# Parallel Computing Runtime for Microsoft .NET Framework

Alexander Chudinov
Program Systems Institute
of the Russian Academy of Sciences
st. Stroitely 24-43
Russia 152140, Pereslavl-Zalesskiy, Yaroslavl region
chudinov@strategypartner.com

Vladimir Roganov
Program Systems Institute
of the Russian Academy of Sciences
Research Center for Multiprocessor Systems
Russia 152140, Pereslavl-Zalesskiy, Yaroslavl region
var@skif.botik.ru

## ABSTRACT

The .NET Framework offers comprehensive and flexible thread APIs that allow the efficient implementation of multithreaded applications. These APIs can, however, only be utilized within Symmetric Multiprocessors (SMPs), which have a very limited scalability. For larger systems, which are in the PC world mostly represented as clusters of SMPs, other paradigms like message passing or handcrafted hybrid systems have to be used. These approaches are generally more difficult to program and require major code changes compared to sequential codes. This paper presents an extension to the .NET Framework, which implements the concept of automatic dynamic parallelization of programs. The extension provides both ease-of-use and scalability in development of parallel programs. This drastically eases the use of clusters and opens cluster architectures to a whole range of new potential users and applications.

The extension offers a new model of a computation process based on the following concepts:

- It uses functional programming paradigm, which perfectly fits for parallel computations. A program is a set of functions. Each function can have several inputs and outputs.
- Functions body can be implemented using imperative programming style. The only constraint is that a function must not have any side effects.

The joint of these two different programming concepts allows to combine global parallelism with local efficiency and ease of implementation of separate functions.

These concepts are naturally integrated into .NET Framework by means of using templatized classes (introduced in C# with generics) that encapsulate all low-level details such as threading, synchronization, scheduling, load balancing, etc., which makes the extension a powerful tool for implementing high-performance parallel programs.

**Keywords**
Automatic dynamic parallelization of programs, parallel computations, .NET Framework, .NET Remoting, .NET Threading

***1st Int.Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computing***

*February 6-8, 2003, Plzen, Czech Republic.*
Copyright UNION Agency – Science Press
ISBN 80-903100-3-6

## 1. INTRODUCTION

The paper is based on the principles developed in the existing T-system [Tsy02a],[Abr00a] and will use them to develop tools for parallel computing in the Microsoft .NET framework.

The T-system is a parallel computing framework that includes operating runtime for parallel applications execution and relevant programming tools. The goal of the T-System is organization of parallel computation process that effectively uses computation resources of all cluster nodes.

The T-system represents computation process as a computational network that includes data (ready and non-ready variables) and T-processes – elementary subroutines (granules of parallelism). It uses external scheduling algorithms for effective use of resources of each cluster node.

We will use the C# with generics language [Ken02ba] to implement parallel programming tools for the Microsoft .NET Framework.

The T-system for .NET Framework should possess the following characteristics:

- Compatibility with regular .NET code – semantics should be the same – the parallel .NET program and regular .NET program should give the same results.
- Easy to modify – adaptation for different operating systems and hardware configurations.
- Modularity – allows conduct simultaneous work on different systems parts and to easily customize/optimize the system for different hardware platforms and operating systems.
- High performance and ability to tune the system for different operating systems and hardware platforms.

## 2. WHAT IS T-EXTENSION?

We will call a base programming system a tuple {a language, a compiler, an execution runtime} that allows to create and execute programs on a regular sequential computer. By a language we understand a universal programming language with support for variables and functions, e.g. C, C++, C#, FORTRAN.

The T-system is a tuple {language T-extension (T-superstructure), compiler T-extension, execution runtime T-extension} that implements a automatic dynamic parallelization concept for programs that use functional programming concept on a high level (functions interoperation) and have bodies written in imperative style.

Modern programming systems include debugging tools, data structures and basic algorithms libraries, etc. So the fully developed T-system should have support for such components.

We will call a T-extension a T-specific constituent of the T-system over the base programming system X: T:X -> TX.

This paper will consider C# as a base language X and deal with its T-extension T#. Nevertheless, the most of the conclusions of this paper can be extrapolated to other programming systems.

It is important that a T-extension be as much as possible orthogonal to the base programming system. The T-extension is orthogonal to the base system if the properties of the base system and new properties of the extended system are independent. In other words: using T-constructions should not alter semantics of the C# language.

This characteristic puts strict requirements to the implementation of non-ready variables and work with those variable during program parallel execution. Since a user should, where possible, not to see these restrictions, the main burden for supporting this requirement should be on the T-system itself.

Also a special attention should be paid to the "smooth" syntax extension that can be obtained by using .NET attributes. This allows:

- Easily adapt already written systems.
- Compile and debug applications first in non-parallel environments using standard compilers.

## 3. SUPPORT FOR T-LANGUAGES SEMANTICS

The idea to use functional programming features for parallel computations is old enough. The most important of these features is the fact that functions without side effects can be computed in arbitrary order, so a programmer is freed of a number of problems connected to computations synchronization. However, there is no program architecture that is generic enough, and the advantages of functional approach are compensated by programming style limitations which often are hardly compatible with traditional imperative programming languages. That is why a pure parallel graph reduction is primarily used for implementation of functional programming languages [Gph00a].

The T-system has variables, more over the body of each T-function can be written using imperative programming paradigm (this is one of the main advantages of the T-system that allows to combine a global parallelism with a local performance and ease of implementation of separate functions). On the T-functions level we have pure functional paradigm. Since these different programming styles (one of which has immutable data and in the other data can change) join on the T-functions interaction level, it is logical to suppose that the data that exist on the joint of these two styles will change for one of them and will look immutable (be implicitly fixed) for the other.

The properties of T-variables are as follows:

- A variable holds a reference to a value that can be either ready or non-ready.

- Non-ready variable can have one or more suppliers (T-functions) during its lifecycle. When the last supplier is lost it becomes ready and cannot change its value.
- At each time a non-ready T-variable has a value that is accessible only for its supplier. Other T-functions stop when they try to access this variable until it becomes ready.

## 4. IMPLEMENTATION

### TVar, TVal, TPtr

The semantics described in the previous section can be implemented by the following templatized classes:

#### *4.1.1 TVar*

Templatized class for T-variables. A T# programmer deals only with TVar objects and TPtr pointers, but never directly with TVal values.

The lifecycle of a T-variable is limited by the time of execution of the function that created it. TVar can be either "hot" or "frozen".

- "Hot" TVar contains either a non-ready value or a value itself and can be modified.
- "Frozen" TVar contains a reference to TVal object and cannot be modified.

"Hot-frozen" transition can take place when a variable is passed as an out parameter to the T-function.

"Frozen-hot" transition takes place during second assignment operation.

Since T-variables exist only inside the functions, they won't need to think about synchronization of access to their values and/or pass it between cluster nodes.

#### *4.1.2 TVal*

Templatized class for T-values.

A T-value can be either ready or non-ready. If during program execution one accesses a non-ready value, thread execution stops until a value is computed (becomes ready).

A value has a counter of its consumers. When one tries to change a ready value, a consumers counter is checked. If it has only one reader a value becomes non-ready and a reader becomes a writer. If there are several consumers then readers have the same instance of TVal and writers get a newly created copy of TVal. If TVal does not have consumers it is destroyed.

TVal can contain either a value or a reference to another TVal (with real value). This additional reference level is necessary for fast copy of TVal.

TVal implementation takes care of data copying between cluster nodes. During copying the TVal values are blocked for reading.

#### *4.1.3 TPtr*

Templatized class for T-pointers. The pointers also can be either "hot" or "frozen". "Hot" pointers reference TVar objects. "Frozen" pointers reference TVal objects. This implements the semantics of copying a value by pointer.

A pointer becomes "Frozen" when it is passed as an out argument of a T-function.

### T-Functions

T-functions are implemented as templatized classes parameterized by function body and its arguments. To implement function body one needs to override method body() of the base T-function class and define a custom constructor to pass necessary arguments. An example of T-function is shown below:

```
class Tfib<int>: TFun<T>
{
    private int n;
    public TFib(int _n) {n = _n;}
    protected override void body()
    {
        if( n < 2 ) TOut.assign(n);
        else
        {
            TFib fib1 = new TFib(n-2);
            fib1.run();
            TFib fib2 = new TFib(n-1);
            fib2.run();
            TOut.assign(
              fib1.TOut.value() +
              fib2.TOut.value()
            );
        }
    }
}
```

## 5. T-SYSTEM ARCHITECTURE

As it was described above the T-system consists of three main components: T-Superstructure (T-semantics), T-compiler (or more general T-development environment) and T-runtime (Mobile Threads Objects and References, or shortly MoTOR).

### T-Superstructure (T-semantics)

The semantics is as described in the previous section. It is implemented via TVar, TVal, TPtr classes and T-functions. All changes in the semantics should not touch T-runtime.

### T-Development environment

Will include t-converter T#->C# and in future may include debugging tools.

## MoTOR (Mobile Threads Objects and References)

### *5.1.1 Threads*

Threads should provide functionality for running threads on different cluster nodes.

### *5.1.2 GRIS*

This module implements parallel graph reductions algorithm and tasks scheduler. The scheduler can give different priorities to threads as well as start new threads on other cluster nodes. It will use functionality of the Treads module to create new threads.

#### *5.1.2.1 Resource Monitoring*

To fulfill its duties the scheduler needs to know the data about available resources of cluster nodes. So it is necessary to have remote resources monitoring module that will allow get such parameters as CPU usage percent, available physical memory, etc.

#### *5.1.2.2 Migration Advices*

This module will process the data received from the remote resources monitoring module, analyze the code waiting for execution and give advices for most optimal tasks distribution between cluster nodes. In the simplest case the scheduler should create a new thread on the node with most free resources.

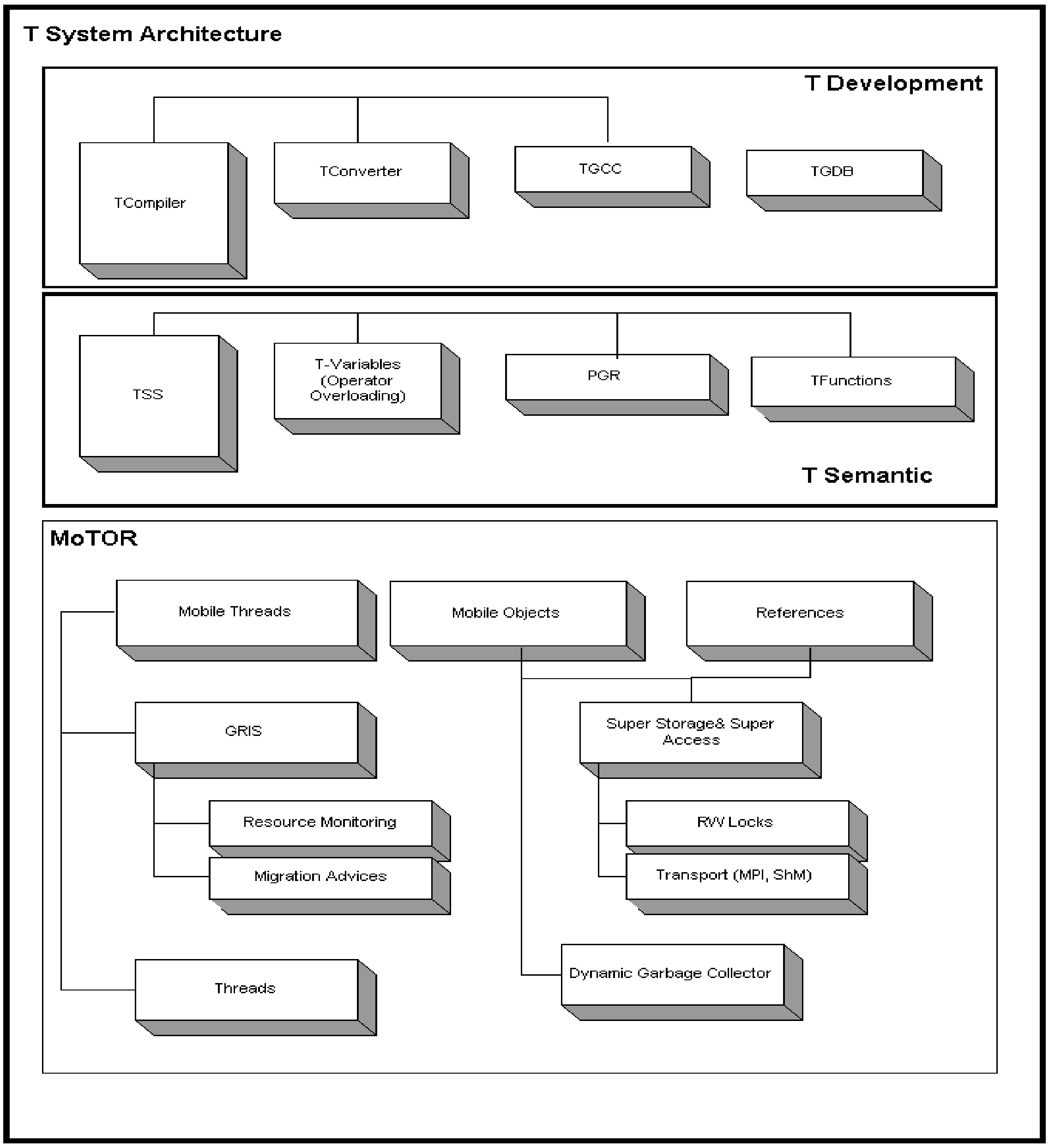

**Figure 1. T-system Architecture.**

### *5.1.3 Mobile Objects & References*

The implementation of this module is nearly completely independent from other parts of the T-system. This includes synchronization of access to the T-values and sending T-values between cluster nodes.

## 6. CONCLUSION

This paper has described the design and implementation of Parallel Computing Runtime for Microsoft .NET Framework based on the principles developed in the existing version of the T-system. Currently we have developed T-semantics (T-superstructure) module. T-development environment and MoTOR modules currently are in the development process.

## 7. REFERENCES

[Abr00a] Abramov, S.M., Vasenin V.A., Mamchits E.E., Roganov V.A., Slepuhin A.F. Dynamic Program Parallelization Based on Parallel Graph Reduction. The Architecture of New Version of T-System Software (In Russian). In Proc of High Performance Computations and Their Applications, Chernogolovka, Russia, pp.261-265.

[Gph00a] Glasgow Parallel Haskell. http://www.cee..hw.ac.uk/~dsg/gph

[Ken02ba] Kennedy, A., Syme D. Design and Implementation of Generics for the .NET Common Language Runtime, Microsoft Research, Cambridge, U.K., 2002

[Tsy02a] T-system. http://skif.pereslavl.ru