A Parallelization of Instance Methods of a .NET Application that Search for Required Structured Data Stored in a Skip List

Igor Košťál

Abstract The skip list is a more memory efficient version of a single level linked list. Searching for the required data elements in a skip list is more efficient than in a single level linked list because a skip list allows us to skip to the searched element in it. We have created a C# .NET application that uses a skip list with structured data in its data elements. This application can perform search operations within these data elements using serial, threaded, and parallelized instance methods, and simultaneously it is able to measure the execution times of particular methods. By comparing these times, we have examined the execution efficiency of parallelized instance methods of the object of the .NET application compared to threaded and serial instance methods of the same object. The results and evaluation of this examination are listed in the paper.

Keywords Parallelization · Skip list · Data element · Structured data · .NET application

1 Introduction

A single level linked list (a simple list) is a dynamic data structure that is used for storing data in applications. However, there are also multi-level linked lists (called skip lists [5]) that are more complicated for creating but searching for the required data elements in them is more efficient because they allow us to skip to the searched element in them.

We dealt with a comparison of execution efficiency of the use of a skip list and simple list in a C# .NET Application [2]. This application had the same structured data of students stored in the data elements of its skip and simple list. From the results of the comparison of the execution times of the search operations and the insertion operation performed by our .NET application in the skip list and execution times of

I. Košťál (✉)
Faculty of Economic Informatics, University of Economics in Bratislava, Dolnozemská cesta 1, 852 35 Bratislava, Slovakia
e-mail: igor.kostal@euba.sk

S. Khakhomov et al. (eds.), *Research and Education: Traditions and Innovations*,
Lecture Notes in Networks and Systems 422,
https://doi.org/10.1007/978-981-19-0379-3_11

the same operations performed by the same application but in a simple list, it was obvious that the use of the skip list in this application was with a bigger number of data elements containing structured data, significantly more efficient [2].

We were interested in now, how to parallelize search operations in a skip list with the structured data of students in its data elements. We have created a C# .NET application that uses a skip list with such data elements. This application parallelizes the execution of search instance methods in its instance methods, and at the same time measures the execution times of its serial and parallel search instance methods. By comparing these times, we have examined the execution efficiency of a parallelization of search instance methods of our C# application. The results and evaluation of this examination are listed in the paper.

2 A Simple List and a Skip List

A one-way linked list (a simple list) (Fig. 1a) is a set of dynamically allocated elements (called nodes), arranged in such a way that each element contains two items—the data and a reference to the next element [2]. The last node has a reference to NIL (a special node, which terminates the list). A linked list is a real dynamic data structure [1]. The number of nodes in a list is not fixed and can grow and shrink on demand [1]. Any application which has to deal with an unknown number of objects will need to use a linked list [1]. We might need to examine every node of the list when searching a simple list. If the list is stored in sorted order and every other node of the list also has a reference to the node four ahead it in the list (a skip list) (Fig. 1b), we have to examine no more than $\lceil n/4 \rceil + 2$ nodes (where n is the length of the list) [5]. This data structure could be used for fast searching for required nodes.

We have created a C# .NET application that uses a skip list, but with structured data of students in its data elements (Fig. 2). Using our .NET application, we have examined the execution efficiency of a parallelization of search instance methods of this application.

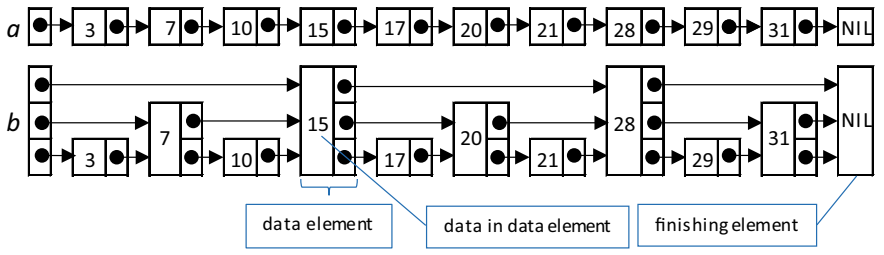


Fig. 1 The simple list **a** and the skip list **b** with simple, unstructured data in their data elements [5]

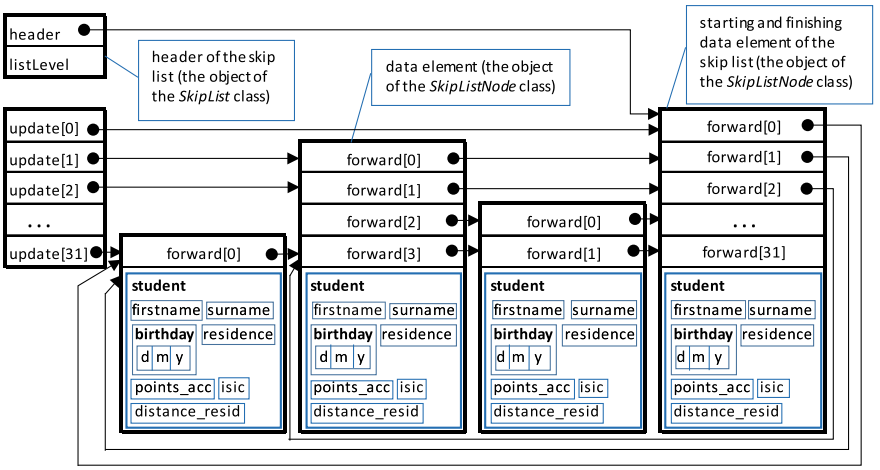


Fig. 2 The internal structure of the skip list that uses our .NET application [2]

3 Parallel Programming in the .NET Framework

Nowadays, many personal computers and workstations have multiple CPU cores that enable multiple threads to be executed simultaneously. To take advantage of the hardware, we can parallelize our code to distribute work across multiple processors. The Microsoft Visual Studio development environment and the .NET Framework 4 enhance support for parallel programming by providing a runtime, class library types, for example, the Task Parallel Library, and diagnostic tools. These features simplify parallel development. We can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool. [3]

The Task Parallel Library (TPL) is a set of public types and APIs (Application Programming Interfaces) in the *System.Threading* and *System.Threading.Tasks* namespaces. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications [3].

We know two kinds of a parallelism in parallel programming in the .NET Framework—data and task parallelism.

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently. The TPL supports data parallelism through the *System.Threading.Tasks.Parallel* class. This class provides method-based parallel implementations of for and foreach loops. We write the loop logic for a *Parallel.For* or *Parallel.ForEach* loop much as we would write a sequential loop. We do not have to create threads or queue work items. The TPL handles all the low-level work for us. When a parallel loop runs, the TPL partitions the data source

so that the loop can operate on multiple parts concurrently [3]. Data parallelism with the *Parallel.ForEach* method is used in a parallelized code of our .NET application.

The TPL is based on the concept of a task, which represents an asynchronous operation. In some ways, a task resembles a thread or *ThreadPool* work item, but at a higher level of abstraction. The term **task parallelism** refers to one or more independent tasks running concurrently [3]. Task parallelism with the *Parallel.Invoke* method is used in a parallelized code of our .NET application.

Several overloads of the *Parallel.For* and *Parallel.ForEach* methods are used in data parallelism. We introduce a brief description of this overload of the *Parallel.ForEach* method from [3] that is used in a parallelized code of our .NET application. This description explains how this method operate in this code.

The *Parallel.ForEach* method executes a foreach operation on an *IEnumerable* in which iterations may run in parallel.

Syntax
public static ForEach<TSource>(IEnumerable<TSource> source, Action <TSource> body);
Type Parameters
TSource—the type of the data in the source.

Parameters
IEnumerable<TSource> source—an enumerable data source.
Action<TSource> body—the delegate that is invoked once per iteration.

Remarks
The *body* delegate is invoked once for each element in the source enumerable. It is provided with the current element as a parameter. The *Parallel.ForEach* method does not guarantee the order of execution. Unlike a sequential *ForEach* loop, the incoming values are not always processed in order.

We also introduce a brief description of the *Parallel.Invoke* method from [3] that is used in a parallelized code of our .NET application. This description explains how this method operate in this code.

The *Parallel.Invoke* method executes each of the provided actions, possibly in parallel.

Syntax
public static void Invoke (params Action[] actions);

Parameters
Action[] actions—an array of Action to execute.

Remarks
This method can be used to execute a set of operations, potentially in parallel. No guarantees are made about the order in which the operations execute or whether they execute in parallel. This method does not return until each of the provided operations has completed, regardless of whether completion occurs due to normal or exceptional termination.

4 C# .NET Application with Serial, Threaded, and Parallel Instance Methods

Our .NET application was developed in the C# language in the development environment Microsoft Visual Studio 2019 Enterprise for the Microsoft .NET Framework version 4 and for the Microsoft Windows 10 operating system. The .NET application has the structured data of students stored in the data elements of a skip list. This application can perform search operations within these data elements using serial, threaded, and parallelized instance methods, simultaneously it is able to measure the execution times of particular methods, and write these times into the *LogFile.txt* disc file. The source code of this application is divided into two parts that are saved into files:

SkipLNodeClasses.cs—the *SkipList* class is defined here. This class includes member methods that are able to perform all search operations such as searching for students (data elements) according to their points for accommodation, year of birth, surname, ISIC, and according to distance between their residence and university in a skip list (*find_1st_pts*, *find_1st_y*, *find_1st_surname*, *find_1st_isic*, *find_1st_distanceOfResid*), inserting of a new student (a data element) sorted into a skip list (*insert_1st*). The *SkipList* class also includes service member methods serving to delete particular data elements or all data elements. The *SkipListNode* nested class is defined within the scope of the *SkipList* class. The *SkipListNode* class object represents one data element (node) of a skip list. In the *SkipList* class, two structures the *Student* and the *date* are declared, by which the data part of the data element of the skip list is created. From the *SkipLNodeClasses.cs* source file was built (.dll) library into the *SkipListNode.dll* file that is the dynamic run-time component of our .NET application.

SkipList_WinForm1.cs—the *SkipList_WinForm1* class that is derived from the *System.Windows.Forms.Form* class is defined here. This system class represents a window that makes up an application's user interface. The *SkipList_WinForm1* class includes event handlers of all controls of the user interface of our .NET application and helper methods. The *SerialSearchBtnClick*, *SearchOnMultipleThrdsBtnClick*, *ParallelInvokeSearchBtnClick*, and *ParallelForEachSearchBtnClick* event handlers of four buttons contain serial, threaded, and parallelized search operations source code that allows us to search for students (data elements) in a skip list according to several required parameters, for example, according to several points for accommodation, several distances of a residence of students from the university, etc. The user inserts these several different required parameters into the *requiredValueTB* input text box of the .NET application. From this text box, they are read by the *Parse_requiredValueTB* helper member method, which is called by each of the mentioned event handlers.

The *SerialSearchBtnClick* event handler (Fig. 4) searches for students (data elements) in the skip list according to several required parameters in serial. In the displayed part of the source code of this event handler, students are searched for according to the 4 required distances of their residence from the university. The

event handler performs a search on one thread in serial using four calls to the *SearchStudentsWithDistOfResid* helper member method (Fig. 3), which performs the real search for particular students (data elements) in the skip list.

The *SearchOnMultipleThrdsBtnClick* event handler (Fig. 5) searches for students (data elements) in the skip list according to several required parameters on multiple threads. In the displayed part of the source code of this event handler, students are searched for according to the 4 required distances of their residence from the university. The event handler performs a search on four threads by calling the *SearchStudentsWithDistOfResid* helper member method (Fig. 3), which performs the real search for particular students (data elements) in the skip list.

```
private string SearchStudentsWithDistOfResid(int distance_f) {
    string str_result = "";       int i = 0;

    if (flag_stream == 1)
    while (i < students_count) {
        if (ref_arr_students[i].isic != 30000000000000001)
                // calling the instance method of the 'list' object (this is object of the 'SkipList' class)
            str_result += list.find_1st_distanceOfResid(ref_arr_students[i], distance_f);
        i++;
    } . . .
    return str_result;
}
```

Fig. 3 The part of the source code of the *SearchStudentsWithDistOfResid* helper member method

```
private void SerialSearchBtnClick(object sender, EventArgs e) {
    string str_result = "";  string str_result1 = "";  string str_result2 = "";  string str_result3 = "";
    string str_result4 = ""; . . .
    int requiredValuesCount = Parse_requiredValueTB(); // calling the helper member method
    . . .
    // if the 'residDistance_RB' radio button is checked
    else if (residDistance_RB.Checked) {
        // if the user inserted 4 values into the 'requiredValueTB' text box
        if (requiredValuesCount == 4) { . . .
            int dist1 = numbersIn_requiredValueTB[0];  int dist2 = numbersIn_requiredValueTB[1];
            int dist3 = numbersIn_requiredValueTB[2];  int dist4 = numbersIn_requiredValueTB[3];

         // perform four tasks in serial on the source data
         str_result1 = SearchStudentsWithDistOfResid(dist1); // calling the helper member method
         str_result2 = SearchStudentsWithDistOfResid(dist2);
         str_result3 = SearchStudentsWithDistOfResid(dist3);
         str_result4 = SearchStudentsWithDistOfResid(dist4);
         . . . } . . . } . . . }
```

Fig. 4 The part of the source code of the *SerialSearchBtnClick* event handler

```
private void SearchOnMultipleThrdsBtnClick(object sender, EventArgs e) {
    string str_result = "";  string str_result1 = "";  string str_result2 = "";  string str_result3 = "";
    string str_result4 = ""; . . .
    int requiredValuesCount = Parse_requiredValueTB(); // calling the helper member method
    . . .
    else if (residDistance_RB.Checked) { // if the 'residDistance_RB' radio button is checked
        // if the user inserted 4 values into the 'requiredValueTB' text box
        if (requiredValuesCount == 4) {
            . . .
    // create the delegate object and bind to the 'SearchStudentsWithDistOfResid' target method
    AsyncMCallerWF1C callerM1 = new AsyncMCallerWF1C(SearchStudentsWithDistOfResid);
    . . .
    // execute 4 methods asynchronously on 4 secondary threads
    IAsyncResult asyncState3 = callerM1.BeginInvoke(numbersIn_requiredValueTB[3], null, null);
    IAsyncResult asyncState2 = callerM1.BeginInvoke(numbersIn_requiredValueTB[2], null, null);
    IAsyncResult asyncState1 = callerM1.BeginInvoke(numbersIn_requiredValueTB[1], null, null);
    IAsyncResult asyncState0 = callerM1.BeginInvoke(numbersIn_requiredValueTB[0], null, null);
    . . .
    // capture all return values
    str_result1 = callerM1.EndInvoke(asyncState3);
    str_result2 = callerM1.EndInvoke(asyncState2);
    str_result3 = callerM1.EndInvoke(asyncState1);
    str_result4 = callerM1.EndInvoke(asyncState0);
    . . . } . . . } . . . }
```

Fig. 5 The part of the source code of the *SearchOnMultipleThrdsBtnClick* event handler

The *ParallelInvokeSearchBtnClick* event handler (Fig. 6) searches for students (data elements) in the skip list according to several required parameters in parallel, using task parallelism. In the displayed part of the source code of this event handler, students are searched for according to the 4 required distances of their residence from the university. The event handler performs a search in parallel by calling the *Parallel.Invoke* method, in which task parallelism is implemented. The *SearchStudentsWithDistOfResid* helper member method (Fig. 3) is called in the lambda expressions of the *Parallel.Invoke* method. This helper method performs the real search for particular students (data elements) in the skip list.

The *ParallelForEachSearchBtnClick* event handler (Fig. 7) searches for students (data elements) in the skip list according to several required parameters in parallel, using data parallelism. In the displayed part of the source code of this event handler, students are searched for according to the 4 required distances of their residence from the university. The event handler performs a search in parallel by calling the *Parallel.ForEach* method, in which data parallelism is implemented. The *SearchStudentsWithDistOfResid* helper member method (Fig. 3) is called in the lambda expression of the *Parallel.ForEach* method. This helper method performs the real search for particular students (data elements) in the skip list.

```
private void ParallelInvokeSearchBtnClick(object sender, EventArgs e) {
    string str_result1 = "";  string str_result2 = "";  string str_result3 = "";  string str_result4 = "";  . . .
    int requiredValuesCount = Parse_requiredValueTB(); // calling the helper member method  . . .
    else if (residDistance_RB.Checked) { // if the 'residDistance_RB' radio button is checked
        // if the user inserted 4 values into the 'requiredValueTB' text box
        if (requiredValuesCount == 4) { . . .
            int dst1 = numbersIn_requiredValueTB[0];  int dst2 = numbersIn_requiredValueTB[1];
            int dst3 = numbersIn_requiredValueTB[2];  int dst4 = numbersIn_requiredValueTB[3];
        // perform four tasks in parallel on the source data
        Parallel.Invoke(
            () => { str_result1 = SearchStudentsWithDistOfResid(dst1); }, //close the first Action
            () => { str_result2 = SearchStudentsWithDistOfResid(dst2); },//close the second Action
            () => { str_result3 = SearchStudentsWithDistOfResid(dst3); }, // close the third Action
            () => { str_result4 = SearchStudentsWithDistOfResid(dst4); } // close the fourth Action
            ); // close Parallel.Invoke . . . } . . . } . . . }
```

Fig. 6 The part of the source code of the *ParallelInvokeSearchBtnClick* event handler

```
private void ParallelForEachSearchBtnClick(object sender, EventArgs e) { . . .
    int requiredValuesCount = Parse_requiredValueTB(); // calling the helper member method  . . .
    else if (residDistance_RB.Checked) { // if the 'residDistance_RB' radio button is checked
        if (requiredValuesCount > 0) {
            int k = 0;  string local_str_result = "";
            int[] integer_inputs = new int[requiredValuesCount];
            while (k < requiredValuesCount) {
                integer_inputs[k] = numbersIn_requiredValueTB[k];  k++; }
        // the 'local_strings' object represents a thread-safe, unordered collection of objects
        var local_strings = new ConcurrentBag<string>();
        Parallel.ForEach(integer_inputs, currentInput =>
        {  local_strings.Add(SearchStudentsWithDistOfResid(currentInput));  });
        List<string> strings_list = local_strings.ToList();

        foreach (string str in strings_list)  local_str_result += str;
    . . . } . . . } . . . }
```

Fig. 7 The part of the source code of the *ParallelForEachSearchBtnClick* event handler

The *SkipList_WinForm1.cs* source file was built into the *SkipList_WinForm.exe* file. The *SkipList_WinForm.exe* is a Windows application that uses the dynamic run-time component *SkipListNode.dll*.

As we mentioned above, our .NET application can perform search operations within data elements of a skip list using serial, threaded, and parallelized instance methods, and simultaneously it is able to measure the execution times of particular methods. By comparing these times, we have examined the execution efficiency of parallel search operations in a skip list in an experiment.

5 Experiment, its Results and their Brief Analysis

We assume that parallel search operations of our .NET application searching for students (data elements) in the skip list according to several required parameters are more execution efficient than serial and threaded search operations of the same .NET application searching for the same students in the same skip list according to the same several required parameters. With an increasing number of data elements in the skip list, the execution efficiency of parallel search operations should increase compared to the execution efficiency of serial and threaded search operations.

To verify these assumptions, we performed an experiment using our .NET application. During this experiment, our .NET application worked with 4 large sets of student data, which were stored in 4 disk files *students_c.txt*, *students100c.txt*, *students200c.txt* and *students300c.txt*. For each search, the .NET application always loaded data from one of these four disk files into its skip list. The first set of data after loading into the skip list of the .NET application contained 11 data elements with structured data of students. The other 3 data sets, also after loading into the .NET application skip list, contained 100, 200 and 300 data elements with structured data of students. The .NET application searched for students (data elements) in each of these data sets loaded in the skip list according to several required parameters.

During the first search for students (data elements) in a skip list with 11, 100, 200 and 300 data elements according to 4 (77; 61; 69; 41) and 10 (77; 61; 69; 41; 55; 89; 65; 93; 78; 75) required points for accommodation, we gradually measured the execution times of the *SerialSearchBtnClick* serial instance method, the *SearchOnMultipleThrdsBtnClick* threaded instance method, the *ParallelInvokeSearchBtnClick* parallel instance method, and the *ParallelForEachSearchBtnClick* parallel instance method using a .NET application.

During the second search for students (data elements) in a skip list with the same 11, 100, 200 and 300 data elements according to 4 (137; 65; 112; 454) and 10 (137; 65; 112; 454; 5; 101; 98; 78; 139; 6) required distances of residence of students from the university, we used a .NET application to gradually measure the execution times of the same four instance methods.

The .NET application, which searched for the required data in the skip list and measured the execution times of particular searches, was running on a computer with the following basic hardware configuration: Intel Core i5-8250U Processor (6 MB Cache, 1.60 GHz, 4 GT/s, 4 Cores, 8 Threads), RAM: 8 GB. The Microsoft Windows 10 Home, 64 bit operating system and the Microsoft .NET Framework 4 were installed on this computer.

One of the outputs of our .NET application displayed in its text box after searching for students according to 4 required parameters, distances between their residence and the university: 137; 65; 112; 454 [km] in the skip list with 100 data elements, is shown in Fig. 8. The same output writes the .NET application into the *LogFile.txt* disc file. The .NET application writes the results of other search operations into the same *LogFile.txt* file, too.

```
[          ISIC      surname    first name  (points for acc.)  residence (dist. to univ.)  date of the birth]

36104758538294289  Novak  Peter      (69) Humenne  (454)  2000-03-09
36105758538295289  Pelak  Peter      (70) Humenne  (454)  2000-03-10
36106758538296289  Pelak  Jan        (71) Humenne  (454)  2000-03-11
36104278107495531  Sykora  Frantisek  (77) Dubnica-nad-Vahom  (137)  1999-12-06
36072128769834583  Sykora  Andrej     (77) Dubnica-nad-Vahom  (137)  1999-12-05
36105278107496531  Sykora  Dezider    (78) Dubnica-nad-Vahom  (137)  1999-12-07
36073128769835583  Sykora  Peter      (78) Dubnica-nad-Vahom  (137)  1999-12-06
36106278107497531  Sykora  Filip      (79) Dubnica-nad-Vahom  (137)  1999-12-08
36074128769836583  Sykora  Roman      (79) Dubnica-nad-Vahom  (137)  1999-12-07
36107278107498531  Sykora  Arpad      (80) Dubnica-nad-Vahom  (137)  1999-12-09
36075128769837583  Sykora  Jakub      (80) Dubnica-nad-Vahom  (137)  1999-12-08
36100427025140126  Tell  Viliam       (93) Sturovo  (112)  2000-01-01
36130757022430100  Jankovic  Jan      (75) Piestany  (65)  2002-06-04

  Search time: 00:00:00.0013853, 1,3853 ms
```

Fig. 8 The output of our .NET application displayed in its text box after searching for students according to 4 required parameters, distances between their residence and university: 137; 65; 112; 454 [km] in a skip list with 100 data elements

We calculated the speedups of every parallel instance method using p computing elements (CPUs) by the following formula

$$Speedup(p) = T_{serial}/T(p)$$

The term T_{serial} refers to the execution time of the serial version of the given method.

The execution times of search operations performed by search instance methods of our .NET application on all four sets of skip lists and speedups of parallel search instance methods of this application are shown in the following graphs (Fig. 9).

Brief Results Analysis. From the graphs showing execution times of serial, threaded, and parallel search instance methods and speedups of parallel search instance methods of the object of our .NET application in dependency on the number of data elements in a skip list, it is obvious that parallel methods have significantly shorter execution times and greater speedups at higher numbers of data elements compared to serial and threaded methods. A significant difference between these execution times can be seen at the number of data elements 200. Parallel instance methods achieve the highest speedups when searching in a skip list with 300 data elements and the smallest speedups (sometime less than 1, which is the deceleration) when searching in a skip list with 11 data elements.

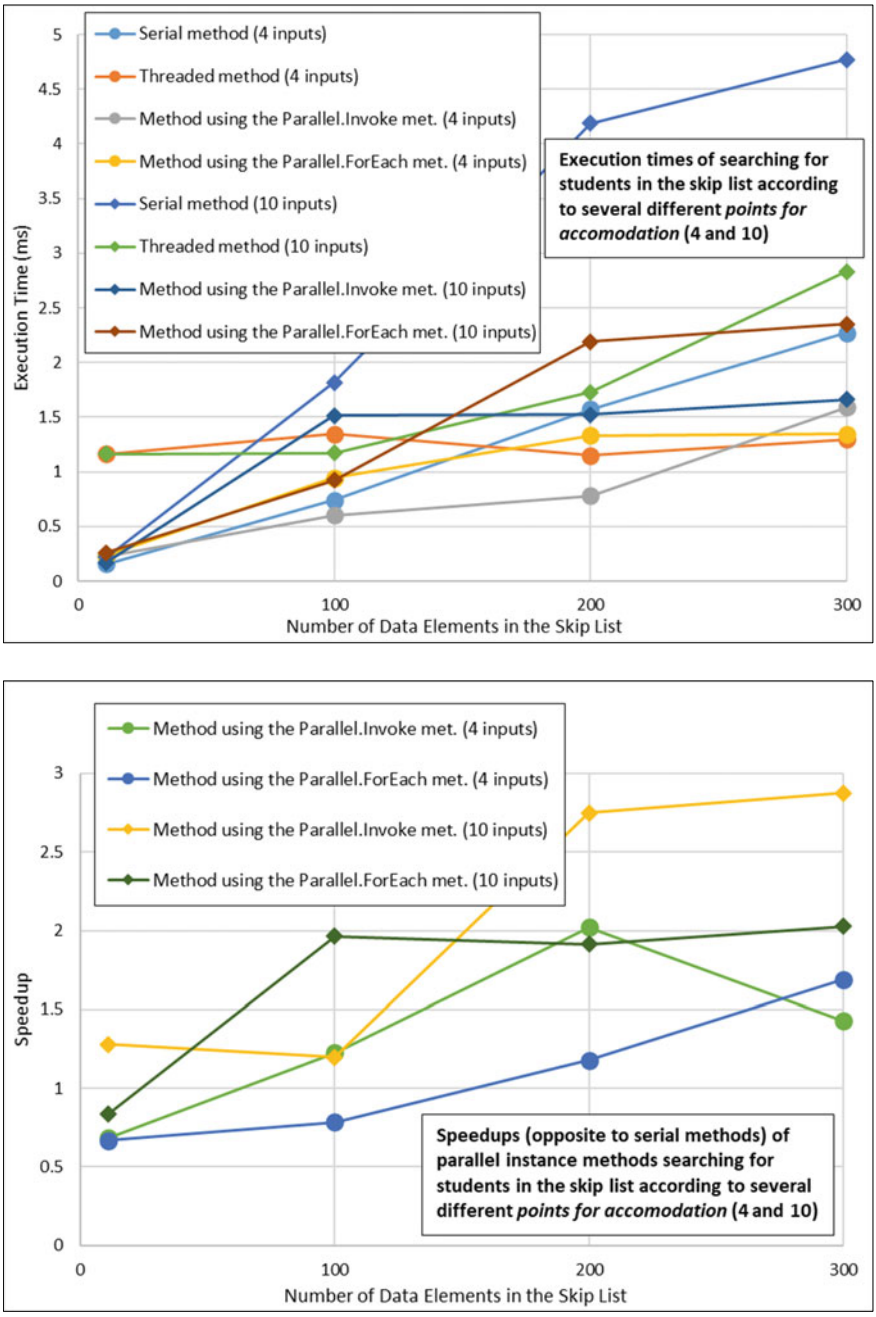
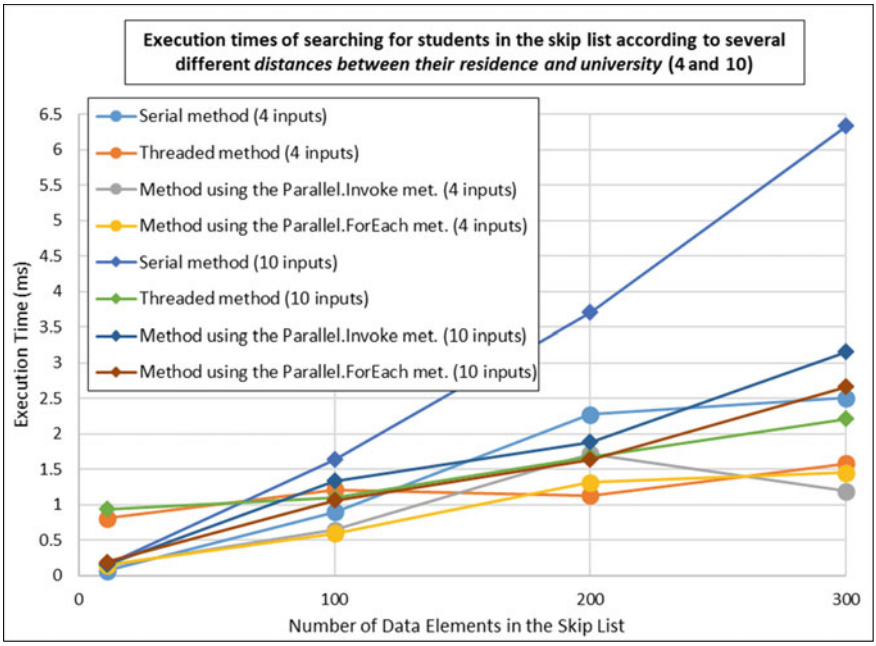


Fig. 9 The execution times of all search instance methods and speedups parallel search instance methods performed by our .NET application in the skip list



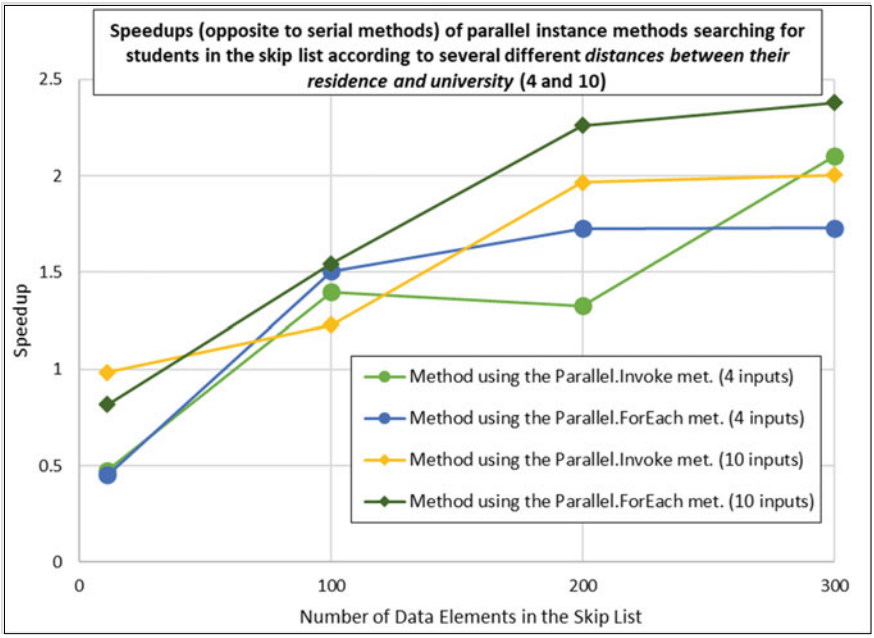


Fig. 9 (continued)

6 Conclusion

From the results of the comparison of the execution times of the serial, threaded, and parallel search instance methods and speedups of parallel search instance methods performed by our .NET application in the skip list, it is obvious that parallel instance methods of the object of this application are able to search required students (data elements) in a skip list with significantly shorter execution times at higher numbers of data elements than serial and threaded instance methods of the same object of this .NET application. From these comparison results, we can also say that for a skip list with small numbers of data elements containing structured data, e.g., 11 data elements, the execution efficiency of serial, threaded, and parallel search instance methods of the object of the .NET application is approximately the same and speedups of parallel methods are minimal, even sometime they are less than 1. However, if we use a big number of such data elements in a skip list of a .NET application, 200 or more, it is a good choice to use parallel search instance methods of the object of this application to search for required students (data elements) in a skip list.

References

1. Adamchik, V.S.: Linked Lists. http://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html. Accessed 17 July 2021
2. Košťál, I.: Comparison of execution efficiency of the use of a skip list and simple list in a .NET application. In: Laukaitis, G. (eds.) Recent Advances in Technology Research and Education Proceedings of the 17th International Conference on Global Research and Education Inter-Academia – 2018. LNNS, vol. 53, pp. 252–259. Springer, Cham (2019)
3. Microsoft Corporation: Documentation. https://docs.microsoft.com. Accessed 17 July 2021
4. Niemann, T.: Sorting and Searching Algorithms. epaperpress.com (1999)
5. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)