Parallel Programming Based on Microsoft.NET TPL

Zuomin Luo, Quanfa Zheng[1], Xinhong Hei[2]
School of Computer Science and Engineering
Xi'an University of Technology
Xi'an, China
zqf0687@hotmail.com[1]
heixinhong@xaut.edu.cn[2]

Nasser Giacaman
Department of Electrical and Computer Engineering
The University of Auckland
Auckland, New Zealand
n.giacaman@auckland.ac.nz

***Abstract*—In order to reduce the complexity of traditional multithreaded parallel programming, this paper explores modern task-based parallel programming using the Microsoft.NET Task Parallel Library (TPL). Firstly, this paper utilizes the two main parallel programming models: Data Parallelism and Task Parallelism, which are supported by TPL. Then we employ two experiments to apply both models. Finally, the paper shows and analyses the performance of our applications. Through experiments we show that TPL's new task-based parallel programming approach can dramatically alleviate programmer burden and boost the performance of programs.**

Keywords-Parallel programming; TPL; Performance; Data parallelism; Task Parallelism

I. INTRODUCTION

Nowadays, multi-core microprocessors are ubiquitous. With multi-core processors, clock speeds are not increasing with newer hardware as much as in the past. However, common sequential programs barely benefit from multi-core hardware advances; programmers need to adapt their programming model to take advantage of multi-core processing units in order to get performance improvements. Traditionally, programmers focused on a multithreaded programming model to boost executing speed and resource utilization. There are some popular threaded shared memory programming models (such as Pthreads and OpenMP) and message passing programming models (such as Message Passing Interface (MPI)). These models actually provide powerful tools for parallel programming and have even become the de facto standard for their respective domain. But the classical shared memory model, the programming unit for parallelism is the thread, and it requires the programmer to create them, assign work to them, manage their existence, while also requiring knowledge of the underlying physical shared memory architecture (e.g. number of available processing cores) to achieve performance improvements.

Fortunately, the Microsoft .NET Framework 4.0 introduces a modern task-based programming model to express parallelism. Specifically, this includes the Task Parallel Library (TPL), Parallel LINQ and many supporting classes that make writing parallel program with C# simpler and easier than ever before. In this paper we mainly base on TPL to explore the parallel performance of a C# program and application. The basic unit of the TPL is the *task*, which lets the programmer focus primarily on the business logic of the problem being solved instead of on the mechanics of how it will get done.

This paper uses both data parallelism and task parallelism programming models supported by TPL with C#. Through two concrete experiments of matrix multiplication and image blender, the detailed process of design parallel procedure will be explained. Finally, the performance of the two parallel experiments implemented by TPL will be evaluated.

II. DATA PARALLELISM MODEL

Data parallelism refers to scenarios in which the same operation is performed concurrently on elements in a source collection or array. The .NET framework provides new constructs to achieve data parallelism by using *Parallel.For* and *Parallel.Foreach* constructs, instead of sequential *For* and *Foreach* loops. *For* and *Foreach* loops are cornerstones of .NET development, but both keywords create sequential loops where an iteration is not started until the previous iteration has completed. On the contrary, TPL contains support for parallel loops in which the iterations are performed with some degree of concurrency. Parallel loops create and manage *tasks* (i.e. the units of concurrency) on behalf of the user. They also provide some very useful convenience methods to coordinate those *tasks*. Many existing applications can be decomposed so that they can be executed in parallel. Take for example the following method for multiplying two matrices:

```
Void MatrixMult( int size, double [ , ] A,
                 double[ , ] B, double [ , ] C){
  for (int i = 0; i < size; i++)
    for( int j = 0; j < size; j++)
      for( int k = 0; k < size; k++)
        C[i , j] += A[i , k] * B[k , j];
}
```

In this example, the outer iterations are independent of each other and can potentially be done in parallel. A straightforward and traditional way to parallelize this algorithm would be to use size threads, where each thread would execute the two inner loops with its corresponding iteration index. But that would be expensive in terms of memory (since each thread needs a stack) and in terms of time (other additional overhead, such as operating system management of a large number of threads).

It is better to use a small pool of threads, and assign to them a set of lightweight *tasks* to execute, where a *task* is a finite CPU bound computation, like the body of a for loop. So instead of threads we should create *tasks*, each of them executing the two inner loops for its iteration index. Moreover, the *tasks* would be executed by n threads, where

Published by Atlantis Press, Paris, France.

n is typically the number of processors. Using tasks instead of threads has many benefits, not only are they more efficient, they also abstract away from the underlying hardware and the OS specific thread scheduler. Now we revisit above example that realized by TPL method:

```
Void ParMatrixMult( int size, double [ , ] A,
                    double[ , ] B, double [ , ] C){
  Parallel. for ( 0, size, option, i => {
      for( int j = 0; j < size; j++)
          for( int k = 0; k < size; k++)
            C[i , j] += A[i , k] * B[k , j];
  });
}
```

In section 4 we will compare the performance of sequential and parallel matrix computation.

III. TASK PARALLELISM MODEL

The Task Parallel Library (TPL) introduces the concept of *task*. Task parallelism is the process of running these *tasks* in parallel. A *task* is an independent unit of work, which runs with a program. The TPL utilizes the threads under the hood to execute these tasks in parallel. The design and number of threads to use is dynamically calculated by the runtime environment. When independent computations are started in different tasks, we use a model of task parallelism.

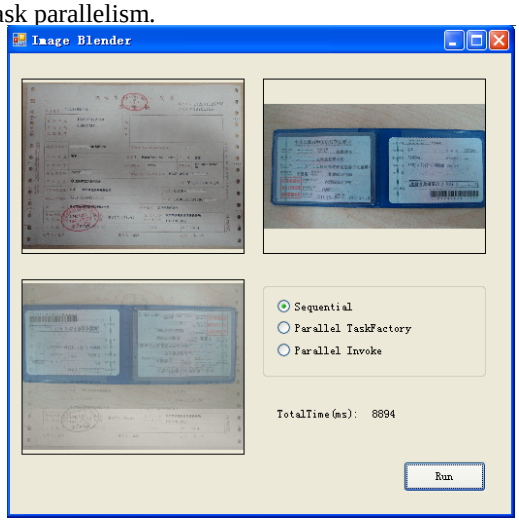

Figure 1. Screenshot of image blender application

The TPL provides *Parallel.Invoke* and *Task.Factory* to launch several methods in parallel. *Parallel.Invoke* is the simplest expression of the parallel task pattern. This method returns when all the tasks are finished. Similarly, the *StartNew* method of the *Task.Factory* class creates and schedules a new *task* that executes the delegate method that is provided as its arguments. In addition, we can wait for parallel tasks to complete by calling the *Task.WaitALL* method [3]. In the light of verifying task parallelism efficiency, we create an image blender application which processing two images in parallel. One image is to be gray processed and the other is to be rotated 180 degree (as Fig. 1 shows), so two images processing operations are performed on each of them and must be complete before the images can be blended.

As Fig. 1 shows, the left image is to be gray processed, while the right one is rotated 180 degrees. Then after both processing completed, the two images would be blended. The implementation of image blender by sequential and parallel *tasks* is given as following:

(1) Sequential code:

```
SetToGray( image1, layer1);
Rotate( image2, layer2);
Blend(layer1, layer2,Graphics blender);
```

(2) Parallel Task.Factory code:

```
Task gray = Task.Factory.StartNew(
      () => SetToGray(image1, layer1));
Task rotate = Task.Factory.StartNew(
      () => Rotate(image2, layer2));
Task.WaitAll(gray, rotate);
Blend(layer1, layer2,Graphics blender);
```

(3) Parallel.Invoke code:

```
Parallel.Invoke(
    ()= > SetToGray(image1, layer1),
    () => Rotate(image2, layer2));
Blend(layer1, layer2,Graphics blender);
```

And the next section will show the performance of the image blender application running on above three ways respectively.

IV. PERFORMANCE EVALUATION

We now check the performance of the two TPL parallel applications: matrix multiplication and image blender.

We firstly test the data parallelism performance of matrix multiplication. Test coefficient matrix is 1024 × 1024 and test results are as follows: Fig. 2 indicates corresponding relationship between execution time and number of processors running on sequential and parallel version of matrix multiplication. Fig. 3 shows the speedup of parallel matrix multiplication. As we can see from Fig. 2 and Fig. 3, matrix multiplication achieves expected linear speedup with increasing multi-core processing hardware resources. All experimental tests were run on a 2-socket quad-core Intel Xeon E5504 machine with 8GB RAM memory running Windows Server HPC Edition (64 bit).

In order to verify the latter application, we run it on multi-core system as well and record the execution time on different running mode. The results are given as Tab. 1.

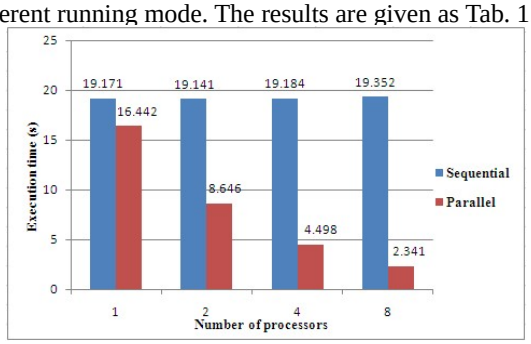

Figure 2. Execution time and number of processors

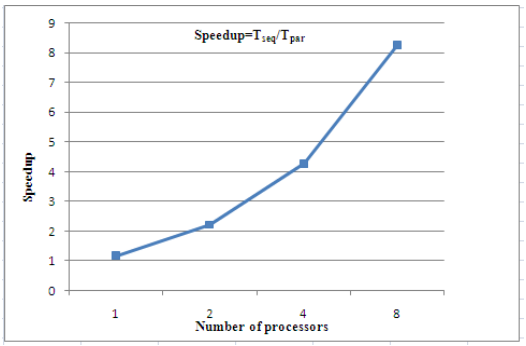

Figure 3. Speedup of parallel matrix multiplication

TABLE I. Execution time of image blender application running on multiple cores

Processor cores	Execution time (seconds)		
	Sequential	*Task Factory*	*Parallel Invoke*
2	9.122	6.226	6.240
4	9.156	6.512	6.637
8	9.213	6.856	6.831

From Tab. I, we detected that parallel running speedup over the sequential version would be nearly the same regardless of running on how many cores microprocessor. And two base task parallelism patterns (i.e. *Task.Factory* and *Parallel.Invoke*), get the similar performance advances. Some may doubt the parallel improved scalability using TPL, however, ignore what the *task* we have distributed and what part of task would be parallelized, which are issues need to be paid attention during our parallel programming.

Fig. 4 explains the above speedup we obtained by parallel execution flow. As we can see, *SetToGray* and *Rotate* methods need 5.932 seconds to run in parallel instead of 8.918 seconds in sequence. In this case, two tasks only take up two cores to execute, furthermore, burden on imbalanced workload. That is why we get the above speedup of this application. Above all, it is programmer responsibility to identify potential parallelism and achieve fully parallelized and perfectly scalable code.

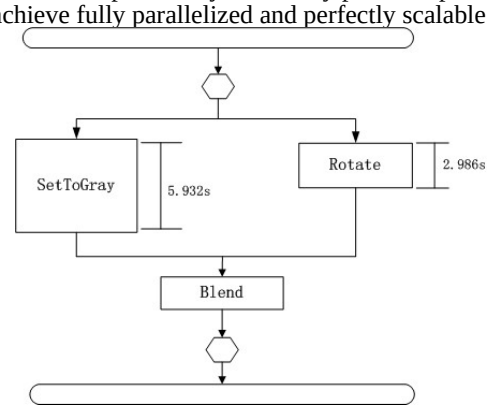

Figure 4. Parallel execution flow of image blender

V. CONCLUSION

Parallel programming is very challenging but excellent new task-based programming models offered by Microsoft.NET Framework and Visual Studio 2010 make it easier for us. In this paper we start with TPL, introduce its common parallel programming models and verify their efficiency through our two practical experiments. Although the performance improvement is specific to our test environment and applications, it shows clearly that parallel programming using TPL will raise the performance of application significantly and give the chance to programmer to exploit available multi-core resources.

ACKNOWLEDGMENT

The authors wish to acknowledge the support of the National Natural Science Foundation of China (No. 61100173) and fund of Shaanxi Province Education Department (No. 11JK1038), as well as Scientific Research Starting Foundation for Doctors of Xi'an University of Technology (No.116211105).

REFERENCES

[1] A. Freeman, Pro .NET 4 Parallel Programming in C#, Apress, 2010.

[2] G. C. Hillar, Professional Parallel Programming with C#: Master Parallel Extensions with .NET 4, Wrox, 2010.

[3] C. Campbell, R. Johnson, A. Miller and S. Toub, Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures, Microsoft Press, 2010.

[4] D. Leijen, W. Schulte and S. Burckhardt, "The Design of A Task Parallel Library," Proceeding of the 24th ACM SIGPLAN conference on object oriented programming systems languages and applications (OOPSLA), pp. 227–242, 2009.

[5] H. Vandierendonck and T. Mens, "Techniques and Tools for Parallelizing Software," IEEE Software, Vol. 29, No. 2, pp. 22–25, 2012.

[6] T. G. Mattson, B. A. Sanders and B. L. Massingill, Patterns for Parallel Programming, Addison-Wesley Professional, 2004.