A Parallel Skeleton Library for DICE

Antonios Valais

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2018

Abstract

This thesis presents the design, implementation and evaluation of a type-safe shared-memory parallel Skeleton library with full compatibility on the Distributed Informatics Computing Environment (DICE). The library has been designed to minimise the complexity of use, provides four data-parallel Skeletons (Map, Reduce, MapReduce, Scan), and has been implemented with the C++11 standard and PThreads (C++11) library. The evaluation of the library has been conducted with a simple benchmark suite and "real-world" problems, and the performance and programmability of the library has been compared with a sequential and non-Skeleton parallel implementation of each problem.

Acknowledgements

- I would like to express my gratitude to Murray I. Cole, who supervised this thesis, and introduced me to the world of Algorithmic Skeletons.
- To my family, where without their firm support I would had never accomplish this work.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Antonios Valais*)

Table of Contents

1 Introduction **1**

1.1 Simplified Parallel Programming 1

1.2 Principal Goal and Proposed Solution 2

1.3 Thesis Structure 3

2 Background and Related Work **5**

2.1 Higher-order Functions 5

2.2 Algorithmic Skeletons 6

2.3 Related Work 7

2.4 Information about DICE 8

3 Design and Implementation **9**

3.1 Overview 9

3.1.1 Interface Design Overview 9

3.1.2 Skeleton Module Overview 10

3.2 Map Skeleton 11

3.2.1 Interface 12

3.2.2 Implementation 13

3.3 Reduce Skeleton 19

3.3.1 Interface 19

3.3.2 Implementation 22

3.4 MapReduce Skeleton 28

3.4.1 Interface 28

3.4.2 Implementation 31

3.5 Scan Skeleton 37

3.5.1 Interface 38

3.5.2 Implementation 39

4 Evaluation **47**
4.1 Overview 47
4.2 Technical Information 47
4.3 Benchmarks 48
4.3.1 Map Skeleton 49
4.3.2 Reduce Skeleton 53
4.3.3 MapReduce Skeleton 56
4.3.4 Scan Skeleton 62
4.4 Real World Applications 68
4.4.1 K-Means clustering with Reduce and Map 69
4.4.2 K-Means clustering with MapReduce 72
4.4.3 Cumulated Function with Map and Scan 74

5 Conclusions and Future Work **79**

Bibliography **81**

Chapter 1

Introduction

Since the beginning of the 21st century, the trend that each subsequent generation of processors was faster than its precedent one had come to an end. This is mainly due to the fact that Moore's law [24] can no longer be sustained with the available technology.

However, the growth of processing power on computer systems is maintained by incorporating multiple processing units (cores) in a single computing component. Therefore, for the effectiveness of this solution, the existence of software that exploits multiple cores is necessary.

The creation of such software, is attained with a special technique that is commonly termed "Parallel Programming", and its popularity has been increased the last years, as a consequence of the embedding of multi-core processors in most electronic devices. From workstations and supercomputers, to smart-watches and cutting-edge refrigerators, pervasive parallelism has become a fact.

1.1 Simplified Parallel Programming

Parallel programming in most cases is accompanied with a major draw-back, and this is created due to the fact that the available mainstream programming libraries for developing software that holds the property of concurrency, are chained with a significant degree of complexity which in most cases is irrelevant with the algorithm under development. For instance, the programmer with the majority of those libraries, is obligated to express his algorithmic ideas in a concept where the execution order of each statement is non-deterministic, and so additional instructions need to be introduced to ensure the correctness of the program's outcome. Consequently, with this class of libraries, the probability of creating erroneous software is augmented.

For this reason, researchers have created an abstraction layer over the aforementioned libraries, that captures common parallel patterns and allows the implementation of a parallel algorithm to be expressed as a sequential program. This layer provides specified higher order functions, called “Algorithmic Skeletons” [6], and the programmer simply defines the functions that express the operation that needs to be applied concurrently, while the Skeletons encapsulate efficiently the parallel implementation.

1.2 Principal Goal and Proposed Solution

Since 1990, more than 20 parallel Skeleton libraries have been implemented [15]. In this list, the “average” parallel Skeleton library, supports multi-CPU and/or multi-GPU hardware, and provides a set of Data-parallel and Task-parallel Skeletons, which is accompanied by a Skeleton-based data type. Based on their programming paradigm, they can be categorized in the groups of: “Functional” (Eden [20], HDC [17]), “Object-oriented”(Muesli [5], SkeTo [21]), “Coordination” (P3L [3], SCL [10]), and “Imperative” (eSkel [7], SKElib [9]).

However, parallel Skeleton libraries have dependencies on software frameworks and tools, that can make them hard to be used in teaching context, as those dependencies have to be maintained by students and teaching instructors. Specifically, in the School of Informatics in University of Edinburgh, the standard Distributed Informatics Computing Environment (DICE), the platform that is used by students and staff, is upgraded in a regular basis, and as a consequence the aforenamed dependencies form an impediment. For example, SkePU2 [14] parallel Skeleton library, requires a GCC compiler of version 4.9, LLVM [18], and to enable its full functionality, CUDA [23], OpenCL [25], and OpenMP [8] frameworks.

Therefore, the project’s goal is “*the design, implementation, and evaluation of a parallel Skeleton library, that operates without dependencies of unavailable tools in the standard DICE environment, and provides a simple programming interface to allow students already familiar with C++ to investigate the world of Skeletons*”.

The thesis presents a type-safe parallel Skeleton library, with 4 data-parallel Skeletons (Map, Reduce, MapReduce, and Scan), implemented with the 2011 C++ standard and the PThreads library (C++11), in order to be used for structured parallel programming in teaching context, with system requirements that can be covered, and not only, in a DICE environment.

1.3 Thesis Structure

In chapter 2, an introduction to Algorithmic Skeletons and their connection with higher-order functions is given, followed by a presentation of related work, and a description of DICE. In chapter 3, the design and implementation of our library is presented. We discuss the common interface and module architecture of our 4 data-parallel Skeletons, and then we analyse the provided interface and implementation of each Skeleton. Furthermore, in chapter 4, the evaluation of our Skeleton library is analysed, which is separated in two parts. In the former, the evaluation based on a simple benchmark suite is presented, and the latter, demonstrates the performance and programmability of our library on "real-world" applications. Finally in the last chapter, accompanied with the conclusions of our contribution to the world of Skeletons, future extensions of our work are stated.

Chapter 2

Background and Related Work

In this chapter, we relate the terms of Algorithmic Skeleton and higher-order function, and then we present the work of another Skeleton library. Finally, a description of DICE is given.

2.1 Higher-order Functions

In programming, a named block of code is termed as “function”. Usually, a function accepts various inputs, processes those inputs, and returns a result. For example, we can define a function that accepts an integer (the input), increments the value by 1 (the process), and returns the summation (result). In many occasions, we are not interested of the exact implementation of the process, as long as we are informed of the function’s operation, and with such a case, our only concern is how to “operate” the function. This is indicated by the function’s type, and in our example this can be denoted as:

$$\text{f : integer} \longrightarrow \text{integer}$$

Figure 2.1: The type of a function named “f”, that accepts an integer and produces an integer.

Suppose that we would like to change the operation of this function, and instead of addition, we want to apply subtraction. In this case, the terms will remain the same, and only the operand will change. One way to apply this modification is to edit the “block of code”, and replace the symbol of the operation, but this approach does not allow the transition to be made in runtime. Therefore, we can define our function to accept an additional argument, that is used as the operator for the two values, and this new argument, that is a function as well, accepts two arguments and applies the desired operation on them. A function that accepts as argument, or returns, another function, is termed “higher-order function”. In our example, our modified function’s type can be denoted as:

$$f \; : \; ((\text{integer} \rightarrow \text{integer}), \text{integer}) \rightarrow \text{integer}$$

Figure 2.2: The type of a function named “f”, that accepts another function as the first argument (same as figure 2.1), an integer as the second, and produces an integer.

There are many patterns that can be expressed as a higher-order function, and in most cases they apply a user-defined function (the operation) on a collection. In functional programming, some of the most common higher-order functions are: Map, Reduce, and Scan.

Map, is an operation on a collection, where we apply a function, called “elemental”, on each element of that collection, and retrieve a new collection, or the updated one, with the elemental’s outcome per element.

Reduce (or Fold), is the operation when we want to aggregate a number of elements to a single one. For example, the aggregation could express something simple, as the summation of integers, or perhaps a more complicated procedure, as merging a set of databases. The function applied to the elements is termed the “combiner”.

Scan (or Prefix-sum), is the operation when we want to accumulate a collection of items, like in the case of Reduce, but in addition to producing a single instance with the total result, we also want a partial reduction up to each item. In other words, for every element in a collection, Scan is a reduction that includes the element and all its precedents. The operation applied is called “successor”.

2.2 Algorithmic Skeletons

An Algorithmic Skeleton is a higher-order function that applies a pattern, with the goal as defined by Murray Cole [6] being:

> *“to define a programming system which, while appearing non-parallel to the programmer, admits only programs which it can guarantee to implement efficiently in parallel”*

In other words, a Skeleton provides an interface, similar to a common higher-order function, that creates an abstraction layer over the parallel implementation of the desired pattern, and not only ensures that the implementation is attained with an optimum performance, but also allows the programmer to use the Skeleton without profound knowledge of the underlying hardware system.

An example for those properties is that a Skeleton could support either, or both, Message-Passing or Shared-Memory parallel programming models, and on top of that, a Skeleton could provide task-parallel and/or data-parallel problem decomposition. Moreover, as a Skeleton's interface is based on higher-order functions, basic Skeletons can be used for the composition of more complex patterns or Skeletons.

Finally, Skeletons can be divided into three main categories . Data-parallel, Task-parallel and Resolution [15]. Some data-parallel Skeletons are Map, Stencil, Reduce, Scan, and Fork, while some examples for the category of task-parallel Skeletons are Sequential, Pipeline, and Farm. A classic resolution Algorithmic Skeleton is the Fixed Degree Divide & Conquer.

2.3 Related Work

The parallel Skeleton library that influenced our library's interface and choice of Skeletons is SkePU2 [14]. SkePU2 is an improvement of SkePU [13], a C++ parallel Skeleton library targeting multi-core and multi-GPU systems, by providing type-safe Skeletons and greater flexibility. SkePU2 is implemented in C++ as well, but with the use of variadic templates (C++11) and LLVM as "source-to-source" translator.

The library provides 6 Algorithmic Skeletons : Map, Reduce, Scan, MapReduce, MapOverlap, Call, and two main smart-containers : Vector and Matrix. After the creation of a Skeleton functor, the smart-containers can be passed to the parenthesis operator, not only by reference, but also by iterator, where the latter case allows the Skeletons to perform on a subset of the smart containers.

Specifically, the Map Skeleton of the library is three-way variadic. It accepts as first argument the output container, and then a group of arguments (of any number N) holding the input containers. If N is greater than 1, an element of each input container will be passed to the elemental. Optionally, the Skeleton accepts scalar variables that are passed directly to the elemental. Finally, Map can provide the index of each element to the elemental.

Reduce Skeleton presumes the associativity of the combiner, and besides the case of a Vector, the reduction of a Matrix can be achieved in five ways: in one or two dimensions, per column or per row, and as a row major order Vector.

MapReduce Skeleton is simply a Reduce after a Map, Scan Skeleton can be either inclusive or exclusive for a given associative successor, and MapOverlap Skeleton is a 1D or 2D stencil operation that can be parametrised for the boundary values. Finally, Call Skeleton has the only task to call the user defined function(s) on its input in the same way as the Map Skeleton.

The implementation of SkePU2 is divided into three parts. The sequential, the source-to-source compilation with LLVM, and the back-end that is responsible for the parallel implementation of the Skeletons. The last part is an enhanced version of the SkePU [13] implementation, aided by the new features of 2011 C++ standard. The second part has the role to transform the sequential code provided by the user to a parallel syntax that can be used by OpenMP, CUDA, and OpenCL frameworks.

SkePU2 library's dependencies are a GCC compiler (version 4.9), LLVM, and if CUDA is used, Nvidia CUDA compiler (version 7.5).

2.4 Information about DICE

The Distributed Informatics Computing Environment (DICE) provides "computing infrastructure to support the research, teaching and administrative requirements of the School of Informatics at the University of Edinburgh" [12]. It is a Linux distribution, based on Scientific Linux, and it is available to students, research postgraduates, and staff. To this date, for the development of C++ programs, DICE provides the GCC (version 4.8) and Clang (version 3.4) compilers, and there is no support for the CUDA or OpenCL framework.

Chapter 3

Design and Implementation

3.1 Overview

In this chapter, firstly we will present the common API features of our Skeletons, to provide a general idea of their usage, which have been influenced by the design choices of Skepu2 Skeleton library. Secondly, we will discuss our common Skeleton module architecture, and afterwards, the description of each Skeleton's specific design and implementation is presented.

3.1.1 Interface Design Overview

Each Skeleton's usage is divided into two stages - the initialisation, and the execution. In the former stage, the user calls the equivalent for each Skeleton function, in order to instantiate an object that encapsulates the Skeleton's logic, where this function in each case, has the same name as the Skeleton to be used. For example, for "Reduce Skeleton", the function's name is "Reduce".

Commonly with these constructor-functions, the user needs to provide for the first argument(s) the logic that the Skeleton will apply, which could be expressed either through a plain C++ function pointer, a lambda function, or any C++ object that overloads "operator()" (including "std::function" objects). Secondly, as the last argument, the user could define the number of threads to be used for the Skeleton's instance, otherwise a call to ''‘std::thread::hardware_concurrency()", which returns the number of hardware thread contexts, including virtual ones (i.e. twice the number of CPU cores), will be made and the value returned will be assigned to the number of threads.

The second stage for a Skeleton, is to call the functor's parenthesis operator by providing the data under process, and a buffer for the operation's result. For this operator, the first argument relates to the output buffer, and the second one to the input collection. The type of those two arguments is an "std::vector" with the appropriate element type for each case, and the exception occurs for the former argument with the Reduce Skeleton, where the output parameter has the same type as the input vector's element. In addition through this call, the user could provide further arguments, and those will be passed directly to each invocation of the function(s) the user has provided in stage 1. This allows the programmer to have access to external data inside these functions, a property which otherwise would require the use of global variables. However, those extra arguments must be treated as read-only variables, otherwise, in multiple scenarios, based on user's actions, the Skeleton's behaviour is undefined. This is due to the fact that, an extra argument, as it is actually a shared memory location between each function invocation, a write operation on it inside the function's scope, is not, and cannot be, synchronised by the Skeleton, thus will lead to an undetermined value. This is an agreement with the programmer, and the library does not attest that the programmer will abide with it.

```
int extraArg = 3; size_t nThreads = 4;
std::vector<int> input = {1,2,3,4,5};
std::vector<bool> output;
auto elemental = [](int e,int extraArg){ return e!=extraArg;};

auto map = Map(elemental,nThreads); //first stage
map(output,input,extraArg); // second stage

//(result) output: {true,true,false,true,true}
```

Listing 3.1: An example of the two stages with Map Skeleton.

3.1.2 Skeleton Module Overview

The common architecture of our modules, is expressed through a wrapper class, which encapsulates the components of each Skeleton. Generally each module has three components: the implementation class, the threads' argument class, and the class(es) that bound the developer's defined behaviour of each Skeleton to an object. An abstract module UML is presented in figure 3.1.

The purpose of the wrapper class is to forbid the user from creating an "Implementation" object by calling the constructor directly, as we preferred to provide a special function for this task, similarly with Skepu2 Skeleton library, which in our humble

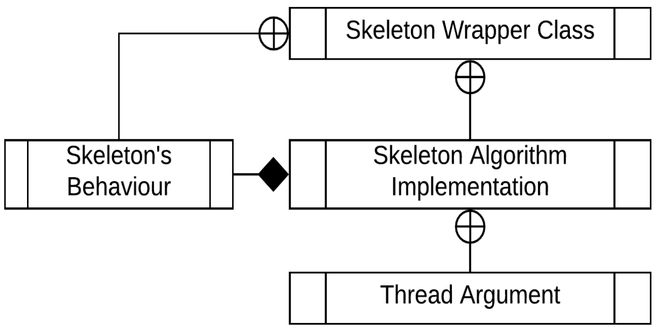


Figure 3.1: Abstract Skeleton Module.

opinion is more elegant.

Furthermore, this design approach could have been altered with a simpler one, without requiring the wrapper class. In this case, the “Implementation” class could be the frontier of our module, with the remaining classes of the module embedded to it. Nevertheless, this design choice was given thought thoroughly, and the conclusion was to stick with the use of a wrapper class, as this will provide better encapsulation of the whole module for potentially future enhancements, while maintaining the access level from the user side to only the two calls for the two stages of handling each Skeleton.

3.2 Map Skeleton

In sequential programming, the Map pattern is implemented with an iteration form, where the two main characteristics consist of that every iteration step is independent from the others, and secondly the number of elements in the collection is known in advance. Taking a step back, this description matches the simplest case of a concurrent application, as the first characteristic ensures no access race over data, and the second allows us to create divisions over the collection, based on the number of processes, without further concerns on the assumption of a simple implementation.

In the parallel world, the Map operation executes the same task as the sequential version, potentially in a fully distributed way, but depending on the programming language, and parallel library that might be used, this concurrent pattern introduces both execution time and lines of code overhead compared to the sequential version. The minimisation of this overhead, in both dimensions, is the purpose of existence of the Map Skeleton.

With the Map Skeleton, the whole process of turning the Map pattern into a concurrent one, is encapsulated by the implementation. Thus the programmer will see the parallel Map operation as a sequential one, through a function call, similarly with the C++ standard library function “transform”.

3.2.1 Interface

One of the main goals for our Skeleton library is to provide a minimum complexity of programmability. For this reason, Map Skeleton follows the common two-phase interface of our Skeleton library. An example is given in listing 3.4.

3.2.1.1 Initialising a Map object

The first phase consists of the function-object (functor) initialisation. By calling the global function “Map”, the user passes as a compulsory argument the elemental function, and as an optional argument the number of threads to be used with this Map instance. In case the second argument is not defined, the implementation uses the maximum hardware concurrency available on the machine. The returned value of the “Map” function is an instance of the Map Skeleton implementation class, which encapsulates the the whole algorithm of the Map Skeleton.

3.2.1.2 Invoking the Map operation

The next stage, after the user has created the functor, is to call the Map operation by passing two, or arbitrary more, arguments to its overloaded parenthesis operator. The first argument is an “std::vector”, where the Skeleton’s outcome will be placed, and the second argument is again an “std::vector”, where the user holds the input data that the Map operation will process. The input vector’s content type, does not necessary have to match the one of the output vector.

Following the second argument, the interface allows the user to pass extra arguments, from none to more, where those arguments are not processed by any mean by the Skeleton, and are passed directly to the user defined elemental function. As the described interface of the parenthesis operator covers the required communication between the user and the Map Skeleton, the return type of the functor is defined as void.

3.2.1.3 Elemental function

The user must follow a strict interface on his definition of the elemental function. The requirement of this interface, is that the function takes as a first argument the element of the input vector, and will return a value, of different instance, where no restriction exists for the type to remain the same. Furthermore, in case that the user will provide additional arguments to the Map functor call, the elemental function must accept those extra arguments in the same order. The function's type is shown in figure 3.2.

$$\text{elemental} : \mathrm{T}_1 \ [, \mathrm{Ts}...] \longrightarrow \mathrm{T}_2$$

Figure 3.2: Function type of elemental, with $T_1, T_2, Ts...$ of any type. T_1 is the Input element type, T_2 is the Output type, and $Ts...$ is the optional variadic type of extra arguments.

3.2.1.4 Usage example

```cpp
int extraArg = 3; size_t nThreads = 4;
std::vector<int> input = {1,2,3,4,5};
std::vector<bool> output;
auto elemental = [](int a,int extraArg){ return a!=extraArg;};

auto map = Map(elemental,nThreads); //first stage
map(output,input,extraArg); // second stage

//(result) output: {true,true,false,true,true}
```

Listing 3.2: An example of the two stages with Map Skeleton.

3.2.2 Implementation

For reader's convenience, we will begin our implementation description with a class hierarchy illustration of our Map Skeleton. The UML in figure 3.3 presents the final version of our design. The architecture of the Skeleton remained the same, as shown in figure 3.3 through all our versions, except the second version, where we introduced two additional classes to help with the cause as explained in section 3.2.2.2.

Describing in a top-down way, as illustrated on the UML diagram, MapSkeleton class is a wrapper class to encapsulate the privately defined inner classes Elemental and MapImplementation. Thus, in order to instantiate MapImplementation, the existence of a "friend" function, called Map, takes place, which acts as a simplified constructor or Factory pattern, with the first argument (templated) holding the elemental function and the second (optional) argument defining the number of threads to be

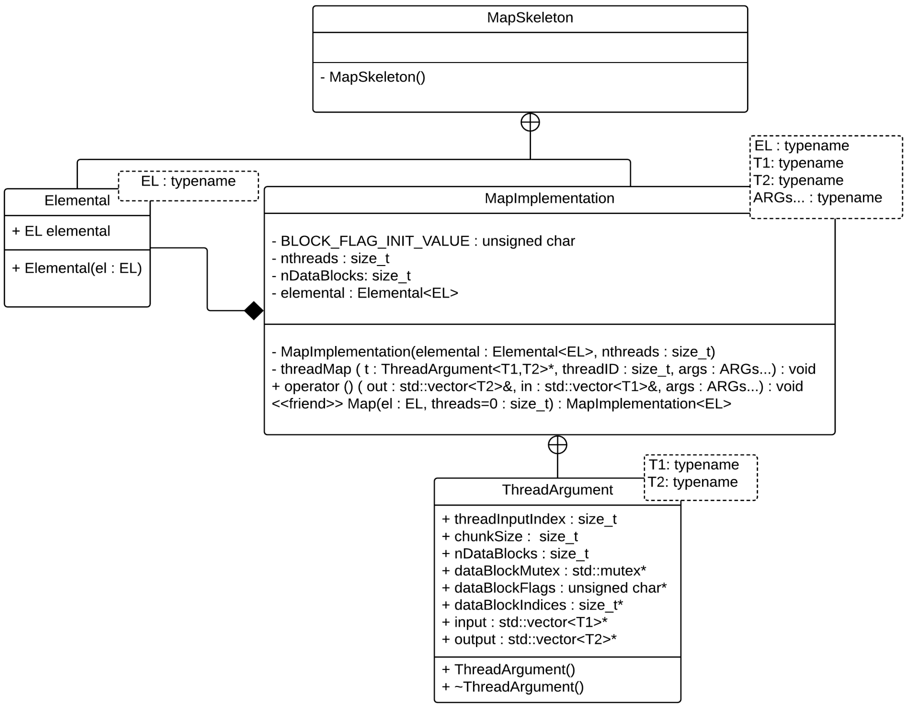

MapSkeleton
- MapSkeleton()
Elemental
EL : typename
+ EL elemental
+ Elemental(el : EL)
MapImplementation
EL : typename
T1: typename
T2: typename
ARGs... : typename
- BLOCK_FLAG_INIT_VALUE : unsigned char
- nthreads : size_t
- nDataBlocks: size_t
- elemental : Elemental<EL>
- MapImplementation(elemental : Elemental<EL>, nthreads : size_t)
- threadMap (t : ThreadArgument<T1,T2>*, threadID : size_t, args : ARGs...) : void
+ operator () (out : std::vector<T2>&, in : std::vector<T1>&, args : ARGs...) : void
<<friend>> Map(el : EL, threads=0 : size_t) : MapImplementation<EL>
ThreadArgument
T1: typename
T2: typename
+ threadInputIndex : size_t
+ chunkSize : size_t
+ nDataBlocks : size_t
+ dataBlockMutex : std::mutex*
+ dataBlockFlags : unsigned char*
+ dataBlockIndices : size_t*
+ input : std::vector<T1>*
+ output : std::vector<T2>*
+ ThreadArgument()
+ ~ThreadArgument()

Figure 3.3: Map Skeleton Module.

used for this instance. The core of the Skeleton is composed of MapImplementation and ThreadArgument classes, and their behaviour varies between our three different implementations.

A summary of those three versions is as follows. In our first version (**Baseline**), a simple approach is implemented, where we divide in roughly equal chunks the input collection and each thread iterates its correlated chunk and produces the result by applying the elemental function. The second version (**Version 2**), removes thread creation on every functor call (stage 2), by creating the threads on the first stage of the Skeleton, and keeping them idle until there is work assigned to them, and this is achieved with the use of a thread pool. Finally the third version (**Version 3**), introduces a dynamic load balance mechanism, to achieve higher performance in case the required process time for each element is different. Each version is analysed further in the following sections.

3.2.2.1 Map: A simple approach (Baseline version)

The initial solution was none other than the idea of equally dividing (as equally as possible) the input vector to the number of threads, and then having each thread to apply the elemental function to each own chunk's elements, while assigning the function's return value to the output vector.

When the user calls the parenthesis operator, the number of threads defined to that point, is checked whether it exceeds the size of the input vector, which in that case will result to threads being created and staying idle. In this case we update the number of threads to half the size of the input vector. An argument exists here, to set the number of threads in this case, equal to the size of the input vector, so we can achieve maximum concurrency, and thus minimising the total execution time. This is partially true, as it only holds in the case where the execution time of the function's work per element, is at least twice, than thread's creation time. Analysing this further, it holds that if the function's work requires exactly the same time as a single thread creation, we will achieve the minimum total execution time, by creating at most half of input size threads. Following this direction, we can conclude that the less the execution time of the elemental function, the less number of threads need to be created to achieve the minimum total execution time, as long as the input size is not greater than the available hardware concurrency - or user defined number of threads. Finally, since we do not have a mechanism to indicate us the actual execution time of an elemental invocation, we decided to take only one step towards this logic, and on the case described before, we assign the number of threads equal to half of the input size.

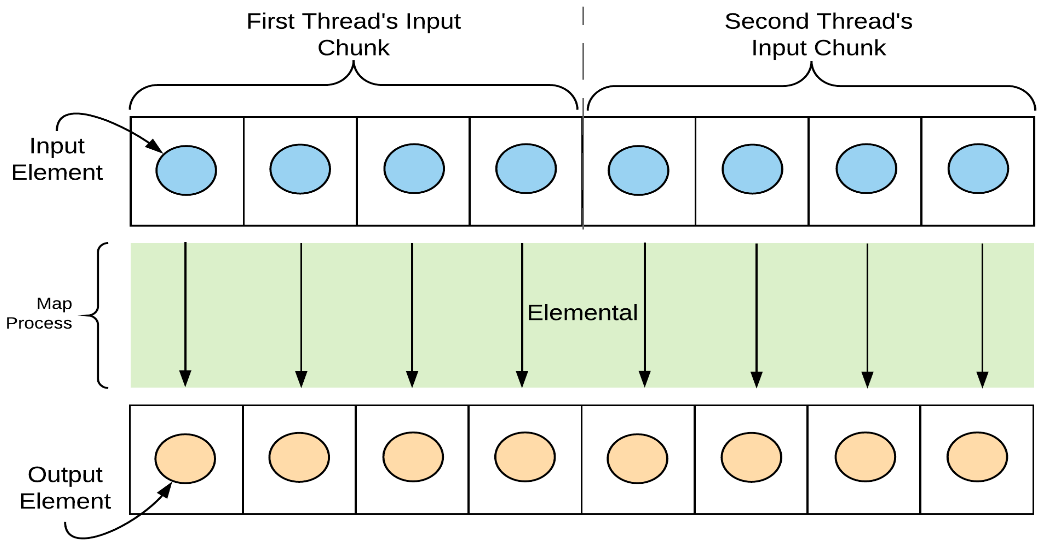


Figure 3.4: Vector of 8 Elements, handled by 2 threads.

Furthermore, the remaining implementation of the parenthesis operator, is to create the ThreadArguments and the actual thread processes. The important thing here is, that the threads do not have access on the output vector itself, but to a secondary "tempOuput" vector, and after the thread join in this method, the temporary vector is assigned to the actual output object. This is an extra safety measure, for the specific case that the user will pass the same object, both as an output vector argument, and as "extra" argument, which will allow the user to modify the output vector inside the elemental function, even though the usability contract for the elemental function is that all arguments must be treated as read-only.

Finally in this version, the "threadMap" method, which is the thread's task, iterates over the thread's chunk on the input object and assigns the return value of the elemental function to the equivalent index of the temporary output vector.

3.2.2.2 Map: An implementation with a thread pool (Version 2)

With the completion of the first version of Map Skeleton, we wanted to investigate further for a way to remove the overhead of thread creation on every parenthesis operator call of our Skeleton. Hence, we thought that we can create the threads on the first stage of the Skeleton, where the threads will stay idle, without consuming resources, until the user will activate the second stage. To test this design, we implemented the Skeleton so each Map object, will have its own thread pool (threads being idle while work is not available).

The first stage of the Skeleton, includes the creation of a thread pool, which is an instance of the new class ThreadPool, with size equal to the defined thread number. This class has two roles: to receive and execute the tasks from the Map Skeleton, and secondly to indicate the Map Skeleton when the tasks are completed.

For this goal, ThreadPool incipiently creates two groups of semaphores, each equal in size as the number of threads in the pool. All semaphores could either have zero or one as their value, where the first group indicates the availability of a task for the corresponding thread, while the second group denotes the completion of the task likewise. After this, the creation of the thread processes takes place, and their execution method is presented on algorithm 1.

ThreadPool's second public method, after the constructor, is "receiveTasks". This method takes as argument the tasks given from the Map Skeleton, assigns them to the threads, and after this, signals the equivalent threads to execute their newly assigned job. Finally this method returns only after a job completion signal has been received from all the working threads.

The last method of the aforementioned class is its destructor, and it is important to

```
Data: task, terminate
while true do
    wait for task;
    if terminate is true then
        exit;
    else
        task.execute();
        signal that work is done;
    end
end
```

Algorithm 1: ThreadPool's thread method.

talk about this, as through its calling, the flag is raised for all the threads to break their iteration, followed by a task-ready signal.

The last two introduced classes for this version, is “Task” and its inherited class for this Skeleton, “MapTask”. Task is a pure abstract class providing one virtual method: “execute”. The purpose of Task is to provide a common interface, in order to allow us to use TreadPool with other skeletons as well. MapTask, that implements this interface, applies the same logic, as the first version of Map, for the thread task, which is none other than to iterate the thread's input chunk, while applying the elemental function and assigning the return value on the output vector. This class encapsulates all the necessary properties for this operation.

By covering all the structural additions this version required, the final part is to analyse Map object's operator() call. The logic of this method is to create the Map-Tasks, and pass the required informations for each Task, similarly with the first version's ThreadArgument. Once this is completed, it propagates them to the ThreadPool instance through the “receiveTasks” method, and the design ensures that when “receiveTasks” is returned, the Map job is completed.

The benchmarking presented in section 4.3.1.2 for version 2, compared to version 1, indicates that the overhead is similar, and in some cases even worse. In addition to that, version 1 has a more simplified structure, and therefore between those two versions, the first one is chosen for further optimisation.

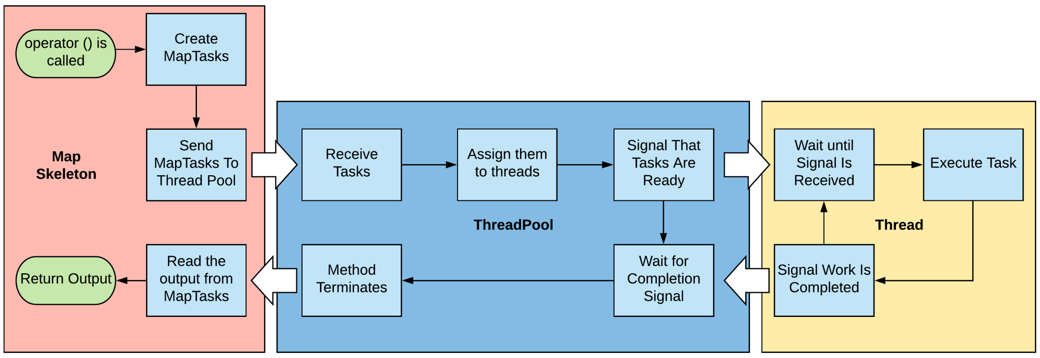


Figure 3.5: MapTask Process.

3.2.2.3 Map: An implementation with task-stealing (Version 3)

For our final adaptation of Map Skeleton, the research was focused on load balancing of the input vector. Our goal was to achieve similar performance, from total execution time perspective, whether all elements of the input vector require the same processing time, or not. To achieve this, we implement the concept of task-stealing.

The architecture remained the same as version 1, with the difference that now each thread, has the potential to process all input elements, in contrast to only the ones in its own chunk as with version 1. Of course this creates an issue of synchronisation on element access between threads, and in order to solve this, we introduced a subdivision of the input chunk, the data block.

Algorithmically, a thread starting by its own chunk, will iterate its data blocks, while ensuring that the elements of each one, have not been processed by another thread, and if not, will apply them to the elemental and will raise a flag, marking the data block completed. Once all the data blocks are marked, the thread will proceed to the neighboring chunk, and will repeat the procedure, until a full circle of the input vector is achieved. The data block size is set to 10 per cent of the chunk size.

The benefit of this algorithm, is in case where a thread is spending far greater time on processing its assigned elements, than the remaining threads do on their own, and the latter ones move on and assist to speed-up the process.

Technically, this is achieved with the use of mutexes and a flagging system. Each data block, has its own flag, indicating whether the elements in it have been processed or not. As multiple threads have the potential to modify the same flag at the same time, each input chunk is correlated with a mutex as well, in order to provide a critical section for the threads when they update those flags. Thus, when a thread reaches a data block, the related mutex is locked, the flag is checked to be up, and if so, it lowers

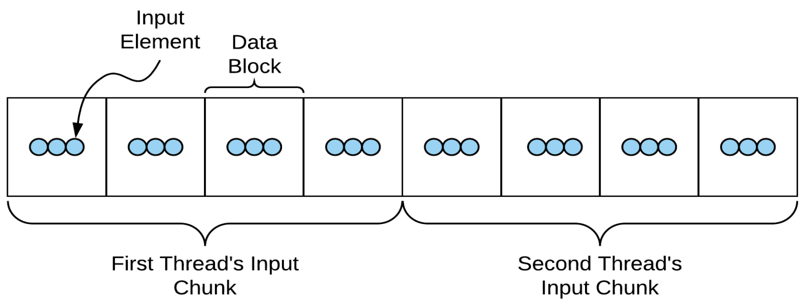


Figure 3.6: Vector of 24 Elements, handled by 2 threads with data block size of 3. This is an artificial example, the actual implementation will not create a data block of size 3 in such a case.

the flag, unlocks the mutex and accesses the elements. The algorithm is presented in figure 3.7

3.3 Reduce Skeleton

Sequentially, Reduce is an iteration of applying the combiner, on two elements, and producing one, until the whole collection is accumulated to a single value. Were the combiner is associative and commutative, then the order of combining the elements is not limited to a sequential one, an important detail, as it allows different algorithmic designs to be introduced for the task.

For a parallel implementation of Reduce, the simplest solution comes with a tree ordering as presented in figure 3.10. Potentially, if we can afford $\frac{N}{2}$ processes for the task, where N the number of input elements, then the reduction can be solved in $O(\log N)$ time [22]. While this may be true, in most situations we cannot provide enough processes to achieve this complexity, thus slightly more complicated algorithms than tree reduction exist to attain a parallel Reduce.

Our library provides a concurrent implementation of this pattern through the Reduce Skeleton, and in this chapter we will discuss our three different implementations and design choices.

3.3.1 Interface

Reduce provides the same two-phase interface as explained in section 3.1.1. The first phase includes the construction of the reduce functor, while through the second one,

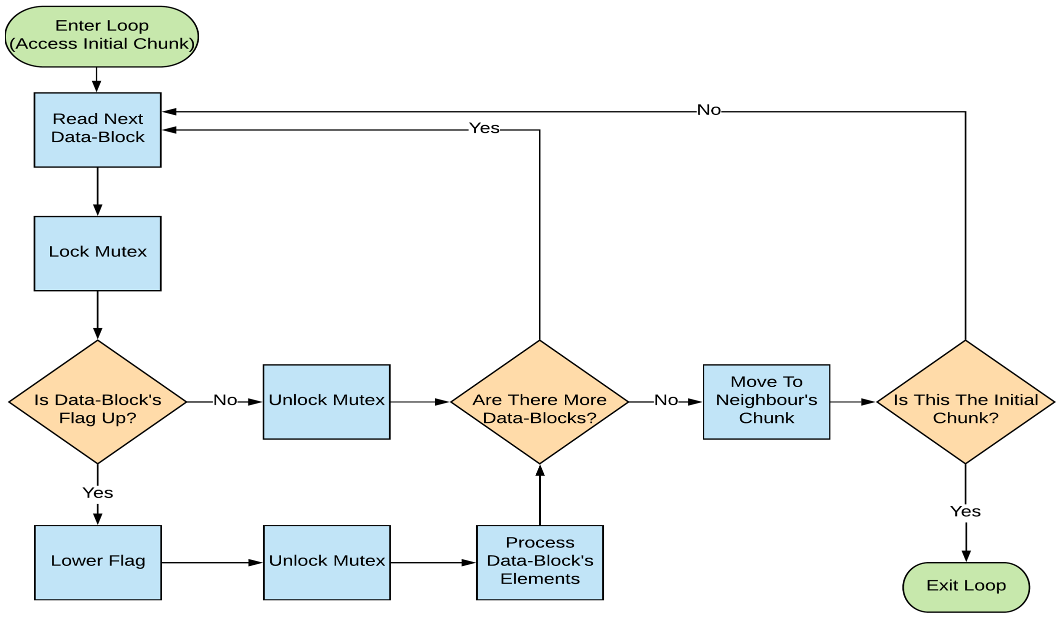

Enter Loop (Access Initial Chunk)
Read Next Data-Block
Lock Mutex
Is Data-Block's Flag Up?
No
Yes
Unlock Mutex
Lower Flag
Unlock Mutex
Process Data-Block's Elements
Are There More Data-Blocks?
Yes
No
Move To Neighbour's Chunk
Is This The Initial Chunk?
No
Yes
Exit Loop

Figure 3.7: Map Version 3, Task-stealing.

the programmer launches the actual operation, by providing the corresponding data for the cause. An example is given in listing 3.3.

3.3.1.1 Initialising a Reduce object

The creation of the Reduce functor is made through the global function “Reduce”. The programmer is demanded in this step to provide the combiner function as the first argument of the function, and discretionary, to define the number of threads to be used by this Skeleton instance as the second parameter. Contingent upon the latter argument is not used, based on our common Skeleton design, once again the available hardware parallelism is fully used by default.

3.3.1.2 Invoking the Reduce operation

Once the Reduce functor is created, its functionality is available to the user through its only publicly available method, the parenthesis operator. Analogous to the Map Skeleton, the first two arguments are mandatory. The first one is a variable, or an object, for the reduce output value, and the second one, is the vector that contains the values to be accumulated. Further arguments can be passed in this call, and those will be forwarded in each combiner invocation by the Skeleton. Once again, the return type of the functor call is inescapably void. Finally, an important note is that the input vector's element type must match the type of the output collector.

3.3.1.3 Combiner function

The programmer must follow a rigorous interface when the combiner is defined. This instructs that the first two, compulsory arguments, and the return value, must be of the same type, identical with the input vector's element type. Moreover, if extra parameters are passed during the Reduce functor call, those must be defined in the same order on the combiner's extra arguments. Furthermore, all arguments, mandatory or not, must be handled as read-only, due to reasons explained in the following paragraph. The combiner's type is illustrated in figure 3.8.

$$\text{combiner : } T\ ,T\ \ [,\ Ts...] \longrightarrow T$$

Figure 3.8: Function type of combiner, with T,Ts of any type. T is the element type, Ts is the optional variadic type of extra arguments.

In addition, the library assumes that the combiner holds both the properties of associativity and commutativity. This as explained earlier, allows the order of the combiner's appliance to exceed a sequential one. Were this convention is not followed, the Skeleton's result will be corrupted.

A special case where the element type is a pointer exists with the combiner. Analysing this with the simple case of a sequential reduction, for N elements, it is required for the fulfilment of the operation to invoke internally the combiner $N-1$ times. On the assumption that the type is a pointer, the user could either create a new pointer and assign to it the merged result, or return directly either pointer of the combiner-input parameters if for some reason the combination is not required. As this decision is hidden from the library's perspective, we presume the combiner applies the former one, which is the most frequent for a combine action. For this reason, the Skeleton has the duty, when it is safe, to deallocate the first $N-2$ combiner return values, as it is not possible for the user to do so. As a result, we forbid that the returned pointer, be the same instance as one of the arguments, since this will result potentially to a deallocated Reducer outcome value.

An alternative design could have been achieved, where in condition that the Reduce type is a pointer, for every combiner call, the return value would have been compared with its "parents". In case an equality was found, as of memory address, then the deallocation mechanism would not have taken action. However, this would have brought a significant amount of overhead, and so we preferred to make this contract with the user as a trade-off.

3.3.1.4 Usage example

```
size_t nThreads=4;
std::vector<bool> input = {true,true,false,true,true};
bool output;
auto combiner = [](bool a, bool b){ return a && b; };

auto reduce = Reduce(combiner,nThreads); //first stage
reduce(output,input); //second stage

//(result) output: false
```

Listing 3.3: An example of the two stages with Reduce Skeleton.

3.3.2 Implementation

Once again, we will analyse our Skeleton architecture with the aid of a UML class diagram presented in figure 3.9, which shows the third version of the module (Task-stealing). Commonly with our Skeletons, the implementation of Reduce is wrapped with a class (ReduceSkeleton), providing public access only to ReduceImplementation class, which has its constructor declared private as well, in order to apply our policy of instantiating a Skeleton functor, exclusively with the use of a global function, in this case “Reduce”.

As illustrated, Combiner, a private inner class, has the role of encapsulating the combiner function that the user defines, and ThreadArgument, carries the necessary information that a thread requires for the realisation of its task. ThreadArgument in this case, compared to the equivalent class of Map module, holds additional properties, as it is slightly more complicated to apply task-stealing with the Reduce procedure, as explained in section 3.3.2.3

Similarly with Map Skeleton, we have experimented with three implementation ideas. The first one (**baseline version**), is applying an enhanced tree structure to address the problem of concurrent reduction, and based on that, we have investigated towards two performance issues. Firstly to avoid thread creation overhead on each functor invocation (**Version 2**) like in the case of Map Skeleton, and secondly (**Version**

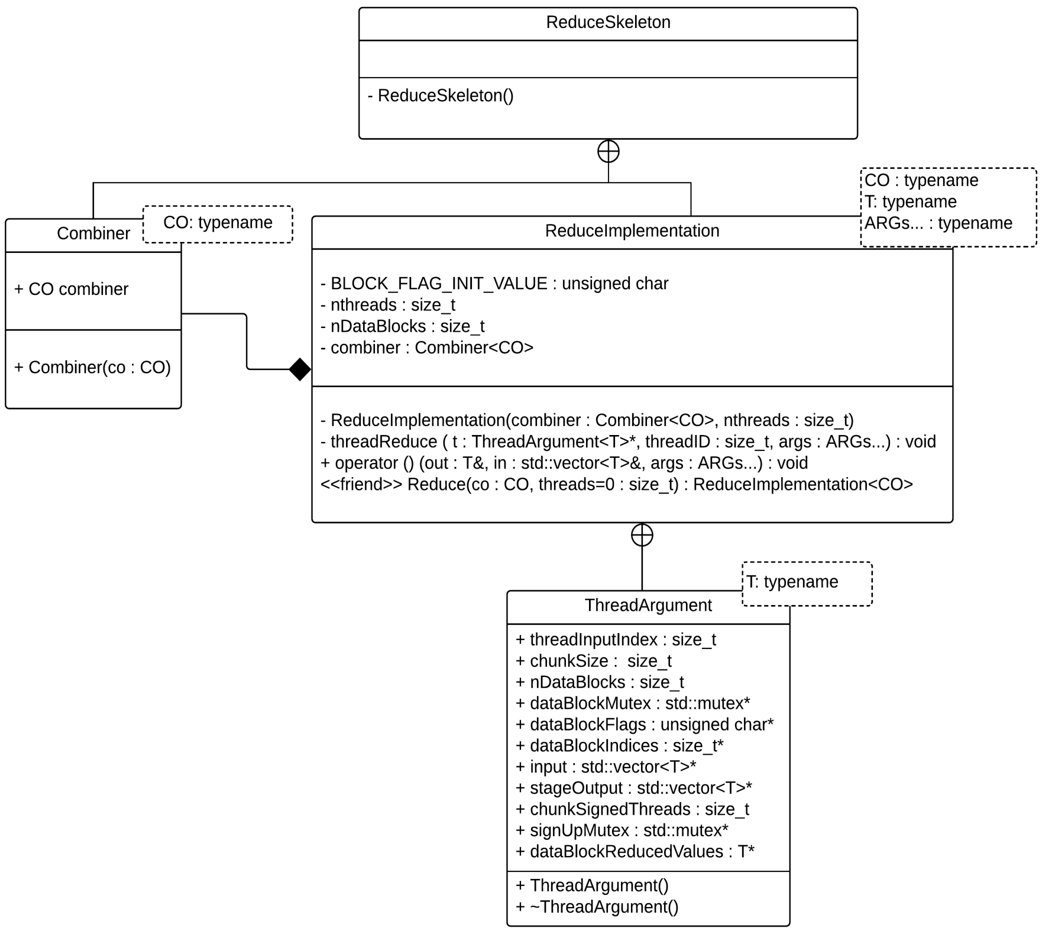

ReduceSkeleton
- ReduceSkeleton()
Combiner
CO: typename
+ CO combiner
+ Combiner(co : CO)
ReduceImplementation
CO : typename
T: typename
ARGs... : typename
- BLOCK_FLAG_INIT_VALUE : unsigned char
- nthreads : size_t
- nDataBlocks : size_t
- combiner : Combiner<CO>
- ReduceImplementation(combiner : Combiner<CO>, nthreads : size_t)
- threadReduce (t : ThreadArgument<T>*, threadID : size_t, args : ARGs...) : void
+ operator () (out : T&, in : std::vector<T>&, args : ARGs...) : void
<<friend>> Reduce(co : CO, threads=0 : size_t) : ReduceImplementation<CO>
ThreadArgument
T: typename
+ threadInputIndex : size_t
+ chunkSize : size_t
+ nDataBlocks : size_t
+ dataBlockMutex : std::mutex*
+ dataBlockFlags : unsigned char*
+ dataBlockIndices : size_t*
+ input : std::vector<T>*
+ stageOutput : std::vector<T>*
+ chunkSignedThreads : size_t
+ signUpMutex : std::mutex*
+ dataBlockReducedValues : T*
+ ThreadArgument()
+ ~ThreadArgument()

Figure 3.9: Reduce Skeleton Module (UML).

3), a dynamic load balance mechanism. Each version is presented in the following sections.

3.3.2.1 Reduce: An enhanced tree structure (Baseline version)

Algorithmically, there could be two distinct stages for the completion of a parallel Reduce. In the first stage, after the division of the input vector to the number of threads, each thread applies a reduction on the elements of its own chunk, and on the second stage, with the aid of a binary tree, we combine the values of each subdivision, resulting to the reduction outcome.

Before we move on to explain the algorithm of the two stages, two important notes have to be given on how we process an input vector with significantly small amount of elements. Firstly, to divide the input vector into chunks, we have to ensure first, that after the division, each chunk will hold at least two elements for the reduction to be possible and efficient. This means that in the case where the number of threads defined, is greater than half the input vector size, the latter number is used to delineate the number of threads to be created. Last but not least, as to handle the extreme case of an input vector with a length up to three elements, a sequential reduction is taken place inside the parenthesis operator, without the requirement of thread creation, as three elements in total, would have been handled by only one thread as explained in the first note.

Moving on, the first stage's implementation is conceptually straightforward. Given the boundary indices of the thread's chunk, we access this sub-vector sequentially in pairs, applying the combiner on them, and aggregating the outcomes. Once the chunk's reduction has been completed, we store this value to a secondary vector (stageOutput), where each thread has its own storage on it. The last vector is required in order to apply a tree-based reduction in stage two. Once all threads have completed their first-stage task, and we ensure this by applying a "synchronisation barrier" as presented in algorithm 2, we move on to the second phase.

```
Data: threadsArrived = 0
Data: mutex, condVariable
Function barrier(threadsToWaitFor):
    lock the mutex;
    threadsArrived = threadArrived + 1;
    if threadsArrived is equal to threadsToWaitFor then
        notify all threads waiting to resume;
        threadsArrived = 0;
        unlock the mutex ;
    else
        wait on condVariable and unlock the mutex;
    end
    return;
```

Algorithm 2: Reusable synchronisation barrier.

For convenience, we will explain the second phase with the use of diagram 3.10. Given 8 threads for the reduction operation, the vector produced by phase 1 (stageOutput), has a length of 8 as well. This phase's task is to reduce this vector concurrently, and this is achieved with the tree structured presented in the diagram. The implementation is based on the algorithm presented in [22].

Algorithmically, this phase is an iteration, where at the beginning of each step, the first half of the active threads, proceed to "pick" the appropriate pair and combine the values that it holds, while the other half exits the loop and is terminated, as its service is no longer required. The decision of whether a thread proceed or not to the next step, is based on the boolean value of $ThreadID < threadsInStep$, where $threadsInStep$ is initialised to total number of threads, and is halved after each step. On each step, as we move upwards the tree, the thread that handles the combination, stores the result into the location correlated to it on the "stageOutput" vector. The thread responsible for the combination is decided once again based on its ID. The first thread (ID=1) handles the first pair, the second thread the second pair, and so on. Thus, at the end of this loop, the total synthesis of the input elements, is retrieved by reading the first element of the aforementioned vector. Notably, a synchronisation barrier is used at the end of each iteration, to ensure that the values of the vector are ready for the following step.

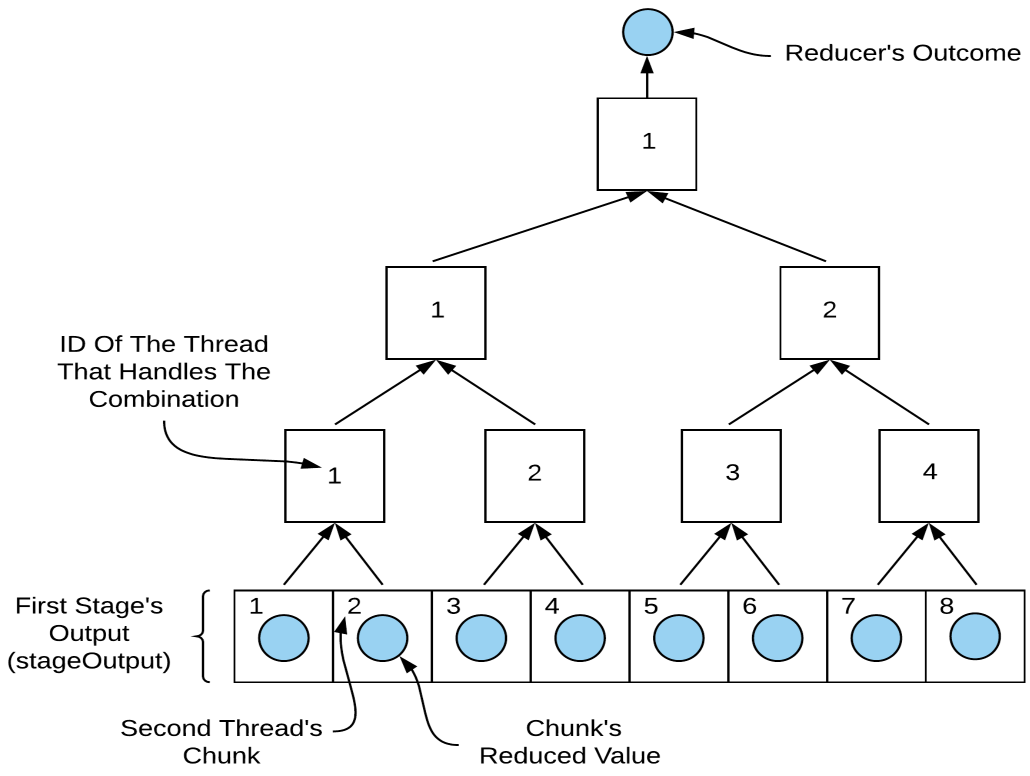

Reducer's Outcome
1
1
2
ID Of The Thread That Handles The Combination
1
2
3
4
First Stage's Output (stageOutput)
1
2
3
4
5
6
7
8
Second Thread's Chunk
Chunk's Reduced Value

Figure 3.10: Reducer's second stage - the tree structure with 8 threads available.

Technically, there is an important issue to be taken care of on the second stage, and this issue rises with the condition that the number of threads initially, or in the previous iteration step, is odd. This unavoidably, must be checked in each iteration, and only the first thread is assigned to handle the reduction of the “carry” element, in each occurrence, with its own. Finally, in order to simplify the calculation of the “carry” index, we have chosen the approach of using the first half of the active threads on the next iteration, and not to follow an implementation that would make use of modulo to decide which threads will proceed. Due to our algorithm’s ability to handle the “carry” element as well, the appellation of “Enhanced Tree Structure” is given.

3.3.2.2 Reduce: An implementation with a thread pool (Version 2)

Equivalently with Map Skeleton, Reduce makes use of the ThreadPool class, as explained in section 3.2.2.2, in order to remove thread creation overhead on each functor call. The difference comes with the inheritance of the abstract class Task, where in the case, is through “ReduceTask”. Likewise in our previous case, the thread’s logic is positioned in Task’s “execute” method, and it is identical with the Reduce’s first version.

Similarly with Map, Reduce’s thread pool is created on the first stage (the functor’s initialisation), and as the same practice applies here as well, the threads created, are idle until the second stage is triggered by the programmer. This version’s performance compared to the first one, has the same trend as the equivalent comparison of the Map Skeleton. Further discussion on this is given in section 4.3.2.2, accompanied with the benchmark analysis.

3.3.2.3 Reduce: An implementation with task-stealing (Version 3)

This section describes the algorithm used on top of our Reduce baseline version, in order to apply a dynamic load balance between each thread’s procedure. This has been achieved once again with the concept of “Data blocks”, although, Reduce demands a slightly more complicated algorithmic structure to deliver the required behaviour. Based on what has been already been mentioned in section 3.2.2.3 for Data blocks comporting, here we will discuss only the additional features that this implementation requires.

To begin with, task-stealing is applied to the first phase, where each thread’s chunk is divided further into data blocks, as shown in figure 3.6. Our goal is to produce the “stageOutput” vector, through a dynamic load-balance technique, which will be fed to the second phase, in order to compute the total reduced value. After this, as the nature of the enhanced tree structure achieves maximum concurrency for the cause, the second phase is not modified in this version.

Conceptually, with the introduction of data-blocks on the first phase, we face exactly the same problem as our initial one - to combine in parallel a number of values. This is based on the fact that now, once all data blocks in a chunk have been reduced, a combination of these data-block values has to be made, in order to compute the chunk's reduced value. Thus, a "mini reduce" has to be applied on each chunk, with the challenge to do so, by keeping all the threads productive, which means to reduce the data-block values per chunk, without placing threads in an idle condition.

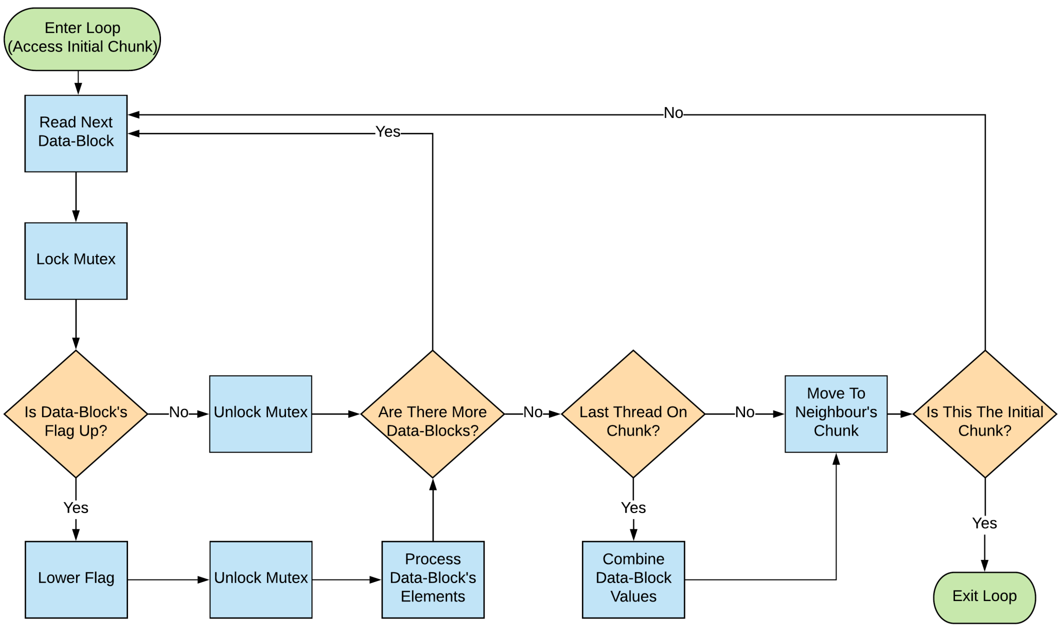

Enter Loop (Access Initial Chunk)
Read Next Data-Block
Lock Mutex
Is Data-Block's Flag Up?
No
Unlock Mutex
Are There More Data-Blocks?
No
Last Thread On Chunk?
No
Move To Neighbour's Chunk
Is This The Initial Chunk?
Yes
No
Yes
Lower Flag
Unlock Mutex
Process Data-Block's Elements
Yes
Combine Data-Block Values
Yes
Exit Loop

Figure 3.11: Reduce Version 3, Phase 1, Task-stealing.

To solve this, we used the algorithm presented in figure 3.11, where the main idea is that the last working thread in a chunk, will combine all final data-block values as well in a sequential way. A question may rise here, for why we did not use the binary tree in this situation, and the answer is that a tree implementation could not avoid a crucial state. That state is that it is required to use more than one thread to benefit from a tree reduction, which implies that some threads, would have to "wait" for other ones before the tree-reduction is started. In contrast, by applying our strategy, we allow all the other threads that worked on a chunk before the last one, to move on to a different chunk and speed-up the whole process.

Finally, once the data blocks of each chunk have been accumulated, we produce the "stageOutput" vector, and phase two begins, producing at the end the final result.

3.4 MapReduce Skeleton

Thinking in a functional style, it is often that the combination of Map and Reduce Skeletons is necessitated. A Map operation, followed by a Reduce on the former's outcome, is surprisingly common among many programming tasks. The most frequent task could be with the case where given a collection of items, the transformation of that to a second one is required initially, before we move on to the aggregation of the latter that will produce the desired outcome.

For example, suppose we have a collection of objects that represent students, and we want to answer the query of how many of them are twenty-four years old. One way to solve this, could be by iterating the student collection, and for each student that matches the criteria, on a secondary collection of integers, we simply set the correlated position to 1 (Map). After that, we sum (Reduce) all the values of the second collection, thus obtaining our answer.

In addition to that, it is also helpful if we could create groups, or keys, on the Map output, and cumulate per key in the Reduce phase. This pattern, with the grouping feature, is what we have designed and implemented in parallel as MapReduce Skeleton, influenced by the specifications of Google's equivalent algorithmic abstraction [11].

3.4.1 Interface

Generally, MapReduce receives a vector of elements, and produces a vector of key-value pairs, where for the latter vector, an alternative option of key-vector(values) pairs is provided to the user for further programming flexibility.

By following our common Skeleton interface, MapReduce is handled by creating the functor object, and following that, the user provides the data for the operation. Furthermore, MapReduce, as a combination of two Skeletons, requires on instantiation phase, two functions that define the operations respectively, plus a hash-function as explained in the following sections.

As the output of the first function is the input of the second one, and since our Skeleton supports reduction per key, the user has to follow a specific interface on their definition, which has been altered compared to the ones where the Skeletons are independent. To avoid any confusion, we reference on them as "mapper" (elemental), and "reducer" (combiner).

Based on the type restrictions presented in [11] for MapReduce, we decided to apply similar rules to the programmer, during his declaration of the aforementioned functions. Conceptually, the mapper receives an element, and produces a list of intermediate key-value pairs, while the reducer accepts a key and a list with all the correlated values to that key, which then reduces to a second list of values. The latest list could be of any size (including zero). The exact type restrictions for the mapper and the reducer are presented in figure 3.12.

3.4.1.1 Initialising a MapReduce object

In order to create a MapReduce function-object, the developer has to call "MapReduce" function, by passing three function for the first three arguments, and optionally as the last parameter the number of threads to be used in this instance. The first function is the mapper, the second one the reducer, and the last one is a hash function targeting the intermediate key variables that the mapper produces.

Moreover, the hash function is essential for the Skeleton's operation, as it is the only tool used for the distinguishment of the keys produced by the mapper, in order so their values to be grouped and propagated to the reducer. Further details are given in the implementation section of this chapter.

3.4.1.2 Invoking the MapReduce operation

The returned object of "MapReduce" function, is used in order to start the Skeleton's procedure. The user is obligated to provide two collections, for the first two arguments of the parenthesis operator, and based on his mapper and reducer implementations, the equivalent extra arguments.

The first argument is used by the Skeleton to store its output, where the second one is the Skeleton's input. There are two (overloaded) parenthesis operator methods provided by the Skeleton, and their difference lies on the output's collector type.

As explained earlier, the reducer might produce a list with zero, one, or more values per key. Thus, in case the programmer's decision is that the reducer will always return a list with up to one element, for convenience, the collection of the results can be made through a vector of <key,value>pairs. Otherwise, the collector would have to be a vector of <key,vector(value)>pairs. The exact type choices for the output collector is presented in figure 3.13, under MapReduceImplementation - operator().

Concerning the extra arguments defined for mapper and reducer, a technical issue rises, thus a contract has to be made with the programmer. The contract is that in case of extra arguments, those arguments must be received in the same order, both by the mapper and reducer. The reason is that as the extra arguments are handled through a C++ variadic template, it is impossible for the language (with the available compiler version) to separate those arguments, in order to allow us to "split" between the ones used by the mapper, and the ones used by the reducer. A solution to this could have been achieved, by requesting from the user to wrap in a std::tuple the extra arguments for each case, and then to provide those to the functor call, but we decided that this approach introduces further "lines-of-code" for the user, thus the convention of defining all the extra arguments both for mapper and reducer is made.

3.4.1.3 Mapper, Reducer, and Hash functions

The function types of mapper, reducer, and hasher are illustrated in figure 3.12. Similarly with the case of Reduce Skeleton's combiner function, the developer has to treat all arguments for both functions, including any extra parameters, as read-only. In addition, returned variables of mapper and reducer, that instantiate a pointer type, have to be of a new instance, due to the fact that MapReduce supports memory deallocation for the variables that require to inside its scope.

$$\begin{array}{rll}
\text{mapper}: & T_1\ [,\ \text{Ts...}] & \longrightarrow \ \text{std::list< std::pair< } T_2\text{ , }T_3\text{ > >} \\
\text{reducer}: & T_2\text{ , std::list< }T_3\text{ > }[,\ \text{Ts...}] & \longrightarrow \ \text{std::list< }T_3\text{ >} \\
\text{hasher}: & T_2 & \longrightarrow \ \text{std::size\_t}
\end{array}$$

Figure 3.12: Function types of mapper, reducer, and hasher, with $T_1, T_2, T_3, Ts...$ of any type. T_1 is the Input element type, T_2 is the Key type, T_3 is the Value type, and $Ts...$ is the optional variadic type of extra arguments. Notice that the variadic type is exactly the same for both mapper and reducer.

Analysing this further, the deallocation mechanism is essential in case the returned pair of the mapper encapsulates pointer types. Following the same arguments as presented in section 3.3.1.3, the values passed to the reducer, are deallocated directly after the reducer's call to avoid memory leaking. Similarly, as in case of pointer type, the action is applied for keys produced by the mapper, since the programmer could allocate more than one instance for the same key (same as defined by the hash function), and will receive only one instance of them as an argument to the reducer.

An alternative option, and only for the values - not for the keys, is to apply a comparison mechanism after each reducer invocation, to recognise whether the user has returned the same instance as one of its arguments or not, and to trigger memory deallocation accordingly. Likewise combiner's case, as this approach introduces an overhead, it has been rejected.

Finally, the programmer has to be extremely careful with the definition of the hash function, as it is the only way of our implementation for identifying a key that is produced by the mapper. Thus, the hash function takes as parameter a key, with the type as defined by the user, and returns specifically a value of type "std::size_t". A hash function that produces the same value for different keys, could potentially corrupt the effectiveness of the Skeleton.

3.4.1.4 Usage example

```
std::vector<int> input = {24,24,24,28,28,25};
std::vector<std::pair<int,int>> output;

auto mapper = [](int age) {
  std::list<std::pair<int, int>> res = {
   std::make_pair(age,1) };
  return res;
};

auto reducer = [](int key, std::list<int> values){
  std::list<int> sum = { (int)values.size() };
  return sum;
};

auto hasher = [](int key){return std::hash<int>{}(key);};

auto mapReduce = MapReduce(mapper,reducer,hasher);//1st stage
mapReduce(output,input);//2nd stage

//(result) output: {(25,1),(28,2),(24,3)}
```

Listing 3.4: An example of the two stages with MapReduce Skeleton.

3.4.2 Implementation

The structure of MapReduce module is analogous with the ones we have seen in the previous sections. It consists of, a wrapper class (MapReduceSkeleton) which ensures that the programmer will have access only to the necessary classes and methods for handling the Skeleton, three classes (Mapper,Reducer,Hasher) that encapsulate the user-defined behaviour of MapReduce, one class (ThreadArgument) that provides the essential information a thread needs during its execution, and finally, the core class of the module (MapReduceImplementation) that implements the algorithm of the Skeleton. The module's class UML diagram is shown in figure 3.13.

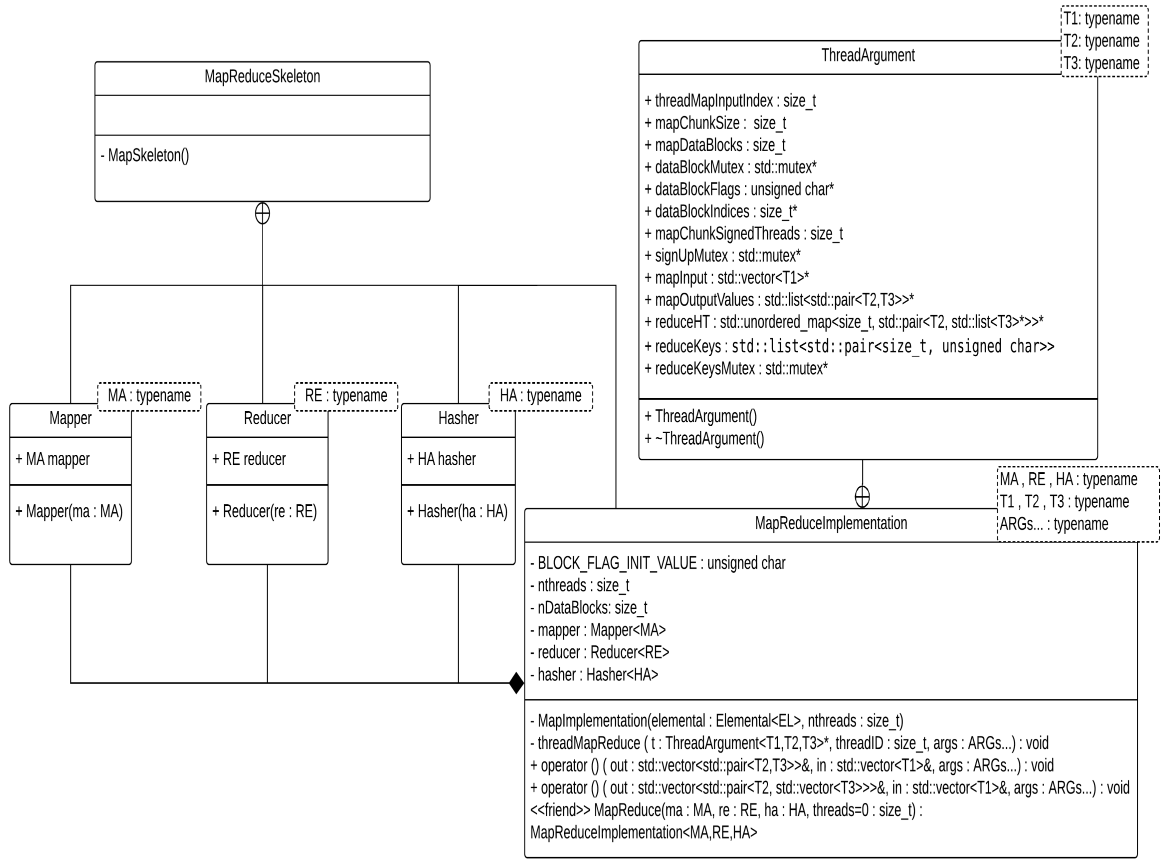

T1: typename
T2: typename
T3: typename
ThreadArgument
+ threadMapInputIndex : size_t
+ mapChunkSize : size_t
+ mapDataBlocks : size_t
+ dataBlockMutex : std::mutex*
+ dataBlockFlags : unsigned char*
+ dataBlockIndices : size_t*
+ mapChunkSignedThreads : size_t
+ signUpMutex : std::mutex*
+ mapInput : std::vector<T1>*
+ mapOutputValues : std::list<std::pair<T2,T3>>*
+ reduceHT : std::unordered_map<size_t, std::pair<T2, std::list<T3>*>>*
+ reduceKeys : std::list<std::pair<size_t, unsigned char>>
+ reduceKeysMutex : std::mutex*
+ ThreadArgument()
+ ~ThreadArgument()
MapReduceSkeleton
- MapSkeleton()
MA : typename
RE : typename
HA : typename
Mapper
+ MA mapper
+ Mapper(ma : MA)
Reducer
+ RE reducer
+ Reducer(re : RE)
Hasher
+ HA hasher
+ Hasher(ha : HA)
MA , RE , HA : typename
T1 , T2 , T3 : typename
ARGs... : typename
MapReduceImplementation
- BLOCK_FLAG_INIT_VALUE : unsigned char
- nthreads : size_t
- nDataBlocks: size_t
- mapper : Mapper<MA>
- reducer : Reducer<RE>
- hasher : Hasher<HA>
- MapImplementation(elemental : Elemental<EL>, nthreads : size_t)
- threadMapReduce (t : ThreadArgument<T1,T2,T3>*, threadID : size_t, args : ARGs...) : void
+ operator () (out : std::vector<std::pair<T2,T3>>&, in : std::vector<T1>&, args : ARGs...) : void
+ operator () (out : std::vector<std::pair<T2, std::vector<T3>>>&, in : std::vector<T1>&, args : ARGs...) : void
<<friend>> MapReduce(ma : MA, re : RE, ha : HA, threads=0 : size_t) :
MapReduceImplementation<MA,RE,HA>

Figure 3.13: MapReduce Skeleton Module (UML).

In the following sections we present three different implementations of MapReduce. The first version (**Baseline Version**), handles the problem through three distinct stages: the mapper stage, the intermediate stage, and the reducer stage. The second version (**Version 2**), optimises the intermediate stage, and finally the third version (**Version 3**), built on top of the second one, applies task-stealing to the mapper and reducer stages.

3.4.2.1 MapReduce: The initial approach (Baseline version)

Initially, once the parenthesis operator has been called by the programmer, a pre-process is made to create each thread's parameters for their work, which includes the sub-division of the input vector to relatively equal chunks for each thread. Then the concurrent process is launched, and the algorithm of this is described in the following paragraphs.

Based on what has been discussed in the previous paragraphs, MapReduce receives a vector of elements, applies the mapper to each of them, collects a list of key-value pairs from every mapper invocation, groups the values per key, and forwards those groups along with the correlated key to the reducer, which produces the final result. Our approach splits this behaviour into three stages.

We will begin by analysing the first stage (mapper stage), which assigns each thread to iterate its chunk, and to apply the mapper on each element, while storing the returned values. The returned values are stored by appending them to a list of pairs, separate for each thread, and at the end of this stage, only one thread is responsible for the total merge of these lists. The merging is handled sequentially, as list concatenation is of constant time, equal to the number of threads for this case. After this point, the mapper stage is completed, and the merged list holding all key-value pairs, is passed to the second stage, where it is processed in order for the required value grouping, per key, to be achieved.

In the second stage (intermediate stage), each thread iterates the merged list that was produced earlier, with the purpose to group the values per key, and following that, one thread is assigned to equally distribute those keys among the threads, based on the number of values each key holds. The way this is achieved is firstly with the help of the hash-function provided by the user, and secondly with a distinct for each thread hash table.

During the iteration of the merged list, each thread invokes the hasher with the key, and if the hash value is related to the thread, then the key-value pair is added, or appended, to the thread's hash table. The relation between the thread and the hash value is determined by the boolean value of :

$$H \ modulo \ T == t_i$$

where H is the key's hash value, T the total number of threads, and t_i the thread's ID.

A technical note has to be made here, and it concerns the structure of the hash table, which uses the hash value as its own key and not the key produced by the mapper (The exact type of the hash table is shown in the Module's UML, under ThreadArgument, reduceHT).

Once all values have been assorted, one thread makes the heuristic and assigns keys to each thread, in order to achieve an equal distribution of work among the threads during the reducer stage. Conceptually, each thread is given a work-capacity amount W_c, based on the number of threads in use, and each key has a work-demand amount

W_d, based on the number of values it holds. Our goal is to achieve the utmost use of each thread's work-capacity after the distribution of keys.

$$W_c = \frac{1}{T}$$
$$W_d = \frac{V_k}{V_t}$$

Where T the number of threads, V_k the key's number of values, V_t the total number of values.

In detail, the algorithm for a balanced distribution of keys is described through three steps. Firstly, the thread responsible for the task, iterates each thread's hash table, and creates a record that holds for each key the following information: $[t_i,H,W_d]$, with t_i the id of the thread whose hash table stores the key, H the key's hash value, and W_d the key's work-demand. The second step is to sort this record by work-demand in descending order, and the third step is to distribute the workload among the threads. The distribution is achieved by iterating and assigning keys to a thread, in the order they are stored in the record, while accumulating the work-demand that has been distributed to the thread. Once the work-demand given is greater than the thread's work-capacity, the process continues with the next thread (relative to thread-id) until there are no more keys.

In addition, a final check before we proceed to the last stage has to be made, in order to ensure that there are no threads without keys assigned to them, while another thread exceeds the work-capacity. This is handled by iterating backwards the threads (based on thread-id), and in case a thread with more than one key and a workload greater than the work-capacity has been found, one of its keys is given to the first thread. The iteration then takes another step, until there are no more threads, or the first thread has overcame the work-capacity. The algorithm is shown in figure 3.14.

The final stage (reducer stage) has the threads iterating their assigned keys and applying them, with their relative values, on the reducer, and once all the threads have completed this point, the thread-function is terminated. The execution resumes in the parenthesis operator, and the reducer's return values along with their keys are gathered and placed on the output collection that the user has provided.

3.4.2.2 MapReduce: an improvement for the intermediate stage (Version 2)

An issue rises with the intermediate stage of our baseline version, and it concerns the step where all threads have to iterate the "merged-list" in order to assort the values into

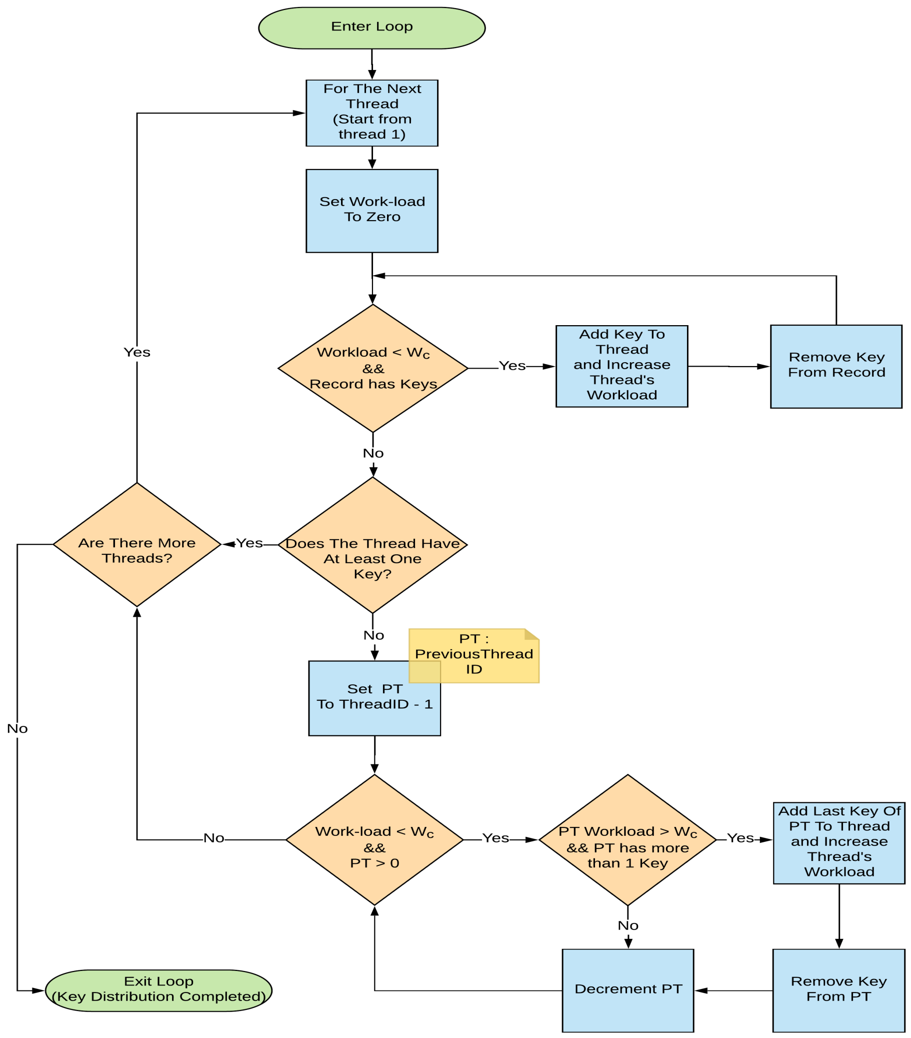

Enter Loop
For The Next Thread (Start from thread 1)
Set Work-load To Zero
Workload < Wc && Record has Keys
Yes
Add Key To Thread and Increase Thread's Workload
Remove Key From Record
No
Does The Thread Have At Least One Key?
Yes
Are There More Threads?
Yes
No
PT : PreviousThread ID
Set PT To ThreadID - 1
Work-load < Wc && PT > 0
No
Yes
PT Workload > Wc && PT has more than 1 Key
Yes
Add Last Key Of PT To Thread and Increase Thread's Workload
Remove Key From PT
Decrement PT
No
Exit Loop (Key Distribution Completed)

Figure 3.14: MapReduce Version 1 - Balanced Distribution of Keys.

keys. This step is inevitable with the algorithm presented in the previous section, and has a complexity of $O(N)$, with N the number of values. Moreover, this complexity is unrelated to the number of threads provided, which prevents our Skeleton to achieve total parallelism. Thus, effort has been given for a better implementation that will make an efficient use of the available concurrency, and this implementation is presented in this section.

The problem rises with the design choice that sets each thread to store the mapper's returned pair, to a list of pairs, and not to a hash table directly. This narrows the design space, allowing as a monadic solution to the problem of grouping the values per key, to the one described in the baseline version.

For this reason, we have altered our algorithm before that point, having each thread to insert, or append, the mapper's pair to its own hash table, and after the whole input vector has been processed by all the threads, a parallel-merge of each thread's hash table takes place, providing us the same intermediate state, which is to have the values grouped by key.

The combination of the hash tables is applied exactly with the same technique used in the case of Reduce-Skeleton's chunk aggregation - the enhanced concurrent tree reduction (see section 3.3.2.1). In this case, each tree node represents a hash table, and their combination is achieved by inserting to the left hast table the keys and values of the right one (in case those keys are not present to the destination), or by appending the values of the right hash table to the values of the left one (in case their key exists in both nodes). In addition, the same "carry" mechanism is applied here as well, taking care of odd number of nodes in any tree-level. Similarly with the equivalent procedure in the Reduce Skeleton, the result is stored to the "threadArgument" of the first thread.

After the fusion of the hash tables has been completed, the operation remains the same as the baseline version, where one thread makes the heuristic and assigns an equal amount of workload to each thread for the reducer stage, which follows right after.

3.4.2.3 MapReduce: An implementation with task-stealing (Version 3)

Once again, the issue of an unbalanced workload has to be settled, only that now it concerns both the mapper and reducer stages. Thus, we applied the same algorithm presented in the previous Skeletons' implementations, which addresses the dynamic allocation of work among threads through the logic of "Data-Blocks".

The mapper stage, shares the same complication as Reduce Skeleton, when it comes to applying the concept of data blocks. The complication is that only by the time all chunk's data blocks have been processed, the chunk's hash table can be created, otherwise further thread synchronisation issues arises. This shares the same pattern with task-stealing in a Reduce operation, thus we followed the same solution which defines that the last thread working on the chunk, will have to create the hash table as well.

Furthermore, in order to apply the data block technique to the mapper stage, we have to increase the Skeleton's memory consumption, as we temporary store the mapper's returned pairs, in order for the last thread responsible for the chunk, to have access to all chunk's pairs and create the hash table. The algorithm is illustrated in figure 3.15.

The intermediate stage, combines the hash tables of each thread in the same way as the second version of this Skeleton, although, the balanced load-distribution part of this stage has been removed. This is due to the fact that it is no longer required as we

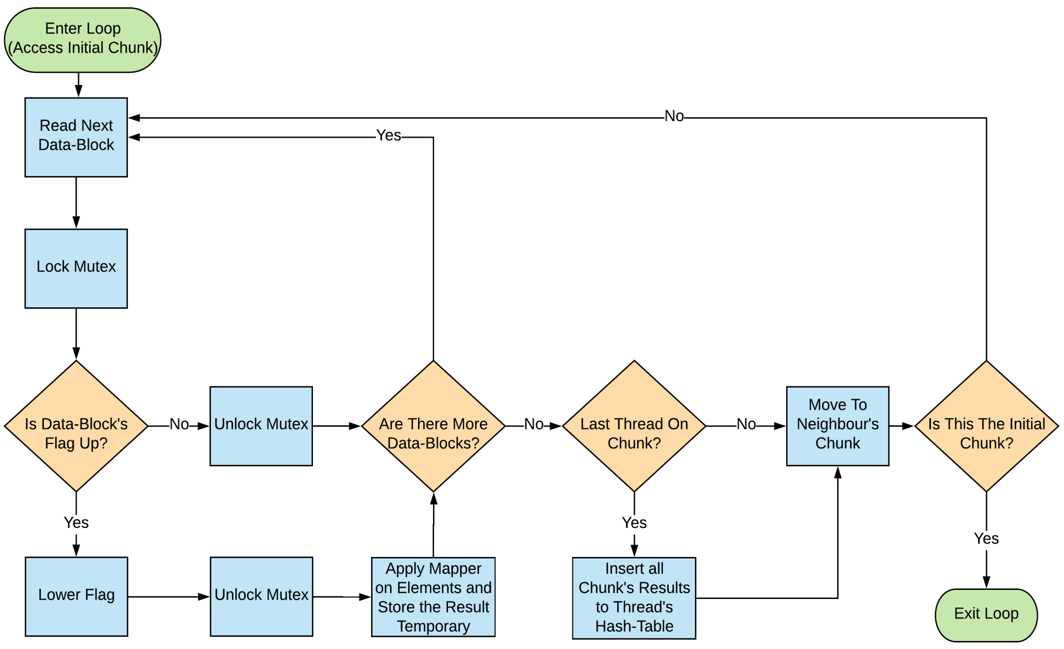

Enter Loop (Access Initial Chunk)
Read Next Data-Block
No
Yes
Lock Mutex
Is Data-Block's Flag Up?
No
Unlock Mutex
Are There More Data-Blocks?
No
Last Thread On Chunk?
No
Move To Neighbour's Chunk
Is This The Initial Chunk?
Yes
Yes
Yes
Lower Flag
Unlock Mutex
Apply Mapper on Elements and Store the Result Temporary
Insert all Chunk's Results to Thread's Hash-Table
Exit Loop

Figure 3.15: MapReduce Version 3 - Dynamic Load Balance on Mapper Stage.

apply dynamic key allocation in the reducers stage.

The dynamic key allocation is achieved by applying a similar technique as with data blocks. For each key in the combined hash table, a flag is correlated to it, indicating whether the key has been processed or not. Then in the reducer stage, the threads simply iterate the combined hash table, and they process the first key that its flag is up, while lower the flag afterwards. Technically, through the use of critical sections (with std::mutex), the implementation ensures that the same key will not be processed by two different threads.

3.5 Scan Skeleton

In sequential programming, Scan is expressed as an iteration over the targeted collection, where in each step, an element and its previous one are applied to the successor, and the result is stored in the position of the former.

A parallel implementation of Scan, by looking its sequential algorithm, is enigmatic. This is due to the fact that for computing the result in a given collection, we notice that each outcome element depends on its precedent. Moreover, if we want to apply the Scan operation on a collection, in a non-sequential order, then it is necessary that the successor holds the properties of associativity and commutativity. These two properties are important for a parallel implementation of Scan, which compared to its sibling pattern - Reduce, is slightly more baroque.

In this section we present the interface provided by our inclusive Scan Skeleton, and our design for a parallel implementation of Scan, which is based on previous research [22].

3.5.1 Interface

Following our library design, Scan Skeleton is handled through two instructions. The first one being the initialisation of a Scan functor, and the second one the execution of its algorithm over an input collection via its parenthesis operator. An example is given in listing 3.5.

3.5.1.1 Initialising a Scan object

The initialisation of a Scan functor is compassed by invoking the "Scan" function. In this step, the user is requested to define as the first argument the successor function, and optionally as a second argument, the number of threads that is preferred to be used by the instance. Based on our library arrangement, were the later argument is absent, the Skeleton will use the maximum hardware concurrency available.

3.5.1.2 Invoking the Scan operation

The second step for applying a parallel Scan, is to prompt the functor by providing at least two arguments. The former being a vector that will receive the operation's outcome, and the latter being a vector holding the elements under process. Concerning the content types of those two vectors, as Scan operation itself does not allow them to be different, we expect from the programmer that for both of them, the type will be the same. Otherwise, a compiler error will rise.

Succeeding the mandatory arguments, a definition of extra parameters that will be passed directly to each successor call are allowed.

3.5.1.3 Successor function

The restrictions applied with the programmer's definition of the successor, consist firstly with the rule that the function must accept at least two arguments with identical type, and the result produced must have the same type as those first two arguments. Secondly, this common type must match the type of the input-data collection the user will provide, and finally, in existence of additional arguments, those arguments once more, must be retrieved in the same order as they are arranged in the functor call. The successor's function type is shown in figure 3.16.

$$\text{successor} : \text{T} , \text{T} \ [, \text{Ts...}] \longrightarrow \text{T}$$

Figure 3.16: Function type of successor, with $T, Ts...$ of any type. T is the element type, $Ts...$ is the optional variadic type of extra arguments.

Alike with the case of the combiner in Reduce Skeleton, the successor must be associative and commutative to allow the Scan Skeleton to produce valid results. This covenant is expected to be followed by the programmer, thus the library does not verify its compliance.

The final convention concerns the fact that in case the input vector's type is a pointer, the returned value of the successor function needs to be a new instance, created inside the successor. Similarly with the previous discussion in section 3.3.1.3, this permits the Scan Skeleton's memory deallocation mechanism to act without defiling the final result.

3.5.1.4 Usage example

```cpp
std::vector<int> input = {0,1,1,2,3,5,8};
std::vector<int> output;
auto successor = [](int a,int b){ return a+b; };

//The two stages of Scan, expressed as a one-line statement
Scan(successor)(output,input);

//(result) output: {0,1,2,4,7,12,20}
```

Listing 3.5: An example of the two stages with Scan Skeleton.

3.5.2 Implementation

The module that bundles our Scan Skeleton, is comprised of a wrapper class and its shielded structures, as it inherits our abstract Skeleton design. Subsequently to our first two implementations, an additional structure, termed "ScanTreeNode", is introduced for assisting our core class that effectuates our algorithms - "ScanImplementation". Finally, the basic auxiliary classes "ThreadArgument" and "Successor" are incorporated in all versions for the cause. The final version of the module, is presented in the UML diagram in figure 3.17.

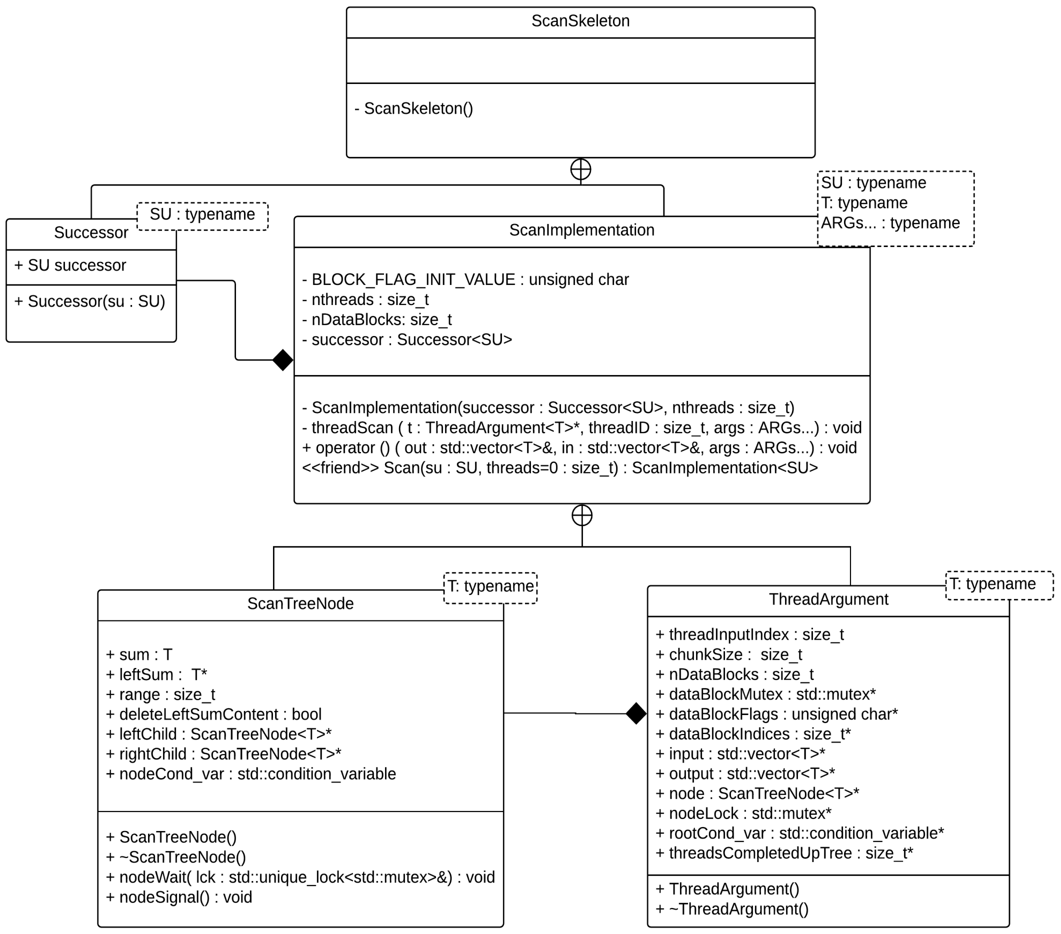

ScanSkeleton
- ScanSkeleton()
SU : typename
T: typename
ARGs... : typename
Successor
SU : typename
+ SU successor
+ Successor(su : SU)
ScanImplementation
- BLOCK_FLAG_INIT_VALUE : unsigned char
- nthreads : size_t
- nDataBlocks: size_t
- successor : Successor<SU>
- ScanImplementation(successor : Successor<SU>, nthreads : size_t)
- threadScan (t : ThreadArgument<T>*, threadID : size_t, args : ARGs...) : void
+ operator () (out : std::vector<T>&, in : std::vector<T>&, args : ARGs...) : void
<<friend>> Scan(su : SU, threads=0 : size_t) : ScanImplementation<SU>
T: typename
ScanTreeNode
+ sum : T
+ leftSum : T*
+ range : size_t
+ deleteLeftSumContent : bool
+ leftChild : ScanTreeNode<T>*
+ rightChild : ScanTreeNode<T>*
+ nodeCond_var : std::condition_variable
+ ScanTreeNode()
+ ~ScanTreeNode()
+ nodeWait(lck : std::unique_lock<std::mutex>&) : void
+ nodeSignal() : void
T: typename
ThreadArgument
+ threadInputIndex : size_t
+ chunkSize : size_t
+ nDataBlocks : size_t
+ dataBlockMutex : std::mutex*
+ dataBlockFlags : unsigned char*
+ dataBlockIndices : size_t*
+ input : std::vector<T>*
+ output : std::vector<T>*
+ node : ScanTreeNode<T>*
+ nodeLock : std::mutex*
+ rootCond_var : std::condition_variable*
+ threadsCompletedUpTree : size_t*
+ ThreadArgument()
+ ~ThreadArgument()

Figure 3.17: Scan Skeleton Module (UML).

The sections that follow, describe our four implementation for a Scan Skeleton. Initially (**Baseline version**), a basic three-phase implementation is shown. With this version, after the division of the input vector to thread-chunks, a scan operation on each chunk, followed by a partial prefix-sum that includes only the last values of each chunk, is concluded by applying the successor function to each value of a chunk and its precedent's last element. Secondly (**Version 2**), a task-stealing enhancement for the first and third phase of the initial version is demonstrated, which did not meet our expectations, and concessed its place to our third version (**Version 3**), that optimises the second phase of our original version by applying a special binary tree algorithm. Finally, our last version (**Version 4**), applies task-stealing once again, but in this case only to the third phase of our third version.

3.5.2.1 Scan: A three-phase adaptation (Baseline version)

A preparation stage is essential once the functor is invoked. This stage concerns the creation of the threads' parameters, carried by instances of "ThreadArgument" class, and the important part to note is that similarly with the other Skeletons, those instances carry the essential information that defines each thread's corresponding section in the input vector - the chunk. In addition, in the same way as Reduce Skeleton, a chunk is required to possess at least two elements, in order for a prefix-sum operation inside the chunk to be substantial. Otherwise, the number of threads to be used, are reduced to half of the input vector's size. Equally important for discussion, and for the same reason, the boundary case of an input vector with up to three elements in length, is handled sequentially inside the parenthesis operator to avoid unnecessary overhead for the case. Once the decision is made, and all the threads have been activated, a three-phase algorithm is executed. The algorithm implemented is based on the work of the author in [22].

The first phase assigns each thread to perform a sequential scan on its chunk, while storing the result to the output vector, and after this sub-task, a synchronisation barrier ensures that each thread has completed its assignment before we proceed to the second phase. The following phase concerns only one thread, selected by its ID, and the thread's responsibility is to apply a partial scan that includes only the final elements of each chunk, with the current values of the output vector. After a second use of the synchronisation barrier, once the this phase is completed, all the threads except the first one, which has its chunk ready for submission to the user, enter the third phase. On this last phase of this short algorithm, each - still active - thread applies the successor to the previous chunk's last element, and every element of its own chunk, up to the penultimate element. After the completion of those three phases, the Scan operation is completed and the algorithm is illustrated in figure 3.18.

For convenience, the first phase, as it applies a scan operation on each chunk, will be referenced as "**scan phase**". The second one remains the "**second phase**", and the third phase, as it can be seen as a map operation on the elements of a chunk, that includes as an extra variable the last element of the precedent chunk, will be captioned as "**map phase**".

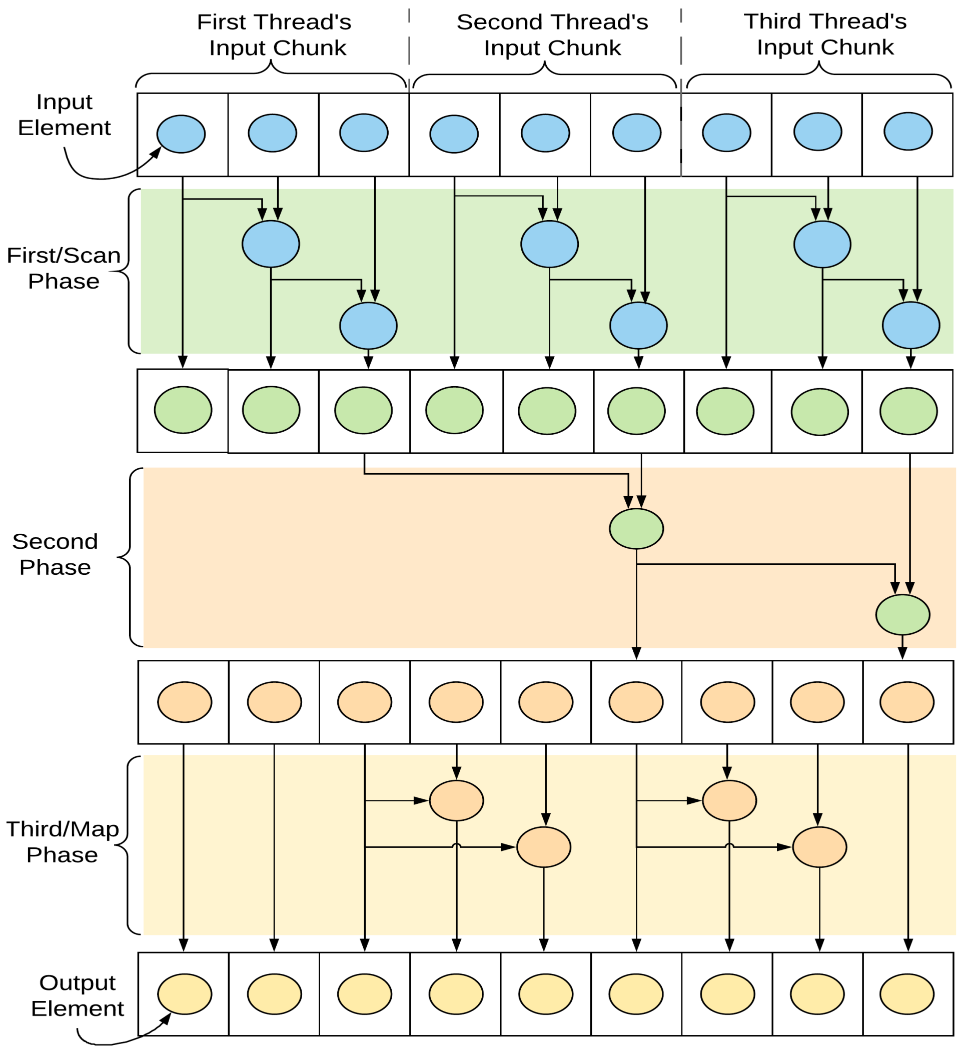

First Thread's Input Chunk
Second Thread's Input Chunk
Third Thread's Input Chunk
Input Element
First/Scan Phase
Second Phase
Third/Map Phase
Output Element

Figure 3.18: The three phases of Baseline version, with 9 elements handled by 3 threads.

3.5.2.2 Scan: Full incorporation of task-stealing (Version 2)

On the second version, based on its successful employment with our previous Skeletons, we adjusted our dynamic load balance mechanism on Scan Skeleton as well. In order to do so, we have implemented our “Data-block” algorithm on the scan and map phases of our baseline version.

From the first moment towards applying the data block logic on the scan phase, we realise that we encounter exactly the same problems that come along with our initial task. The only difference being that instead of applying a scan to the whole input vector and its chunks, we now apply the operation on each chunk and its data-blocks. Consequently, we proceeded by applying a fusion of our baseline version’s algorithm, and the data-block's algorithm, as discussed in previous sections, to each chunk.

Conceptually, the modified scan phase algorithm is divided into three main steps, equivalently with the baseline version's phases. On the first step, we apply a sequential prefix-sum operation on each data-block, handled by the thread assigned to it, and the second step has the last thread active on the chunk, to utilise a partial prefix-sum that includes only the last element of each data-block. The third step, which corresponds to the map phase of the baseline version , after a synchronisation barrier ensuring that all threads have completed the first two steps, instructs each thread to invoke the successor function with the previous data-block's last element and each element before the last one of the current data-block.

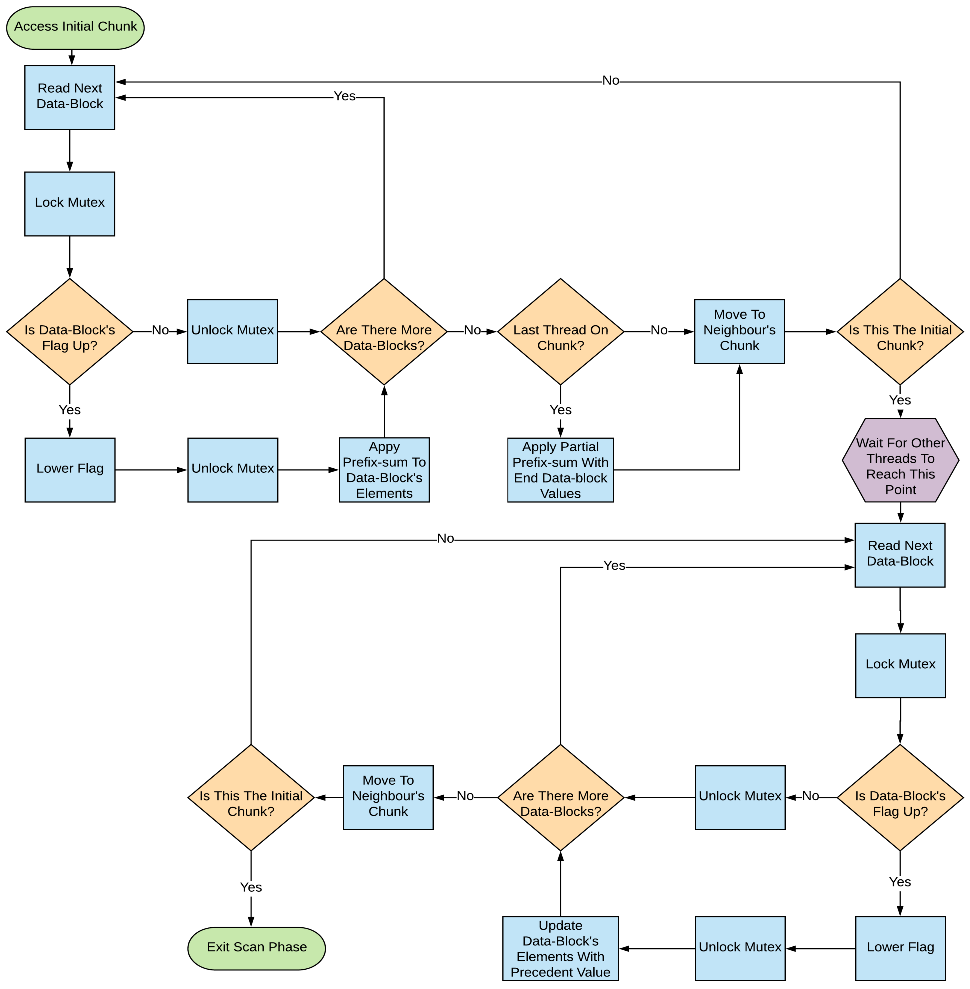

Access Initial Chunk
Read Next Data-Block
No
Yes
Lock Mutex
Is Data-Block's Flag Up?
No
Unlock Mutex
Are There More Data-Blocks?
No
Last Thread On Chunk?
No
Move To Neighbour's Chunk
Is This The Initial Chunk?
Yes
Yes
Yes
Lower Flag
Unlock Mutex
Appy Prefix-sum To Data-Block's Elements
Apply Partial Prefix-sum With End Data-block Values
Wait For Other Threads To Reach This Point
No
Read Next Data-Block
Yes
Lock Mutex
Is This The Initial Chunk?
Move To Neighbour's Chunk
No
Are There More Data-Blocks?
Unlock Mutex
No
Is Data-Block's Flag Up?
Yes
Yes
Exit Scan Phase
Update Data-Block's Elements With Precedent Value
Unlock Mutex
Lower Flag

Figure 3.19: Scan version 2. The Scan-phase algorithm.

This modification of the scan phase, as shown in figure 3.19, increases the complexity to a large extent. Specifically, the initial solution has a complexity for the scan phase of $O(\frac{N}{P})$, where N the length of the input vector, and P the available concurrency. The remodelled solution has the corresponding complexity, in best case, to $O((\frac{N}{P}-1)+(P-1)+(\frac{N \cdot P-N}{P^2}))$, where each term matches the complexity of each step in the modified scan phase. The overhead introduced has a great impact on performance, and this will be analysed further in the evaluation chapter (section 4.3.4.2).

In concern of the map phase, the algorithm applied is almost identical with the Data-block algorithm presented in Map Skeleton. The difference occurs with the fact that in this situation, the precedent chunk's last element is applied with each element of the current chunk to the successor, providing the final result.

Since this version's increased complexity on the scan phase has deteriorated the Skeleton's performance, we decided to partially reject it (as task-stealing on the map phase will be applied again on the fourth version), and move on to our third version, which enhances the baseline version's second phase.

3.5.2.3 Scan: An enhanced two-pass binary tree (Version 3)

For our third version, we decided to optimise the second phase of our baseline version. The reason is that as this phase is handled sequentially, and depending on the execution time of the successor function, even a potentially small number of successor function invocations, which is equal to the number of threads in use, could introduce a significant amount of bottleneck to the whole process.

In order to transform our second phase's implementation to a parallel one, we followed the algorithm presented in [4]. Algorithmically, a concurrent scan with N elements and P threads, where $N = P$, could be achieved with the use of a "two-pass binary tree". Thus, we adapt this in our second phase, by processing only the last element of each chunk, resulting to our desired partial scan. The tree is build bottom-up, the "Up-pass", and produces the result in a top-down passage, the "Down-pass", while making an efficient use of the available parallelism.

The Up-pass is conceptually the same operation as the binary tree used in the case of Reduce Skeleton (section 3.3.2.1), with the difference that now each node holds two values. The first one, is the aggregated sum-value of the node's children - "sum", and the second one is the prefix-sum of all the elements preceding the node's leftmost leaf - "left".

The Down-pass, starting from the root, instructs each node to pass its "left" value to its left child's "left" value, and assign to its right child's "left" value the result of the successor function with arguments the node's "left" and the left child's "sum" values. Finally, the leafs combine their own "left" and "sum" values to produce their prefix-sum result. An example is shown in figure 3.20.

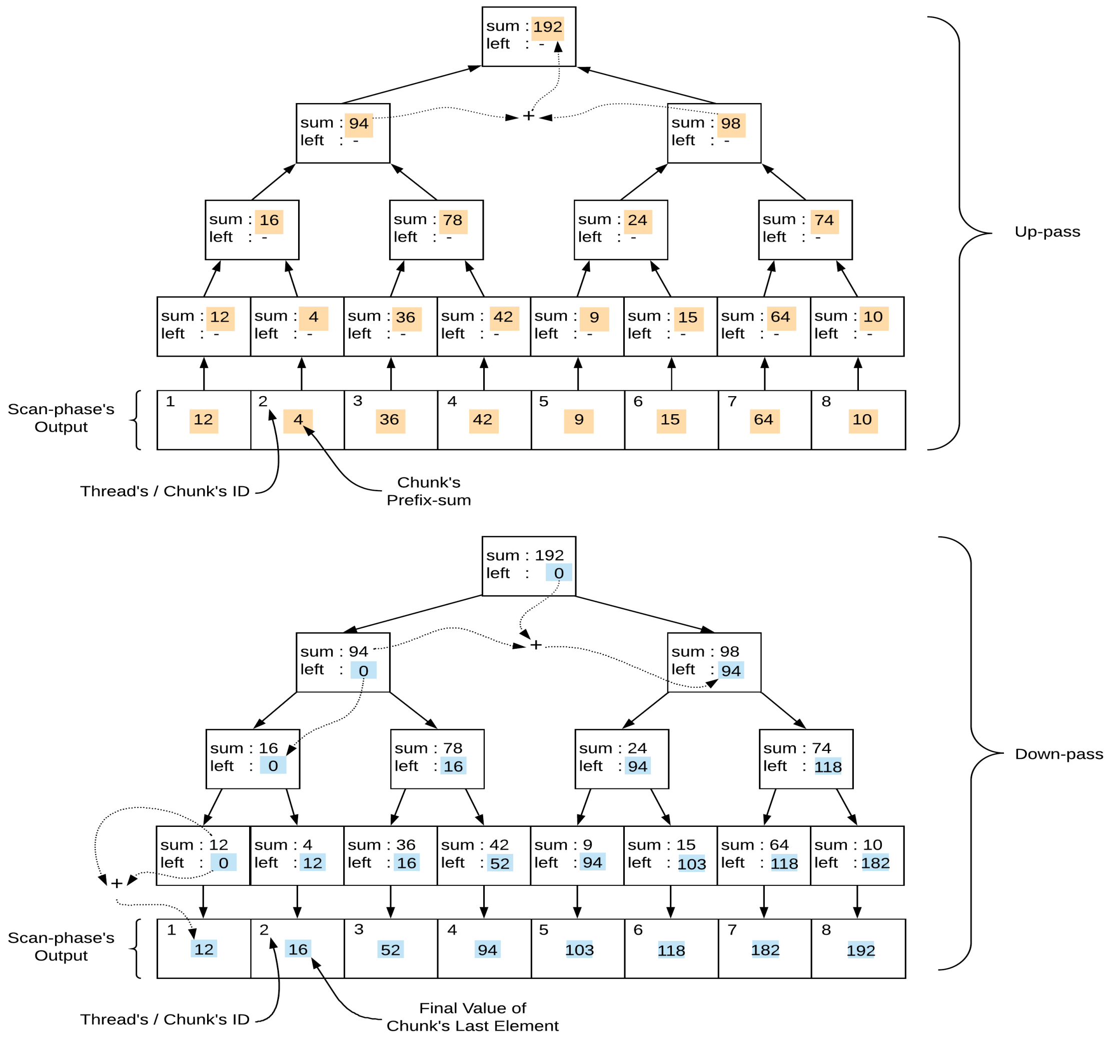


Figure 3.20: A Scan of 8 integers, with addition as successor operation, handled with the use of a two-pass binary tree of 8 threads.

Technically, starting from the leafs, each thread creates and initialises a node, and based on its ID, a decision is made whether the thread will proceed upwards or wait in a condition variable to be woken in the Down-pass. The implementation defines that the left child/thread will move up and create the parent node, and the right child/thread will hibernate until the tree is traversed downwards. When we reach a level where the only active thread is the first one (based on its ID), we know that we have reached the root-level and the Up-pass is completed.

For the Down-pass, starting from the root, the thread after completing the required calculations for its children, before moving on to work as its left child, signals the right child/thread to "resume" and continue with the Down-pass phase.

To conclude this version's description, the "enhanced " term is given for our contribution to the algorithm to handle an odd number of nodes in any tree level. In contrast with the equivalent improvement in the Reduce Skeleton, this time the carry element, which is always the rightmost node in the occasion, is "combined" with its (left) neighbour and the operation is handled by the thread responsible for that neighbour.

3.5.2.4 Scan: Dynamic load balance on map-phase (Version 4)

The last version implemented, reintroduces task-stealing, but this time based on the third version of the Skeleton and for the map phase alone. The algorithm is identical with the part presented in section 3.5.2.2, and the performance as shown in the evaluation chapter (see section 4.3.4.4) is improved.

Chapter 4

Evaluation

4.1 Overview

In this chapter we present the library's performance on various tasks. Firstly, we illustrate and discuss the proficiency of each Skeleton version, in the same order as presented in chapter 3, with a number of simple benchmarks that we have developed. Those benchmarks measure execution time of a task, and compare the metrics of a sequential and a Skeleton implementation. The demonstration includes results from a Desktop-DICE machine, and from a personal computer (Non-DICE machine) with better hardware support. Secondly, we exemplify further the quality of our library, by solving real-life problems, and their solutions have been implemented in three different ways. The first one is with a sequential implementation, the second one is with a parallel implementation that makes direct use of the Pthreads library, and the last one is achieved only with the aid of our Skeleton library. For the latter case, we compare execution time and programmability in terms of "lines-of-code".

4.2 Technical Information

The benchmarks were conducted on two different machines. The first one being a Desktop DICE computer, and the second one a personal computer (Non-DICE environment). The specifications of each machine are shown in table 4.1. As stated in the table, both Desktop-DICE and Non-DICE machines have 4 physical processors, but the Non-DICE machine holds the Simultaneous MultiThreading (SMT) technology, and so the former operates with 4 CPUs and the latter with 8 (two virtual cores per processor). For this reason we expect better processing power on the Non-DICE ma-

	Dekstop DICE	Non-DICE Computer
CPUs	4	8 (With SMT)
Physical Processors	4	4
CPU max freq. (GHz)	3.60	3.50
L1 cache	32K	32K
L2 cache	256K	256K
L3 cache	6144K	6144K

Table 4.1: Hardware specifications of Desktop DICE and Non-DICE Computer

chine. In addition, both machines have the exact same cache-memory hierarchy with the exact same size, which indicates that the cache performance should be roughly the same on both computers.

All evaluation programs were compiled with g++ version 4.8.5, except the benchmarks for Reduce and Map Skeletons. For those two Skeletons, clang++ version 3.4.2 was used, as the aforementioned version of g++ does not support all C++11 features that are necessary for the "Thread-pool implementations". Both these compiler versions are available on DICE. In addition, all programs were compiled with the default optimisation level (O2).

4.3 Benchmarks

During the development of our library, in order to verify the efficiency of each version, we implemented simple test suites for each Skeleton, that measure and compare the average execution time of a Skeleton call and a equivalent sequential code. The evaluation of each Skeleton version is based on these results.

As the actual task given to each Skeleton in this stage, does not consume a significant amount of time per input element, we have added the ability to control the granularity of the workload artificially on each Skeleton function invocation (i.e. the elemental for the Map Skeleton), with the use of the "Collatz function" [2]. This function is defined as:

$$f(n) = \begin{cases} \dfrac{n}{2} & , if\ n\ is\ even \\ \\ 3 \cdot n + 1 & , if\ n\ is\ odd \end{cases}$$

Even if it is not yet proven, starting from any positive integer, if we apply this integer to the function, and recursively the function's result, eventually we will reach the value "1". Thus, we can use this function in computer science in order to give an impossible optimisation task to the compiler, which means on each Collatz function call, for a large enough initial number, the binary code produced by the compiler will have to execute each and every iteration step of that function in order to produce the correct result. For this reason, we call the Collatz function inside the Skeleton's element procedure, and we simulate a greater "workload" per element. After experimentation, we have chosen the value "871" for N (Collatz seed), that produces 178 iterations before its termination, in order for the performance scaling of each Skeleton, while we introduce more than 1 thread, to be easily recognised.

In addition, On each benchmark related graph, we present two different metrics. the left axis (brown) measures the average Skeleton execution time over the average sequential execution time (speed-up), and the right axis (blue) is the actual average execution time of the Skeleton. The brown bars correlate to the left axis, and the blue line to the right one. The green mark represents the average sequential time for the task. A speed-up of "1" means that the execution time for the Skeleton and the sequential code is exactly the same. On the horizontal axis we see the number of threads used on each run. Moreover, for each figure, a legend is given, that states the available hardware concurrency (virtual CPUs), the number of runs executed to calculate the average time, the number of elements per run, and the Collatz seed in use.

Finally, for each Skeleton, the baseline version benchmarks are presented for both machines (Non-DICE and Desktop-DICE), and for every version after, the presentation and comparison is made only on the recorded performance of the Non-DICE machine.

4.3.1 Map Skeleton

Our test for Map Skeleton defines an elemental function that for a given element, the map result is produced by adding that element with a constant value. The actual C++ function is presented in listing 4.1.

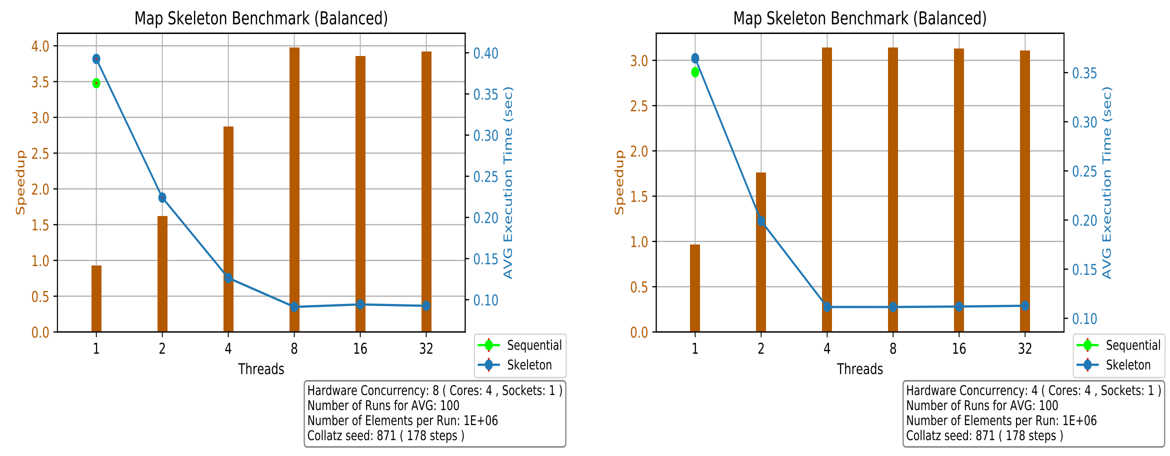
```cpp
//constantValue is an extra argument for the Map Skeleton
int elemental(int element, int constantValue) {
 collatz();

 return element + constantValue;
}
```

Listing 4.1: The elemental function used in benchmark.

After the construction of a Map functor, we apply this operation on a vector of 10^6 integer elements, and the total result is validated with an equivalent sequential procedure, which is an iteration of applying the elemental function to all the input elements. This test is repeated for a range of threads in use by the Skeleton, and then for all three versions of Map. The generated graphs with the metrics are presented in the following sections.

4.3.1.1 Map: Baseline version

As illustrated in the figure 4.1 (a), the speed-up is slightly less than 1 while using only 1 thread, revealing the Skeleton's overhead, and it is doubled right after with the use of 2 threads. The speed-up keeps rising up to 8 threads, and then as the available hardware concurrency stops, the Skeleton reaches a plateau. A similar trend is achieved when the measurement is done on a Desktop DICE machine as presented in the figure, subplot (b), with the only difference that the peak is achieved by the time we use 4 threads, which is normal based on the hardware on this machine.



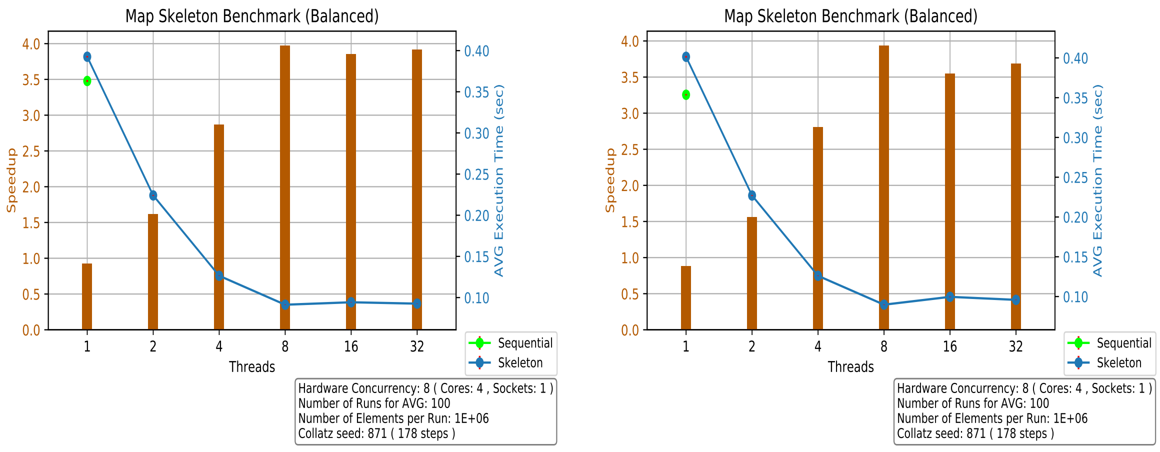
(a) Non-DICE Machine (b) Desktop DICE Machine

Figure 4.1: Baseline version for a non-DICE and a Desktop DICE machine, for an equal amount of workload per element (Balanced).

4.3.1.2 Map: Second version (thread pool)

The second version of Map, where we introduced the concept of thread-pool, indicates that with a significant number of elements (10^6) and a Collatz seed that produces 178 steps, does not improve the average execution time. This is shown in figure 4.2, and we can observe that for the Baseline version (a), the performance gain is barely greater than the one we achieved with a thread pool implementation (b). This is based on the fact that the extra algorithmic structure required on the second version of Map, introduced an overhead per functor call, which appears to be approximately the same with the overhead generated, per functor call, by thread creation on the baseline version.

It should be noted that further investigation should have been made in this case, which would use less number of elements per run (less workload), so that the second version's overhead discussed on the previous paragraph, would have been proven that it is not hidden by the process time required for all the elements.

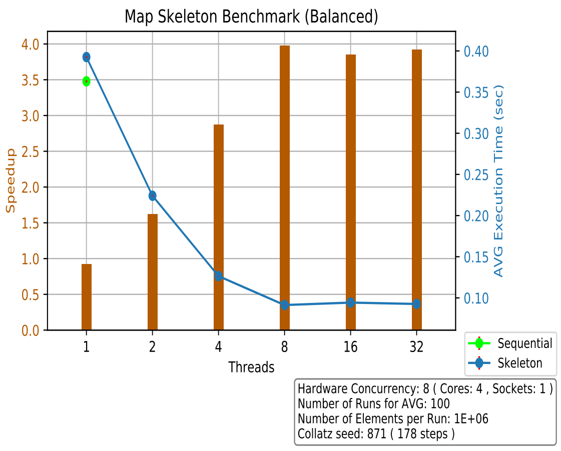


(a) Baseline Version (b) Second version (thread pool)

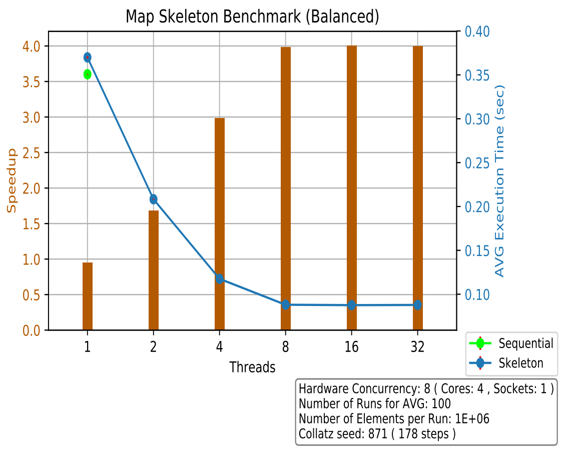
Figure 4.2: Map: Baseline (a) and second (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements.

4.3.1.3 Map: Third version (dynamic load balance)

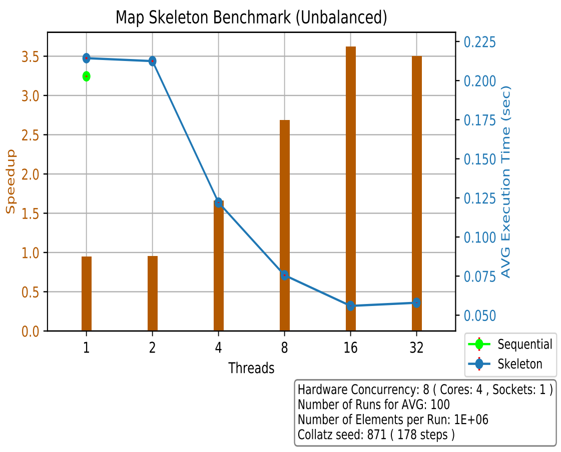
On our third version, where we introduced the task-stealing mechanism, performance-wise, we observe that we achieve almost identical results with our baseline version in the case of an even amount of process time per element. As illustrated in figure 4.3, the only observable speed-up contrast between (a) and (b), occurs while deploying 4 threads for the run, and since the difference is approximately 0.2, the total performance is considered the same. In addition, the fluctuation with more threads than the available concurrency is not worthwhile for discussion, as our experiments revealed that the behaviour above this limit is unpredicted in this extent.



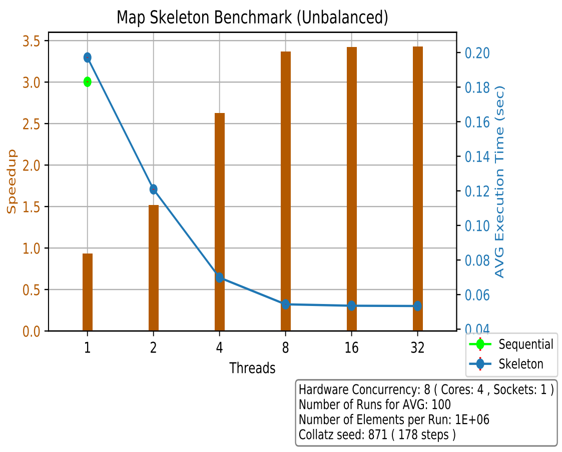
(a) Baseline Version (Balanced)



(b) Third version (Task-stealing, Balanced)



(c) Baseline Version (UnBalanced)



(d) Third version (Task-stealing, UnBalanced)

Figure 4.3: Map: Baseline (a) and third (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements. Sub-figures (c) and (d) present the versions' comparison for Unbalanced workload

In contrast, when the comparison is made with an unbalanced workload, which is achieved by invoking the Collatz function with a number that produces only 1 step for half the elements, and our common Collatz seed that produces 178 steps for the other half, the performance is completely different between those two versions. As we see in the same figure (c and d), the Baseline version has around the same performance gain either with the use of 1 or 2 threads. This performance is increased only after the employment of 4 threads, and escalates up to the point we operate with 16 threads.

It is interesting that we see a growth with more threads (16) than the available hardware concurrency (8), and the explanation on this behaviour is that this is feasible due to the fact that the workload is uneven among the elements. To explain this further we will start with the case of 1 thread. Let us assume that for half the elements, the execution time of the elemental has a time-cost of “1” (same as the Collatz seed for this half) per element, and that the addition operation is included in that time. For the other half the Collatz seed is 178, thus with the same logic the time-cost will be “178” per element. For N elements and 1 thread, the total execution time will be $\frac{N+178 \cdot N}{2}$.

When we employ 2 threads, since our algorithm for the Baseline version divides the input vector into halves, the first thread undertakes the first half of elements, that has an execution time of $\frac{N}{2}$, and the second thread will handle the second half, that has an execution time of $\frac{178 \cdot N}{2}$. Since the threads work simultaneously, the second thread’s execution time overlaps the first one’s, resulting to a total execution time equal to $\frac{178 \cdot N}{2}$, which is slightly less than the one consumed when we use 1 thread. For this reason, we gain performance by the time we introduce 4 threads to the task, as the “costly” half is handled by 2 threads in this case, thus we observe a doubling in speed-up. By following this analysis for all the remaining number of threads, the metrics that make the brown bars in sub-figure (c) to look like they have been shifted to the right are justified.

This trend is not occurring with our third version, as in this implementation, the threads that have completed their work on the first half, move on to assist the threads on the time-demanding section, and so the total execution time for each case, is more or less the same as when we process a balanced workload.

It should be noted that the decision to create the extra artificial demand to only the second half of the input vector, is made in order to present the “best-case” scenario for the third version’s enhancement, and as “worst-case”, where the Collatz seed would be alternating between “1” and “178” throughout the whole vector, theoretically the performance gain would have had no difference. We aim to present such a case in our future work.

4.3.2 Reduce Skeleton

The validation of the Reduce Skeleton is achieved with a similar test as the Map Skeleton, with the combiner function applying the addition operation to its arguments. The actual C++ function is presented in listing 4.2.

```
double combiner(double a, double b) {
 collatz();

 return a + b;
}
```

Listing 4.2: The combiner function used in benchmark.

For our benchmark, we process 10^6 real numbers, and for each Skeleton run, the reduced value is checked to match the result of an equivalent sequential code. The sequential code is simply an iteration of invoking the combiner function on the elements that are used as input for the Skeleton. In addition, the speed-up measured for each Skeleton run, is relative to the sequential code. Likewise the Map Skeleton, the benchmarks for each Reduce Skeleton version are presented in the following sections.

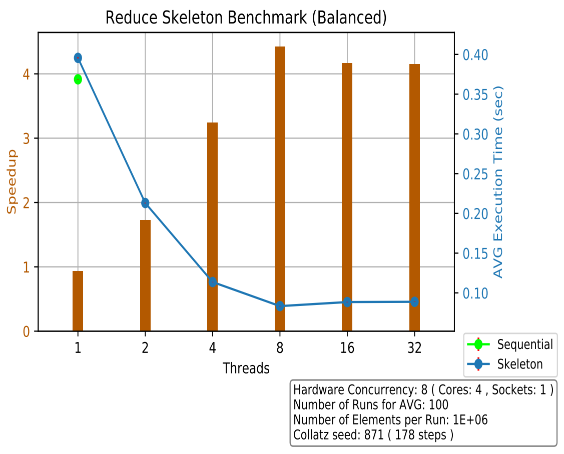
4.3.2.1 Reduce: Baseline version

In the first graph presented for this version (figure 4.4, (a)), the performance on a non-DICE computer for the Reduce Skeleton, has about the same trend with the Map Skeleton's baseline version on the same machine. By starting with a speed-up close to "1" for 1 thread, a growth for every time we double the number of threads is achieved, up to the point of 8 threads (SMT cores) where the peak occurs. The bar chart shows that for every run that we double the available concurrency, while using no more threads than the actual physical cores (4), the speed-up is doubled as well. This information indicates that the Reduce Skeleton's baseline version is up to the expectations for a parallel implementation of the pattern.

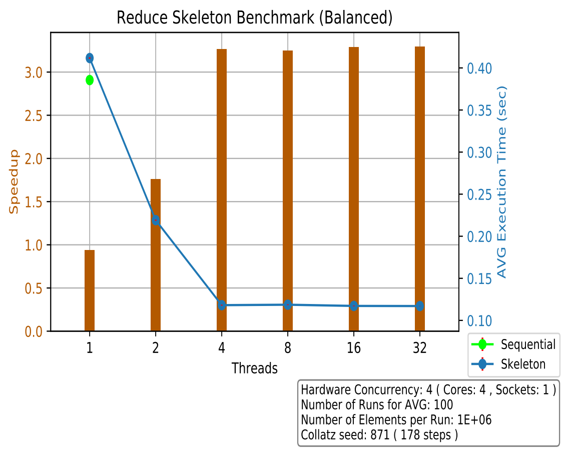
On the second sub-figure (b), the benchmark took place on a Desktop DICE computer. We observe that the same tendency occurs on this machine, which has the performance gain to be doubled for every time we increase the number of threads by the same factor. The bar charts are flatten after we cross the available physical cores (4) on this computer, and as the there is no support of SMT cores in this case, the behaviour is considered normal.

4.3.2.2 Reduce: Second version (thread pool)

As presented in the bar charts in figure 4.5, the two versions, baseline (a) and second (b), have the same behaviour relative to speed-up when we use as many threads as the machine's physical cores. By the time we employ 8 threads (equal to SMT cores), we witness a surge on the metric for the second version, of approximately 0.3 units.



(a) Non-DICE Machine



(b) Desktop DICE Machine

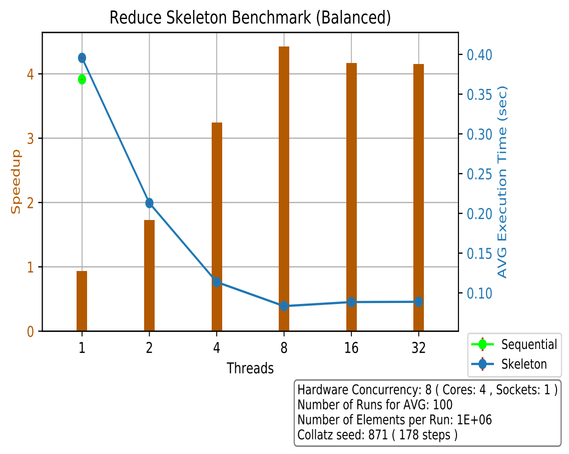
Figure 4.4: Baseline version for a non-DICE and a Desktop DICE machine, for an equal amount of workload per element (Balanced).

This observation requires profound knowledge of the SMT mechanism in order to be justified, and the author does not hold this knowledge to comment any further. This benchmark, similarly with the Map Skeleton's thread pool case, indicates that the second version only manages to replace the thread creation overhead on every functor call, with the required overhead of a thread pool implementation. For this reason, the decision was made to announce the baseline version as the "winner", in addition to that the baseline version's Skeleton module has significantly simpler structure.

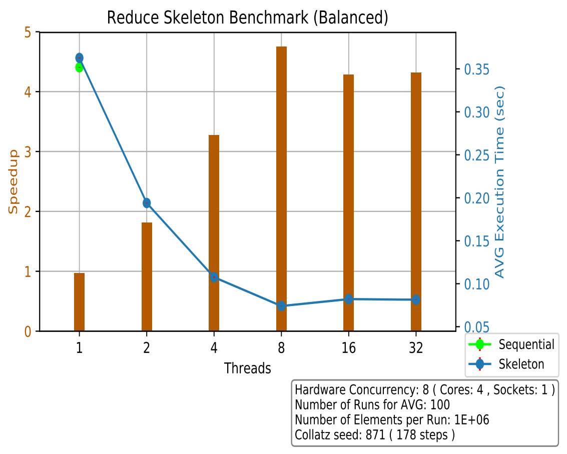
Finally, as already mentioned in the corresponding section for Map Skeleton, in future work, the thread pool implementation must be tested under smaller workload, in order to verify that the behaviour we observe in the graphs is irrelevant of the input process demand. Meaning that the similarity in the overhead, is not affected by the total input processing time.

4.3.2.3 Reduce: Third version (dynamic load balance)

The last version implemented for Reduce Skeleton, handles the problem of an uneven processing time per combiner call. In the plots presented in figure 4.6, we see that when the comparison is made with a balanced workload, both versions achieve almost identical performance. The only measured difference transpires while using 4 threads, one for each physical core, and it is estimated to 0.2 units. This is an indication that the Data-block algorithm presented in section 3.3.2.3, raises slightly the Reduce Skeleton's overhead.



(a) Baseline Version



(b) Second version (thread pool)

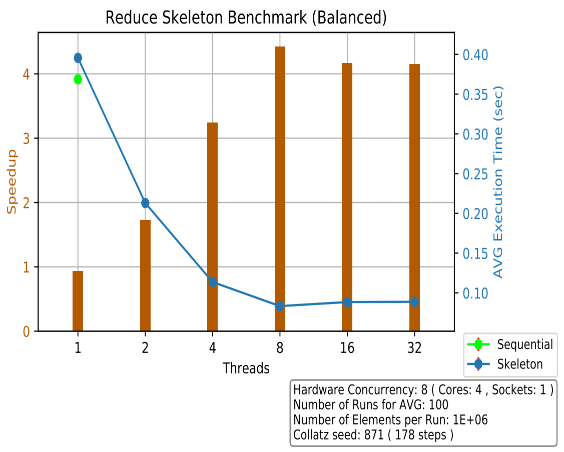
Figure 4.5: Reduce: Baseline (a) and second (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements.

However, when we compare the baseline and third version on an unbalanced workload, we observe that the third version can be declared the "winner" without a doubt. This is due to the fact that after the case of 1 thread, for every point in the horizontal axis up to 16 threads, the performance gain for the third version is considerably greater than its opponent. Relative to the baseline version, the "right-shifted" bars effect for the unbalanced benchmark, is based on the same reason as explained for the Map Skeleton in section 4.3.1.3. Once gain, the unbalanced benchmark assigns a Collatz seed that produces 178 steps for half the combiner's invocations, and on the other half, a seed that produces 1 iteration for the Collatz function.

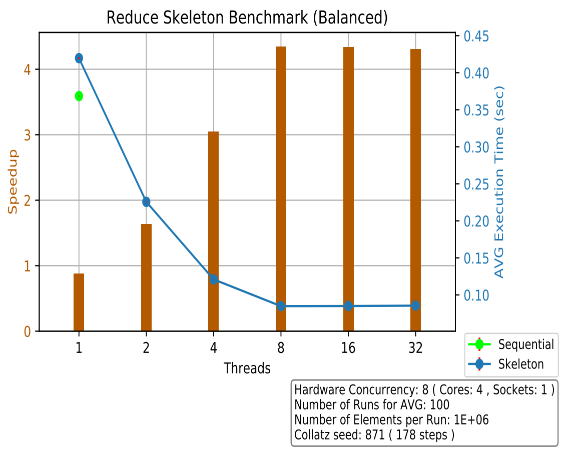
4.3.3 MapReduce Skeleton

To benchmark our MapReduce implementations, we have chosen a slightly more complicated definitions for the Mapper and Reducer functions. We decided to test the Skeleton with the same number of elements (10^6) as with the previous benchmarks that we have presented, but with the case of MapReduce, this number represents the intermediate number of values that the Skeleton processes, and not the number of input elements.

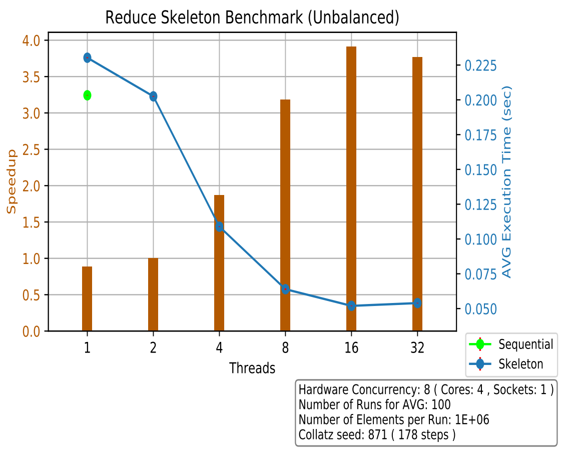
To achieve this, we defined the size of the input vector to 10^3, and we implemented a mapper function that for each element, produces 10^3 key-value pairs. The keys are the enumeration of the integers in range $[1,10^3]$, and the value of the pairs is the same



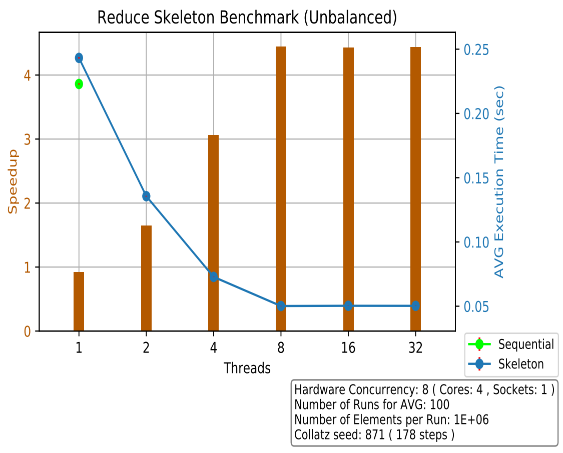
(a) Baseline Version (Balanced)



(b) Third version (Task-stealing, Balanced)



(c) Baseline Version (UnBalanced)



(d) Third version (Task-stealing, UnBalanced)

Figure 4.6: Reduce: Baseline (a) and third (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements. Sub-figures (c) and (d) present the versions' comparison for Unbalanced workload

input element. The reducer function accumulates the values per key, and returns a list that contains only that accumulated value. The C++ functions are shown in listings 4.3 and 4.4 respectively.

```
std::list<std::pair<int,double>>
mapper(int element, int inputVectorSize) {

 collatz();

 std::list<std::pair<int,double>> result;
 int i = 0;

 while(i++ < inputVectorSize)
```

```
  result.push_back(std::make_pair(i, (double) element));

 return result;
}
```

Listing 4.3: The mapper function used in benchmark.

```
std::list<double> reducer(int key, std::list<double> values) {

 collatz();

 std::list<double> result;
 double bucket = 0.0;

 for( auto &v : values )
  bucket += v;

 result.push_back( bucket );
 return result;
}
```

Listing 4.4: The reducer function used in benchmark.

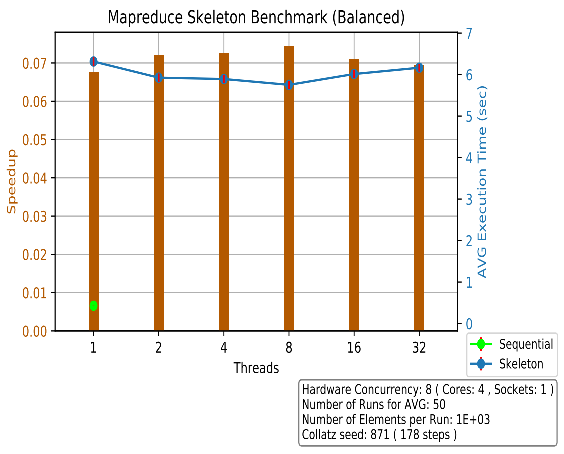
4.3.3.1 MapReduce: Baseline version

The bar charts presented in figure 4.7 reveal that our initial implementation for the Skeleton has an extremely poor performance. Either in the case of a Desktop DICE (b) or in a Non-DICE machine (a), while the sequential code's execution time is approximately 0.5 seconds, the Skeleton manages to deliver the same output, but it is slower by a factor of 10.

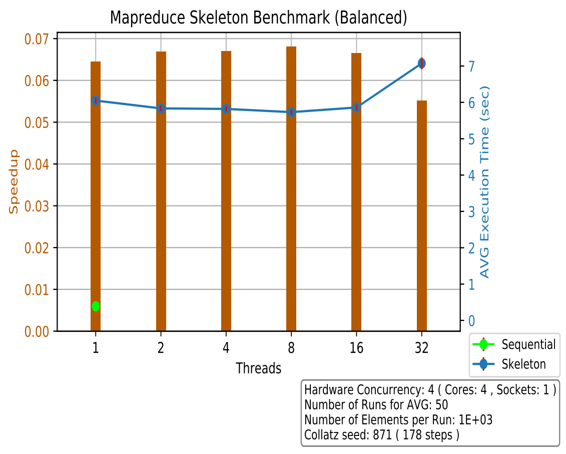
In detail, for both machines, the execution time per run is around 6 seconds, and this performance is almost irrelevant to the number of threads in use. This indicates that the baseline version's sequential code-block (see "intermediate stage" in section 3.4.2.1), creates a severe congestion for the process, and verifies that our decision to optimise this code section, which was based on the Skeleton's profiling during the implementation stage, was necessary.

4.3.3.2 MapReduce: Second version (improved intermediate stage)

The optimization applied to the intermediate stage of the baseline version, which formed our second version, as indicated by the plots in figure 4.8, is proved to be



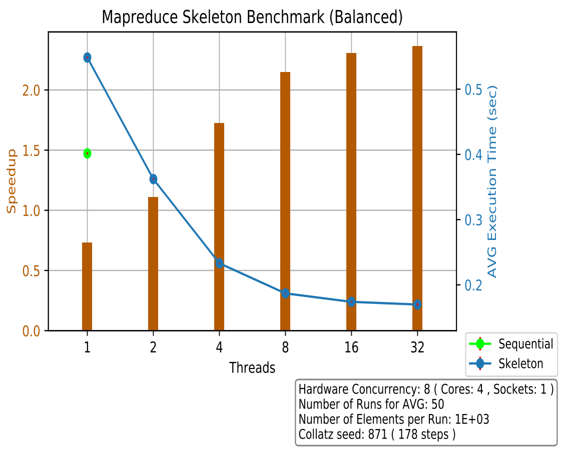
(a) Non-DICE Machine



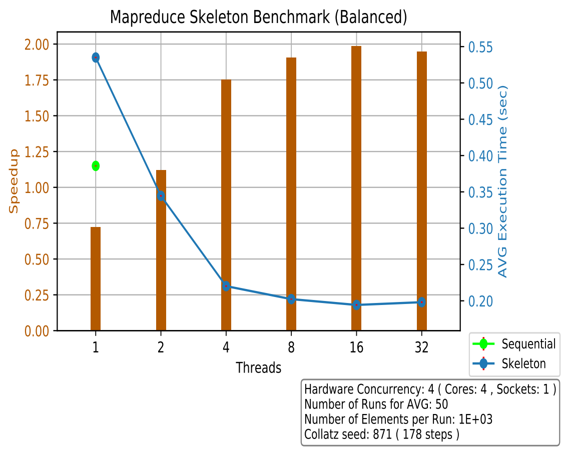
(b) Desktop DICE Machine

Figure 4.7: Baseline version for a non-DICE and a Desktop DICE machine, for an equal amount of workload per element (Balanced).

colossal. With this version, the Skeleton achieves the basic property of a parallel implementation, which is to perform better every time we increase the number of CPU cores in use, and manages to perform at least 10 times faster, compared to the baseline version, even when we employ only 1 thread for the task.



(a) Non-DICE Machine



(b) Desktop DICE Machine

Figure 4.8: Second version for a non-DICE and a Desktop DICE machine, for an equal amount of workload per element (Balanced).

However, we observe that the performance gain is not doubled as we scale the number of threads, a trend that exists in Reduce and Map Skeleton, and this is due to the fact that MapReduce Skeleton has greater overhead compared to the aforesaid ones. This overhead, as explained in chapter 3, is a necessary sequential stage, in order to prepare an equally distributed among threads reducer input, and effects the performance in such degree, that even when we use 4 cores, the speed-up is still less than a factor of 2.

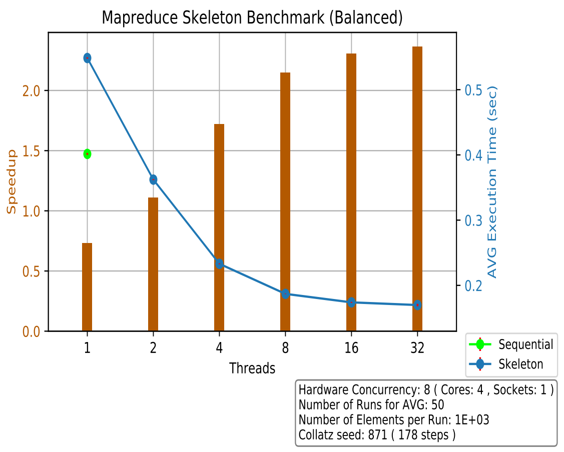
While it may be true that the speed-up is not is the same scale as the previously discussed cases (Reduce and Map Skeletons), the second version of MapReduce accomplishes far better benchmarks compared to our initial approach, and worthy provides the foundations for our third version.

4.3.3.3 MapReduce: Third version (dynamic load balance)

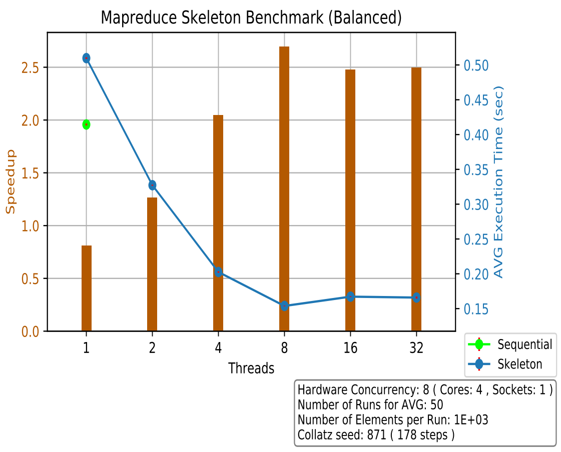
As we can see in figure 4.9, the third version (b) attains the same curve, for the average execution time over the provided concurrency, as the second version (a) for a balanced workload, but this curve is shifted downwards. This is due to the fact that the former has more or less 25 per cent better speed-up, compared to the latter. This is achieved because the third version, almost completely removed the sequential code in the "intermediate stage" of the Skeleton (where we pursue to equally distribute the workload of keys for the reducer stage), by introducing the task-stealing mechanism in reducer stage, which declares the second version's key-preparation stage unnecessary. Moving onwards to the unbalanced benchmark, we will firstly present our different technique, compared to Reduce and Map Skeleton, for simulating an uneven workload.

Since MapReduce is a "compound" Skeleton, we had to alter our strategy for injecting additional workload per element. This is accomplished by defining the mapper to produce the same amount of key-value pairs (10^3), for half the input elements, and the other half to produce none. Likewise, the reducer accumulates the values for half of its keys, while returning 0.0 for the rest. Finally in order to maintain 10^6 intermediate values, we defined the input vector to carry $(2 \cdot 10^3)$ elements. The modified versions of mapper and reducer are shown in listings 4.5 and 4.6.

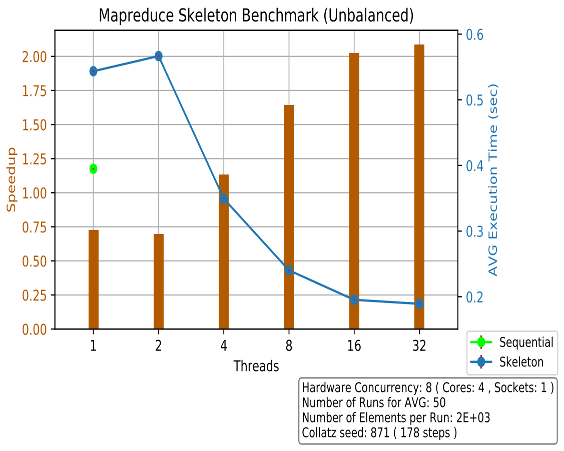
```
std::list<std::pair<int,double>>
mapper(int element, int inputVectorSize) {
 std::list<std::pair<int,double>> result; int i = 0;
 if( input > inputVectorSize/2 ) {
  collatz();
  while(i++ < inputVectorSize/2)
   result.push_back(std::make_pair(i, (double) input));
 }
 return result;
```



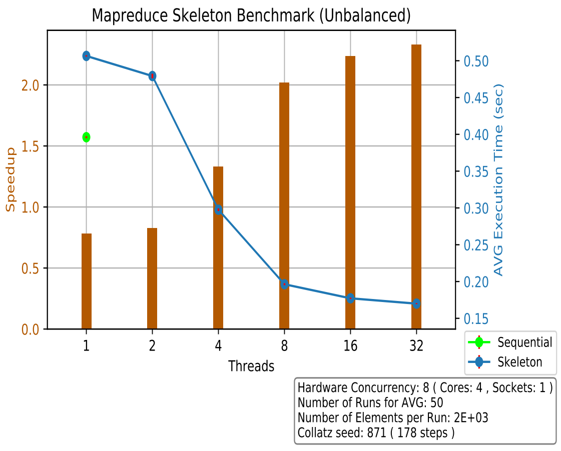
(a) Second Version (Balanced)



(b) Third version (Task-stealing, Balanced)



(c) Second Version (UnBalanced)



(d) Third version (Task-stealing, UnBalanced)

Figure 4.9: MapReduce: Second (a) and third (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 intermediate values. Sub-figures (c) and (d) present the versions' comparison for Unbalanced workload

```
}
```

Listing 4.5: The mapper function used in unbalanced benchmark.

```
std::list<double>
reducer(int key, std::list<double> values
  ,int inputVectorSize) {

 std::list<double> result;
 double bucket = 0.0;

 if( key < inputVectorSize/4 ){
  collatz();
  for( auto &v : values )
   bucket += v;
 }
 result.push_back( bucket );
 return result;
}
```

Listing 4.6: The reducer function used in unbalanced benchmark.

As we see in sub-figure (c), the second version's performance is deteriorating when we use 2 threads. The explanation for this behaviour, is that with unbalanced workload, similarly with Map Skeleton, the first thread does not assist on the second one's work, and since the second chunk is responsible for the total execution time (as it carries the heavy-work), the second thread, "single-handedly" , completes the mapper's task. Thus, for the mapper stage, the execution time is the same , either with the use of 1 or 2 threads. In addition, while using 1 thread, the hash-table merging, explained in section 3.4.2.2, is not triggered, and the algorithm illustrated in figure 3.14, has only its upper loop executed. These steps after the mapper stage are not avoided when we use 2 threads, and for this reason we measure a peak on average execution time. In contrast, as illustrated in sub-figure (d), with the third version the performance is immediately improved when we introduce parallelism, as the first thread assists the second one.

Finally, as our definition of reducer does not guarantee that only half the threads will have heavy tasks, this benchmark does not exhibits the full potential of the version. To explain this further, this is due to the fact that we do not force the keys that accumulate values to be assign to specific threads, thus they have the potential to spread evenly to all threads.

4.3.4 Scan Skeleton

Our last benchmark set presented in this section relates to Scan Skeleton. In contrast with the other Skeletons, the evaluation suite in this case has specific criteria for each version, resulting the number of input elements under process not being the same across all tests, and 3 different successor functions to be defined.

Synoptically, the first version that implements a straightforward parallel algorithm as presented in section 3.5.2.1, is compared against the second version, which applies dynamic load balance on both scan and map phases of the algorithm. As the former version outperforms the latter, it proceeds to be compared with the third version, that introduces the two-phase binary structure, on two different test cases. Finally, as the third version outperforms in this distinct case the baseline version, it is set side by side with our fourth implementation, where we have re-introduce task-stealing for the map phase alone.

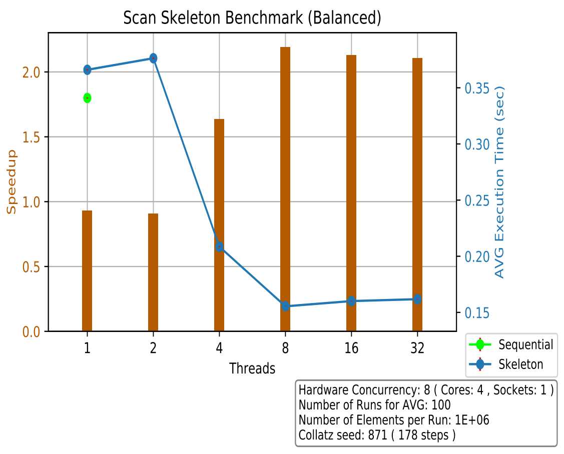
4.3.4.1 Scan: Baseline version

For this benchmark, we defined the successor function presented in listing 4.7, and applied the prefix-sum operation on 10^6 input elements.

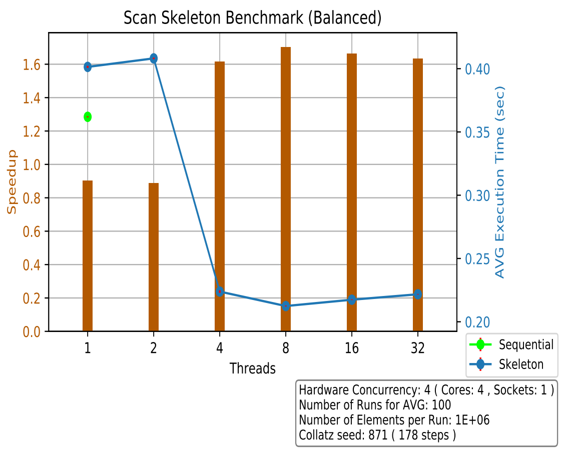
```
double successor(double a, double b) {

    collatz();

    return a+b;
}
```

Listing 4.7: The successor function used in baseline version's benchmark.

As it is shown in figure 4.10, Scan has a similar behaviour as the one illustrated in figure 4.9 for MapReduce Skeleton. The performance drop we witness when 2 threads are assigned to the task, is due to the fact the version's second and third phase, is not triggered with the use of 1 thread.



(a) Non-DICE Machine



(b) Desktop DICE Machine

Figure 4.10: Baseline version for a non-DICE and a Desktop DICE machine, for an equal amount of workload per element (Balanced).

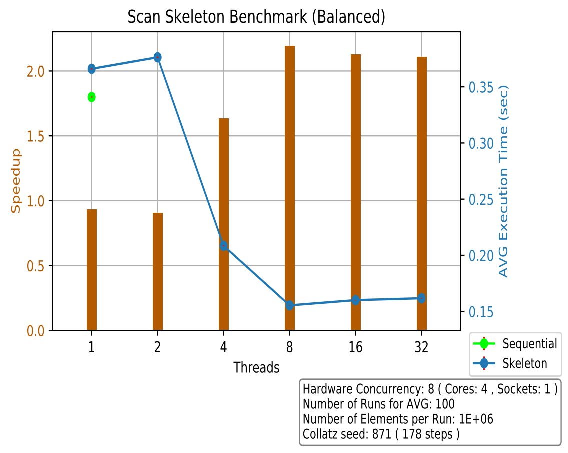
To explain this further, with N input elements (10^6) and 1 thread, by the end of the first phase of the Skeleton, the final output is ready, as we only have 1 chunk. So in this case we access N elements. When we employ 2 threads, the second and third phase of the algorithm is enabled, and 2 chunks are created. For the first phase (scan) we access again N elements, but the process time is halved with the available concurrency, which is equivalent of processing $\frac{N}{2}$ elements sequentially. Likewise, the same applies for the third phase (map), and so the total execution time tends to be exactly the same as when we provide only 1 thread to the Skeleton ($\frac{N}{2} + \frac{N}{2} = N$). Finally, a small deterioration is present, as the second phase is also enabled in the process.

Furthermore, by creating 4 threads, we observe that the speed-up is almost doubled from its precedent value, a trend that does not keep up by doubling the concurrency once more. With the maximum available CPUs (8 with SMT), the Skeleton only manages to attain a performance gain of approximately 2.2 in Non-Dice environment, and based on that, we can conclude that our implementation has a great amount of overhead, that prevents the performance to scale equivalently with the number of available cores.

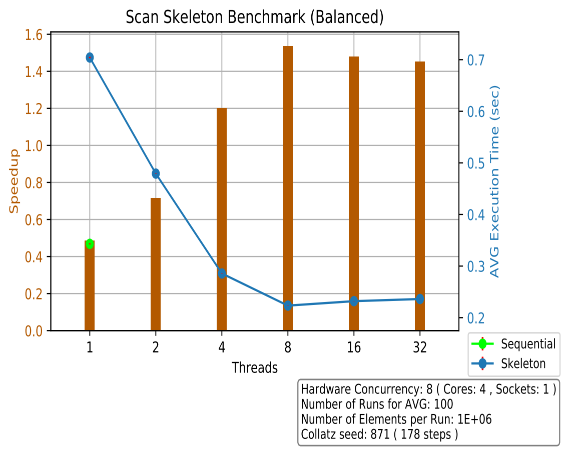
4.3.4.2 Scan: Second version (task-steal in scan and map phases)

For our second version, the test case is set the same as with the baseline version. The successor applies an addition to its arguments, and the Skeleton has been fed with 10^6 input elements.

The bar chart comparison presented in figure 4.11 for a balanced workload on a Scan operation, proves the theoretical complexity presented in section 3.5.2.2 for the second version in practice.



(a) Baseline Version



(b) Second version

Figure 4.11: Scan: Baseline (a) and second (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements.

The second plot (b) of the figure, illustrates that when we provide 1 thread to the

Skeleton, the performance is more than twice slower, relatively the sequential code's measurement. Subsequently, the implementation requires the fourfold number of cores in order to barely surpass the speed-up of 1, and twice as many cores to achieve its maximum performance gain of 1.5.

This performance is due to the fact that the "scan phase" of the algorithm, is literally a complete scan operation that is included as part in the initial one. Moreover, this is also the main reason that the average execution time from 1 to 2 threads on the second version, does not follow the same trend as in sub-figure (a) of the baseline version. Meaning that when handled by a single thread, the thread has a great overhead to complete the same task.

Based on the results, the second version does not meet the essential requirements for our parallel library, thus the baseline version was chosen for further optimisation.

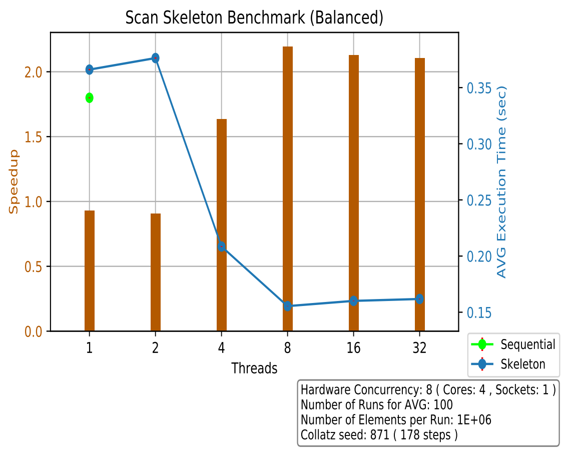
4.3.4.3 Scan: Third version (two-pass binary tree)

The third benchmark case for the Scan Skeleton, is related to the third implementation that replaces the sequential "second phase" of Scan, with a concurrent "two-pass binary tree".

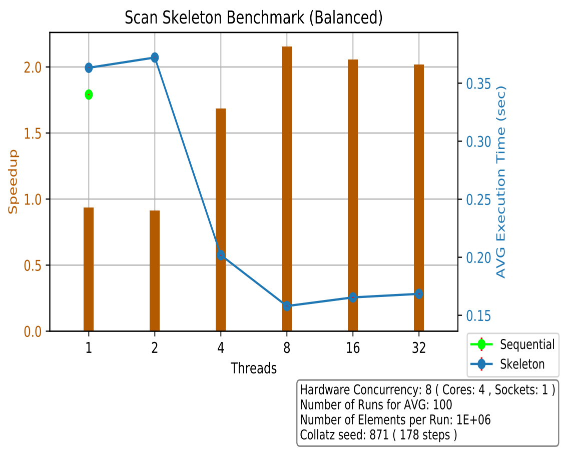
As illustrated in figure 4.12, when we use the same successor function as presented in the previous section, 10^6 input elements, and a Collatz seed that produces 178 iterations, the speed-up for baseline version (a) and third version (b) is almost identical. After a thoroughly investigation, we discovered that with this testing environment, the potential of the third version is not revealed. This is due to the fact that, the third version's enhancement, requires a significant amount of execution time per successor call to thrive, a property that this benchmark does not hold. However, it is in our benefit that the third version's improvement, does not create an overhead on such simple case.

In order to test to the point the "two-pass binary tree", firstly we redefined the successor function to create greater artificial workload per call, which is achieved by iterating the Collatz function 12,000 times, leading to 10 milliseconds per successor call (on the non-DICE machine), and secondly by initialising only 64 input elements for each run. The latter number was chosen carefully, as it serves us in specific way. This is that it keeps the same number of elements processed by the "scan" and "map" phase per chunk (8), as with the number of "chunk's last values" that the binary tree handles (8), when we deploy 8 threads (the maximum hardware concurrency on the non-DICE computer).

This way, given T the successor's execution time, for 8 threads, the "scan phase"



(a) Baseline Version



(b) Third version

Figure 4.12: Scan: Baseline (a) and third (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements.

in roughly finished after $8 \cdot T$, which is about the same with the "map phase" too. For the "tree phase", even though there are more successor invocations in total than the equivalent sequential code of baseline's version, with the introduced parallelism, the actual execution time for this phase is $2 \cdot T \cdot log(8)$, which is equal to $6 \cdot T$.

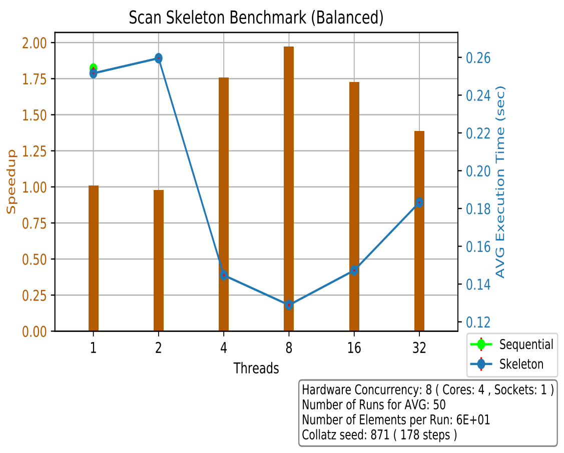
As we see in the bar charts presented in figure 4.13, the third version is marginally slower for the first 4 cases (up to 8 threads). This can be justified by taking into account the binary tree's overhead, and the almost equal (theoretical) execution time of the "second phase" for the baseline and third version, $P \cdot T$ and $2 \cdot T \cdot log(P)$ respectively (with P the number of threads).

The third version starts to win the race by the time we use more than 8 threads. By following the same analysis, with 16 threads, the sequential version requires $16 \cdot T$ for the second phase, while the third version only $2 \cdot T \cdot log(16)$, which is half as much. Moreover, with 32 threads, the time spend in the second phase is 3 times less.

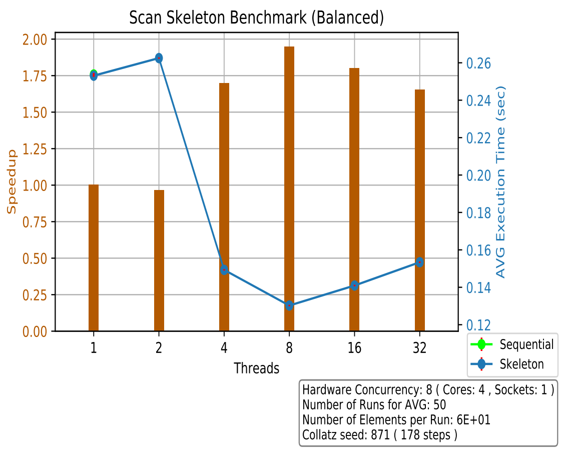
This concludes that the third version, sustains the same performance with a usage case as shown in the beginning of this section, but it pays-off when we have a problem with small number of elements, or a big number of available cores, and a time-costly successor function.

4.3.4.4 Scan: Fourth version (task-steal in map phase)

For the last benchmark of Scan Skeleton, we tested the performance of our design choice to apply the dynamic load balancing mechanism on the map phase. To test



(a) Baseline Version



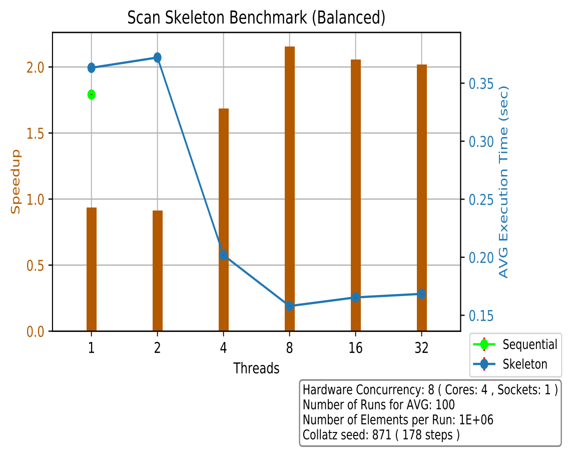
(b) Third version

Figure 4.13: Scan: Baseline (a) and third (b) version comparison, with 10 milliseconds per successor call, and 64 elements.

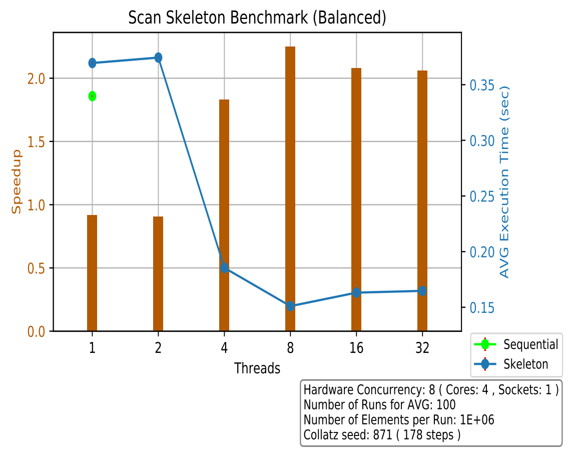
this, we applied the same technique as in the case of Map and Reduce Skeleton, which defines the successor function to invoke Collatz with a seed that produces 1 iteration for the first half of the elements, and a seed that produces 178 iterations for the other half.

As we see in figure 4.14, the speed-up comparison between the third and fourth version indicates that for an evenly distributed workload ((a) and (b)), the two versions behave the same. The bottom side of the figure illustrates the performance gain of the two versions, on an unbalanced amount of working time per successor call, and the fourth version is recorded to perform significantly better when we introduce parallelism, taking into account that the first two phases of the algorithm remain the same.

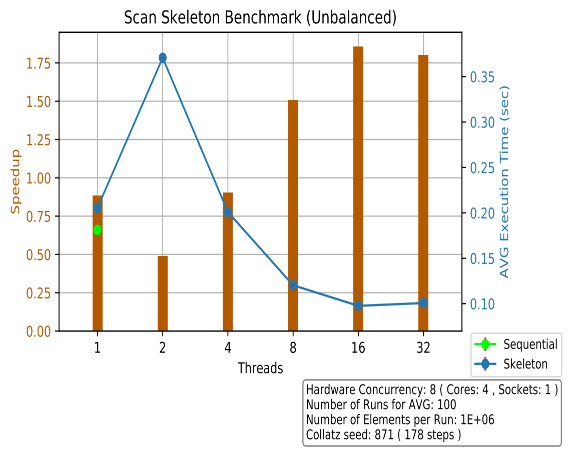
It is important to discuss the dramatic fall in speed-up occurring by the time we introduce 2 threads to the process in sub-figure (c). As explained earlier in section 4.3.4.1, by employing only 1 thread, the second and third (map) phase of the Skeleton are not activated. At the point we introduce 2 threads, the third version takes twice as long as with the use of 1 thread, due to the fact that the second half of the input, which is handled by only 1 thread, is processed twice (scan and map phase). Once we double (4) the threads that handle the prolonged half, we observe the discussed performance gain to be doubled as well.



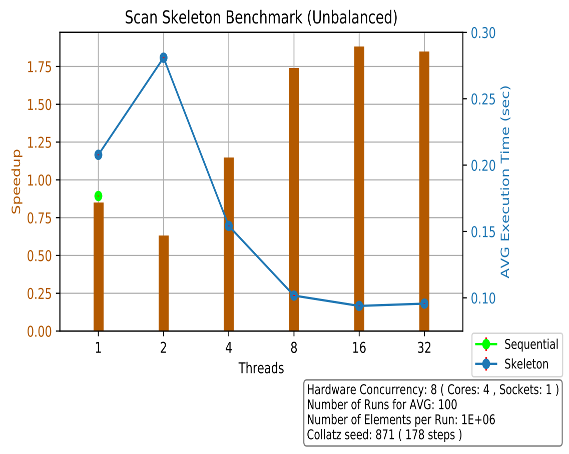
(a) Third Version (Balanced)



(b) Fourth version (Task-stealing, Balanced)



(c) Third Version (UnBalanced)



(d) Fourth version (Task-stealing, UnBalanced)

Figure 4.14: Scan: Third (a) and fourth (b) version comparison for an equal amount of workload per element (Balanced) and 10^6 elements. Sub-figures (c) and (d) present the versions' comparison for Unbalanced workload

4.4 Real World Applications

In this section we present the efficiency of our Skeleton Library, both in terms of performance and programmability, by implementing and evaluating solutions to realistic problems that carry a considerable level of complexity. The problems have been chosen to match the utility of our library, and are broadly used in computer science.

The first problem is the "K-Means clustering with Reduce and Map" and the second "K-Means clustering with MapReduce". Both problems solve the "K-Means clustering" [19], an unsupervised classification algorithm which is applied when we seek clusterization on an unlabelled dataset. This algorithm achieves the grouping of data, based on their features, into “K” constellations. The number K, along with the data, is provided by the user, and the algorithm’s outcome is firstly the centroids of each computed cluster, and secondly the related cluster ID for each data point.

The third problem is termed “Cumulation of a function” [22], which addresses the task of integrating a sub interval of a function in constant time. This is attained by creating a table that contains the definite integrals of a given function, $f(x)$, from the beginning of the sub interval and up to every sample that is defined for the independent variable x. In section 4.4.3, we present and evaluate an implementation with the aid of Map and Scan Skeletons.

Finally, for each problem, the Skeleton implementation is compared against, a sequential and a parallel implementation that makes direct use of the Pthreads library. The comparison is made both performance wise and in terms of programmability (as lines of code), where the performance (recorded on the non-DICE machine), is demonstrated as the speed-up for the Skeleton and PThreads implementation’s execution time over the sequential. For the Skeleton implementations, the last version of each module, as presented in chapter 3, has been used. Moreover, for each performance metric, the maximum (theoretical) speed-up with the use of 8 threads is indicated, based on Amdahl’s law [1]. This limit is calculated as follows:

$$Maximum\ (Theoretical)\ Speedup\ =\ \frac{S}{M(P)} \tag{4.1}$$

$$M(P)\ =\ N+\frac{S-N}{P} \tag{4.2}$$

Where S the total execution time of the sequential implementation, N the execution time of the non-parallelizable sequential part, and P the number of processes (threads) in use.

4.4.1 K-Means clustering with Reduce and Map

K-Means algorithm, as presented in figure 4.15, consists of two phases. The first is an iteration for each data-point, in order to find its closest centroid, and the second phase, depending on that at least on data-point has been assigned to a different cluster, iterates each cluster and calculates its new centroid. The termination criteria is that after the completion of the first phase, no data-point has changed cluster.

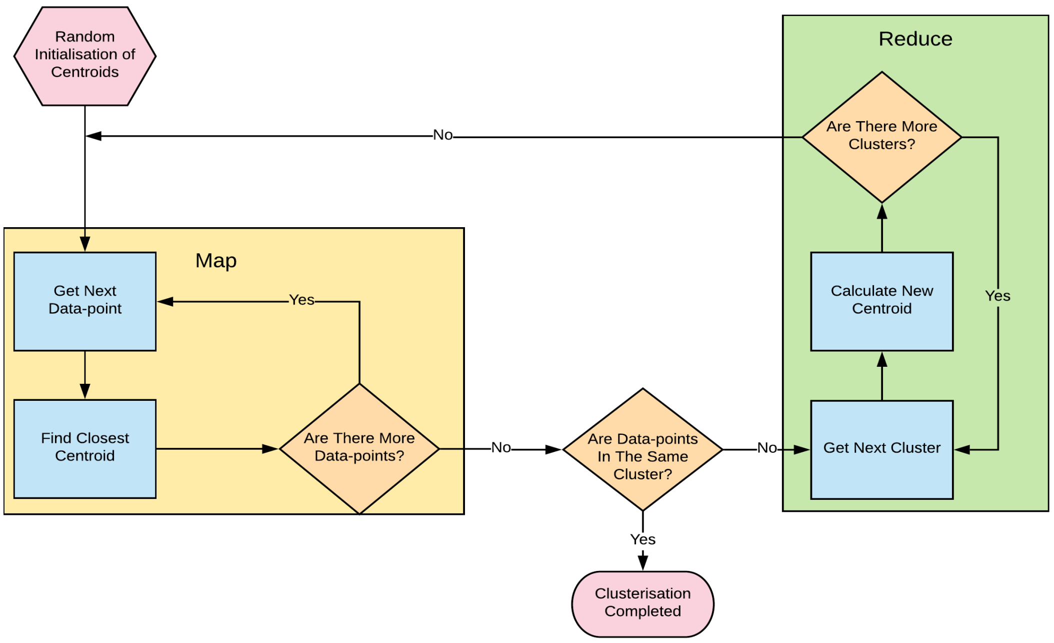


Figure 4.15: K-Means Algorithm (simplified).

For the implementation, as shown in the diagram, the fist phase can be expressed as a Map operation, and the second as a Reduce. These two operation where used as a guideline for all three solutions (Skeleton, Pthread, Sequential), due to the fact that the comparison in terms of programmability and performance, requires a common base. Thus, all three implementations obtain the result with the use of the Elemental and Combiner functions.

To explain this further, it can be argued that we could had provided a specifically optimised implementation for each case, for example the sequential version could have avoided some steps that are required for the Skeleton solution, but if this had happened, in our point of view we would had ended to compare three programs that simply provide the same result, without regard of how the solution has been achieved (our common base).

To give a counter-example, suppose we had implemented the ultimate C++ library or even a C++ compiler, that for a given C++ code, each line of that code is executed in parallel. In this case, to compare its effectiveness against a usual compiler, we would had use exactly the same C++ source file in both cases, otherwise we would had no common base. For this reason, as the Skeletons' process is expressed through these two operations, those operations are used by the the other two solutions as well.

Furthermore, since the inputs of the algorithm are the data-points, and the number of clusters (K) to be created, for our performance evaluation, we scale the number of data-point under process and the desired number of clusters, in order to measure and compare the scalability of our Skeletons. In addition, the data-points in our evaluation, are randomly selected points in a 2D plane with dimensions [(0,1),(0,1)], and so is the initial centroid of each cluster.

4.4.1.1 Results

The bar chart presented in figure 4.16, illustrates the speed-up of the Skeleton and Pthread implementations' average execution time, over the sequential. In detail, the number of clusters is represented by the horizontal axis, and each coloured bar in the grid, is correlated with the number of data-points used, either with Pthread or Skeleton solution. Moreover, the black horizontal lines above the bars, indicate the maximum speed-up that theoretically could have been achieved, based on Amdahl's law.

Similarly with the bar charts presented in the Benchmark section, on the bottom right corner we have the information of the Number of threads in use, the machine's maximum concurrency (with SMT), and the number of runs in order to compute the average execution time.

As presented in figure 4.16, almost in all cases, the implementation with direct use of the PThread library is performing better than the Skeleton solution, relatively to the maximum theoretical speed-up. The Skeleton implementation surpasses its competitor only in the last third of the horizontal axis (red bars), and with the premise that only 10^5 data-points have been processed.

Generally, both implementations (Skeleton and PThread) attain about $\frac{4}{5}$ of Amdahl's law limit with the set of 10^6 points, recording the speed-up peak (2.0) with 16 clusters, while for our small set of points (10^4), both implementations barely made it to a speed-up of 1.1 across the horizontal axis. This indicates that our parallel implementation, either with PThreads or Skeletons, is depending on the problem size to attain higher performance.

If we analyse exactly what the variables of the bar chart represent, we will realise that the horizontal axis indicates the Reduce Skeleton's ability to scale, and the number of points, the same feature for the Map Skeleton. This is due to the fact that, as illustrated in figure 4.15, the Map skeleton operates per data-point, while the Reduce Skeleton per cluster.

Based on this, the scalability of the two Skeleton implementations is proven by the bar chart, and addition to the fact that with the combination of Reduce and Map Skeletons we scored about 80 per cent of the maximum theoretical speed-up with the

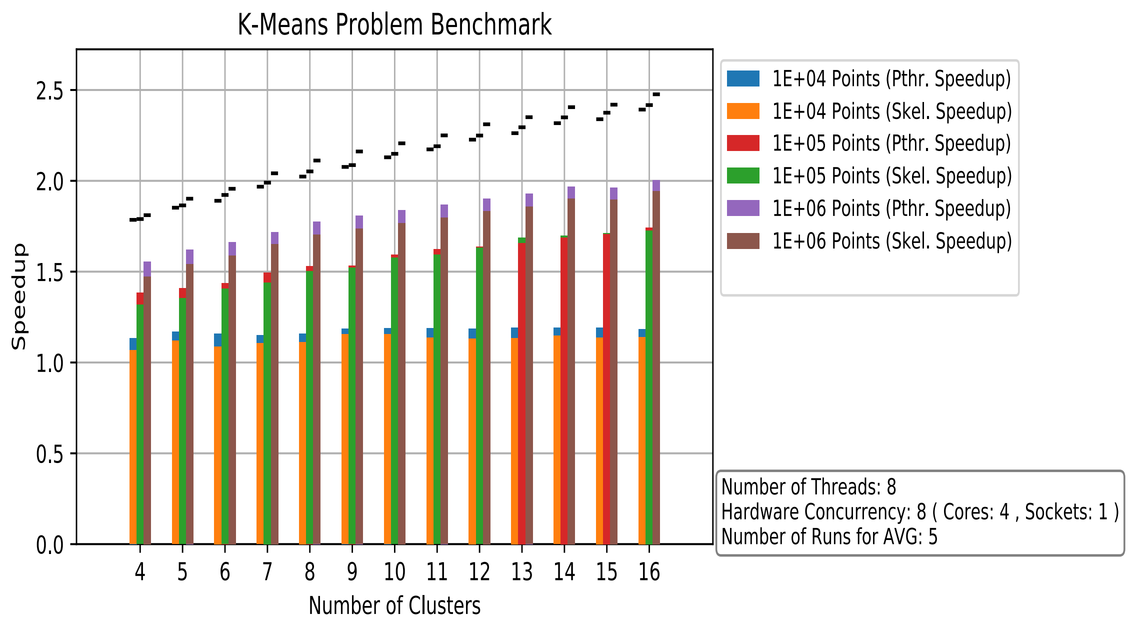


Figure 4.16: K-Means with Reduce and Map Skeletons.

Implementation	**Common Lines of Code (Elemental and Successor)**	**Total Lines of Code**
Sequential	31	57
Skeleton (Reduce and Map)		65
Explicit Use of PThreads Library		117

Table 4.2: Lines of code required from each solution.

large data set, this basic performance test for these two Skeletons is passed.

In terms of programmability, as presented in table 4.2, the Skeleton implementation is ahead by only 8 lines of code from the Sequential one (57), and the latter number is doubled (117) when we make explicit use of the PThreads library. This simple metric is an indication that besides the performance gain we accomplish with our library, the programmer has to write in this case half as much instructions, which means that the production of an erroneous software has smaller probability to occur.

4.4.2 K-Means clustering with MapReduce

With this problem, the algorithm description is the same as presented in section 4.4.1, with the only difference that both reduce and map phases of the diagram are bundled in our MapReduce Skeleton. For comparison, we used the same data as in section

4.4.1 and the same logic on how the sequential and Pthreads implementation produce their result, with the difference that now they make use of the Mapper and Reducer functions.

4.4.2.1 Results

In contrast with the solution provided with Reduce and Map Skeletons, MapReduce, the Skeleton that is a perfect match for K-Means, scores a poor performance. As shown in figure 4.17, the Skeleton only manages to achieve, 20-40 per cent of the optimum speed-up (based on Amdahl's law) across all metrics. This is two to four times slower than the Solution provided in the previous section, which incorporates two Skeletons.

Granted that, it should be noted that since MapReduce's K-Means implementation actually bundles the whole algorithm of each iteration in a single skeleton call, the execution time of the non-parallelizable sequential part (N), is almost next to nothing relatively to the total sequential time. This information justifies why the theoretical performance gain limit, is close to the number of available threads.

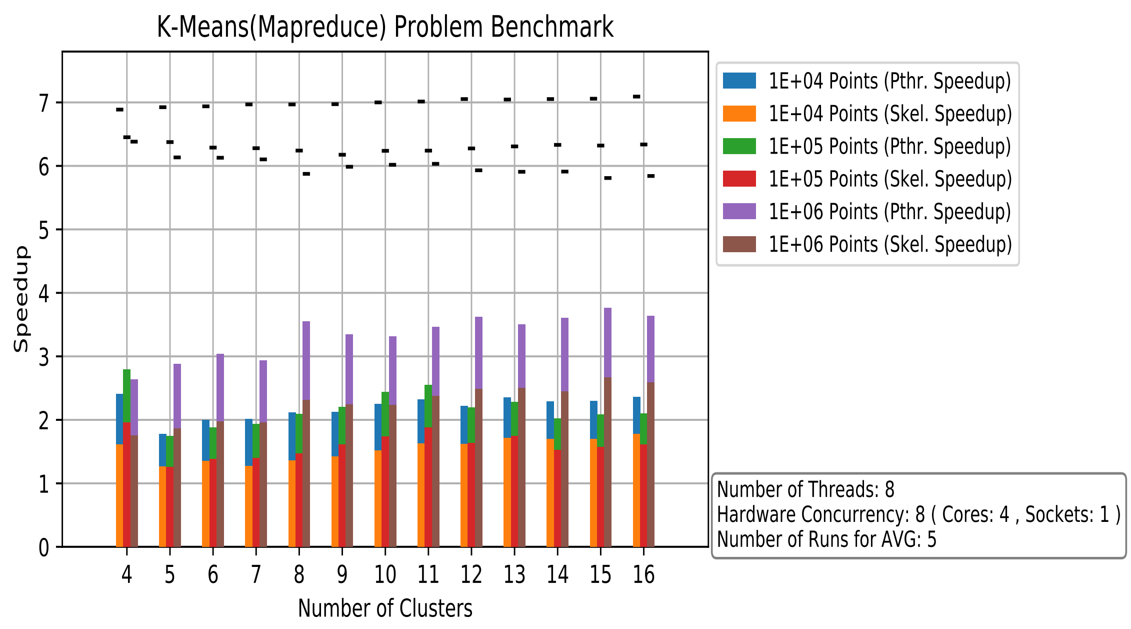


Figure 4.17: K-Means with MapReduce Skeleton.

Moreover, on the larger benchmarks (10^6 Points), MapReduce attains roughly two thirds of the PThreads' performance, a difference that is getting slightly smaller when we process less data-points. More specifically, the highest speed-up occurs by PThreads at 3.8, for 10^6 data-point and 15 clusters, while MapReduce has approxi-

Implementation	**Common Lines of Code (Elemental and Successor)**	**Total Lines of Code**
Sequential	27	70
Skeleton (MapReduce)		58
Explicit Use of PThreads Library		146

Table 4.3: Lines of code required from each solution.

mately 2.8 for the same conditions. The biggest difference of speed-up takes place with the same number of points, but with 8 clusters, where MapReduce is measured to hold a speed-up of 2.2, while PThreads obtain a mark of 3.5.

The data on table 4.3 indicate once more, that compared to a parallel implementation without Skeletons, the amount of code is far greater. Specifically, with our library we are required to provide almost three times less code, and for this specific problem, this amount of work is even less than the sequential code.

We can conclude that MapReduce indeed provides a simplified interface for the two operations (Reduce and Map), which allows us to program with greater ease, but performance-wise, the Skeleton's overhead is clearly present, and in need of further optimisation to be able to be compared with the Skeletons that it combines.

4.4.3 Cumulated Function with Map and Scan

In our last section of this Chapter, we will demonstrate the effectiveness of Scan and Map Skeletons, by implementing an algorithm that computes the cumulation of the function in the domain of [1,10]. The function is defined in equation 4.3 , and its graph is shown in figure 4.18. The algorithm of the method is presented in figure 4.19, and as illustrated, it can be implemented directly with use of the forenamed Skeletons.

$$f(x) = \sin(2x) \cdot \log(x+1) + \frac{x}{8}; \tag{4.3}$$

Basically, this algorithm produces a table (summed-area table), where each record holds the area of the function, up to the subinterval that the record represents, and in our case, we use the Simpson method [16] for the numerical integration of the subinterval. After computing all records of the table, we can approximate any subinterval of the function in constant time, by subtracting the area-record that correlates to the lower limit of our integral, from the area-record of the upper limit.

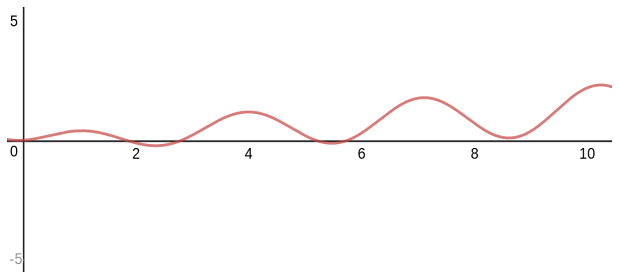


Figure 4.18: The function's graph.

The sequential algorithm consists of two steps. The first one, is an iteration that computes the area of each subinterval, and the second one, applies a prefix-sum operation on the result of the first step. The Skeleton implementation, executes the former step with a Map operation, and with a Scan operation the latter. The equivalent parallel implementation is given to our third solution that makes explicit use of the PThreads library.

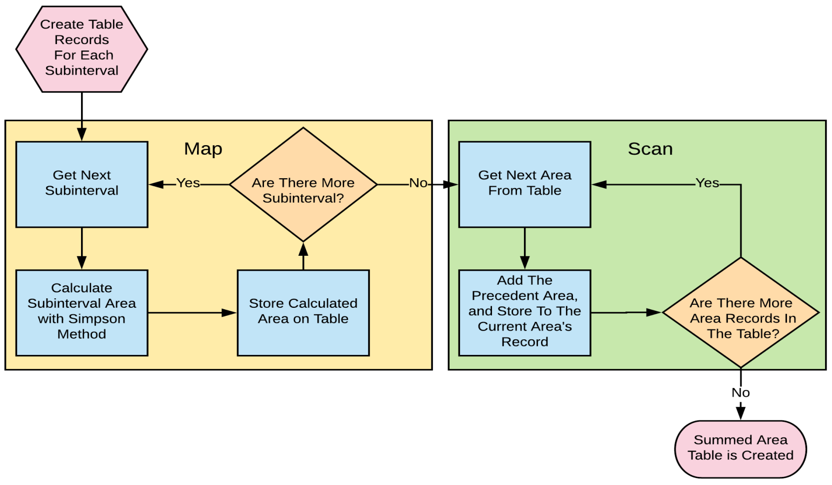

Create Table Records For Each Subinterval
Map
Get Next Subinterval
Yes
Are There More Subinterval?
No
Calculate Subinterval Area with Simpson Method
Store Calculated Area on Table
Scan
Get Next Area From Table
Yes
Add The Precedent Area, and Store To The Current Area's Record
Are There More Area Records In The Table?
No
Summed Area Table is Created

Figure 4.19: Cumulated Function Algorithm (simplified).

For the evaluation, we scale the problem in two dimensions. The first one is the size of the summed-area table, and the second one is the number of sub-divisions the Simpson method creates to compute the subinterval's area. The scaling of the former dimension introduces greater workload for the Scan Skeleton, while the latter has the same impact for the Map Skeleton.

The results presented in figure 4.19, indicate that on average a speed-up of 4 is achieved. Based on Amdahl's law, this is 50 per cent of the upper bound speed-up, thus this performance of the Skeletons can be described as moderate.

In comparison with direct use of the PThreads library, the performance gain of the

Implementation	**Common Lines of Code (Elemental and Successor)**	**Total Lines of Code**
Sequential	12	22
Skeleton (Map and Scan)		21
Explicit Use of PThreads Library		116

Table 4.4: Lines of code required from each solution.

Skeleton implementation is better for the first three sets of the summed-area table's size and with 10^3 sub-intervals, while for the remaining sets in the same category the performance is almost the same. Moreover, a similar trend occurs by defining 10^2 subintervals for the Simpson method, with the difference being marked with a summed-area table of size 10^4, where the Skeleton implementation has outperformed its competitor by a speed-up factor of 1.

From the bar chart, we can conclude that as the problem size rises, so does the speed-up of each Skeleton. This is based on the fact that while we increase the number of Simpson subintervals, the speed-up of the Map Skeleton is increased as well, and the same trend, but with a lower growth, is taking place when we scale the size of the table, which relates to Scan Skeleton.

Finally, as presented in table 4.4, the Skeleton implementation requires almost 6 times less code to be written compared to the PThreads implementation, and has about the same lines with the sequential implementation.

We can conclude that even though Scan and Map Skeleton provide a good level of programmability, based on our simple criterion, the implementation of Map, and especially of Scan skeleton, require further optimisation to attain a greater degree of performance, based on this evaluation case.

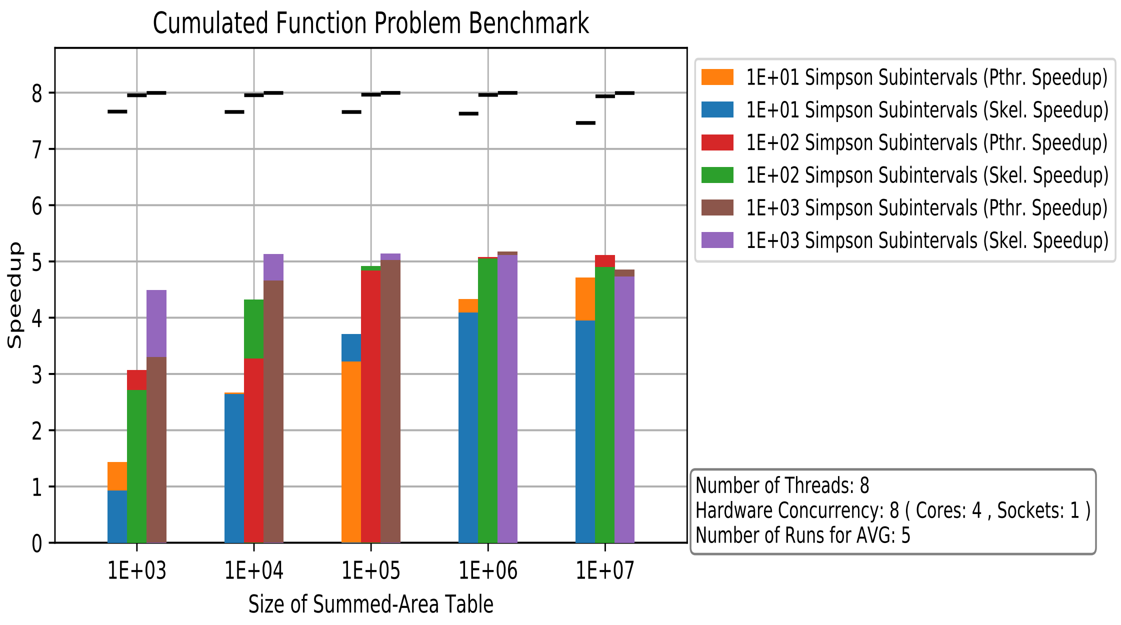


Figure 4.20: Cumulated function performance of Skeleton and PThreads implementations.

Chapter 5

Conclusions and Future Work

This thesis has presented a type-safe, shared-memory parallel Skeleton library, that provides four data-parallel Skeletons, specifically Map, Reduce, MapReduce, and Scan, and allows the development of structured parallel programs, with the only system requirement being a GCC compiler (minimum version 4.8). Therefore, the principal goal of the project, that requests a Skeleton library with the capability to be fully operational on a DICE environment, has been accomplished.

Based on the interface design choices, algorithms, development process, and evaluation of each Skeleton, our assessment of the project indicates firstly that for our secondary criterion - degree of programmability, the library as compared with a non-Skeleton parallel implementation, attains a decent level of the metric, relative to sequential programming.

However, based on our primary criterion - performance, the metrics reveal that the first two Skeletons (Map and Reduce), achieve great performance gain on simple benchmarks, with respect to the available physical processors, while for the last two Skeletons (MapReduce and Scan), based on the same benchmarks suite, the performance can be described as moderate. Moreover, when greater algorithmic complexity is introduced, as with our "real-world" applications, when we combine our Skeletons to solve those problems, the performance is significantly inferior, especially with the evaluation case of MapReduce on the KMeans algorithm.

Its is also important to be mentioned, that the evaluation of the Skeletons' versions, should have been conducted with more precision. Specifically, we believe that the "Thread-pool" versions' benchmarking, for Map and Reduce, was not adequate to reveal the actual potential of the implementation. Therefore, our decision during development stage to reject this enhancement might have been misguided. In addition, the “Dynamic load balancing” versions should have been examined with a variety of data-block sizes, and the unbalanced-benchmarks should have had included test cases with a different distribution of the heavy workload. These notes will be addressed in our future work.

Our future plan, beyond the aforementioned notes, includes the creation of another data-parallel algorithmic Skeleton - Stencil, and a the introduction of a task-parallel algorithmic Skeleton - Pipeline. In addition, we will include performance comparison of our work with our source of inspiration, the SkePU2 Skeleton library.

Bibliography

[1] 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*, pages 483–485. ACM, 1967.

[2] Ş. Andrei and C. Masalagiu. About the collatz conjecture. *Acta Informatica*, 35(2):167–179, 1998.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. Summarising an experiment in parallel programming language design. In *International Conference on High-Performance Computing and Networking*, pages 7–13. Springer, 1995.

[4] G. E. Blelloch. Pre x sums and their applications. Technical report, Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[5] P. Ciechanowicz, M. Poldner, and H. Kuchen. The münster skeleton library muesli: A comprehensive overview. Technical report, Working Papers, ERCIS-European Research Center for Information Systems, 2009.

[6] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[7] M. I. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[8] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[9] M. Danelutto and M. Stigliani. Skelib: parallel programming with skeletons in c. In *European Conference on Parallel Processing*, pages 1175–1184. Springer, 2000.

[10] J. Darlington and H. To. Building parallel applications without programming. *JR Davy end PM Dew, editor, Abstract Machine Models for Highly Parallel Computers*, pages 140–154, 1993.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] DICE. *www.dice.inf.ed.ac.uk. (2018) Informatics Computing - School of Informatics - University of Edinburgh*, 2018 (accessed August 2018).

[13] J. Enmyren and C. W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[14] A. Ernstsson, L. Li, and C. Kessler. SkePU2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1):62–80, feb 2018.

[15] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Pract. Exper*, 40:1135–1160, 2010.

[16] P. E. Hennion. Algorithm 84: Simpson's integration. *Communications of the ACM*, 5(4):208, 1962.

[17] C. A. Herrmann and C. Lengauer. : A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(02n03):239–250, 2000.

[18] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] S. Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[20] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[21] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st international conference on Scalable information systems*, page 13. ACM, 2006.

[22] M. D. McCool, A. D. Robison, and J. Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes*, page 16. ACM, 2008.

[24] R. Schaller and R. R. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, jun 1997.

[25] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, may 2010.