Chapter 4 - The .NET Parallel Toolkit

In light of everything we have presented in the previous chapters, one can conclude that multi-core programming is no easy task. In fact, without a solid training in multi-threading, it can prove quite tedious and error prone. Errors such as data races (the concurrent modification of shared variables by several threads), data dependencies or oversubscription are frequent, especially for beginners in this field. Additionally, the non-deterministic nature of the execution of multi-threaded applications makes debugging harder, because some problems only appear in certain instruction interleaving scenarios.

Even if we do overcome these problems and our parallel program works correctly, we are still faced with the task of actually getting a better execution speed on multiple cores, as well as maintaining this trend as the number of cores increases. Therefore, other important traits of our code, besides correctness, must be load-balancing and scalability. As if these were not enough, the presence of dedicated caches for each core, as well as some compiler optimizations introduce an entire new host of problems related to inconsistent memory read / write operations.

The large number of problems that arise from programming in parallel has naturally prompted the appearance of tools to aid parallel programmers in their effort to create correct and efficient software. Such tools include Intel's VTune Performance Analyzer and Parallel Studio, or various other IDE integrated tools, such as the thread debugger present in Microsoft's Visual Studio 2010.

Additionally, parallel libraries have also begun to appear, relieving the programmer of managing a large portion of the thread management code. Such libraries include both older ones that have been continuously improved, like openMP or MPI, as well as very new ones, like the .NET Task Parallel Library, or Intel's TBB. In this case however, there is still continuous debate about hand threading (where all the details of threading can be controller, but errors can easily appear) vs. libraries (where threading is much easier, but also less flexible and the finer details are obscured from the programmer).

The .NET Parallel Toolkit is a flexible suite of three applications that attempt to help with both discovering potential threading errors (loop dependencies, data races, unwanted nested parallelism), generating working parallel code (that can be later modified) and evaluating the load-balance and scalability of a multi-threaded application.

The Parallel Advisor component is tasked with statically analyzing your source code and attempting to determine loop dependencies in one-dimensional arrays, which are of the form array[i - x], where i is the iteration variable of a for loop, potential data races (modification of variables that are not local to the scope of the loop we are trying to parallelize) or unwanted nested parallelism (such as creation of threads or queuing tasks in a thread pool) that may hurt performance.

The Parallel Converter component takes your source code and transforms the areas marked for parallelization (more on this later) into .NET parallel code, using the constructs prezented in Chapter 3. It can convert for loops (with certain restrictions), while loops and independent tasks and it can also introduce critical sections, atomic operations and memory barriers.

Finally, the Thread Profiler can perform timed sampling of a multithreaded executable and present you with statistics on the minimum, average and maximum CPU utilization of each thread that the application creates. This helps you identify load-balancing issues.

In the next sections, we will take a closer look at the pragmas behind this tool's parallelization effort and see how we can use them to obtain good results.

4.1 Pragmas for code evaluation and parallelism generation in the .NET Parallel Toolkit

In order to use the Parallel Toolkit to scan and convert your source code, you must first think about which sections of your are worth and can be parallelized. After this initial consideration, you can mark the discovered source code portions with special comments, inspired by openMP pragmas (which we described in detail in Chapter 2). In fact, this parallel micro-language implements a subset of openMP's pragmas and clauses and even extends some of them with new features. We will describe them in detail in the following subsections.

4.1.1 The parallel for pragma

The parallel for pragma can be used to parallelize a for loop by placing the following special comment on the line directly above it:

```
//@ parallel for schedule [chunk-size] [nosync]
```

The schedule clause is not optional and must be one of the following:

static	When present without the chunk-size parameter, it simply distributes the iterations of the for loop to the number of created threads in order, attempting to maintain an equal number. For example, 15 iterations will be distributed to 4 threads as (1, 2, 3, 4) - (5, 6, 7, 8) - (9, 10, 11, 12) - (13, 14, 15). In case the *chunk-size* parameter is present, each thread gets a chunk of *chunk-size* iterations, then another one, in a circular patterns, until there are no more iterations to distribute. For example, with chunk-size = 2, 15 iterations will be distributed to 4 threads as (1, 2, 9, 10) - (3, 4, 11, 12) - (5, 6, 13, 14) - (7, 8, 15). This scheduling policy is only good when the time to execute each iteration is the same.
dynamic	The dynamic schedule clause sets up a task queue which is filled with the iterations in order. Each thread goes to the queue, gets an iteration, and when it has finished executing it, it takes another one and so on. If the *chunk-size* parameter is specified, then each task in the queue consists of *chunk-size* iterations, instead of just one. This strategy is useful for load-balance when execution time varies among iterations, but inquired extra overhead due to the shared queue.

The optional *nosync* clause specifies that the implicit synchronization barrier for the threads executing the for loop should not be generated.

If you attempt to write a parallel for pragma that contains errors, the Parallel Advisor will give you a fatal error and ignore it.

Not all for loops are candidates for parallelization; only those that conform to the following restrictions:

```
for((int / long) i = initialValue; i < finalValue; i++) { statements }
```

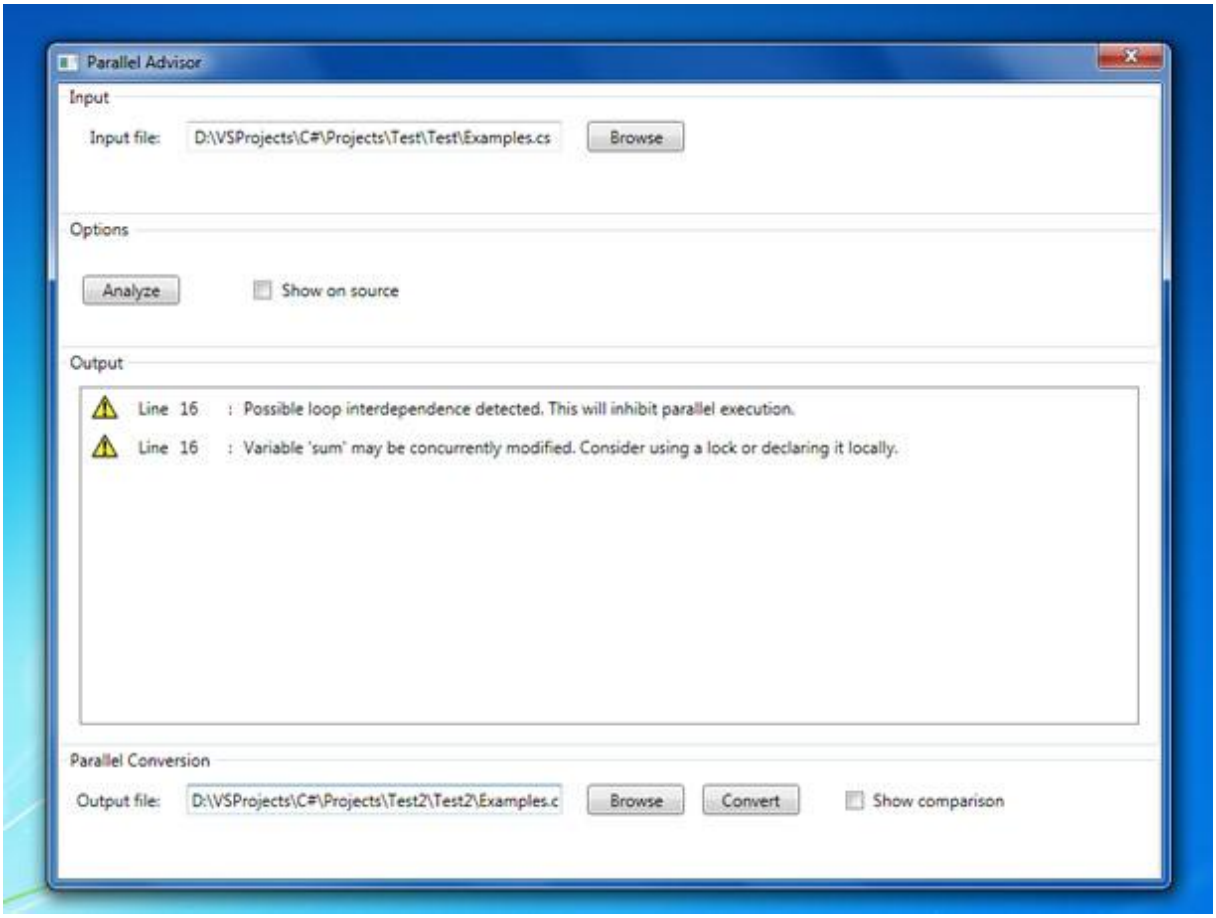
Therefore, the same iteration variable must be assigned to, compared and incremented; this variable must be of type int or long. The initialization part of the for must be an equality assignment, the condition part must be a less than comparison, and the step must be and increment. Finally, the statements must be enclosed in curly braces, even if only one is present. If you target a for loop that does not conform to these rules, you will get a fatal error and the loop will be ignored.

Once we have marked a for loop for parallelization, we may analyze the source code with Parallel Toolkit's Advisor component. This tool will scan for one-dimensional array interdependencies, data races and nested parallelism and present you with a list of warnings. After we have reviewed and potentially fixed the problems, we may go ahead and use the code conversion component to generate .NET threaded code.

Here is an example of a for loop parallelization that generates warnings:

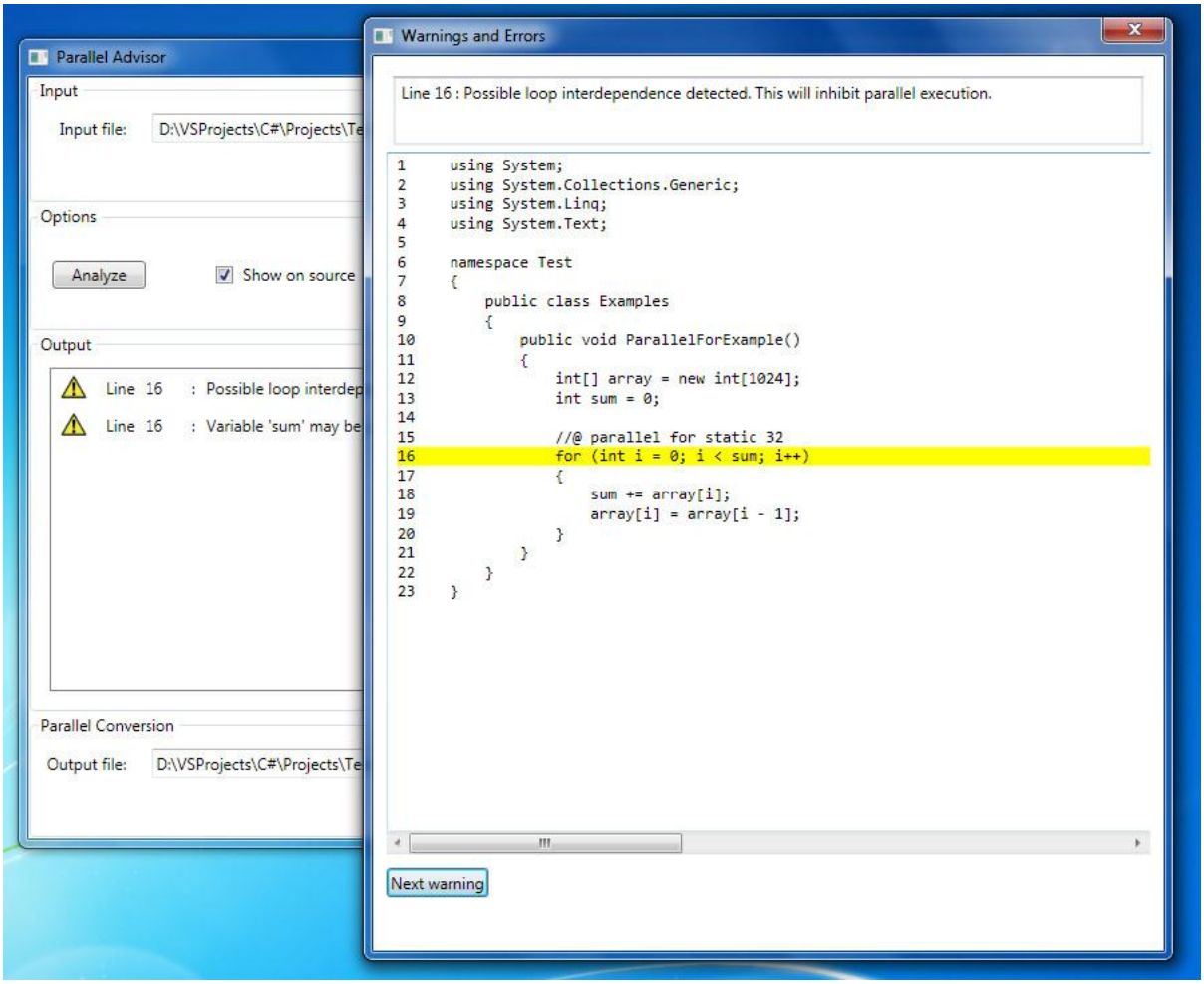
```
int sum = 0;
//@ parallel for static 32
for(int i = 0; i < 100; i++)
{
	sum += array[i];
	array[i] = array[i - 1];
}
```

Here are the warnings that the Parallel Advisor produces:



Screenshot: Warnings produced by the Parallel Advisor component.

We may also see these warnings more clearly marked on the source code by checking the "Show on source" checkbox:



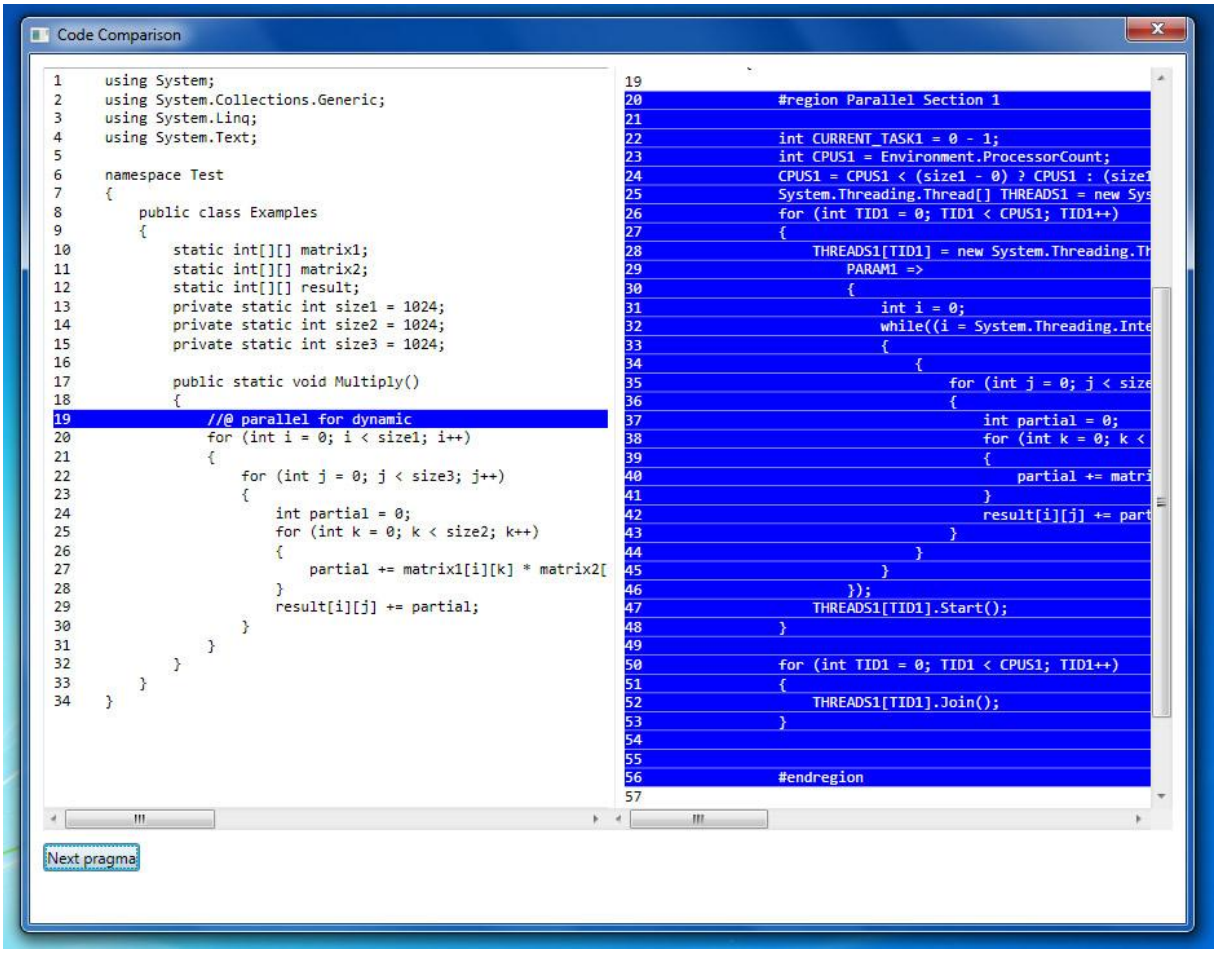
Screenshot: Each warning is marked on the source code.

These warnings may be ignored however, at our own risk, and we may proceed with converting the code to parallel code. Let's take another example, this time of a correctly parallelized for loop, the matrix multiplication algorithm:

```
//@ parallel for dynamic
for (int i = 0; i < size1; i++)
{
    for (int j = 0; j < size3; j++)
    {
        int partial = 0;
        for (int k = 0; k < size2; k++)
        {
            partial += matrix1[i][k] * matrix2[k][j];
        }
        result[i][j] += partial;
    }
}
```

This is a classic example of an algorithm that can be parallelized using a data decomposition pattern: since the iterations of the outer loop may be executed in any order, we can simply rewrite this for loop in parallel, using whatever scheduling strategy we choose.

By selecting an output file and clicking on "Convert", the code conversion tool will produce .NET parallel code. Optionally, checking the "Show comparison"checkbox will present us with a comparison between the initial code and the final code:



Screenshot: Code comparison: initial code vs. final code.

The actual code generated in this case is presented below. Notice that the converter will neatly place your code inside a region, so that you may collapse it.

```
#region Parallel Section 1

int CURRENT_TASK1 = 0 - 1;
int CPUS1 = Environment.ProcessorCount;
CPUS1 = CPUS1 < (size1 - 0) ? CPUS1 : (size1 - 0);
System.Threading.Thread[] THREADS1 = new System.Threading.Thread[CPUS1];
for (int TID1 = 0; TID1 < CPUS1; TID1++)
{
    THREADS1[TID1] = new System.Threading.Thread(
        PARAM1 =>
        {
            int i = 0;
            while((i = System.Threading.Interlocked.Increment(ref CURRENT_TASK1)) <
                    size1)
            {
                {
                    for (int j = 0; j < size3; j++)
                    {
                        int partial = 0;
                        for (int k = 0; k < size2; k++)
                        {
                            partial += matrix1[i][k] * matrix2[k][j];
                        }
                        result[i][j] += partial;
                    }
```

```
                }
            }
        });
    THREADS1[TID1].Start();
}

for (int TID1 = 0; TID1 < CPUS1; TID1++)
{
    THREADS1[TID1].Join();
}


#endregion
```

Obviously, the converted code will contain an extra set of variables for the required parallelization operations: getting the number of available cpus, creating threads, setting up the task queue, etc. The list of generated variables and their types can be found in the following table:

Variable name	Description
int CPUS[index]	The number of available cores.
int TID[index]	The id of the current thread outside the thread code.
Thread[] THREADS[index]	The array that holds the team of threads.
object PARAM[index]	Thread lambda expression parameter.
int ID[index]	The id of the current thread inside the thread code.
int INDEX[index]	Iteration variable over the number of cores.
int / long REMAINDER[index]	The remainder obtained by dividing the total iteration count to the number of available cores.
int / long START[index]	The start index from which the current thread begins computations.
int / long END[index]	The end index where the current thread stops computations.
int / long CURRENT_TASK[index]	The current task processed by each thread in a dynamic parallel for loop.
int GAP[index]	The chunk-size, in case it is provided.

where index is a unique number assigned to each parallel for loop incrementally.

4.1.2 The parallel while pragma

The parallel while pragma, can be used to parallelize certain while loops. Since a while loop may not have a predictable ending point, we cannot offer a coherent way to distribute iterations to threads. Therefore, each thread created will execute the entire while loop and the actual ending condition is left up to the programmer. The general idea is that the while exit condition depends on some variable that is modified inside the loop. Hence, if we keep this variable shared and operate on it atomically inside each thread, the exit condition of the loop will occur for all the threads simultaneously, causing them to exit. We will follow this up with an example shortly.

The general structure of the parallel while pragma is:

```
//@ parallel while [nosync]
```

Specifying the *nosync* flag will not generate the implicit synchronization barrier at the end of the while scope.

When encountering this pragma, the Parallel Advisor component will scan for data races and nested parallelism and issue warnings accordingly.

Let's consider the example of finding the first N prime numbers. A brute force approach will be to have a while loop that starts with a current value of 3 and checks the if the current value is prime, then increments it by 2 (since except for the number 2, all other primes are odd numbers). If a prime is found, N is decremented. The while loop stops when N reaches 0. The code would look like this:

```
public static void FindPrimes(int primeCount)
{
    int currentNo = 3;
    primeCount--; // we know 2 is prime.

    //@ parallel while
    while (primeCount > 0)
    {
        bool isPrime = true;
        for (int d = 2; d < currentNo / 2; d++)
        {
            if (currentNo % d == 0)
            {
                isPrime = false;
                break;
            }
        }

        if (isPrime)
        {
            //@ parallel atomic
            primeCount--;
            // print or store currentNo.
        }

        //@ parallel atomic
        currentNo += 2;
    }
}
```

Notice that in order to parallelize this while loop, we have placed parallel while pragma above it. Also, since we are forced to operate atomically on primeCount and currentNo (to keep track of the tasks), parallel atomic pragmas have been inserted to make sure these variables are updated atomically. The code generated look like this:

```
public static void FindPrimes(int primeCount)
{
    int currentNo = 3;
    primeCount--; // we know 2 is prime.


    #region Parallel Section 1

    int CPUS1 = Environment.ProcessorCount;
    int TASKS1 = CPUS1;
    System.Threading.Thread[] THREADS1 = new System.Threading.Thread[CPUS1];
```

```
    for (int TID1 = 0; TID1 < CPUS1; TID1++)
    {
        THREADS1[TID1] = new System.Threading.Thread(
            PARAM1 =>
            {
                while (primeCount > 0)
                {
                    bool isPrime = true;
                    for (int d = 2; d < currentNo / 2; d++)
                    {
                        if (currentNo % d == 0)
                        {
                            isPrime = false;
                            break;
                        }
                    }

                    if (isPrime)
                    {

                        #region Parallel Section 2

                        System.Threading.Interlocked.Decrement(ref primeCount);

                        #endregion


                        // print or store currentNo.
                    }


                    #region Parallel Section 3

                    System.Threading.Interlocked.Add(ref currentNo, 2);

                    #endregion


                }
            });
            THREADS1[TID1].Start();
    }

    for (int TID1 = 0; TID1 < CPUS1; TID1++)
    {
        THREADS1[TID1].Join();
    }


    #endregion
}
```

Note that openMP does not have a similar construct.

4.1.3 The parallel tasks pragma

Often in our programs, we can find sections of code that are independent and can be executed in parallel easily. We will refer to these code sections as tasks. In case our code contains such independent tasks, the .NET Parallel Toolkit offers the parallel tasks pragma. The general form is the following:

```
//@ parallel tasks [pooled] [nosync]

[//@ parallel task [nosync]
{
      // task scope
}

[//@ parallel task [nosync]
{
      // task scope
}]]

//@ end tasks
```

The parallel tasks pragma must have a corresponding end tasks pragma. The threads created to execute the tasks are implicitly joined at the end tasks pragma, unless the *nosync* flag is specified. Several tasks may be specified between the start and end pragmas.

Unique to the .NET Parallel Toolkit is the fact that we can specify a *pooled* flag to the parallel tasks pragma. If the flag is off, then a thread is created for each task encountered, even if the number of tasks exceeds the number of available cores. If the flag is on, then only a number of threads equal to the number of available cores is created, and they will execute the tasks dynamically, from a queue.

Another unique feature is selective synchronization for each task. Hence, tasks that have the *nosync* flag on will not be joined at the end tasks pragma. Note that this only works in non-pooled mode, where a 1:1 mapping exists between tasks and threads.

The Parallel Advisor component will scan each task scope for data races and nested parallelism.

A good example of using this construct is the mergesort algorithm that has two independent recursive calls that may be executed in parallel. The mergesort code may be as follows:

```
static int[] inputArray;
static int[] tempArray;
static int cpus = Environment.ProcessorCount;

static void merge(int lo, int m, int hi)
{
    int i, j, k;

    for (i = lo; i <= hi; i++)
    {
        tempArray[i] = inputArray[i];
    }

    i = lo;
    j = m + 1;
    k = lo;

    while (i <= m && j <= hi)
    {
        if (tempArray[i] < tempArray[j])
        {
            inputArray[k++] = tempArray[i++];
        }
        else
        {
            inputArray[k++] = tempArray[j++];
        }
```

```
    }


    while (i <= m)
    {
        inputArray[k++] = tempArray[i++];
    }
}

static void mergesort(int lo, int hi)
{
    if (lo < hi)
    {
        int m = (lo + hi) / 2;
        mergesort(lo, m);
        mergesort(m + 1, hi);
        merge(lo, m, hi);
    }
}

static void parallel_mergesort(int lo, int hi, int level)
{
    if (lo < hi)
    {
        int m = (lo + hi) / 2;
        if (Math.Pow(2, level) <= cpus)
        {
            //@ parallel tasks

            //@ parallel task
            {
                parallel_mergesort(lo, m, level + 1);
            }

            //@ parallel task
            {
                parallel_mergesort(m + 1, hi, level + 1);
            }

            //@ end tasks
        }
        else
        {
            mergesort(lo, m);
            mergesort(m + 1, hi);
        }
        merge(lo, m, hi);
    }
}
```

Notice that we have split the mergesort method into two versions, in order to parallelize it efficiently. We cannot have the two recursive calls execute in parallel for each level of recursion, as it would oversubscribe the system. Hence, the second mergesort method only goes until the number of recursive calls is equals the number of available cores, then calls the sequential version. The generated code for the parallel_mergesort method is this:

```
static void parallel_mergesort(int lo, int hi, int level)
{
    if (lo < hi)
    {
        int m = (lo + hi) / 2;
```

```
        if (Math.Pow(2, level) <= cpus)
        {

            #region Parallel Section 1

            int TASKCOUNT1 = 2;
            System.Threading.ParameterizedThreadStart[] TASKLIST1 = new
System.Threading.ParameterizedThreadStart[TASKCOUNT1];
            bool[] SYNCLIST1 = new bool[TASKCOUNT1];


            TASKLIST1[0] =
                PARAM1 =>
                {
                    parallel_mergesort(lo, m, level + 1);
                };
            SYNCLIST1[0] = true;


            TASKLIST1[1] =
                PARAM1 =>
                {
                    parallel_mergesort(m + 1, hi, level + 1);
                };
            SYNCLIST1[1] = true;


            System.Threading.Thread[] THREADS1 = new
System.Threading.Thread[TASKCOUNT1];
            for(int TID1 = 0; TID1 < TASKCOUNT1; TID1++)
            {
                THREADS1[TID1] = new System.Threading.Thread(TASKLIST1[TID1]);
                THREADS1[TID1].Start();
            }

            for(int TID1 = 0; TID1 < TASKCOUNT1; TID1++)
            {
                if(SYNCLIST1[TID1] == true)
                {
                    THREADS1[TID1].Join();
                }
            }


            #endregion

        }
        else
        {
            mergesort(lo, m);
            mergesort(m + 1, hi);
        }
        merge(lo, m, hi);
    }
}
```

4.1.4 Synchronization constructs

The .NET Parallel Toolkit offers two constructs that may used to generate synchronization code: the parallel lock and the parallel atomic pragmas.

The general form of the parallel lock statement is as follows:

```
//@ parallel lock <synchronization variable>
{
        // synchronized code
}
```

This construct generates a critical section in the form of C#'s lock statement (described in Chapter 3), applied to the given synchronization variable. Note that this variable is assumed to exist and will not be generated for you. This is because the code generator cannot assume the scope in which you want the variable to exist (it may even be a cross-class variable, in which case it may not even be created in the same file as the annotated code).

This construct is useful for serializing access to a certain portion of code, such that the threads that encounter it will not execute that portion of code in parallel. More on critical sections can be found in Chapters 2 and 3.

Updating a single numeric variable is an operation that is too fast for the overhead incurred by entering and leaving a critical section, so a non-blocking locking mechanism is a better idea. For this situation, we may use the parallel atomic pragma:

```
//@ parallel atomic
<single instruction that updates a long or int variable>
```

The update instruction may be performed used the following operators: ++, --, += or -= on a long or int variable. Depending on which operator was used, the operation is converted to one of the following method calls (which are covered in detail in Chapter 3):

- Interlocked.Increment for ++;
- Interlocked.Decrement for --;
- Interlocked.Add for += and -=;

Finally, as explained in Chapter 3, special care must be taken when reads and writes of shared variables are done by multiple threads, as a wide array of optimizations such as instruction reordering or data caching may break your code on multi-core machines. In such cases, it is useful to be able to introduce special instructions called memory barriers. The .NET Parallel Toolkit offers us the parallel membarrier pragma:

```
//@ parallel membarrier
```

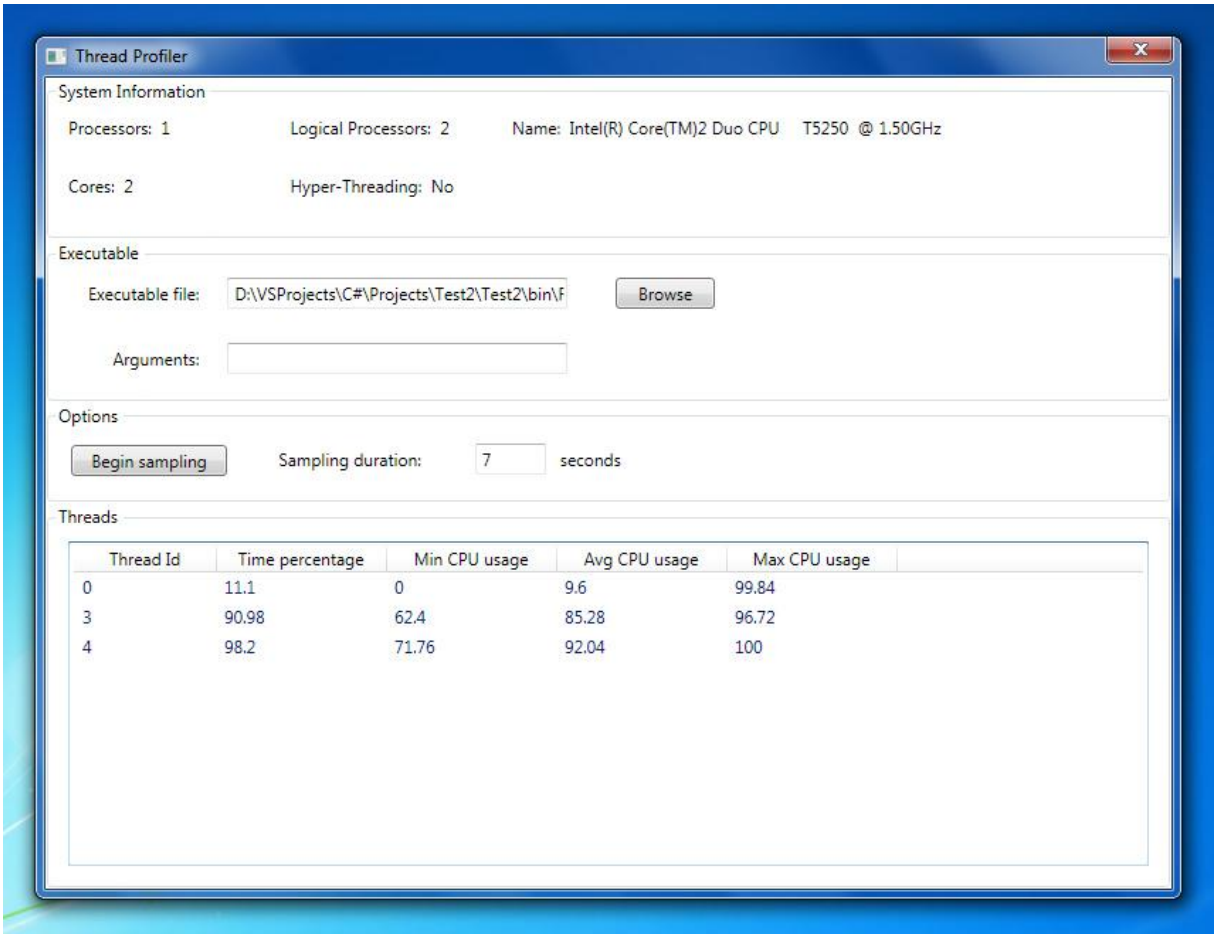
which is converted to a call to Thread.MemoryBarrier. The introduction of such an instruction guarantees that no read / write operations are reordered relative to it and also that all the values of shared variables are read from main memory, rather than each core's private cache.

4.2 The Thread Profiler component

Chapter 2 explained the importance of load-balance for the efficiency of a parallel application. Sharing the workload equally across all threads with minimal work sharing overheads results in fewer cycles wasted with idle threads not advancing the computation, and thereby leads to improved performance.

Therefore, we considered that an essential feature of any parallel toolkit would be an application that allows us to perform a sampling on the execution of a multithreaded application and determine how well that application is load-balanced. To this end, the .NET Parallel Toolkit offers the independent Thread Profiler component.

This component is very versatile in that it can be used to asses the load-balance of any executable, not just those produced using the other tools in the suite. The user interface is friendly and easy to use, as you can see in the following diagram:



Screenshot: Thread Profiler window.

As we can see, the Thread Profiler foremost provides processor information, detecting the processor model, number of physical processors, number of cores, number of logical processors (which may differ from the number of cores in case we have a processor with HT Technology) and whether or not we have HT enabled.

Next, we may select an executable file from the hard drive and optionally provide command line arguments.

To begin execution sampling, we must enter a positive integer in the "Sampling duration" textbox and click on "Begin Sampling". The selected executable is then run in a separate process. The sampling stops when either the specified number of seconds has elapsed, or the process stops by itself, whichever occurs first. When this happens, the "Threads" table is populated with CPU usage information for each thread spawned by the sampled application.

In the above example, we ran a 7 second sampling on the execution of a program that performs a parallel matrix multiplication. As we can see, besides the master thread (thread id 0), two additional worker threads have been spawned, that worked 85% and 92% on average, respectively. This is pretty good, but not optimal, as it indicates that one core was idle for 15% of the time, while the other one for 8% of the time (this may also indicate, however, that other threads in the system ran on the processor, which can occur if we are not using a dedicated machine for our tests).

Typically, an average usage of over 90% for each thread indicates a good load-balance of our application. Causes for load-imbalance may be excessive synchronization (threads waiting to acquire locks), memory fetches, I/O etc. so be sure to minimize these if you want good performance.

4.3 Future research

As we have seen, the .NET Parallel Toolkit is very helpful when writing .NET parallel code, by helping us work in a declarative manner and pointing out several flaws that may occur in our parallel design.

The subject of multi-core programming is vast and provides plenty of possible research directions, especially regarding tools.

One of these may be the complete adaptation of openMP for the C# language, together with compiler support. That is, instead of generating parallel C# code, we should be able to simply add pragmas to our source code and the compiler should generate parallel IL (intermediate language) directly, without programmer intervention.

Whether or not we implement the previous major improvement, there is still room for a lot of feature improvements: the Parallel Advisor component could be vastly extended to perform project-level dependency checks in order to detect cross-class data dependencies. Additionally, whether more compact parallel code may be generated is up to debate.

Another component can be added, which is tasked with application runtime sampling and suggesting performance improvements related to: cache performance, memory bandwidth usage, etc.

Furthermore, the entire tool suite can be made into a web application, so the users may simply work from the browser. A many-core server may provide an on-line service for evaluating the load-balance of applications written by programmers who do not possess such a machine (for example, my application works well on a dual-core machine, but I'd like to see how it performs on an 8 or 16-core machine).