId, cgId, connected_cgId, cgtypeId, 0.0, new ConcurrentDictionary<int, CounterWithFacts>(), wwp_inherited_cgId, wwp_inherited_cgId_dateStart);
    //</Exjobb>
load_cg_from_calc_cons_mem(cg, clientId, cgId, dayDate);
return cg;
}

// get CG and all counters and cons for a day (from factDayCons)
private CGWithFacts getCG( //SqlConnection conn,
    int clientId,
    int cgId,
    DateTime dayDate)
{
    CGWithFacts cg = null;
    CounterGroup o_cgid = m_collCGs.GetEntityWithId(cgid);
    // Get cgType out of cg_props
    int cgtypeid = (int)o_cgid.CounterGroupType;
    CGWithFacts.CGTYPE_ENUM cgType = (CGWithFacts.CGTYPE_ENUM)cgtypeid;
    // get connected_cgId out of config_tree / cg_props
    int wwp_inherited_cgId = -1;
    DateTime? wwp_inherited_cgId_dateStart = null;
    if (o_cgid.collCGWWPs.Count > 0) {
        foreach (CGWarmWaterPart wwp in o_cgid.collCGWWPs) {
            if (dayDate >= wwp.dateStart && (wwp.dateStop == null || dayDate <= wwp.dateStop)) {
                wwp_inherited_cgId = cgId;
                wwp_inherited_cgId_dateStart = wwp.dateStart;
                break;
            }
        }
    }
    if (wwp_inherited_cgId == -1 && cgType == CGWithFacts.CGTYPE_ENUM.CG)
wwp_inherited_cgId = 0; // CG kan ej ärva wwp så sätt till 0 (klar, men ingen ärvning finns)

//<Exjobb> Changed to ConcurrentDictionary. The things within /* */ are variables that might //speed up the processes since you wont have to recalculate the size of the dictionary in that case. //I haven't done any tests with/without them though so that's why I'm not using them at the moment. /*
int initialCapacity = 101;
int numProcs = Environment.ProcessorCount;
int concurrencyLevel = numProcs * 2;
*/
ConcurrentDictionary<int, CounterWithFacts> counters = new ConcurrentDictionary<int, CounterWithFacts>(/*concurrencyLevel, initialCapacity*/);

//</Exjobb>
switch (cgType)
{
  case CGWithFacts.CGTYPE_ENUM.CG:
  // case CGWithFacts.CGTYPE_ENUM.CG_SUB:
  // case CGWithFacts.CGTYPE_ENUM.CG_SUB_DISCO:
  {
    // load cons for cg from cdv_mem
    // remember to multiply with counterConstant!
    // use cg_props.counterids to enumerate all counters for this
cgid
double cgcons = 0.0;
    foreach (int counterId in o_cgid.CounterIds.Values)
    { // todo: we could ignore days not within counterDateStart and counterDateStop interval. // as an alternative we can reason that cdv should not contain any data outside the start/stop date interval.
      // get constant out of counter_props
double counterConstant = m_collCounters.GetEntityWithId(counterId).Constant;
      var q1 = from cdv in m_collCounterDailyConsumption
                   where (cdv.counterId == counterId && cdv.dayDate ==
                   dayDate)
                   // ? where (cdv.clientId == clientId &&
cdv.counterId == counterId &&
cdv.dayDate == dayDate)
                   select cdv;

      foreach (CounterDayValue cdv in q1)
      {
        // calculate CG cons by adding cons from all counters
cgcons += (cdv.dayValue * counterConstant);
        // add to counters
        CounterWithFacts c = new CounterWithFacts(clientid, counterId);
        c.Cons = (cdv.dayValue * counterConstant);
        //<Exjobb> Due to the change to ConcurrentDictionary
        AddOrUpdate //has to be used instead of add.
counters.AddOrUpdate(c.CounterId, c, (key, oldValue) =>
c);
        //</Exjobb>
      }
  }
}
```csharp
    //load cons for CG from calc_cons_mem
    double cgcons = 0.0;
    double cgdistcons = 0.0;
    double cgcorrcons = 0.0;
    string selectfilter = String.Format("clientId={0} and cgId={1} and dayDate='{2}'", clientid, cgid, dayDate);
    var q1 = from ocons in m_collCalcCons
              where (ocons.cgId == cgid && ocons.dayDate == dayDate)
              select ocons;

    // foreach (cdv_mem.calcConsRow r in calc_cons_rows)
    foreach (calcCons r in q1)
    {
        // we get multiple rows (permutation of counterId and valueType)
        // but cgId is always the same
        CounterWithFacts c = null;
        // add to counters - if not already present
        if (counters.ContainsKey(r.counterId))
        {
            c = counters[r.counterId];
        }
        else
        {
            c = new CounterWithFacts(clientid, r.counterId);
            //<Exjobb> Due to the change to ConcurrentDictionary
            counters.AddOrUpdate(r.counterId, c, (key, oldValue) => c);
            //</Exjobb>
        }

        // use a switch to set the cons value into the correct
        valueType
        switch (r.calcValueType)
        {
            case 1:
                c.Cons = r.dayValue;
                break;
            case 2:
                c.Distributed_Cons = r.dayValue;
                break;
            case 3:
                c.Correction_Cons = r.dayValue;
                break;
        }
    }
```
// use a switch to set the cons value into the correct valuetype
switch (r.calcValueType)
{
    case 1:
        cgcons += r.dayValue;
        break;
    case 2:
        cgdistcons += r.dayValue;
        break;
    case 3:
        cgcorrcons += r.dayValue;
        break;
}

cg = new CGWithFacts(clientid, cgid, connected_cgId, cgtypeid, 
0.0, counters, wwp_inherited_cgId, wwp_inherited_cgId_dateStart);

cg.Cons = cgcons;
    cg.Distributed_Cons = cgdistcons;
    cgCorrection_Cons = cgcorrcons;
    }
    break;
}

// Sanity check so that sum of all counters consumption equals cg consumption
double cgSum = 0;
foreach (CounterWithFacts c in counters.Values)
{
    cgSum += c.Cons;
} 
if (cgSum != cg.Cons) throw new ApplicationException("cgSum != cons");
return cg;

// get CG and all counters and cons for a day (from factDayCons)
private CGProperty loadConfig_CGProps_DB(SqlConnection conn, int clientid, int cgid)
{
    using (SqlCommand sqlCmd = new SqlCommand())
    {
        int connected_cgId = 0;
        CGProperty.CGTYPE_ENUM cgTypeId;


        sqlCmd.Parameters["@clientId"].Value = clientid;
        sqlCmd.Parameters["@cgId"].Value = cgid;

        sqlCmd.CommandText = " select distinct cg.clientId, cg.cgId, cg.cgTypeId, ";
        sqlCmd.CommandText += " (case when cg.cgTypeId=1 then null      when 
        cg.cgTypeId=2 then cgsub.parentCgId      when cg.cgTypeId=3 then null      when 
        cg.cgTypeId=4 then cgm2.parentCgId      when cg.cgTypeId=5 then cgstat.parentCgId      when 
        cg.cgTypeId=6 then null      when cg.cgTypeId=7 then null end) as connectedCgId, ";
```
sqlCmd.CommandText += " cg.cgDateStart, cg.cgDateStop, cgstat.sharePercent ";
sqlCmd.CommandText += " from countergroup cg ";
sqlCmd.CommandText += " left outer join counter_countergroup ccg on 
c cg.clientid = ccg.clientid and ccg.cgid = cg.cgid ";
sqlCmd.CommandText += " left outer join counter_subCounter cgsb on 
c cgsb.clientid = cg.clientid and cgsb.cgid = cg.cgid ";
sqlCmd.CommandText += " left outer join counter_group cstat on 
c cstat.clientid = cg.clientid and cstat.cgid = cg.cgid ";
sqlCmd.CommandText += " left outer join counter_CalcM2 cgm2 on 
c cgm2.clientid = cg.clientid and cgm2.cgid = cg.cgid ";
sqlCmd.CommandText += " left outer join dimCG on dimCG.clientid = cg.clientid ";
sqlCmd.CommandText += " left outer join dimClient on dimClient.clientid = cg.clientid ";
sqlCmd.CommandText += " where cg.clientid = @clientId and cg.cgid = @cgId ";

CGProperty cg = null;
sqlCmd.Connection = conn;
sqlCmd.CommandText = "SELECT 
    cg.clientid, cg.cgid, cg.cgTypeId, 
    ccg.cgDateStart, ccg.cgDateStop, 
    cstat.sharePercent 
FROM countergroup cg 
    LEFT OUTER JOIN counter_countergroup ccg ON 
    cg.clientid = ccg.clientid AND ccg.cgid = cg.cgid 
    LEFT OUTER JOIN counter_subCounter cgsb ON 
    cgsb.clientid = cg.clientid AND cgsb.cgid = cg.cgid 
    LEFT OUTER JOIN counter_group cstat ON 
    cstat.clientid = cg.clientid AND cstat.cgid = cg.cgid 
    LEFT OUTER JOIN counter_CalcM2 cgm2 ON 
    cgm2.clientid = cg.clientid AND cgm2.cgid = cg.cgid 
    LEFT OUTER JOIN dimCG ON dimCG.clientid = cg.clientid 
    LEFT OUTER JOIN dimClient ON dimClient.clientid = cg.clientid 
WHERE cg.clientid = @clientId AND cg.cgid = @cgId ";

SqlDataReader reader = sqlCmd.ExecuteReader();
if (reader.Read())
{
    clientId = reader.GetInt32(0);
    cgid = reader.GetInt32(1);
    cgTypeId = (CGProperty.CGTYPE_ENUM)reader.GetInt32(2);
    if (!reader.IsDBNull(3)) connected_cgId = reader.GetInt32(3); else connected_cgId = -1;
    cgDateStart = reader.GetDateTime(4);
    cgDateStop = new DateTime(2100, 12, 31);
    if (!reader.IsDBNull(5)) cgDateStop = reader.GetDateTime(5);
    switch (cgTypeId)
    {
    case CGProperty.CGTYPE_ENUM.CG:
        // case CGProperty.CGTYPE_ENUM.CG_SUB:
        // case CGProperty.CGTYPE_ENUM.CG_SUB_DISCO:
        //     // get all counters connected to cg via counter_countergroup
        //     // using (SqlConnection conn2 = new SqlConnection(getConnStr()))
        //     // {
        //     //     conn2.Open();
        //     //     getCGCounters(conn2, clientId, cgId, cgType, counters);
        //     //     conn2.Close();
        //     // }
        //     // //cg = new CG(clientid, cgId, connected_cgId, cgTypeId, 0.0, counters);
        //     break;

    case CGProperty.CGTYPE_ENUM.CG_STATIC:
        // Get share %
        double sharePercent = 0.0;
        if (!reader.IsDBNull(6)) sharePercent = reader.GetDouble(6);
        cg = new cg_CalcStaticProps(cg.clientId, cg.cgId, cg.cgIdParent, sharePercent);
        // break;

    default:
        cg = new CGProperty(clientId, cgId, connected_cgId, cgTypeId, 
        cgDateStart, cgDateStop);
        break;
    }
}
```
    reader.Close();
}
else
    throw new ApplicationException("getCG :: Could not obtain cgTypeId for the cgId.");

    return cg;
}

private CounterGroup loadConfig_CGProps_DB(ref RCSession _sessionHolder, int clientId, int cgId)
{
    CounterGroupManager mgrCG = new CounterGroupManager(_sessionHolder);
    CounterGroup ocg = mgrCG.GetItem(cgid, LoadOptions.LoadEntityWWP | LoadOptions.LoadEntityCGUsage);

    if (ocg != null)
    {
        DateTime cgDateStop = (ocg.DateStop != null) ? (DateTime)ocg.DateStop : new DateTime(2100, 12, 31);
        ocg.DateStop = cgDateStop;

        int connectedToCGgId = -1;

        switch (ocg.CounterGroupType)
        {
            case CounterGroupType.AreaDivider:
                AreaDividerManager mgrM2 = new AreaDividerManager(_sessionHolder);
                AreaDivider cgm2 = mgrM2.GetItem_DivideInfoOnly(cgid);
                if (cgm2 != null && cgm2.ConnectedCounterGroupId != null)
                    connectedToCGgId = (int)cgm2.ConnectedCounterGroupId;
                break;

            case CounterGroupType.PercentageDivider:
                PercentageDividerManager mgrPercentage = new PercentageDividerManager(_sessionHolder);
                PercentageDivider cgstat = mgrPercentage.GetItem_DivideInfoOnly(cgid);
                if (cgstat != null)
                    connectedToCGgId = cgstat.ConnectedCounterGroupId;
                break;

            case CounterGroupType.SubCounterGroup:
                SubCounterDividerManager mgrSub = new SubCounterDividerManager(_sessionHolder);
                SubCounterDivider cgsub = mgrSub.GetItem_DivideInfoOnly(cgid);
                if (cgsub != null)
                    connectedToCGgId = cgsub.ConnectedCounterGroupId;
                break;

            default:
                break;
        }
    */
    ocg.ConnectedToCGId = connectedToCGgId;
    }
else
    throw new ApplicationException("getCG :: Could not obtain cgTypeId for the cgId.");

    return ocg;
private void deleteFrom_DBCalcCons(ref RCSession _sessionHolder)
{
    calcConsManager mgrCalcCons = new calcConsManager(_sessionHolder);
    bool deleted = mgrCalcCons.DeleteAll();
}

private void insertInto_DBCalcCons_from_collCalcCons_BULK(ref RCSession _sessionHolder, int clientId)
{
    int numrows = -1;
    // bulkcopy
    var bulk = from ocons in m_collCalcCons
               // where (ocons.calculated == 1 && !(ocons.dayValue == 0.0 &&
               // ocons.consnorm == 0.0 && (ocons.wwp == null || (ocons.calcValueType > 1 && ocons.wwp ==
               // 0.0))))
               where (ocons.calculated == 1)
               select new
    {
        clientId = ocons.clientId,
        counterId = ocons.counterId,
        cgId = ocons.cgId,
        calcValueType = ocons.calcValueType,
        dayDate = ocons.dayDate,
        dayValue = ocons.dayValue,
        calctime = ocons.calctime,
        consnorm = ocons.consnorm,
        wwp = ocons.wwp,
        // degreeDayKey = ocons.degreeDayKey
        degreeDayKey = 1266534
    };
    numrows = bulk.Count();
    if (numrows == 0)
        return;

    IDataReader dr = bulk.AsDataReader();
    using (SqlBulkCopy bcp = new SqlBulkCopy(_sessionHolder.ConnectionString))
    {
        bcp.DestinationTableName = "dbo.calcCons_v25";
        // in case the columns order in 'bulk' does not match the columns order in
        // table 'calcCons_v25'
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("clientId", "clientId"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("counterId", "counterId"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("cgId", "cgId"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("calcValueType", "calcValueType"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("dayDate", "dayDate"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("dayValue", "dayValue"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("calctime", "calctime"));
        bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("consnorm", "consnorm"));
    }
bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("wwp", "wwp");
bcp.ColumnMappings.Add(new SqlBulkCopyColumnMapping("degreeDayKey", "degreeDayKey");

bcp.BatchSize = 5000;
bcp.WriteToServer(dr);
}
return;

private void insertInto_DBFactDayCons_from_DBCalcCons_period_DB(SqlConnection conn, int clientId, DateTime startDate, DateTime stopDate)
{
    using (SqlCommand sqlCmd = new SqlCommand())
    {
        sqlCmd.CommandType = CommandType.StoredProcedure;
        // sqlCmd.CommandText = "usp_insertFactDayCons_from_calcCons2_period_v25";
        sqlCmd.CommandText = "usp_insertFactDayCons_from_calcCons";
        sqlCmd.Parameters.Add(new SqlParameter("@clientId", SqlDbType.Int));
        sqlCmd.Parameters.Add(new SqlParameter("@startDate", SqlDbType.DateTime));
        sqlCmd.Parameters.Add(new SqlParameter("@stopDate", SqlDbType.DateTime));
        sqlCmd.Parameters["@clientId"].Value = clientId;
        sqlCmd.Parameters["@startDate"].Value = startDate;
        sqlCmd.Parameters["@stopDate"].Value = stopDate;
        sqlCmd.Connection = conn;
        DateTime t1 = DateTime.Now;
        sqlCmd.CommandTimeout = 600;
        int numRowsAffected = sqlCmd.ExecuteNonQuery();
        DateTime t2 = DateTime.Now;
        Debug.WriteLine(t2.Subtract(t1));
    }
}
private void insertInto_DBFactDayCons_from_DBCalcCons_period_MUTEX(SqlConnection conn, int clientId, DateTime startDate, DateTime stopDate)
{
    // Serialize access to aggregate table so that multiple instances of
cCalcConsole does not overload database
    // obtain mutex for given clientId
    using (Mutex mut = new Mutex(false, "serialize_FactDayCons_" + _dbConnstrName))
    {
        bool bAquired = mut.WaitOne(_wait_for_db_mutex_milliseconds, false); // 120
        if (bAquired)
        {
            try
            {
                insertInto_DBFactDayCons_from_DBCalcCons_period_DB(conn, clientId, startDate, stopDate);
            }
            finally
            {
                mut.ReleaseMutex();
            }
        }
        else
        { second wait max
        }
throw new ApplicationException("insertFactDayCons_from_calcCons could not obtain serialization mutex " + "serialize_" + _dbConnstrName + " within 120 seconds.");

private void deleteFrom_DBFactDayCons_byCounter_DB(ref RCSession _sessionHolder, int clientId, int counterId, DateTime startDate, DateTime stopDate)
{
    factDayConsManager mgrFactDayCons = new factDayConsManager(_sessionHolder);
    int deletedRows = mgrFactDayCons.Delete(counterId, startDate, stopDate);
}

private void deleteFrom_DBFactDayCons_byCounter_MUTEX(ref RCSession _sessionHolder, int clientId, int counterId, DateTime startDate, DateTime stopDate)
{
    // Serialize access to aggregate table so that multiple instances of rcCalcConsole does not overload database
    // obtain mutex for given clientId
    using (Mutex mut = new Mutex(false, "serialize_FactDayCons_" + _dbConnstrName))
    {
        bool bAquired = mut.WaitOne(_wait_for_db_mutex_milliseconds, false); // 120 second wait max
        if (bAquired)
        {
            try
            {
                deleteFrom_DBFactDayCons_byCounter_DB(ref _sessionHolder, clientId, counterId, startDate, stopDate);
            }
            finally
            {
                mut.ReleaseMutex();
            }
        }
        else
            throw new ApplicationException("deleteFrom_DBFactDayCons_byCounter_MUTEX could not obtain serialization mutex " + "serialize_" + _dbConnstrName + " within 120 seconds.");
    }
}

private void collCalcCons_delete(int clientId, int cgId, DateTime dayDate)
{
    IEnumerable<calcCons> q1 = m_collCalcCons.Where(cons => (cons.cgId == cgId && cons.dayDate == dayDate));
    int iCount = q1.Count() - 1;
    for (int i = iCount; i >= 0; i--)
    {
        m_collCalcCons.Remove(q1.ElementAt(i));
    }
}

private void collCalcCons_delete(int clientId, int counterId, int cgId, DateTime dayDate)
{
    IEnumerable<calcCons> q1 = m_collCalcCons.Where(cons => (cons.cgId == cgId && cons.counterId == counterId && cons.dayDate == dayDate));
    int iCount = q1.Count() - 1;
    for (int i = iCount; i >= 0; i--)
    {
        m_collCalcCons.Remove(q1.ElementAt(i));
    }
}
private void collCalcCons_insert(int clientId, int counterId, int cgId, DateTime dayDate, double dayValue, CALCCONS_TYPE calcConsType, DateTime calcTime, int calculated) {
    // NaN indicates trouble with division by zero.
    // Throw an explicit exception here so that the problem becomes more obvious in the calculations error field.
    if (double.IsNaN(dayValue)) throw new ApplicationException("insertCalcCons trying to store a NaN (not a number) value. NaN indicates trouble with division by zero.");

    calcCons cons = new calcCons();
    cons.clientId = clientId;
    cons.counterId = counterId;
    cons.cgId = cgId;
    cons.calcValueType = (int)calcConsType;
    cons.dayDate = dayDate;
    cons.dayValue = dayValue;
    cons.calctime = calcTime;
    cons.calculated = calculated;
    m_collCalcCons.Add(cons);
}

private double zeroIfNaN(double value) {
    if (double.IsNaN(value)) return 0.0;
    return value;
}

//<Exjobb>
/// <summary>
/// A class added to use in the parallel loop to know what counters to delete after the loop has finished.
/// </summary>
class ForDeletion {
    public CGWithFacts CG { get; set; }
    public DateTime date { get; set; }
}
//</Exjobb>

7.2.2 rcCalcV5_classes.cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Timers;
using System.Threading;
using System.Data.SqlClient;
using System.Data;

//using System.Data.SqlClient;
using System.Diagnostics;
using Momentum.RC.BusinessEntities.CounterObjects;
using Momentum.RC.BusinessEntities.Calculate;
using Momentum.RC.Bll;
using Momentum.RC.Bll.CounterObjects;
using Momentum.RC.Bll.Calculate;

namespace Momentum.RC.CalculationEngine.algoEval.rcCalc_version5
{
    #region inner classes

    public struct KeyCounterCG
    {
        public int counterId;
        public int cgId;
    }

    public struct KeyStruct
    {
        public int fromCGId;
        public int tariffId;
    }

    public struct ValueStruct
    {
        public double self_cons;
        public double dist_cons;

        public double self_cons_norm;
        public double dist_cons_norm;
    }

    public struct Value2Struct
    {
        public double self_cons;
        public double dist_cons;

        public double self_cons_norm;
        public double dist_cons_norm;
    }

    public class Period
    {
        public DateTime startDate;
        public DateTime stopDate;
    }

    /// <summary>
    /// CG and divider () information (used when calculate consumption on a daily basis)
    /// </summary>
    class config_entry
    {
        public int _child_cgId;
        public CGTYPE_ENUM _cgtype;
public enum CGTYPE_ENUM
{
    // Note that these Id's must match with the RC database table cgType
    CG = 1, // Anläggnign
    //CG_SUB = 2, // Fördelare - undermätare (kopplad), (um)
    //CG_DIFF = 3, // Fördelare - ? differensfördelare, (\)
    //CG_M2 = 4, // Fördelare - areafördelare, (m2)
    //CG_STATIC = 5, // Fördelare - procentfördelare, (%)  
    //CG_SUB_DISCO = 6, // Fördelare - fristående undermätare (ej kopplad), (?)
    //CG_ALGO = 7 // Fördelare - algorithmfördelare, (algo) eller (f(x))
}

public config_entry(int child_cgid, CGTYPE_ENUM cgtype)
{
    _child_cgid = child_cgid;
    _cgtype = cgtype;
}

/// <summary>
/// Information about a CG (date independent)
/// </summary>
class CGProperty
{
    public enum CGTYPE_ENUM
    {
        // Note that these Id's must match with the RC database table cgType
        CG = 1,
        //CG_SUB = 2,
        //CG_DIFF = 3,
        //CG_M2 = 4,
        //CG_STATIC = 5,
        //CG_SUB_DISCO = 6,
        //CG_ALGO = 7
    }

    protected int _clientId;
    protected int _cgId;
    protected int _connectedToCgId;
    protected CGTYPE_ENUM _cgType;

    protected DateTime _cgDateStart;
    protected DateTime _cgDateStop;

    protected Dictionary<int, int> _counterIds = null;

    public CGProperty(int clientId, int cgId, int connectedToCgId, CGTYPE_ENUM cgType,
          DateTime cgDateStart, DateTime cgDateStop)
    {
        _clientId = clientId;
        _cgId = cgId;
        _connectedToCgId = connectedToCgId;
        _cgType = cgType;
        _cgDateStart = cgDateStart;
        _cgDateStop = cgDateStop;
        _counterIds = new Dictionary<int, int>();
    }

    public int clientId
    {
        get { return _clientId; }
    }
}
public int cgId
{
    get { return _cgId; }
}
public int cgIdParent
{
    get { return _connectedToCgId; }
}
public CGTYPE_ENUM cgType
{
    get { return _cgType; }
}
public DateTime cgDateStart
{
    get { return _cgDateStart; }
    set { cgDateStart = value; }
}
public DateTime cgDateStop
{
    get { return _cgDateStop; }
    set { cgDateStop = value; }
}
public Dictionary<int, int> CounterIds
{
    get { return _counterIds; }
    set { _counterIds = value; }
}

class factDayCons_data
{
    protected double _cons;
    protected double _corr_cons;
    protected double _distributed_cons;
    protected DateTime _calctimestamp;

    public factDayCons_data()
    {
    }

    public factDayCons_data(double cons, double corr_cons, DateTime calctimestamp) // ,
    double cons_norm, double cons_warmwaterPart, double cons_energy, double cons_energy_norm,
    double cons_energy_warmwaterPart)
    {
        _cons = cons;
        _corr_cons = corr_cons;
        _calctimestamp = calctimestamp;
    }

    public double Cons
    {
        get { return _cons; }
        set { _cons = value; }
    }

    public double Distributed_Cons
    {
        get { return _distributed_cons; }
        set { _distributed_cons = value; }
    }

    public double Correction_Cons
    {
        get { return _corr_cons; }
    }
```csharp
public DateTime CalcTimestamp
{
    get { return _calctimestamp; }
    set { _calctimestamp = value; }
}

class CounterWithFacts : factDayCons_data
{
    protected int _clientId;
    protected int _counterId;

    public Guid guid = Guid.NewGuid();

    public CounterWithFacts(int clientId, int counterId) :
        base() // cons, cons_norm, cons_warmwaterPart, cons_energy, cons_energy_norm, cons_energy_warmwaterPart
    {
        _clientId = clientId;
        _counterId = counterId;
    }

    // This constructor make copies the data from the passed in Counter object
    public CounterWithFacts(CounterWithFacts c) :
    {
        _clientId = c.ClientId;
        _counterId = c.CounterId;
    }

    public int ClientId
    {
        get { return _clientId; }
    }

    public int CounterId
    {
        get { return _counterId; }
    }
}

class CGWithFacts : factDayCons_data
{
    public enum CGTYPE_ENUM
    {
        // Note that these Id's must match with the RC database table cgType
        CG = 1,
        //CG_SUB = 2,
        //CG_DDIFF = 3,
        //CG_M2 = 4,
        //CG_STATIC = 5,
        //CG_SUB_DISCO = 6,
        //CG_ALGO = 7
    }

    protected int _clientId;
    protected int _cgId;
```
protected int _connectedToCgId;
protected CGTYPE_ENUM _cgType;

protected int _WWP_inheritedFromCgId = -1;
protected DateTime? _WWP_inheritedFromCgId_dateStart = null;

//protected double _correction;
//<Exjobb> Changed to ConcurrentDictionary
ConcurrentDictionary<int, CounterWithFacts> _counter_dict = null;
public CGWithFacts(int clientId, int cgId, int connectedToCgId, CGTYPE_ENUM cgType, double cons, ConcurrentDictionary<int, CounterWithFacts> counter_dict, int wwp_inheritedFromCgId, DateTime? wwp_inheritedFromCgId_dateStart)
//</Exjobb>
{
    _clientId = clientId;
    _cgId = cgId;
    _connectedToCgId = connectedToCgId;
    _cgType = cgType;
    _cons = cons;
    _counter_dict = counter_dict;

    _WWP_inheritedFromCgId = wwp_inheritedFromCgId;
    _WWP_inheritedFromCgId_dateStart = wwp_inheritedFromCgId_dateStart;
}

public CGWithFacts(int clientId, int cgId, int connectedToCgId, int cgTypeId, double cons, ConcurrentDictionary<int, CounterWithFacts> counter_dict, int wwp_inheritedFromCgId, DateTime? wwp_inheritedFromCgId_dateStart)
{
    _clientId = clientId;
    _cgId = cgId;
    _connectedToCgId = cgIdParent;
    _cgType = (CGTYPE_ENUM)cgTypeId;

    _cons = cons;
    _counter_dict = counter_dict;

    _WWP_inheritedFromCgId = wwp_inheritedFromCgId;
    _WWP_inheritedFromCgId_dateStart = wwp_inheritedFromCgId_dateStart;
}

public int clientId
{
    get { return _clientId; }
}

public int cgId
{
    get { return _cgId; }
}

public int cgIdParent
{
    get { return _connectedToCgId; }
}

public CGTYPE_ENUM cgType
{
    get { return _cgType; }
}

//<Exjobb> Changed to ConcurrentDictionary
public ConcurrentDictionary<int, CounterWithFacts> Counters
//</Exjobb>
{
    get { return _counter_dict; }
}

public int wwp_inheritedFromCgId
{
    get { return _WWP_inheritedFromCgId; }
}

public DateTime? wwp_inheritedFromCgId_dateStart
{
    get { return _WWP_inheritedFromCgId_dateStart; }
}

} #endregion