Digital Signal Processing 93 (2019) 70–86

Contents lists available at ScienceDirect

Digital Signal Processing

www.elsevier.com/locate/dsp

Staggered parallel short-time Fourier transform

Alfonso B. Labao *, Rodolfo C. Camaclang III, Jaime D.L. Caro

Department of Computer Science, College of Engineering University of the Philippines Diliman, Philippines

ARTICLE INFO

Article history:
Available online 19 July 2019

Keywords:
Short time Fourier transform
Signal processing
Parallel algorithms
Fourier transform algorithmics

ABSTRACT

In this paper, we present the staggered parallel short-time Fourier transform, an algorithm that uses a quasi-parallel procedure to compute exact STFT coefficients of 1D signals. The algorithm leverages parallelism with the capacity of feedforward STFT algorithms to re-use prior computations. It performs this by carefully organizing input signals and collecting past computations into 2D memory buffers. Re-using stored information in memory enables fast computation of up to $N/2$ FFTs in parallel. The algorithm's time complexity is at $O[6T]$ under an abstract circuit implementation – achieving a complexity measure that is independent of sample complexity N. Its time complexity is asymptotically equivalent with the best possible exact algorithm of $O[T]$ time complexity, with a constant efficiency at $O[1]$ relative to the best known sequential algorithm. Its efficiency property holds whether in an abstract circuit implementation or in a CPU implementation with limited number of cores. In general, the algorithm consumes less processors than other parallel STFT algorithms but can potentially require more memory. To test the algorithm's properties, we implement several STFT algorithms in a CPU with varying numbers of cores. These algorithms use either FFT, iterative, or feedforward schemes to capture the range of existing STFT algorithms for comparison. From our experimental results, our proposed algorithm has the least running time among exact STFT algorithms, while consuming less CPU processors than other forms of parallel implementations.

1. Introduction

The short-time Fourier Transform developed by Dennis Gabor is frequently used to compute Fourier transforms of non-stationary signals, where parameters such as amplitude and frequency experience non-periodic temporal variations. Non-stationary behavior in signals renders it difficult for simple stationary functions to adequately model the entire length of the signal. However, the STFT addresses these by means of iterated Fast Fourier Transforms (FFT) over fixed sample lengths N. The STFT has had several applications ranging from signal analysis [1], [2], [3], acoustics [4], speech and audio processing [5], [6], power frequency harmonics and DC system analysis [7], [8].

As analyzed in [9], implementations of the STFT algorithm are characterized by: (1) standard FFT butterfly algorithms and their parallel variants [10], [11], or by (2) iterative methods that use computations similar to the Goertzel algorithm [12], [13], [14], [15], [16], [10], [17], or (3) feedforward or pruned STFT methods [9], [16]. Almost all existing STFT implementations are derivatives of these 3 classes. Several parallel variants exist for the first and second groups, i.e. [11], [12], but the third group so far is lacking in parallel implementations.

The first group of algorithms use the standard FFT algorithm. Parallel implementations of these algorithms are heavy on processor consumption and incur several redundant computations. On the other hand, the second group of iterative methods use less computational power and perform the fastest theoretically. However, their recursive computations incur accumulative errors which accumulate over time as per [9]. The third group uses pruned STFT, or as per [9] – the feedforward short-time Fourier transform. The feedforward short-time Fourier transform performs exact computations of the STFT unlike iterative methods that incur errors [12], [13]. At the same time, it re-uses prior computations to avoid redundant computations unlike the first group. This comes at the cost of maintaining an auxiliary memory to store prior computations, but processor and even time complexity improvements between the third and first group may be significant.

In this paper, we propose a quasi-parallel implementation that uses ideas under the third group of STFT algorithms, i.e. feedforward algorithms, [9]. The algorithm uses the structure of the STFT to re-use prior computations and avoid computing a full FFT recursion. To the best of the author's knowledge, the proposed algorithm is the first to propose a systematic quasi-parallelization of feedforward methods since prior algorithms under the third group

* Corresponding author.
E-mail address: alfonso.labao@upd.edu.ph (A.B. Labao).

https://doi.org/10.1016/j.dsp.2019.07.003
1051-2004/

are mostly sequential. We use the same computations as a feedforward STFT, but we present a method to carefully organize input signals and store multiple past information in 2D memory buffers. Re-using information in these 2D memory buffers allows parallel computation of a batch size of $N/2$ FFTs, where N refers to the size of the FFT sample, i.e. sample complexity. However, the algorithm is not considered as a purely parallel implementation since there are interim breaks to update online memory buffers. These online memory buffers store structural computations of consecutive prior FFT's - for use in future computations.

We present in our analysis the correctness and complexity properties of our algorithm under an abstract circuit set-up, where it is assumed that the required number of processors are available at each computational step. In terms of correctness, the algorithm computes exact coefficients unlike the second group (iterative STFT) that incur errors over time [9]. The algorithm also consumes less processors than the first group (purely parallel STFT) since it does not compute full FFT recursions. Its time complexity is among the fastest at $O[6T]$ - asymptotically equivalent with the fastest STFT algorithm (parallel iterative [12]) that performs at $O[T]$. However, we note that the theoretically fastest exact STFT algorithm at $O[T]$ is inefficient and can potentially consume a polynomial number of processors relative to sample complexity N. On the other hand, the proposed algorithm's efficiency is $O[1]$ relative to the best sequential algorithm, indicating that it consumes much less processors than other FFT-based parallel algorithms. The algorithm however can potentially consume more memory since it needs an auxiliary memory buffer to store past computations.

We present analysis on algorithm complexity in the context of a CPU set-up. Experimental results in the CPU agree with the derived theoretical behavior, albeit with some adjustments to take into account limitations in the number of cores and additional overheads in thread creation/synchronization. So far, the proposed quasi-parallel algorithm performs the fastest among the exact STFT algorithms (i.e. first and third groups), and computes exact coefficients unlike the second group. It also consumes less CPU compared to the first group while consuming comparable memory with other parallel implementations.

2. Existing FFT and STFT algorithms

2.1. DFT and FFT butterfly algorithm

Many STFT algorithms rely on the Cooley and Tukey [18] fast Fourier transform [FFT], which is a Decimation in Time algorithm to compute the DFT f of any N-dimensional signal x. Instead of standard DFT computations, the FFT follows a recurrence structure, under the observation that DFT computations from prior stages can be re-used in future stages. This reduces sequential computation time to $O[N \log N]$ from $O[N^2]$ (where time is measured in terms of additions and multiplications). The recurrence of an FFT follows Eq. (1), where f_l denote the computations in the nodes of a butterfly FFT computation structure [11]. Eq. (2) shows f_l as a function of even (E_l) and odd (O_l) portions of the butterfly with twiddle factor (W^l).

$$f_l = \sum_{k=0}^{N-1} x[k]W^{kl} = \underbrace{\sum_{k=0}^{N/2-1} x[2k]W^{(2k)l}}_{\text{DFT of even-indexed part } (E_l)} + \underbrace{\sum_{k=0}^{N/2-1} x[2k+1]W^{(2k+1)l}}_{\text{DFT of odd-indexed part } (O_l)} \quad (1)$$

$$f_l = E_l + W^l O_l \qquad f_{l+\frac{N}{2}} = E_l - W^l O_l \quad (2)$$

2.2. Short-time Fourier transform algorithm

Given non-stationary signals X of length T, the short-time Fourier transform algorithm computes FFTs of sample complexity N, with $T > N$. The STFT of a discrete signal is shown in Eq. (3), where $x \in X$.

$$STFT_{n,k} = \sum_{m=n}^{n+(N+1)} x[m]W^{km} \quad (3)$$

Eq. (3) computes the STFT at frequency kn/N under a sample period of size N, where $k = 0..[N-1]$. As per [9], [12], [10], [13], this procedure could be parallelized. But as per [9], parallel implementations lead to significant memory overhead costs. To save on overhead consumption, iterative methods can be used following a recursive formula (Eq. (4)) from [9].

$$STFT_{n,k} = W^k \Big[STFT_{n-1,k} + x[n-1+N] - x[n-1] \Big] \quad (4)$$

Eq. (4) also corresponds to iterative procedures from [13], [12], which results in very fast STFT computations since it operates in linear time $O[TN]$ under fixed T. However, iterative methods incur accumulation errors as each frequency is updated with incoming values.

2.3. Feedforward short term Fourier transform

The feedforward STFT [9] re-uses prior FFT computations to improve computational efficiency of the STFT. It is slower than iterative methods but does not incur errors. To show this method, let n denote the index over N-sized samples of a signal where $n = 0, 1, 2, ..N$. The FFT butterfly of each N-dimensional sample has s_N stages indexed by s, where $s = 1...s_N = [\log(N) + 1]$. For each stage s, let k denote indices over butterfly nodes. We see that given an FFT over the nth sample of an STFT, the kth and $k+1$th element of stage s in its butterfly is related to the $k/2$-indexed elements of butterflies computed from earlier samples of the STFT, i.e. $n-1, n-2, \ldots$. This is expressed in Eq. (5), where $f_{s,k}$ refers to the kth element at stage s of the interim DFT over the nth sample, and $f_{s,k}$ is a combination of $b_{s,k/2}(m)$ and f_r. Here, f_r is the $k/2$ element from stage $s-1$ of prior FFT butterfly m – computed as the STFT iterates over samples $n = 1, 2, \ldots$. Also, $m = \mod(i, N/2^s) - 1$, and f_r is multiplied by twiddle factor ω. The feedforward STFT maintains a buffer b, where $b_{s,k/2}(m)$ stores $f_{s-1,k/2}$ of the mth prior DFT from previous samples.

$$f_r = f_{s-1,k/2} * \omega \quad (5)$$

$$f_{s,k} = b_{s,k/2}(m) + f_r \qquad f_{s,k+1} = b_{s,k/2}(m) - f_r$$

Relating Eq. (5) to the example in Fig. 2, let $n = 1$ (representing STFT 1), with $s = 1$. Using the feedforward short term Fourier transform, we have $f_{1,0} = f_{(4-9)}$ and $f_{1,1} = f'_{(4-9)}$, as well as $m = \mod(1, 8/2^1) - 1 = 0$, so that $b_{1,0}(0) = x(4)$. Following Eq. (5), we get $f_{1,0} = x(4) + W' f_{0,1} = x(4) + W'x(9)$. Similarly, $f_{1,1} = x(4) - W'x(9)$. This relationship is shown in Fig. 1.

3. Proposed algorithm

3.1. Proposed staggered parallel STFT

Both the Cooley and Tukey [18] and the feedforward algorithms [9] re-use past computations to improve efficiency. We follow these same principles for our proposed algorithm that computes

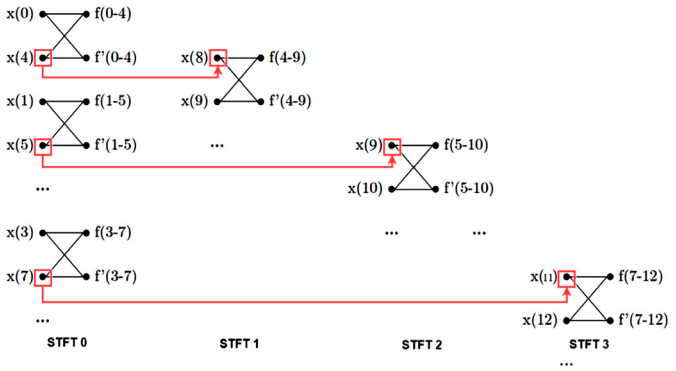


Fig. 1. Node-sets in the first stage $s = 1$ of three (3) sequential STFT's. Following the feedforward STFT algorithm [9], it is observed that computations of sequential STFT's are related. In this case, given a $N = 8$ signal, the 2nd input $x(4)$ of STFT 0 is mapped to the 7th input of STFT 1, while the 4th input $x(5)$ is mapped to the 7th input of STFT 2, while the 7th input $x(7)$ of STFT 0 is mapped to the 8th input of STFT 3. This relationship could be analogously applied to larger node-sets in later stages and forms the basis for Claim 1.

$N/2$ consecutive FFT's in parallel – without performing full recursions (similar to feedforward STFT). This procedure is different from what we define as purely parallel algorithms which perform full recursions for each FFT or from feedforward algorithms (which are sequential FFT's). Our algorithm is a staggered parallel procedure that falls between parallel and sequential schemes.

3.1.1. Preliminaries

To start, we provide the following definitions that will be repeatedly used throughout the paper. The first definition below (Definition 1) is an elementary definition over stages of FFT computations.

Definition 1. Let s denote the stage of an FFT. The total number of stages s_N given a sample complexity of N elements is $s_N = \log N + 1$. Let indices of stages s range from $s = 1..s_N$ (i.e. an index array that begins at 1).

The next definition (Definition 2) presents the concepts of node-sets. The concept of node-sets will be used repeatedly in our algorithm's procedure and in our analysis.

Definition 2. Let F_n denote an FFT computation over sample $\{x[n]..x[n+N-1]\}$. A node-set in F_n is denoted by $\lambda_s^m(n)$. The node-set $\lambda_s^m(n)$ is composed of intervals of neighboring computations f_s^* in stage s of the FFT butterfly. In particular, for the first stage $s = 1$, each input x is a node-set in itself. For stages $s > 1$, a set of FFT computations $\{f_s^*\}$ belongs to node-set $\lambda_s^m(n)$ if they are computed using a common node set $\lambda_{s-1}^m(n)$ from prior stage $s-1$.

Graphically, the node-sets $\lambda_s^m(n)$ are consecutive nodes in the butterfly - related under the addition/subtraction operations of Eq. (2). There are several possible $\lambda_s^m(n)$ for each stage s, and m serves as an individual node-set's index. Examples of node-sets are shown in Fig. 2, where nodes in the butterfly enclosed in a box belong to a single node-set. In what follows, we define the notation $|m_s|$ which represents the number of node-sets in each stage s.

Definition 3. Each stage s has a total of $|m_s|$ node-sets, where $|m_s| = N/2^{s-1}$. The number of elements $\{f_s^*\}$ in each node-set λ_s^m add up to a total of $2|k| = 2^{s-1}$ elements, where $|k| = 2^{s-2}$.

For instance, in Fig. 2's FFT with $N = 8$ elements, the 1st stage ($s = 1$) has 8 node-sets $|m_1| = 8/2^0 = 8$. These node sets are the individual nodes themselves, i.e. $\lambda_1^0 = x[0], \lambda_1^1 = x[1] \ldots \lambda_1^7 = x[7]$. For stage $s = 2$, there are 4 node sets $|m_2| = 8/2^1 = 4$, which are $\lambda_2^0, \lambda_2^1, \lambda_2^2, \lambda_2^3$. These are enclosed in red boxes in the 2nd stage of the butterfly in Fig. 2.

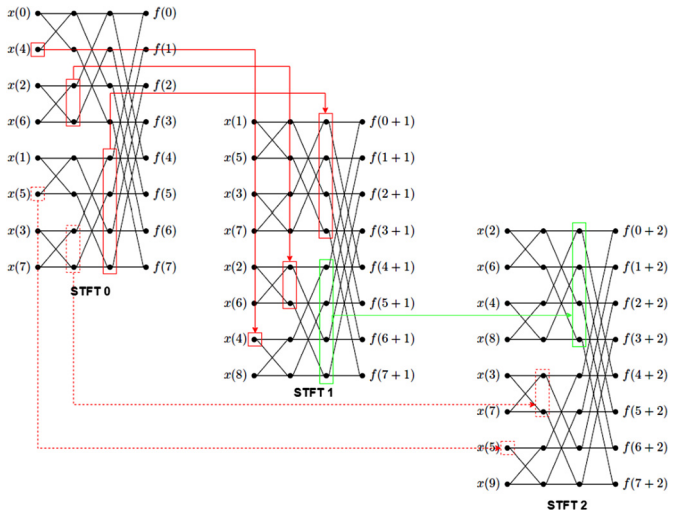


Fig. 2. Graphical illustration of re-using computations given node-sets (butterfly nodes enclosed in boxes) from STFT 0 to STFT 2, following the feedforward algorithm from [9]. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Definition 4. For each stage s of FFT n, we define its terminal node-set $\lambda_s^{|m_s|-1}(n)$ as the node-set whose index m is $\lfloor |m_s| - 1 \rfloor$, i.e. the last odd-indexed node-set in F_n's butterfly.

Using the above definitions, we provide a central claim (Claim 1 below) that will be used to prove correctness properties of our algorithm. Claim 1 generalizes the formulas in [9], by computing FFT functions for any arbitrary n and s. The main equation in Claim 1 is Eq. (6), which is an invariant equation.

Claim 1. *For all FFTs n, we define a special index t as:*

$$t = \mod(n-1, |m_s|) + 1 = \mod(n-1, 2^{s-1}) + 1$$

With this definition, the index $[n-t]$ points to a prior FFT in the STFT (i.e. an FFT with lower index than n). If we set $m = t$, we have the following relationship (Eq. (6)) for terminal node-sets $\lambda_s^{|m_s|-1}(n)$ that obeys structural FFT conditions in Eq. (2).

$$f_{n,s,|m_s|-1,2k} = f_{n-t,s-1,m,k} + \omega f_{n,s-1,2|m_s|-1,k} \tag{6}$$
$$f_{n,s,|m_s|-1,2k+1} = f_{n-t,s-1,m,k} - \omega f_{n,s-1,2|m_s|-1,k}$$

Corollary 1. *The variable t is a decreasing function of s, such that the range of samples needed decrease as stages progress.*

The proof for Claim 1 is in the Appendix A. From Claim 1, we see that to compute a terminal node-set, only portions of prior computations are needed along with some current inputs. We express this in Corollary 2.

Corollary 2. *The needed information to compute the terminal node-set $\lambda_s^{|m_s|-1}(n)$ for all stages of FFT n are:*

1. $\{f_{n-t,s-1}\}$ *from terminal node-set* $\lambda_{s-1}^{|m_s|-1}(n-t)$ *in stage* $s-1$ *of prior FFT indexed at* $[n-t]$
2. $\{f_{n,s-1}\}$ *from node-set* $\lambda_{s-1}^{|m_s|-1}(n)$ *in stage* $s-1$ *of current FFT n*

This means that as long as we maintain a memory consisting of $\lambda_{s-1}^{|m_s|-1}(n-t)$ needed for each stage (i.e. past terminal node-sets), we can eventually compute the single terminal node-set at stage s_N. In particular, the very last terminal node-set found in stage s_N denoted by $\lambda_{s_N}^{|m_{s_N}|-1}(n)$ is equal to the FFT coefficients (Corollary 3).

Corollary 3. *FFT coefficients are contained in the terminal node-set at the last stage s_N, i.e. $\lambda_{s_N}^{|m_{s_N}|-1}$.*

3.1.2. Proposed algorithm principle and outline

Corollaries 2 and 3 form the bases for our proposed algorithm which we refer to as Algorithm 1. Algorithm 1 processes each stage $s = 1..s_N$ of $N/2$ consecutive FFTs, with sample complexity N. These $N/2$ FFTs re-use information from 2D buffer M, and a temporary 2D buffer M'. M and M' are formed from terminal node-set computations $\lambda_s^{|m_s|-1}(n-t)$. For this algorithm to be correct, it has to obey the invariant equation (Eq. (6)). Algorithm 1 is shown below with several technical notations, but the succeeding sub-section provides an example to illustrate the steps of the algorithm with more clarity. All stages s in the FFT computation are indexed from $\left[1..(s_N = \log N/2 + 1)\right]$. These represent the column indices of memory buffers M and M'. The node-sets in each stage s are indexed by m_s and range from $m_s = [0..N/2^{s-1} - 1]$. There are $2|k|$ elements inside each node-set, where $|k| = 2^{s-2}$ and k ranges from $k = \{0..|k| - 1\}$. The row indices of M and M' range from $i = [1..(N/2)]$ to denote the FFT indices. However, indices for the input signal X range from $i = [0..T]$, where T is the total number of samples that are powers of 2. This is to allow a special 0 notation for the first N-sized sample from X.

Staggered Parallel Algorithm 1 Outline

Step 1 Create 2D memory buffers M and M', both of size $N/2 \times (N/2 - 1)$. Each element in M and M' contains a terminal node-set $\lambda_s^{|m_s|-1}$. Columns of M are used to compute stages $s = 1...(s_N - 1)$ of $N/2$ FFTs in the batch. Rows of M represent indices over FFTs $n = 1..(N/2)$. M' is a temporary storage to keep information for the next batch.

Step 2 Initialize M by performing a standard FFT F_0 over the first N-sized sample from X, i.e. $\{x[0]..x[N-1]\}$. For each column $s = 1..(s_N - 1)$ in M, fill up rows $1..N/2^s$ with node-sets from F_0 using this procedure:

- For each stage s of the FFT, a tuple of node-sets $\{\lambda_s^{m_s}\}$ are stored in M. These node-sets are selected on the basis of their index m_s, where m_s is odd.

At the end of the procedure, M contains all odd-indexed node-sets for each stage s of F_0.

Step 3 From FFTs $n = 1..(T - N)$ of the STFT, compute batches of $N/2$ FFTs using the scheme below.

1. Let i refer to the index of the first sample $x \in X$, that is used to compute the batch of FFTs. For the first stage $(s = 1)$ of the FFTs, collect the next $N/2$ samples from X, i.e. $\{x[i+N-1]..x[i+N-1+N/2-2]\}$.
2. Pair each of the newly collected $N/2$ samples with terminal node sets from prior computations. These node sets are found in the 1st column of M. Pairing is done in order, where node-set from row 1 of M is paired with $\{x[i+N-1]$, and node-set from row 2 of M is paired with $\{x[i+N-1+1]$, etc.
3. For stages $s = 2..s_N$, compute the following in parallel for all $N/2$ FFTs in the batch
   (a) Compute $2|k|$ elements of terminal node-set $\lambda_s^{|m_s|-1}(n)$ following Eq. (6). Eq. (6) is as follows, for $k = \{0..|k| - 1\}$:

$$f_{n,s,|m_s|-1,2k} = f_{n-t,s-1,m=t,k} + \omega f_{n,s-1,2|m_s|-1,k}$$
$$f_{n,s,|m_s|-1,2k+1} = f_{n-t,s-1,m=t,k} - \omega f_{n,s-1,2|m_s|-1,k}$$

   where $t =$ mod $(n-1, |m_s|) + 1$ as mentioned before.
   To easily compute Eq. (6) using M, substitute the first terms of Eq. (6) with the terminal node-sets located in column $s-1$ of M. These node sets represent $\lambda_{s-1}^{|m_{s-1}|}(n-t)$. The second terms in Eq. (6) are substituted with prior computations $\lambda_{s-1}^{|m_{s-1}|}(n)$, from stage $s-1$ of the respective FFT n.
   (b) (Update M) Append to column s of M beginning at row $i = N/2^s + 1$, a $[N/2 - N/2^s]$-sized set consisting of terminal node-set computations $\lambda_s^{|m_s|-1}(n^*)$. These node-sets are taken from FFTs indexed by $n^* = [1...[N/2 - N/2^s]$ in the batch, namely:

$$\left\{\lambda_s^{|m_s|-1}(1), \quad \lambda_s^{|m_s|-1}(2) \quad ... \quad \lambda_s^{|m_s|-1}([N/2 - N/2^s])\right\}$$

   (c) (Update M') Set elements in column s of M' from row 1 up to row $i = N/2^s$ the $[N/2^s]$-sized set of terminal node-sets $\lambda_s^{|m_s|}(n^*)$. These node-sets are taken from FFTs indexed by $n^* = [(N/2 - N/2^s + 1)..N/2]$ in the batch, namely:

$$\left\{\lambda_s^{|m_s|-1}(N/2 - N/2^s + 1), \quad \lambda_s^{|m_s|-1}(N/2 - N/2^s + 2) \quad ... \quad \lambda_s^{|m_s|-1}(N/2)\right\}$$

Step 4 Output the terminal node-set $\lambda_{s_N}^{|m_{s_N}|-1}(n)$ of stage $s = s_N$ for all $N/2$ FFTs - representing their coefficients

Step 5 Re-initialize M for the next $N/2$ sized batch by setting $M = M'$. Then go back to step 3.

3.2. Algorithm example

To show a step-by-step breakdown of Algorithm 1, we use a 16-element linear input signal X as an example. Let the sample complexity of each FFT be $N = 8$.

Example Description

1. Signal: $X = \{0+0i, 1+0i, 2+0i, 3+0i, 4+0i, 5+0i \ ... \ 15+0i\}$
2. Length of signal: 16, with sample size (N): 8
3. No. of stages (s_N): $\log N + 1 = \log 8 + 1 = 4$ (from s_1 to s_4)

Step 1 [create buffers M and M']

As shown in Fig. 3, the 1st 2D buffer M is created with size 4×3 (using the formula $N/2 \times (N/2 - 1) = 8/2 \times (8/2 - 1)$ from Step 1). The same is done for M'. The 4 rows of M will be used to compute Fourier transforms $F_1 - F_4$. In this example, F_1 is computed using samples $X[1] - X[8] = \{1+0i...8+0i\}$, F_2 is computed using $X[2] - X[9] = \{2+0i...9+0i\}$, F_3 is computed using $X[3] - X[10] = \{3+0i...10+0i\}$, etc.

Step 2 [initialize M]

M is initialized using a fast Fourier transform (F_0) over the first $N = 8$ elements of X. In this case, F_0 is computed using samples $X[0] - X[8] = \{0+0i...7+0i\}$. This is shown in Fig. 3 where samples $X[0] - X[8]$ are in the first column of F_0. After computing F_0, columns of M are filled up in this way:

Algorithm 1 Proposed Algorithm Pseudocode.

Input: X ▷ input signal
Initialize M, M' ▷ initialize memory 1 and 2
Set $S_N = \log N + 1$ ▷ set no. of stages

procedure INITIALIZE(X)
  FFT(X[0:N]) ▷ perform FFT on 1st N signals
  for $s = 1 : (S_N - 1)$ **do** ▷ set M to initial FFT
    for $k = 0 : (N/2^s)$ **do**
      M[s][k] = FFT[s][2k+1] ▷ odd-indexed FFT

procedure MAIN ROUTINE(input X)
  Call **INITIALIZE** ▷ initialize memory buffer M
  for n = 1 : length(X) $-N$ **do**
    for $s = 1 : S_N$ **do**
      initialize B ▷ create temporary input buffer
      for i = 1 : $N/2$ **do**
        m = M[s-1][i] ▷ get prior memory
        if s == 1 **then** ▷ first stage
          $\lambda_1[i]$ = x[i+n+N-1] ▷ get next signal $x \in X$
          M'[s-1][i] = $\lambda_1[i]$ ▷ update M' with next signal
        B = B U [λ_s[i],m,s] ▷ update input buffer
      F = **PARALLEL(B)** ▷ parallel routine
      idx = $N/2^s$ ▷ set update index
      λ_s = F ▷ update node-set λ_s with F
      M'[s][0:idx] = F[$\left[\left(\frac{N}{2}\right) - \text{idx} + 1\right]$:] ▷ update M'
      M[s][(idx+1):] = F[:$\left[\left(\frac{N}{2}\right) - \text{idx}\right]$] ▷ update M
    Output λ_{s_N} ▷ FFT output
    M = M' ▷ re-initialize M for next batch

procedure PARALLEL(B)
  λ = B[n][0] ▷ input current node-set $\lambda^m_{s-1}(n)$
  m = B[n][1] ▷ input prior node-set $\lambda^m_{s-1}([n-t])$
  s = B[n][2] ▷ input stage integer s
  F_n = [ ] ▷ create output buffer for nth node-set
  for $k = 0 : 2^s - 2$ **do**
    $xr = \lambda \times \omega(s, k, n)$ ▷ ω is twiddle factor
    $x0 = m[k] + xr$
    $x1 = m[k] - xr$
    F^n[k] = x0
    F^n[k+1] = x1
  Re-arrange F_n ▷ Set even elements to 1st half, and odd to 2nd half
  return F_n ▷ FFT output for stage s of nth signal

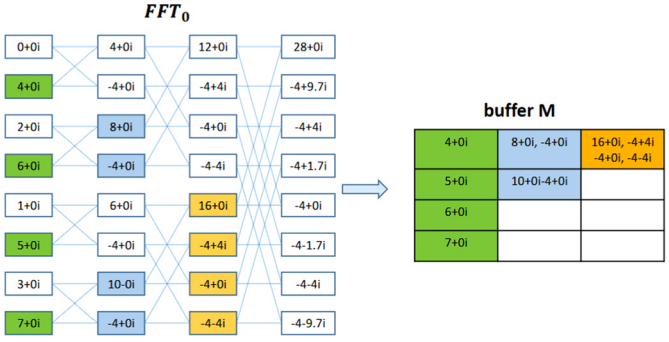


Fig. 3. Steps 1-2.

- For column 1 of M, put odd-indexed node-sets $\lambda^m_1(0)$ of F_0 from stage $s = 1$, where $m = \{1, 3, 5, 7\}$. These node-sets are put in M from the top row down to row 4. This corresponds to the green boxes in Fig. 3.
- For column 2 of M, put odd-indexed node-sets $\lambda^m_2(0)$ of F_0 from stage $s = 2$, where $m = \{1, 3\}$ in rows 1 and 2. Each node-set in this case has 2 elements. This corresponds to the blue boxes in Fig. 3.
- For column 3 of M, put the single odd-indexed node-set $\lambda^1_2(0)$ of F_0 from stage $s = 3$ in row 1. λ^1_2 in this case has four elements, corresponding to the orange boxes in Fig. 3.

Step 3 [stage 2 of FFT, part 1]

For step 3, we compute the terminal node set in stage $s = 2$ for F_1 to F_4. Using the concepts shown previously, we have for stage 2:

1. $|m_s| = N/2^{s-1} = 8/(2^1) = 4$, i.e. there are 4 node-sets for each F
2. the terminal node-set is $\lambda_2^{|m_s|-1} = \lambda_2^3$
3. $|k| = 2^{s-2} = 2^{2-2} = 1$, where k ranges from $k = \{0..1-1\}$, i.e. $k = \{0\}$
4. terminal node-set λ_2^3 has $2|k| = 2$ elements.

Next we compute t, to determine which prior samples are used for computing λ_2^3 given Eq. (6). Using the formula, we have for F_1 and F_2 the following. The same is done to compute t for F_3 and F_4.

1. for F_1, we have $t = \mod(1-1, 4) + 1 = 1$, so that $[n-t] = [1-1] = 0$, i.e. the node set from F_0. This is in row 1 and column $s-1 = 1$ of M.
2. for F_2, we have $t = \mod(2-1, 4) + 1 = 2$, so that $[n-t] = [2-2] = 0$, i.e. the node set from F_0. This is in row 2 and column $s-1 = 1$ of M.

Afterwards, collect the next 4 signals $\{x[9], x[10], x[11], x[12]\}$, which are $\{8+0i, 9+0i, 10+0i, 11+0i\}$. We now compute the elements of λ_2^3 using Eq. (6). As an example, we write the process for the 1st and 2nd elements of $\lambda_2^3(1)$ of F_1 using $t = 1$ and $t = 2$ respectively. The same is done for F_2 to F_4. In these computations, information in M is needed.

1st element of $\lambda_2^3(1)$ in F_1 with $k = 0$

$$f_{n,s,|m_s|-1,2k} = f_{n-t,s-1,t,k} + \omega f_{n,s-1,2|m_s|-1,k}$$
$$\implies f_{1,2,3,0} = f_{1-1,2-1,1,0} + \omega f_{1,2-1,8-1,0}$$
$$= f_{0,1,1,0} + \omega f_{1,1,7,0}$$
$$\implies f_{1,2,3,0} = M[1,1] + \omega X[8]$$
$$(M[1,1] \text{ is 1st row, 1st col of } M)$$
$$\implies 12 + 0i = (4 + 0i) + \omega(8 + 0i)$$

2nd element of $\lambda_2^3(1)$ in F_1 with $k = 0$

$$f_{n,s,|m_s|-1,2k+1} = f_{n-t,s-1,t,k} - \omega f_{n,s-1,2|m_s|-1,k}$$
$$\implies f_{1,2,3,2} = f_{2-2,2-1,1,0} + \omega f_{1,2-1,8-1,0}$$
$$= f_{0,1,1,0} - \omega f_{1,1,7,0}$$
$$\implies f_{1,2,3,0} = M[1,1] - \omega X[8]$$
$$(M[1,1] \text{ is 1st row, 1st col of } M)$$
$$\implies -4 + 0i = (4 + 0i) - \omega(8 + 0i)$$

For more detail, given M, we have $M[1,1] = 4+0i$, $M[2,1] = 5+0i$, $M[3,1] = 6+0i$ and $M[4,1] = 7+0i$. Combining these with the collected samples $\{x[8]..x[11]\}$, we have the following for λ_2^3 of $F_1 - F_4$. These match with Fig. 4, where the last two elements of each FFT (λ_2^3) are filled.

$$F_1: \begin{matrix} M[1,1] + \omega(8+0i) = 12+0i \\ M[1,1] - \omega(8+0i) = -4+0i \end{matrix} \Big\} \text{ node-set } \lambda_2^3(1)$$

$$F_2: \begin{matrix} M[2,1] + \omega(9+0i) = 14+0i \\ M[2,1] - \omega(9+0i) = -4+0i \end{matrix} \Big\} \text{ node-set } \lambda_2^3(2)$$

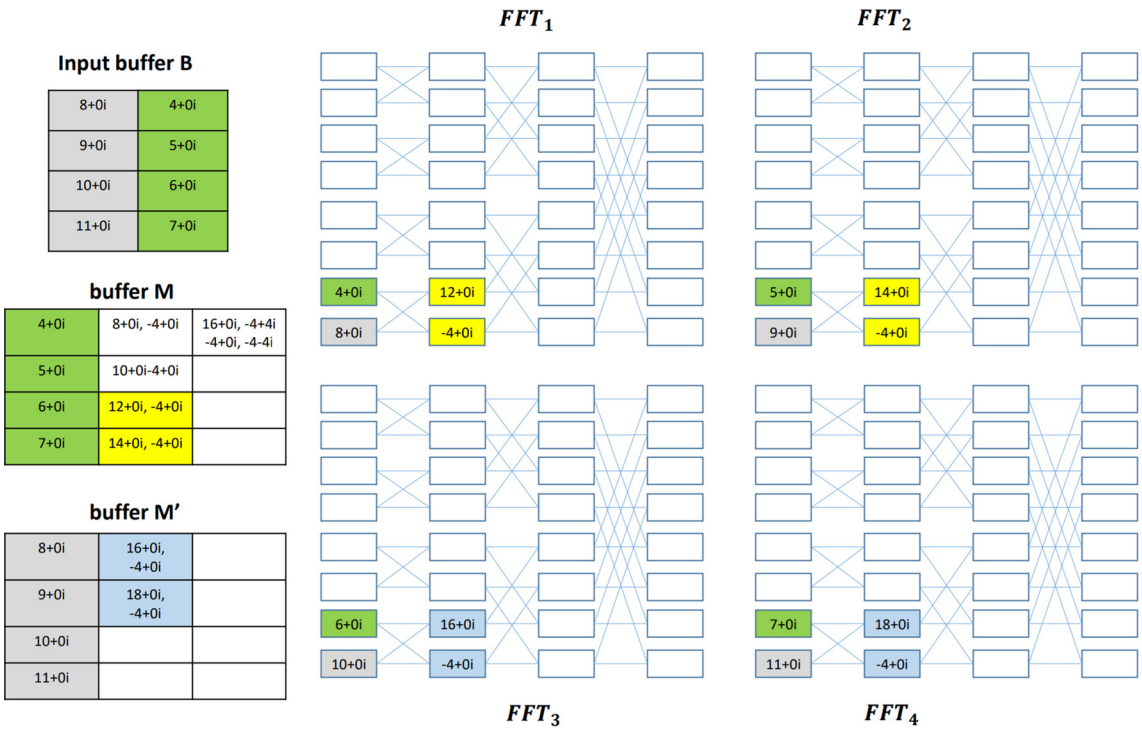


Fig. 4. Step 3, stage 2.

$$F_3: \left.\begin{array}{l} M[3,1]+\omega(10+0i)=16+0i \\ M[3,1]-\omega(10+0i)=-4+0i \end{array}\right\} \text{node-set } \lambda_2^3(3)$$

$$F_4: \left.\begin{array}{l} M[4,1]+\omega(11+0i)=18+0i \\ M[4,1]-\omega(11+0i)=-4+0i \end{array}\right\} \text{node-set } \lambda_2^3(4)$$

Step 3 [stage 2 of FFT, part 2]

After computing terminal node-sets λ_2^3 of $F_1 - F_4$, buffers M and M' are updated with the computed λ_2^3's. We first update M, using the following:

1. Collect node-sets λ_2^3 from FFTs $1..[N/2 - N/2^s] = \{1..[8/2 - 8/2^2]\}$ (referring to $\lambda_2^3(1)$ and $\lambda_2^3(2)$)
2. From row $(N/2^s + 1)$, (i.e. row 3) up to the last row of M, append $\lambda_2^3(1)$ and $\lambda_2^3(2)$. These are $[12+0i$ and $-4+0i]$ for $\lambda_2^3(1)$ and $[14+0i$ and $-4+0i]$ for $\lambda_2^3(2)$.

This process is shown in Fig. 4, where yellow colored boxes – representing $\lambda_2^3(1)$ and $\lambda_2^3(2)$ of F_1 and F_2 respectively are put in 3rd and 4th rows of M.

Step 3 [stage 2 of FFT, part 3]

Afterwards, M' is updated with a terminal node-set λ_2^3 from $F_1 - F_4$. The rows of M' that needs to be updated with terminal node-sets are rows $\{1..N/2^s\} = \{1..8/2^2\}$, i.e. rows 1 and 2. The FFT sources for these terminal node-sets are FFTs $\{(N/2 - N/2^s + 1)..N/2\} = \{4 - 8/2^2 + 1..4\} = \{3, 4\}$. This means that the needed node-sets are $\lambda_2^3(3)$ and $\lambda_2^3(4)$. This is shown in Fig. 4, where the blue colored boxes for $\lambda_2^3(3)$ (containing $16+0i$ and $-4+0i$) and $\lambda_2^3(4)$ (containing $18+0i$ and $-4+0i$) are put in rows 1 and 2 of M'.

Step 3 [stage 3 of FFT]

For stage $s = 3$, the same process is repeated, where we have $|m_s| = 2$, i.e. there are 2 node-sets, and the terminal node-set is λ_3^1 with 4 elements since $2|k| = 4$. We have the following for t:

1. $F_1 : t =$ mod $(1-1, 2)+1 = 1$ so that $[n-t] = 0$
2. $F_2 : t =$ mod $(2-1, 2)+1 = 2$ so that $[n-t] = 0$
3. $F_3 : t =$ mod $(3-1, 2)+1 = 2$ so that $[n-t] = 1$
4. $F_4 : t =$ mod $(4-1, 2)+1 = 2$ so that $[n-t] = 2$

Using the t's computed, it can be verified that the needed f values needed for Eq. (6), i.e. $f_{[n-t],s-1,t,k} = f_{[n-t],2,t,k}$ for $n = \{1..4\}$ and $k = \{0..1\}$, are located in $M[1,2] - M[4,2]$. Following similar steps as shown above, terminal node-sets λ_3^1 for $F_1 - F_4$ are computed. Their values are shown in Fig. 5.

Afterwards, following the same steps, M's rows are updated with the newly-computed λ_3^1. The rows to be updated are from $N/2^3 + 1 = 2$ up to the last row 4. The λ_3^1 to be put in the rows of M are from Fs $1..[4 - 8/2^3]$ or $F_1..F_3$. This is shown in Fig. 5, where F_1 to F_3 append their respective λ_3^1 to rows 2-4 of M. Lastly, M' is updated. The rows to be updated range from 1 to $N/2^s = 1$. Since we get 1, it means that only 1 row is updated (the first row). The value to be put in row 1 is λ_3^1 from F $\{(N/2 - N/2^s + 1)..N/2\} = \{4\}$, i.e. F_4. As shown in the blue colored box of Fig. 5, λ_3^1 from F_4 is put in row 1 of M'.

Step 4

Repeating the same process for stage $s = 4$, we get the values shown in Fig. 6 for λ_4^1 which equal with the FFT coefficients.

Step 5

The last step transfers the values of M to M'. Afterwards, Step 3 is repeated with the next $N/2 = 4$ sized batch of Fs. This continues until the last sample with index $T - N$ is reached.

3.3. Algorithm correctness

The claims below present correctness properties of Algorithm 1. They state that the procedure of Algorithm 1 computes exact coefficients of a batch of $N/2$ FFT's, and agrees with the invariant FFT equation (Eq. (6)). This is a non-trivial task since the algorithm avoids full recursions, but merely re-uses prior computations. Our central correctness claims are as follows, with their proofs in the Annex.

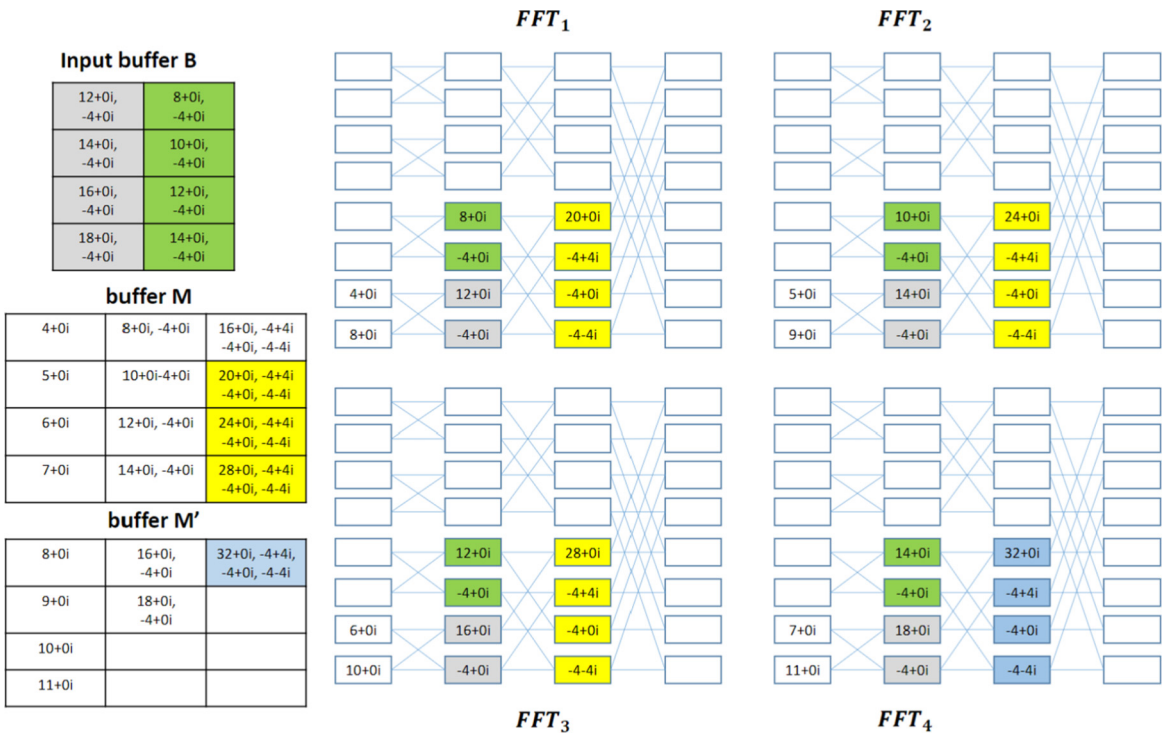


Fig. 5. Step 3, stage 3.

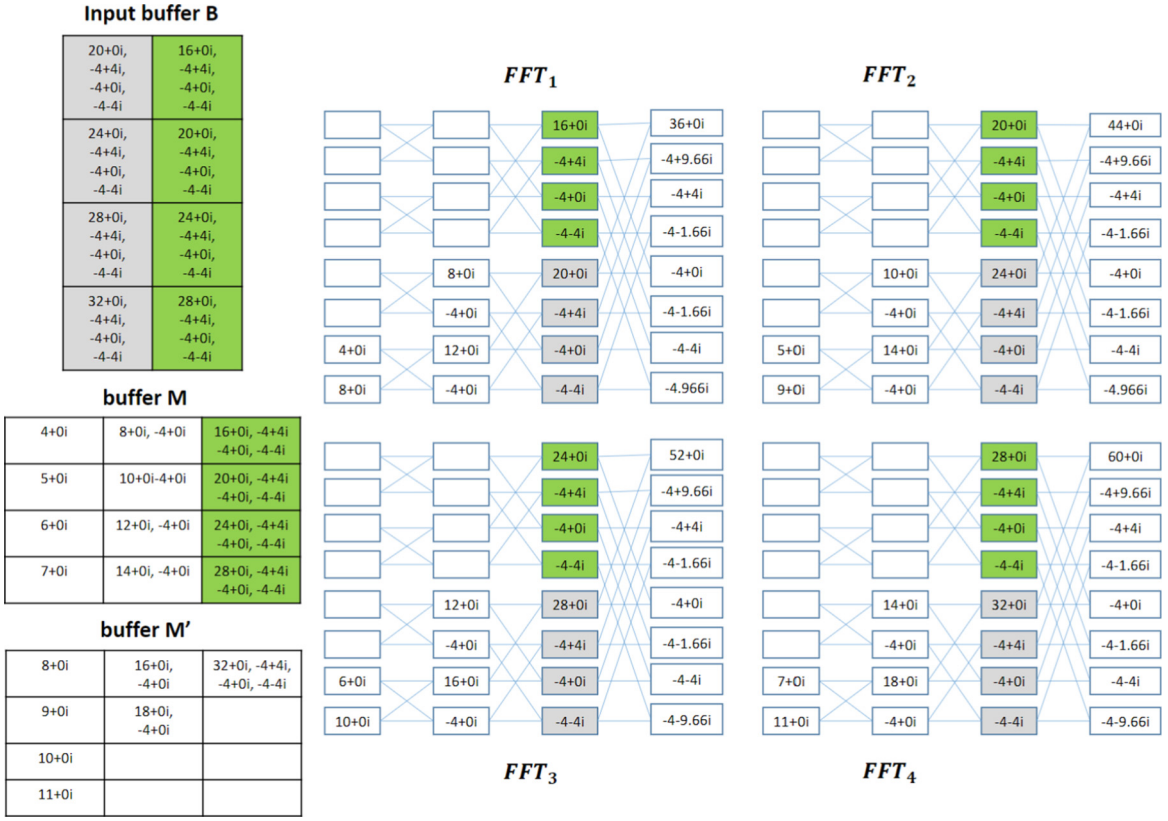


Fig. 6. Step 3, stage 4.

Claim 2. *For each stage s the memory buffer M (of Algorithm 1) contains the needed information* $\lambda_{s-1}^{|m_{s-1}|}(n)$ *to compute all* $2|k|$ *elements in terminal node-set* $\lambda_s^{|m_s|-1}(n)$*, for all* $n = 1..N/2$ *FFTs in the batch.*

Claim 3. *For each FFT in the batch, the procedure in step 3a of Algorithm 1 correctly computes* $\lambda_s^{|m_s|-1}(n)$ *- following the invariant conditions of Eq. (6).*

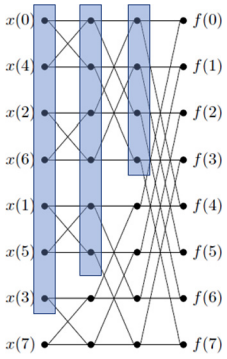


Fig. 7. Redundant computations (shaded in blue) in an FFT that are saved under Algorithm 1.

4. Complexity analysis and experiments

4.1. Time complexity, processor consumption and efficiency of Algorithm 1

In this subsection, we present properties of Algorithm 1. For time complexity, the algorithm incurs $S_N = \log N + 1$ stages given a N-sized sample. For each stage $s \geq 2$, the PARALLEL sub-routine loops from $k = 0..|k|$, resulting in $|k|$ computation pairs for $2|k|$ elements of the terminal node-set. Each computation pair computes 1 addition, 1 subtraction (i.e., 2 adders), and 1 multiplication. Total additions and subtractions for each FFT in a batch are then: $\sum_{s=2}^{\log N+1} 2|k| = \sum_{s=2}^{\log N+1} 2^{s-1} = [2N-2]$. On the other hand, total multiplications are: $\sum_{s=2}^{\log N+1} |k| = \sum_{s=2}^{\log N+1} 2^{s-2} = [N-1]$. Hence we have:

Claim 4. *Given a signal of length T, sample complexity N, and a batch size of $N/2$, each FFT in the batch computes $3N-3$, or $O[N]$ operations in Algorithm 1 (additions and multiplications combined). Since all FFT operations in the batch are done in parallel, this implies that a total of $3N-3$ or $O[N]$ parallel operations are computed in each batch.*

Corollary 4. *Compared to a DFT computation, up to $O[N^2 - N]$ computations are saved in each FFT under Algorithm 1. Compared to an FFT computation, up to $O[N(\log(N) - 1)]$ redundant computations are saved in each FFT under Algorithm 1.*

Proof. A DFT computes $O[N^2]$ operations (additions and multiplications combined) per FFT. We get the result by subtracting Algorithm 1's $O[N]$ from this number. Similarly, an FFT computes $O[N \log N]$ operations per FFT, and subtracting $O[N]$ from this number, we get $O[N(\log N - 1)]$, since Algorithm 1 avoids redundant computations shown in Fig. 7. □

To compute for Algorithm 1's total STFT time complexity, we have Claim 5 below. In this claim, time complexity is measured in terms of the number of parallel addition/multiplication operations computed throughout the STFT.

Claim 5. *Given a signal of length T, sample complexity N, and a batch size of $N/2$, STFT time complexity of Algorithm 1 is $O[6T]$.*

Proof. Given a signal of length T, we add all parallel additions and multiplications for all $(T-N)/(N/2)$ batches. From Claim 4, each batch incurs a total of $3N-3$ parallel additions and multiplications, and thus we have: $\frac{T-N}{N/2} \times (3N-3) = \frac{6(N-1)(T-N)}{N} = O[6T]$ □

Claim 6. *Given a signal of length T, sample complexity N, and a batch size of $N/2$, processor requirements of Algorithm 1 is $O[N/2]$.*

Proof. From Claim 4, $O[N]$ operations are computed in parallel for each batch. Each parallel operation computes coefficients of $N/2$ FFTs, hence, $O[N/2]$ parallel processors are required for each parallel timestep in a batch. □

Efficiency properties:

To compute efficiency, let $O[TN]$ be the speed of the best possible sequential algorithm. This is reasonable since there is no parallelism over N coefficients. Following measures in [11], Algorithm 1 is shown to have high efficiency.

Claim 7. *Let $O[TN]$ be the speed of the best possible sequential STFT algorithm. The speed-up of Algorithm 1 linearly increases with sample complexity N, i.e. with a speed-up of $O[N]$.*

Claim 8. *Let $O[TN]$ be the speed of the best possible sequential STFT algorithm. The efficiency of Algorithm 1 relative to the best possible sequential STFT algorithm given $N/2$ processors is linear at $O[1]$.*

From Claim 5, Algorithm 1's STFT time complexity is a function of batch size $N/2$, being equal to $6T - (6T/N) - 6N + 6$. It follows that for a certain batch size, Algorithm 1's time complexity may reach either a maximum or minimum point, depending on the derivative of the time complexity function according to $N/2$.

Lemma 1. *Given a signal of length T, sample complexity N, and a batch size of $N/2$. Let t_s refer to Algorithm 1's time complexity. The first derivative of t_s according to $N/2$ is as follows:*

$$\frac{\partial t_s}{\partial [N/2]} = \frac{\partial [6(T - (T/N) - N + 1)]}{\partial [N/2]} = \left[\frac{3T}{(N/2)^2} - 12\right]$$

From Lemma 1, we see that t_s is a decreasing function of $N/2$. We also note that $\partial t_s / \partial [N/2]$ is a parabolic function, and attains its maximum at a certain point N'. Using these facts, we derive the following Claims.

Claim 9. *Given a signal of length T, let N' define sample complexity $N' = \sqrt{T}$. For all values of N past N', Algorithm 1's time complexity decreases as a function of batch size $N/2$.*

Proof. Equating $\partial t_s / \partial [N/2]$ to 0, we get: $\frac{\partial t_s}{\partial [N'/2]} = 0 \implies \frac{3T}{(N'/2)^2} - 12 = 0 \implies \frac{4T}{N'^2} = 4 \implies N' = \sqrt{T}$. □

Claim 10. *Let the number of FFT coefficients computed at each timestep be defined as 'computational efficiency'. Given a signal of length T, sample complexity N, and a batch size of $N/2$, the computational efficiency of Algorithm 1 increases as N grows. However, at large values N, the rate of growth in computational efficiency approaches zero.*

Proof. Total number of STFT coefficients amount to $(T-N) \times N$. To get the average number of coefficients computed at each timestep, we divide this number by the total time complexity to get: $\frac{(T-N)N}{6T-(6T/N)-6N+6}$. The derivative of this function with respect to N is $\frac{N(N-2)}{6(N-1)^2}$. Let this function be referred to as f. For a certain value $N'' > 0$, f is increasing. This means that as N grows, the average number of coefficients computed at each timestep grows as well. However, if we take the second derivative with respect to N,

STFT Algorithm Implementations
Sequential Algorithms:
1. feedforward: (exact) feedforward algorithm from [9]
2. iterative: (approximate) iterative algorithm from [12]
3. sequential: (exact) standard divide-and-conquer FFT from [11], [18]
Parallel Algorithms:
4. parallel batch: (exact) $N/2$ parallel standard FFTs similar to [11]
5. parallel batch constant: (exact) constant batch size B of parallel FFTs
6. parallel iterative: (approximate) N parallel algorithm from [12]
7. staggered parallel: (exact) proposed Algorithm 1
8. staggered parallel V2: (exact) Algorithm 1 with parallel memory transfers
9. fully parallel $*$: (exact) theoretical parallel FFT derived from [11]

Fig. 8. Algorithm Implementations.

we get: $\frac{1}{3(N-1)^3}$. Call this function as g. We see that as $N \to \infty$, $g \to 0$. This means that while the average number of coefficients computed at each timestep is growing as a function of N, the rate of growth gradually approaches 0. □

Claim 5 to Claim 10 apply to abstract circuit set-ups where available processors equal those required by the algorithm. But these may not be reflected accurately in CPU implementations given limitations in available cores.

4.2. Exact and approximate algorithm implementations

For analysis, 9 algorithms are surveyed (Fig. 8). These algorithms capture the range of existing STFT implementations indicated in [9], which can be roughly classified as standard STFT, iterative STFT, or feedforward (pruned) STFT, along with their parallel variants. Almost all STFT algorithms are derivatives of these classes, with Algorithm 1 being the first parallel algorithm for the feedforward family - to the best of the authors' knowledge. In Fig. 8, the class of standard STFTs are represented by (3) sequential, (4)-(5) parallel batch and (9) fully parallel. The class of iterative STFTs are represented by (2) iterative and (6) parallel iterative, while the class of feedforward STFTs are represented by (1) feedforward and (7)-(8) staggered parallel (Algorithm 1). The (9) fully parallel algorithm derived from [11] allocates a processor for each of the N coefficients under a $O[N]$ batch size. This differs from (4) and (5), which allocate a single processor for each of batch member but not on a per coefficient basis. (9) fully parallel is analyzed from a theoretical perspective and is not implemented in a CPU due to its exponential requirement in processors.

Algorithm complexity is assessed in terms of time, processor and memory complexity. For time complexity, each addition/multiplication operation is considered a timestep. But to measure timesteps in the context of parallelism, we introduce effective timesteps defined as:

$$\text{Effective timesteps per batch} = \begin{cases} \#\text{ of } (\times/+) \text{ that operate in parallel,} \\ \quad \text{if alg. is parallel} \\ \#\text{ of } (\times/+) \text{ per FFT,} \\ \quad \text{if alg. is sequential (batch size} = 1) \end{cases}$$

4.2.1. Error performance

Among the 9 algorithms, 2 incur errors, which are the iterative methods [12]. The rest compute exact FFTs as shown in Table 1. From experiments, given a 2^{17} sized-signal with values bounded $\in [0, 1]$, accumulative absolute errors of iterative methods may reach ≥ 0.6 with 64-bit floats.

Table 1
Errors incurred by STFT algorithms.

Algorithm	Errors?	Algorithm	Errors?
1. Feedforward	no	5. Parallel batch constant	no
2. Iterative	yes	6. Parallel iterative	yes
3. Sequential	no	7. and 8. Staggered parallel (proposed)	no
4. Parallel batch	no	9. Fully parallel	no

Table 2
Time complexity of STFT algorithms. B in (5) parallel batch constant refers to the constant batch size.

Algorithm	Batch size	Effective timesteps per batch	STFT time complexity
1. Feedforward	1	$3N-3$	$O[3TN]$
2. Iterative	1	$2N+1$	$O[2TN]$
3. Sequential	1	$4N \log N$	$O[4TN \log N]$
4. Parallel batch	$N/2$	$4N \log N$	$O[8T \log N]$
5. Parallel batch constant	B	$4N \log N$	$O[(4TN \log N)/B]$
6. Parallel iterative	1	1	$O[T]$
7. and 8. Staggered parallel (proposed)	$N/2$	$3N-3$	$O[6T]$
9. Fully parallel	N	$\log N$	$O[T]$

4.2.2. Time complexity

For a signal of length T, time complexity is computed as:

$$\text{STFT time complexity} = \frac{T}{\text{batch size}} \times \text{effective timesteps per batch}$$

Algorithm time complexities are shown in Table 2. The fastest algorithms are theoretically (6) and (9). However, (6) incurs errors while (9)'s processor requirements can reach up to $O[N^2]$. On the other hand, proposed (7)-(8) algorithms operate in time $O[6T]$ - independent of sample complexity N. In terms of asymptotic performance, the proposed (7) is asymptotically equivalent to the best known iterative algorithm (6), and also with the best possible exact algorithm (9).

4.2.3. Adjusted time complexity given number of CPU cores

Table 3 shows adjusted time complexity given C cores in a CPU, defined as:

$$\text{Adjusted STFT time complexity} = O\left[\frac{\text{Total \# of } (\times/+) \text{ per batch}}{\text{\# of cores in CPU}} \times \frac{T}{\text{batch size}}\right]$$

Adjusted time complexity does not factor-in thread creation or synchronization costs. Sequential algorithms are not affected since they only consume 1 core. But parallel algorithms' speedup degrades substantially. FFT-based (4) and (9) have significantly higher time complexity – now functions of N and T. In a CPU, (4), (5) and (9) can be potentially slower than (1) and (2). On the other hand, (7)-(8) remains comparable with the fastest algorithm under a CPU, i.e. (6). This implies that the proposed (7)-(8) can be realistically implemented in a CPU.

4.2.4. Processor and memory complexity

For processor complexity, we use the formula from [11], where the number of processors equal the number of $(\times/+)$ operations at each effective timestep. Processor complexities are shown in Table 4. Sequential algorithms have a processor consumption of $O[1]$, while parallel algorithms incur a processor complexity of $O[N]$. (6) has a processor complexity of $O[2N]$ while the proposed (7)-(8) have processor complexities of $O[N/2]$. This is less than (9)'s

Table 3
Adjusted time complexity of STFT algorithms in the context of CPU implementations. C refers to the maximum number of cores available in a computer, where it is assumed that $C \leq$ batch size.

Algorithm	Batch size	Total # of $(\times/+)$ per batch	Adjusted STFT time complexity
1. Feedforward	1	$3N-3$	$O[3TN]$
2. Iterative	1	$2N+1$	$O[2TN]$
3. Sequential	1	$4N\log N$	$O[4TN\log N]$
4. Parallel batch	$N/2$	$(4N^2\log N)/2$	$O[2T(N\log N)/(C)]$
5. Parallel batch constant	B	$(4BN\log N)$	$O[4T(BN\log N)/C]$
6. Parallel iterative	1	$2N+1$	$O[2TN/C]$
7. and 8. Staggered parallel (proposed)	$N/2$	$3N^2/2-3N/2$	$O[3TN/C]$
9. Fully parallel	N	$4N^2\log N$	$O[4T(N\log N)/C]$

Table 4
Processor and Memory complexity of STFT algorithms. In an actual CPU set-up, all parallel algorithms have $O[C]$ processor complexity, where C denotes the number of cores.

Algorithm	Average batch memory	Auxiliary memory	Memory complexity	Processor complexity
1. Feedforward	$O[1]$	$O[N/2]$	$O[N/2]$	$O[1]$
2. Iterative	$O[1]$	none	$O[1]$	$O[1]$
3. Sequential	$O[1]$	none	$O[1]$	$O[1]$
4. Parallel batch	$O[N/2]$	none	$O[N]$	$O[N/2]$
5. Parallel batch constant	$O[B]$	none	$O[B]$	$O[B]$
6. Parallel iterative	$O[1]$	none	$O[1]$	$O[2N]$
7. and 8. Staggered parallel (proposed)	$O[N/2]$	$O[(N/2)^2]$	$O[N^2]$	$O[N/2]$
9. Fully parallel	$O[N^2]$	none	$O[N^2]$	$O[N^2]$

Table 5
Speedup and efficiency of exact parallel algorithms against the best exact sequential algorithm (feedforward).

Exact algorithm	SpeedUp	Efficiency	Adjusted SpeedUp	Adjusted efficiency
4. Parallel batch	$O[N/\log N]$	$O[2/\log N]$	$O[C/\log N]$	$O[1/\log N]$
5. Parallel batch constant	$O[B/\log N]$	$O[B/\log N]$	$O[C/B\log N]$	$O[1/B\log N]$
7. and 8. staggered parallel (proposed)	$O[N/2]$	$O[1]$	$O[C]$	$O[1]$
8. Fully parallel	$O[N]$	$O[1/N]$	$O[C/\log N]$	$O[1/\log N]$

large processor complexity of $O[N^2]$ while remaining asymptotically equal in speed. In an actual CPU set-up, all parallel algorithms have $O[C]$ processor complexity, where C denotes the number of cores. Memory complexity is defined as a function of average batch memory and auxiliary memory, where: Memory Complexity = Average Batch Memory + Auxiliary Memory. Here, average batch memory is the average # of FFT coefficients when computing a batch. Sequential algorithms do not consume much memory. But for (4), average batch memory is $O[N/2]$ since $N/2$ FFTs are computed in parallel while (5) has a constant memory requirement of $[B]$ given non-varying batch sizes. Auxiliary memory applies only for feedforward and staggered parallel. From Table 4, highest memory consumption is the proposed (7)-(8), which may require up to $O[N^2/4]$ memory for storing M and M'.

4.2.5. Speedup and efficiency for exact parallel algorithms

Speedup and efficiency follow standard formulas in [11]. Table 5 shows speedup and efficiency measures for exact parallel algorithms against the best known exact sequential algorithm (1). Theoretically, the proposed (7)-(8) have better speed-up than (4)-(5), with an almost linear efficiency. (9) has the highest speed-up at $O[N]$, but it is also least efficient at $O[1/N]$. Table 5 also shows the adjusted speedup and efficiency under a CPU. From Table 5, the speed-up of algorithms (4), (5) and (9) degrade under C cores. But (7) and (8)'s speed-up remains independent of sample complexity N, and keeps linear efficiency of $O[1]$.

4.3. Experimental results

For experiments, we implement 8 of the 9 algorithms in an Intel® Xeon® E-2176M 6 core processor with vPro™ (2.70 GHz, up to 4.40 GHz with Turbo Boost Technology, 12 MB Cache) with 12 virtual cores and 64 GB RAM (under an Ubuntu 18.04 OS). We did not implement algorithm (9) since its high processor requirements prevents its potential speed in Table 2 from taking place (initial experiments show that (9) is as fast as (4)-(5) in a CPU).

CPU implementations are written in C++. For parallel algorithms (4)-(8), we use the OpenMP parallel library [19]. OpenMP creates several threads that operate concurrently. C++ codes are implemented under the $-O2$ optimization flag with a gcc compiler [20]. If $-O2$ is not included in compilation, time performance for all algorithms degrade proportionally, but their relative time performance rankings remain. CPU and RAM consumption are comparable without the $-O2$ flag. Some algorithms are also implemented in Python, where results are shown in the Annex. Each algorithm (for both C++ and Python) is implemented under a bound of 12, 6, or 3 cores to investigate the effect of number of cores on performance. The incoming input signal is a complex function with length $T = 2^{17} = 131,072$ and sample complexities range from $N = 2^8$ to $N = 2^{15}$. Memory, processor, and time are measured using the *top* performance monitor in Ubuntu 18.04. Performance results are shown in Fig. 9. From these figures, we observe the following (where algorithms (7) staggered parallel and (8) staggered parallel V2 are understood to follow the proposed Algorithm 1).

Time complexity performance analysis

– For any number of cores, the fastest algorithm is (5) parallel iterative following the expected results in Table 3, with a time complexity of $O[2TN/C]$. But this algorithm incurs errors. Comparing (5) with the proposed (7)-(8) in Fig. 9, the gap in speed can be approximated by a constant multiplier. This follows adjusted complexity results in Table 3.

– The fastest exact algorithms are Algorithm 1's (7)-(8) variants. They are faster than the best sequential algorithm, i.e. (1) feedforward, by a factor of around 50%. But the effect of parallel memory transfers in (8) is not significant. This implies that computational time is dominated by addition/multiplication operations.

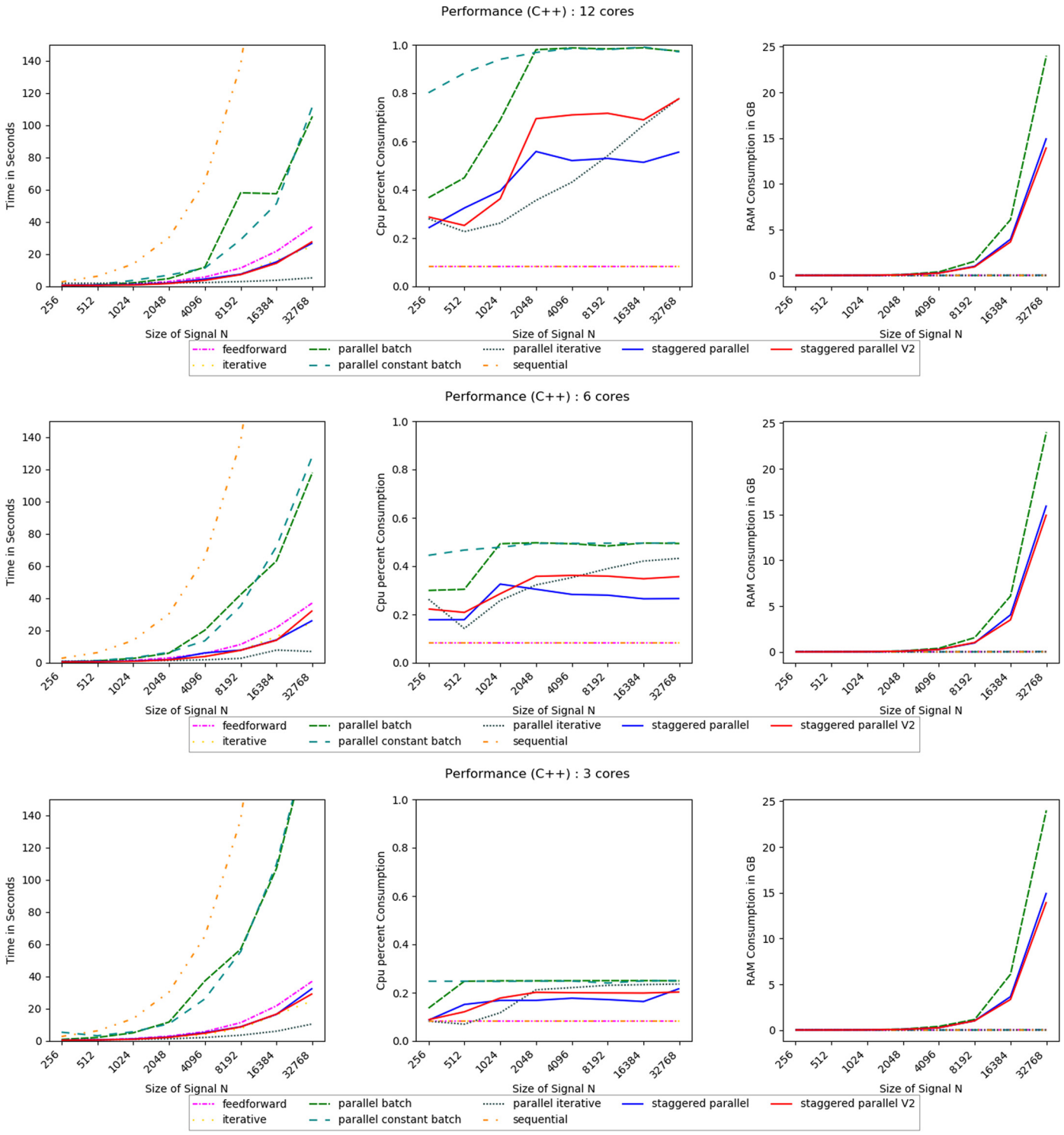


Fig. 9. CPU implementations in C++ under 3, 6 and 12 cores.

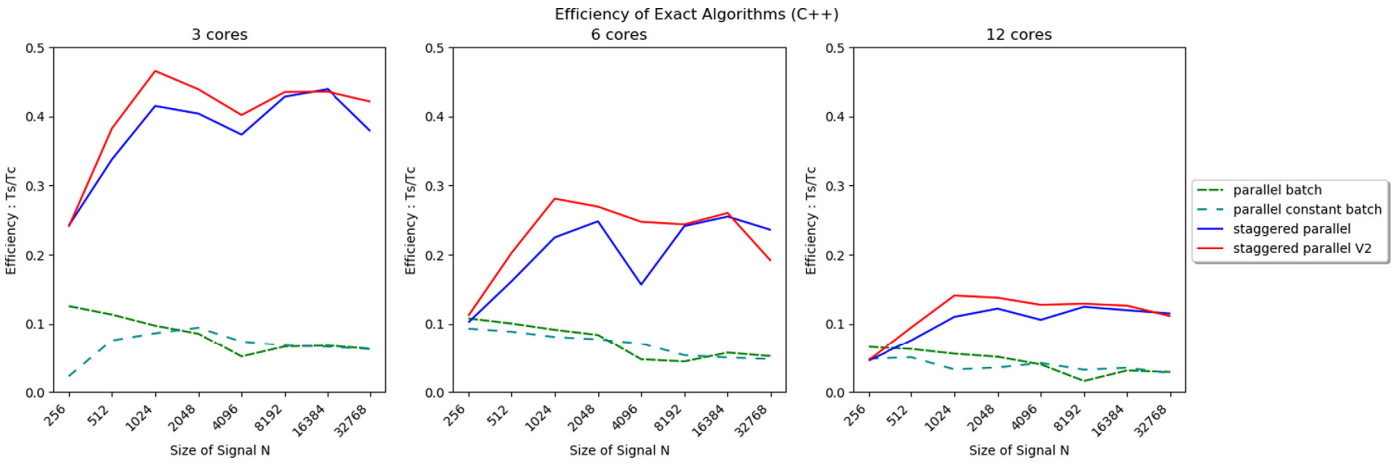
– Starting at a sample complexity of $N = 2048$ to $N = 4096$, the performance of (4) parallel batch degrades significantly. This follows from its adjusted STFT time complexity of $O[T(N \log N)/C]$ in Table 3. If $(TN \log N)$ is much larger compared to C, the effect of parallelism is hampered.

– Comparing the effect of number of cores on time complexity, the proposed staggered parallel algorithms (7) and (8) maintain almost equal speed if cores are lowered from 12 to 3. If cores are lowered from 12 to 3 for (4) and (5), running time (at 3 cores) drastically increases. This follows the results of Table 3, where (7) and (8)'s speed against core availability are not as affected as (4) and (5).

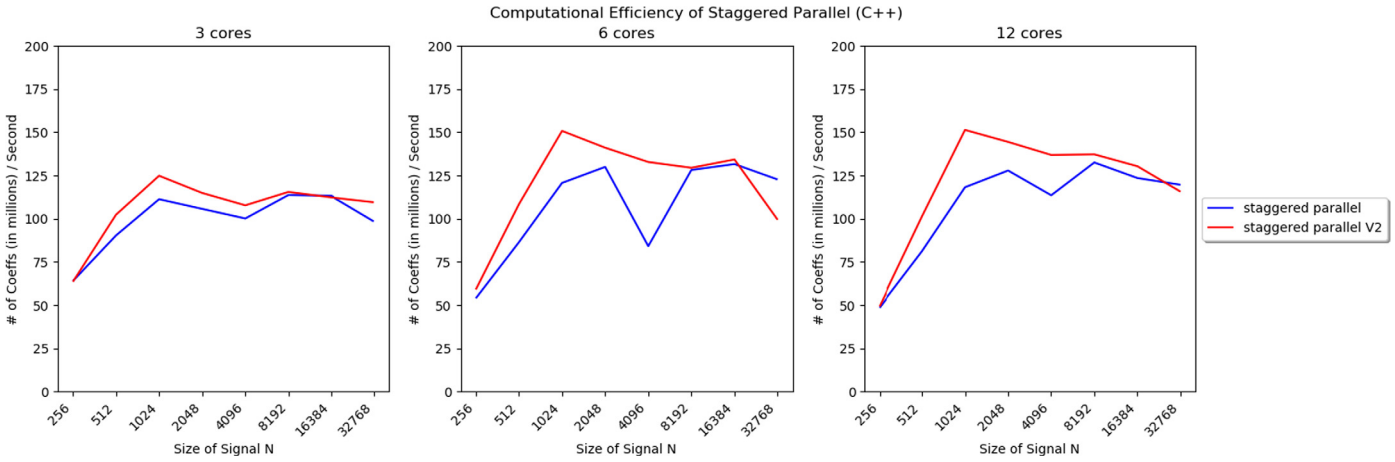
Processor complexity performance analysis

– Parallel algorithms (4)-(8) consume the most number of CPU cores, with large processor consumption at $N = 2048$ to $N = 4096$.

– Among parallel algorithms, (4)'s processor consumption is the largest, following Table 3, where its adjusted time complexity is $O[N^2 \log N]$ - larger than staggered parallel's $O[N^2]$.



(a) Efficiency of exact parallel algorithms in C++



(b) Computational Efficiency of Alg. 1 in C++

Fig. 10. Efficiency performance results.

– Even at a large sample complexity $N = 32768$, the proposed staggered parallel algorithms (7)-(8) do not consume 100% CPU average

Memory complexity performance analysis

– As expected from Table 4, (4) and the proposed (7)-(8) consume the most memory, where memory consumption dramatically increases for algorithms (4), (7) and (8) at sample complexity $N = 8192$ to $N = 16384$.

– (5) has exponentially less memory consumption than (4) due to its constant memory complexity of $O[B]$. However, their CPU consumption remain comparable.

In Fig. 10, we investigate the efficiency performance of exact parallel algorithms (4), (7) and (8). Fig. 10a shows efficiency under standard measures in [11]. We also show in Fig. 10b the computational efficiency performance of the proposed staggered parallel algorithms (7)-(8), where computational efficiency is defined as follows, following Claim 10:

$$\text{Computational Efficiency} = \frac{\text{\# of FFT coefficients computed}}{\text{Time in seconds}}$$

From Fig. 10a, staggered parallel algorithms (7)-(8) are more efficient than batch parallel. This follows the $O[1]$ linear efficiency of (7)-(8) in Table 5, which holds for either an abstract circuit set-up, or in a CPU set-up. From Fig. 10a, (4) and (5) have a decreasing efficiency trend as N grows. On the other hand efficiency for the proposed (7) and (8) have a flatter trend on average. This implies that efficiency of (7) and (8) are more independent over N than (4) and (5). Efficiency in general decreases as the number of cores grow from 3 to 12. This may imply that increasing the number of cores do not directly result in efficiency gains – due to several factors such as hardware configuration, compiler procedures, and increased coordination of threads among the CPU cores. From Fig. 10b, computational efficiency of (7)-(8) increase from $N = 256$ to $N = 2048$ for any number of cores. This follows Claim 9, where at $N' = \sqrt{T} = \sqrt{2^{17}} \sim 362$, Algorithm 1's efficiency increases. Computational efficiency increases drastically from $N = 256$ to $N = 1024$ for any number of cores. But for $N > 1024$, computational efficiency plateaus – following Claim 10. At larger values of N, computational efficiency slightly decreases.

5. Conclusion

In this paper we presented a quasi-parallel algorithm to compute the STFT of a 1D signal. The proposed algorithm provides a quasi-parallel method under the feedforward STFT class of algorithms – where prior computations are re-used to avoid redundant computations. Among the exact algorithms surveyed, the proposed algorithm is among the fastest, with a theoretical time complexity of $O[6T]$. This time complexity is asymptotically equivalent with the fastest approximate STFT algorithm (parallel iterative) – differing by only a constant factor. But whereas parallel iterative incurs errors, the proposed algorithm computes exact STFT coefficients. On the other hand, the algorithm is also asymptotically

equivalent with the fastest exact STFT (of speed $O[T]$) while consuming less processors. Processor consumption is comparable with FFT-based parallel implementations at $O[N/2]$, but with a more efficient measure of $O[1]$ relative to the best known sequential algorithm. The $O[1]$ efficiency holds for either an abstract circuit set-up, or in a CPU set-up with limited number of cores. This efficiency property makes the proposed algorithm practical for CPU implementations. However, memory consumption is relatively higher at $O[N^2]$ – due to the need to maintain an 2D auxiliary memory to store past information. From experimental results, the proposed algorithm is faster than other exact STFT implementations for any number of available CPU cores. Following analysis in processor and memory complexity, the proposed algorithm's CPU consumption is less than some FFT-based parallel implementations, while consuming relatively more RAM than other algorithms.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to acknowledge the Engineering Research and Development for Technology (ERDT) Program, through the Department of Science of Technology (DOST), Philippines for the Ph.D. scholarship grant of Alfonso B. Labao.

Appendix A

Claim 1. *For all FFTs n, we define a special index t as:*

$$t = \mod(n-1, |m_s|) + 1 = \mod(n-1, 2^{s-1}) + 1$$

With this definition, the index $[n-t]$ points to a prior FFT in the STFT (i.e. an FFT with lower index than n). If we set $m = t$, we have the following relationship (Eq. 6) for terminal node-sets $\lambda_s^{|m_s|-1}(n)$ that obeys structural FFT conditions in Eq. (2).

$$\begin{aligned} f_{n,s,|m_s|-1,2k} &= f_{n-t,s-1,m,k} + \omega f_{n,s-1,2|m_s|-1,k} \\ f_{n,s,|m_s|-1,2k+1} &= f_{n-t,s-1,m,k} - \omega f_{n,s-1,2|m_s|-1,k} \end{aligned} \tag{6}$$

Proof. Without loss of generality, let $N = 8$ as in Fig. 2. We assume that the FFT 0 at stage $s = 1$ has already been computed. By induction, we initially set $n = 1$ and $s = 2$, so that the number of node-sets is $|m_2| = 4$ (recall the equation $|m_s| = N/2^{s-1}$, so that $|m_s| = 8/2^{2-1} = 4$). Using the stated Claim, we have $t = m = \mod(0, 4) + 1 = 1$, so that $[n-t] = 1 - 1 = 0$, indicating that the needed prior FFT computations are derived from FFT 0. Using the definitions provided before, we see that $|k| = 2^{2-2} = 1$ so that for node-set $\lambda_2^{|m_2|-1}(1) = \lambda_2^3$, the number of elements is $2|k| = 2$. We then have the following equations for elements $k = \{0, 1\}$ which could be verified to obey the invariant Eq. (2).

$$\begin{aligned} f_{1,2,3,0} &= f_{0,1,1,0} + \omega f_{1,1,7,0} \\ f_{1,2,3,1} &= f_{0,1,1,0} - \omega f_{1,1,7,0} \end{aligned} \tag{7}$$

Still holding the Claim as true, we check if Eq. (6) holds given variations in stage index s and FFT index n. Let us vary s first from $s = 2$ to $s = 3$ while keeping n fixed. The resulting node-set to be examined is $\lambda_3^{|m_3|-1}(1)$, i.e. $n = 1$, $s = 3$, $|m_3| = 2$. It follows that terminal node-set's index is $|m_3| - 1 = 1$. We have $t = m = \mod(0, 2) + 1 = 1$, and $k = 0, 1$, $|k| = 2$, and $2|k| = 4$ so that:

$$\begin{aligned} f_{1,3,1,0} &= f_{0,2,1,0} + \omega f_{1,2,3,0} \qquad f_{1,3,1,2} = f_{0,2,1,1} + \omega f_{1,2,3,1} \\ f_{1,3,1,1} &= f_{0,2,1,0} - \omega f_{1,2,3,0} \qquad f_{1,3,1,3} = f_{0,2,1,1} - \omega f_{1,2,3,1} \end{aligned} \tag{8}$$

Again, it could be verified that the above equations are consistent with the invariant Eq. (2). Similarly, let us vary n from $n = 1$ to $n = 2$ while keeping s fixed for all k in $\lambda_3^{|m_3|-1}(2)$, i.e. $n = 2$, $s = 3$, $|m_3| = 2$. We have $t = m = \mod(2, 2) + 1 = 2$, $|m_3| - 1 = 1$, and $k = 0, 1$ with $2|k| = 4$, so that:

$$\begin{aligned} f_{2,3,1,0} &= f_{0,2,2,0} + \omega f_{1,2,3,0} \qquad f_{2,3,1,2} = f_{0,2,2,1} + \omega f_{1,2,3,1} \\ f_{2,3,1,1} &= f_{0,2,2,0} - \omega f_{1,2,3,0} \qquad f_{2,3,1,3} = f_{0,2,2,1} - \omega f_{1,2,3,1} \end{aligned} \tag{9}$$

Likewise, the above equations are consistent with the invariant Eq. (2). Lastly, we can vary both n and s to $n = 4$ and $s = 4$ for all k in $\lambda_4^{|m_4|-1}(4)$, i.e. we set $n = 2$, $s = 4$, $|m_4| = 1$. We have $t = m = \mod(2, 1) + 1 = 1$, $|m_4| - 1 = 0$ and $k = 0..3$ with $|k| = 4$, and $2|k| = 8$, so that:

$$\begin{aligned} f_{2,4,0,0} &= f_{1,3,1,0} + \omega f_{2,3,1,0} \qquad f_{2,4,0,4} = f_{1,3,1,2} + \omega f_{2,3,1,2} \\ f_{2,4,0,1} &= f_{1,3,1,0} - \omega f_{2,3,1,0} \qquad f_{2,4,0,5} = f_{1,3,1,2} - \omega f_{2,3,1,2} \\ f_{2,4,0,2} &= f_{1,3,1,1} + \omega f_{2,3,1,1} \qquad f_{2,4,0,6} = f_{1,3,1,3} + \omega f_{2,3,1,3} \\ f_{2,4,0,3} &= f_{1,3,1,1} - \omega f_{2,3,1,1} \qquad f_{2,4,0,7} = f_{1,3,1,3} - \omega f_{2,3,1,3} \end{aligned} \tag{10}$$

Hence, the claim's proposition (Eq. (6)) is verified for variations in either n and s, and could be generalized for larger n or s under arbitrary N. □

Lemma 2. *Let t and s be the same as defined in Claim 1, we have the following relationship for any arbitrary $\theta \in \mathbb{N}$:*

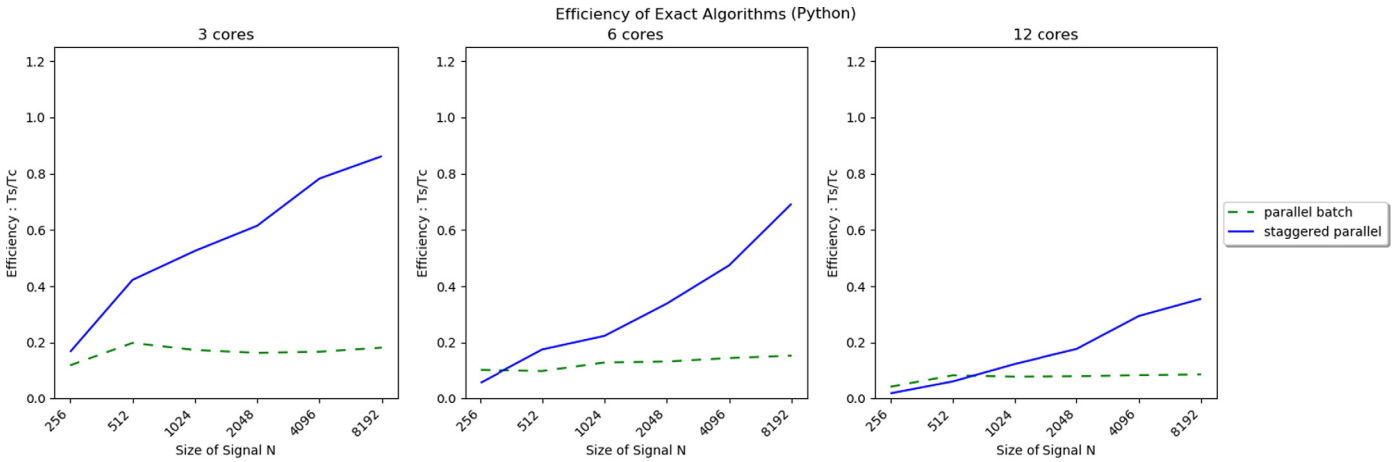
$$\theta - t = \left[\theta - \mod\left(\theta - 1, \frac{\theta}{2^{s-1}}\right) + 1\right] = \left[\theta - \frac{\theta}{2^{s-1}}\right]$$

Claim 2. *For each stage s the memory buffer M (of Algorithm 1) contains the needed information $\lambda_{s-1}^{|m_{s-1}|}(n)$ to compute all $2|k|$ elements in terminal node-set $\lambda_s^{|m_s|-1}(n)$, for all $n = 1..N/2$ FFTs in the batch.*

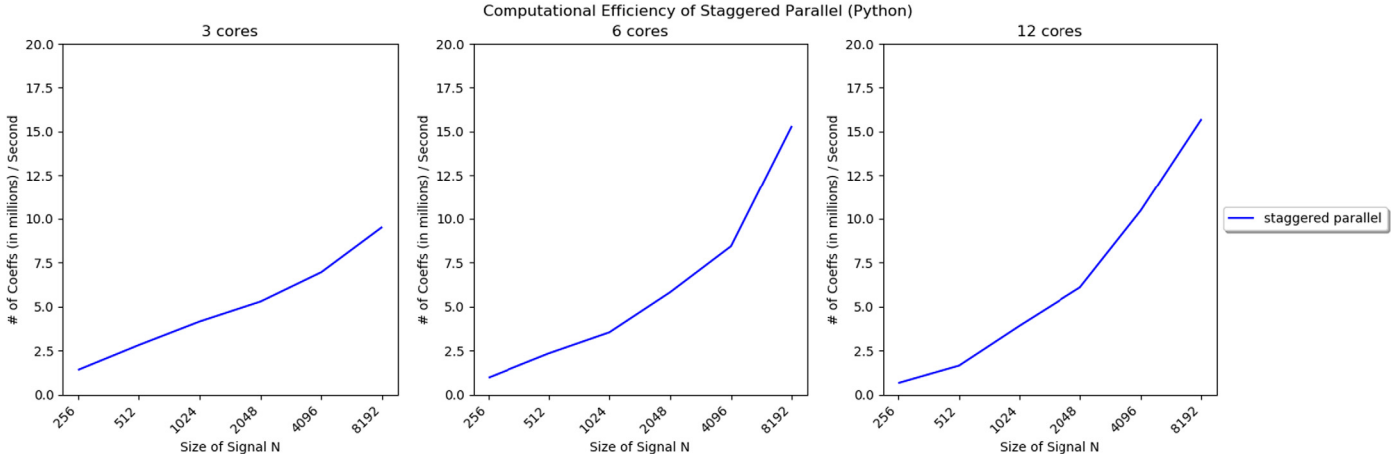
Proof. For each stage s, following Claim 1, we can compute all $2|k|$ elements in $\lambda_s^{|m_s|-1}(n)$ (of FFTs $1...N/2$) using two objects: $\lambda_{s-1}^{|m_s|-1}(n-t)$ and $\lambda_{s-1}^{|m_{s-1}|}(n)$. The latter term $\lambda_{s-1}^{|m_{s-1}|}(n)$ can be derived from computations at stage $s-1$ of the respective FFT and follows trivially from the steps in Algorithm 1. What we need to show is that M contains all the needed $\lambda_{s-1}^{|m_s|-1}(n-t)$ for each stage s of any arbitrary FFT n.

The first stage $s = 1$ does not need computing since it consists of inputs x. For the second stage $s = 2$, we can easily verify that the initialization routine in step 2 of Algorithm 1 provides the correct $\lambda_1^{|m_1|-1}(n-t)$ for all $N/2$ rows of M under column $[s-1] = 2-1 = 1$. To show that all the columns of M contains the needed factors for all other stages $s > 2$, we now examine each FFT n' in the batch under any arbitrary stage $s' > 2$. To compute $\lambda_{s'}^{|m_s|-1}(n')$ for each n' and s', we need a node-set from stage $s'-1$ of FFT $[n'-t]$, where $t = \mod(n'-1, N/2^{s-1}) + 1$ as per Claim 1.

From Corollary 1, t is a decreasing function of s. However, this means that as stages progress, the size of possible values of $[n'-t]$ grows as well. To see this, we can check that for $s = 2$, the prior terminal node-set computation $\lambda_1^{|m_1|-1}(1)$ from FFT 1 is all that is needed to compute the terminal node-set $\lambda_2^{|m_2|}(n')$ for all n'. But for higher stages $s > 2$, the needed node-set can range up to



(a) Efficiency of exact parallel algorithms in Python



(b) Computational Efficiency of Alg. 1 in Python

Fig. 11. Efficiency of exact parallel algorithms and computational efficiency of Algorithm 1 (staggered parallel) in Python.

$[n' - \mod(n', N/2^{s-1}) + 1]$ – an increasing function of s and n'. Hence, to ensure that $\lambda_s^{'|m'_s|}(n')$ can be computed for all s' and n', there has to be a sufficiently 'large' number of prior computed node-sets from FFTs $n^* < N/2$ in M. But for all stages $s' > 2$, the farthest prior node-set needed is for FFT $n' = N/2$ since $[n' - t]$ is an increasing function of n' as stated in the previous paragraph. In this case, we have the following:

$$N/2 - t = \left[N/2 - \mod(N/2 - 1, N/2^{s'-1}) + 1\right]$$

Using Lemma 2, we substitute $\theta = N/2$, and we apply the Lemma to get:

$$\left[N/2 - \mod(N/2 - 1, N/2^{s'-1}) + 1\right] = \frac{N}{2} - \frac{N}{2^{s'}} = N/2 - t$$

Let $\frac{N}{2^{s'}} = \psi$, the key point here is that for $n' = N/2$, we have the relationship $[N/2 - t] = [N/2 - \psi]$, where the 'farthest' node-set from FFT $N/2 - \psi$ should at least be included in column $s' - 1$ of M. Hence, for each stage s', the sufficiently 'large' set should contain $\lambda_{s'-1}^{|m_{s'}|-1}(n^*)$ from FFTs $n^* = 1..[N/2 - \psi = N/2 - N/2^{s'}]$. But we can see that at step 3b of Algorithm 1, each column s of M is appended with computations $\lambda_s^{|m_s|-1}(n^*)$ derived from FFTs $n^* = 1...[N/2 - N/2^s]$. This occurs during the update sub-routine after a parallel computation at stage s. With $s' = s$, under column s of M, there are FFTs $n^* = 1...[N/2 - N/2^{s'}]$. This thereby justifies the claim for all n in a sample batch, i.e. M contains all the needed node-sets. This claim can be generalized for any arbitrary $N/2$-sized batch. In particular, it can be shown that step 3c sets M' to be equal to the initialized state of M right after step 2 for the next batch in the pipeline. In particular, 6 guarantees that M is set to M' to initiate the parallel loop in the next batch. □

Claim 3. *For each FFT in the batch, the procedure in step 3a of Algorithm 1 correctly computes $\lambda_s^{|m_s|-1}(n)$ - following the invariant conditions of Eq. (6).*

Proof. We can prove this by induction. Initially, for stage $s = 2$, all $N/2$ FFTs (arranged according to rows with index $i = n = 1..N/2$) have the correct prior computation $\lambda_1^{|m_1|-1}$ under column $s = 1$. This is due to step [2] in Algorithm 1 for the first batch, and 6 for succeeding batches.

For $s > 2$, suppose that the hypothesis is true. This means that for each FFT n, $\lambda_s^{|m_s|-1}(n)$ is computed by substituting the first terms in Eq. (with the array element in row i and column $s - 1$ of M (representing $\lambda_{s-1}^{|m_s|-1}(n - t)$), while the second terms in Eq. (6) are substituted with saved computations $\lambda_{s-1}^{|m_s|-1}(n)$ from stage $s - 1$ of the respective FFT n. Assume that the second terms of Eq. (6) have been correctly substituted with $\lambda_{s-1}^{|m_s|-1}(n)$ from stage $s - 1$ of the respective FFT n. What is left to show is that the array element in row i and column $s - 1$ of M is indeed equal to the desired $\lambda_{s-1}^{|m_s|-1}(n - t)$ needed to correctly compute Eq. (6).

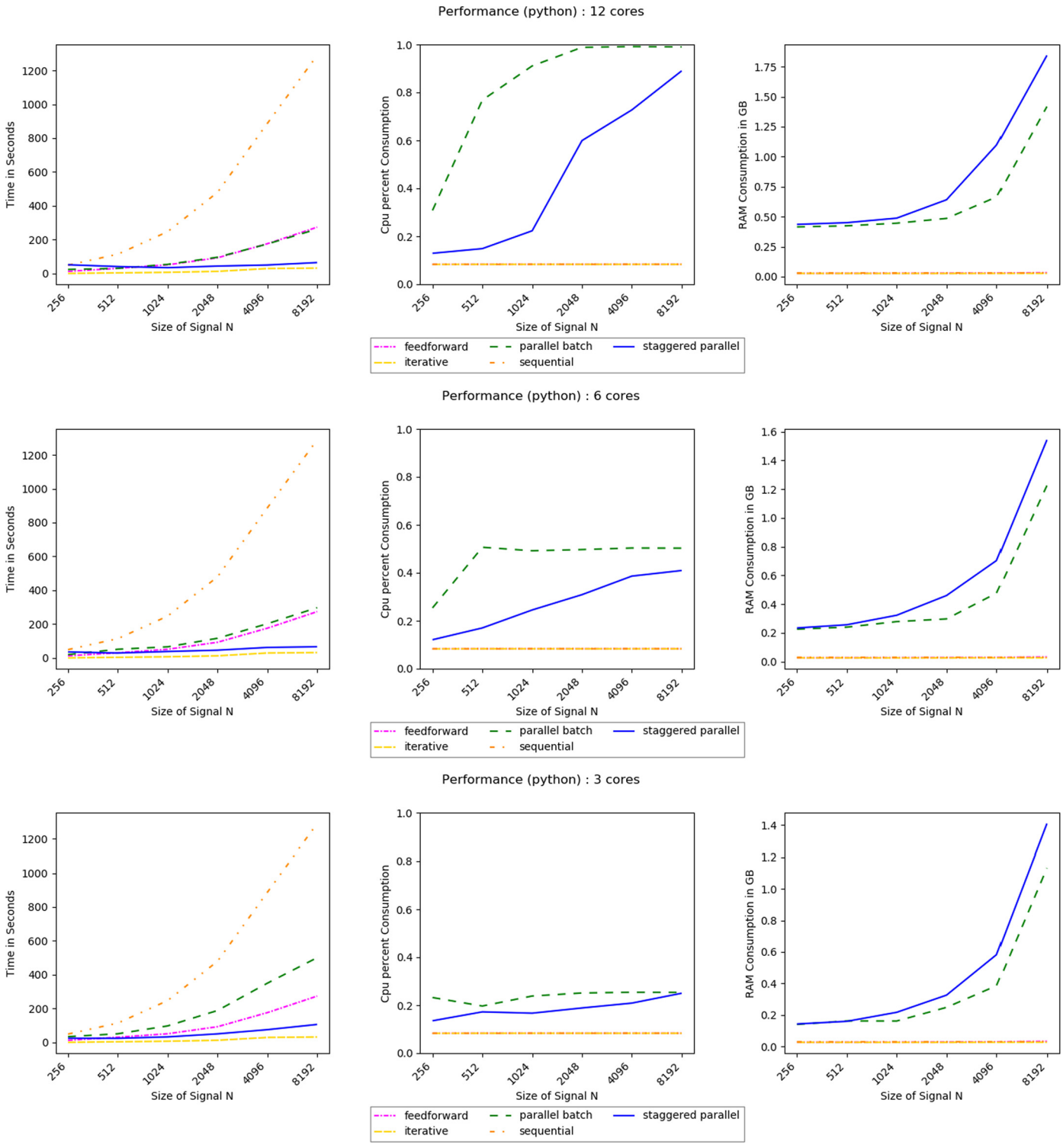


Fig. 12. CPU Implementations in Python under 3, 6, and 12 cores.

From Claim 1, for each n, we need the prior FFT indexed by $[n-t] = [n \mod (n-1, N/2^{s-1}) + 1]$ to compute $\lambda_s^{|m_s|-1}(n)$. Using the induction process, we hold Claim 3 as true, and iterate through stages $s = 3..s_N$. We can see that for stage $s = 3$, following step 3a of Algorithm 1, we use the $j = s-1 = 2$ column to retrieve M's 'version' of $\lambda_2^{|m_2|-1}(n-t)$ for each FFT n. Let us call this version of M as $\hat{\lambda}_2^{|m_2|-1}(i)$, where i is an index over row i of column $s-1 = 2$.

For elements of column $s = 2$, the rows are constructed using steps [2] and step 3b of Algorithm 1. From steps [2] and 3b, we see that for rows $i = 1..N/2^s$ (i.e. rows 1 and 2 if $N = 8$) we have $\hat{\lambda}_2^{|m_2|-1}(0)$ (i.e. the node-sets from FFT F_0). But for rows $i = [N/2^s + 1]..[N/2]$ (i.e. rows 3 and 4 if $N = 8$), we have $\hat{\lambda}_2^{|m_2|-1}(n^*)$, with $n^* = \{1, 2\}$, i.e. the node-sets from FFT 1 and 2. Hence, we can see that the node-sets $\hat{\lambda}_2$ in column 2 for all rows of M are indeed equal to node-sets from FFT 0-2, i.e. in the order of. $\hat{\lambda}_1^{|m_1|-1}(0)$, $\hat{\lambda}_1^{|m_1|-1}(1)$, and $\hat{\lambda}_1^{|m_1|-1}(2)$. This follows from the trivial nature of computing stage $s = 2$ as mentioned previously. Thus the Claim is verified for stage $s = 2$ and stage $s = 3$ can be correctly

computed by assigning each node-set from FFT n to row $i = n$ of column 2.

Starting at stage $s = 3$, we induct the same pattern produced by step 3b for stages $s > 2$. For rows $i = 1..N/2^s$, we have $\hat{\lambda}_2^{|m_2|-1}(0)$. But for rows $i = [N/2^s+1]..[N/2]$, we have $\hat{\lambda}_2^{|m_2|-1}(n^*)$ respectively from FFTs $n^* = 1..[N/2 - N/2^s]$ - in that order. We can generalize this with the following relationship over the rows of Algorithm 1's M under column s:

Given stage s, row i and $\alpha_s = N/2^s + 1$:

$$
\begin{aligned}
M[s,i] &= \hat{\lambda}_s^{|m_s|}(0) && \text{for all } i < \alpha_s \\
M[s,\alpha_s] &= \hat{\lambda}_s^{|m_s|}(1) && \text{for } i = \alpha_s \\
M[s,\alpha_s+1] &= \hat{\lambda}_s^{|m_s|}(2) && \text{for } i = \alpha_s + 1 \\
M[s,\alpha_s+2] &= \hat{\lambda}_s^{|m_s|}(3) && \text{for } i = \alpha_s + 2 \\
&\cdots\cdots \\
M[s,\alpha_s+l] &= \hat{\lambda}_s^{|m_s|}(l+1) && \text{for } i = \alpha_s + l
\end{aligned}
$$

To prove that Algorithm 1 follows Eq. (6), we need to show that Eq. (6) can be reduced to a computation that uses the relationships $M[s-1,n] = \hat{\lambda}_{s-1}^{|m_s|-1}(0)$ for all $n < \alpha_{s-1}$ and that $M[s-1,\alpha_{s-1}+l] = \hat{\lambda}_{s-1}^{|m_s|-1|}(n-t)$ for all $n \geq \alpha_{s-1}$.

From Eq. (6), we see that the first term is element $f_{n-t,s-1,m=t,k}$ from $\hat{\lambda}_{s-1}^m([n-t])$. Since stage $s = 2$ holds trivially, we induct the pattern of Eq. (6) for stages $s \geq 3$. For stage $s = 3$, we have $[n'-t] = 0$ for all FFTs $n' < N/2^2$, and we only need FFT elements $f_{0,2,m=t,k}$ contained in $\hat{\lambda}_2^{m=t}(0)$. But for FFTs $n' \in [N/2^2 < n' \leq N/2]$, we have $[n'-t]$ ranging from 1 to 2 and $f_{1,2,|m_2|,k}$ and $f_{2,2,|m_2|,k}$ are both needed.

Similarly, for stage $s = 4$, we observe that under Eq. (6) we have $[n'-t] = 0$ for all FFTs $n' \leq N/2^{s-1=3}$. But for FFT $n' = N/2^3 + 1$, we have $[n'-t] = 1$, followed by FFT $n' = N/2^3 + 2 = 3$ where $[n'-t] = 2$. Thus, for stages $s > 4$, we can verify the same pattern and generalize that given stage s, we have the following:

$$
\begin{aligned}
[n'-t] &= 0 && \forall n' \leq N/2^{s-1} \\
[n'-t] &= n' - N/2^{s-1} + 1 && \forall n' \in [N/2^{s-1} < n' \leq N/2]
\end{aligned}
$$

But if we substitute $N/2^{s-1}$ with α_{s-1}, we see a correspondence between FFT n and its needed $\hat{\lambda}_{s-1}^m([n-t])$ from row $n = i$ under column $s-1$ of M. Let f_n be shorthand for elements $f_{n,s,m,k}$ derived from $\lambda_s^m(n)$, we have the following to finally prove the Claim:

For all $i = n' < \alpha_{s-1}$:

$$
\begin{aligned}
[n'-t] = 0 &\rightarrow f_0 \in \lambda_{s-1}^m(0) \\
&\rightarrow \hat{\lambda}_{s-1}^m(0) \\
&\in M[s,i]
\end{aligned}
$$

For all $i = n' \in [\alpha_{s-1} < n' \leq N/2]$:

$$
\begin{aligned}
[n'-t] &= n' - N/2^{s-1} + 1 \\
&\rightarrow f_0 \in \lambda_{s-1}^m(n' - N/2^{s-1} + 1) \quad f_0 \text{ from Eq. (6)} \\
&= \hat{\lambda}_{s-1}^m(n' - N/2^{s-1} + 1) \quad \hat{\lambda} \text{ are the node-sets in M} \\
&= \hat{\lambda}_{s-1}^m(l+1) \\
&\in M[s-1,\alpha_{s-1}+l] \\
&= M[s-1,i=n'] \quad \text{column } s-1 \text{ of M under row } i
\end{aligned}
$$

Hence, the element in row i of M under column $s-1$ (i.e. $M[s-1,i=n']$) equals the needed $\lambda_{s-1}^m([n-t])$ to solve the node-set $\lambda_s^m(n)$ for all FFTs n at stage s. □

In Figs. 11 and 12, we present performance results under Python. Parallel implementations here use the joblib package [21] that create separate processes to mimic parallelism in threads. This is needed since Python has a GIL that prevents thread-based parallelism. But from the figures below, it can be observed that the time, processor and memory complexity of the different algorithms are similar to their C++ implementations. Parallel iterative and staggered parallel V2 are not implemented since their parallel routines are largely affected by process initialization in joblib, resulting in inaccurate performance vis-a-vis their complexities under an abstract circuit.

References

[1] D. Griffin, J. Lim, Signal estimation from modified short-time Fourier transform, IEEE Trans. Acoust. Speech Signal Process. 32 (2) (1984) 236–243.

[2] X. Zhu, G.T. Beauregard, L.L. Wyse, Real-time signal estimation from modified short-time Fourier transform magnitude spectra, IEEE Trans. Audio Speech Lang. Process. 15 (5) (2007) 1645–1653.

[3] R. Tao, Y.-L. Li, Y. Wang, Short-time fractional Fourier transform and its applications, IEEE Trans. Signal Process. 58 (5) (2010) 2568–2580.

[4] B.G. Ferguson, B.G. Quinn, Application of the short-time Fourier transform and the Wigner–Ville distribution to the acoustic localization of aircraft, J. Acoust. Soc. Am. 96 (2) (1994) 821–827.

[5] J. Allen, Applications of the short time Fourier transform to speech processing and spectral analysis, in: Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'82, vol. 7, IEEE, 1982, pp. 1012–1015.

[6] H. Erdogan, J. Hershey, S. Watanabe, J. Le Roux, Method for enhancing audio signal using phase information, US Patent 9,881,631, Jan. 30 2018.

[7] P.S. Wright, Short-time Fourier transforms and Wigner-Ville distributions applied to the calibration of power frequency harmonic analyzers, IEEE Trans. Instrum. Meas. 48 (2) (1999) 475–478.

[8] K. Satpathi, Y.M. Yeap, A. Ukil, N. Geddada, Short-time Fourier transform based transient analysis of VSC interfaced point-to-point dc system, IEEE Trans. Ind. Electron. 65 (5) (2018) 4080–4091.

[9] M. Garrido, The feedforward short-time Fourier transform, IEEE Trans. Circuits Syst. II, Express Briefs 63 (9) (2016) 868–872.

[10] S. Zhang, D. Yu, S. Sheng, A discrete stft processor for real-time spectrum analysis, in: IEEE Asia Pacific Conference on Circuits and Systems, 2006, APCCAS 2006, IEEE, 2006, pp. 1943–1946.

[11] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes, Elsevier, 2014.

[12] K.R. Liu, Novel parallel architectures for short-time Fourier transform, IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process. 40 (12) (1993) 786–790.

[13] D. Petranovic, S. Stankovic, L. Stankovic, Special purpose hardware for time frequency analysis, Electron. Lett. 33 (6) (1997) 464–466.

[14] S. Tomazic, On short-time Fourier transform with single-sided exponential window, Signal Process. 55 (2) (1996) 141–148.

[15] W. Chen, N. Kehtarnavaz, T. Spencer, An efficient recursive algorithm for time-varying Fourier transform, IEEE Trans. Signal Process. 41 (7) (1993) 2488–2490.

[16] M. Covell, J. Richardson, A new, efficient structure for the short-time Fourier transform, with an application in code-division sonar imaging, in: [Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing, IEEE, 1991, pp. 2041–2044.

[17] M.A. De Jesús, Y. Rodríguez, M. Teixeira, L. Vicente, Non-Uniform Discrete Short-time Fourier Transform a Goertzel Filter Bank Approach, CRC, Mayaguez, PR, 2004.

[18] J.W. Cooley, How the FFT gained acceptance, in: Proceedings of the ACM Conference on History of Scientific and Numeric Computation, ACM, 1987, pp. 97–100.

[19] L. Dagum, R. Menon, OpenMP: an industry-standard API for shared-memory programming, Comput. Sci. Eng. 1 (1998) 46–55.

[20] R.M. Stallman, GGC Developer Community, Using the GNU Compiler Collection: A GNU Manual for GCC Version 4.3, vol. 3, CreateSpace, Paramount, CA, 2009.

[21] G. Varoquaux, JobLib: lightweight pipelining: using Python functions as pipeline jobs, [Online; accessed <today>], https://pypi.org/project/joblib/.

Alfonso B. Labao is a researcher under the Algorithms and Complexity laboratory of Computer Science, University of the Philippines. He does research on theoretical computer science topics such as algorithmics and parallelism.

Rodolfo C. Camaclang III is currently a graduate student under the Computer Vision and Machine Intelligence laboratory of the University of the Philippines, Computer Science Department. His research interests include Machine Learning and Natural Language Processing.

Jaime Caro received the PhD Mathematics degree from the University of the Philippines, in 1996. He is Professor of Computer Science at the University of the Philippines Diliman and heads its Service Science and Software Engineering Laboratory. He was Assistant Vice President for Development of the University of the Philippines for 14 years and was past President of the Computing Society of the Philippines (CSP). His research interests include Information Systems, Combinatorial Optimization, Technology in Education, and Health Informatics.