Parallel Implementation of an Exact Filter Design Optimization Algorithm on Distributed-Memory Systems

Élio António Figueiredo dos Santos

Thesis to obtain the Master of Science Degree in
Electrical and Computer Engineering

Supervisor(s): Prof. Paulo Ferreira Godinho Flores
Prof. José Carlos Alves Pereira Monteiro

Examination Committee
Chairperson: Prof. Nuno Cavaco Gomes Horta
Supervisor: Prof. Paulo Ferreira Godinho Flores
Member of the Committee: Prof. Nuno Filipe Valentim Roma

November 2016
Dedicated to my parents, Melita and Tony.
Acknowledgements

Writing this note of thanks is the finishing touch on my thesis. It has been a long period of intense learning for me, not only in the scientific arena, but also on a personal level. Writing this thesis has had a big impact on me. I would like to reflect on the people who have supported and helped me so much throughout this period.

I can only begin by expressing my gratitude to my parents, António dos Santos and Amélia de Figueiredo, who are the reason I am here today. They always supported me in my aspirations and contributed so that I could study here, so far away from home, so that I could conclude such an important phase of my live. They have truly been a pillar to me.

I would also like to express my special thanks of gratitude to my supervisors, Prof. José Monteiro and Prof. Paulo Flores, who gave me the golden opportunity to do this thesis on the topic Parallel Implementation of an Exact Filter Design Optimization Algorithm in Distributed Memory Systems. They supported from the beginning and were always available to help, allowing me to learn and achieve great results. I am truly thankful to them and their valuable guidance.

In addiction, I would like to thank my girlfriend, Mayara Mejias, for the continuous support, especially during the long nights. Being with her is one of my biggest achievements and she makes all the effort worth the while. Also my little sister, Gaby, whom I share so much, from childhood stories to toughs and ideals. She makes me always want to be better and better. She keeps me on track, and makes me want to keep doing a good job and making a good example. And Bruno Figueiredo, my best friend, cousin, brother from another mother, true companion and rival. Now, we share even a degree, which we are finishing at the same time. I can only say I hope we can achieve our dreams together.

I have a big family, so I want to express my gratitude to every single one of them. My aunts, my uncles, my cousins, my brother, my sister Rita and her two beautiful daughters, who represent the hope I have for the next generation. Finally my friends. We were able to support each other, and I hope we will continue to do so for a long period of time.

Thank you very much, everyone!

Élio Figueiredo dos Santos

Lisboa, December 8, 2016
Abstract

The filter design optimization (FDO) problem consists in finding a set of coefficients that meets pre-established filter constraints to yield a filter design with the least design complexity. SIREN is an exact FDO algorithm which assumes that the coefficient multiplications in the filter design are realized under a shift-adds architecture. It can guarantee the minimum design complexity, but can only be applied to filters with a small number of coefficients. Since SIREN uses a depth-first search (DFS) method, it can be optimized through a parallel implementation, in order to enable its application to larger filters.

This thesis presents the proposed solutions for the parallel implementation of the SIREN algorithm on distributed-memory systems using MPI, and discusses the results obtained and their limitations. Since SIREN's search space is very unbalanced, the workload has to be divided dynamically, to be distributed as evenly as possible amongst the available processors. As such, three different load balancing strategies are presented: Liberal Work Forwarding (LWF), Quasi-Liberal Work Forwarding (QLWF) and Fixed Depth Breadth-First Search (FDBFS) load balancing strategies.

The LWF load balancing strategy proved to be the best strategy. With this load balancing strategy, the CPU times were greatly improved and better speed-ups were obtained. Still, some limitations regarding the scalability of the optimized algorithm were found. With the parallel implementation, the SIREN algorithm becomes faster and can compute filters with a larger number of coefficients (up to 60), in reasonable amount of times.

Keywords: Parallel depth-first search, distributed computing, MPI, filter design optimization problem, finite impulse response filter, multiplierless design.
Resumo

O problema da otimização do projeto de filtros (FDO do inglês) consiste em encontrar um conjunto de coeficientes que satisfazem restrições pré-estabelecidas para implementar filtros com complexidade mínima. SIREN é um algoritmo exato de FDO que assume que as multiplicações dos coeficientes no filtro são realizadas segundo uma arquitetura de deslocamento-soma (shift-adds). O algoritmo consegue garantir que o filtro projetado tenha complexidade mínima, mas só pode ser aplicado a filtros com um número reduzido de coeficientes. Uma vez que o SIREN usa um método de procura em profundidade (DFS do inglês), pode ser paralelizado e otimizado para ser aplicável a filtros maiores.

Esta tese apresenta as propostas para a paralelização do algoritmo SIREN num sistema distribuído usando MPI, e discute os resultados e limitações obtidos. Sendo o espaço de procura do SIREN muito desequilibrado, o trabalho deve ser dividido dinamicamente, de forma que a carga computacional seja distribuída de forma balanceada pelos processadores disponíveis. Assim, três estratégias de balanceamento de carga são apresentadas: Encaminhamento de Trabalho Liberal (LWF), Encaminhamento de Trabalho Quasi-Liberal (QLWF) e Busca em Largura com Profundidade Fixa (FDBFS).

Os melhores resultados foram obtidos com a estratégia de balanceamento de carga LWF. Com esta estratégia, os tempos de CPU do algoritmo foram melhorados consideravelmente e foram obtidos os melhores speed-ups. Porém, algumas limitações relativas à escalabilidade do algoritmo foram encontradas. Em suma, foi possível otimizar o algoritmo SIREN para ser mais rápido e realizar em tempos razoáveis filtros com um maior número de coeficientes (até 60).

Palavras-Chave: Paralelização de procura em profundidade, computação em sistemas distribuídos, MPI, problema da otimização do projeto de filtros, filtro de resposta ao impulso finita, design sem multiplicadores.
# Contents

Acknowledgements

Abstract

Resumo

List of Tables

List of Figures

Acronyms

## 1 Introduction

1.1 Motivation

1.2 Objectives

1.3 Thesis Outline

## 2 Filter Design Optimization Problem

2.1 Filter Design Optimization Algorithms

2.2 Multiple Constant Multiplications

## 3 Parallel Computing

3.1 MPI - Message Passing Interface

3.2 Parallel Algorithms for DFS

## 4 SIREN

4.1 Linear Programming

4.2 SIREN Description

4.3 SIREN Depth-First-Search Method

## 5 SIREN Parallelization

5.1 Starting Point

5.2 Proposed Solution

5.3 Implementation
5.4 Load Balancing .......................................................... 26
  5.4.1 Liberal Work Forwarding ........................................ 26
  5.4.2 Quasi-Liberal Work Forwarding ............................ 31
  5.4.3 Fixed Depth BFS ................................................. 32
5.5 Summary on the Experimental Results ......................... 35

6 Conclusions ................................................................. 39
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of CPU Times of SIREN - MATLAB vs C version.</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Specifications of random Symmetric FIR filters used during the testing phase.</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Specifications of benchmarks Symmetric FIR filters.</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of FDO Algorithms on FIR Filters of Table 5.3.</td>
<td>37</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Transposed form FIR filter design .................................................. 1

2.1 Zero-phase frequency response of a low-pass FIR filter. .................. 6

2.2 Multiplierless realization of constant multiplication using the DBR technique [1]: 21x and 53x. .............................................................. 8

2.3 Multiplierless realization of 21x and 53x: (a) exact CSE algorithm [2]; (b) exact GB algorithm [3]; (c) approximate GB algorithm [3] modified to handle a delay constraint. .................. 8

4.1 Search tree formed by the DFS method. ................................. 18

5.1 Speed-Up of Liberal Work Forwarding Load Balancing .................. 28

5.2 Average Speed-Up of Liberal Work Forwarding Load Balancing ....... 29

5.3 Percentages of Idle Time for each processor on a filter with N=43 ....... 30

5.4 Average Percentage of Idle Time for Liberal Work Forwarding Load Balancing ........... 31

5.5 Speed-Up of Quasi-Liberal Work Forwarding ............................... 32

5.6 Average Speed-Up of Quasi-Liberal Work Forwarding .................. 32

5.7 Speed-Up of Fixed Depth BFS Load Balancing ............................... 34

5.8 Average Speed-Up of Fixed Depth BFS Load Balancing .................. 34

5.9 Average Speed-Up for all the Load Balancing Strategies Implemented ................................. 36
Acronyms

BFS  Breadth-First Search.
CSD  Canonical Signed Digit.
CSE  Common Subexpression Elimination.
DFS  Depth-First Search.
DSP  Digital Signal Processing.
DTTDA  Dijkstra Token Termination Algorithm.
EWL  Effective Wordlength.
FDBFS  Fixed Depth BFS.
FDO  Filter Design Optimization.
FIR  Finite Impulse Response.
GB  Graph-Based.
HPC  High-Performance Computing.
IEEE  Institute of Electrical and Electronics Engineers.
IIR  Infinite Impulse Response.
ILP  Integer Linear Programming.
ISO  International Organization for Standardization.
LP  Linear Programming.
LWF  Liberal Work Forwarding.
MCM  Multiple Constant Multiplication.
MDTTDA  Modified Dijkstra Token Termination Algorithm.
MILP  Mixed Integer Linear Programming.

MIMD  Multiple Instruction, Multiple Data.

MPI   Message Passing Interface.

NPR   Normalized Peak Ripple.

QLWF  Quasi-Liberal Work Forwarding.

SFDT  Straightforward Filter Design Technique.
Chapter 1

Introduction

Digital filtering is one of the most important operations in digital signal processing (DSP). It is usually realized using infinite impulse response (IIR) filters or finite impulse response (FIR) filters. In DSP applications, a FIR filter is usually preferred over an equivalent IIR filter, due to its output stability and phase linearity properties [4], despite requiring a larger number of coefficients.

The output of an N-tap FIR filter $y(n)$ is computed as:

$$y(n) = \sum_{i=0}^{N-1} h_i \cdot x(n - i) \quad (1.1)$$

where $N$ is the filter length, $h_i$ is the $i^{th}$ filter coefficient and $x(n - i)$ is the $i^{th}$ previous filter input.

The expression in (1.1) can be realized in a straightforward way, known as direct form, or in its transposed form. In this work we are only interested in the transposed form realization of (1.1), which is shown in Figure 1.1. Here, the design complexity is dominated by the multiplications of the coefficients by the filter input, known as the multiple constant multiplications (MCM) block.

![Figure 1.1: Transposed form FIR filter design](image)

The design of a FIR filter can be performed in two steps. First, obtain a set of coefficients, that respects the filter constraints, and second, design the multiplier block of the filter.

Since the filter coefficients are determined beforehand and fixed, and the realization of a multiplier in hardware is expensive in terms of area, delay and power, the MCM operations (multiplier block of the FIR filter) are generally implemented using only shifts, adders and substracters, in a shift-adds architecture [5]. Note that shifts by a constant value can be implemented using only wires representing no hardware cost.

Thus, a well known problem [6] is defined as: given a set of constants to be multiplied by an input
variable, find the minimum number of adders/subtracters that realize the constant multiplications. This is an NP-complete problem, even in the case of a single constant multiplication [7].

In this context, the second step of the design of a multiplierless FIR filter, i.e., the design of the multiplier block, can be done under a shift-adds architecture using an MCM algorithm. There are many efficient MCM algorithms that aim to maximize the sharing of partial products among the constant multiplications, targeting the minimization of the number of operations and also the optimization of gate-level area, delay, throughput and power dissipation of the design [2,3,6,8,9].

Finally, the filter design optimization (FDO) problem is defined as finding a set of filter coefficients satisfying the filter constraints which yields a filter design with minimum complexity. It can also be defined as: given the filter specification $f_{\text{spec}}$, denoted as a five-tuple (filter length $N$, passband $\omega_p$, and stopband $\omega_s$ frequencies, and passband $\delta_p$ and stopband $\delta_s$ ripples), find a set of coefficients that satisfies the filter constraints leading to a filter design with minimum number of adders/subtracters.

### 1.1 Motivation

Due to the widespread application of filters, the FDO problem has received a tremendous interest and consequently many algorithms to find the desired set of coefficients have been developed.

SIREN [10] is an exact FDO algorithm that exhaustively searches for the filter coefficients, using a depth-first search (DFS) method, guaranteeing the minimum design complexity under a minimum coefficient bit-width. However, it can only be applied to filters with a small number of coefficients. Another algorithm, NAIAD [10], is an approximate algorithm with the same objectives as SIREN, but can handle filters with a larger number of coefficients using less computational resources and can find better solutions than other existing FDO heuristics.

Since SIREN, unlike NAIAD, is an exact algorithm and as such provides the best solutions (minimum number of adders/subtracters that satisfy the constraints), any attempt to improve it by diminishing the limitations, in terms of computational time and number of filter coefficients, is very appealing and justified. Moreover, the algorithm shows characteristics fit for parallelization, namely the fact that the search space is explored using a DFS method, for which there are several parallelization techniques.

As referred before, an FDO problem consists in finding an optimum set of coefficients. With a larger number of coefficients it is possible to describe a FIR filter closer to the desired (ideal) response. However, existing algorithms can only deal with a limited number of coefficients, so it makes sense that the parallelization of the algorithm is done using distributed-memory systems, which can allow significant scalability and as such diminish the limitations on the size of the filter.

Distributed computing can be defined as a collection of distinct computer resources working to reach a common goal, where the communications and coordination of the actions are done through messages. Each node/machine is set to perform a different task/application, so the workload is divided amongst them. Due to these aspects, distributed computing is highly scalable.
1.2 Objectives

In this context and given the algorithms and results obtained in [10], this work aims to optimize SIREN, an exact FDO algorithm, by resorting to distributed computing, and compare the obtained results with other existing algorithms to conclude on its benefits. Hence, it is performed a comprehensive study on the parallelization problem of the SIREN algorithm and it is proposed a solution, using message passage interface (MPI) to perform distributed computing.

The basic approach to parallelize a DFS algorithm with an unbalanced search space/tree is to split it in subtrees amongst each processor. Since the search tree is unbalanced, the load balancing has to be done dynamically, where each processor sends/requests work to the others.

This report discusses and presents the results of three different solutions/strategies for the load balancing problem and concludes which one is the best. Two of this strategies use only the DFS method, where each processor sends/receives work to/from the next/previous processor, respectively. The other strategy is a combination between Breadth First Search (BFS) and DFS methods, in order to improve the load balancing from the beginning of the computation.

1.3 Thesis Outline

The report is organized as follows. Chapter 2 presents the state of the art, background concepts and related work regarding the FDO problem. Chapter 3 presents a background on parallel computing using MPI and parallelization of DFS algorithms. Chapter 4 describes the SIREN algorithm and its DFS method. Chapter 5 presents the proposed solution for the parallelization, including three different load balancing strategies and the results obtained for each case. Finally, the conclusions on which is the best strategy and its benefits are presented in Chapter 6.
Chapter 2

Filter Design Optimization Problem

This chapter gives the background concepts and presents an overview of the algorithms for the FDO problem and the methods for the shift-adds design of the MCM block.

2.1 Filter Design Optimization Algorithms

The zero-phase frequency response of a symmetrical FIR filter, is given as

\[ G(\omega) = \sum_{i=0}^{\lfloor M \rfloor} d_i h_i \cos(\omega(M - i)) \cdot x(n - i) \]  

(2.1)

where \( M = \frac{(N - 1)}{2} \) and \( d_i = 2 - K_{i,M} \) with \( K_{i,M} \) as the Kronecker delta\(^2\), \( h_i \in \mathbb{R} \) with \(-1 \leq h_i \leq 1\), and \( \omega \in \mathbb{R} \) is the angular frequency. The zero-phase frequency response of a low-pass FIR filter is shown in Figure 2.1. Assuming the desired pass-band and stop-band gains equal to 1 and 0, respectively, a low-pass FIR filter must satisfy the following constraints \[12\]:

\[ 1 - \delta_p \leq G(\omega) \leq 1 + \delta_p, \quad \omega \in [0, \omega_p] \]

\[ -\delta_s \leq G(\omega) \leq \delta_s, \quad \omega \in [\omega_s, \pi] \]  

(2.2)

where \( \omega_p \) and \( \omega_s \) represent, the passband and stopband frequencies, respectively, and \( \delta_p \) and \( \delta_s \) denote, respectively, the passband and stopband ripples (see Figure 2.1). The pass-band gain is not relevant for many DSP applications and can be compensated in the filter design. Thus, a scaling factor \( s \) can be added into the filter constraints as a continuous variable, where \( s^l \) and \( s^u \) are the lower and upper bounds of \( s \), respectively \[13, 14\]:

\[ s(1 - \delta_p) \leq G(\omega) \leq s(1 + \delta_p), \quad \omega \in [0, \omega_p] \]

\[ s(-\delta_s) \leq G(\omega) \leq s(\delta_s), \quad \omega \in [\omega_s, \pi] \]

\[ s^l \leq s \leq s^u \]  

(2.3)

\(^1\)The frequency response of an asymmetric filter is found in [11]

\(^2\)The \( K_{a,b} \) function is 1 when \( a \) is equal to \( b \). Otherwise is 0.
There are many FDO algorithms with different objectives, whether it is to obtain the least number of filter coefficients or the least computational time. A straightforward filter design technique (SFDT) consists in two steps:

i) given the filter frequency and amplitude constraints, find the coefficients that respect the constraints, using a filter design method, such as windowing [15], McClellan-Parks-Rabiner algorithm [16], or linear programming [17];

ii) realize the multiplier block of the FIR filter, using the minimum number of adders/subtracters, which will be described in Section 2.2.

FDO algorithms incorporate sophisticated techniques to explore the possible set of coefficients that satisfy the filter constraints. They include local search [18–20], and exhaustive search methods, including branch-and-bound [12, 13, 21], DFS [22, 23], and MILP [24–26] techniques. The local search methods can be applied to high-order filters, but cannot ensure the optimal solution, since they do not explore the entire search space. On the other hand, the exhaustive search methods can only be applied to low-order filters due to the exponential growth of the search space [12, 22, 23].

Some FDO algorithms [12, 18, 21, 24–27] reduce the complexity of the filter design by searching for coefficients with the minimum number of nonzero digits, since a coefficient represented with a few nonzero digits requires fewer number of operations. After a solution is obtained, the algorithms of [12, 27] find the common partial products using one of the algorithms described in Section 2.2. However, these methods may yield filters with a large number of operations, since the sharing of partial products is not considered during the search of coefficients.

The methods of [13, 14, 20, 22, 23] search for coefficients that exploit the partial product sharing. The algorithm of [14] uses an MILP method to consider all possible coefficients satisfying the filter constraints and finds coefficients that include the most common nonzero digits. [13] uses a branch-and-bound algorithm and a CSE heuristic to share the common subexpressions among the coefficient
multiplications. The method of [20], uses a canonical signed digit (CSD) representation, and finds the coefficients that include the most 101 and 10¯1 digit patterns (subexpressions) and satisfy the filter constraints. The algorithm of [22] uses a subexpression basis set that is dynamically expanded as coefficients are synthesized at each depth of its search tree. The technique of [23] is based on the algorithm of [22], but explores the entire search space under a given coefficient bit-width, known as quantization value. It is able to be aware of whether an optimum solution is obtained.

The SIREN and NAIAD algorithms [10] follow the SFDT, since they aim to determine the filter coefficients that respect the filter constraints, and, at the same time, they aim to minimize the hardware, by reducing the number of adder-subtracters. Other filter design algorithms [28] commonly used in MATLAB design filters given the frequency and amplitude constraints, but are less concerned in terms of hardware, since the designs are not optimized in terms of area (adder-subtracters).

2.2 Multiple Constant Multiplications

The constant multiplications of an MCM block (see Figure 1.1) are implemented in the form of \( y_0 = h_0 x, y_1 = h_1 x, ..., y_{N-1} = h_{N-1} x \), where \( x \) denotes the filter input. A straightforward technique for their shift-adds design is called digit-based recording (DBR) [1]. This method begins with the definition of the constants under a number representation, like canonical signed digit (CSD) [8] or binary. After, for the nonzero digits in the representation of constants, the variables are shifted according to the digit positions and then added/subtracted with respect to the digit values. As a simple example, consider \( h_0 = 21 \) and \( h_1 = 53 \) in CSD representation [10]. The decompositions of the constant multiplications \( y_0 = 21x \) and \( y_1 = 53x \) in CSD are listed as:

\[
\begin{align*}
y_0 &= 21x = (10101)_{CSD}x = x \ll 4 + x \ll 2 + x \\
y_1 &= 53x = (10\bar{1}0101)_{CSD}x = x \ll 6 - x \ll 4 + x \ll 2 + x
\end{align*}
\]

which lead to a design of 5 operations, as shown in Figure 2.2.

To reduce the complexity of the MCM design even further, the partial products among the constant multiplications can be shared. For the shift-adds design of the MCM block, the existing methods can be grouped in two categories: common subexpression elimination (CSE) [2, 8, 29] and graph-based (GB) [3, 6, 30] techniques. In the CSE algorithms, constants are initially defined under a particular number representation. Then, the “best” subexpression, generally the most common, is chosen considering possible subexpressions that can be extracted from the nonzero digits in the constants representation. On the GB methods, there are no restrictions to any particular number representation and the intermediate partial products that enable the realization of the constant multiplication are found with the minimum number of operations. The GB methods consider a larger number of realizations of a constant and may obtain better results than the CSE method, but have higher computational complexity.

For the MCM example [10], the exact CSE algorithm [2] finds the most common subexpression.
Figure 2.2: Multiplierless realization of constant multiplication using the DBR technique [1]: \(21x\) and \(53x\).

Figure 2.3: Multiplierless realization of \(21x\) and \(53x\): (a) exact CSE algorithm [2]; (b) exact GB algorithm [3]; (c) approximate GB algorithm [3] modified to handle a delay constraint.

\[5x = (101)_{\text{CSD}}x\] (Figure 2.3(a)), with constants defined under CSD, to obtain the minimum solution with 4 operations. The exact GB method of [3] finds the intermediate subexpression \(3x\) and then obtains the minimum solution with 3 operations (Figure 2.3(b)).

The minimum adder-steps an MCM block instance with \(N\) constants is given by

\[\text{MAS}_{\text{MCM}} = \max_i \left\lceil \log_2 S(h_i) \right\rceil,\]

where \(h_i\) represents a single constant multiplication, \(S(h)\) is the number of nonzero digits in the CSD representation and \(0 \leq i \leq N - 1\) [29].

There are several algorithms [2, 31, 32], which can find the fewest number of operations that realize the constant multiplications without violating a given delay constraint \(dc\), where \(dc \geq \text{MAS}_{\text{MCM}}\). For the example, the minimum adder-steps of both \(21x\) and \(53x\) is 2. The approximate algorithm [3] modified to handle a delay constraint finds a solution with 4 operations when \(dc\) is 2 (Figure 2.3(c)). On the other hand, the exact GB algorithm [3] finds a solution with one more adder-step, but one less operation (Figure 2.3(b)).
Chapter 3

Parallel Computing

Parallel computing is the use of a parallel computer to reduce the time needed to solve a single computational problem. A parallel computer can be classified in two categories: multicomputers (distributed-memory system) and centralized multiprocessors (shared-memory system) [33]. A multicomputer is a parallel computer constructed out of multiple computers and all the interconnection network. Each processor only has direct access to its own local memory. Without a shared address space, processors interact with each other by passing messages, and there are no cache coherence problems to solve. In contrast, a centralized multiprocessor is a system in which all CPUs share access to a single global memory, which supports communication and synchronization among processors.

Parallel programming is done using a language that allows you to explicitly indicate how different portions of the computation may be executed concurrently by different processors. A standard library specification for parallel programming in multicomputers is the message passing interface (MPI).

This chapter presents some basic concepts regarding MPI and its use, and after presents an overview of existing parallel algorithms for DFS.

3.1 MPI - Message Passing Interface

MPI (Message Passing Interface) is a standard specification for message passing libraries [33, 34]. MPI has been ported to virtually every commercial parallel computer and free libraries meeting the MPI standard are available. Programming a message passing software using MPI can bring advantages in terms of portability, efficiency, and flexibility. Although it is not an IEEE or ISO standard, it has become the “industry standard” for writing message passing programs on High-performance computing (HPC) platforms. Originally, MPI was designed for distributed memory architectures, but their libraries are nowadays adapted to handle both types of underlying memory architectures seamlessly.

MPI provides many advantages, namely: it involves explicit parallelism, which leads to a better performance; it makes available a number of optimized collective communication routines; it allows parallel access to memory, and in general, communications and computation can be overlapped; also, communications often offer synchronization naturally.
Some of the basic functions for programming in MPI and their use are listed below.

- **MPI_Init**: to initialize MPI;
- **MPI_Comm_rank**: to determine a process's ID number;
- **MPI_Comm_size**: to find the number of processes executing in parallel;
- **MPI_Reduce**: to perform a reduction operation;
- **MPI_Barrier**: to perform a barrier synchronisation;
- **MPI_Finalize**: to finalize MPI.

In distributed-memory systems each processor has direct access only to the instructions and data stored in its local memory. As such, it is required an interconnection network which supports message passing between processors. Processor A may send a message containing some of its local data values to processor B, giving processor B access to these values. A task is instantiated as a process in the message-passing model.

The existence of the interconnection network means there is a channel between every pair of processes, that is, every process can communicate with every other process. The user specifies the number of concurrent processes when the program begins, and typically the number of active processes remains constant throughout the execution of the program. Every process executes the same program, but because each one has a unique ID number, different processes may perform different operations as the program unfolds. A process alternately performs computations on its local variables and communicates with other processes or I/O devices.

MPI makes available the following two functions for the communication between processors:

- **MPI_Send**: To send an array A from a process i to another process j;
- **MPI_Recv**: So that a process j can receive an array as B, from process i.

**MPI_COMM_WORLD** is a communicator, which represents a set of processes and the private communication channels between those processes. Communicators allow collective operations, which is one of the advantages of using MPI. Some of the available collective operations and their use are shown below:

- **Broadcast**  
  - **MPI_Bcast**: A process (source) sends data to all other processes;
- **Scatter**  
  - **MPI_Scatter**: Subparts of a single large array are distributed amongst processes;
- **Gather**  
  - **MPI_Gather**: Each process contributes with its local data, which is gathered into a larger array;
- **Reductions**  
  - **MPI_Reduce**: Combine elements in an input buffer from each process;
- **Synchronization**  
  - **MPI_Barrier**: A process has to wait at the barrier, until all the others processes have reached it.
For a point-to-point communication, consider the following example where a process 0 sends an array A to another process 1, which receives it as B:

* Process 0 *

```c
#define TAG 123
double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1, TAG, MPI_COMM_WORLD);
```

* Process 1 *

```c
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD, &status);
OR
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode. Blocking routines only return upon completion, meaning that they are safer for work. For example, a blocking send only returns after is safe to modify the buffer (send data) for reuse. Non-blocking routines return almost immediately, without waiting for the communication event to complete. They are used to overlap computation with communication to improve the performance. But since they are less safe, more caution should be taken when calling these routines. Some off the MPI non-blocking routines are:

- `MPI_Isend`: To perform a non-blocking send of an array to another process;
- `MPI_Irecv`: To perform a non-blocking receive of an array from another process;
- `MPI_Iprobe`: To perform a non-blocking test for a message identified by a source and tag.

### 3.2 Parallel Algorithms for DFS

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root, and searches as deeply as possible along each branch before backtracking and searching another path. Speeding up any algorithm is usually significant on its own. But in order to benefit from sub-linear DFS algorithms, constructing the graph/tree itself should not be more time-consuming than the time required to perform the search. An example, is when the search space is constructed in parallel, or when we need to search a given graph many times, starting at different vertices. As such, there are several parallel algorithms for performing DFS [35–38].

According to [35, 37], we can parallelize DFS by sharing the work to be done among a number of processors, where each processor searches a disjoint part of the search space in a serial depth-first fashion. Essentially, we want to divide the search space into N separate trees, where N is the number of processors. An open list of unvisited nodes and a closed list of visited nodes are usually maintained locally by each processor. When a processor has finished searching its part of the search space, it tries
Algorithm 1 Parallel DFS

Parallel DFS: Processor $P_i$

1: while (not terminated) do
2: if (stack[$i$] = empty) then
3: GETWORK();
4: while (stack[$i$] ≠ empty) do
5: DFS(stack, stack[$i$]);
6: GETWORK();
7: TERMINATION_TEST();

Algorithm 2 Recursive DFS

Recursive DFS

1: function DFS($G, v$)
2: label $v$ as visited;
3: for all $G$.adjacentVertex($v$) as $w$ do
4: if (vertex $w$ not visited) then
5: DOWORK($w$);
6: DFS($G, w$);

For each processor, the (part of) state space to be searched can be efficiently represented by a stack. The depth of the stack is the depth of the node being currently explored and each level of the stack keeps track of untried alternatives. Each processor maintains its own local stack on which it executes DFS. When the local stack is empty, it takes some of the untried alternatives of another processor’s stack.

In the implementation described in [35], all the search space is given to one processor at the start of each iteration, and other processors are given null spaces (i.e., null stacks). From then on, the search space is divided and distributed among various processors. The basic parallel DFS routine in each of the processors and a recursive DFS routine are shown in Algorithms 1 and 2, respectively. $P_i$ denotes the $i^{th}$ processor, stack[$i$] denotes the stack of the $i^{th}$ processor, $G$ represents a sub-graph/tree and $v$ denotes a vertex of $G$.

Once a processor finishes the search in its available space, using the DFS($G, v$) routine, it calls GETWORK() to get more work. If new work is not received (from the targets tried in GETWORK()), then the routine TERMINATION_TEST() is called to test if all the other processors have finished. If the termination test fails, then GETWORK() is called again to get some work. The procedure GETWORK() is architecture dependent. The following version is presented in [35] as a good alternative for shared-memory multiprocessors.

function GETWORK()

for $j = 0 ...$ NumRetry-1 do

target = (target + 1) mod N;

if (work is available at the stack of processor $P_{target}$ above the cutoff depth) then

lock stack[$target$];

pick work from target;

unlock stack[$target$];

return;

The procedures GETWORK and TERMINATION_TEST involve communication with other processors. By restricting these communications with immediate neighbours only, the overhead can be reduced. On a distributed-memory system, whenever a processor needs work, it sends a request for work to one of its neighbors.
neighbours. Each immediate neighbour is polled for work in a round-robin fashion [35]. If the neighbour has work (i.e., unsearched alternatives) available above a cutoff depth, it sends it to the requesting processor; otherwise it sends a reject message. If the requesting processor receives a reject, then it tries to get work from another neighbour. All processors service requests from their neighbour(s) periodically. The stack transfer can be done in two ways:

1. Copy the relevant node information from every level of stack and transfer it.
2. Copy the operator sequence applied and regenerate the stack in the requesting processor using the initial node and the operator sequence.

The choice between the two is determined by the ratio $\frac{r_{\text{comm}}}{r_{\text{search}}}$, where $r_{\text{comm}}$ is the rate of transfer of bytes between processors and $r_{\text{search}}$ is the rate of exploring the search tree. Also, two parameters of the algorithm important for performance on a given architecture and a given problem, are the splitting strategy and a cutoff depth [35,37]. The splitting strategy determines which part and amount of the work should be sent in each exchange. Usually, sending half of the available work leads to better results. The cutoff depth is chosen to assure a minimum amount of work to be sent, to prevent more overhead. The termination detection is also an important subject for the DFS parallelization.

**Dijkstra’s Token Termination Detection Algorithm**

Another important subject to address in the DFS parallelization problem is the termination detection. The Dijkstra Token Termination Algorithm (DTTDA) [37] considers a simplified scenario, where $p$ processors are connected in a logical ring. In this scenario, once a processor becomes idle, it never receives more work. The algorithm begins when processor $P_0$ becomes idle, which then initiates a token. This token is sent to the next processor in the ring, $P_1$. At any stage in the computation, if a processor receives a token, the token is held at the processor until the computation assigned to the processor is complete. On completion, the token is passed to the next processor in the ring. If the processor was already idle, the token is passed to the next processor. Note that if at any time the token is passed to processor $P_{p-1}$, then all processors $P_0, \ldots, P_{p-1}$ have completed their computation. Processor $P_{p-1}$ passes its token to processor $P_0$ when it becomes idle. Upon receiving the token, processor $P_0$ knows that all processors have completed their computation and the algorithm can terminate.

Such a simple scheme cannot be applied to search algorithms where the communications between nodes happens in different ways, such are the cases where the load balancing is done dynamically. In these cases, after a processor becomes idle, it may receive more work from other processors. The token termination detection scheme thus must be modified. This modification will be addressed in Chapter 5.
Chapter 4

SIREN

In this chapter the SIREN [10] algorithm is described. The chapter begins with a brief description of Linear Programming, which is the technique used by SIREN to determine the filter coefficients. Then, a description of the SIREN algorithm and its depth-first search (DFS) method are presented.

4.1 Linear Programming

SIREN uses Linear Programming (LP), a technique to optimize (minimize or maximize) a linear objective function, subject to a set of linear (equality and inequality) constraints. LP problems can be expressed as:

\[
\begin{align*}
\text{minimize :} & \quad f = c^T \cdot x \\
\text{subject to :} & \quad A \cdot x \geq b \\
& \quad lb \leq x \leq ub
\end{align*}
\]

In (4.1), \(x\) represents the vector of variables to be determined, \(c\) is a cost value associated with each variable \(x_j\), \(1 \leq j \leq n\), \(A \cdot x \geq b\) is a set of linear constraints, and \(lb\) and \(ub\) represent the lower and upper bounds of variables, respectively.

There are many polynomial-time algorithms [39,40] for the LP problem, that assume the variables to be real numbers. On the other hand, there is no polynomial-time algorithm [41] for the cases where all or some variables are restricted to integers, as in pure Integer LP (ILP) or Mixed ILP (MILP) problems, because these LP-problems become NP-complete.

4.2 SIREN Description

The SIREN algorithm [10, 42] takes the 5-tuple \(f\text{spec}\) denoting the filter specifications as input and returns a set of filter coefficients yielding a minimum number of adders/subtractors in the filter design and satisfying the filter constraints. Its pseudocode is shown in Algorithm 1, where \(Q\) stands for the

---

1 The minimization objective can be easily converted to a maximization objective by negating the cost function. Less-than-or-equal and equality constraints are accommodated by equivalences, \(A \cdot x \leq b \iff -A \cdot x \geq -b\) and \(A \cdot x = b \iff (A \cdot x \geq b) \land (A \cdot x \leq b)\), respectively.
where $N$, $\delta$, and $\omega$ in [10], using a symmetric FIR filter with quantization value used to convert floating-point numbers to integers. SIREN was described in detail in [10], using a symmetric FIR filter with $fspec$ ($N=8, \omega_p=0.2\pi, \omega_s=0.7\pi, \delta_p=0.01, \delta_s=0.01$) as an example, where $N$ is the filter length, $\omega_p$ and $\omega_s$ are, respectively, the passband and stopband frequencies, and $\delta_p$ and $\delta_s$ are the passband and stopband ripple, respectively.

First, the search space is restricted, by finding the lower and upper bounds of coefficients and scale factor $s$ (to compensate the passband gain - see Section 2.1) using the $ComputeBounds$ function. To find the lower bounds of coefficients the following LP problem is solved for each coefficient $h_i$:

$$
\begin{align*}
\text{minimize} : & \quad f = h_i \\
\text{subject to} : & \quad s(1 - \delta_p) \leq G(\omega) \leq s(1 + \delta_p), \quad \omega \in [0, \omega_p] \\
& \quad -s(\delta_s) \leq G(\omega) \leq s(\delta_s), \quad \omega \in [\omega_s, \pi] \\
& \quad h^l \leq h \leq h^u \\
& \quad s^l \leq s \leq s^u
\end{align*}
$$

(4.2)

where $s^l$ and $s^u$ represent the lower and upper bounds of $s$, initially set to 0.01 and 100 respectively, and $h^l$ and $h^u$ denote the lower and upper bounds of each filter coefficient, which were initially set to -1 and 1, respectively. The value of $h_i$ in the LP solution corresponds to its lower bound $h^l_i$ and is stored in $h^l$. In a similar way, the upper bound of each coefficient $h^u_i$ is found when the cost function is $f = -h_i$ and is stored in $h^u$. Thus, the sets $h^l$ and $h^u$ consist of the floating-point lower and upper bounds of all coefficients, respectively. The values of $s^l$ and $s^u$ are found in a similar way. The number of LP problems to be solved in symmetric filters is given by $2\lfloor M \rfloor + 4$, where $M = (N - 1)/2$ (see Section 2.1).

For the example, the floating-point lower and upper bounds of filter coefficients are computed as $h^l = \{h^l_0, h^l_1, h^l_2, h^l_3\} = \{-0.0966, -0.0915, 0.0015, 0.0039\}$ and $h^u = \{h^u_0, h^u_1, h^u_2, h^u_3\} = \{-0.0003, -0.0002, 0.4144, 1\}$ respectively. Also, $s^l$ and $s^u$ are 0.01 and 2.53, respectively.

Second, the $OrderCoeffs$ function finds an ordering of coefficients to be used in its DFS method while constructing the search tree (described ahead). The main reason behind finding an ordering of coefficients is the fact, observed in experiments, that if coefficients with narrower intervals between upper and lower bound are placed in lower depths of the search tree, fewer decisions are made and conflicts occur earlier. The coefficients are sorted in ascending order according to their $h^u_i - h^l_i$ values and their indices $i$ are stored in this order in $O$. Thus, the runtime of SIREN is reduced significantly, still exploring all possible values of coefficients. For the example, the ordering of coefficients is $O = \{1, 0, 2, 3\}$.

---

**Algorithm 3** THE SIREN algorithm

SIREN($fspec$)

1. $Q = 0$, sol = \{
2. $(h^l, h^u, s^l, s^u) = ComputeBounds(fspec)$
3. $O = OrderCoeffs(h^l, h^u)$
4. repeat
5. $Q = Q + 1, H^l = [(h^l \cdot 2^Q)], H^u = [(h^u \cdot 2^Q)]$.
6. if CheckValidity($H^l, H^u$) then
7. $sol = DFS(fspec, O, Q, H^l, s^l, s^u)$
8. until sol $\neq \emptyset$
9. return sol.

---
Third, in the iterative loop of SIREN, starting with the quantization value \( Q \) equal to 1, the floating-point lower (upper) bound of each coefficient is multiplied by \( 2^Q \), rounded to the smallest following (the largest previous) integer, and is stored in \( H_i^l(H_u^u) \). The validity of these sets \( H_i^l \) and \( H_u^u \) is tested by the CheckValidity function by simply checking each coefficient if \( H_i^l \) is less than or equal to \( H_u^u \). If they are not valid \( (H_i^l > H_u^u) \), this function returns zero. In this case, \( Q \) is increased by one, \( H_i^l \) and \( H_u^u \) are updated, and the CheckValidity function is applied again.

If \( H_i^l \) and \( H_u^u \) are valid, the DFS method is then applied, exploring all possible values of each coefficient in between \( H_i^l \) and \( H_u^u \), to find a set of filter coefficients which respects the filter constraints and yields the minimum design complexity, or to prove that there exists no such a set of filter coefficients. If the former condition occurs, \( sol \) is returned. If the latter condition occurs, \( Q \) is increased by one, \( H_i^l \) and \( H_u^u \) are updated, and the DFS method is applied again. Hence, SIREN ensures that its solution is a set of fixed-point filter coefficients obtained using the smallest \( Q \) value.

Note that \( Q \) is an important parameter in the filter design. When \( Q \) increases, the bit-widths (sizes) of coefficients increase. Thus, such coefficients lead to larger sizes of registers and structural adders in the register-add block of the transposed form (Figure 1.1). Also, most probably, they lead to a large number of operations in the multiplier blocks of both forms (direct and transposed forms). Similar to \( Q \), the solution quality of an FDO algorithm is evaluated by the effective wordlength (EWL) of a set of coefficients [12, 22, 23], computed as \( \max\{\log_2|h_i|\} \) with \( 0 \leq i \leq N - 1 \) when fixed-point coefficients are considered.

### 4.3 SIREN Depth-First-Search Method

In the DFS method of SIREN, the search tree is constructed based on the ordering of coefficients \( O \), where a vertex at depth \( d \), \( V_d \), denotes the filter coefficient whose index is the \( d \)-th element of \( O \), i.e., \( h_{O(d)} \). An edge at depth \( d \) of the search tree, i.e., a fanout of \( V_d \), stands for an assignment to the vertex from \([V_d^l, V_d^u]\) where \( V_d^l (V_d^u) \) denotes the lower (upper) bound of \( V_d \). Note that the values of the vertex at depth \( d \) are assigned incrementally starting from \( V_d^l \) to \( V_d^u \).

When \( d \) is 1, the DFS method assigns \( H_i^l_{O(1)} \) and \( H_u^u_{O(1)} \) to \( V_1^l \) and \( V_1^u \), respectively, and sets the value of the vertex \( V_1 \) to \( V_1^l \). At any depth greater than 1, \( d > 1 \), although the lower and upper bounds of a vertex can be taken from \( H_i^l \) and \( H_u^u \), respectively, tighter lower and upper bounds can be computed, since the values of coefficients at previous depths have been determined and fixed. The lower bound of the vertex \( V_d \) is computed by solving the following LP problem, where the non-determined coefficients and \( s \) are the continuous variables of the LP problem.

\[
\begin{align}
\text{minimize:} \quad & f = h_{O(d)} \\
\text{subject to:} \quad & s(1 - \delta_p) \leq G(\omega)/2^Q \leq s(1 + \delta_p), \quad \omega \in [0, \omega_p] \\
& -s(\delta_u) \leq G(\omega)/2^Q \leq s(\delta_u), \quad \omega \in [\omega_u, \pi] \\
& H_i^l \leq h_i \leq H_u^u, \quad i \in [O(d), O([M] + 1)] \\
& s^l \leq s \leq s^u \\
& h_{O(1)}...h_{O(d-1)}: \text{determined and fixed}
\end{align}
\tag{4.3}
\]

\( h_{O(1)}...h_{O(d-1)}: \) determined and fixed

17
In this LP problem, the lower and upper bounds of all non-determined coefficients are taken from $H^l$ and $H^u$, respectively. The upper bound of $V_d$ is computed when the cost function is changed to $f = -h_O(d)$. If there exist feasible solutions for both LP problems, this lower (upper) bound is rounded to the smallest following (the largest previous) integer and assigned to $V_d^l$ ($V_d^u$). If $V_d^u \geq V_d^l$, they are determined to be the lower and upper bounds of $V_d$. Whenever there is no feasible lower or upper bound for $V_d$ or $V_d^u < V_d^l$, the search is backtracked chronologically to the previous vertex until there is a value to be assigned between its lower and upper bounds.

When the values of all coefficients are determined, i.e., the leaf at the final depth of the search tree is reached (when $d$ is $\lfloor M \rfloor + 1$ for symmetric filters), the implementation cost of the transposed form filter is computed as $TA = MA + SA$, where $TA$ is the total number of operations in the filter, and $MA$ and $SA$ are the number of arithmetic operations in the MCM block and the number of structural adders in the register-add block, respectively (Fig. 1.1). While $MA$ is found using the exact MCM method [3], $SA$ is computed based on the nonzero coefficients. No adder is needed for a coefficient equal to 0 in the register-add block. This coefficient set is stored in $sol$ if its $TA$ value is smaller than that of the best one found so far, which was set to infinity in the beginning of the DFS method.

To prune the search tree, the $TA$ value is estimated when depth $d$ is greater than $2M/3$ for symmetric filters. This value was chosen to be close to the bottom of the search tree, so that efforts can be spared in computing an estimate that usually does not yield a backtrack. To make this estimation, the lower bound on $MA$ is found using the determined coefficients [43]. The lower bound on $SA$ is found after all non-determined coefficients are set to a value. To do so, the upper and lower bound interval of each non-determined coefficient is checked if 0 is included. If so, this non-determined coefficient is set to 0. Otherwise, it is assumed to be a constant different from 0.

The DFS method terminates when all possible values of coefficients have been explored. If $sol$ is empty, it is guaranteed that there is no set of filter coefficients which can be selected from their quantized lower and upper bounds respecting the filter constraints. Otherwise, $sol$ consists of fixed-point coefficients that lead to a filter with minimum design complexity, satisfying the filter constraints.

For the example, when $Q$ is 5, the quantized lower and upper bounds of filter coefficients are $H^l = \{-3, -2, 1, 1\}$ and $H^u = \{-1, -1, 2, 32\}$, respectively. Note that no solution was found with $Q < 5$. The
search tree constructed by the DFS method when $Q$ is 5 is shown in Figure 4.1. The $a$ and $b$ in $[a, b]$ given next to each vertex stand respectively for its lower and upper bounds which are dynamically computed as coefficients are fixed during the DFS search. In this figure, the actual traverse of the DFS method on filter coefficients can be followed from top to bottom and from left to right. Conflict denotes that given the already determined coefficients, there exists no feasible lower/upper bound for the current depth vertex. Pruned indicates that the set of determined coefficients cannot lead to a better solution than the best one found so far. Success represents that the set of coefficients leads to a solution satisfying the filter constraints.

Observe that the SIREN search space will be very unbalanced. As the values of coefficients are determined, the intervals between the lower and upper bounds of coefficients are reduced when compared to those in the original $H^l$ and $H^u$. If the DFS method was not equipped with these techniques, that order the filter coefficients, determine the lower and upper bounds of coefficients dynamically, and prune the search space, in the worst case, it would consider $\prod_{i=0}^{\lfloor M \rfloor} (H_u^i - H_l^i + 1)$ possible sets of coefficients for a symmetric filter, i.e., the number of leafs at the final depth of the search tree, in each iteration of SIREN. For the example, this value is 2496 when $Q$ is 5. However, the DFS method ensures the minimum solution visiting just 1 leaf at the final depth and 10 branches.

The performance of SIREN depends heavily on the minimum quantization value $Q$, the filter length $N$, and the exact MCM algorithm [3]. The $Q$ value has an impact on the number of runs of the DFS method, the lower and upper bounds of coefficients, the number of branches in the search tree, and the sizes of coefficients which affect the performance of the exact MCM algorithm [3]. The $N$ value has an effect on the performance of the exact MCM algorithm and on the depth of the search tree. The performance of the exact MCM algorithm is related to the number and size of coefficients [3].

As such, the performance of SIREN can be increased by developing a parallel version of the DFS method. Although the whole search space is very unbalanced, it can be dynamically divided into many small parts and they can be explored in a reasonable time simultaneously. However, the FDO problem is NP-complete [23], and hence, heuristics are indispensable for filters with a large number of coefficients.

**Pseudo-Code**

Algorithm 4 represents a pseudo-code for the SIREN DFS method that was described. Here, DetermineLUBCoefs represents the routine to find the lower and upper bound of the vertex $V_d$ by solving the LP problem in Equation 4.3. EstimateLbImpCost is the routine to estimate $TA$ and prune the search space, while DetermineHcubSol is the computation of the implementation cost, $TA = MA + SA$, when the final depth of the search tree is reached ($Max\_Depth = \lfloor M \rfloor + 1$).
Algorithm 4 The SIREN DFS Method

DFS \((f_{\text{spec}}, O, Q, H, s^l, s^u)\): Vertex \(V_d\), depth \(d\)

1: repeat
2: \([V^l_d, V^u_d] \leftarrow \text{DetermineLUBCoefs}(f_{\text{spec}}, O, Q, V_d, s^l, s^u)\)
3: \(\text{if } V^l_d \leq V^u_d \text{ then}\)
4: \(\text{if } d < (\text{Max Depth} - 1) \text{ then}\)
5: \(V_d + 1 \leftarrow V^l_d\)
6: \(d \leftarrow d + 1\)
7: \(\text{if } d > 2M/3 \text{ then}\)
8: \(\text{cost} \leftarrow \text{EstimateLbImpCost}(f_{\text{spec}}, O, Q, V_d, s^l, s^u)\)
9: \(\text{if } \text{cost worse than } sol \text{ then}\)
10: \(V_d \leftarrow V_d + 1\) \(\triangleright \text{Pruning}\)
11: \(\text{else}\)
12: \(\text{Backtrack}(V_d)\)
13: \(\text{else}\)
14: \(sol' \leftarrow \text{DetermineHcubSol}(f_{\text{spec}}, O, Q, V_d, s^l, s^u)\)
15: \(\text{if } sol' \text{ better than } sol \text{ then}\)
16: \(sol \leftarrow sol'\)
17: \(\text{if } V_d < V^u_d \text{ then}\)
18: \(V_d \leftarrow V_d + 1\)
19: \(\text{else}\)
20: \(\text{Backtrack}(V_d)\)
21: \(\text{else}\)
22: \(\text{if } V_d < V^u_d \text{ then}\)
23: \(V_d \leftarrow V_d + 1\)
24: \(\text{else}\)
25: \(\text{Backtrack}(V_d)\)
26: until \(d = -1\) \(\triangleright \text{Until all the vertices are visited}\)
27: return \(sol\)
Chapter 5

SIREN Parallelization

This chapter presents and discusses the proposed and implemented solutions, and the results obtained for the optimization of the SIREN algorithm using parallel computing. As mentioned in the previous chapter, SIREN uses a DFS with Branch and Bound method to build the search tree, using precalculated weights. Due to the pruning performed, the search space will be very unbalanced, which hinders the load balancing. As such, the load balancing has to be done dynamically, using more complex strategies and with more communications between processes.

Section 5.2 proposes a solution for the problem, whereas the following section presents an overall description of the parallelization method implemented. Section 5.4 is dedicated to the load balancing, where three different strategies are discussed and the results for each one are presented. A summary on the results are presented in Section 5.5.

5.1 Starting Point

SIREN was originally written in MATLAB®, so the first step in order to implement the parallel version was to implement a C version of the algorithm. Simply with this conversion, the performance of SIREN was greatly improved. Table 5.1 presents the results obtained for some examples using SIREN MATLAB and C versions. Both versions used lp.solve_5.5.2.0 as an LP solver, and were run on a PC with Intel Xeon at 2.1 GHz under Linux. As we can see from the table 5.1, it was possible improve the CPU time, with an average speed-up of approximately 3.

The speed-up ($S$) represents a gain in performance and is given by the fraction between the CPU-Times obtained from two different implementations of the same problem. In the previous case, it was given by the fraction between the CPU times of the MATLAB version of SIREN and the newly implemented C version. For the parallel implementation, the speed-up represents the fraction between CPU time of the serial version ($T_{serial}$ - CPU time from C version) and the CPU time of the parallel version ($T_{parallel}$), and the objective is to obtain a speed-up that is equal (linear) or higher (supra-linear) than the number of processors:

$$S = \frac{T_{serial}}{T_{parallel}}$$

(5.1)
Table 5.1: Summary of CPU Times of SIREN - MATLAB vs C version.

<table>
<thead>
<tr>
<th>Filter Size (N)</th>
<th>CPU Time</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MATLAB</td>
<td>C</td>
</tr>
<tr>
<td>30</td>
<td>7m56s</td>
<td>4m08s</td>
</tr>
<tr>
<td>35</td>
<td>34m20s</td>
<td>13m8s</td>
</tr>
<tr>
<td>38</td>
<td>4m24s</td>
<td>1m18s</td>
</tr>
<tr>
<td>40</td>
<td>13m45s</td>
<td>4m44s</td>
</tr>
<tr>
<td>43</td>
<td>2h</td>
<td>46m</td>
</tr>
</tbody>
</table>

After the conversion, the attention was focused on the parallelization problem.

5.2 Proposed Solution

Based on the four steps of the Foster’s methodology [33], we begin the design of a parallel algorithm by partitioning the problem, then we handle communications, make agglomerations, and lastly, do the mapping. MPI provides the tools that facilitate communications and the load balancing.

From the basic idea in Section 3.2, the most important problems that arises in SIREN’s case are the following. First is to decide the best way to divide the workload (partitioning), i.e, the size each processor gets and when they get it. Second, decide the way that a processor/node gets an unsearched part of the tree from the others, from which processor it receives it, and which information should be sent (communication). Also, the amount of communications and the size of each message should be kept to a minimum to reduce overhead, although there is a greater flexibility regarding the size of each message (two small messages exchanges usually cause more overhead and delay than one larger message).

Load Balancing

Intuitively, the best way to divide the work is in static equal sized parts. If \( P \) processors are available, they all do redundant work to construct the tree, until at least \( P \) nodes are available. At this point, each processor continues to work only a specific part of the tree. When the search tree is unbalanced, some processors will have different amounts of work and some will be idle too soon. This problem can be reduced by doing redundant work some levels further in the tree, i.e, dividing the work load when \( n \text{ nodes} > 2P \).

Since the search tree in SIREN’s case is very unbalanced (see Section 4.3), even with the initial division each processor may have different amounts of work. As such, the splitting has to be done dynamically. One way of doing this is, whenever a processor needs work, it sends a request to one of its neighbours, as mentioned in Section 3.2. If a neighbour has available work, it sends it to the requesting processor. Otherwise, it sends a reject signal. When a reject signal is received, the processor asks for work from another neighbour. In order to reduce the overhead, we can assume that a processor only has 2 neighbours.

The unexplored states can be conveniently stored in a stack. Each processor maintains its own local
stack on which it executes DFS. When a processor’s local stack is empty, it requests untried alternatives from another processor’s stack. When a work transfer is made, the work in the donor’s stack is split into two stacks and one of them is sent to the requester.

Regarding the splitting strategy, it has been concluded that a half-split strategy of the available work generally leads to a higher performance [37, 44]. If the work given out is too small, the requester would become idle too soon. If it is too large, then is the sender who would become idle too soon. Given that the tree is very unbalanced, the best strategy to remove the nodes from one processor to send to another is to pick-up half of the available nodes above a cutoff depth ($\theta$) [37]. The cutoff should be chosen to avoid sending very small amount of work, i.e., the work done below $\theta$ has to take much longer time when compared to the stack transfer time. However, to choose a cutoff depth and determine half of the available nodes may not be a simple task in the SIREN algorithm, since the search space and solution are being dynamically updated.

As such, a different splitting strategy, that may be easier to implement but less efficient, consists in simply assigning some of the first unvisited sub-trees, at the lowest depth of the tree, to the requesting processor. In other words, upon a work request, a processor starts searching at the tree root for the first parent node (at the lowest level), which has subtrees or a leaf that has not been visited yet. It then marks the subtree as visited and sends its information to the requester.

**Communication**

Communications occur when a processor requests and sends work, when it finds a better solution and to detect the end of computation. The important information to be sent in each transfer are: the quantization value $Q$, the upper and lower bound sets ($H^u \leq H^l$), the depth $d$ of the vertex, and the already determined/fixed filter coefficients at depth $d$ ($h_{O(1)} \ldots h_{O(d-1)}$). A vertex at depth $d$, $V_d$, and the lower and upper bound of $V_d$, $V^l_d$ and $V^u_d$ (see Section 4.2), are computed based on the previous values so they don’t need to be sent.

When a solution (set of coefficients that leads to a filter design which satisfies the constraints) is found, it should be broadcast to all other nodes, so that pruning can be done. Although MPI offers a routine to make an efficient broadcast (see Section 3.1), it uses blocking send and receive functions. For this case, the routine is not very useful, since the broadcast has to be done in the middle of the search, which can cause a delay for some of the processors, and our aim is to improve the CPU time. As such, we want to perform a broadcast without blocking, so that the search can be resumed as fast as possible, independently of the other processors.

An approach is to use an non-blocking send with a designated tag to all the others processors when a solution is found, and each processor checks at the end of a computation with a nonblocking receive, whether the message has been posted by anyone. This is also the approach that can be used by all the processors to receive a work request. Another approach would be for each processor to create a specific task for handling the broadcast (for example, using Pthreads [34]). Since the broadcast is not done frequently, the tasks would be idle most of the time, causing little delay in the processor work, even in the worst case scenario where only a single core is available.
Termination Detection

Finally, the termination detection should also be given special attention. Since the search only finishes when all available nodes have been visited or pruned, all nodes need to verify if the others have also finished the work, so that the computation can be halted.

One possible way is to use a Modified Dijkstra Token Termination Algorithm (MDTTDA) \(^1\) [37]. In this algorithms scheme, \(P\) processors are organised into a ring, and a token has to traverse the ring to determine if the computation has ended. A processor only sends the token after receiving it and becoming idle itself. However, it can receive work after it has finished its part of the work and sent the token. As such, this algorithm states that a processor can be in one of two states: \textit{black} or \textit{white}. Initially, all processors are in state \textit{white} and the algorithm proceeds as follows:

1. When process \(\text{Proc}_0\) becomes idle, it makes itself \textit{white} and initiates a \textit{white token} to be sent to process \(\text{Proc}_1\).\(^2\)
2. If a processor \(\text{Proc}_i\) sends work to another processor \(\text{Proc}_j\) and \(j < i\), then processor \(\text{Proc}_i\) becomes black.
3. If processor \(\text{Proc}_i\) has the token and \(\text{Proc}_i\) becomes idle, then it passes the token to \(\text{Proc}_{i+1}\). If \(\text{Proc}_i\) is black, then the colour of the token is set to black before it is sent to \(\text{Proc}_{i+1}\) (this means the token must traverse the ring again). If \(\text{Proc}_i\) is white, the token is passed unchanged.
4. After \(\text{Proc}_i\) has passed the token to \(\text{Proc}_{i+1}\), \(\text{Proc}_i\) becomes \textit{white}.

The algorithm terminates when processor \(P_0\) receives a \textit{white token} and is itself idle. This algorithm ensures a correct termination detection, since it accounts the possibility of a processor receiving work after it has already been accounted for by the token.

5.3 Implementation

The implemented solutions were based on the one proposed in the previous section. However, some modifications were required, which are going to be discussed in the next sections. To obtain the parallelization, it was important to identify when to send/receive and request work, when to broadcast/check if a new solution was found and how to determine the end of computation.

The load balancing (partitioning and splitting strategy) proved to be a major concern, so its discussed in the next section, where different strategies with their respective results are presented. This section presents the decisions regarding communications and termination detection.

For the serial implementation of the SIREN algorithm, after a valid quantization value \(Q\) is found and the upper and lower bound sets \((H^l, H^u)\) are computed, the information that changes during the DFS method are the ones regarding a single branch of the tree. These values are the current vertex \(V_d\), its lower and upper bounds, \([V_d^l, V_d^u]\) and the already determined and fixed coefficients, \(h_{O(1)}...h_{O(d-1)}\) (vertices at lower depths and their respective upper bounds). This information is stored in a matrix

\(^1\)The original DTTDA assumes that if a processor becomes idle, it does not get more work. Process \(\text{Proc}_0\) initiates and sends a token and termination occurs when \(\text{Proc}_0\) receives the token again. See Section 3.2

\(^2\)The token travels in the sequence \(\text{Proc}_0, \text{Proc}_1, \ldots, \text{Proc}_{P-1}, \text{Proc}_0\)
search:lubas of size $2 \times \text{Max\_Depth}$, where the value in search:lubas[0][d] corresponds to the current vertex being processed, $V_d$, and the upper bound of the vertex, $V_u^d$, is stored in search:lubas[1][d]. Also the values of previous decisions $h_{O(1)}...h_{O(d-1)}$ and their respective upper bounds are stored in search:lubas[k][0,...,d-1], where $k = 0, 1$.

In the parallel implementation, $Q$ and $\{H^l, H^u\}$ have the same values for all processors, since they are computed before the DFS method. As such, the information that needs to be exchanged are the current vertex $V_d ([V_l^d, V_h^d])$, and the previous fixed decisions $h_{O(1)}...h_{O(d-1)}$, i.e., the matrix search:lubas and the current depth $d$.

For the first two load balancing strategies implemented (Sections 5.4.1 and 5.4.2), a simple vector with size $2 \times \text{Max\_Depth} + 1$ is sufficient for the exchange. After sending part of its work, a processor only needs to update search:lubas and $d$. If it has no more work, it will proceed to make a work request. As such, and also to prevent major changes to the code structure, there is no need to implement a stack to control the work requests in these strategies, since all the necessary information can be stored in search:lubas.

In order to minimize overhead, each processor only has 2 neighbours. The processors can be organized into a ring, the same way as in the MDTTDA (see Section 5.2), which means that each processor only sends work to the next one, and only receives from the previous one. Although this topology reduces some overhead and is simpler to implement, it will limit the scalability, especially since at the beginning of computation there will be a big delay reaching all the processors in the ring (the $i^{th}$ processor will receive work at least at the $i^{th}$ iteration). As such, for the future implementation special attention should be given the initial division and also a different topology should be used.

When a solution which satisfies the filter constraints is found, it should be broadcast to all other nodes, so that pruning can be done. To prune the search space, the information required are the total number of operations, TA, and the EWL, so they are the only values that need to be broadcast. The broadcast was implemented by making a non-blocking send with a designated tag to all the others processors and each processor checks whether the message has been posted by anyone with a non-blocking receive. Each processor have to check if a better solution was found in two different situations. First, always after computing a vertex in order to perform pruning. And second, when it is idle, since all nodes have to know if they are the one who has the best solution, so at the end of computation they can send the set of coefficients to $P_0$, which will realize the final filter design.

Finally, the termination detection was implemented using the MDTTDA, that was described in the previous section. The algorithm terminates when processor $P_0$ receives a white token and is itself idle. $P_0$ then proceeds to send a END token, so that all other processors know that the computation is finished.

**Pseudo-Code**

Algorithm 5 is a pseudo-code that represents the changes implemented in SIREN DFS method (compare with Algorithm 4). It is important to understand when the communications occur in the parallel implementation.
implementation. When a processor becomes idle (working \( \neq 1 \), line 2), it first makes a work request to the previous process. Then it continuously checks if it has received work, if it has received a termination token, or if a new best solution has been broadcast. When the processor is working, the DFS method continues normally, as in the atomic version. After computing a new vertex, it checks if a work request has been made and proceeds to send work, accordingly to the load balancing strategy. When the search is backtracked (to a previous unvisited vertex), if all vertices have been visited (it reaches depth \( d = -1 \)), it means that the processor has finished its work and it becomes idle (working = 0). Also, when a new best solution is found it has to be broadcast. The load balancing strategies that were implemented are discussed in the next section.

5.4 Load Balancing

This section presents the load balancing strategies that were implemented, for the SIREN parallelization problem. Three different strategies, which were called Liberal Work Forwarding (LWF), Quasi-Liberal Work Forwarding (QLWF) and Fixed Depth BFS (FDBFS), and their respective results, are discussed in the following subsections.

5.4.1 Liberal Work Forwarding

The first load balancing strategy implemented is defined in the following manner. Each processor always sends work (subtrees) upon a work request from the next processor, and requests work to the previous one when they have finished. It is slightly different than the strategies mentioned in 5.2. The easiest strategy consisted on simply assigning the first unvisited sub-tree, at the lowest depth of the tree, to the requesting processor. However, the initial division has to be done in a different way, since each node only requests or sends work to the previous or next node, respectively.

So this strategy was adopted, in order to obtain good results with little changes to the code. This approach implies that the largest part of the work, or all of the work (in the case there is only one subtree), is sent to the next processor. At the beginning, only the first node \((P_0)\) has work, and as such, all the other nodes make a work request to their previous node, and wait. Since the nodes are organized into a ring \((P_0, P_1, ..., P_{p-1}, P_0)\), there will be a delay until work reaches all nodes, since processor \(P_i\) will receive work at least on the \(i^{th}\) iteration.

Upon a work request, a processor will be a depth \(d\), and have a vertex \(V_d\) and all of the previous decisions \(\{V_0, ..., V_{d-1}\}\). Starting at the beginning of the tree (root), the processor searches for the lowest depth \(d_l\)\(^4\), where there are unvisited subtrees. If there is more than one subtree available at depth \(d_l\), i.e., \(V^u_{d_l} > V_{d_l}\), the processor keeps the current branch/subtree, and sends all the others subtrees at this level, which means it keeps \(V_{d_l}\) and sends \([V_{d_l} + 1, V^u_{d_l}]\), and assigns \(V_{d_l}\) to \(V^u_{d_l}\). If the only branch available is the current one, which happens when \(d_l = d\) and \(V_{d} = V^u_{d}\), it sends all of his work \(\{V_0, ..., V_d\}\) to the next processor and becomes idle, proceeding to make a work request to the previous processor.

\(^4\)The lowest depth \(d_l\) starts at 0 and is incremented until unvisited subtrees are found or it reaches depth \(d\) \((d_l = \{0, ..., d\})\)
Algorithm 5 The SIREN Parallel DFS Method

Parallel_DFS (fspec, O, Q, H, s', s''): Processor Pi, Vertex Vd, depth d

1: repeat
2:   if working ≠ 1 then
3:     RequestWork(Pi−1)
4:   while 1 do
5:     if (Received Work) then  ▷ Non-Blocking Receive
6:        ReceiveWork (Pi−1)  ▷ Depends of the LB Strategy
7:        working ← 1
8:        break
9:     if (Received Termination Token) then
10:        if token = END then
11:           the_end ← 1
12:           break
13:     if (has token) then
14:        Forward(token, Pi+1)
15:     if (Received New Best Solution) then
16:        UpdateSolution(sol)
17:     if working = 1 then
18:        [Vd', Vd] ← DetermineLUBCoefs(fspec, O, Q, Vd, s', s'')
19:        if Vd' ≤ Vd then
20:           if (Received New Best Solution) then
21:              UpdateSolution(sol)
22:           if d < (Max_Depth − 1) then
23:              Vd+1 ← Vd'
24:              d ← d + 1
25:           if d > 2M/3 then
26:              if (Received Work Request) then
27:                 SendWork(Pi+1);  ▷ Depends of the LB Strategy
28:              if cost worse than sol then
29:                 if Vd < Vd' then  ▷ Pruning
30:                    Vd ← Vd + 1
31:                 else
32:                    Backtrack(Vd)  ▷ If all node are visited, Pi becomes idle (working ← 0)
33:              else
34:                 sol' ← DetermineHubSol(fspec, O, Q, Vd, s', s'')
35:                 if sol' better than sol then
36:                    sol ← sol'
37:                 Broadcast(sol)  ▷ Non-blocking Broadcast
38:                 if Vd < Vd' then
39:                    Vd ← Vd + 1
40:                 else
41:                    Backtrack(Vd)  ▷ If all node are visited, Pi becomes idle (working ← 0)
42:              else
43:                 if Vd < Vd' then
44:                    Vd ← Vd + 1
45:                 else
46:                    Backtrack(Vd)  ▷ If all node are visited, Pi becomes idle (working ← 0)
47:     until the_end = 1  ▷ Until it receives an END token
48: return sol

27
As mentioned in 5.1, the original SIREN algorithm was written in MATLAB, so a C version was implemented first, which resulted on a speed-up of approximately 3. Taking the results obtained with the SIREN C version as reference, a few examples using this load balancing strategy were run, for different numbers of processors ($P = \{2, 4, ..., 32\}$). The examples were run using `lp_solve_5.5.2.3` as an LP solver, in a Cluster\(^5\) composed of PCs with Intel Core i5 at 2.80GHz, 3.20GHz and 3.40GHz (Quad-cores), under Linux.

We should keep in mind that this strategy was adopted for its simplicity and not for its efficiency, as a first attempt for parallelization. As previously mentioned, the topology of the ring limits the scalability. Also, with this strategy each processor sends a big part of his work, so it can become idle rather quickly and proceed to ask work to his neighbour. This could cause starvation\(^6\) for some of the processors.

However, the results obtained were very good, as shown in Figures 5.1 and 5.2. Figure 5.1 presents the speed-up $S$ obtained from filters with different lengths, $N = \{35, 38, 40, 43\}$. It shows the relation between the results obtained from MATLAB, serial C version and from different number of processors, $P = \{2, 4, 8, 12, 24, 32\}$. The results from the C version are taken as reference ($S_C = 1$), and the results of the MATLAB version ($S_{MATLAB} \approx S_C / 2.5$) are also shown for comparison. From this chart we can conclude that the results are very good, since $S$ presents a growth with $P$ better than linear ($S > P$), until they reach a maximum value or ceiling, $S_{Ceil}$. We also notice that larger filters have higher ceilings values, since they reach higher speed-ups for a larger number of processors.

This results are much better than expected. Due to the topology and strategy, the speed-up should not have a supra-linear behaviour, and not even linear (Amdahl’s law). But in this case, we can conclude that the pruning is what makes it possible. With this strategy, the pruning characteristic of the algorithm is maximized. Every time a processor finds a new solution that satisfies the filter constraints, the search space will be diminished. With this parallel implementation, the best solution will be found much faster than in the serial implementation, so the search space will be considerably smaller, which justifies the

\(^{5}\)Cluster from RNL of Instituto Superior Técnico

\(^{6}\)Starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work.
supra-linear behaviour.

Figure 5.2 presents the average speed-up (AvS) of the results shown in Figure 5.1, for the same filters and number of processors, where each result was ran three times. From this figure we can see that the AvS presents a behaviour slightly better than linear at the beginning, since AvS > P. However, AvS starts growing slower when the number of processors is higher than 16, until it reaches a ceiling value, AvS_Ceil, where it even starts decreasing. This means that, for this strategy and these filters, the best CPU times are obtained when the number of processors is 16, and with more processors, the speed-up does not improve and the results could be worse.

This limitation is to be expected from any parallel implementation. From Amdhal’s law, we know that the theoretical speed-up of the execution increases with the improvement of the resources of the system (more processors), but regardless the magnitude of the improvement, the theoretical speed-up is always limited by the part of the task that cannot benefit from the improvement. In this case, the topology of the search tree is the limit the scalability. SIREN makes use of upper and lower bounds to limit the solutions and pruning is performed, which leads us to conclude that the search tree may expand more in height than width (see section 4.3). So, for a filter with size N, the search space will have a limited width (number of sub-trees) which will result in a limited speed-up.

But we can not predict the width of the search tree, since it depends not only on the size of the filter, N, but also on the filter constraints (frequency and ripple passband and stopbands) and on the load balancing strategy. So, to guarantee that the results obtained were the best for these filters topologies, we want to make sure that the limitation of the speed-up is not caused by the load balancing technique implemented, since in this case there is a delay until every processor starts computing and the processors are organized into a ring. Moreover, since each processor always sends some or all of the work upon a request, this may cause some of them to be idle during long periods of time.

In order to study the reason for the scalability limitation, and hopefully obtain better results, the idle times for a filter with N=43 were analysed, by considering the results with different number of processors. First, it is shown the percentage of idle time for each processor (Figure 5.3). From these charts we
conclude that, for a small number of processors, the load balancing is being correctly performed, since the idle times are very low. However, for a higher number of processors, the great majority of them are idle during the most part of the computation, which means only some of the processors are doing all the work. Specifically for this filter, we conclude that when \( P > 16 \), around 16 to 20 processors are doing all the work, while all the others are idle most of the time, so it makes sense to limit the number of processors to 20.

Then, Figure 5.4 shows the average percentage of idle times (\( \text{AvPIT} \)) for different filters, \( N = \{35, 38, 40, 43\} \), and number of processors, \( P = \{2,4,8,12,16,24,32,64\} \). From this graph, it is clear that the \( \text{AvPIT} \) time increases with the number of processors. Also, taking \( P=16 \) for example, we notice that the \( \text{AvPIT} \) for the filter with \( N=38 \) is 25%, while for the filter with \( N=43 \), it is only 5%. This means that the bigger the filter the latter the percentage of idle time increases with the number of processors, i.e., for bigger filters, a higher number of processors are working and \( S_{Ceil} \) is higher. However, when \( P=64 \), \( \text{AvPIT} \) for all the different filters is almost 80%, which means with a higher number of processors, the percentage of idle time increases significantly.

Again, we conclude that the topology of the search space could be the major cause of the limitation in the scalability, and bigger filters with wider search spaces will have bigger speed-ups. But there is still the possibility that the load balancing is not being executed in the best way, due to the problems previously mentioned. Therefore, another load balancing strategy was implemented and is presented in the next section.
5.4.2 Quasi-Liberal Work Forwarding

As previously mentioned, this load balancing strategy was implemented in order to improve the idle time of each processor during computation and improve the scalability. This strategy was based on the previous one (Section 5.4.1), with a small difference. Upon a work request, the processor searches for the first unvisited subtrees (or branches), at the lowest depth $d_l$. Like in the previous strategy, if there is more than one subtree available at depth $d_l$, it keeps the current branch/subtree, and sends all the others at this level, i.e., it keeps $V_{d_l}$ and sends $[V_{d_l} + 1, V_{d_l}^u]$, and assigns $V_{d_l}$ to $V_{d_l}^u$. If the only branch available is the current branch, i.e., $d_l = d$ and $V_{d_l} = V_h$, the processor keeps it and continues to work on its subtree. When reaching the next level ($d + 1$), the process is repeated, which means that, if the processor computes more than one node as a child ($V_{d+1} < V_{d+1}^u$), then it will send the work $([V_{d+1} + 1, V_{d+1}^u])$ to the requester. Although in this strategy each processor can still send the majority of its work, this simple change will diminish the number of communications, since a processor only sends work when it has more than one branch available, while the requesting processor has to wait for the sender to have more work, or for the computation to end.

The results are shown in Figures 5.5 and 5.6. As we can see from Figure 5.5, at the beginning (reducing number of processors) the results are slightly better than with the previous load balancing strategy ($S_{QLWF} \geq S_{LWF}$, for $P < 12$). Again, this supra-linear behaviour is caused by the pruning of the SIREN algorithm. Also, since upon a work request each processor will keep its last branch and compute it instead of sending it, we conclude that there are less communications before reaching a solution, which explains why the results were slightly better for a small number of processors.

However, for a number of processors greater than 12, the behaviour is different, since smaller filters reach an upper limit faster than bigger filters. For example, when $P=16$, the speed-up for the filter with $N=38$ is approximately 10, while for the filter with $N=43$ the speed-up is closer to 19. However the scalability problem continues to exist and it even gets worse, since the speed-up starts reaching its limit when $P=12$. Note that the behaviour observed of the idle time for each processor was very similar to the
previous strategy, therefore it is not shown.

It can be concluded that in each case the scalability limitation is caused by the topology of the search space, that limits the number of processors and does not allow greater improvement. This implies that filters with bigger length \( N \) and different topologies will have a higher/better speed-ups.

### 5.4.3 Fixed Depth BFS

For the strategies presented in 5.4.1 and 5.4.2, no special attention were given to the initial work division. As such, in an attempt to obtain better results based on a more balanced division from the start of the computation, a breadth-first search (BFS) approach was implemented. This approach uses both, DFS and BFS. Basically, at the beginning of computation, every processor starts redundant work, in BFS, until a they reach a pre-established maximum depth, \( d_{\text{max}} \), where each processor continues its own work (different search branches) using DFS, with the LWF load balancing strategy.

The original idea, was that every processor had to compute redundant work, until there were enough work to split among the processors, i.e., the sum of all the differences between the vertices \( N_{V_d} \)

\[7N_{V_d} = \sum_{i=0}^{m} (V_{d}^h - V_{d}^l), \text{ where } m \text{ is the total number of vertices at depth } d.\]
subtrees) would be greater than the number of processors \( P \) (\( N_{V_d} > 2P \) or \( N_{V_d} > 3P \)). However, it was concluded that there were cases where this condition was never satisfied, since the search space did not grow sufficiently in width (\( N_{V_d} < 2P \)), as previously mentioned. Once again we conclude that the topology of the search space could be the cause of the ceiling value reached by the previous strategies. So, in this case it was concluded that each processor should do redundant work until they reach a maximum depth of 2 (\( d_{\text{max}} = 2 \)), then the available work is divided among the processors and the DFS begins.

In order to implement BFS, the code had to be change significantly. In DFS, at a certain depth it is only necessary to keep the value of the current vertex and its upper bound, as well as previous decisions, which is done using the matrix \textit{search\_lbas} (see 5.3). Every time \( V_d \) is assigned to a new value, the previous one from the same depth is discarded. For example, at depth \( d \), if \( V_d \) is pruned (it leads to a solution worse then the current best) and \( V_d < V_d^n \), it will be incremented (\( V_d = V_d + 1 \)), so the previous \( V_d \) and all its children are discarded.

For the BFS, it is necessary to keep every vertex at the current depth, and their respective parents, which means saving all the vertices \( V_{d_0}, \ldots, V_{d_m} \), where \( m \) is the number of vertices at depth \( d \). The best way to implement this is using queues, where we store all the information necessary for the computation. Each instance of the queue will have a depth \( d \), a vertex \( V_d \) and all of his parents (previous decisions).

This implies that more memory is required for this strategy, so the depth of the BFS should be limited. Processors start computing using BFS, until they reach depth 2 (\( d_{\text{max}} = 2 \)), or the amount of work is big enough to split among the processors (\( N_{V_d} > 2 \times \text{num\_proc} \)). Here, \text{num\_proc} represents the total number of processors. Then, each processor determines which branches are assigned to itself to compute.

A good way to realize this division is for each \( P_i \) to have intercalated work: \( i \equiv j \mod \text{num\_proc} \), where \( i \) is the processor id and \( j \) is the index of the available vertex (\( i = 0, \ldots, N_{V_d} - 1 \)). But in this case we want each processor to have consecutive work, since two consecutive vertices could have the same parent, which means less information is stored in the stack. As such, each processor calculates the size of the block that is assigned to it. For example, if \( N_{V_d} = 8 \) and \( P = 3 \), \( P_0 \) and \( P_1 \) will have blocks of size 3 (3 vertices to compute each) and \( P_2 \) will have a block of size 2.

Then, each vertex and their respective parents (previous decisions) are saved in the stack. Each processor will have his own stack. When the DFS computation starts, each processor will first verify if it has any work in the stack. If is does not have, it will then proceed to make a work request, as in the previous strategies.

The results obtained for this load balancing strategy are shown on the graphs in Figures 5.7 and 5.8. From these graphs it becomes clear that the results were not better than the previous ones, since a ceiling value for the speed-up becomes evident sooner (when \( P=12 \)). Also, for a number of processors higher than 16, the computation time becomes much slower.

This result could be caused by the fact that in this strategy, a solution that enables pruning is found latter than in the previous strategies, so the search space could become bigger. Basically, in the previous strategies one of the processors will reach the leaf of the search space and find a solution faster than the
other processors, which means the others can perform pruning. In this case, several processors could reach the leaf at the same time, meaning that other worse solutions were found, without performing pruning (the search space will be bigger). Also, there is a overhead of having two different search methods, since a transition from BFS to DFS is necessary. The fact that the search space may not grow sufficiently in width could worsen this problem. In the worse case scenario, if at depth 2 only one or two branches are available for work, this strategy becomes similar to the previous ones, but since it started the search in BFS it will reach the current depth of computation latter than the previous strategies. Also, there will still be a lot of processors idle and waiting for work.

Again, this result supports the idea that the search space topology has great influence on the results of an optimized SIREN algorithm. As mentioned in Section 4.3, as the values of coefficients are determined, the intervals between the lower and upper bounds of coefficients are reduced when compared to those in the original $H^l$ and $H^u$. This happens because SIREN's DFS method is equipped with techniques that order the filter coefficients, determine their lower and upper bounds dynamically, and prune the search space, ensuring the minimum solution with one leaf at the final depth and the least possible amount of branches.

Figure 5.7: Speed-Up of Fixed Depth BFS Load Balancing

Figure 5.8: Average Speed-Up of Fixed Depth BFS Load Balancing
Table 5.2: Specifications of random Symmetric FIR filters used during the testing phase.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Type</th>
<th>( N )</th>
<th>( \omega_p )</th>
<th>( \omega_s )</th>
<th>( \delta_p )</th>
<th>( \delta_s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F35</td>
<td>Low-pass</td>
<td>30</td>
<td>0.3\pi</td>
<td>0.5\pi</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>F40</td>
<td>Low-pass</td>
<td>38</td>
<td>0.3\pi</td>
<td>0.5\pi</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>F43</td>
<td>Low-pass</td>
<td>43</td>
<td>0.15\pi</td>
<td>0.3\pi</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>F50</td>
<td>Low-pass</td>
<td>50</td>
<td>0.1\pi</td>
<td>0.3\pi</td>
<td>0.001</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Table 5.3: Specifications of benchmarks Symmetric FIR filters.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Type</th>
<th>( N )</th>
<th>( \omega_p )</th>
<th>( \omega_s )</th>
<th>( \delta_p )</th>
<th>( \delta_s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Low-pass</td>
<td>15</td>
<td>0.2\pi</td>
<td>0.8\pi</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>G1</td>
<td>Low-pass</td>
<td>16</td>
<td>0.2\pi</td>
<td>0.5\pi</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Y1</td>
<td>Low-pass</td>
<td>30</td>
<td>0.3\pi</td>
<td>0.5\pi</td>
<td>0.00316</td>
<td>0.00316</td>
</tr>
<tr>
<td>Y2</td>
<td>Low-pass</td>
<td>38</td>
<td>0.3\pi</td>
<td>0.5\pi</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>A</td>
<td>Low-pass</td>
<td>59</td>
<td>0.125\pi</td>
<td>0.225\pi</td>
<td>0.01</td>
<td>0.001</td>
</tr>
<tr>
<td>S2</td>
<td>Low-pass</td>
<td>60</td>
<td>0.042\pi</td>
<td>0.14\pi</td>
<td>0.012</td>
<td>0.001</td>
</tr>
<tr>
<td>L2</td>
<td>Low-pass</td>
<td>63</td>
<td>0.2\pi</td>
<td>0.28\pi</td>
<td>0.028</td>
<td>0.001</td>
</tr>
</tbody>
</table>

In conclusion, the reason for the previous strategies having supra-linear speed-ups is the pruning performed by the algorithm, which is also the main reason for the limited scalability, since when pruning is performed in parallel the best solution will be found much faster and the size of the search space will be much smaller.

5.5 Summary on the Experimental Results

This section presents a summary on the experimental results obtained. Table 5.2 shows the specifications of 4 randomly generated symmetric FIR filters that were used for the experimental tests and speed-ups from previous sections. Table 5.3 shows the specifications of 10 symmetric FIR filters which are commonly used in evaluation of FDO algorithms.

Figure 5.9 shows the average speed-ups obtained for all the three load balancing strategies that were implemented. In this figure it becomes obvious that the LWF load balancing was the best strategy for a parallel implementation of the SIREN algorithm, since not only we obtained better speed-ups for the same filters and number of processors, but it also proved to be more scalable. This is due to the fact that, despite the bad initial division (delay for the work to reach all processors), this strategy took the most advantage of the pruning performed by SIREN, since the best solution is found much faster and the search space will be significantly diminished. Again, we conclude that the limitations on the scalability is caused by the topology of the search space and that for bigger filters a higher number processors could be used.

Table 5.4 was build from the results shown in [10] and presents the results of SIREN, NAIAD [10], and other algorithms whose results were taken from [12, 19, 23]. In order the compare the results of these algorithms with the ones from the optimized SIREN and conclude on his benefits, the results of SIREN C version and parallel versions were added to the table. Here, SIREN denotes SIREN MATLAB version, SIREN\textsubscript{C} denotes the SIREN C version, SIREN\textsubscript{LW} and SIREN\textsubscript{LW} denote the LWF parallel version.
Figure 5.9: Average Speed-Up for all the Load Balancing Strategies Implemented

of SIREN, using respectively 16 and 32 processors. In this table, BST and TT denote respectively the CPU time required to find the best solution and the total CPU time. For each filter, the FDO methods were sorted according to their results on i) EWL, ii) TA, iii) TT, and iv) BST in descending order. For the table, it is shown only the best of the three implemented parallel solutions for each filter. Note that the CPU time limit for SIREN and SIREN parallel was 2 days and 1 day, respectively.

By analysing the table, we conclude that on filters including less than 20 coefficients, X1 and G1, the best results at the best CPU times were found by SIREN_C. On filters including less than 40 coefficients, Y1 and Y2, SIREN_16 finds a solution with the minimum number of operations and minimum EWL value much faster than SIREN MATLAB version and all the other algorithms. This experiment indicates that the parallel version of SIREN, not only finds the minimum solution, but also finds it in the least amount of time when compared with the other algorithms.

On filters including around 60 coefficients, A, S2, and L2, SIREN_32 could ensure the minimum quantization for the filter S2, where it finds a better solution than other FDO algorithms and in a reasonable time (≈ 2h). For this filter SIREN_MATLAB could not guarantee that the solution found was minimum, since it reached the time limit. For the filters A and L2, although SIREN_32 could not ensure the minimum quantization due to its CPU time limit (1 day), it found better solutions than the other algorithms, except for the L2 case where [23] finds a better solution.

With this optimization it is still not possible to apply SIREN to filters including more than 100 coefficients, since the CPU times to find a solution are greater than the time limit. However, if the time limit is sufficiently increased, the algorithm will eventually find a solution or even guarantee the minimum solution, and with more resources (more processors), the time needed to achieve it will be smaller.

This leads us to conclude that for filters including less than 40 coefficients, the new version of SIREN should always be preferred over other existing algorithms. For filters including up until 60 coefficients, the minimum solution is found in a reasonable amount of time for some cases, and in other cases a better solution than other existing algorithms is found in the time limit. Also, for even bigger filters, the higher the number of processors the better for the performance and if the limit is increased sufficiently,
Table 5.4: Summary of FDO Algorithms on FIR Filters of Table 5.3.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Method</th>
<th>EWL</th>
<th>MA</th>
<th>SA</th>
<th>TA</th>
<th>BST</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1 (N=15)</td>
<td>[20]</td>
<td>13</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>1m3s</td>
<td>1m3s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>&lt;1s</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>&lt;1s</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>NAIA_{c}</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>&lt;1s</td>
<td>–</td>
</tr>
<tr>
<td>G1 (N=16)</td>
<td>[23]</td>
<td>7</td>
<td>2</td>
<td>13</td>
<td>15</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>6</td>
<td>3</td>
<td>15</td>
<td>18</td>
<td>50s</td>
<td>50s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>6</td>
<td>2</td>
<td>15</td>
<td>17</td>
<td>&lt;1s</td>
<td>&lt;1s</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>6</td>
<td>2</td>
<td>15</td>
<td>17</td>
<td>&lt;1s</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Y1 (N=30)</td>
<td>[23]</td>
<td>10</td>
<td>6</td>
<td>23</td>
<td>29</td>
<td>–</td>
<td>21m30s</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>9</td>
<td>7</td>
<td>23</td>
<td>30</td>
<td>5m55s</td>
<td>6m3s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>9</td>
<td>6</td>
<td>23</td>
<td>29</td>
<td>&lt;1s</td>
<td>&lt;1s</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>9</td>
<td>6</td>
<td>23</td>
<td>29</td>
<td>&lt;1s</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>Y2 (N=38)</td>
<td>[12]</td>
<td>12</td>
<td>–</td>
<td>–</td>
<td>39</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>11</td>
<td>9</td>
<td>29</td>
<td>38</td>
<td>15m21s</td>
<td>19m18s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>10</td>
<td>9</td>
<td>29</td>
<td>38</td>
<td>3m52s</td>
<td>4m29s</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>10</td>
<td>9</td>
<td>29</td>
<td>38</td>
<td>2s</td>
<td>6s</td>
</tr>
<tr>
<td>A (N=59)</td>
<td>[12]</td>
<td>10</td>
<td>18</td>
<td>58</td>
<td>76</td>
<td>3h2m</td>
<td>4h14m</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>10</td>
<td>16</td>
<td>56</td>
<td>72</td>
<td>43m34s</td>
<td>44m13s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>10</td>
<td>14</td>
<td>52</td>
<td>68</td>
<td>–</td>
<td>2d2h</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>10</td>
<td>14</td>
<td>52</td>
<td>68</td>
<td>14h57m</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>NAIA_{c}</td>
<td>10</td>
<td>16</td>
<td>52</td>
<td>68</td>
<td>–</td>
<td>2d</td>
</tr>
<tr>
<td>S2 (N=60)</td>
<td>[23]</td>
<td>11</td>
<td>27</td>
<td>59</td>
<td>86</td>
<td>23m</td>
<td>27m</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>10</td>
<td>17</td>
<td>59</td>
<td>76</td>
<td>–</td>
<td>16h42m</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>9</td>
<td>14</td>
<td>57</td>
<td>71</td>
<td>16h4m</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>9</td>
<td>14</td>
<td>57</td>
<td>71</td>
<td>20m20s</td>
<td>2h7m</td>
</tr>
<tr>
<td>L2 (N=63)</td>
<td>[12]</td>
<td>11</td>
<td>17</td>
<td>60</td>
<td>77</td>
<td>55m1s</td>
<td>1h4m</td>
</tr>
<tr>
<td></td>
<td>NAIAD</td>
<td>10</td>
<td>18</td>
<td>62</td>
<td>80</td>
<td>26m</td>
<td>54m</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>10</td>
<td>16</td>
<td>60</td>
<td>76</td>
<td>1d23h</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>SIREN_{c}</td>
<td>10</td>
<td>16</td>
<td>60</td>
<td>76</td>
<td>5h47m</td>
<td>1d</td>
</tr>
<tr>
<td></td>
<td>NAIA_{c}</td>
<td>10</td>
<td>17</td>
<td>56</td>
<td>73</td>
<td>–</td>
<td>16h28m</td>
</tr>
</tbody>
</table>

A minimum solution will be eventually found. As such, when the objective is obtaining the minimum solution without much concern for the CPU time and resources, SIREN should also be chosen as an FDO algorithm.
Chapter 6

Conclusions

This thesis addressed the parallelization of SIREN, an exact filter design optimization algorithm, in a distributed system. The FDO problem is generally known as the problem of optimizing the number of operations in the FIR filter design, while satisfying the filter constraints. SIREN is an algorithm that uses a depth-first search method to find the fewest operations in the shift-adds design of the coefficient multiplications. However, it can only handle filters with a small number of coefficients. In this report it was presented some background on SIREN algorithm, including linear programming, multiple constant multiplications and FDO problem.

In order to benefit from SIREN characteristics for larger filters, i.e., obtain solutions with minimum design complexity for a higher number of coefficients, the need for optimizing the algorithm arose. This work consisted in the exploit of this parallelization problem and the search for the most efficient solution. As such, background on parallel computing, MPI and parallel DFS algorithms were also given.

Since SIREN uses a DFS method, it was possible to parallelized it by diving the search space among a number of processors. The search space is very unbalanced, so the division has to be done dynamically, so that the workload can be divided as evenly as possible amongst the available processors. SIREN was originally written in MATLAB, so the first step was to implement a SIREN C version, which greatly improved the performance of the algorithm, reaching an average speed-up of 3. After the C version, three load balancing strategies were implemented for the parallel version, which were called: Liberal Work Forwarding (LWF), Quasi-Liberal Work Forwarding (QLWF) and Fixed Depth Breadth First Search (FDBFS) load balancing strategies.

The LWF load balancing proved to be the best strategy to parallelize the SIREN algorithm, since the speed-ups obtained were higher in comparison with the other strategies. For this strategy, each processor sends the work to the next processor and receives from the previous one (processors organized into a ring). The work being sent is either a large part of his work, which is a set of unvisited vertices at the lowest level, or all of his work, in the case it had only one vertex available. The speed-up showed a behaviour slightly better than linear for a small number of processors (less then 16), but a ceiling value was always reached for more processors, meaning that the strategy implemented is not completely scalable. This strategy has a few problems, namely the fact that the ring topology causes a delay on the work
reaching all processors and a great amount of work is sent every time, causing some processors to be idle during long period of times. However, the results were very good, thanks to the pruning performed by the algorithm. Due to the pruning, every time a solution is found, the search space becomes smaller and in this case, since the best solution is found rather quickly, the search space will be significantly reduced, which explains the great speed-up that was obtained.

With this optimization, SIREN can now be used with filters with a larger number of coefficients, reaching even 60, and the computation becomes more efficient and faster for smaller filters. The speed-ups are very good for a certain number of processors, but a ceiling value is always reached at some point. The limitation on the scalability of this strategy is caused by the nature of the search space that is being constructed, since in the SIREN algorithm the intervals on each depth are reduced (previous decisions are fixed), the lower and upper bound are determined dynamically and pruning is performed. This means that the search space will not grow significantly in width, and as such, depending on the size of the filter, it won’t require a big number of processors to reach the best computation time.

As such, we conclude that the pruning not only is the cause for the great results (supra-linear speed-ups), but also justifies the limitation on the scalability, since for a limited sized search tree, only a limited number of processors will increase the performance.

As future work, the parallelization technique could be improved by revising the initial work division, for example, by making intercalated processors compute different parts of the search tree from the start. Also, making each processor exchange work with more processors, and determine the amount of work to be sent, instead of all the work, to guarantee a better load balancing. This would allow the use of SIREN algorithm to design even larger filters \((N > 60)\) in a reasonable amount of time, that only algorithms which don’t guarantee the minimum complexity are able to compute.
Bibliography


