PROJECT 7 - First Contact with Open .NET

Completed on 13-Feb-2016 (47 days) -- Updated on 19-Nov-2016

Introduction >

Open .NET Overview

Microsoft has decided to make the whole .NET Framework open source. It is a huge step in the right direction with lots of implications, although all this is outside the scope of the current project. Only the following code repositories are relevant here:

- CoreCLR. It contains the most basic parts. The majority of the code is written in (mostly unmanaged/unsafe) C# and C++.
- CoreFX. It contains the remaining parts of the .NET Framework. Most of the code is written in (mainly managed) C#, but also in C++ & VB.NET.
- Roslyn. It contains the last versions of the C#/VB.NET compilers (and all what is related to compilation/Visual Studio, including a new subsystem delivering relevant insights about the whole process). Most of the code is written in (mainly managed) C# & VB.NET.

As shown in the corresponding page, I am mostly experienced in (managed) C# and VB.NET, also in algorithm-building/efficiency-improvement. From the point of view of the programming language, Roslyn (or even CoreFX) seems more appealing to me; on the other hand, the exact programming language is not a big problem, mainly when dealing with properly-working existing codes and by bearing in mind my relevant expertise in different environments.

This first open .NET project deals with CoreCLR because of providing the kind of access to basic parts which is more appealing to a code-improver like me (at least for open-source contributions, this is my favourite role). More specifically, this project is mostly focused on `ParseNumber` (and on all the code related to it), which takes care of the string-analysis actions needed by the parsing methods of all the .NET numeric types (although this project is mostly focused on `decimal.Parse` and `decimal.TryParse`). It is contained in the file Number.cs, which is part of *mscorlib.dll* where the most essential .NET functionalities are defined.

OVER-8-MONTH-LATER UPDATE

The PR associated with this project was accepted by the .NET team and merged into the CoreCLR code; the proposed modifications were even extended to other repositories (i.e., CoreRT and CoreFX).

Most of the contents of the current project are still valid, although the following issues have to be taken into account:

- I preferred to not remove certain parts which, despite being somehow obsolete, contain valuable ideas.

All the contents which are related to *CoreRun.exe* belong to this category. Note that, when this project was written, the user was forced to rely on that program directly via command prompt (not the case with the current Core-based approach). To know more about this specific issue, take a look at the also-updated Project 8.

- Additionally to comments on these lines at the bottom of various sections, I added the new Tests > Additional Tests.

First Contributions

I firstly analysed Roslyn because of seeming the a-priori-most-appealing-to-me repository (as already explained), although this experience wasn't too good. Additionally and unlikely what happens with CoreCLR/CoreFX, this code is not too intuitively appealing to someone in my situation (i.e., experienced .NET programmer expecting to easily locate "recognisable parts") as far as it is mainly focused on the C#/VB.NET compilers. In any case, I will most likely contribute to Roslyn in the future.

The aforementioned not-too-positive episode is helpful to understand why I quickly liked CoreCLR so much. This time, I could easily find the code associated with virtually any feature (e.g., all what is contained in the C#/VB.NET `System` namespace). Having full access to the pure essence of the .NET Framework, my primary programming environment during the last few years! That's why it didn't take me too long to post my two first issues (i.e., some hours after having downloaded the code for the first time and on the next day):

- ISSUE #2285 "Problems with thousands separators when parsing strings to numbers"
- ISSUE #2290 "Decimal type not able to parse scientific notation"

Both of them are related to the behaviour of the current version of `decimal.Parse` and, indirectly, to other parse methods of this and different numeric types. Curiously, I didn't realise about any of these issues (i.e., thousands separators behaving unexpectedly and scientific notation not being fully supported) at work, but while answering questions in stackoverflow.com (my SO profile). In any case, I thought that both behaviours were worth correcting, that's why I created those issues and actively participated in the associated discussions.

After reading most of the comments in the aforementioned threads, anyone should easily understand that I didn't enjoy this experience too much. The fact that both issues were closed and the associated proposals rejected didn't bother me even a bit; on the other hand, I did find the behaviours of some participants completely unacceptable, mainly in the issue #2285. Anyone interested in knowing more about what happened there can take a look at both threads and at the associated external references (IMO all this is a good condensed summary of most of my online self-describing efforts, one of the requirements of the new Attitude 2.0, a required-but-very-unappealing-to-me work).

In general terms, I am quite happy with various outputs of this curious experience; for example: very clear ideas about what the .NET community implies (i.e., lots of people, not necessarily too knowledgeable or reasonable) or the format of my contributions there (i.e.,

not-too-controversial-issues + forking + pulling + waiting). Additionally, I got various good advices and met some quite knowledgeable people.

General Impression

I support any attempt to share relevant knowledge with a big enough community and, consequently, any form of open software. Additionally to these generic open-is-good ideas, the fact that most of the code is precisely written in .NET (C# & VB.NET) is a further motivation for me: full access to the pure essence of my favourite programming languages which is precisely written in such languages!

The code in CoreCLR (and also in the other repositories described in the first section) is huge, but also well-structured/-commented; an experienced enough programmer can start messing around with it right away. The compilation process is not simple and its outputs are not precisely excellent, as explained in other parts of this project (e.g., CoreRun.exe). The distribution among different languages (managed/unmanaged C# & C++; and also VB.NET in other repositories) is not as systematic as it should ideally be. On the other hand, this is a quite logical output for a so big project where relevant code migrations have occurred; additionally, experienced programmers shouldn't have problems when dealing with more-or-less-generic algorithms (i.e., not heavily relying on the specific features of the given programming environment) in properly-working codes almost independently upon the used language. The high quality of almost each bit of code in CoreCLR is also somehow relevant here; for example: the fact of having worked on a virtually-perfect piece of software (+ my I-have-to-get-something-good-here motivation) explains the highly-optimised output of this project (i.e., modified version of `ParseNumber`), not too likely to be created under different conditions even by someone like me who is always keen on writing efficient codes.

Regarding the contribution process, I am still not in a position to have a worthy opinion on this front (after having completed this project and pulled the modified version of `ParseNumber`, I will update this part with my impressions). I had already a first interaction with the .NET community which didn't like too much (as already explained), although I take this kind-of-weird episode as a good lesson learned. In fact, I don't even have a bad opinion about the .NET community, just a better-than-just-merely-intuitive understanding of it. Lastly, I am still not sure about what to think regarding the .NET team attitude, although everything looks nice so far (will also update this part after the future forking-pulling-waiting process will be over).

In summary, I have got a very good first impression (+ a somehow-problematic episode which helped me understand better the "local peculiarities") and am certainly looking forward to contributing to this project in a relevant way.

OVER-8-MONTH-LATER UPDATE

I have had quite a few other interactions with the .NET team/community in CoreCLR/CoreFX/Roslyn and my overall impression is quite positive. There are many people

from different backgrounds and with different expectations, but most of the discussions are meaningful and the contributions relevant.

The .NET team behaviour is certainly remarkable: quick and reasonable responses for virtually any situation. Some important decisions (e.g., non-trivial PR acceptance) are understandably associated with quite long waiting times.

Code Overview >

CoreCLR

As explained in the first section, CoreCLR is one of the three major open .NET repositories where this project is precisely focused. It contains the most basic parts of the .NET Framework:

- The .NET Core runtime ("CoreCLR").
- The base library ("mscorlib").
- The garbage collector.
- The JIT compiler.
- Data types and low-level classes.

Most of the code in the CoreCLR repository is written in unmanaged/unsafe C# and in C++. My experience in both languages is quite limited (although I am very experienced in managed C#/.NET), but this fact will certainly not be a problem here because of the following two reasons:

- I specialise in algorithm building and rely on a quite minimalistic approach (i.e., rarely use the most complex features of the given programming environment). The defining feature of unmanaged C# is precisely keeping everything very simple from the in-built-feature-utilisation point of view.
- All my contributions to the open .NET project are expected to be focused on improving existing-and-already-quite-optimised codes, an aspect where my eventually-not-too-extensive experience in the given language is still less important. Note that I am also planning to work on new implementations, but most of this code is expected to be very similar to the existing one. For example: I will not be writing the code of the future `decimal.TryParseExact` method (which will be accounting for various currently-unsupported behaviours, like the ones described in my first two issues in the CoreCLR repository #2285 and #2290) completely from scratch, but will adapt the one of `Decimal.TryParse` (and its associated methods, like `ParseNumber`).

The current project deals with the optimisation of `ParseNumber` (i.e., part of the *mscorlib* library, inside the file Number.cs), which takes care of all the string-analysis actions of the .NET numeric types; this project is mostly focused on `Parse`/`TryParse` of `decimal`, but also accounts for `int`, `long` & `double`. Note that all these parsing methods are written in unmanaged/managed C# & C++, although `ParseNumber` (and, consequently, this project) is mostly written in unmanaged C#.

Methods

As explained in the first section, I got a quite good impression about the overall quality of the open .NET code (i.e., not just in CoreCLR, but also in CoreFX and Roslyn): very clear structure, efficient algorithms, descriptive comments, etc. All the parts analysed here (i.e., various methods in the file Number.cs) are certainly not an exception to this statement.

This project is focused on the optimisation of `ParseNumber` and associated resources. All its code is written in (almost-exclusively-)unmanaged C#, although C++ is also used in *Number.cs* even in parts which are closely related to `ParseNumber`. The following three different methods will be considered:

- `Boolean ParseNumber(ref char* str, NumberStyles options, StringBuilder sb, Boolean parseDecimal)`: it analyses the inputs to the given parsing method, like `Decimal.Parse` (i.e., string to be converted into number and arguments taking care of additional issues, like culture-related format); then outputs the numerical characters which will be later transformed into the given type (i.e., `decimal`) by `NumberBufferToDecimal`. All this code is unmanaged and highly optimised (e.g., relevant usage of bitwise operations). Most of the performance-improvement modifications of this project were precisely done to this method.
- `char* MatchChars(char* p, char* str)`: it is called from `ParseNumber`, while looping through the characters in the input string, and returns the position after the given target (`str`). There is a second overload (`char* MatchChars(char* p, string str)`) allowing the target to also be `string`.
- `Boolean IsWhite(char ch)`: very simple method checking whether the given character is blank. Anecdotally and unlikely the two aforementioned methods, this is managed code.

Ideally, this algorithm should be optimised by completely rethinking how the problem is being faced; specifically, the not-too-practical/efficient way in which the input information is stored provokes unnecessary problems. For example: over-complicated/less-efficient algorithms or over-usage of resources (e.g., most of potential inputs aren't analysed). Nevertheless, such an expectation is beyond the scope of the current project and its final goal (i.e., pull request to CoreCLR, whose likelihood to be accepted would be much lower in case of including modifications in existing features). The `decimal.TryParseExact` method which I am planning to create in the near future (likely to be part of Project 9) will certainly take care of these issues, also account for various functionalities not supported by the current parsing approaches.

CoreRun.exe

This project is about improving part of the CoreCLR source code; that's why recompiling the whole code (or, at least, the *mscorlib* library) seems an a-priori basic requirement.

Compiling all the code in the CoreCLR repository is not completely straightforward (although step-by-step instructions are available); even after being generated, the new libraries (e.g., *mscorlib.dll*) cannot be used right away, a quite logical consequence of what is being affected (i.e., some of the basic libraries used by all the .NET applications running on that computer).

CoreRun.exe is one of the outputs generated when building the whole CoreCLR repository and allows any .NET executable to rely on the newly-created libraries. This file is so peculiar that the whole Project 8 is precisely focused on analysing its security implications.

The most relevant-to-this-project feature of *CoreRun.exe* is to have a noticeable impact on the performance of the given .NET file: the same executable under exactly the same conditions can be orders of magnitude slower when run via *CoreRun.exe*. Just this fact might not be too relevant on account of the goals of the current project (i.e., relative measurements of performance, where 5 vs. 10 is identical to 50 vs. 100), but these variations are definitely not-consistent what does represent an immediate deal-breaker.

Standalone applications will be used in all the performance-comparison tests; they will be run directly (i.e., as conventional executables on Windows), rather than via *CoreRun.exe*. On the other hand, *CoreRun.exe* will be used in all the tests confirming the identity between old/new versions of the code (i.e., all the numeric-type parsing methods, like `decimal.Parse`, tested against a big set of inputs by relying on the old/new *mscorlib.dll*).

Tests >

Preliminary Ideas

Unexpectedly, I have spent most of my time here by trying to come up with the best approach to measure the performance differences between both methods (i.e., old & new versions of `ParseNumber`). The fact that the original version was already optimised is the main reason explaining so many difficulties in this in-principle-easy part. Additionally, the optimisation process wasn't precisely straightforward (e.g., I found some of the Irrelevant Issues quite surprising), what avoided me to have clear enough ideas regarding the expected measurements (i.e., being sure about the exact effects of certain modification would have been helpful to quickly spot problems with the tests).

Although the basic structure of the main program didn't change appreciably since the start, the whole testing approach (i.e., the C# program, its inputs and the way in which the time differences were measured) had many relevant changes. Roughly speaking, it passed through the following stages:

- Firstly, I relied on *CoreRun.exe* because of assuming that this was the best way to emulate realistic conditions. As already explained, this assumption was quickly proven wrong: *CoreRun.exe* has an important negative (and not consistent) impact on the given .NET executable performance. Nevertheless, *CoreRun.exe* is used for the last validation stage, where the modified `ParseNumber` version is tested under as-realistic-as-possible conditions; this is done with ParseNumber_Validation.exe which iterates through the `Parse` and `TryParse` methods of various types (i.e., `decimal`, `int`, `long` and `double`).
- For my first tests without *CoreRun.exe*, I relied on two different executables (e.g., *new.exe* & *old.exe*). But back then the gap between both approaches wasn't too

relevant and running two different programs represented an unacceptable increase in uncertainty. That's why I didn't try this option for too long.

- Even before moving to the both-in-the-same-file approach, I was aware about the associated increase of uncertainty (i.e., running one method affects the time measurements of the other one). After trying different ways to minimise this influence (e.g., setting different pauses at different points, pre-warming or affecting `GC`), I confirmed that the most reliable methodology was alternating the order in which the versions were measured (note that I also tested a random-order approach which was proven less reliable).
- Then I moved back to two different executables; also tried to make the testing algorithm as efficient as possible to confirm whether these effects (i.e., smaller pauses between consecutive calls) had a relevant effect on the observed performance differences. The resulting application was the first version of ParseNumber_Test2.exe, the definitive testing program. There were some relevant changes after this, but all of them are listed in the corresponding section.

Testing Approaches

As already explained, finding out the best testing environment wasn't easy and that's why I tried different approaches. Roughly speaking, I used (and created) two programs: ParseNumber_Test.exe and ParseNumber_Test2.exe. The first one accounts for both old (i.e., original `ParseNumber` code) and new (i.e., modified `ParseNumber`, expected to be notably quicker) versions; and the second one only deals with one version every time (i.e., two different executables have to be generated); additionally, the second version is much more efficient (e.g., minimalist time tracking).

Regarding `ParseNumber` and related methods (i.e., `MatchChars` and `IsWhite`), there have always been two different classes: `New` for the improved versions and `Old` for the original ones. I have added slight-and-not-affecting-performance modifications in both of them to minimise the code size; it has to be noted that the original version of `ParseNumber` (i.e., the one used inside *mscorlib.dll*) relies on `internal` classes which cannot be accessed via Visual Studio.

The basic structure of the two aforementioned programs is formed by the following three nested loops:

- The main and most external loop performs a `finalMax` number of iterations. The version (i.e., `Old.ParseNumber` or `New.ParseNumber`) which is considered by all the loops below is defined here (i.e., the specific type of method to consider: `0` for `New.ParseNumber` and `1` for `Old.ParseNumber`).
- The second loop iterates through the contents of *inputs.txt*. This file includes randomly-varying numbers (one per line) and is generated by ParseNumber_Gen.exe. Four different types of inputs can be generated: `decimal`, `int`, `long` and `double`; all the tests were focused on `decimal` and that's why this feature is only required by ParseNumber_Validation.exe. An additional issue to bear in mind is that the values from *inputs.txt* might be altered on account of the `NumberStyles` value under consideration, as explained below.

- The third and most internal loop iterates through a list of `NumberStyles` and calls the corresponding `ParseNumber` version. Initially, it accounted for all the possible values (i.e., `NumberStyles enum` members) with no further modification; for example: calling `Old.ParseNumber` (or `New.ParseNumber`) with the inputs `"5.5"` & `NumberStyles.AllowDecimalPoint`, `"5.5"` & `NumberStyles.AllowParentheses` and so on. Later, I started to account for special cases, where the given `NumberStyles` value is associated with certain modifications in the input string; for example: `"5"` & `NumberStyles.AllowDecimalPoint`, but then `"(5)"` & `NumberStyles.AllowParentheses`. Note that `ParseNumber` is used under many different input conditions, like the `Parse`/`TryParse` methods of all the numeric types (e.g., `decimal`, `int`, `double`, etc.), and that not all the `NumberStyles` values are always effective; for example: `NumberStyles.AllowHexSpecifier` and `NumberStyles.HexNumber` only make sense*** with integer types (e.g., `int` or `long`). On the other hand and after confirming the similarities among different types, the performance tests were completely focused on `decimal`; the remaining main numerical types were exclusively considered during the final validation via *CoreRun.exe* (i.e., by using ParseNumber_Validation.exe). Nevertheless, I did enough tests to conclude that the input conditions don't have a relevant effect on the observed new-old performance differences; anyone is welcome to confirm this point by taking advantage of the easily-modifiable structure of all the involved codes.

  *** NOTE: despite not being supported, the situation is valid in appearance (i.e., Visual Studio doesn't show any kind of warning or error for these erroneous inputs). For example: `decimal` cannot deal with hexadecimal inputs and, consequently, `decimal.Parse("FFFFFFFF", NumberStyles.AllowHexSpecifier)` triggers an error; what doesn't happen with supported alternatives, like `long.Parse("FFFFFFFF", NumberStyles.AllowHexSpecifier)`. These situations represent, in my opinion, an additional reason for creating the new `decimal.TryParseExact` (I am planning to work on it during the next months and, most likely, make it part of the new Project 9); even for redefining the parsing actions of all the numeric types (at least, the `NumberStyles` usage).

The conditions under which the tests were performed also varied in a quite relevant way. The configuration proven to deliver the most stable measurements is defined by the following:

- Closing all the running applications which might affect the measuring process.
- Running the given program three (or even five) times, one after the other, by storing the final values (e.g., `sw.ElapsedMilliseconds` in the last version of *ParseNumber_Test2.exe*). With *ParseNumber_Test2.exe*, this process needs to be repeated with the two programs (i.e., considering `New.ParseNumber` and `Old.ParseNumber` respectively).
- Inputting all the aforementioned values in ParseNumber_TestCalcs.exe to get the final measurements. Note that this program outputs three main variables: `averageGapNew` & `averageGapOld` which indicate the % variation of each new(old) value with respect to all the remaining new(old) values; and `averDiff` which is the % difference between the averages of all the old/new values. If the measurements are stable enough (i.e., `averageGapNew` & `averageGapOld` below 1%), the final result would be given by `averDiff`; otherwise, the process would have to be repeated under different conditions (i.e., higher `finalMax` and/or elements in *inputs.txt*).

I performed all the tests on two different computers with the following configuration:

- Computer 1: 3.4 GHz and 12 GB of RAM. Windows 10 Pro 64-bit.
- Computer 2: 2.27 GHz and 3 GB of RAM. Windows 8.1 N 64-bit.

The results from the tests were always different on account of the computer under consideration; although they were very consistent for each computer. I spent a relevant amount of time without being able to find a set of conditions minimising the differences between both machines; my conclusion is that reaching this goal might be possible, but only under extreme conditions (i.e., relevant number of inputs/iterations, what would also provoke disproportionately high old-new differences). As explained in the next section, the most important result of the tests has been getting very consistent and easily-reproducible measurements (where the new version has always been faster), rather than specific values.

Definitive Tests

The option of measuring both methods together (i.e., ParseNumber_Test.exe) was quickly proven too unreliable, mainly with big sets of inputs. That's why I relied on ParseNumber_Test2.exe during most of the testing process. Nevertheless, this program passed through various relevant modifications:

- My initial intention when trying two different programs (i.e., one for `New.ParseNumber` and another one for `Old.ParseNumber`) was to get more insights into the unstable behaviour of *ParseNumber_Test.exe*. Additionally, I wanted to know whether a more efficient testing program (i.e., running `ParseNumber` under more demanding conditions) might be more beneficial for one of the versions.
- While testing this new version with *PerfView.exe* (i.e., expressly recommended in the CoreCLR documentation to measure performance variations), I realised that the comparisons might be based upon the outputs of this profiler (e.g., process or CPU time). And this is where the second stage of *ParseNumber_Test2.exe* started: I removed all its internal time measurements and relied on the *PerfView.exe* outputs. *ParseNumber_Test2.exe* became much more efficient and I could confirm that `New.ParseNumber` performs better under more demanding conditions. Curiously, this change occurred at the same time than a tiny-but-influential bug appeared in the `New` class (i.e., a `'\0'` in one of the `MatchChars` overloads was replaced with a `'0'`). This bug provoked the new version to be notably faster, a variation which I assumed that was provoked by the relevant modifications in *ParseNumber_Test2.exe*. As a consequence of this curious episode, I relied on the *PerfView.exe*-based approach for some days (i.e., longer than what would have happened in other scenario) and published wrong information in social media (i.e., in my Twitter and GitHub accounts).
- After the aforementioned bug was fixed and the new-old gap dropped drastically, I tried to further-optimise the testing program and the first decision was removing *PerView.exe* from the picture; also replaced the old time measurements with the simplistic end-minus-start-times. This is precisely the last version which I used in the final tests referred below.

For all the final performance tests, I used the conditions described in the previous section. Main ideas:

- 10000 iterations of the main loop in *ParseNumber_Test2.exe* (i.e., `finalMax = 10000`); and 20000 records in *inputs.txt*, generated by *ParseNumber_Gen* (i.e., `totInputs = 20000`).
- Both programs *new.exe* (i.e., accounting for *New.ParseNumber*) and *old.exe* (i.e., accounting for *Old.ParseNumber*) were run three times, one after the other, and all the final values (i.e., `sw.ElapsedMilliseconds`) stored.
- The aforementioned measurements were input into ParseNumber_TestCalcs.exe to determine the final results (i.e., `averDiff`, the difference between the averages of both sets of values as %), by assuming that the measuring process is valid (i.e., `averageGapNew` and `averageGapOld` below 1%). Note that these minimum conditions of validity have always been met with the aforementioned inputs.

Even despite the numerous attempts and relevant testing efforts, I am still not in a position to deliver an absolutely valid (i.e., suitable to be easily tested anywhere else) result about the new-old differences, other than: the new one is certainly quicker. If I could set my own computers as an absolute reference of validity I would say the following:

- On the less powerful computer (computer 2), you can easily (i.e., under the proposed conditions; but even under less strict ones, like 5000 iterations & 10000 inputs) get a 6.3-6.5% difference.
- The most powerful computer (computer 1) used to deliver 7.0-7.5% with a previous version. Now, it should be able to reach 8% and above; although a problem with one of the last Windows 10 updates has made this computer too unstable to confirm this assumption (by bearing in mind the aforementioned statement: the exact value isn't too relevant).

On the other hand, a notable increase in the input conditions (e.g., 50000 inputs) might provoke the aforementioned values to be notably bigger; or even by using a different approach (i.e., the old *ParseNumber_Test.exe*). Same conclusions when using *PerfView.exe*: the new version will always be notably better in all the aspects (i.e., lower CPU/process time and CPU usage), but the exact values will change depending upon the computer and the input conditions.

Additional Tests

Originally, I added the current section to comply with the .NET team request of further validating my proposal with the CoreFX tests; although this option was quickly proven inadequate.

The CoreFX tests don't refer to the code in the CoreCLR repository; that's why I had to look for some CoreFX methods similar enough to the ones being modified here (i.e., `MatchChars` and `ParseNumber` in Number.cs). In fact, I found the exact same methods in the file FormatProvider.Number.cs (note that these versions were also modified).

Unfortunately, I could only find one test accounting for the aforementioned code: the parse test for `BigInteger` in System.Runtime.Numerics; this wasn't precisely good news because the proposed improvements are much more noticeable in decimal types. This test was quickly deemed irrelevant. In any case, note that I firstly misinterpreted the not-saying-much results

as a consequence of relying on *CoreRun.exe* (the Core-based alternative uses this program internally). Bear in mind that all the references in this project to *CoreRun.exe* come from equivalent not-necessarily-applicable-anymore ideas; also that the updated version of Project 8 deals with this specific issue.

By following the advice of a .NET team member, I developed a much simpler testing program which, despite performing notably worse than the other tests on my computer (but reaching the never-seen-before 10% on his computer!), allowed this PR to be finally accepted and merged.

Improvements >

Introduction

As already explained, I firstly came with the idea for this project after some interactions with the .NET community about `decimal.Parse`. `ParseNumber` is the method in charge of performing most of the string-analysis actions which are triggered when calling `decimal.Parse`/`decimal.TryParse` (or `Parse`/`TryParse` of any other numeric type). The main goal of this project is to come up with a more efficient version of `ParseNumber` (& related methods). Although most of my efforts have been focused on `decimal` (i.e., big proportion of the performance-comparison tests), I also considered other input situations (i.e., accounting for additional types in a few performance tests to confirm the `decimal` conclusions or when doing the final *CoreRun.exe*-based validation with ParseNumber_Validation.exe).

The whole optimisation process wasn't too easy; but setting up a reliable performance-comparison framework was much more difficult, as explained in the sections of the previous part (Preliminary Ideas, Testing Approaches & Definitive Tests). I also found various issues whose effects on performance were proven less important than expected; all of them are included in the next section.

The pulled modifications were thoroughly tested under different conditions and confirmed to improve the performance of `ParseNumber` (and, consequently, of all the .NET numerical-type parsing methods). Additionally to getting lots of valuable insights into various issues (e.g., code optimisation of unsafe C# and performance measurements under extreme conditions), I also realised about some limitations in the current parsing methodology (what represented an additional confirmation of the need of the upcoming `decimal.TryParseExact`).

Irrelevant Issues

I have a relevant expertise improving (the performance of) random algorithms and C# is one of my favourite languages; I do have limited unmanaged-C# experience, but as already explained this shouldn't be a problem. Despite of all that, I did find some difficulties while working on this optimisation; even various issues whose effect on performance (i.e., shorter/longer

execution time of a standalone application including the modified version of `ParseNumber`) was different than expected. On the other hand, this is a more or less logical consequence of the true motivation behind the current optimisation process: my previously-recognised extra-motivation to get a good enough result no matter what.

Below these lines, I am including a list of modifications of the original code which were proven to not have a positive effect on performance.

- The immediate idea coming to my mind after first seeing the code was replacing the `option` binary operations (e.g., `(options & NumberStyles.AllowLeadingWhite) != 0`) with variables, because their values never changed. Binary operations are certainly quick, but wouldn't it be quicker to rely on a variable rather than performing the same operation various times? This assumption was proven wrong: creating a new (`NumberStyles` or even `Boolean`) variable is slower than the original version, where binary operations are performed every time. Additionally, relying on updated-in-each-iteration variables to store `state` binary operations (e.g., `(state & StateSign) == 0`) isn't a good idea either.
- I also tried the approach in the previous point with constants. For example: `const Int32 StateParens2 = -0x0003` to convert `state &= ~StateParens` into `state &= StateParens2`. This time, getting a worse performance was less surprising; the only chunks of code allowing such a configuration, the aforementioned one and `state |= StateSign | StateParens`, aren't too relevant (i.e., virtually no time is spent there in any input scenario).
- The aforementioned ideas are not applicable to non-binary operations. For example: replacing all the occurrences of `bigNumber` with its `(sb != null)` equivalence is slower.
- There was a specific modification which seemed to deliver a better performance in the first tests, but which was finally proven inadequate (i.e., indifferent from a performance point of view and, consequently, not part of the modifications to be pulled). It consisted in replacing the `char` variables with their `int` equivalences, but only when being involved in mathematical operations. For example: converting `ch >= '0' && ch <= '9'` into `ch >= 48 && ch <= 57`.

Final Version

This whole project has become much more complex than initially expected. The most difficult part was coming up with a reliable and overall-applicable testing framework to accurately measure the performance variations. Nevertheless, the results of this extra-effort are certainly worthy: not only a more efficient `ParseNumber` version, but also lots of relevant insights about performance improvement (comparison and testing) under very demanding conditions.

I did come up with the first good enough version of the new code relatively quickly, but didn't make the final decision about certain parts until after all the tests were completed. In fact, I found various issues whose almost irrelevant effect on performance was kind of surprising to me; all of them are mentioned in the previous section.

The last version of the modified code (i.e., the one in my pull request to CoreCLR) includes various changes in `ParseNumber` and in the second `MatchChars` overload with respect to the

original version. The table below these lines includes a reference to each of them, together with a % estimation of its contribution to the overall improvement (as defined in the last section of the Tests part).

Modification	**Contribution (%)**
`altdecSep` & `altgroupSep` removal	40
`signflag` removal	30
Redefinition of `IsWhite(ch)`-headed conditions	12
Redefinition of `MatchChars(char* p, char* str)`	11
`bigNumberHex` removal	7

All the aforementioned modifications refer to variable and method names present in the original version of the code. The justification for all these improvements should be more or less evident, except perhaps the `bigNumberHex` removal. Note that this modification implies to remove a `Boolean` variable (`bigNumberHex`) by replacing it with the less-efficient chunk of code `(bigNumber && ((options & NumberStyles.AllowHexSpecifier) != 0))`. Such a trade-off is positive because of happening in a part which is rarely used (i.e., all the benefits of removing one variable, but almost none of the slower-code drawbacks). Note that `bigNumberHex` is only used with hexadecimal inputs (i.e., a not-too-likely scenario); and even in this situation, exclusively with the non-numerical characters (note that a hexadecimal value might consist only in numbers). Thus, the affected condition is verified most of the times through the first short-circuited term by ignoring the new slower code.).

Conclusions

The first conclusion is that meeting the main goal of this project (i.e., improving the performance of `ParseNumber` & related methods) has been more difficult than expected; as a result of this unplanned-complexity, quite a few relevant outputs have been generated. Examples: various open-source applications (i.e., ParseNumber_Test.exe, ParseNumber_Test2.exe, ParseNumber_Gen.exe, ParseNumber_TestCalcs.exe & ParseNumber_Validation.exe), good insights into the best methodologies to accurately measure time differences between similar programs or more knowledge about performance improvements in unmanaged-C# codes. In general, this has been a very good first experience in open .NET (also my first open-source contributions ever).

Just by looking at the original goal of this project (i.e., more efficient version to be pulled to CoreCLR), the conclusion is that the final result has also been very good: the modified code is clearly quicker (i.e., every time and under all the testing conditions) without affecting the original algorithm at all and without adding any negative issue (e.g., lower readability or scalability).

The whole evolution of this project has also been a good mood-booster (and self-promotion). I chose the first chunk of code I saw in CoreCLR (by pure accident, while pre-analysing my

original `decimal`-related concerns, #2285 and #2290), which happened to be in one of the most basic, old and highly-optimised parts of the whole .NET Framework (i.e., numeric type parsing inside `mscorlib.dll`). I started this optimisation project even before having analysed the code in depth. Everything gets more complicated than expected, what makes me spend here much more time and effort than originally planned (by bearing in mind that this is a R&D, no-income-generating activity). And as a result of all this, I deliver what, in my opinion, is the most interesting & comprehensive project about programming since this site was created!

OVER-8-MONTH-LATER UPDATE

As explained in similar updates in other sections, this PR was accepted and even provoked modifications in other repositories (i.e., CoreRT and CoreFX versions of the same methods). Although it took quite long (over 8 months), I understand that modifications affecting a so essential part (i.e., used by the `Parse`/`TryParse` methods of all the numeric types) cannot be accepted right away. I am certainly happy with how everything went.