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).

Index Terms—Parallel depth-first search, distributed computing, MPI, filter design optimization problem, finite impulse response filter, multipliertless design.

I. INTRODUCTION

Digital filtering is an important operation in digital signal processing (DSP), which is usually realized using infinite impulse response (IIR) filters or finite impulse response (FIR) filters. DSP applications usually prefer a FIR filter over an equivalent IIR filter, due to its output stability and phase linearity properties [1], despite requiring a larger number of coefficients. The output of an N-tap FIR filter $y(n)$ is computed as:

$$y(n) = \sum_{i=0}^{N-1} h_i \cdot x(n - i)$$  \hspace{1cm} (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) 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 the transposed form realization of (1), which is shown in Figure 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. The MCM operations are generally implemented using a shift-adds architecture [2], since the realization of a multiplier in hardware is expensive in terms of area, delay and power.

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 number of adders/subtracters. SIREN [3] 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. Since SIREN guarantees the minimum design complexity, its optimization could bring some benefits. Moreover, the algorithm uses a DFS method, which makes it fit for parallelization. With a larger number of coefficients it is possible to describe a FIR filter closer to the desired (ideal) response, so the parallel implementation should be done on distributed-memory systems, which can allow significant scalability.

Given the algorithms and results obtained in [3], this work aims to optimize SIREN 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 this parallelization problem and a solution is proposed, with three different different load balancing strategies.

The rest of this article is organized as follows. Section II presents the background concepts for the FDO problem. Section III presents the background on parallel computing. Section IV describes the SIREN algorithm and its DFS method. Section V presents the implemented solution for the parallelization, including three different load balancing strategies and a summary on the obtained results. Conclusions are given in Section VI.
II. FILTER DESIGN OPTIMIZATION PROBLEM

This section gives the background for the FDO problem.

A. Linear Programming

Linear Programming (LP) is 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 (2), \( x \) 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.

B. Filter Design Optimization

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

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

where \( M = (N-1)/2 \) and \( d_i = 2 - K_{i,M} \) with \( K_{i,M} \) as the Kronecker delta\(^1\), \( h_i \in \mathbb{R} \) 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. Assuming the desired pass-band and stop-band gains are equal to 1 and 0, respectively, a low-pass FIR filter must satisfy the following constraints [5]:

\[
\begin{align*}
1 - \delta_p & \leq G(\omega) \leq 1 + \delta_p, \quad \omega \in [0, \omega_p] \\
-\delta_s & \leq G(\omega) \leq \delta_s, \quad \omega \in [\omega_s, \pi]
\end{align*}
\]

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 (Figure 2). To compensate the gain in the filter design, 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 [6]:

\[
\begin{align*}
s(1 - \delta_p) & \leq G(\omega) \leq s(1 + \delta_p), \quad \omega \in [0, \omega_p] \\
s(-\delta_s) & \leq G(\omega) \leq s(\delta_s), \quad \omega \in [\omega_s, \pi] \\
s^l & \leq s \leq s^u
\end{align*}
\]

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 [7], McClellan-Parks-Rabiner algorithm [8], or linear programming [9];

\[\text{ii) realize the multiplier block of the FIR filter, using the minimum number of adders/subtracters, which will be described in Section II-C.}\]

There are many FDO algorithms with different objectives, whether it is to obtain the least number of filter coefficients or the least computational time. The SIREN algorithm [3] aims to determine the filter coefficients that respect the filter constraints, and, at the same time, minimize the hardware, by reducing the number of adder-subtracters. Other filter design algorithms [10] commonly used in MATLAB design filters given the frequency and amplitude constraints, but are less concerned in terms of hardware (designs are not optimized in terms of area).

C. Multiplierless Design of the MCM Block

The constant multiplications of an MCM block (see Figure 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) [11]. This method begins with the definition of the constants under a number representation, like canonical signed digit (CSD)\(^4\) 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 [3]. The decompositions of the constant multiplications \( y_0 = 21x \) and \( y_1 = 53x \) in CSD are listed as:

\[
\begin{align*}
y_0 &= 21x = (10101)_{\text{CSD}} x = x \ll 4 + x \ll 2 + x \\
y_1 &= 53x = (1010101)_{\text{CSD}} x = x \ll 6 - x \ll 4 + x \ll 2 + x
\end{align*}
\]

To reduce the complexity of the MCM design even further, the partial products among the constant multiplications can

\[
\text{A integer can be written in CSD using } n \text{ digits as } \sum_{i=0}^{n-1} d_i 2^i, \text{ where } d_i \in \{1, 0, -1\} \text{ and } 1 \text{ denotes } -1, \text{ with } 0 \leq i \leq n-1. \text{ Under CSD, nonzero digits are not adjacent and the minimum number of nonzero digits is used.} \]

\[\text{1The minimization objective can be easily converted to a maximization objective by negating the cost function. Less-than-than and equality constraints are accommodated by equivalences, } A \cdot x \leq b \Longleftrightarrow -A \cdot x \geq -b \text{ and } A \cdot x = b \Longleftrightarrow (A \cdot x \geq b) \land (A \cdot x \leq b), \text{ respectively.}\]

\[\text{2The frequency response of an asymmetric filter is shown in [4] and [3].}\]

\[\text{3The } K_{a,b} \text{ function is 1 when } a \text{ is equal to } b. \text{ Otherwise is 0.}\]

\[\text{4An integer can be written in CSD using } n \text{ digits as } \sum_{i=0}^{n-1} d_i 2^i, \text{ where } d_i \in \{1, 0, -1\} \text{ and } 1 \text{ denotes } -1, \text{ with } 0 \leq i \leq n-1. \text{ Under CSD, nonzero digits are not adjacent and the minimum number of nonzero digits is used.} \]
be shared. For the shift-adds design of the MCM block, the existing methods can be grouped in two categories: common subexpression elimination (CSE) [12], [13], [15] and graph-based (GB) [14], [16], [17] techniques. 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 [3], the exact CSE algorithm [13] finds the most common subexpression $5x = (101)_{\text{CSD}}X$ (Figure 3(a)), with constants defined under CSD, to obtain the minimum solution with 4 operations. The exact GB method of [14] finds the intermediate subexpression $3x$ and then obtains the minimum solution with 3 operations (Figure 3(b)). The minimum adder-steps of an MCM block instance with N constants is given by $MAS_{\text{MCM}} = \max_i \lfloor \log_2 S(h_i) \rfloor$, where $h_i$ represents a single constant multiplication, $S(h)$ is the number of nonzero digits in the CSD representation and $0 \leq i \leq N - 1$ [15]. For the example, the minimum adder-steps of both $21x$ and $53x$ is 2. The approximate algorithm [14] modified to handle a delay constraint finds a solution with 4 operations when $dc = 2$ (Figure 3(c)). On the other hand, the exact GB algorithm [14] finds a solution with one more adder-step, but one less operation (Figure 3(b)).

III. PARALLEL COMPUTING

Parallel computing is used to reduce the time needed to solve a single computational problem. A parallel computer can be classified in two categories: multicomputers (distributed-memory systems) and centralized multiprocessors (shared-memory systems) [18]. A distributed-memory system is composed out of multiple computers and all the interconnection network. Each processor only has direct access to its own local memory. Processors inter act with each other by passing messages, and there are no cache coherence problems to solve. In contrast, in a centralized multiprocessor all CPUs share access to a single global memory, which supports communication and synchronization among processors.

Parallel programming allows us to explicitly indicate how different portions of the computation may be executed concurrently by different processors. A standard library specification for parallel programming in distributed-memory systems is the message passing interface (MPI). This section presents some basic concepts regarding MPI and its use, and after presents an overview of existing parallel algorithms for DFS.

A. Message Passing Interface

MPI (Message Passing Interface) is a standard specification for message passing libraries [18], [19]. 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. Moreover, MPI involves explicit parallelism, which leads to a better performance; makes available a number of optimized collective communication routines; and in general, communications and computation can be overlapped.

In distributed-memory systems, it is required an interconnection network (channel) which supports message passing between processors, allowing every processor to communicate each other and send some of its local data. 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.

MPI makes available several functions for the communication between processors, such as MPI_Send and MPI_Recv, among others, and defines MPI_COMM_WORLD as 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 MPI_Bcast (to send data to all other processes), MPI_Gather (to gather the local data of each process into a larger array), among others. MPI also offers non-blocking routines, that return almost immediately, without waiting for the communication event to complete. They are used to overlap computation with communication, to improve the performance.

B. Parallel DFS

Depth-first search (DFS) is an algorithm for traversing tree or graph data structures. Starting at the root, it searches as deeply as possible along each branch before backtracking and searching another path. Since different paths only depend on previous decisions (branches are independent), it is possible to explore the search tree in parallel. Many algorithm use DFS, so there are several techniques for performing parallel DFS [20]–[23].

According to [20], [22], 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. Each processor usually keeps an open and a closed lists of unvisited and visited nodes, respectively. When a processor has finished searching its part of the search space, it tries to get an unvisited node from the
other processors. Some DFS algorithms search for a specific goal. When it is found, the processor uses a non-blocking send to tell the other processors, and the search is halted. Other algorithms (like SIREN) explore the entire search space to guarantee the best solution, so a newly found best solution is broadcast to all other processors, and the computation continues. Eventually all the processors run out of work, and the search will terminate.

For each processor, the (part of) tree to be searched can be 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 requests some of the untried alternatives of another processor’s stack.

In the implementation described in [20], 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. Once a processor finishes the search in its available space, it calls a routine to request for more work. Whenever a processor needs work, it sends a request to one of its neighbors [20]. The splitting strategy is also an important parameter for the routine to request/get work is architecture dependent.

Requesting for work and performing a termination test involves communication with other processors. So, by restricting these communications with immediate neighbours only, the overhead can be reduced. Whenever a processor needs work, it sends a request for work to one of its neighbours [20]. The splitting strategy is also an important parameter for the performance on a given architecture and a given problem. It 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 [20], [22]. The termination detection is also an important subject for the DFS parallelization and will be addressed in section V.

IV. SIREN

The SIREN algorithm [3], [24] takes a 5-tuple \( fspec \) denoting the filter specifications as input and returns a set of filter coefficient yielding a minimum number of adders/subtractors in the filter design and satisfying the filter constraints. Its pseudo-code is shown in Algorithm 1, where \( Q \) stands for the quantization value used to convert floating-point numbers to integers. SIREN was described in detail in [3], using a symmetric FIR filter with \( fspec \) \( (N=8, \omega_p=0.2\pi, \omega_s=0.7\pi, \delta_p=0.01, \quad \delta_s=0.01) \) 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 \( s \) (scaling factor to compensate the passband gain - see Section II-B) using the ComputeBounds function. To find the lower bounds of coefficients the following LP problem is solved for each coefficient \( h_i \):

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

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 \( h^l_i \). 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 \( h^u_i \). Thus, the sets \( h^l_i \) and \( h^u_i \) 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 \( 2[M]+4 \), where \( M = (N-1)/2 \) (see Section II-B).

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

Second, the OrderCoefs function finds an ordering of coefficients to be used in its DFS method while constructing the search tree (described ahead). 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 \( O \). For the example, the ordering of coefficients is \( O = \{1, 0, 2, 3\} \).

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 \( H^l(H^u) \). The validity of these sets \( H^l \) and \( H^u \) is tested by the CheckValidity function by simply checking each coefficient if \( H^l_i \) is less than or equal to \( H^u_i \). If they are not valid (\( H^l_i > H^u_i \)), this function returns zero. In this case, \( Q \) is increased by one, \( H^l_i \) and \( H^u_i \) are updated, and the CheckValidity function is applied again.

If \( H^l_i \) and \( H^u_i \) are valid, the DFS method is then applied, exploring all possible values of each coefficient in between \( H^l_i \) and \( H^u_i \), to find a set of filter coefficients which respects the

Algorithm 1 THE SIREN algorithm

SIREN(\( fspec \))
1: \( Q = 0, \ sol = \{\} \)
2: \( (h^l, h^u, s^l, s^u) = \text{ComputeBounds}(fspec) \)
3: \( O = \text{OrderCoefs}(h^l, h^u) \)
4: \( \text{repeat} \)
5: \( Q = Q + 1, H^l = [(h^l \cdot 2^Q)], H^u = [(h^u \cdot 2^Q)] \)
6: \( \text{if} \ CheckValidity(H^l, H^u) \text{ then} \)
7: \( \quad sol = \text{DFS}(fspec, O, Q, H^l, H^u) \)
8: \( \text{until} \ sol \neq \emptyset \)
9: \( \text{return} \ sol. \)
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, \( H^l \) and \( H^u \) 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, 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). Similar to \( Q \), the solution quality of an FDO algorithm is evaluated by the effective wordlength (EWL) of a set of coefficients [5], [25], [26], computed as \( \max\{\lceil\log_2|c_i|\rceil\} \) with \( 0 \leq i \leq N-1 \) when fixed-point coefficients are considered.

A. SIREN DFS 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 \( d = 1 \), the DFS method assigns \( H^l_{O(1)} \) and \( H^u_{O(1)} \) to \( V^l_1 \) and \( V^u_1 \), respectively, and sets the vertex of the search tree \( V_1 \) to \( V^l_1 \). At any depth greater than \( 1 \), \( d > 1 \), although the lower and upper bounds of a vertex can be taken from \( H^l \) and \( H^u \), respectively, tighter lower and upper bounds can be computed, since the values of coefficients at previous depths have been determined and fixed. The lower boundary 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.

\[
\begin{align*}
\text{minimize: } & f = h_{O(d)} \\
\text{subject to: } & s \left(1-\delta_s\right) \leq G(\omega)/2Q \leq s \left(1+\delta_p\right), \quad \omega \in [0, \omega_p] \\
& -s \left(\delta_s\right) \leq G(\omega)/2Q \leq s \left(\delta_s\right), \quad \omega \in [\omega_s, \pi] \\
& H^l_i \leq h_i \leq H^u_i, \quad i \in O(d), O(\lceil M \rceil + 1) \\
& s^l \leq s \leq s^u \\
& h_{O(1)}, \ldots, h_{O(d-1)}: \text{determined and fixed} \\
\end{align*}
\]

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 \( V_d^u \). 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 \( \lceil M \rceil + 1 \) 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). While MA is found using the exact MCM method [14], 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 [27]. 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 \( Q = 5 \), 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 \( Q = 5 \) is shown in Figure 4. 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

![Fig. 4: Search tree formed by the DFS method.](attachment:image.png)
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 their lower and upper bounds are reduced when compared to those in the original $H^l$ and $H^u$. The DFS techniques, that order the filter coefficients, determine the lower and upper bounds of coefficients dynamically, and prune the search space, ensures the minimum solution, visiting a minimum number of leafs (at the final depth) and branches.

The performance of SIREN depends heavily on the minimum quantization value $Q$, the filter length $N$, and the exact MCM algorithm [14]. 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 [14]. 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 [14]. 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.

V. SIREN PARALLELIZATION

This section presents and discusses the proposed and implemented solutions, and the results obtained for the optimization of the SIREN algorithm using parallel computing. Section V-A presents a solution for the problem and an overall description of the parallelization method implemented. Sections V-B, V-C and V-D are 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 V-E.

SIREN was originally written in MATLAB, so the first was to implement a C version of the algorithm. Simply with this conversion, the performance of SIREN was greatly improved, resulting in an average speed-up\(^5\) of 3.

A. Implemented Solution

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 (splitting strategy) proved to be a major concern, so its discussed in the next subsection, where different strategies with their respective results are presented.

For the serial implementation of the SIREN algorithm, after a valid quantization value $Q$ is found and the upper and lower bound sets ($H^l$ e $H^u$) 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 $V_d$ and the already determined and fixed coefficients, $h_{O(1)}...h_{O(d-1)}$. This information is stored in a matrix $\text{search\_lubas}$ of size $2 \times \text{Max\_Depth}$\(^6\), where the value in $\text{search\_lubas}[0][d]$ corresponds to the current vertex being processed, $V_d$, and the upper bound of the vertex, $V^u_d$ is stored in $\text{search\_lubas}[1][d]$. Also the values of previous decisions $h_{O(1)}...h_{O(d-1)}$ are stored in $\text{search\_lubas}[k][0,\ldots,d-1]$, 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 the matrix $\text{search\_lubas}$ and the current depth $d$. After sending part of its work, a processor only needs to update $\text{search\_lubas}$ 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 for the first two strategies implemented, since all the necessary information are stored in $\text{search\_lubas}$.

In order to minimize overhead, each processor only has 2 neighbours. The processors can be organized into a ring, which means that each processor only sends work to the next one, and only receives from the previous one. 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 before computing a vertex in order to perform pruning. And second, when it is idle, since all nodes have to know if they have the best solution, so at the end of computation they can send it to $\text{Proc}_0$, which will realize the final filter design.

Finally, the termination detection was implemented using a Modified Dijkstra’s Token Termination Detection Algorithm (MDTTDA)\(^7\). 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: black or white. Initially, all processors are in state white and the algorithm proceeds as follows:

1) When process $\text{Proc}_0$ becomes idle, it makes itself white and initiates a white token to be sent to process $\text{Proc}_1$\(^8\).

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.

\(^5\) Speed-up = $T_{\text{MATLAB}}/T_{\text{C}}$, where $T_{\text{C}}$ is the speed-up of the C version, $T_{\text{MATLAB}}$ and $T_{\text{C}}$ are the CPU times of the original MATLAB version and C version of SIREN, respectively

\(^6\) $\text{Max\_Depth} = \lceil M \rceil + 1$, for symmetric filters

\(^7\) 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

\(^8\) The token travels in the sequence $\text{Proc}_0, \text{Proc}_1, ..., \text{Proc}_{p-1}, \text{Proc}_0$
3) If processor $Proc_i$ has the token and becomes idle, it passes the token to $Proc_{i+1}$. If $Proc_i$ is black, token is set to black before it is sent to $Proc_{i+1}$. If $Proc_i$ is white, the token is passed unchanged.

4) After $Proc_i$ has passed the token to $Proc_{i+1}$, $Proc_i$ becomes white.

The algorithm terminates when processor $Proc_0$ receives a white token and is itself idle. $Proc_0$ then proceeds to send an END token, so that all other processors know that the computation is finished.

B. Liberal Work Forwarding

For first load balancing strategy implemented, 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. 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 ($Proc_0$) 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 ($Proc_0, Proc_1, ..., Proc_{n-1}, Proc_0$), there will be a delay until work reaches all nodes, ($Proc_i$ 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 $d_i^0$, where there are unvisited subtrees. If there is more than one subtree available at depth $d_i$, i.e., $V_{d_i}^0 > V_{d_i}$, the processor keeps the current branch/subtree, and sends all the other subtrees at this level, which means it keeps $V_d$ and sends $\{V_u^0+1, V_{d_i}^0\}$, and assigns $V_{d_i}$ to $V_u^0$. If the only branch available is the current one, which happens when $d_i = d$ and $V_d = V_d^0$, 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.

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 GRID\(^{10}\) 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. However, the results obtained were very good, as shown in Figure 5. This figure presents the speed-ups\(^{11}\) obtained from filters with different lengths, $N = \{35, 38, 40, 43\}$, for the MATLAB, serial C version and different number of processors, $P = \{2, 4, 8, 12, 24, 32\}$, where the results from the C version are taken as reference ($S = 1$). From this chart we can conclude that the speed-ups present a growth with $P$ better than linear ($S > P$), until they reach a maximum value or ceiling, $S_{\text{Ceil}}$.

\(^{9}\)The lowest depth $d_i$ starts at 0 and is incremented until unvisited subtrees are found or it reaches depth $d (d_i = \{0, ..., d\})$

\(^{10}\)Cluster from RNL of Instituto Superior Técnico

\(^{11}\) $S = T_{\text{serial}}/T_{\text{parallel}}$, where $S$ is the speed-up, $T_{\text{serial}}$ and $T_{\text{parallel}}$ are the CPU times of the serial and parallel implementation, respectively.

We also notice that larger filters have higher ceilings values, since they reach higher speed-ups for a larger number of processors. This happens because of the pruning. Every time a new solution is found, the search space will be diminished and if the best solution is found quickly, the search space will be significantly smaller. From the previous results, the maximum average speed-up $AvS_{\text{Ceil}}$, is obtained when the number of processors is 16. 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.

There are two reasons that could be the cause of this problem. The first would be the topology of the search tree itself, which does not allow a greater improvement. This is the most plausible theory, since 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 expands more in height than width (see section IV-A). The second reason could be the load balancing technique implemented, namely the fact that there is a delay until every processor starts computing. 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 speed-up 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. From the results 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. 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, so another load balancing strategy was implemented and is presented in the next section.

C. Quasi-Liberal Work Forwarding

This strategy was based at the previous one, with a small difference. Upon a work request, the processor searches for
the first unvisited subtrees (or branches), at the lowest depth $d_1$. If there is more than one subtree available, the process is identical to the previous strategy. If the only branch available is the current branch, i.e., $d_i = d$ and $V_d = V_d^h$, 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 $(V_{d+1} < V_{d+1}^h)$, then it will send the work $([V_{d+1} + 1, V_{d+1}^h])$ to the requester. The requesting processor has to wait for the sender to have more work, or for the computation to end, diminishing the number of communications.

The results are shown in Figures 6. As we can see from Figure 6, at the beginning the results are slightly better than with the previous load balancing strategy ($S_{QLWF} \geq S_{LWF}$, for $P < 12$). However, for a number of processors greater than 12, the behaviour is different, since smaller filters reach an upper limit faster than bigger filters. However the scalability problem continues to exist and it even gets worse, since the speed-up stars reaching its limit when $P=12$. Note that the behaviour observed for the idle times for each processor was very similar to the previous strategy. Once again, 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.

### D. Fixed Depth BFS

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 BFS, until they reach a pre-established maximum depth, $d_{max}$, where each processor continues its own work (different search branches) using DFS, with the LWF load balancing strategy.

For the BFS, it is necessary to keep every vertex at the current depth, and their respective parents $(V_{d0}, ..., V_{dm})$, 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 ($d_{max} = 2$), or the amount of work is big enough to split among the processors ($N_{d} \geq 2 \times num\_proc$). Here, $num\_proc$ represents the total number of processors. Then, each processor determines which branches are assigned to itself to compute. We want each processor to have consecutive work, since two consecutive vertices could have the same parent. As such, each processor calculates the size of the block that is assigned to it. 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 in Figures 7. From this graphs it becomes clear that the results were not better than the previous ones, since the ceiling becomes evident when $P=12$. Also, for a number of processors higher than 16, the computation time becomes even slower.

Again, this result supports the idea that the search space topology has great influence on the results of an optimized SIREN algorithm. 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.

### E. Summary on the Experimental Results

This section presents a summary on the experimental results obtained. Figure 8 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 it resulted in better speed-ups, but it also proved to be more scalable. This is due to the fact that 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 I shows the specifications of 10 symmetric FIR filters which are commonly used in evaluation of FDO algorithms. Table II presents the results of SIREN C and parallel versions, SIREN original MATLAB version, NAIAD, and other algorithms whose results were taken from [3], [5], [26], [28]. SIREN denotes SIREN MATLAB version, SIREN\textsubscript{C} denotes the SIREN C version, SIREN\textsubscript{16} and SIREN\textsubscript{32} denote the LWF parallel version 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. Note that the CPU time limit for SIREN, parallel version of SIREN and NAIAD was 1 day, 2 days and 4 hours, respectively.

The result showed that for filters including less than 40 coefficients, X1, G1, Y1, and Y2, SIREN\textsubscript{C} and SIREN\textsubscript{16} find a solution with the minimum number of operations and minimum EWL value faster than all the other algorithms. On filters including around 60 coefficients, A, S2, and L2, the parallel version of SIREN 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). Although it could not ensure the minimum quantization for the filters A and L2 due to its CPU time limit, it found better solutions than other algorithms. 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 too long.

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. Also, 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 then other existing algorithms is found in the time limit. As such, when the objective is obtaining a better solution without much concern for the CPU time, SIREN should also be chosen as an FDO algorithm.

### Table I: 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>0.0016</td>
<td>0.0016</td>
</tr>
<tr>
<td>Y2</td>
<td>Low-pass</td>
<td>38</td>
<td>0.2( \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>

### Table II: Summary of FDO Algorithms on FIR Filters of Table I.

<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>[29]</td>
<td>13</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>1m3s</td>
<td>1m3s</td>
</tr>
<tr>
<td></td>
<td>NAID</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>&lt;1s</td>
<td>2s</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>&lt;1s</td>
</tr>
<tr>
<td></td>
<td>SIREN\textsubscript{C}</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>&lt;1s</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>G1 (N=16)</td>
<td>[6]</td>
<td>7</td>
<td>2</td>
<td>13</td>
<td>15</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td></td>
<td>NAID</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\textsubscript{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>5m55s</td>
<td>6m3s</td>
</tr>
<tr>
<td></td>
<td>NAID</td>
<td>9</td>
<td>7</td>
<td>23</td>
<td>30</td>
<td>5m55s</td>
<td>6s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>9</td>
<td>6</td>
<td>23</td>
<td>29</td>
<td>2m17s</td>
<td>7m56s</td>
</tr>
<tr>
<td></td>
<td>SIREN\textsubscript{C}</td>
<td>9</td>
<td>6</td>
<td>23</td>
<td>29</td>
<td>8s</td>
<td>19s</td>
</tr>
<tr>
<td>Y2 (N=38)</td>
<td>[30]</td>
<td>12</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>15m21s</td>
<td>19m18s</td>
</tr>
<tr>
<td></td>
<td>NAID</td>
<td>11</td>
<td>9</td>
<td>39</td>
<td>38</td>
<td>15m21s</td>
<td>11s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>10</td>
<td>9</td>
<td>38</td>
<td>38</td>
<td>3m52s</td>
<td>4m29s</td>
</tr>
<tr>
<td></td>
<td>SIREN\textsubscript{C}</td>
<td>10</td>
<td>9</td>
<td>38</td>
<td>38</td>
<td>2s</td>
<td>6s</td>
</tr>
<tr>
<td>A (N=59)</td>
<td>[5]</td>
<td>10</td>
<td>18</td>
<td>58</td>
<td>76</td>
<td>3h2m</td>
<td>4h14m</td>
</tr>
<tr>
<td></td>
<td>NAID</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>54</td>
<td>68</td>
<td>2d</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>SIREN\textsubscript{C}</td>
<td>10</td>
<td>16</td>
<td>52</td>
<td>68</td>
<td>14h57m</td>
<td>2d</td>
</tr>
<tr>
<td>S2 (N=60)</td>
<td>[26]</td>
<td>17</td>
<td>59</td>
<td>86</td>
<td>96</td>
<td>27m</td>
<td>27m</td>
</tr>
<tr>
<td></td>
<td>NAID</td>
<td>16</td>
<td>57</td>
<td>72</td>
<td>80</td>
<td>49m19s</td>
<td>1h10s</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>14</td>
<td>57</td>
<td>71</td>
<td>71</td>
<td>16h4m</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>SIREN\textsubscript{C}</td>
<td>14</td>
<td>57</td>
<td>71</td>
<td>71</td>
<td>20m20s</td>
<td>2h7m</td>
</tr>
<tr>
<td>L2 (N=63)</td>
<td>[25]</td>
<td>17</td>
<td>60</td>
<td>77</td>
<td>80</td>
<td>5m51s</td>
<td>1h4m</td>
</tr>
<tr>
<td></td>
<td>NAID</td>
<td>16</td>
<td>60</td>
<td>76</td>
<td>76</td>
<td>1d23h</td>
<td>2d</td>
</tr>
<tr>
<td></td>
<td>SIREN</td>
<td>10</td>
<td>60</td>
<td>76</td>
<td>547m</td>
<td>1d</td>
<td>1d</td>
</tr>
<tr>
<td></td>
<td>SIREN\textsubscript{C}</td>
<td>10</td>
<td>60</td>
<td>76</td>
<td>547m</td>
<td>1d</td>
<td>1d</td>
</tr>
<tr>
<td></td>
<td>[26]</td>
<td>17</td>
<td>56</td>
<td>73</td>
<td>73</td>
<td>~</td>
<td>16h28m</td>
</tr>
</tbody>
</table>

---

Fig. 8: Average Speed-Up (ALL LB STRATEGIES)

VI. CONCLUSIONS

This thesis addressed the parallelization of SIREN, an exact filter design optimization algorithm, in a distributed system. 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 had to be done dynamically. SIREN was originally written in MATLAB, so the first step was to develop a SIREN C version, which greatly improved the performance of the algorithm, reaching an average speed-up of 3. After, three load balancing strategies were implemented for the parallel version: 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. The
speed-up showed a behaviour slightly better than linear for a small number of processors. This happens thanks to the pruning performed by the algorithm, that limits the search space. However, the strategy is not completely scalable. It was concluded that the limitation on the scalability of this strategy was 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.

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. As future work, the parallelization technique could be improved by revising the initial work division and making each processor exchange work with more processors, to guarantee a better load balancing. This would allow the use of SIREN algorithm to design even larger filters \(N > 60\), that only algorithms which don’t guarantee the minimum complexity are able to compute.

REFERENCES