LINQ and Performance

Steffen Ploetz
16 Nov 2022 CPOL

Is LINQ the right technology for processing large amounts of data in runtime-relevant environments?

Anyone using LINQ should not only be aware of the advantages, but also pay attention to the disadvantages - performance is definitely one of the significant considerations.

Introduction

It has been 15 years (2007) since Microsoft introduced LINQ in .NET 3.5. That should be enough time for it to mature into an all-around useful tool.

The use of LINQ has several aspects:

- Deferred execution and lazy evaluation (better utilization of multi-threading capable systems)
- Shortening of the code (readability and reusability)
- Runtime overhead of the technologies used (performance)

In this article, I will take a look at one aspect of LINQ only - performance.

Background

LINQ has - at least in my environment - electrified all C# programmers with its appearance and subsequently led to massive use of this technology.

I would like to ask here, how much of the marketing shine remains, if one looks at LINQ completely soberly - as an innovative combination of extension methods and anonymous delegates (lambda expressions) with deferred execution (yield)?

To answer this question for the performance aspect, I want to filter an object list with LINQ and sort the result.

The first specificity of this approach is that LINQ has to process the list up to the last element. Thus, I **intentionally eliminate** runtime advantages that can be achieved using `Any()` or `First()` and focus on cases where it is mandatory to process all elements.

The second specificity of this approach is that LINQ works against the main memory (and not against a database). Thus, I **intentionally eliminate** the runtime influences that accesses to a database would introduce - no matter if it is a (small) in-memory database, a classic server database or a hybrid of them. One typical application of LINQ - database queries - is therefore not reasonably represented in the tests. This is because adding database queries to the tests would also be well suited to falsify the results:

- The effectiveness of LINQ database queries is influenced more by the database driver than by the programming approach.
- If the LINQ database queries are executed against a cached database, the results then still come from main memory.
- How stupid does the programming approach have to be to be compared to LINQ? I would expect stored procedures (parameterizable and already compiled SQL queries) to be vastly superior to LINQ queries.

Update 1: In the comments section, there are a lot of comments that address the self-imposed restriction of not using LINQ for database accesses in the tests. They range from criticism (this leaves out the essential use case) to agreement (stored SQL procedures are the better way for efficient database queries anyway). With my approach, I only and exclusively wanted to eliminate the side effects that database accesses would bring to the tests. No more and no less.

Tests

Of course, the results depend on the multithreading capability of the underlying runtime system, operating system and hardware - so only comparing different approaches with each other that are running in the same environment is useful. For this reason, I have prepared 4 test cases that can be compared with each other.

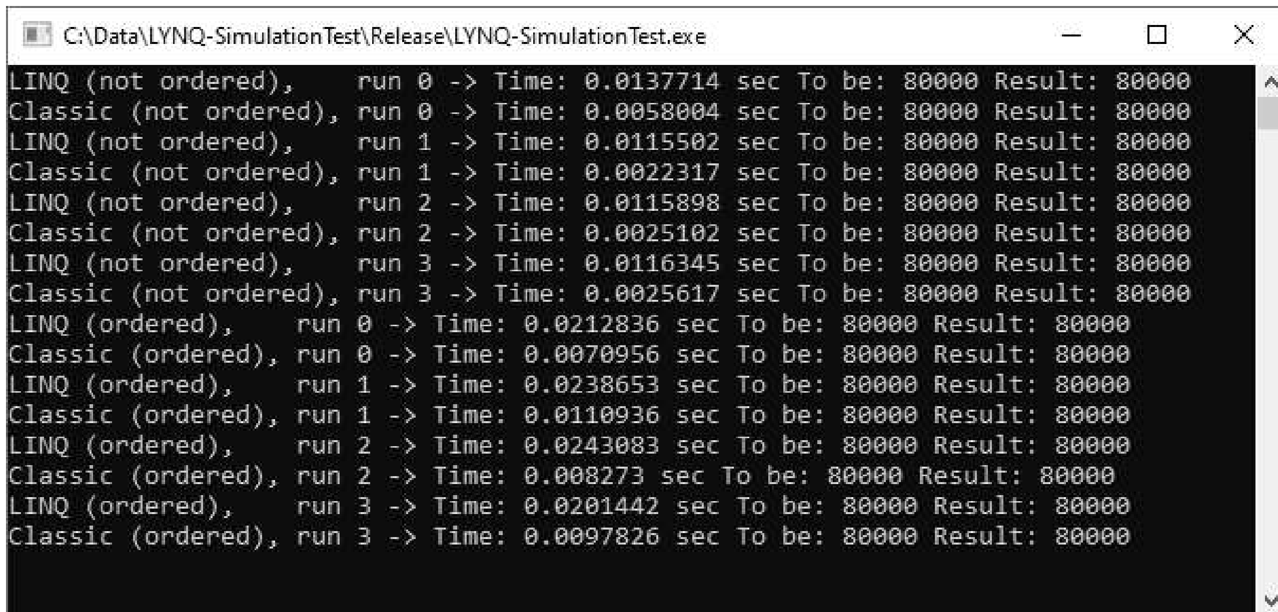
- **Test case 1**: LINQ with C++ (without and with ordering)
- **Test case 2**: classic (`for(...)` loops, `if(...)` and `sort(...)`) with C++ (without and with ordering)
- **Test case 3**: LINQ with C# (without and with ordering)
- **Test case 4**: classic (`for(...)` loops, `if(...)` and `sort(...)`) with C# (without and with ordering)

And the test cases will be executed on two different environments ...

Test on Win10

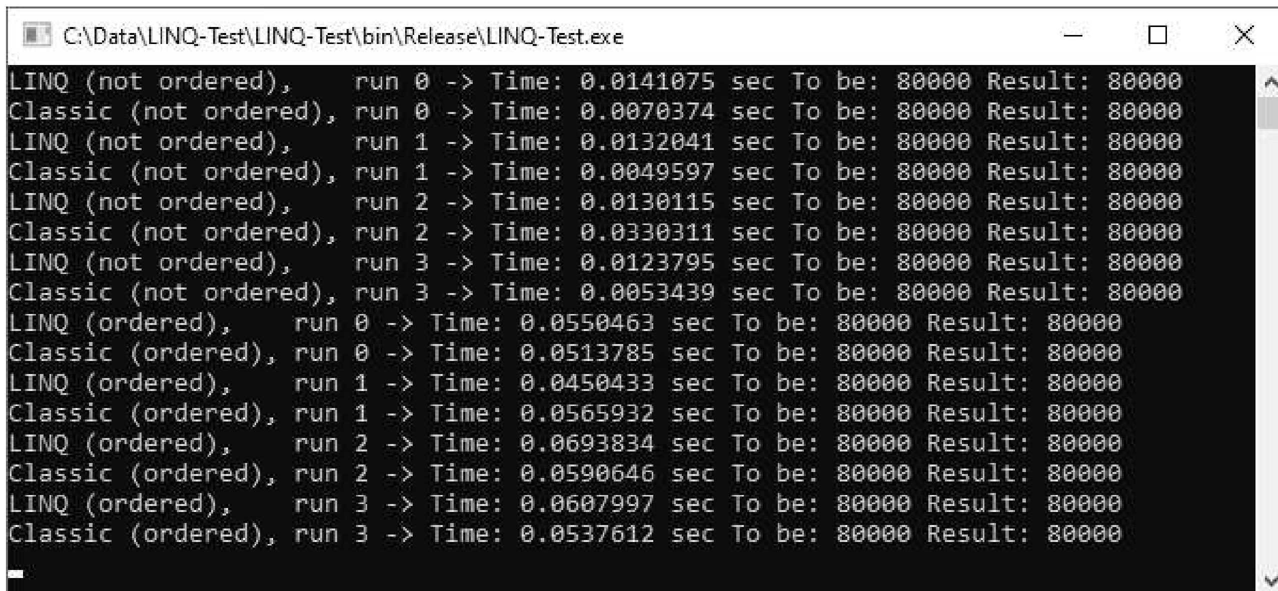
The first test environment is a Lenovo T550 with i7-5600U @ 2.6GHz, Windows 10 x64, .NET 4.7.2 and 16GB memory.

The C++ test results are:



- The C++ code has been compiled for x86 architecture with ISO C++ 14 Standard and /O2 optimization flag.
- I've tested x64 architecture as well - but without observable differences.

The C# test results are:



Test on openSUSE

The second test environment is a Lenovo T520 with i5-2540M @ 2.6GHz, openSUSE Leap 15.4 x64, .NET 6.0 and 12 GB memory.

Update 1: When I first published the tests, .NET 6.0 was current and .NET 7.0 was just around the corner. Many comments in the comments section raise the question if the results won't be much better with .NET 7.0. At the end of the article I have added measurements that cover this issue.

The C++ test results are:

```
Terminal - steffen@localhost:...orTest/bin/Release
File  Edit  View  Terminal  Tabs  Help
steffen@p200300cc9f030f67c4efee171ed7750d:~/Projects/CodeBlocks/LINQ-SumulatorTes
t/bin/Release> ./LINQ-SumulatorTest
LINQ (not ordered),    run 0 -> Time: 0.008394298 sec To be: 80000 Result: 80000
Classic (not ordered), run 0 -> Time: 0.002444084 sec To be: 80000 Result: 80000
LINQ (not ordered),    run 1 -> Time: 0.006051372 sec To be: 80000 Result: 80000
Classic (not ordered), run 1 -> Time: 0.001291909 sec To be: 80000 Result: 80000
LINQ (not ordered),    run 2 -> Time: 0.00683667 sec To be: 80000 Result: 80000
Classic (not ordered), run 2 -> Time: 0.001244364 sec To be: 80000 Result: 80000
LINQ (not ordered),    run 3 -> Time: 0.005269867 sec To be: 80000 Result: 80000
Classic (not ordered), run 3 -> Time: 0.001236021 sec To be: 80000 Result: 80000
LINQ (ordered),    run 0 -> Time: 0.011532112 sec To be: 80000 Result: 80000
Classic (ordered), run 0 -> Time: 0.005204993 sec To be: 80000 Result: 80000
LINQ (ordered),    run 1 -> Time: 0.010443046 sec To be: 80000 Result: 80000
Classic (ordered), run 1 -> Time: 0.005546519 sec To be: 80000 Result: 80000
LINQ (ordered),    run 2 -> Time: 0.010595727 sec To be: 80000 Result: 80000
Classic (ordered), run 2 -> Time: 0.005195087 sec To be: 80000 Result: 80000
LINQ (ordered),    run 3 -> Time: 0.011429437 sec To be: 80000 Result: 80000
Classic (ordered), run 3 -> Time: 0.005281643 sec To be: 80000 Result: 80000
```

- The C++ code has been compiled with GNU C++ 12-lp154 defaults and /O3 optimization flag.

The C# test results are:

```
Terminal - steffen@localhost:...e/net6.0/linux-x64 (on p200300cc9f030fad32636d4a8fa7ba67.dip0.t-ip
File  Edit  View  Terminal  Tabs  Help
steffen@p200300cc9f030fad32636d4a8fa7ba67:~/Projects/dotNET/LINQ-Test/bin/Releas
e/net6.0/linux-x64> ./LINQ-Test
LINQ (not ordered),    run 0 -> Time: 0.0182580 sec To be: 80000 Result: 80000
Classic (not ordered), run 0 -> Time: 0.0081393 sec To be: 80000 Result: 80000
LINQ (not ordered),    run 1 -> Time: 0.0167326 sec To be: 80000 Result: 80000
Classic (not ordered), run 1 -> Time: 0.0092186 sec To be: 80000 Result: 80000
LINQ (not ordered),    run 2 -> Time: 0.0160955 sec To be: 80000 Result: 80000
Classic (not ordered), run 2 -> Time: 0.0096272 sec To be: 80000 Result: 80000
LINQ (not ordered),    run 3 -> Time: 0.0163441 sec To be: 80000 Result: 80000
Classic (not ordered), run 3 -> Time: 0.0078963 sec To be: 80000 Result: 80000
LINQ (ordered),    run 0 -> Time: 0.0446850 sec To be: 80000 Result: 80000
Classic (ordered), run 0 -> Time: 0.0336500 sec To be: 80000 Result: 80000
LINQ (ordered),    run 1 -> Time: 0.0331585 sec To be: 80000 Result: 80000
Classic (ordered), run 1 -> Time: 0.0310524 sec To be: 80000 Result: 80000
LINQ (ordered),    run 2 -> Time: 0.0349818 sec To be: 80000 Result: 80000
Classic (ordered), run 2 -> Time: 0.0303455 sec To be: 80000 Result: 80000
LINQ (ordered),    run 3 -> Time: 0.0322253 sec To be: 80000 Result: 80000
Classic (ordered), run 3 -> Time: 0.0332981 sec To be: 80000 Result: 80000
```

Comparison of the Results

To better compare the results, I ignore the worst test run (highest run time) and take the median of the three remaining test runs. Which produces this summary picture:

line	system	HW	code	attempt	ordering	median	C++ vs. C#	LINQ vs. classic
1	Win 10	T550	C++	LINQ	no	11.5 ms	13 % better	~3 times worse
2	Win 10	T550	C#	LINQ	no	13.0 ms	13 % worse	
3	Win 10	T550	C++	classic	no	2.5 ms	112 % better	~3 times better
4	Win 10	T550	C#	classic	no	5.3 ms	112 % worse	
5	Win 10	T550	C++	LINQ	yes	21.2 ms	159 % better	inconsistent (C++ ~2.5 times worse, C# comparable)
6	Win 10	T550	C#	LINQ	yes	55.0 ms	159 % worse	
7	Win 10	T550	C++	classic	yes	8.2 ms	554 % better	inconsistent (C++ ~2.5 times better, C# compaable)
8	Win 10	T550	C#	classic	yes	53.7 ms	554 % worse	
9	openSUSE	T520	C++	LINQ	no	6.0 ms	171 % better	~3.5 times worse
10	openSUSE	T520	C#	LINQ	no	16.3 ms	171 % worse	
11	openSUSE	T520	C++	classic	no	1.2 ms	575 % better	~3.5 times better
12	openSUSE	T520	C#	classic	no	8.1 ms	575 % worse	
13	openSUSE	T520	C++	LINQ	yes	10.5 ms	206 % better	inconsistent (C++ ~2 times worse, C# comparable)
14	openSUSE	T520	C#	LINQ	yes	32.2 ms	206 % worse	
15	openSUSE	T520	C++	classic	yes	5.2 ms	496 % better	inconsistent (C++ ~2 times better, C# comparable)
16	openSUSE	T520	C#	classic	yes	31.0 ms	496 % worse	

Discussion of the Expectations and Results

C++ vs. C#

- I expect a clear advantage of C++ over C#.
- Expectation met.

C++ is up to ~5.5-times faster than C# - this is not dramatic for applications, but can become a knockout criterion for platforms and services. Obviously, STL and C++ compiler optimizers do an excellent job here.

Update 1: There are some comments in the comments section that seem to refer to the comparison between C++ and C# being unfair.

The core points are the statements that C++ was invented to develop close to hardware (operating systems, platforms, drivers, ...) and C# was invented to develop fast and least error-prone (application system level).
In general, I share the view, even if I have a problem with "inventing" - after all, it was an evolutionary development.

Nevertheless, I generally agree with the thesis, in a few cases I also use a combination of C++ and C# for my real projects at the application system level - because even with .NET Core or .NET 7.0, mixed mode DLLs can be written with little effort.

LINQ vs. classic (*for(...) loops, if(...) and sort(...)*)

- I expect LINQ to have a clear disadvantage over the classic approach.
- Expectation met. However, not as clearly as expected.

LINQ is always worse than the classic approach (`for(...)` loops, `if(...)` and `sort(...)`) for C++ but not for C#.

I think the reason LINQ holds up so well in C# is because of the special data structures. While LINQ in C++ uses `vector<Element>` for source data and results, LINQ in C# uses `List<Element>` for source data and `System.Linq.Enumerable.WhereListIterator<Element>` or `System.Linq.OrderedEnumerable<Element, int>` for results. The second `datatype` provides explicit access to the sort criterion and could also be seen as no longer comparable or cheating. But let's be mild.

Win10 vs. openSUSE

I don't want to comment it further, but the tests seem to run faster om Linux with much weaker hardware. Maybe it's because of Microsoft Defender - which is ALWAYS running in my Windows machines.

Scaling

The quite acceptable results for C# compared to C++ surprised me (I had expected at least one power of ten as the difference) as well as pleased me (apparently 15 years of development have really had a positive effect on C#).

Nevertheless, I wanted to be sure that C# is really that strong. So I simply multiplied the amount of data twenty times from `640000` to `12800000`.

The scaled C++ test results are:

```
Terminal - steffen@localhost:...orTest/bin/Release
File  Edit  View  Terminal  Tabs  Help
steffen@p200300cc9f030f67c4efee171ed7750d:~/Projects/CodeBlocks/LINQ-SumulatorTest/bin/R
elease> ./LINQ-SumulatorTest
LINQ (not ordered),    run 0 -> Time: 0.11326329 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 0 -> Time: 0.036335732 sec To be: 1600000 Result: 1600000
LINQ (not ordered),    run 1 -> Time: 0.10891088 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 1 -> Time: 0.036110437 sec To be: 1600000 Result: 1600000
LINQ (not ordered),    run 2 -> Time: 0.109496879 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 2 -> Time: 0.035541164 sec To be: 1600000 Result: 1600000
LINQ (not ordered),    run 3 -> Time: 0.108317698 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 3 -> Time: 0.035936728 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 0 -> Time: 0.232962613 sec To be: 1600000 Result: 1600000
Classic (ordered), run 0 -> Time: 0.121536514 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 1 -> Time: 0.235924944 sec To be: 1600000 Result: 1600000
Classic (ordered), run 1 -> Time: 0.123634458 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 2 -> Time: 0.237758469 sec To be: 1600000 Result: 1600000
Classic (ordered), run 2 -> Time: 0.125234768 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 3 -> Time: 0.236946256 sec To be: 1600000 Result: 1600000
Classic (ordered), run 3 -> Time: 0.125044982 sec To be: 1600000 Result: 1600000
```

The scaled C# test results are:

```
Terminal - steffen@localhost:...e/net6.0/linux-x64 (on p200300cc9f030fad32636d4a8fa7ba67.dip0.t-ipconnect.
File  Edit  View  Terminal  Tabs  Help
steffen@p200300cc9f030fad32636d4a8fa7ba67:~/Projects/dotNET/LINQ-Test/bin/Release/net6
.0/linux-x64> ./LINQ-Test
LINQ (not ordered),    run 0 -> Time: 0.2308481 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 0 -> Time: 0.1075582 sec To be: 1600000 Result: 1600000
LINQ (not ordered),    run 1 -> Time: 0.2078712 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 1 -> Time: 0.1065754 sec To be: 1600000 Result: 1600000
LINQ (not ordered),    run 2 -> Time: 0.1718491 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 2 -> Time: 0.1089847 sec To be: 1600000 Result: 1600000
LINQ (not ordered),    run 3 -> Time: 0.1714981 sec To be: 1600000 Result: 1600000
Classic (not ordered), run 3 -> Time: 0.1057252 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 0 -> Time: 0.6818162 sec To be: 1600000 Result: 1600000
Classic (ordered), run 0 -> Time: 0.6836711 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 1 -> Time: 0.6363989 sec To be: 1600000 Result: 1600000
Classic (ordered), run 1 -> Time: 0.6521390 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 2 -> Time: 0.6361483 sec To be: 1600000 Result: 1600000
Classic (ordered), run 2 -> Time: 0.6487451 sec To be: 1600000 Result: 1600000
LINQ (ordered),    run 3 -> Time: 0.9221625 sec To be: 1600000 Result: 1600000
Classic (ordered), run 3 -> Time: 0.8348759 sec To be: 1600000 Result: 1600000
```

Which produces this summary picture:

<table>
<tr><th>line</th><th>system</th><th>HW</th><th>code</th><th>attempt</th><th>ordering</th><th>median</th><th>C++ vs. C#</th><th>LINQ vs. classic</th></tr>
<tr><td>1</td><td>openSUSE</td><td>T520</td><td>C++</td><td>LINQ</td><td>no</td><td>108.9 ms</td><td>58 % better</td><td rowspan="2">~2.5 times worse</td></tr>
<tr><td>2</td><td>openSUSE</td><td>T520</td><td>C#</td><td>LINQ</td><td>no</td><td>171.8 ms</td><td>58 % worse</td></tr>
<tr><td>3</td><td>openSUSE</td><td>T520</td><td>C++</td><td>classic</td><td>no</td><td>35.9 ms</td><td>197 % better</td><td rowspan="2">~2.5 times better</td></tr>
<tr><td>4</td><td>openSUSE</td><td>T520</td><td>C#</td><td>classic</td><td>no</td><td>106.7 ms</td><td>197 % worse</td></tr>
<tr><td>5</td><td>openSUSE</td><td>T520</td><td>C++</td><td>LINQ</td><td>yes</td><td>250.9 ms</td><td>153 % better</td><td>inconsistent (C++</td></tr>
</table>

line	system	HW	code	attempt	ordering	median	C++ vs. C#	LINQ vs. classic
6	openSUSE	T520	C#	LINQ	yes	636.4 ms	153 % worse	~2 times worse, C# comparable)
7	openSUSE	T520	C++	classic	yes	135.1 ms	383 % better	inconsistent (C++ ~2 times better, C# comparable)
8	openSUSE	T520	C#	classic	yes	652.1 ms	383% worse	

Discussion

All statements already made for the smaller data set are confirmed.

Using the Code

Update 1: The following code (C++ and C#) represents the initial version (first code generation) of my tests except for bug fixes (that have been integrated meanwhile). Many comments in the comment section make suggestions to make the code more efficient. This ranges from more modern notations (e.g. for the properties) to recommendations for benchmark libraries.

First of all, thanks for that! I have tried/implemented many of these suggestions. For the suggestions to make the C++ code more efficient, it turned out that the GNU C++ compiler optimizes so efficiently that the code changes had no measurable effects. For the suggestions to make the C# code more efficient, with a lot of good will you can read an improvement into the measurement results - but the C# compiler also optimizes very efficiently.

From the suggestions for benchmark libraries, I only took the idea of providing a clean interface for result evaluation and presenting the results more compactly and clearly. The benchmark libraries for C++ and C# differ so much that it would affect the comparability of the approaches too much and so I decided to use a self-implemented solution that works exactly identically in C++ and C#.

I am happy to provide the enhanced code here:
Improved code for C++ Download LINQ-SimulationTest.zip
Improved code for C# Download LINQ-Test.zip

The C++ Test Code (first code generation)

The code is exactly the same for Win10 and openSUSE.

C++

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <chrono>
#include "boolinq.h"
```

```
using namespace std;
using namespace boolinq;

static const size_t VECTORSIZE = 640000;

class Element
{
private:
    int tag;
    int order;
    int data;

public:
    int getTag() const { return tag; }
    void setTag(int t) { tag = t; }
    int getOrder() const { return order; }
    void setOrder(int o) { order = o; }
    int getData() const { return data; }
    void setData(int d) { data = d; }

    inline bool operator<(const Element& b)
    {
        return this->order * VECTORSIZE + this->data < b.order* VECTORSIZE + b.data;
    }
};

void checkResults(vector<Element> resultVector)
{
    int d = 0;
    int o = 0;
    for (auto it = resultVector.begin(); it != resultVector.end(); it++)
    {
        if (d > it->getData() && o > it->getOrder())
            throw "Ordering failed";
        d = it->getData();
        o = it->getOrder();
    }
}

void testClasic(vector<Element> testVector, bool order, int run)
{
    auto startTime = chrono::high_resolution_clock::now();

    vector<Element> resultVector;

    for (auto it = testVector.begin(); it != testVector.end(); it++)
    {
        if (it->getTag() < 1)
        {
            resultVector.push_back(*it);
        }
    }
    if(order)
        sort(resultVector.begin(), resultVector.end());

    if (resultVector.size() != VECTORSIZE / 8)
        throw "Number of result elements not correct!";
```

```
    checkResults(resultVector);

    auto endTime = chrono::high_resolution_clock::now();

    auto ordering = (order ? "ordered" : "not ordered");
    double timeTaken = chrono::duration_cast<chrono::nanoseconds>
                          (endTime - startTime).count() / 1000000000.0;
    cout << "Classic (" << ordering << "), run " << run << " -> Time: "
         << setprecision(9) << timeTaken << " sec" << " To be: "
         << VECTORSIZE / 8 << " Result: " << resultVector.size() << endl;
}

void testLINQ(vector<Element> testVector, bool order, int run)
{
    auto startTime = chrono::high_resolution_clock::now();

    auto resultVector = (order ?
        from(testVector).where([](const Element& element)
                            { return element.getTag() < 1; })
                        .orderBy([](const Element& element)
                            { return element.getOrder() * VECTORSIZE +
                                     element.getData(); })
                        .toStdVector() :
        from(testVector).where([](const Element& element)
                            { return element.getTag() < 1; })
                        .toStdVector());
    if (resultVector.size() != VECTORSIZE / 8)
        throw "Number of result elements not correct!";

    checkResults(resultVector);

    auto endTime = chrono::high_resolution_clock::now();

    auto ordering = (order ? "ordered" : "not ordered");
    double timeTaken = chrono::duration_cast<chrono::nanoseconds>
                           (endTime - startTime).count() / 1000000000.0;
    cout << "LINQ (" << ordering << "),    run " << run << " -> Time: "
         << setprecision(9) << timeTaken << " sec" << " To be: "
         << VECTORSIZE / 8 << " Result: " << resultVector.size() << endl;
}

int main()
{
    vector<Element> testVector(VECTORSIZE);
    for (size_t index = 0; index < VECTORSIZE; index++)
    {
        testVector[index].setTag(index % 8);
        testVector[index].setOrder(index % 12);
        testVector[index].setData(index);
    }

    for (int run = 0; run < 4; run++)
    {
        testLINQ(testVector, false, run);
        testClasic(testVector, false, run);
    }
```

```
    for (int run = 0; run < 4; run++)
    {
        testLINQ(testVector, true, run);
        testClasic(testVector, true, run);
    }

    int i;
    cin >> i;

    return 0;
}
```

The C# Test Code (first code generation)

The code is exactly the same for Win10 and openSUSE.

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

namespace LINQtest
{
    public static class Settings
    {
        public const int VECTORSIZE = 640000;
    }

    class Element : IComparable<Element>
    {
        private int tag;
        private int order;
        private int data;

        public Element(int t, int o, int d)
        {
            tag = t;
            order = o;
            data = d;
        }

        public int Tag { get { return tag; } set { tag = value; } }
        public int Order { get { return order; } set { order = value; } }
        public int Data { get { return data; } set { data = value; } }

        public int CompareTo(Element? b)
        {
            if (b == null)
                return 1;
            return (b.order * Settings.VECTORSIZE + b.data) -
                   (order * Settings.VECTORSIZE + data);
        }
```

```
    }

    class Program
    {
        static void checkResults(IEnumerable<Element> resultVector)
        {
            int d = 0;
            int o = 0;
            foreach (var element in resultVector)
            {
                if (d > element.Data && o > element.Order)
                    throw new Exception("Ordering failed");
                d = element.Data;
                o = element.Order;
            }
        }

        static void testClasic(List<Element> testVector, bool order, int run)
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();

            var resultVector = new List<Element>(testVector.Count / 8);

            foreach (var element in testVector)
            {
                if (element.Tag < 1)
                    resultVector.Add(element);
            }

            if (order)
                resultVector.Sort();

            if (resultVector.Count != Settings.VECTORSIZE / 8)
                throw new Exception("Number of result elements not correct!");

            checkResults(resultVector);

            sw.Stop();

            var ordering = (order ? "ordered" : "not ordered");
            Console.WriteLine(
                "Classic ({0}), run {1} -> Time: {2} sec To be: {3} Result: {4}",
                ordering, run, sw.Elapsed.ToString().Substring(7),
                Settings.VECTORSIZE / 8, resultVector.Count());
        }

        static void testLINQ(List<Element> testVector, bool order, int run)
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();

            var resultVector = (order ?
                    testVector.Where(element => element.Tag < 1)
                              .OrderBy(element => element.Order *
                                       Settings.VECTORSIZE + element.Data) :
                    testVector.Where(element => element.Tag < 1)).ToList();
```

```
            if (resultVector.Count() != Settings.VECTORSIZE / 8)
                throw new Exception("Number of result elements not correct!");

            checkResults(resultVector);

            sw.Stop();

            var ordering = (order ? "ordered" : "not ordered");
            Console.WriteLine(
                "LINQ ({0}),    run {1} -> Time: {2} sec To be: {3} Result: {4}",
                ordering, run, sw.Elapsed.ToString().Substring(7),
                Settings.VECTORSIZE / 8, resultVector.Count());
        }

        static void Main(string[] args)
        {
            var testVector = new List<Element>(Settings.VECTORSIZE);
            for (int index = 0; index < Settings.VECTORSIZE; index++)
            {
                testVector.Add(new Element(index % 8, index % 12, index));
            }

            for (int run = 0; run < 4; run++)
            {
                testLINQ(testVector, false, run);
                testClasic(testVector, false, run);
            }
            for (int run = 0; run < 4; run++)
            {
                testLINQ(testVector, true, run);
                testClasic(testVector, true, run);
            }

            var key = Console.ReadKey();
        }
    }
}
```

Code Compilation on Win10

I used Visual Studio 2019 projects for C++ and C#.

Code Compilation on openSUSE

To compile the C++ code, I created a Code::Blocks 20.3 project and used the `gcc-c++ 12`.

To compile the C# code I used NET6.0 SDK and this shell script:

Shell

```
#               |<-- The path to the *.scproj file
|<-- The target folder (the result will also be in the bin/Release folder).
```

```
dotnet publish . --configuration Release --framework net6.0 --output ./pub
                 --self-contained false --runtime linux-x64 --verbosity quiet
```

Update 1: In the meantime, I upgraded from .NET 6.0 to .NET 7.0. All I had to do for the conversion of the tests was to change from `net6.0` to `net7.0` in the shell script and in the project file (*.csproj).

Points of Interest

I wanted to know how far apart LINQ and the classic approach (`for(...)` loops, `if(...)` and `sort(...)`) are in their respective best manifestations.

It turns out that for C++, the classic approach is the most performant while for C#, the use of LINQ is recommended. And there seem to be good reasons for this. While LINQ for C++ only means additional overhead, LINQ in C# can more than compensate this additional overhead by the special data structures (`System.Linq.Enumerable.WhereListIterator<Element>` and `System.Linq.OrderedEnumerable<Element, int>`).

Update 1: In the meantime, I upgraded from .NET 6.0 to .NET 7.0. The following shows the measurement results on openSUSE (on Win10 it has already been shown that the spread of the measurement results hardly allows reliable statements).

These measurement results were obtained with the enhanced code, the more compact and clearer output of the measurement results (new clean benchmark interface) and the increase of the runs to 9 evaluated runs.

I repeated the tests 3 times - and could not detect any acceleration on .NET 7.0:

```
Platform environment: Unix
Runtime environment:  .NETCoreApp,Version=<code>v6.0</code>

| Measurement set       |     N |       Mean |      Error |     StdDev | Ratio |
| --------------------- | ----- | ---------- | ---------- | ---------- | ----- |
| LINQ (not ordered)    |     9 |    0.197 s |    0.037 s |    0.020 s |  1.00 |
| Classic (not ordered) |     9 |    0.106 s |    0.000 s |    0.000 s |  0.54 |
| LINQ (ordered)        |     9 |    0.658 s |    0.041 s |    0.019 s |  3.35 |
| Classic (ordered)     |     9 |    0.686 s |    0.037 s |    0.020 s |  3.49 |

| Measurement set       |     N |       Mean |      Error |     StdDev | Ratio |
| --------------------- | ----- | ---------- | ---------- | ---------- | ----- |
| LINQ (not ordered)    |     9 |    0.197 s |    0.038 s |    0.020 s |  1.00 |
| Classic (not ordered) |     9 |    0.106 s |    0.000 s |    0.000 s |  0.54 |
| LINQ (ordered)        |     9 |    0.659 s |    0.041 s |    0.019 s |  3.35 |
| Classic (ordered)     |     9 |    0.690 s |    0.036 s |    0.019 s |  3.51 |

| Measurement set       |     N |       Mean |      Error |     StdDev | Ratio |
| --------------------- | ----- | ---------- | ---------- | ---------- | ----- |
| LINQ (not ordered)    |     9 |    0.196 s |    0.035 s |    0.019 s |  1.00 |
| Classic (not ordered) |     9 |    0.106 s |    0.001 s |    0.001 s |  0.54 |
```

```
| LINQ (ordered)        |     9 |   0.659 s  |   0.039 s  |   0.018 s  |  3.36 |
| Classic (ordered)     |     9 |   0.691 s  |   0.035 s  |   0.019 s  |  3.53 |

Platform environment: Unix
Runtime environment:  .NETCoreApp,Version=<code>v7.0</code>

| Measurement set       |     N |       Mean |      Error |     StdDev | Ratio |
| --------------------- | ----- | ---------- | ---------- | ---------- | ----- |
| LINQ (not ordered)    |     9 |   0.196 s  |   0.038 s  |   0.020 s  |  1.00 |
| Classic (not ordered) |     9 |   0.105 s  |   0.001 s  |   0.000 s  |  0.54 |
| LINQ (ordered)        |     9 |   0.661 s  |   0.050 s  |   0.025 s  |  3.38 |
| Classic (ordered)     |     9 |   0.764 s  |   0.035 s  |   0.019 s  |  3.91 |

| Measurement set       |     N |       Mean |      Error |     StdDev | Ratio |
| --------------------- | ----- | ---------- | ---------- | ---------- | ----- |
| LINQ (not ordered)    |     9 |   0.198 s  |   0.038 s  |   0.020 s  |  1.00 |
| Classic (not ordered) |     9 |   0.108 s  |   0.000 s  |   0.000 s  |  0.55 |
| LINQ (ordered)        |     9 |   0.660 s  |   0.042 s  |   0.020 s  |  3.33 |
| Classic (ordered)     |     9 |   0.778 s  |   0.036 s  |   0.018 s  |  3.93 |

| Measurement set       |     N |       Mean |      Error |     StdDev | Ratio |
| --------------------- | ----- | ---------- | ---------- | ---------- | ----- |
| LINQ (not ordered)    |     9 |   0.197 s  |   0.038 s  |   0.020 s  |  1.00 |
| Classic (not ordered) |     9 |   0.103 s  |   0.001 s  |   0.001 s  |  0.52 |
| LINQ (ordered)        |     9 |   0.657 s  |   0.042 s  |   0.019 s  |  3.33 |
| Classic (ordered)     |     9 |   0.771 s  |   0.036 s  |   0.019 s  |  3.91 |
```

History

- 1st November, 2022: Initial version
- 6th November, 2022: Correction: The sorting in C++ was not switched off. Consequently, all measurements for C++ had to be redone as well.
- 9th November, 2022: Correction: The sorting in C# was sorting the input data instead the result data, all measurements for C# had to be redone as well.
- 16th November, 2022: Update 1 - Enhanced code for C++ and C#, .NET 7.0

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Written By

Steffen Ploetz

CEO Symbioworld GmbH

Germany

I am currently the CEO of Symbioworld GmbH and as such responsible for personnel management, information security, data protection and certifications. Furthermore, as a senior programmer, I am responsible for the automatic layout engine, the simulation (Activity Based Costing), the automatic creation of Word/RTF reports and the data transformation in complex migration projects.

The main focus of my work as a programmer is the development of Microsoft Azure Services using C# and Visual Studio.

Privately, I am interested in C++ and Linux in addition to C#. I like the approach of open source software and like to support OSS with own contributions.

Comments and Discussions

53 messages have been posted for this article Visit **https://www.codeproject.com/Articles/5345621/LINQ-and-Performance** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Article Copyright 2022 by Steffen Ploetz
Everything else Copyright © CodeProject, 1999-2022

Web03 2.8:2022-11-08:1