Reforming the Trees – C# and F# Comparison

Natela Archvadze[1], Merab Pkhovelishvili[2]
[1]I.Javakhishvili Tbilisi State University, Tbilisi, Georgia,
[2]Georgian Technical University, Institute Computation Mathematics, Tbilisi, Georgia
[1]*natarchvadze@yahoo.com*, [2]*merab_4@list.ru*

***Abstract*— During the recent years, creation of parallel secure programs has become one of the main goals since micro-processors are no more a rarity. Functional languages are assisting us to keep up the parallelism enabling inclusion of unchangeable structures of data, which can be transferred between streams and computers, even without additional security for the streams or atomic access. Besides, functional languages simplify creation of libraries, supporting parallel processing (for example, asynchronous working streams F#). In this article we consider the information on trees and main operations: finding elements in the tree, inserting and removing elements and also traversing the tree by means of two programming languages F# and C#. We will compare calculation process time parameters.**

Keywords*— *functional programming; data structure; tree;

I. INTRODUCTION

Recently language F# has filled up the Microsoft®.NET Framework. This functional language [1] provides with security according to the types, not bad productivity, and also can be used as a scenario language – and all this within the platform frame.NET. As all concepts of functional programming penetrate into the main languages (C#, Visual Basic) through such technologies as generalized types .NET or LINQ, language F# is obliged its success to the .NET community.

Programs on F# look more laconic [2]. From any aspect, «typography» is less: not only symbol quantity decreases, but also quantity of places, where compilation requires indicates the type of variable, argument or return value. It means that lesser code volume needs to be services.

By efficiency F# can be compared with C#, however, if we compare it with languages that are same laconic as F# is, namely, with dynamic languages and scenario languages, it has better characteristics. And as in many dynamic languages, F# is equipped with means that enable to check data: write a code fragment and fling it in interactive mode.

We consider one and the same problem analysis in this article for two languages of .Net platform programming: F# and C# and we will compare time parameter of calculation process. Namely, we will construct trees and consider main operation of the trees: finding element, insertion of element and removing of the node and traversing the tree. Each module realization on both languages is accomplished.

The article considers the situation, when the problems dynamically add to working stack. This is known as dynamic parallelism of the problem. Simple example of the dynamic parallelism – a problem, which inserts recurse in the realization of algorithm. Dynamic parallelism problems are also known, as recursive division or "divide and govern".

Dynamic parallelism represents a parallel execution of several independent calculations [3]. Called activity with multiple lists of arguments. Each list of such sets serves as simultaneous is called. Calls are not depended on each other; after its ending the activity is considers completed and the ends. For example, when you count the knots quantity in data structure, represented with binary tree, you can count left and right sub-tree knots and then sum the results up.

II. TREE - MAKING

Tree is a Tree<T> class object, where T is a member of the given tree, Tree is made up of node and two sub-trees:

C#:

```
publicclassTree<T>
    {
    public T Data { get; set; }
    publicTree<T> Left { get; set; }
    publicTree<T> Right { get; set; }
    }
```

F# :

```
type Tree<'T> =
                | Node of Tree<'T> * 'T * Tree<'T>
                | Leaf
```

Tree is made by recursive function MakeTree from maximum quantity of the given elements. Realization of the search algorithms were checked on the trees, which were made according to the sub-tree direction by random 50/50 C# and F# language methods.

C#:

```
publicstaticTree<string> MakeTree
                        (int nodeCount, double density)
    {
if (nodeCount < 1)
    thrownewArgumentOutOfRangeException("nodeCount");
if (!(0 < density && density <= 1.0))
        thrownewArgumentOutOfRangeException("density");
return MakeTree(nodeCount, density, 0, newRandom());
    }
staticTree<string> MakeTree
        (int nodeCount, double density, int offset, Random r)
    {
var flip1 = r.NextDouble() > density;
var flip2 = r.NextDouble() > density;
```

```
var newCount = nodeCount - 1;
int count1 = flip1 && flip2 ? newCount / 2 : flip1 ? newCount
: 0;
int count2 = newCount - count1;
if (r.NextDouble() > 0.5)
        { var tmp = count1;
          count1 = count2;
          count2 = tmp;
        }
returnnewTree<string>()
        {  Data = offset.ToString(),
           Left = count1 > 0 ?
                  MakeTree(count1, density, offset + 1, r) : null,
           Right = count2 > 0 ? MakeTree
                  (count2, density, offset + 1 + count1, r) : null
        };
    }
```

F# :

```
let makeTree nodeCount density =
if nodeCount < 1 then invalidArg "nodeCount out of range"
if not (0.0 < density && density <= 1.0) then invalidArg
"density""out of range"

letrec makeTreeUtil nodeCount density offset (r:Random) =
    let flip1 = r.NextDouble() > density
    let flip2 = r.NextDouble() > density
    let newCount = nodeCount - 1
    let count1, count2 =
    let c = if flip1 && flip2 then newCount / 2 elif flip1
                                          then newCount else 0
                    if r.NextDouble() > 0.5 then c, newCount - c
                                          else newCount - c, c
    let l = if count1 > 0 then makeTreeUtil count1 density
                                        (offset + 1) r else Leaf
    let r = if count2 > 0 then makeTreeUtil count2 density
  (offset + 1 + count1) r else Leaf  Node(l, offset.ToString(), r)
            makeTreeUtil nodeCount density 0 (new Random())
```

Where, nodeCount is quantity of remaining elements, Offset – is current elements meaning. r – a generator of random numbers, by which the sub-tree direction is calculated. As it seems, their codes are called recursively until the remaining quantity of elements reaches 0.

III. Tree Traverse Methods

Tree traverse is accomplished by parallel and non-parallel methods. Each method is evaluated by executed time.

A. Making schedule from the given tree

SequentialCall method called SequentialWalk method, which will pass first left sub-tree elements and then right sub-tree elements. At each step in result we write current node meaning.

Delegate action, can be used to forward method into the parameter class. Encapsulated method has to coincide with methos signature driven by this delegate method, this means that encapsulated method has to have one parameter, traversed by requirement and must not return the meaning.

C# :

```
staticvoid SequentialCall(Tree<string> tree)
    { for (int i = 0; i < N; i++)
      { List<string> result = newList<string>();
        SequentialWalk(tree, (data) =>
{TimePerformance.DoCpuIntensiveOperation(Time);
                result.Add(data);
      } };
      }
Console.WriteLine();
    }
staticvoid SequentialWalk<T>(Tree<T> tree,
                              Action<T> action)
    { if (tree == null) return;
      action(tree.Data);
      SequentialWalk(tree.Left, action);
      SequentialWalk(tree.Right, action);
    }
```

F#:

```
let SequentialCall n time tree =
for i in 0 .. n - 1 do
let result = new ResizeArray<_>()
      tree |> sequentialWalk (fun data ->
TimePerformance.DoCpuIntensiveOperationSimple time |>
ignore result.Add(data) )
    Console.WriteLine()
letrecprivate sequentialWalk action tree =
match tree with
  | Node(l, d, r) ->
    action d
    sequentialWalk action l
    sequentialWalk action r
  | _ -> ()
```

SequentialCall method called (accepts) SequentialWalk method, which will pass first left sub-tree elements, then rightside. At each step in result we write current node meaning. Delegate action can be used to forward method into the parameter class. Encapsulated method has to coincide with methos signature driven by this delegate method, this means that encapsulated method has to have one parameter, traversed by requirement and must not return the meaning.

In each project there is an assisting project Utilities, in which TimePerformance class methods count time spent on operation.

So, that, by called of SequentialCall method on C#, to pass the tree, when the maximum quantity of elements equals - 1000 and 0.55 density, it will need 00:00:12.23 sec for the same operation on F#it needed only: 00:00:11.86.

B. Same methods for parallel processes

In the given method ParallelWalk and parallelWalk in every recursive step dynamically called parallel processes by means of new methods, entered in .Net 4.0 Task.Factory.StartNew. That is why the number of processes

is three times more than number of knots on the tree, this makes the big number. Library of parallel problems (TPL) is designed to solve such situations.

C#:

```
staticvoid ParallelCall(Tree<string> tree)
      {  for (int i = 0; i < N; i++)
         { ConcurrentBag<string> result =
                                    newConcurrentBag<string>();
            ParallelWalk(tree, (data) =>
            {
TimePerformance.DoCpuIntensiveOperation(Time);
               result.Add(data);
            });
         }
      }
staticvoid ParallelWalk<T>(Tree<T> tree, Action<T>
action)
      {
if (tree == null) return;
var t1 = Task.Factory.StartNew(
                  () => action(tree.Data));
var t2 = Task.Factory.StartNew(
                  () => ParallelWalk(tree.Left, action));
var t3 = Task.Factory.StartNew(
                  () => ParallelWalk(tree.Right, action));
Task.WaitAll(t1,t2, t3);
      }
```

F#:

```
let ParallelCall n time tree =
for i in 0 .. n - 1 do
let result = new ConcurrentBag<_>()
         tree |> parallelWalk (fun data ->
 TimePerformance.DoCpuIntensiveOperationSimple
        time |> ignore result.Add(data) )
letrecprivate parallelWalk action tree =
match tree with
   | Node(l, d, r) ->
let t1 = Task.Factory.StartNew(fun () ->
         action(d))
let t2 = Task.Factory.StartNew(fun () ->
         parallelWalk action l)
let t3 = Task.Factory.StartNew(fun () ->
         parallelWalk action r)
      Task.WaitAll(t1, t2, t3)
   | _ -> ()
```

After the called of this method we have received the following results: C#: 00:11:35, F#: 00:11:09.

C. Code Using Dynamic parallelism

TPL switches process creation option AttachedToParent. AttachedToParent keeps the points of the problem, that it created. In this case sub-problem is known as children problem and problem that has created children roblem as parent problem. One can use option AttachedToParent in two cases. The first, when one wants to coordinate parent and children problems. The second, when one wishes to use Microsoft Visual Studio, to see that parent problem fulfillment row is unchanged.

C#:

```
staticvoid Parallel2Call(Tree<string> tree)
      { for (int i = 0; i < N; i++)
         {
ConcurrentBag<string> result =
newConcurrentBag<string>();
            Parallel2Walk(tree, (data) =>
            {
TimePerformance.DoCpuIntensiveOperation(Time);
               result.Add(data);
            });
         }
      }
staticvoid Parallel2Walk<T>(Tree<T> tree, Action<T>
action)
      { if (tree == null) return;
var t1 = Task.Factory.StartNew(() => action(tree.Data),
TaskCreationOptions.AttachedToParent);
var t2 = Task.Factory.StartNew(() =>
Parallel2Walk(tree.Left, action),
TaskCreationOptions.AttachedToParent);
var t3 = Task.Factory.StartNew(() =>
Parallel2Walk(tree.Right, action),
TaskCreationOptions.AttachedToParent);
Task.WaitAll(t1, t2, t3);      }
```

F#:

```
let ParallelAttachedCall n time tree =
for i in 0 .. n - 1 do
let result = new ConcurrentBag<_>()
         tree |> parallelWalkAttached (fun data ->
TimePerformance.DoCpuIntensiveOperationSimple
     time |> ignore result.Add(data) )
l letrecprivate parallelWalkAttached action tree =
match tree with
        | Node(l, d, r) ->
let t1 = Task.Factory.StartNew((fun () ->
         action(d)), TaskCreationOptions.AttachedToParent)
let t2 = Task.Factory.StartNew((fun () ->
         parallelWalkAttached action l),
         TaskCreationOptions.AttachedToParent)
let t3 = Task.Factory.StartNew((fun () ->
         parallelWalkAttached action r),
TaskCreationOptions.AttachedToParent)
       Task.WaitAll(t1, t2, t3)
       | _ -> ()
```

If parent has at least one children problem, the parent will not finish working until all its children problems are finished and its status at this time equals to WaitingForChildrenToComplete.

After the called of this method we have received the following results: C#: 00:11:32, F#: 00:11:11.

D. F # - Asynchronous flows for a specific work method

Async.StartChild <'T> starts child count in asynchrony working process [4]. This enables to perform several asynchrony count at the same time. This method, usually, is used, as a right member of let in F# asynchrony process. At each usage StartChild will let child sample and return an object, which represents t1 or t2 and will wait till the operation ends. At performing t1 awaits for ending of parallelWalkAsync.

F#:

```
let ParallelAsyncCall n time tree =
for i in 0 .. n - 1 do
let result = new ConcurrentBag<_>()
                tree |> parallelWalkAsync (fun data ->
                  async {
TimePerformance.DoCpuIntensiveOperationSimple time |>
ignore result.Add(data) }) |> Async.RunSynchronously
letrecprivate parallelWalkAsync action tree = async {
match tree with
   | Node(l, d, r) ->
// Start recursive processing and process current element
let! t1 = Async.StartChild(parallelWalkAsync action l)
let! t2 = Async.StartChild(parallelWalkAsync action r)
do! action d
// wait for completion of recursive processing & return
let! r1 = t1
let! r2 = t2
return ()
   | _ ->return () }
```

After running this method I have received the following result F#: 00:11:68, which witnesses priority of F# compared to C#, from the point of view of the speed of data process.

IV. ADDING INFORMATION

Let us describe the method, which adds new words to the tree. Using this method, we move the given knot as a result, to receive non-stop system realization.

I am searching for the place to make addition of a new word. Let us say, we have word sequence, which are already added to the tree: apple, application, appearance and let us add "apple-store". After we call method FindNode, knot with letter "e" from the word "apple" will come back to us and the row "-store". As the remainder of the row "-store" is not empty, we will add one more knot to child list. Which we will create for the remainder of the row by constructor. Constructor will create a chain from the letters -_s_t_o_r_e. To have the tree in an alphabetic view, we use function List.SortBy for new childs. If the word remainder stays empty, then we give word ending sign to the last knot isword <- true

```
    member node.Add (str : string) =
let (n, s) = node.FindNode(str)
  match s.Length with
       | 0 -> n.isword <- true
       | _ -> n.childs <- List.sortBy (fun n -> n.letter) (new
Node (s) :: n.childs)  node
```

Next function is a function, which reads sequence of the words from the console. Each row is accepted as a separate word. We compile a list out of all words entered, until the user enters an empty row. An empty row is not added to the list.

```
let rec readList () =
   match stdin.ReadLine() with
       | w when w.Length = 0 -> []
       | w -> w :: readList()
```

Dialogue has the following:

```
Enter source list of words:
apple
application
apple-store
appearance

Find : app
appearance
apple
apple-store
application

Find : appl
apple
apple-store
application

Find : appli
application

Find : finger
```

Summary: Tree-making and traverse algorithms, algorithms of searching in the trees in two languages C# and F# are realized. Their completion speed comparison has been done. F# priority is shown. To test the methods the trees have been made with random numbers algorithm.

REFERENCES

[1] T. Petricek, J.Skeet. „Functional Programmin fot the Real World". - 2009. ISBN-460-1-5987-35

[2] Сошников Д. В. „Программирование на F#. - М.:ДМК Пресс, 2011. – 192с. ISBN 978-5-94074

[3] D.Syme, A. Granicz, A.Cisternino. "Expert F# 2.0" 2010. ISBN-13 (pbk): 978-1-4302-2431-0

[4] L. Hoban. "F# for Parallel and Asynchronous Programming". 2010. ISBN-978-2-45873R.