The use of parallel extensions libraries for scientific and engineering calculations

Gennadiy Burlak[1], José Alberto Hernández Aguilar[2], René Santaolaya Salgado[3], Moisés González García[3]

[1]Centro de Investigación en Ingeniería y Ciencias Aplicadas, [2]Facultad de Ciencias, Universidad Autónoma del Estado de Morelos, Cuernavaca, Mor. Mexico.
[3]Departamento de Ciencias Computacionales, Centro Nacional de Investigación y Desarrollo Tecnológico CENIDET, Cuernavaca, Mor., México
gburlak@uaem.mx

Abstract. We studied the use of the library Microsoft Parallel Extensions to .NET Framework 3.5 for parallel calculations. We have developed a hierarchy of nesting classes having complicated internal structure and have made the bench tests (with graphic user interface (GUI)) not only for simple static cases but also for complex dynamic types. Our tests have shown high speedup of the library. The use of other libraries allowing calculations in parallel for various cross-platform applications is discussed also.

1 Introduction

Dual cores PCs have gradually become the standard in Universities. Quad core PCs are also getting closer, and PCs with greater number of CPUs/cores are also available. Modern scientific problems require large amounts of computational tasks which are well time consuming. Sometimes parallel/ distributed computing of such problems is of critical importance [1], [2]. Nowadays developers have access to PCs with several CPUs/cores; so there is a great task to use the whole computing power of such PCs in order to load all the cores to work effectively in parallel.

Recently was published the Parallel Extensions to .NET Framework 3.5 Community Technology Preview (CTP) that provides a managed programming model for data parallelism, task parallelism, and coordination on parallel hardware unified. Parallel Extensions makes it easier for developers to write programs that scale to take advantage of parallel hardware by providing improved performance as the numbers of cores and processors. Parallel Extensions provides library based support for introducing concurrency into applications written with .NET languages, including e.g. C#. In this release various samples of the library were used for time consuming problems calculating sequentially and in parallel was done (solve nqueen puzzle, sorting example, simple matrix multiplication, etc).

However in the samples mainly was used a data with simple numeric static structure. For instance, the product of random double matrixes with large size was sequentially and in parallel processed. For such configurations it was demonstrated that performance of in parallel calculations gives an essential speedup of calculations. However in engineering calculations frequently one meets more advanced problems in which it is necessary to use the dynamic data with complicated internal structure. Typical example is calculations with complex numbers, Fast Fourier transformations, various numeric transformations with complex matrixes and vectors. Such structures normally must serve as dynamically distributed objects rather than being static ones. For such situations in parallel calculations are of great practical importance. It is well known that among the popular programming languages only FORTRAN and PYTHON have in-built complex data type. In languages as C++ and C# such types have to be created by programmers.

However in languages C ++ and C# we meet other much more important feature: possibility to overload of the standard mathematical operators: addition, subtraction, multiplication and division that allows to extend considerably the meaning of such operators. The operator overloading allows constructing much more advanced numeric (and not only numeric) classes, such as a complex vectors and matrices having quite complicate behavior. However speedup of such structures in parallel mode still has been poorly considered, though it is a logical extension of previous investigation in this area.

In this Report we discuss the computations in parallel for C# that allows distributing the appropriate tasks effectively at all cores available in the system. We will take a very brief look at what is provided by Microsoft's in Microsoft Parallel Extensions to .NET Framework 3.5. The main aim is to discuss how to implement parallelism for dynamic hierarchies of classes with advanced internal structure. We developed and applied such hierarchy of the nesting classes (complex matrixes and complex vectors) as working structures at in parallel calculations and have performed the bench tests. Also we have compared our results with parallel computations for other library.

2 The structure of classes and working examples.

Fig.1 shows the structure of our nested classes for complex numbers, vectors and matrixes. One can see that structural complexity of such objects is much deeper with respect to a simple double type. For evaluation of real meaning of the MS parallel library it is important to investigate the speedup of calculation in parallel for such advanced structures.

In order to make bench tests of in parallel calculations we used MS VS C# 2008 Express Edition, see Fig.2. To handle advanced calculations we had to extend essentially the code of a testing program. The elaboration consists in the following. i) The program was added part of code that allow working with dynamic objects, and ii) the graphic user interface (GUI) has been created. The latter allow selecting the type of test, and also a desired dimension of the

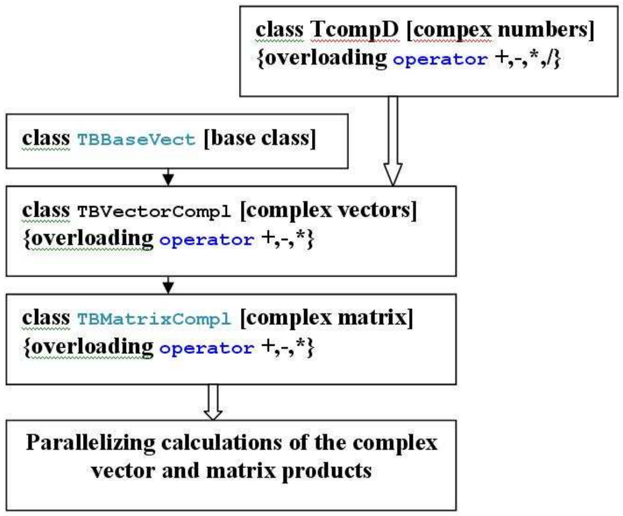

class TcompD [compex numbers]
{overloading operator +,-,*,/}
class TBBaseVect [base class]
class TBVectorCompl [complex vectors]
{overloading operator +,-,*}
class TBMatrixCompl [complex matrix]
{overloading operator +,-,*}
Parallelizing calculations of the complex vector and matrix products

Fig. 1. Logic diagram and structures of nested classes used in our calculations.

complex matrixes. Details of our calculations and GUI are depicted in Fig.3. We have performed the bench tests of library mainly for one of must frequently used in parallel libraries the cycle operator Parallel.For. We did it not only for static numeric data, but also for complex dynamic structures. Results of this work are shown in Fig. 3 and summarized in Table 1.

The following technique was used to create the parallel loops over iteration spaces. For example, let's parallelize the loop, where the iteration range is based on doubles (for further references see [3])

```
for(int i = 0; i<1000; i ++)
{
Process(i);
}
```

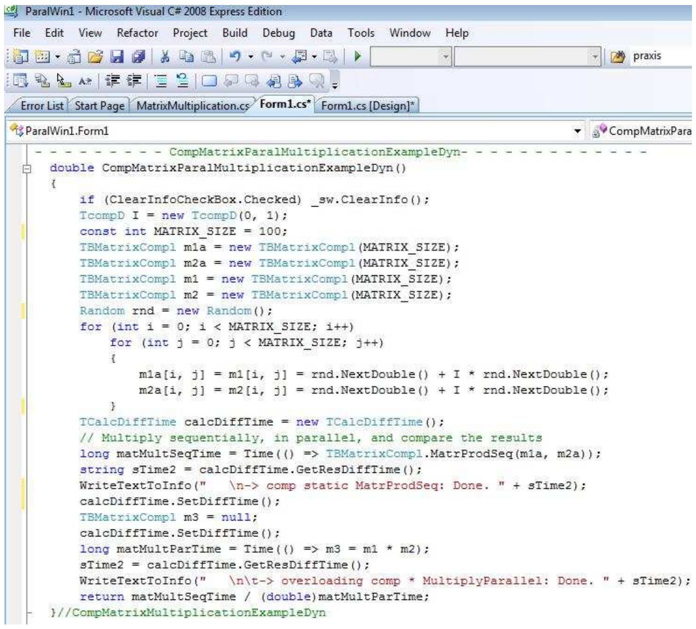
The CTP Parallel.For(,,) only contains overloads For where the iteration variable. As an example, the previously shown loop could be rewritten as:

```
Parallel.For(0, 1000, i =>  //using a lambda expression
{
    double d = i / 1000.0;
    Process(d);
});
```

The pseudocode for multiplication of dynamic complex matrices by means of sequential processing and parallel processing is shown below:

```
Double ParallelMultiplicationOfComplexMatrix(N)

Imaginary number I;
Integer MATRIX_SIZE = N

#Construction of dynamic matrices
MatrixComplex m1a = MatrizComplex(MATRIZ_SIZE)
MatrixComplex m2a = MatrizComplex(MATRIZ_SIZE)
MatrixComplex m1 = MatrizComplex(MATRIZ_SIZE)
MatrixComplex m2 = MatrizComplex(MATRIZ_SIZE)

# Fill Matrices with random Numbers
for i=0 to (MATRIZ_SIZE-1) with increments of 1
for j=0 to (MATRIZ_SIZE-1) with increments of 1
       m1a[i, j] = m1[i, j] = generate_rnd_number() + I * generate_rnd_number()
       m2a[i, j] = m2[i, j] = generate_rnd_number() + I * generate_rnd_number()

# Sequential product
Timer.start()
m3 = ProductSequential(m1a, m2a)
Timer.stop()
SequentialTime = Timer.getdifference()

m3 =null

# Parallel product (Lambda operator => )
Timer.start()
=> m3 = m1 * m2
Timer.stop()
ParallelTime = Timer.getdifference()

# Ratio calculation
ratio = SequentialTime/ParallelTime
return ratio
End ParallelMultiplicationOfComplexMatrix
```

Getting the full workload of multicore processors can be tricky because, in order for a program to make use of more than one core, it must divide its workload in such a way that it does not take more effort than the gains achieved by adding more cores. Most programming languages were written assuming just one processor would be working through the code sequentially, line by line [5]. Above example shows that is really easy to add parallel processing using functional programming and C# extensions.


ParalWin1 - Microsoft Visual C# 2008 Express Edition
File Edit View Refactor Project Build Debug Data Tools Window Help
praxis
Error List Start Page MatrixMultiplication.cs Form1.cs* Form1.cs [Design]*
ParalWin1.Form1
CompMatrixPara

```
- - - - - - - - - - CompMatrixParalMultiplicationExampleDyn- - - - - - - - - - - - - - -
double CompMatrixParalMultiplicationExampleDyn()
{
    if (ClearInfoCheckBox.Checked) _sw.ClearInfo();
    TcompD I = new TcompD(0, 1);
    const int MATRIX_SIZE = 100;
    TBMatrixCompl m1a = new TBMatrixCompl(MATRIX_SIZE);
    TBMatrixCompl m2a = new TBMatrixCompl(MATRIX_SIZE);
    TBMatrixCompl m1 = new TBMatrixCompl(MATRIX_SIZE);
    TBMatrixCompl m2 = new TBMatrixCompl(MATRIX_SIZE);
    Random rnd = new Random();
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++)
        {
            m1a[i, j] = m1[i, j] = rnd.NextDouble() + I * rnd.NextDouble();
            m2a[i, j] = m2[i, j] = rnd.NextDouble() + I * rnd.NextDouble();
        }
    TCalcDiffTime calcDiffTime = new TCalcDiffTime();
    // Multiply sequentially, in parallel, and compare the results
    long matMultSeqTime = Time(() => TBMatrixCompl.MatrProdSeq(m1a, m2a));
    string sTime2 = calcDiffTime.GetResDiffTime();
    WriteTextToInfo("   \n-> comp static MatrProdSeq: Done. " + sTime2);
    calcDiffTime.SetDiffTime();
    TBMatrixCompl m3 = null;
    calcDiffTime.SetDiffTime();
    long matMultParTime = Time(() => m3 = m1 * m2);
    sTime2 = calcDiffTime.GetResDiffTime();
    WriteTextToInfo("   \n\t-> overloading comp * MultiplyParallel: Done. " + sTime2);
    return matMultSeqTime / (double)matMultParTime;
}//CompMatrixMultiplicationExampleDyn
```

Fig. 2. In parallel calculations were tested in MS VS C# 2008 Express Edition.

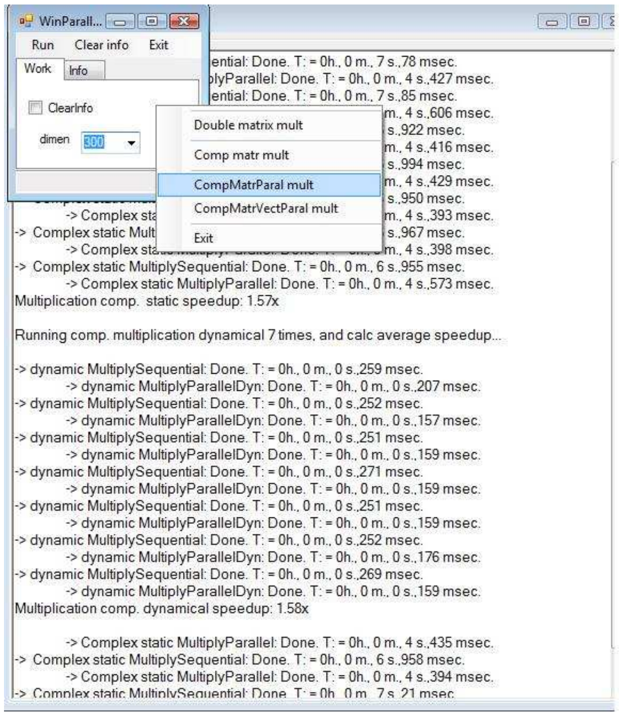
CTP library is quite powerful, easy to use, and provides a lot of different features, which allow solving the different tasks of parallel computations. It provides much more than just a single Parallel.For(). However, there are some issues, which may require other solution for following reasons:

1. The parallel computations extension is targeted for .NET framework 3.5. Recently the .NET Framework 4 Beta 1 is available for download [4]. However, some applications may still want to support earlier .NET framework versions, like 2.0, for example that makes difficult for them to use this extension. The parallel computations extension is not yet included into the standard .NET framework installation, and requires components that could not be installed on a target system.

2. The parallel computations extension provided by Microsoft is aimed to run on Windows systems. However sometimes it is necessary to develop the cross platform applications that also has to run on Linux, e.g. under the Mono environment. In this situation one will be left without the paralleling support.

Therefore further we give attention to the bench tests of parallel CTP library and the parallel library AFORGE [6]. In the latter the use of the dynamic module AFORGE.DLL is required only. This library represents a particular interest as it allows in parallel calculation not only in Windows, but also on cross-platform situations, for example in system Mono Linux. In this Report the following data types were used to in parallel bench tests:

1) static array double [,];



```
                    ential: Done. T: = 0h., 0 m., 7 s.,78 msec.
                    plyParallel: Done. T: = 0h., 0 m., 4 s.,427 msec.
                    ential: Done. T: = 0h., 0 m., 7 s.,85 msec.
                                                m., 4 s.,606 msec.
                                                 s.,922 msec.
                                                m., 4 s.,416 msec.
                                                 s.,994 msec.
                                                m., 4 s.,429 msec.
                                                 s.,950 msec.
          -> Complex sta                        m., 4 s.,393 msec.
->  Complex static Mult                          s.,967 msec.
          -> Complex sta                        m., 4 s.,398 msec.
->  Complex static MultiplySequential: Done. T: = 0h., 0 m., 6 s.,955 msec.
          -> Complex static MultiplyParallel: Done. T: = 0h., 0 m., 4 s.,573 msec.
Multiplication comp.  static speedup: 1.57x

Running comp. multiplication dynamical 7 times, and calc average speedup...

-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,259 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,207 msec.
-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,252 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,157 msec.
-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,251 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,159 msec.
-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,271 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,159 msec.
-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,251 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,159 msec.
-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,252 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,176 msec.
-> dynamic MultiplySequential: Done. T: = 0h., 0 m., 0 s.,269 msec.
          -> dynamic MultiplyParallelDyn: Done. T: = 0h., 0 m., 0 s.,159 msec.
Multiplication comp. dynamical speedup: 1.58x

          -> Complex static MultiplyParallel: Done. T: = 0h., 0 m., 4 s.,435 msec.
->  Complex static MultiplySequential: Done. T: = 0h., 0 m., 6 s.,958 msec.
          -> Complex static MultiplyParallel: Done. T: = 0h., 0 m., 4 s.,394 msec.
->  Complex static MultiplySequential: Done. T: = 0h., 0 m., 7 s.,21 msec.
```

Fig. 3. Simple graphic user interface and our bench tests for in parallel calculations.

2) dynamic array double [,];
3) static array TcompD [,];
4) dynamic array TcompD [,];
5) dynamic class TCompMatrix and TCompVector with the operators overloading;

The speedup factor K is defined as K = T_paral/T_seq, where T_paral is average time of in parallel calculation, while T_ seq is the calculating average time in a sequentially regime. Table 1 summarizes our calculations. A computer with Windows XP SR.3, Intel Core2 duo CPU processor (two processors), and frequency of 2.0 GHz was used.

3 Results and Discussion

Table 1. The speedup factor K of calculations in parallel

Type of data	CTP, 200x200	Aforge, 200x200	CTP, 400x400,	Aforge, 400x400
static matrixes: double[,]	1.94	1.72	1.98	1.98
dynamic matrixes: double[,]	2.04	1.75	2.03	1.80
c				1.82
complex dynamic TcompD[,]	1.67	1.76	1.70	1.75
Multiplication objects TCompMatrix()	1.67	1.76	1.70	1.75
Mult. TCompMatrix() and TCompVector()	2.36	2.24	2.38	2.27

We observe that performance in parallel computations is 0.75, from mean number of speedup factor K from Table 1, which is evaluated in a simple way, 1.75-1=0.75, which exceds the sequential calculations. The speedup of Aforge library is comparable with CTP library at least for Parallel.For operator.For completeness it is worth noting one interesting project that was developed in Parallel Language Research Project, see [7]. Besides we have to refer to other more classical direction of Message Passing Interface (MPI). MPI.NET is a high-performance, easy-to-use implementation of the Message Passing Interface (MPI) for Microsoft's .NET environment, for further references see [8].

Now almost all new servers and computers are running processors with multiple cores, and the software-design community is trying to figure out the best way of making use of this new architecture. Now is possible to use state of the art models and parallel computing programming languages like Chapel [9] or X10 [10], but if we analyze the sucessful of programming languajes like Java or C# in last decade, they felt familiar so it was easy to adopt them. According to [11] People with legacy code need tools that have strong attention to the languages they have written and give them an incremental approach to add parallelism. If languages like X10 and Chapel do turn out to be popular, their advancements will be integrated into more popular languages.

4 Conclusions

We studied the use of the library Microsoft Parallel Extensions CTP to .NET Framework 3.5 for calculations in parallel. In order to make the test deeper we developed dynamic hierarchies of classes having complicated internal structures that were applied as working example of advanced objects. Our testing program with graphic user interface (GUI) has allowed us to concentrate various bench tests. In result we have found that for advanced data the performance in parallel at least on 70%-80% and even more can exceed the sequential calculations. We believe using extensions like the proposed in this paper are the best approach to include parallel processing into new developments.

References

1. Are Magnus Bruaset, Aslak Tveito, Numerical Solution of Partial Differential Equations on Parallel Computers (Lecture Notes in Computational Science and Engineering), Springer, 2006.
2. Wenhua Yu, Raj Mittra, Tao Su, e.a., Parallel Finite-Difference Time-Domain Method, Artech House Publishers, 2006.
3. Eric Eilebrecht's blog, http://blogs.msdn.com/ericeil/archive/2009/04/23/clr-4-0-threadpool-improvements-part-1.aspx
4. Visual Studio 2010 and .NET Framework 4 Beta 1, http://msdn.microsoft.com/es-mx/netframework/dd582936(en-us).aspx
5. GNC.com: Does parallel processing require new languages?, http://www.gcn.com/Blogs/Tech-Blog/2009/06/New-parallel-processing-languages.aspx.
6. AForge.NET, http://www.aforgenet.com.
7. Parallel Language Research Project, http://www.parallelcsharp.com.
8. MPI.NET: High-Performance C# Library for Message Passing, http://www.osl.iu.edu/research/mpi.net;http://www.osl.iu.edu/research/mpi.net/software.
9. Chapel: The Cascade High-Productivity Language, http://chapel.cray.com/
10. X10: The New Concurrent Programming Language for Multicore and Petascale Computing, http://x10-lang.org/
11. Reinders, James. Intel Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism. Publisher O'Reilly Media, 2007.