WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ELECTRONICS

FIELD: INFORMATICS
SPECIALIZATION: INTERNET ENGINEERING

MASTER THESIS

Application of concurrent programming techniques to optimize algorithms and software on the .NET platform

Zastosowanie technik programowania współbieżnego do optymalizacji algorytmów i oprogramowania na platformie .NET

AUTHOR:
Mikołaj Banaszkiewicz

SUPERVISOR:
PhD, Tomasz Kubik, K30W04D03

WROCŁAW, 2021

Abstract

Ever increasing demand for computing power makes parallel programming an invaluable tool for every software engineer. To fully reap the rewards of this paradigm, programmers need to make proper decisions at every step of software design. This thesis aims to alleviate this process with data driven recommendations for developers working on the .NET platform.

To arrive at the set of guidelines multiple versions of algorithms and software were implemented. .NET's Task Parallel Library, PLINQ, MapReduce and Fork/Join patterns and load balancing partitioners were used in the process. Each of the implementations was subjected to exhaustive tests using BenchmarkDotNet benchmarking library,

The results showed that theoretically sound parallel algorithm implemented poorly may even be slower than the unchanged sequential versions. Using functional programming concepts helped creating maintainable and readable solutions, decreasing the change of errors during implementation. Memory thread overhead did not show to be a major issue. Benchmarking at all stages of development proved to be imperative when programming for performance.

Keywords: .NET, Task Parallel Library, parallelism, multithreading, benchmarking

Streszczenie

WciÄ…ŁŽ rosnÄ…cy popyt na moc obliczeniowÄ… sprawia, ŁŽe programowanie rÄłwnolegŁ‚e to nieocenione narzÄ™dzie dla kaŁŽdego inŁŽyniera oprogramowania. By w peŁ‚ni czerpaÄ‡ z zyskÄłw oferowanych przez ten paradygmat, programiŁ›ci muszÄ… podejmowaÄ‡ odpowiednia decyzje na kaŁŽdym etapie projektowania aplikacji. Ta praca ma na celu uŁ‚atwienie tego procesu dla inŁŽynierÄłw pracujÄ…cych na platformie .NET poprzez przedstawienie rekomendacji opartych na danych.

Algorytmy oraz oprogramowanie zostaŁ‚o wielokrotnie zaimplementowane na rÄłŁŽne sposoby jako baza dla zestawu zaleceŁ„. Wykorzystano przy tym Task Parallel Library, PLINQ, wzorce MapReduce oraz Fork/Join i rÄłwnowaŁŽenie obciÄ…ŁŽenia danych. KaŁŽda z implementacji zostaŁ‚a poddana wyczerpujÄ…cy testom przy uŁŽyciu biblioteki BenchmarkDotNet.

Wyniki pokazaŁ‚y, ŁŽe teoretycznie poprawne rÄłwnolegŁ‚e algorytmy zaimplementowane w wÄ…tpliwy sposÄłb mogÄ… byÄ‡ wolniejsze niŁŽ niezmienione wersje sekwencyjne. UŁŽycie pojÄ™c pochodzÄ…cych z programowania funkcyjnego pozwoliŁ‚o na opracowanie czytelnych i utrzymywalnych rozwiÄ…zaŁ„ zmniejszajÄ…cych ryzyko popeŁ‚nienia bŁ‚edu przy implementacji. Koszty zwiÄ…zane z zarzÄ…dzaniem wÄ…tkami byŁ‚y znikome. Analiza porÄłwnawcza okazaŁ‚a siÄ™ byÄ‡ konieczna jako integralna czÄ™Ł›Ä‡ kaŁŽdego etapu projektowania rozwiÄ…zaŁ„ optymalnych wydajnoŁ›ciowo.

Słowa kluczowe: .NET, Task Parallel Library, programowanie rÄłwnolegŁ‚e, wielowÄ…tkowoŁ›Ä‡, analiza porÄłwnawcza

Contents

List of figures

List of tables

List of listings

Abbreviations

CPU *(Central Processing Unit)*
GPU *(Graphics Processing Unit)*
GPGPU *(General Processing on Graphics Processing Unit)*
SMT *(Simultaneous Multi-Threading)*
DLL *(Dynamic Link Library)*
CLR *(Common Language Runtime)*
GC *(Garbage Collector)*
LINQ *(Language INtegrated Query)*
PLINQ *(Parallel Language INtegrated Query)*
TPL *(Task Parallel Library)*

Chapter 1

Overview

One of the most popular observations in computer science is Moore's Law. It was introduced in the mid-1960s when Gordon E. Moore described an empirical relationship [18] projecting that the number of transistors per IC (Integrated Circuit) will increase twofold every year. He predicted that this trend will continue for at least a decade, but remarkably enough this is still true 50 years later. For most of that time CPU (Central Processing Unit) consisted of only one single processing core with performance growing through a combined increase of clock frequency speed and number of components per IC. At some point though, CPUs ran into a bottleneck, further clock frequency increases would come with drastically higher heat production and energy consumption [8]. Thus it became clear that to develop CPUs which will meet ever increasing demand for computing power, new architectural designs have to be engineered.

Before multi-core CPUs became commercialy available, Intel came up with a new idea of boosting performance, SMT (Simultaneous Multi-Threading) and it's implementation - Hyper-Threading. It was first included with Pentium 4 and Xeon processors in 2002. Architecturally, a processor with Hyper-Threading Technology consists of two logical processors per core, each of which has its own processor architectural state. Each logical processor can be individually halted, interrupted or directed to execute a specified thread, independently from the other logical processor sharing the same physical core. Unlike a traditional dual-processor configuration that uses two separate physical processors, the logical processors in a hyper-threaded core share the execution resources. Intel claimed that they would get a performance boost around 15 to 30% [12] compared to other non-Hyper- Threaded CPUs and only increase the size of the die with 5%. AMD in response developed the Bulldozer architecture, released in 2011, based on CMT (Cluster Multi-Threading), but it never delivered satisfactory performance increases, thus it was replaced by the SMT Zen architecture in 2017, which will be described later.

Soon after, in 2005, world's first dual core processors were introduced, first by AMD (Athlon 64 x2) then by Intel (Pentium D). Cores in such processors are separate physical entities and can execute instructions in a truly simultaneous way. In specific architectures cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods.

With the advent of multi-core processors came a complete programming paradigm shift. Performance gains, while substantial, require programmers to divide their programs into concurrent parts and synchronize access to data shared between threads. To fully reap the rewards of the new paradigm programs need to be structured in way which parallelizes as much of it as possible while all necessary sequential fraction of the program are kept to a minimum (Amdahl's law [2]).

.NET Framework, software framework developed by Microsoft, embraced multi-threading from it's very beginning in 2001. In the earliest versions tools available in C#, multi-paradigm programming language, were quite basic, low-level primitives that dealt with monitoring and

synchronization of threads. In 2005 F# was added to .NET family, functional language based on ML (MetaLanguage). As it will be shown, functional paradigm had and still has great influence in the development of parallel programming in .NET. Major advancements came with 2009's .NET Framework 4.0 release.TPL (Task Parallel Library), PLINQ (Parallel Language Integrated Query), concurrent collections and others were introduced in this update.

1.1. Goals and scope

The goal for this thesis is to look into various sequential software pieces and explore the potential performance gains by using using concurrency tools provided by the .NET platform. Known sequential algorithms and existing business software will be used.

Parallel versions will be implemented using diversified set of tools available on .NET platform, namely Task Parallel Library PLINQ, immutable and concurrent collections, with code pieces written both in C#.

Extensive tests on runtime performance and memory allocation will be performed. These will help to draw meaningful conclusions about parallel programming paradigm, set of precautions and guidelines which will help in future software development endevours.

1.2. Thesis structure

Chapter 2 - Concurrency and Threads in .NET - background information on concurrency in general. Broad description of terms, classes, interfaces and tools commonly used in concurrent implementions on the .NET platfom.
Chapter 3 - Test Environment - overview of testing environment. Specification of hardware and software benchmarking tools used in experiments.
Chapter 4 - Algorithms and software - explanation of algorithms and software selected as the test subjects in the thesis. Description of sequential versions of the software, explanation on how they work and implementation of their parallel counterpart.
Chapter 5 - Experiments - results of testing run time performance and memory allocation of sequential and parallel implementations of software pieces.
Chapter 6 - Discussion - in depth analysis of experiments and results, theoretical viewpoints on concurrent programming.
Chapter 7 - Conclusions - summary of results, advice for developing parallel algorithms, consideration for future research.

Chapter 2

Concurrency and Threads in .NET

This chapter serves as an overall introduction to the reader to concurrency in .NET. Common terminology used in parallel computing will be presented together with .NET specific tools and technologies.

2.1. Concurrency terminology

This section defines common terminology used in this thesis. These terms are often used in the same context, but even though they are similar, they have different meanings. It is imperative to be aware of these distinctions in order to be able to reason clearly about software and multithreading.

Sequential programming

Sequential programming is a way of writing code as step by step instructions. It is a convenient and easy to understand approach where mistakes about what to do and when do it are less common. The disadvantage of performing operations this way is that the thread must wait during parts of the process, being effectively blocked. Blocking threads and singular instructions make poor use of device resources. To reiterate, sequential programming is a set of consecutive, progressively ordered instruction execution in linear fashion (fig. 2.1).

Fig. 2.1: Sequential programming as a progressive set of instructions

In imperative and object-oriented languages there is a tendency to write sequential code, with all attention and resources focused on the task currently running. Programs are modeled and executed by performing an ordered set of statements, one after another [25].

Concurrent programming

In computer science, *concurrency* is the ability of different parts of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome [13].

Concurrency is used to achieve real multitasking in an application, by modeling the application into multiple, autonomous processes that run at the same in different threads. As an example let's examine an online video streamer. The program downloads data from the network, decompresses it and displays the video on screen. Concurrency gives the impression that all of these

parts of the program are executing simultaneously, an illusion of parallelism is created. But in a single-core environment, the execution of one thread is temporarily paused and switched to another thread, this is called context switching, as shown in fig. 2.2.

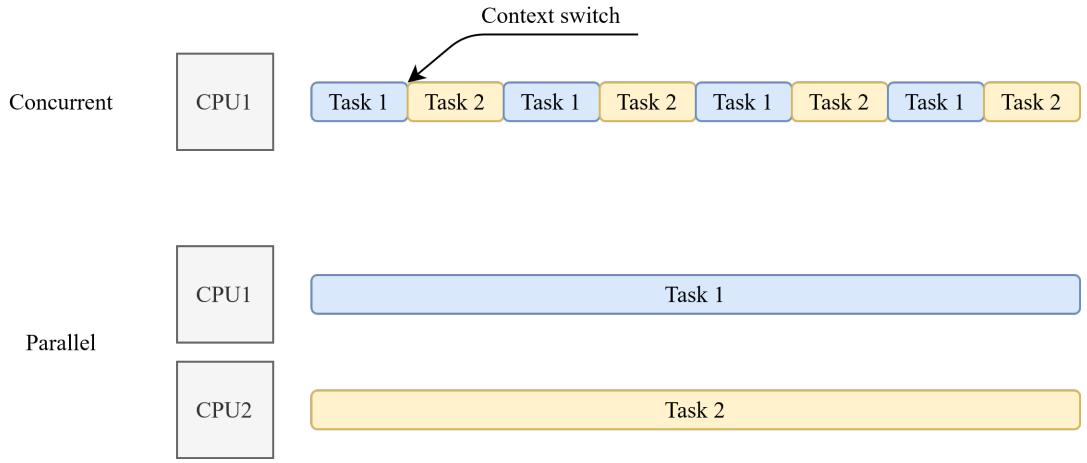

Fig. 2.2: Comparison of concurrent and parallel models

Concurrency is often confused with parallelism [20], but concurrent programs only *may* be executed in parallel by assigning each process to a separate processor or processor core, or distributing a computation across a network [3].

Parallel programming

Parallelism is the idea of processing tasks simultaneously for performance and throughput improvement of a program. Although all parallel programs are concurrent, it was demonstrated that not all concurrency is parallel. Parallel execution is constrained by the runtime environment of the program, devices need to be equiped with multicore CPUs to support it (fig. 2.2).

Timing is the qualifying factor for parallelism. It may be achieved by dividing a single task into multiple, self-contained subtasks. When they are run simultaneously on available cores and their execution overlaps in time then the program is parallel (fig. 2.3).

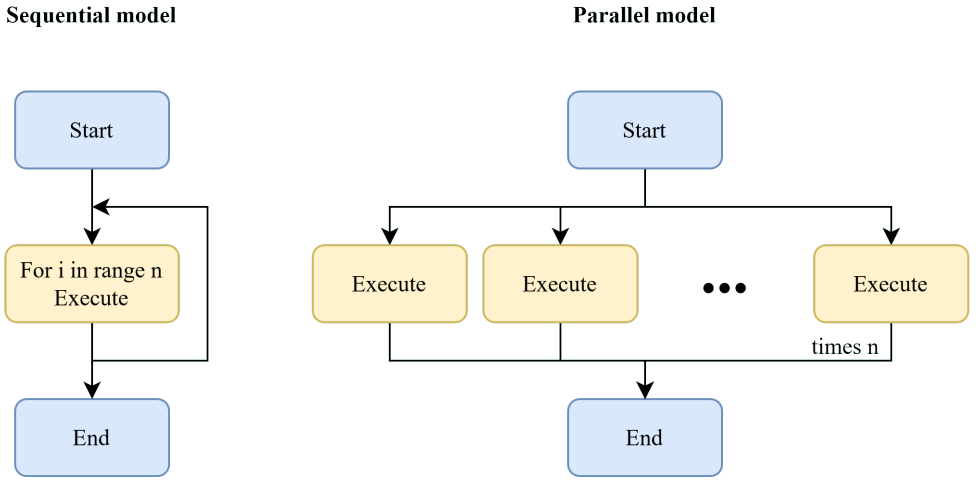

Fig. 2.3: Comparison of sequential and parallel models

CPU vs GPU computing

With the advent of general-purpose computing on graphics processing units one may ask why even bother with CPU parallel programming. Is is true that GPGPU has many advantages over CPU computing, but there two major drawbacks: branching and varying instructions. GPGPU as a technique is very good at dealing with single instructions applied to large amount of data. When branching which is switching instructions during execution happens, GPGPU tends to be outperformed by CPU computing. Thus, for general software development, full of boolean logic and if statements, CPU computing is still the standard way of proceeding [24].

Amdahl's Law

Gene Amdahl made several insights about parallel programming in his 1967 paper [2]. One of these insights would later be formulated into Amdahl's law:

> "...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."

Amdahl's law is a formula which attempts to describe the potential speedup of a parallel program with sequential parts:

$$T_N = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

where

- T_N is the theoretical speedup of execution,
- P is percentage of parallel code,
- N is available number of cores.

This equation emphasizes reducing the amount of sequential code but it makes many simplifying assumptions which make it fit only few very specific scenarios. It doesn't take into account performance saturation, thread overhead or overlapping of serial and parallel code [21]. John L. Gustafson in his article named *"Reevaluating Amdahl's Law"* [10] examined the theoretical speedup in a different and more contemporary perspective, which includes distributed systems and cloud computing. His formula is as follows:

$$T = S + (N * P) \tag{2.2}$$

where

- T is the theoretical speedup of execution,
- S are the sequential units of work,
- P are the units of work that can be executed in parallel,
- N is the number of available cores.

2.2. Concurrency and threads in .NET

This section will introduce .NET, the subject platform of this thesis. While in-depth understanding of it's vast inner workings may be helpful, it is not necessary for comprehension of this paper. Thus only a brief overview of .NET and Common Language Runtime (CLR) will be presented, reserving most space for explanation of .NET's threading and parallel execution tools. For readers desiring deeper knowledge and insights into CLR concepts I highly recommend exhaustive book *CLR via C#* by Jefrrey Richter [22].

.NET and CLR

.NET is a free, open-source development platform developed by Microsoft that supports building and running cross-platform apps of many kinds like:

- web apps, web APIs, and microservices,
- desktop apps,
- serverless functions in the cloud,
- mobile apps.

All applications have access to the same runtime, APIs and class library which are the core elements of the .NET platform.

CLR is the foundational virtual machine component of .NET which implements the Virtual Execution System (VES) defined by Microsoft's Common Language Infrastructure (CLI). The runtime is an agent that provides a managed execution environment, supporting core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness [14].

The managed execution process starts with choosing one of Commong Language Specification (CLS) compliant compilers, most popular being C#, F# and Visual Basic. The compiler translates source code into Microsoft Intermediate Language (MSIL). In this form code is CPU-independent and before it is run it must be converted to native code, usually by just-in-time (JIT) compiler. This process is performed on demand at application run time, when the contents of an assembly are loaded and executed [16]. CLR execution model is presented in fig. 2.4.

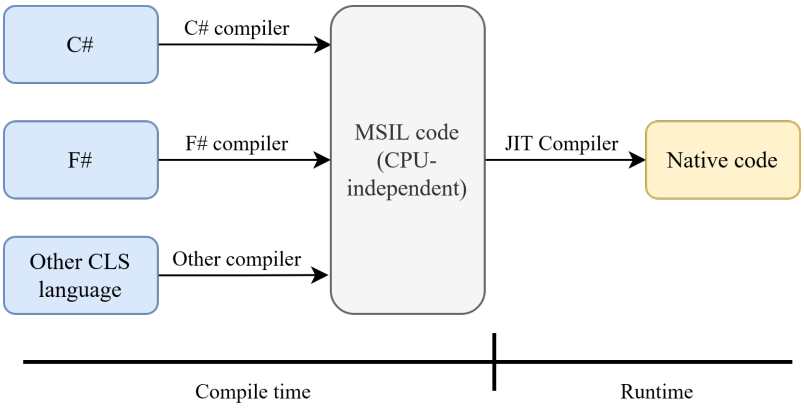

Fig. 2.4: CLR execution model

Threading in .NET

A *thread* is the fundamental unit to which an operating system (OS) allocates CPU time. Numerous threads can run inside of a *process* which is a separate, executing program inside the OS. When a thread is paused, system uses structures built into the thread to save it's context. The context contains all the information required for the thread to effortlessly resume execution. All threads spawned in the process share the same virtual address space and they can access and execute any part of the source code, including the parts currently handled by another thread.

Most .NET programs start execution with a single thread, which is usually called the *primary* thread. If the application requires parallel or asynchronous operations during runtime, additional threads, called *worker threads* are created.

Management of all threads is done through the /emphThread class, including threads created by the CLR and those created outside the runtime that enter the managed environment to execute code. The runtime monitors all the threads in its process that have ever executed code within the managed execution environment. It does not track any other threads.

A managed thread is either a *background thread* or a *foreground thread*. Foreground threads are what keeps the execution environment running. When all foreground threads in a managed process have been stopped, the system closes all background threads and shuts down the process.

Since all threads can accesses properties and methods of an single object, *synchronization* is critical during multithreaded processing. Without it objects might me mutated into invalid state or threads might interrupt or even prevent each other from completing. A class whose members are protected from such interruptions is called *thread-safe* [17].

.NET provides several synchronization strategies including but not limited to:

- Synchronized code regions with *Monitor* class or compiler support.
- Manual synchronization with synchronization primitives.
- Collection classes in the System.Collections.Concurrent namespace.

Thread overhead

Threads are a powerful tool which enable greater responsiveness and throughput of program, but as with every virtualization mechanism, threads have memory consumption and runtime execution performance overhead associated with them. That overhead will be later measured during experiments, so it's important to explore what's causing it.

Creating, destroying and having threads existing in the system has time and space overhead since all threads has one of each of the following:

- **Thread kernel object** – data structure containing thread properties like CPU registers.
- **Thread environment block** – block of memory which contains the head of the thread's exception handling chain.
- **User-mode stack** – stack used for local variables and method arguments.
- **Kernel-mode stack** – stack used for kernel-mode method arguments. Arguments are copied from user-mode stack to kernel-mode stack for security reasons.
- **DLL thread-attach and thread-detach notifications** – `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` flags are passed to unmanaged DLLs when they are loaded by the process.

A CPU can only simultaneously execute as many threads as there are physical cores. Therefore, OS has to share the actual CPU hardware among all the threads (logical CPUs) that are sitting around in the system. In .NET a thread is allowed to run for a time-slice. When it expires, context switch occurs. Every context switch requires:

- Saving the values of CPU's registers to the thread's kernel object.
- Selecting another thread to schedule. If this thread belong to another process, then new virtual address space must be loaded.
- Load values of CPU's registers from the thread's kernel object.

On top of that, if the new thread doesn't use the same code and data as the previous thread, then CPU has to access RAM memory to populate it's cache.

Context switches are needed to provide end users with responsive experience, but there is no other memory or performance benefit to them. Since they are pure overhead, they should be minimized when programming for performance [23].

ThreadPool in CLR

To improve the resource consumption situation, the CLR contains code to manage its own ThreadPool. Instead of creating and deleting threads for every request, it maintains a set of threads for reuse. Tasks from operation queue are dispatched to threads existing in the pool. When the task is completed, thread is returned to the pool instead of being deleted (fig. 2.5).

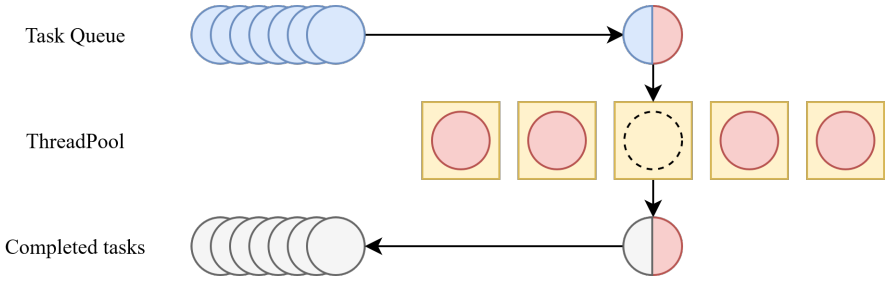

Fig. 2.5: Thread pool reuses threads to decrease resource consumption

CLR's ThreadPool saw many advancements in .NET 4.0 release, facilitating concurrent and parallel execution on multi-core architectures. Two major points were considered during development, quick work dispatch and throttling the degree of parallelism. While the former one is very important, it has relatively low impact on the topic of this paper. However, it is important to know that the CLR uses Hill Climbing algorithm and signal processing techniques to throttle the amount of threads spawned in the ThreadPool. These can, in specific and rare situations, be manually adjusted for further performance increases [9].

TPL

Task Parallel Library (TPL) is a library for .NET that makes it easy to take advantage of potential parallelism in a program. The library relies heavily on generics and delegate expressions to provide custom control structures expressing structured parallelism such as map-reduce in user programs. Types and APIs of TPL are available in the System.Threading and System.Threading.Tasks namespaces.

The TPL abstracts many low-level details like state management, cancellation, partitioning of work or scheduling threads on the ThreadPool. It also dynamically scales the degree of parallelism to optimize usage of available processors. Starting with .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code [5].

PLINQ

Language Integrated Query (or LINQ as it is commonly called) is a query execution engine which features a unified model for querying any enumerable data source in a type-safe manner. Parallel LINQ (PLINQ) is a parallel execution engine that can be used to execute LINQ queries on multi-core systems. PLINQ provides excellent support for implementing declarative data parallelism in applications using the `ParallelEnumerable` and `ParallelQuery` classes which can be found in `System.Linq` namespace.

Most important methods of `ParallelQuery` class are called `.AsParallel()` and `.AsSequential()`. These enable easy conversion of sequential queries into parallel ones and vice versa.

Partitioners

Data source needs to be partitioned into multiples sections before parallelizing an operation on it. Only then these sections can be accessed concurrently my multiple threads. PLINQ and the TPL provide default partitioners which do not use load balancing. Custom partitioners can be provided to change that. Overloads of the `Partitioner.Create` method enable developer's to specify whether it should attempt load balance data between threads. It uses chunk partitioning in which the elements are passed in small chunks to each task when they request it. This approach helps ensure that work is spread evenly throughout all working tasks until the entire query is completed [15].

Fork/Join

The Fork/Join pattern is built from two primary steps. First a single thread splits a given task into a series of subtasks designated to run in parallel. Afterwards the thread waits for the parallel operations to complete so that it can merge the results into the final output (fig. 2.6). When applied recursively, Fork/Join achieves the divide-and-conquer paradigm.

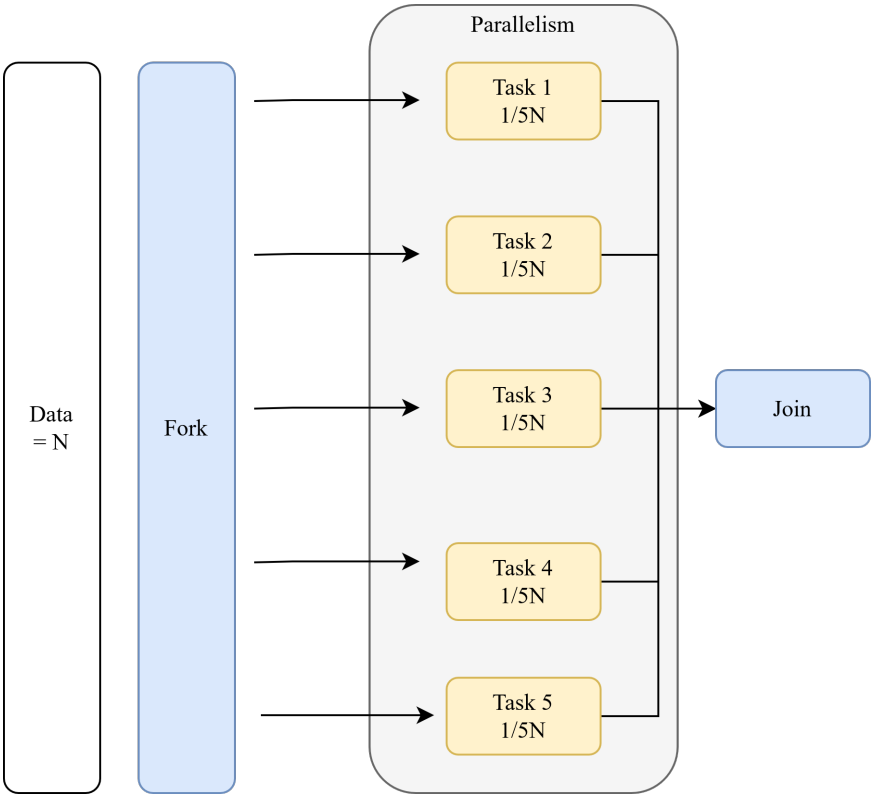

Fig. 2.6: Data parallelism Fork/Join pattern

MapReduce

MapReduce pattern, introduced in *MapReduce: Simplified Data Processing on Large Clusters* [6] paper provides solution for big data analysis. It's designed to enable scalable computations on multiple machines. It has found great use in domains requiring operations on massive datasets like machine learning, image processing or data mining. *map* and *reduce* combinators used in fuctional paradigm were the source of inspiration for the MapReduce model (Fig. 2.7). This style of programming can be used without knowledge of concurrent programming, all low-level details can be handled by the actual runtime (TPL in case of .NET). The overarching

idea of this pattern is to query streams of data by various maps and reductions. *Map* operations are the ones that preserve the number of elements but transform them into a different format. Afterwards the elements are reduced by filtering or aggregating.

In this thesis MapReduce will be implemented in a generic fashion with two main phases:

- *Map* receives the input and is responsible for mapping objects into collections of key-value pairs. The values are grouped by the key value and passed to the second phase.
- *Reduce* aggregates the results from *Map* by applying a function to values with the same key, with goal of reducing the amount of data. From there more functions can be composed into the chain.

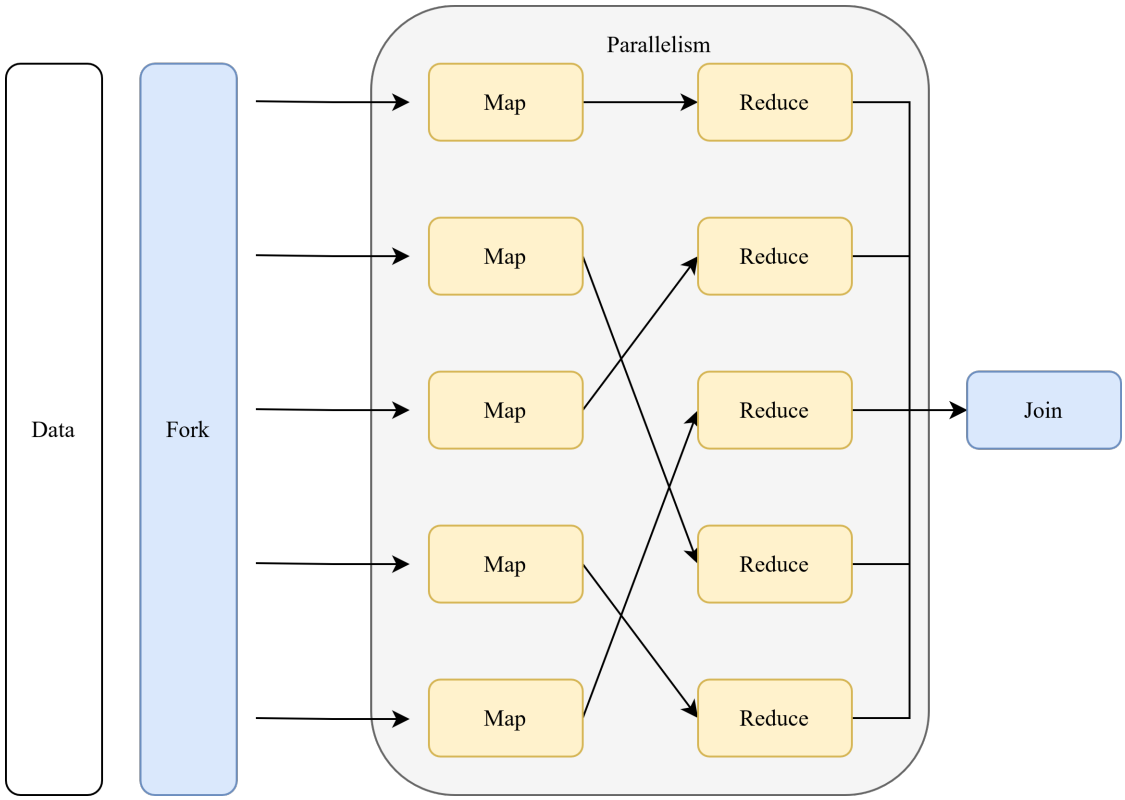

Fig. 2.7: Data parallelism MapReduce pattern

Having explained .NET parallelism tools and concepts used in the thesis, next chapter will present the test environment in which experiments will be conducted.

Chapter 3

Test Environment

This chapter contains information about the test environment used in next chapters including description of benchmarking method and specification of hardware and software used in experiments. These parameters would be required if one set out to reproduce the collected data or to compare experiments conducted on different hardware or software.

3.1. Benchmarking

Tests performed in this paper will be constructed and executed using .NET benchmarking tool, *BenchmarkDotNet*. It is an open source library which helps in transforming methods into benchmarks and producing reproducible experiments. The results are guaranteed to be reliable and precise by the usage of *perfolizer* statistical engine.

The tool is well equiped for the topic of this paper. It measures performance as mean estimated time, removes outlier data, calculates error ranges and standard deviation. Additionaly, `MemoryDiagnoser` tracks GC generations and memory allocations which is important when testing parallel programs. Listing 3.1 showcases one of the benchmarks which will be used during the experiments. These benchmarks were developed with the use of *Pro .NET benchmarking: the art of performance measurement* by Andrey Akinshin [1].

Listing 3.1: Quicksort benchmark class

```
1  namespace Net47Benchmarking
2  {
3    [SimpleJob(RuntimeMoniker.Net472)]
4    [SimpleJob(RuntimeMoniker.NetCoreApp31)]
5    [MemoryDiagnoser]
6    [RPlotExporter]
7    [CsvMeasurementsExporter]
8    public class Net47QuickSortBenchmark
9    {
10     private readonly SequentialImperativeQuickSort
           ↪ _sequentialImperativeQuickSort = new
           ↪ SequentialImperativeQuickSort();
11     private readonly ParallelImperativeQuickSort
           ↪ _parallelImperativeQuickSort = new ParallelImperativeQuickSort
           ↪ ();
12     private readonly OptimizedParallelImperativeQuickSort
           ↪ _optimizedParallelImperativeQuickSort =
13       new OptimizedParallelImperativeQuickSort((int)Math.Log(Environment
             ↪ .ProcessorCount, 2) + 4);
14
```

```
15      private readonly SequentialFunctionalQuickSort
            ↪ _sequentialFunctionalQuickSort = new
            ↪ SequentialFunctionalQuickSort();
16      private readonly ParallelFunctionalQuickSort
            ↪ _parallelFunctionalQuickSort = new ParallelFunctionalQuickSort
            ↪ ();
17      private readonly OptimizedParallelFunctionalQuickSort
            ↪ _optimizedParallelFunctionalQuickSort =
18        new OptimizedParallelFunctionalQuickSort((int)Math.Log(Environment
              ↪ .ProcessorCount, 2) + 4);
19
20      // ReSharper disable once MemberCanBePrivate.Global
21      public int[] Data;
22
23      // ReSharper disable once MemberCanBePrivate.Global
24      [Params(10000, 100000, 1000000, 10000000)] public int N;
25
26      [GlobalSetup]
27      public void Setup()
28      {
29        var random = new Random(42);
30
31        Data = Enumerable
32          .Range(0, N)
33          .Select(_ => random.Next())
34          .ToArray();
35      }
36
37
38      [Benchmark]
39      public ImmutableList<int> SequentialFunctionalQuickSort()
40         => _sequentialFunctionalQuickSort.Sort(Data).Result;
41
42      [Benchmark]
43      public ImmutableList<int> ParallelFunctionalQuickSort()
44        => _parallelFunctionalQuickSort.Sort(Data).Result;
45
46      [Benchmark]
47      public ImmutableList<int> OptimizedParallelFunctionalQuickSort()
48        => _optimizedParallelFunctionalQuickSort.Sort(Data).Result;
49
50      [Benchmark]
51      public ImmutableList<int> SequentialImperativeQuickSort()
52        => _sequentialImperativeQuickSort.Sort(Data).Result;
53
54      [Benchmark]
55      public ImmutableList<int> ParallelImperativeQuickSort()
56        => _parallelImperativeQuickSort.Sort(Data).Result;
57
58      [Benchmark]
59      public ImmutableList<int> OptimizedParallelImperativeQuickSort()
60        => _optimizedParallelImperativeQuickSort.Sort(Data).Result;
61    }
62  }
```

3.2. Hardware and software

Each machine is different in it's own way, it is imperative to use the same hardware for all experiments to accurately benchmark tested software. Thus one cannot expect to receive the same results when conducting testing as presented later on this paper but in different settings. Specification of the machine used for development and benchmarking is presented in tab. 3.1. It's a middle-range (for 2021) machine for personal use.

Tab. 3.1: Machine specification

CPU	AMD Ryzen 5 3600 3.95 GHz 6 Cores 12 Logical Processors
RAM	Patriot 16GB 3000MHz CL16
MB	Asus Prime X470 - Pro
Disk	Samsung SSD 970 EVO Plus
GPU	AMD Radeon RX 5700 XT
OS	Windows 10 x64 Pro Build 19042
BIOS	American Megatrends 5406

AMD Ryzen 5 3600 is built using Zen2 architecture, the successor of AMD's Zen and Zen+ microarchitectures. From the most notable features it enables 2 threads per physical core (SMT) and optimized processor caches: L1 cache with 32 kB per core and 8-way associative input and output, L2 with 512 kB per core and L3 cache with 16MB per core. Fig. 3.1 showcases microarchitecutre overview of Zen2 processors [11].

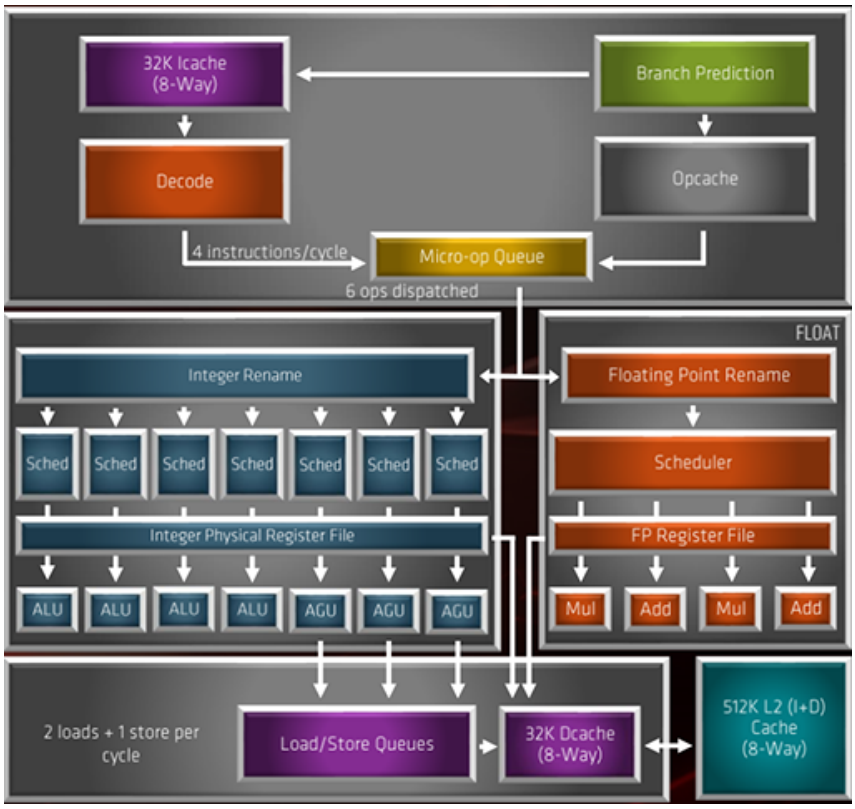

Fig. 3.1: Zen2 microarchitecture

The aforementioned caches function as memory banks between main memory and the CPU. They store operation instructions and frequently used memory locations. The caches are usualy divided into three levels which are checked for hits in a top down manner:

- **L1 cache** is the smallest, but the fastest cache. In case of Zen2 architecture it actually consists of two equal size caches, one for program data, second one for instructions.
- **L2 cache** is next in line, it serves the same function as L1 but it is slightly bigger and slightly slower.
- **L3 cache** is significantly bigger but accesing it will have greater impact on performance. In Zen2 architecture this cache is shared between cores in one chiplet.

Size and latency of the caches is presented in fig. 3.2.

```
stride  4       8       16      32      64      128     256     512
size (Kb)
1       4       4       4       4       4       4       4       4
2       4       4       4       4       4       4       4       4
4       4       4       7       4       4       4       4       4
8       4       4       4       4       4       4       4       4
16      4       4       4       4       4       4       4       4
32      4       4       4       4       4       4       4       4
64      5       6       4       4       5       12      12      12
128     4       4       4       4       5       12      12      12
256     4       4       4       4       5       12      12      12
512     4       4       4       4       6       13      13      14
1024    4       4       4       6       10      20      25      35
2048    4       4       4       4       6       21      25      37
4096    4       4       4       4       6       20      25      35
8192    4       4       8       4       6       21      25      36
16384   4       4       4       4       6       77      150     160
32768   4       4       4       4       6       21      288     505

4 cache levels detected
Level 1         size = 32Kb     latency = 4 cycles
Level 2         size = 512Kb    latency = 12 cycles
Level 3         size = 8192Kb   latency = 27 cycles
Level 4         size = 16384Kb  latency = 129 cycles
```

Fig. 3.2: Ryzen 5 3600 cache sizes and latency

This chapter described how benchmarks will be conducted and specified hardware and software used in testing. Chapter that follows will introduce algorithms and software used in testing and their multiple implementations using parallel techniques.

Chapter 4

Algorithms and software

This chapter contains information about algorithms and software which are used as subjects of this paper. Each subject will be presented with a detailed description of its behaviour alongside multiple implementations with the usage of parallel techniques like Fork/Join and MapReduce materialized using TPL and PLINQ.

4.1. Quicksort

Quicksort is an in-place sorting algorithm, developed by British computer scientist Tony Hoare in 1959. It is a divide-and-conquer algorithm. It works by selecting a 'pivot' and dividing a large array into two sub-arrays of low and high elements, depending if they are bigger or smaller than the pivot. Recursive algorithms, especially ones based on a form of divide-and-conquer , are a great candidate for parallelization and CPU-bound computations. Quicksort pseudocode (shown in listing 4.1) is based on description from *Introduction to Algorithms* [4]. The algorithm takes O(n log n) comparisons to sort n items on average.

Listing 4.1: Sequential Quicksort pseudocode

```
QUICKSORT(A, p, r)
        if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
  QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
        x = A[r]
        i = p - 1
  for j = p to r - 1
                if A[j] <= x
                        i = i + 1
                        exchange A[i] with A[j]
        exchange A[i + 1] with A[r]
        return i + 1
```

Imperative sequential implementation

This version is a "traditional" implementation of the algorithm. It is written in imperative style and quite lengthy (listing 4.2).

Listing 4.2: Imperative sequential quicksort implementation

```
private static void Sort(int[] source, int left, int right)
{
  if (left >= right) return;
  var pivot = Partition(source, left, right);
  Sort(source, left, pivot);
  Sort(source, pivot + 1, right);
}

private static int Partition(int[] source, int left, int right)
{
  var pivot = (right + left) / 2;
  var pivotInt = source[pivot];

  Swap(ref source[right - 1], ref source[pivot]);
  var store = left;
  for (var i = left; i < right - 1; ++i)
  {
    if (source[i].CompareTo(pivotInt) >= 0) continue;
    Swap(ref source[i], ref source[store]);
    ++store;
  }

  Swap(ref source[right - 1], ref source[store]);
  return store;
}

private static void Swap(ref int a, ref int b)
{
  var temp = a;
  a = b;
  b = temp;
}
```

Imperative parallel implementation

Parallel version was implemented in a manner in which somebody who is not well versed in .NET's parallelism would implement it. Many developers know async-await state machine because is often used in projects with asynchronous operations . On top of that no precautions for thread oversaturation were taken (listing 4.3).

Listing 4.3: Imperative parallel quicksort

```
private static async Task Sort(int[] source, int left, int right)
{
  if (left >= right) return;

  var pivot = Partition(source, left, right);

  await Task.Run(() => Sort(source, left, pivot));
  await Task.Run(() => Sort(source, pivot + 1, right));
}
```

Imperative optimized implementation

Optimized version uses depth control and removes the use of async-await state machine (listing 4.4).

Listing 4.4: Imperative optimized quicksort

```
private static void Sort(int[] source, int left, int right, int depth)
{
  if (left >= right) return;

  var pivot = Partition(source, left, right);
  if (depth > 0)
  {
    Parallel.Invoke(
      () => Sort(source, left, pivot, depth--),
      () => Sort(source, pivot + 1, right, depth--));
  }

  else
  {
    Sort(source, left, pivot, depth);
    Sort(source, pivot + 1, right, depth);
  }
}
```

Functional sequential implementation

This version of the algorithm was implemented using LINQ in a functional, declarative manner. It is conspicuous that this version is more concise and easier to read (listing 4.5).

Listing 4.5: Functional sequential quicksort implementation

```
private static ImmutableList<int> Sort(ImmutableList<int> source)
{
  if (!source.Any()) return source;

  var first = source.First();
  var (left, right) = source
    .Skip(1)
    .Partition(x => x <= first);
  var leftSorted = Sort(left);
  var rightSorted = Sort(right);

  return leftSorted
    .Add(first)
    .AddRange(rightSorted);
}
```

Functional parallel implementation

Parallel implementation was achieved in the same way, using `Task.Run` and async-await state machine (listing 4.6).

Listing 4.6: Functional parallel quicksort

```
private async Task<ImmutableList<int>> Sort(ImmutableList<int> source)
{
  if (source.None()) return source;

  var first = source.First();

  var (left, right) = source
    .Skip(1)
    .Partition(x => x <= first);
  var leftSorted = await Task.Run(() => Sort(left));
```

```
  var rightSorted = await Task.Run(() => Sort(right));

  return leftSorted
    .Add(first)
    .AddRange(rightSorted);
}
```

Functional optimized implementation

Optimized version uses depth control and removes the use of async-await state machine (listing 4.7).

Listing 4.7: Functional optimized quicksort

```
private static Task<ImmutableList<int>> Sort(ImmutableList<int> source, int
    ↪  depth)
{
  if (source.None()) return Task.FromResult(source);

  var first = source.First();
  var (left, right) = source
    .Skip(1)
    .Partition(x => x <= first);
  var leftSorted = ParallelizeUnlessDepthIsReached(x => Sort(left, x),
     ↪ depth);
  var rightSorted = ParallelizeUnlessDepthIsReached(x => Sort(right, x),
     ↪ depth);

  return Task.FromResult(leftSorted.Result
    .Add(first)
    .AddRange(rightSorted.Result));
}

private static Task<T> ParallelizeUnlessDepthIsReached<T>(Func<int, Task<T>
    ↪ > func, int depth)
{
  return depth > 0 ? Task.Run(() => func(depth - 1)) : func(0);
}
```

4.2. K-means clustering

K-means, otherwise commonly know as Lloyd's algorithm, is an unsupervised machine learning algorithm which is used to process a data set into clusters. They represent a geometric shape with a mass center, a centroid. Each of the clusters has its own centroid which is the sum of points divided by number of total points. Finding a solution for Lloyd's algorithm is a NP-hard problem even for two clusters. It's complexity can be expressed as O(nkdi) where:

- n is the number of vectors with d-dimensions
- k is the number of cluster
- i is the number of iterations required for the convergence

The algorithm is an iterative process which repeats until it reaches a convergence point or exceeds the limit (sometimes it does not converge). Each of the iterations updates the centroids to produce better clusters. The iteration involves these steps:

1. Sum up points in each cluster.
2. Divide each sum by the number of points in respective cluster.
3. Reassign all points to a closest centroid, distance is calculated through Euclidean distance function.
4. Repeat until locations stabilize.

In the following subsections 3 implementations of centroid recalculation algorithm will be presented. Each of the them is injected into k-means algorithm (listing 4.8) during performance benchmarking.

Listing 4.8: K-means centroid computation

```
//For clarity
using DataSet = System.Collections.Immutable.ImmutableList<System.
    ↪ Collections.Immutable.ImmutableList<double>>;

public static DataSet ComputeCentroids(Func<DataSet, DataSet>
    ↪ updateCentroids, DataSet centroids,
  int rowLength)
{
  for (var i = 0; i <= 1000; i++)
  {
    var (newCentroids, error) = GetNewCentroidsAndError(updateCentroids,
        ↪ rowLength, centroids);

    if (error < 1e-9)
      return newCentroids;

    centroids = newCentroids;
  }

  return centroids;
}

private static (DataSet, double) GetNewCentroidsAndError(Func<DataSet,
    ↪ DataSet> updateCentroids, int rowLength, DataSet centroids)
{
  var newCentroids = updateCentroids(centroids).SortCentroids(rowLength);
  var error = double.MaxValue;

  if (centroids.Count == newCentroids.Count)
    error = centroids.Select((t, j) => DistanceTo(t, newCentroids[j])).Sum(
        ↪ );
```

```
  return (newCentroids, error);
}


private static DataSet SortCentroids(this DataSet result, int RowLength)
  => result.Sort((a, b) =>
  {
    for (var i = 0; i < RowLength; i++)
      if (a[i] != b[i])
        return a[i].CompareTo(b[i]);
    return 0;
  });


public static ImmutableList<double> GetNearestCentroid(DataSet centroids,
    ↪ ImmutableList<double> center)
  => centroids.Aggregate((centroid1, centroid2) =>
    centroid2.DistanceTo(center) < centroid1.DistanceTo(center)
      ? centroid2
      : centroid1);

private static double DistanceTo(this ImmutableList<double> @from,
    ↪ ImmutableList<double> to)
{
  var results = 0.0;
  for (var i = 0; i < @from.Count; i++)
    results += Math.Pow(@from[i] - to[i], 2.0);
  return results;

}
```

Sequential implementation

`UpdateCentroids` evaluates each cluster and reassigns centroids to newly calculated centers in two steps. In the first one `GroupBy` function is used to aggregate the points using the `GetNearestCentroid` function as a key. In the next one, each point grouping is used to calculate new centers for each given point. This step is achieved via local accumulator in `Aggregate` function and collection combining in `Zip` function which threads centroids and accumulator sequences. Finally, each center is determined through dividing with number of points in the cluster (listing 4.9).

Listing 4.9: Sequential k-means algorithm

```
private Func<DataSet, DataSet> UpdateCentroids()
  => centroids =>
    Source
      .GroupBy(row => KMeans.GetNearestCentroid(centroids, row))
      .Select(CalculateCenter)
      .ToImmutableList();

private ImmutableList<double> CalculateCenter(IGrouping<ImmutableList<
    ↪ double>, ImmutableList<double>> points)
  => points
    .Aggregate(new double[RowLength], (acc, item) => acc.Zip(item, (a, b) =
      ↪ > a + b).ToArray())
    .Select(items => items / points.Count())
    .ToImmutableList();
```

Parallel implementation

This implementation uses PLINQ (section 2.2) which easily transforms sequential LINQ query into a parallel one with 2 lines of code. This was enabled by using `Aggregate`, LINQ's equivalent of functional sequence transforming function: `Seq.Fold`. Force parallelism execution mode is used to make sure that PLINQ internal mechanism won't transform the query into a sequential one.

Listing 4.10: Parallel k-means algorithm

```
private Func<DataSet, DataSet> UpdateCentroids()
  => centroids =>
    Source
      .AsParallel()
      .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
      .GroupBy(u => KMeans.GetNearestCentroid(centroids, u))
      .Select(CalculateCenter)
      .ToImmutableList();

private ImmutableList<double> CalculateCenter(IGrouping<ImmutableList<
    ↪ double>, ImmutableList<double>> points)
  => points
    .Aggregate(new double[RowLength], (acc, item) => acc.Zip(item, (a, b) =
       ↪ > a + b).ToArray())
    .Select(items => items / points.Count())
    .ToImmutableList();
```

Parallel implementation with partitioner

To further boost the parallel implementation, data partitioner was used to load balance the data between threads (section 2.2) (listing 4.11).

Listing 4.11: Parallel k-means algorithm with partitioner

```
private Func<DataSet, DataSet> UpdateCentroids()
  => centroids
    => Partitioner
    .Create(Source, loadalBalance: true)
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .GroupBy(u => KMeans.GetNearestCentroid(centroids, u))
    .Select(CalculateCenter)
    .ToImmutableList();

private ImmutableList<double> CalculateCenter(IGrouping<ImmutableList<
    ↪ double>, ImmutableList<double>> points)
  => points
    .Aggregate(new double[RowLength], (acc, item) => acc.Zip(item, (a, b) =
       ↪ > a + b).ToArray())
    .Select(items => items / points.Count())
    .ToImmutableList();
```

4.3. Mandelbrot

Mandelbrot set images are made by sampling complex numbers and testing for each sample point whether the orbit of the critical point $z = 0$ under iteration of the quadratic map remains bounded (eq. 4.1) [26]. Images of the Mandelbrot set display an elaborate boundary that reveals complex structure arising from the application of simple rules. It's one of the best-known examples of mathematical visualization. It was named in the tribute of a pioneer of fractal geometry, Benoit Mandelbrot, by Adrien Douady [7]. It's complexity can be expressed as O(rci) where:

- r is the number of rows in the map
- c is the number of columns in the map
- i is the number of maximum iterations

$$z_n + 1 = z_n^2 + c \tag{4.1}$$

In the following subsections 3 implementations of Mandelbrot set drawing algorithms will be presented. Each of the them is injected into bitmap generating code (listing 4.12) during performance benchmarking. Example visualization made during the experiments is presented in fig. 4.1.

Listing 4.12: Mandelbrot bitmap generation

```
public static Bitmap DrawMandelbrotBitmap(Action<BitmapData, byte[]>
    ↪ mandelbrotDrawingStrategy, int rows, int columns)
{
  var bitmap = new Bitmap(rows, columns, PixelFormat.Format24bppRgb);
  var bitmapData = LockBitmap(bitmap);
  var allPixels = new byte[bitmapData.Stride * bitmap.Height];
  var firstPixel = bitmapData.Scan0;
  Marshal.Copy(firstPixel, allPixels, 0, allPixels.Length);

  mandelbrotDrawingStrategy(bitmapData, allPixels);

  Marshal.Copy(allPixels, 0, firstPixel, allPixels.Length);
  UnlockBitmap(bitmap, bitmapData);

  return (Bitmap)bitmap.Clone();
}

private static BitmapData LockBitmap(Bitmap bitmap)
  => bitmap.LockBits(new Rectangle(0,
      0,
      bitmap.Width,
      bitmap.Height),
    ImageLockMode.ReadWrite,
    PixelFormat.Format24bppRgb);

private static void UnlockBitmap(Bitmap bitmap, BitmapData bitmapData)
  => bitmap.UnlockBits(bitmapData);
```

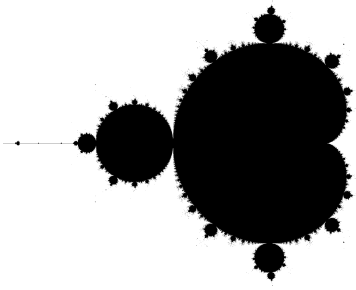

Fig. 4.1: Mandelbrot set visualization

Sequential implementation

This implementation consists of two nested loops. The outer one iterates over the columns of the bitmap while the inner one iterates over its rows. Pixel coordinates are translated into real and imaginary parts of a complex number by using the `ComputeColumn` and `ComputeRow` functions. Afterwards `BelongsToMandelbrot` function checks if the number is part of the Mandelbrot set (listing 4.13).

Listing 4.13: Sequential Mandelbrot algorithm

```
private Action<BitmapData, byte[]> DrawingStrategy()
  => (bitmapData, pixels) =>
  {
    for (var column = 0; column < Columns; column++)
    {
      for (var row = 0; row < Rows; row++)
      {
        var x = Center.ComputeRow(row, Width, Columns);
        var y = Center.ComputeColumn(column, Height, Rows);
        var c = new ComplexNumber(x, y);
        var color = BelongsToMandelbrot(c, 100) ? Color.Black : Color.White
            ↪ ;
        var offset = (column * bitmapData.Stride) + (3 * row);
        pixels.WriteColors(offset, color);
      }
    }
  };

private static bool BelongsToMandelbrot(ComplexNumber number, int
    ↪ iterations)
{
  var zNumber = new ComplexNumber(0.0f, 0.0f);
  var accumulator = 0;
  while (accumulator < iterations && zNumber.Magnitude() < 2.0)
  {
    zNumber = zNumber.MultiplyWith(zNumber).AddTo(number);
    ++accumulator;
  }

  return accumulator == iterations;
}
```

Parallel implementation

Parallel version was implemented using Fork/Join (section 2.2) pattern. TPL (section 2.2) enables us to easily apply this pattern with multiple parallelization constructs, in this case the `Parallel.For` replacement for sequential loops (listing 4.14).

Listing 4.14: Parallel Mandelbrot algorithm

```
private Action<BitmapData, byte[]> DrawingStrategy()
  => (bitmapData, pixels) =>
    Parallel.For(0, Columns - 1, column =>
    {
      for (var row = 0; row < Rows; row++)
      {
        var x = Center.ComputeRow(row, Width, Columns);
        var y = Center.ComputeColumn(column, Height, Rows);
        var c = new ComplexNumber(x, y);
        var color = BelongsToMandelbrot(c, 100) ? Color.Black : Color.White
            ↪ ;
        var offset = (column * bitmapData.Stride) + (3 * row);
        pixels.WriteColors(offset, color);
      }
    });

private static bool BelongsToMandelbrot(ComplexNumber number, int
    ↪ iterations)
{
  var zNumber = new ComplexNumber(0.0f, 0.0f);
  var accumulator = 0;
  while (accumulator < iterations && zNumber.Magnitude() < 2.0)
  {
    zNumber = zNumber.MultiplyWith(zNumber).AddTo(number);
    ++accumulator;
  }
  return accumulator == iterations;
}
```

Double parallel implementation

This version replaced both sequential loops with TPL's `Parallel.For` construct. It will be useful to examine how oversaturation impacts the algorithm performance (listing 4.15)

Listing 4.15: Double parallel Mandelbrot algorithm

```
private Action<BitmapData, byte[]> DrawingStrategy()
  => (bitmapData, pixels) =>
    Parallel.For(0, Columns - 1, column =>
    {
      Parallel.For(0, Rows - 1, row =>
      {
        var x = Center.ComputeRow(row, Width, Columns);
        var y = Center.ComputeColumn(column, Height, Rows);
        var c = new ComplexNumber(x, y);
        var color = BelongsToMandelbrot(c, 100) ? Color.Black : Color.White
            ↪ ;
        var offset = (column * bitmapData.Stride) + (3 * row);
        pixels.WriteColors(offset, color);
      });
    });
```

```
private static bool BelongsToMandelbrot(ComplexNumber number, int
    ↪ iterations)
{
  var zNumber = new ComplexNumber(0.0f, 0.0f);
  var accumulator = 0;
  while (accumulator < iterations && zNumber.Magnitude() < 2.0)
  {
    zNumber = zNumber.MultiplyWith(zNumber).AddTo(number);
    ++accumulator;
  }
  return accumulator == iterations;
}
```

Parallel implementation using value types

In some cases .NET's Garbage Collector can be the bottleneck of the application. Reference type objects are allocated on the heap and are very cheap to use as a function argument since only the pointer is copied. These objects though have a memory overhead and may heavily tax the GC which will stop the execution of the program until cleanup is done. Contrary to that, value type objects are allocated on the stack and will never cause GC to pause the program. This version uses `ComplexNumberStruct` objects which are identical to previous versions except they are value type instead of reference type (listing 4.16).

Listing 4.16: Parallel Mandelbrot algorithm using value types

```
private Action<BitmapData, byte[]> DrawingStrategy()
  => (bitmapData, pixels) =>
    Parallel.For(0, Columns - 1, column =>
    {
      for (var row = 0; row < Rows; row++)
      {
        var x = Center.ComputeRow(row, Width, Columns);
        var y = Center.ComputeColumn(column, Height, Rows);
        var c = new ComplexNumberStruct(x, y);
        var color = BelongsToMandelbrot(c, 100) ? Color.Black : Color.White
            ↪ ;
        var offset = (column * bitmapData.Stride) + (3 * row);
        pixels.WriteColors(offset, color);
      }
    });


private static bool BelongsToMandelbrot(ComplexNumberStruct number, int
    ↪ iterations)
{
  var zNumber = new ComplexNumberStruct(0.0f, 0.0f);
  var accumulator = 0;
  while (accumulator < iterations && zNumber.Magnitude() < 2.0)
  {
    zNumber = zNumber.MultiplyWith(zNumber).AddTo(number);
    ++accumulator;
  }
  return accumulator == iterations;
}
```

4.4. NuGet package ranking

NuGet is Microsoft-supported mechanism for sharing code, a package manager which defines how packages for .NET are created, hosted, and consumed, and provides the tools for each of those roles. It's a central package repository used by almost all .NET developers. Following software is used to download, analyze and rank NuGet packages by summing up the score of its dependencies and adding that to the score of the package itself. The software was implemented using MapReduce pattern (section 2.2). Software has estimated complexity of O($i * nlog_n$), i being the number of operations and n being the amount of packages.

Methods used in package ranking are defined in listing 4.17. These functions will be later plugged into multiple, generic implementations of MapReduce pattern. `Map` function is responsible for extracting key-value tuples from NuGet packages. Key represents the package name and float it's score, which will be later computed into the overall score of the package in the `Reduce` function.

Listing 4.17: Nuget package ranking functions

```
private static IEnumerable<(string, float score)> Map(NuGet.
    ↪ NuGetPackageCache package,
  ImmutableList<(string, float)> packages)
  => package.Dependencies.Select(x => (Domain.PackageName(x.Item1.Item1).
      ↪ Item1, GetScore(package, packages)));

private static float GetRanking(string packageName, ImmutableList<(string,
    ↪ float)> packages)
  => packages.Any(x => x.First() == packageName)
    ? packages.First(x => x.First() == packageName).Second()
    : 1.0f;

private static float GetScore(NuGet.NuGetPackageCache package,
    ↪ ImmutableList<(string, float)> packages)
  => GetRanking(package.PackageName, packages) / package.Dependencies.
      ↪ Length;

private static (string packageName, float) Reduce(string packageName,
    ↪ IEnumerable<float> values)
  => (packageName, values.Sum());
```

Sequential implementation

MapReduce pattern is described in detail in section 2.2. In simple terms `Map` function is responsible for mapping objects into collections of key-value pairs. In this case it will transform `NuGetPackage` objects into collection of packageName-score tuples. `Reduce` function receives collection of values grouped by pairs, in this case groups of scores grouped by the name of the package. The pattern was implemented using LINQ, with eager materialization in both `Map` and `Reduce` to avoid multiple evaluation of the same sequence (listing 4.18).

Listing 4.18: Sequential MapReduce implementation

```
public static IEnumerable<TReduced> MapReduce<TIn, TOutKey, TOutValue,
    ↪ TReduced>(
  this IEnumerable<TIn> source,
  Func<TIn, IEnumerable<(TOutKey, TOutValue)>> map,
  Func<TOutKey, IEnumerable<TOutValue>, TReduced> reduce)
  => source
    .Map(map)
    .Reduce(reduce);
```

```
private static IEnumerable<IGrouping<TOutKey, (TOutKey, TOutValue)>> Map<
    ↪ TIn, TOutKey, TOutValue>(
  this IEnumerable<TIn> source,
  Func<TIn, IEnumerable<(TOutKey, TOutValue)>> map)
  => source
    .SelectMany(map)
    .GroupBy(Tuple.First)
    .ToImmutableList();

private static IEnumerable<TReduced> Reduce<TKey, TValue, TReduced>
(this IEnumerable<IGrouping<TKey, (TKey, TValue)>> source,
  Func<TKey, IEnumerable<TValue>, TReduced> reduce)
  => source
    .Select(x => reduce(x.Key, x.Select(Tuple.Second)))
    .ToImmutableList();
```

Parallel implementation

Parallel implementation adds additional paramater which controls the degree of parallelism to avoid oversaturation. This version was also implemented using PLINQ in a very intuitive and readable way, thanks to the use of MapReduce pattern (listing 4.19).

Listing 4.19: Parallel MapReduce implementation

```
private static ParallelQuery<TSource> ToParallel<TSource>(this IEnumerable<
    ↪ TSource> source, int degreeOfParallelism)
  => source
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .WithDegreeOfParallelism(degreeOfParallelism);

private static IEnumerable<TReduced> MapReduce<TIn, TOutKey, TOutValue,
    ↪ TReduced>(
  this IEnumerable<TIn> source,
  Func<TIn, IEnumerable<(TOutKey, TOutValue)>> map,
  Func<TOutKey, IEnumerable<TOutValue>, TReduced> reduce, int mDOR, int
      ↪ rDOR)
  => source
    .ToParallel(mDOR)
    .Map(map)
    .ToParallel(rDOR)
    .Reduce(reduce)
    .ToImmutableList();
```

Parallel implementation with partitioner

In the last version load balancing partitioner was used to try to evenly spread the work between the threads (listing 4.20).

Listing 4.20: Parallel MapReduce implementation with partitioner

```
private static ParallelQuery<TSource> ToParallel<TSource>(this
    ↪ OrderablePartitioner<TSource> source, int degreeOfParallelism)
  => source
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .WithDegreeOfParallelism(degreeOfParallelism);

private static IEnumerable<TReduced> PartitionerMapReduce<TIn, TOutKey,
    ↪ TOutValue, TReduced>(
```

```
  this IList<TIn> source,
  Func<TIn, IEnumerable<(TOutKey, TOutValue)>> map,
  Func<TOutKey, IEnumerable<TOutValue>, TReduced> reduce, int mDOR, int
      ↪ rDOR)
  => Partitioner
    .Create(source, true)
    .ToParallel(mDOR)
    .Map(map)
    .ToParallel(rDOR)
    .Reduce(reduce)
    .ToImmutableList();
```

With the last piece of software described, this chapter comes to an end. Following chapter will put these algorithms to a test using benchmarks described in chapter 3.

Chapter 5

Experiments

Chapter 3 described how the experiments are to be executed and Chapter 4 described the subjects of the tests, algorithms and software. This chapter contains the results of these experiments. Each of the sections will contain graphs and tables showcasing the results, alongside a brief description of the numbers produced.

5.1. Quicksort

Quicksort experiments were conducted with 3 sizes of arrays, 6 implementations and 2 .NET environment version (tab. 5.1).

.NET version had significant impact on functional implementation of the algortihm. Functional version showed average of $\approx 27\%$ improvement (fig. 5.1) on .NET Core 3.1, while the imperative smaller but still significant $\approx 9\%$ (fig. 5.2).

Async-await version performed worse in all scenarios. Functional version was slower on average by $\approx 54\%$, while the imperative by dramatic $\approx 96\%$. These relations where consistent across all array sizes and version, slightly favouring .NET Core 3.1 (tab. 5.2).

The optimized parallel versions with depth control performed significantly better. LINQ version gained average reduction of $\approx 45\%$ while the imperative one $\approx 21\%$. Here difference between .NET version was more pronounced. A difference of 4 percentage points for the functional one (43% -> 47%) and 7 percentage points for the imperative one (18% -> 25%) (tab. 5.2).

Imperative version of quick sort algorithm performer better than the functional one. On average it was faster by $\approx 95\%$ on .NET Framework 4.7 and $\approx 91\%$ on .NET Core 3.1 (fig. 5.3).

Memory consumption increased in both versions. Functional parallel version consumed $\approx 17\%$ more memory by average. Imperative parallel version increased it $\approx 73\%$. This was consistent among the .NET versions but consumption increased less the bigger the sorted array was (tab. 5.2).

Tab. 5.1: Quicksort benchmarking parameters

Name	**Value**
Small array	10000
Medium array	100000
Large array	1000000
Old version	.NET Framework 4.7
Modern version	.NET Core 3.1

Tab. 5.2: Quicksort benchmarking results

Version	N	Mean [ms]	Error [ms]	StdDev [ms]	Gen 0	Gen 1	Gen 2	Alloc. [MB]
.NET Framework 4.7								
SequentialFunctional	10000	31.39	0.13	0.11	2906	594	250	17
ParallelFunctional	10000	85.54	0.52	0.43	4500	500	167	26
OptimizedParallelFunctional	10000	16.44	0.33	0.29	4000	1000	0	22
SequentialImperative	10000	0.83	0.00	0.00	83	41	0	0.509
ParallelImperative	10000	51.13	0.97	2.21	2364	364	91	8
OptimizedParallelImperative	10000	0.69	0.01	0.01	410	203	0	3
SequentialFunctional	100000	438.21	6.95	5.81	35000	9000	2000	201
ParallelFunctional	100000	950.50	11.81	11.05	51000	11000	2000	287
OptimizedParallelFunctional	100000	265.64	6.11	15.33	40000	9000	1000	242
SequentialImperative	100000	14.39	0.14	0.13	844	375	141	5
ParallelImperative	100000	488.38	4.33	4.05	49000	0	0	84
OptimizedParallelImperative	100000	11.20	0.46	0.56	6063	781	250	19
SequentialFunctional	1000000	5,548.45	69.15	57.74	368000	77000	7000	2,268
ParallelFunctional	1000000	10,845.01	164.33	145.68	532000	92000	4000	3,136
OptimizedParallelFunctional	1000000	3,005.16	78.88	168.10	428000	98000	5000	2,665
SequentialImperative	1000000	215.68	2.12	1.98	8667	3667	1000	51
ParallelImperative	1000000	4,970.80	41.24	36.56	496000	3000	0	843
OptimizedParallelImperative	1000000	184.68	4.67	12.05	67333	4667	1000	149
.NET Core 3.1								
SequentialFunctional	10000	23.07	0.43	0.41	1969	656	219	16
ParallelFunctional	10000	62.52	1.21	1.73	2889	667	222	23
OptimizedParallelFunctional	10000	12.21	0.40	1.07	2000	1000	0	20
SequentialImperative	10000	0.83	0.01	0.01	62	30	0	0.508
ParallelImperative	10000	33.91	0.40	0.38	813	375	0	7
OptimizedParallelImperative	10000	1.08	0.02	0.02	305	152	0	2
SequentialFunctional	100000	321.69	6.40	10.52	23000	6000	1000	190
ParallelFunctional	100000	671.42	6.47	6.05	32000	6000	1000	258
OptimizedParallelFunctional	100000	157.80	5.13	13.88	29000	9000	2000	226
SequentialImperative	100000	12.70	0.19	0.18	578	297	125	5
ParallelImperative	100000	331.97	2.57	2.14	8000	0	0	67
OptimizedParallelImperative	100000	10.02	0.31	0.29	2469	734	234	19
SequentialFunctional	1000000	4,397.27	63.14	59.06	259000	68000	5000	2,160
ParallelFunctional	1000000	7,564.57	85.35	79.84	343000	77000	4000	2,840
OptimizedParallelFunctional	1000000	2,555.12	67.87	72.62	303000	85000	3000	2,520
SequentialImperative	1000000	206.51	1.29	1.14	6667	3667	1000	51
ParallelImperative	1000000	3,380.04	38.11	35.65	82000	4000	0	668
OptimizedParallelImperative	1000000	158.91	2.51	2.1	18250	3750	750	148

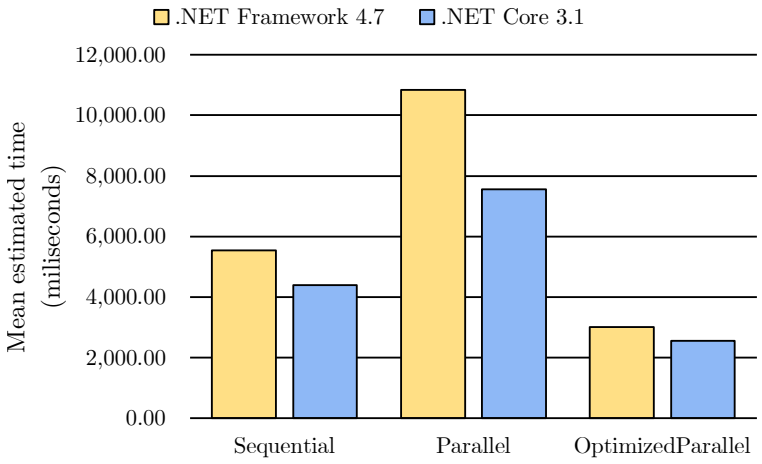

Fig. 5.1: Difference between .NET versions in functional implementation - large array

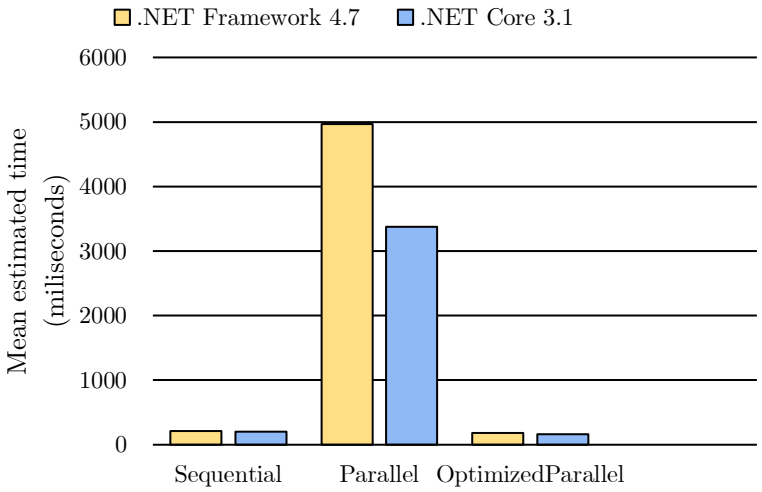

Fig. 5.2: Difference between .NET versions in imperative implementation - large array

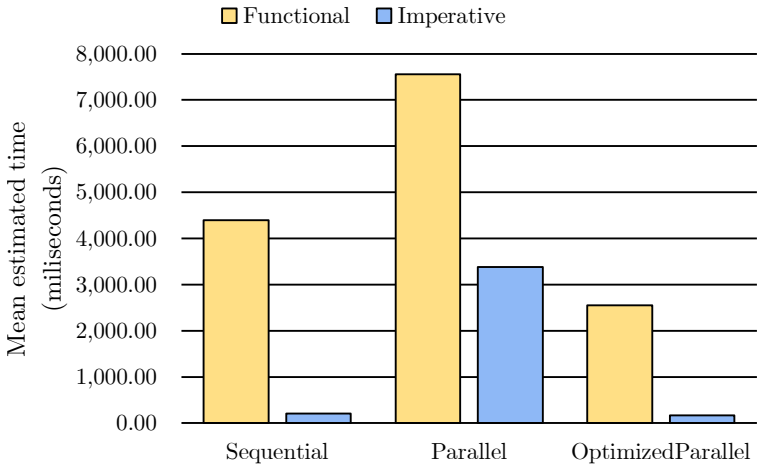

Fig. 5.3: Performance across all version with large array on .NET Core

5.2. K-means clustering

K-means clustering algorithm experiments were conducted using *White wine quality* dataset [19]. Sequential version of the algorithm had MET $\approx 2.45s$ while the parallel one had MET $\approx 0.5s$, for a $\approx 80\%$ reduction at the cost of $\approx 18\%$ increase of memory consumption. Using partitioner further dropped down the MET to $\approx 0.43s$ which is a $\approx 82\%$ reduction from the sequential version and $\approx 15\%$ from the parallel one with equal memory consumption increase (fig. 5.4, fig. 5.5, tab. 5.3).

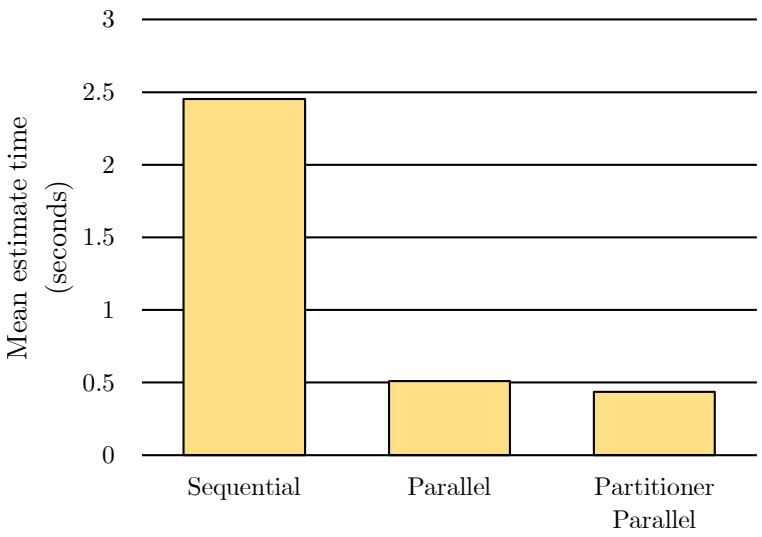

Fig. 5.4: K-means clustering performance

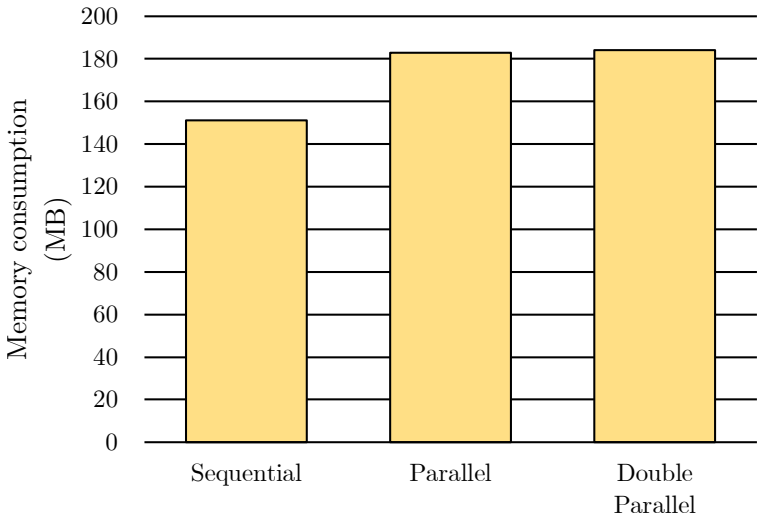

Fig. 5.5: K-means clustering memory consumption

5.3. Mandelbrot

Mandelbrot algorithm experiments were conducted using parameters presented in tab. 5.4. They describe amount of pixels generated and bitmap dimensions. Sequential version of the algorithm had a MET $\approx 5.6s$. Parallel counterpart clocked at MET $\approx 2.1s$, for a $\approx 62\%$ reduction. Parallel version with two *Parallel.For* loops had a MET $\approx 2.7s$ which is $\approx 21\%$ slower than the less

Tab. 5.3: K-means clustering benchmarking results

Version	Mean [ms]	Error [ms]	StdDev [ms]	Gen 0	Gen 1	Gen 2	Alloc. [MB]
Sequential	2,453.7	25.01	20.89	18000	2000	0	151
Parallel	509.6	8.28	8.50	23000	9000	0	183
PartitionerParallel	434.5	2.69	2.52	23000	8000	0	184

parallelized version using only one parallel loop. The best performing version using value type complex number had MET $\approx 1.9s$, for a $\approx 66\%$ reduction compared to sequential version and $\approx 11\%$ reduction to the parallel version. Additionaly, the value type version had 0 GC clean ups and virtually no (compared to others) memory consumption (fig. 5.6 , tab. 5.5).

Tab. 5.4: Mandelbrot benchmarking experiment parameters

Name	Value
Width	2.5
Height	2.5
Column amount	4000
Row amount	4000

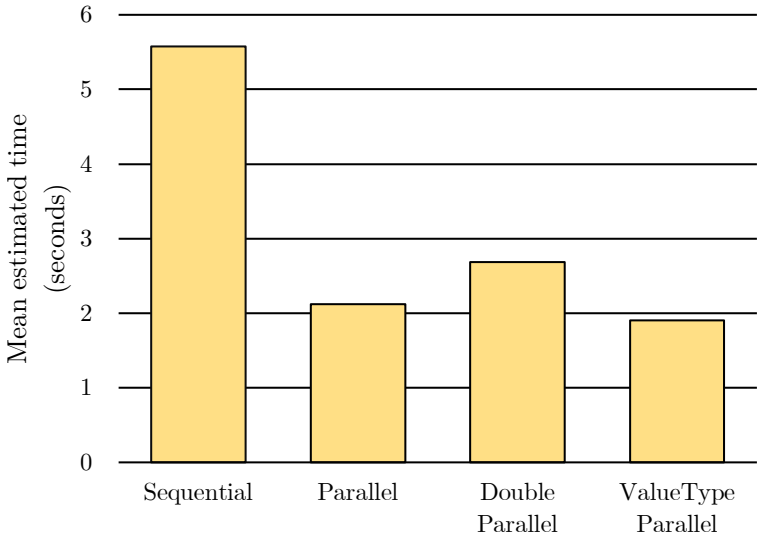

Fig. 5.6: Mandelbrot algorithm performance

Tab. 5.5: Mandelbrot benchmarking results

Version	Mean [s]	Error [s]	StdDev [s]	Gen 0	Gen 1	Gen 2	Alloc. [MB]
Sequential	5.576	0.0564	0.0528	2483000	1000	1000	19851
Parallel	2.120	0.0458	0.1350	2485000	6000	1000	19843
DoubleParallel	2.684	0.0553	0.1632	2487000	39000	2000	19852
ValueTypeParallel	1.903	0.0137	0.0128	0	0	0	46

5.4. NuGet package ranking

NuGet package ranking software was tested using 3 datasets varying in size, other parameters were constast across tests (tab. 5.6).

When testing with the small dataset, sequential version had MET $\approx 1.7s$, while parallel one had MET $\approx 0.35s$ for a $\approx 79\%$ reduction. In this case, usage of partitioner worsened the performance with MET $\approx 0.38s$, which is $\approx 6\%$ slower (fig. 5.7).

During tests with the medium dataset, sequential version had MET $\approx 17.5s$. Parallel version clocket at $\approx 4.6s$, for a $\approx 76\%$ reduction. This time partitioner version improved the performance at MET $\approx 3s$, which is $\approx 35\%$ faster than the bare parallel counterpart (fig. 5.8).

Finally sequential version on the large dataset had MET $\approx 56.9s$ while the parallel one had MET $\approx 14.4s$ which is $\approx 75\%$ less. The partitioner continued the reducing trend at MET $\approx 9.4s$, improving the parallel version by $\approx 42\%$ (fig. 5.9).

Across all dataset the parallel version observed increase of memory consumption of $\approx 14\%$ (fig. 5.10) (tab. 5.7).

Tab. 5.6: Nuget package ranking experiments parameters

Name	**Value**
Map degree of parallelism	10
Reduce degree of parallelism	10
Iterations	10
Small dataset	1000 packages
Medium dataset	5000 packages
Large dataset	10000 packages

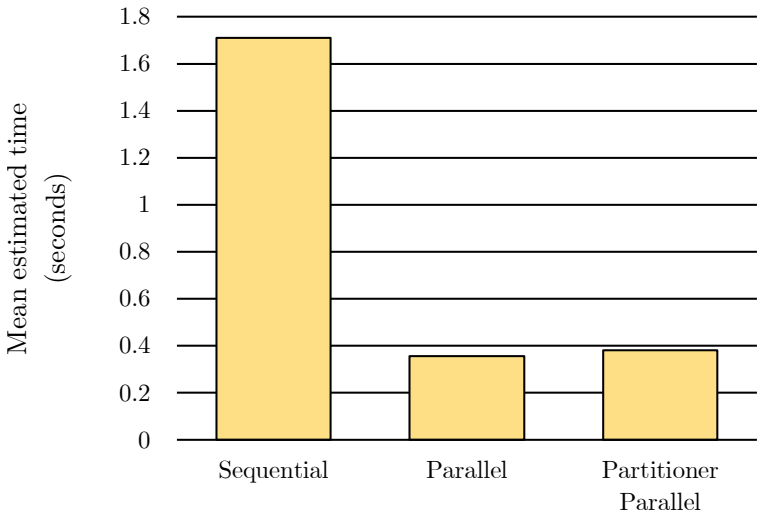

Fig. 5.7: NuGet package ranking performance - small dataset

All experiments were properly displayed and described. Next chapter will discuss the implications of these results and formulate a set of guidelines based on them.

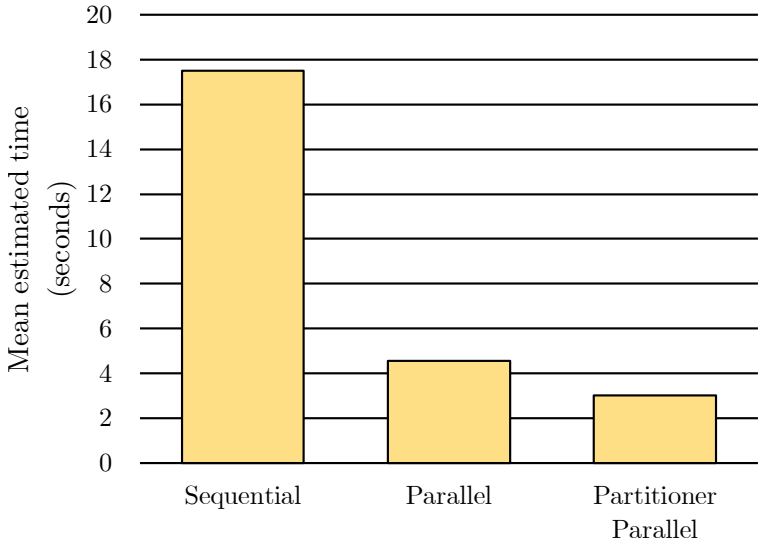

Fig. 5.8: NuGet package ranking performance - medium dataset

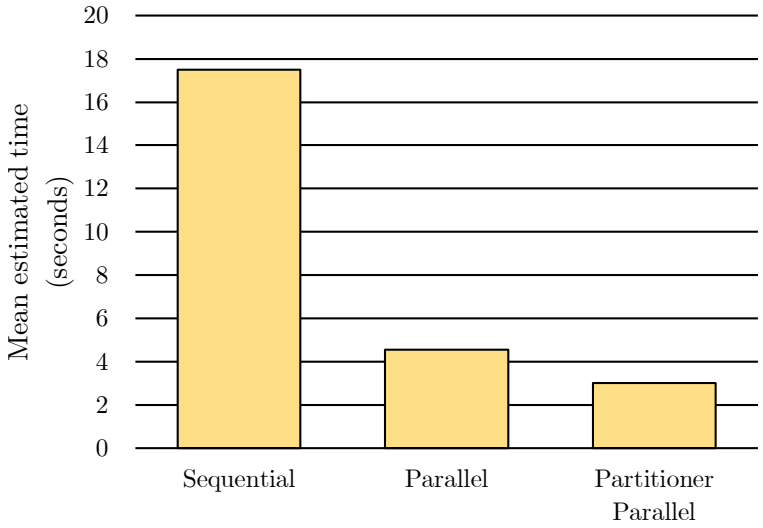

Fig. 5.9: NuGet package ranking performance - large dataset

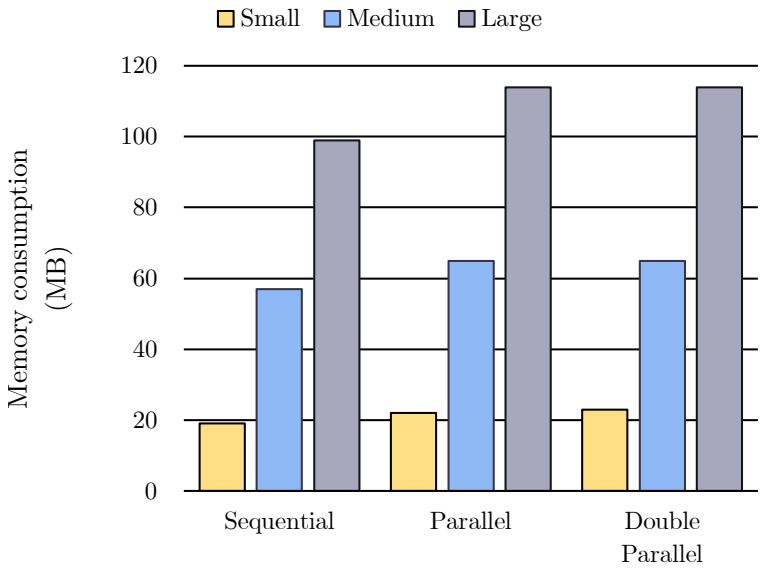

Fig. 5.10: NuGet package ranking memory consumption

Tab. 5.7: Nuget package ranking benchmarking results

Version	**Mean [s]**	**Error [s]**	**StdDev [s]**	**Gen 0**	**Gen 1**	**Gen 2**	**Alloc. [MB]**
Small dataset							
Sequential	1.709	0.004	0.004	2000	0	0	19
Parallel	0.357	0.007	0.012	2000	1000	0	22
PartitionerParallel	0.000	0.006	10.33	2000	1000	0	23
Medium dataset							
Sequential	17.491	0.2455	0.2296 s	7000	3000	0	57
Parallel	4.564	0.0684	0.0606	8000	4000	0	65
PartitionerParallel	3.007	0.0577	0.0730	8000	3000	0	65
Large dataset							
Sequential	56.909	0.6018	0.5630	12000	5000	0	99
Paralllel	14.377	0.0644	0.0571	14000	5000	0	114
PartitionerParallel	8.428	0.0359	0.0300	14000	5000	0	114

Chapter 6

Discussion

After carefully designing the experiments in chapter 3 and 4 and conducting them in chapter5 this part will discuss the outcomes of the benchmarks. After interpreting and analyzing the results, a set of guidelines for future .NET parallel ventures will be presented.

At the stage of development of algorithms and software it quickly became clear that benchmarking is crucial when programming for performance. Sometimes a small tweak in unimportant subroutine would completely change the speed of execution, thus many iterations of the code were discarded even before the experiments proper. Some of these haphazard implementations were included in QuickSort part (section 5.1). Async-await state machine is an useful construct, but it was build for fluidity of applications, not for performance. Without benchmarking such code might have slipped into production environment for a potential disaster. *BenchmarkDotNet* enables programmers to scale-back the tests so they can be used during development or even use them as part of their unit test suite.

Platform version is very significant when it comes to performance. .NET, thanks to being open source, is often updated with small fixes and improvements which aren't always included in the major release notes. Such updates may have big impact on the application, especially if it uses high-level constructs like LINQ. This was confirmed by tests conducted in QuickSort part (section 5.1). Switching from .NET Framework 4.7 to .NET Core 3.1 brought major improvements across all versions of the algorithm. Further test were conducted only with .NET Core 3.1 since the first example was enough to drive the point.

Another point that came out of QuickSort part (section 4.1) is that specific implementation matters. Imperative version of the algorithm highly overperformed the functional one and didn't benefit as much from parallelisation. The cost in this case came from other factors: length of the implementation, understandability and reliability. Imperative version took many iterations to get just right, even with something relatively simple as the QuickSort algorithm. The end result is convoluted, hard to read and understand piece of software. On the other hand, version using LINQ was simple to write and end result is comprehensive and concise. Any further maintenance over this version would be significantly easier. One has to consider cost-benefit of programming for performance when choosing the specific implementation.

When using parallelism it is important to gauge the amount of tasks spawned by the software. Otherwise, resource heavy context switches will overhelm any potential speed benefits of parallel programming. This was demonstrated in QuickSort algorithm (section 5.1) and Mandelbrot set drawing (section 5.3). In the former, lack of depth control together with async-await rendered the implementation completely inefficient. In the latter, using two nested *Parallel.For* loops performed worse than using only one. We can hypothesize that further parallelisation would decrease the performance even more.

Data parallelism Fork/Join and MapReduce patterns proved to be useful and powerful. The former was used in the implementation of K-Means algorithm (section 4.2) and the latter in software ranking NuGet packages (section **??**). Both of these patterns can be implemented in a generic way, allowing programmers to drop them in when they see fit. Parallelisation was again streamlined thanks to the use of LINQ, turning the queries into parallel ones required few PLINQ lines. This makes the code clear and easy to understand, on contrary to often convoluted parallel implementations.

Load balancing with a data partitioner is a strategy worth considering when implementing parallel queries. This addition was especially performant with larger datasets in NuGet package ranking software (section 5.3), not so much with smaller datasets like in the case of Madelbrot algorithm (section 5.3).

Almost all parallel implementations came with a cost of memory consumption oscillating around 15%. Considering that memory is abundant in modern computing machines, this shouldn't be an important factor unless working on a device which has for some reasons significant memory constraints. Special case can be made for algorithm which allocate a lot of memory by object creation, as in the case of Mandelbrot algorithm (section 4.3) which creates millions of objects inside its functions. When Garbage Collector cleanups become the bottleneck and stop the program from executing, it might be worth to consider using value types structures instead (section 5.3).

To summarize this analysis:

- Benchmarking is crucial at all stages of development.
- *BenchmarkDotNet* is recommended as the go to library for .NET benchmarking.
- Modern platforms are optimized for performance, working on newest versions should be a priority.
- Parallel programs should be written with understandability, reliability and maintainability in mind. Functional paradigm offers great tools which enable this goal.
- Depth of parallelisation should always be controlled for, exact numbers should be data driven.
- Fork/Join and MapReduce patterns are recommended in data parallelism problems.
- Data load balancing with partitioners is recommended for large datasets.
- Do not preemptively optimize for memory consumption unless it is required by specific scenario.
- If scenario includes creating many objects inside local function scope, consider using value type instead of reference type.

Chapter 7

Conclusions

The goal of this thesis was to explore how application of concurrency tools proviced by the .NET platform on algorithms and software impacts their performance. This entailed implementing each of the software pieces in multiple versions, using various tools like Task Parallel Library or PLINQ and executing extensive experiments on them.

The outcome of this thesis are multiple implementations of following software pieces:

- QuickSort
- K-means clustering
- Mandelbrot set drawing
- Nuget package ranking

These versions used among others: TPL's *Parallel.For*, PLINQ's parallel queries, load balancing data partitioners and Fork/Join and MapReduce patterns. For each of them tests measuring mean estimated time, memory consumption and others were conducted. Test data was presented in form of tables, graphs and written summaries. Finally, the results of the tests were analyzed and catalyzed into a set of guidelines to aid future parallel ventures in .NET.

Key aspect of this thesis was starting the benchmarking in the early stages of development. Thanks to that many early, faulty implementations were discarded and the versions left might be called truly optimized. The hardest part was the implementation alone, parallel programming even for experienced person is complicated and requires a great dose of focus and dedication. Hopefully this paper will inspire next developers to conduct similar experiments on different algorithms and setups, further expanding this body of literature.

References

[1] A. Akinshin. *Pro .NET benchmarking: The art of performance measurement*. Apress, 2019.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485. Association for Computing Machinery, 1967.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd ed.)*. Addison-Wesley, 2006.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. Mit Press, 3 edition, 2009.

[5] W. S. Daan Leijen and S. Burckhardt. The design of a task parallel library. Technical report, Microsoft Research, 2009.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, San Francisco, CA, 2004.

[7] A. Douady and J. H. Hubbard. Etude dynamique des polynômes complexes, prépublications mathémathiques d'orsay. Technical report, Société Mathématique de France, 1984 / 1985.

[8] S. A. et al. Parallel Computing Research at Illinois: The UPCRC Agenda. Archived 2018-01-11 at the Wayback Machine, November 2008.

[9] E. Fuentes. Concurrency - Throttling Concurrency in the CLR 4.0 ThreadPool. *MSDN Magazine Volume 25 Number 09*, September 2010.

[10] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, May 1988.

[11] H. Hagedoorn. AMD Ryzen 3000 - 7nm ZEN2 - Architecture and Chiplet Design, November 2019. `https://www.guru3d.com/articles-pages/tech-preview-amd-ryzen-with-ryzen-3950x,3.html` [Accessed 23/05/2021].

[12] I. T. J. Intel Corp. Desktop Products Group. Hyper-threading technology architecture and microarchitecture. l, 6(1):4–17, February 2002.

[13] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7*, July 1978.

[14] Microsoft. Introduction to .NET, June 2011. `https://docs.microsoft.com/en-gb/dotnet/core/introduction` [Accessed 21/03/2021].

[15] Microsoft. Custom Partitioners for PLINQ and TPL, March 2017. `https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/custom-partitioners-for-plinq-and-tpl?redirectedfrom=MSDN` [Accessed 27/05/2021].

[16] Microsoft. Managed Execution Process, March 2017. https://docs.microsoft.com/en-gb/dotnet/standard/managed-execution-process [Accessed 21/03/2021].

[17] Microsoft. Managed Threading, March 2017. https://docs.microsoft.com/en-gb/dotnet/standard/threading/ [Accessed 25/03/2021].

[18] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, page 38(8), April 1965.

[19] U. of California. Wine Quality Data Set. https://archive.ics.uci.edu/ml/datasets/wine+quality [Accessed 22/05/2021].

[20] R. Pike. Concurrency is not parallelism. Waza conference, 11 January 2012.

[21] G. Popov, N. Mastorakis, and V. Mladenov. Calculation of the acceleration of parallel programs as a function of the number of threads. *Kliment Ohridski" blvd*, 2010.

[22] J. Richter. *CLR via C#*. Microsoft Press, 4 edition, 2012.

[23] J. Richter. *CLR via C#*. Microsoft Press, 4 edition, 2012.

[24] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. Technical report, Association for Computing Machinery, 2006.

[25] R. Terrell. *Concurrency in .NET modern patterns of concurrent and parallel programming*. Manning, 2018.

[26] B. University. Mandelbrot Set Explorer: Mathematical Glossary. http://math.bu.edu/DYSYS/explorer/def.html [Accessed 22/05/2021].