

# Formal Derivation of Sequential and Parallel Frequency-domain Beamforming Algorithms Implemented with MPI and POSIX threads

ARL-TL-EV-03-18

Field G. Van Zee\*<sup>†</sup>

May 2003

\*Applied Research Laboratories, The University of Texas at Austin, Austin, TX, 78758, [field@arlut.utexas.edu](mailto:field@arlut.utexas.edu)

<sup>†</sup>Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 78712, [field@cs.utexas.edu](mailto:field@cs.utexas.edu)

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Algorithm Theory</b>	<b>2</b>
2.1	Input/Output Specification . . . . .	2
2.2	Beamforming . . . . .	2
2.2.1	CBF . . . . .	3
2.3	Adaptive Beamforming . . . . .	4
2.3.1	MVDR . . . . .	5
2.3.2	DMR . . . . .	5
<b>3</b>	<b>Algorithm Derivation</b>	<b>7</b>
3.1	Notation . . . . .	7
3.2	Correctness of loops . . . . .	8
3.3	Sequential CBF . . . . .	8
3.3.1	Derivation of inner loop over Fourier frequency bin to beamform . . . . .	9
3.3.2	Derivation of outer loop over time sequence . . . . .	13
3.3.3	Derivation of loop over center frequency bin to compute replica vectors . . . . .	16
3.3.4	Final sequential algorithm for CBF . . . . .	20
3.4	Sequential DMR . . . . .	21
3.4.1	Derivation of inner loop over Fourier frequency bin to beamform . . . . .	21
3.4.2	Derivation of inner loop over center frequency bin to compute DMR weights . . . . .	25
3.4.3	Derivation of outer loop over time sequence . . . . .	31
3.4.4	Derivation of loop over center frequency bin to compute replica vectors . . . . .	34
3.4.5	Final sequential algorithm for DMR . . . . .	34
3.5	Data Partitioning for Parallelization . . . . .	35
3.6	Parallel CBF and DMR . . . . .	38
3.6.1	Changes to loop over center frequency bin to compute replica vectors . . . . .	38
3.6.2	Changes to outer loop over time sequence . . . . .	38
3.6.3	Changes to inner loop over Fourier frequency bin to beamform . . . . .	39
3.6.4	Final parallel algorithm for CBF . . . . .	40
3.6.5	Final parallel algorithm for DMR . . . . .	41
<b>4</b>	<b>Algorithm Analysis</b>	<b>43</b>
4.1	Expected Sequential Performance . . . . .	43
4.2	Expected Parallel Performance . . . . .	45
4.2.1	Speedup . . . . .	48
4.2.2	Efficiency . . . . .	50
4.2.3	Isoefficiency . . . . .	51

<b>5</b>	<b>Optimizations</b>	<b>52</b>
5.1	Algorithm-level . . . . .	52
5.1.1	Load-balancing . . . . .	52
5.1.2	Implementations of complex matrix-matrix multiply . . . . .	56
5.1.3	An alternate method of obtaining CSM eigenvalues and eigenvectors . . . . .	57
5.1.4	Implementations of the complex singular value decomposition . . . . .	59
5.1.5	DMR weight computation and application . . . . .	59
5.1.6	Data distribution block size . . . . .	60
5.2	Source-level . . . . .	60
5.2.1	Accumulator variables . . . . .	60
5.2.2	Allocating multidimensional arrays . . . . .	61
5.2.3	Transforming array indexing into pointer arithmetic . . . . .	62
5.3	Compiler-level . . . . .	63
<b>6</b>	<b>Performance Analysis</b>	<b>64</b>
6.1	Experiment details . . . . .	64
6.2	Results . . . . .	65
6.3	Analysis . . . . .	65
<b>7</b>	<b>Conclusions</b>	<b>72</b>

# List of Figures

3.1	Layout for body of CBF loop over Fourier frequency bin . . . . .	11
3.2	Layout for body of CBF loop over time sequence . . . . .	15
3.3	Layout for body of replica loop over center frequency bin . . . . .	19
3.4	Layout for body of DMR loop over Fourier frequency bin . . . . .	24
3.5	Layout for body of DMR loop over center frequency bin . . . . .	27
3.6	Layout for body of DMR loop over time sequence . . . . .	32
5.1	Original layout of data distribution and collection . . . . .	56
5.2	Optimized layout of data distribution and collection . . . . .	57
5.3	Average <code>cgemm</code> runtime (in seconds) for three BLAS implementations . . . . .	58
5.4	Average <code>cgesvd</code> runtime (in seconds) for two LAPACK implementations . . . . .	59
6.1	Beamformer parameter combinations to be used in performance tests . . . . .	64
6.2	Parallel beamformer runtime results (in wall clock seconds) for naïve partitioning . . . . .	65
6.3	Sampled values for $\kappa$ , $\delta$ and $\gamma$ . . . . .	66
6.4	Parallel beamformer runtime results (in wall clock seconds) for load-balanced partitioning . . . . .	66
6.5	Parallel speedup and efficiency for CBF and DMR implemented with MPI with naïve data partitioning. . . . .	67
6.6	Parallel speedup and efficiency for CBF and DMR implemented with pthreads with naïve data partitioning. . . . .	68
6.7	Parallel speedup and efficiency for MPI and pthreads DMR with naïve and load-balanced data partitioning. . . . .	69
6.8	Output from <code>jumpshot</code> tool showing MPI DMR beamformer state across 4 processes as a function of time. The processing in the top window was partitioned naïvely while the data in the bottom window was partitioned with some load-balancing. . . . .	71

# Chapter 1

## Introduction

Frequency-domain beamforming is the computational technique of inferring directional (usually azimuthal) phase and magnitude of arriving signals across an array of acoustic sensors from Fourier transformed element-level times series data. Many flavors of beamforming exist, where each falls under the category of “conventional” or “adaptive.” Conventional beamforming uses a fixed set of shading coefficients, or weights, whose properties are independent of the data. Past research in the field has also produced a wide variety of adaptive beamforming methods which seek to minimize total power output subject to a look-direction constraint [3] [10]. Adaptive beamformers almost universally provide higher quality output, which translates to displays that are easier to read and interpret. However, adaptive beamforming algorithms tend to be much more computationally expensive than conventional methods. Much of this cost is incurred in numerically evaluating the expression for the adaptive weights, which usually includes an  $O(n^3)$  complex Hermitian eigenvalue decomposition on the Fourier data’s cross-spectral density matrices. As a result of the higher demands upon the computing resources present, adaptive beamformers are sometimes limited in application and/or parameter configuration.

The potential for computational intensity in the beamforming process invites an implementation that will utilize the presence of a parallel computing architecture, whether it be a shared-memory symmetric multiprocessor (SMP) or a distributed-memory cluster of processors. Such high-performance parallel beamforming implementations are desirable for both real time environments such as on board naval vessels where incoming data is processed but not stored, and post-processing environments such as ARL:UT where previously recorded data segments may be accessed from magnetic storage mediums for further analysis. Any gains in beamformer performance benefit real time systems by allowing production configurations with more computationally demanding parameters, but also supports researchers as they rapidly develop and test better algorithms and implementations.

The purpose of this document is to trace the development of reasonably efficient, high-performance parallel implementations of a conventional and adaptive beamforming algorithm. For each beamformer, we will first formally derive a sequential algorithm, and then formulate a parallel algorithm based on a partitioning scheme chosen for the input data. Then, we analyze the complexity of the sequential and parallel algorithms. Outlines of various optimizations and related performance issues follow the analysis. And finally, we measure the actual performance of our Message Passing Interface (MPI) and POSIX threads (`pthread`s) implementations and compare the results with the performance predicted from the complexity analysis.

# Chapter 2

## Algorithm Theory

### 2.1 Input/Output Specification

We define frequency-domain beamforming input  $X$  as an  $n \times f \times t$  matrix. The matrix corresponds to the output of a Fast Fourier Transform, and contains data for the  $n$  acoustic sensors at each of the  $f$  Fourier frequency bins for  $t$  time sequences.

The beamforming transformation produces  $B$ , an  $l \times f \times t$  matrix representing the beamformed spectra for  $l$  look directions at each of the  $f$  Fourier frequency bins for  $t$  times sequences.

### 2.2 Beamforming

Westwood *et al* introduces beamforming by first describing the origin of the input data [19]. Element-level time series data is sampled at a rate of  $f_s$ , where each discrete time sequence consists of  $n_{FFT}$  samples. These data are then preprocessed with a Fast Fourier Transform (FFT) to move the data into the frequency domain. The duration of each time sequence (in seconds) is  $\Delta t = n_{FFT}/f_s$  and the separation of each Fourier frequency bin (in Hz) is  $\Delta f = f_s/n_{FFT}$ .

The general equation for frequency domain beamforming of the Fourier frequency bin  $f_j$  at time sequence  $t_i$  takes the form of

$$b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad (2.1)$$

where  $t_i$  is an index to the  $i$ th time sequence,  $f_j$  is an index to the  $j$ th Fourier frequency bin,  $c_j$  is an index to the center frequency associated with the  $j$ th Fourier frequency bin,  $b_{f_j t_i}$  is a  $l \times 1$  complex vector of beamformed spectra,  $x_{f_j t_i}$  is an  $n \times 1$  complex vector of Fourier spectra,  $W_{c_j t_i}$  is an  $n \times l$  complex weight matrix, and  $^H$  is the complex conjugate-transpose operator.

In the context of all references to Eq. (2.1),  $c_j$  should be treated as a mapping from the Fourier frequency bin  $f_j$ . The precise definition of this mapping is introduced in Sec. (2.2.1).

For conventional beamforming, the weight matrix  $W$  may be computed *a priori* and stored for future use. For adaptive beamforming, the weights are recomputed by sampling the data's most recent time sequences for statistical support.

Let us define the beamforming “look direction” as a combination of azimuthal bearing angle  $\phi$  measured east of north and vertical grazing angle  $\theta$ . The direction cosines for the look direction  $(\phi, \theta)$  are given by

$$p(\phi, \theta) = \sin \phi \cos \theta \hat{\mathbf{x}} + \cos \phi \cos \theta \hat{\mathbf{y}} + \sin \theta \hat{\mathbf{z}}. \quad (2.2)$$

For the beamforming look direction specified by  $p(\phi, \theta)$ , the expected arrival signal is described by the

plane wave<sup>1</sup> replica vector  $v$  (which remains constant as a function of time sequence) of dimension  $n \times 1$  whose  $v_j$  elements are given by

$$v = \exp \left[ i \frac{2\pi f_c}{c_{avg}} (a_k - c_{ref}) \cdot p(\phi, \theta) \right] \quad (2.3)$$

where  $f_c$  is the center frequency at which to beamform,  $c_{avg}$  is the average speed of sound across the sensor array,  $a_k$  is the  $xyz$  location of the  $k$ th acoustic element in the sensor array, and  $c_{ref}$  is the phase reference point. The phase reference is usually chosen with more care for dual-array cross-correlation beamforming, however, we will only address the simpler task of energy-detection beamforming which requires beamformed data from only one sensor array. By convention, we choose the geometric center of the array as our phase reference.

For conciseness, let us reference the replica vectors for all desired beamforming look directions at a single center frequency  $f_{c_j}$  by an  $n \times l$  replica matrix  $V_{c_j}$ ,

$$V_{c_j} = \exp \left[ i \frac{2\pi f_c}{c_{avg}} (A_{xyz} - C_{ref})^T P_{dc} \right] \quad (2.4)$$

where  $A_{xyz}$  is a  $3 \times n$  matrix whose  $i$ th column is the  $\hat{x}\hat{y}\hat{z}$  coordinate vector of the  $i$ th sensor of the acoustic array,  $C_{ref}$  is a  $3 \times n$  matrix containing  $c_{ref}$  in every element,  $P_{dc}$  is a  $3 \times l$  matrix whose  $i$ th column is the  $\hat{x}\hat{y}\hat{z}$  direction cosine vector for the  $i$ th look direction, and  $T$  is the matrix transpose.

### 2.2.1 CBF

Conventional beamforming (CBF) calls for complex weights which may be obtained directly from the replica vectors as

$$W = \frac{1}{n} V. \quad (2.5)$$

The weights are normalized by  $n$  such that the complex beamformer output in each look direction is unity:

$$w^H v = 1. \quad (2.6)$$

CBF weights are computed at center frequency bins  $c_j$ . Center frequency bins cover the same processing band as the Fourier frequency bins, but are typically spaced further apart. For CBF, the input parameter  $f_{bw}$  determines this spacing in units of Fourier frequency bins. A consequence of only computing the weights at center frequencies is that several Fourier frequency bins must “share” the same weights submatrix. That is, weights computed at some center frequency will be applied to all Fourier bins associated with that center frequency bin. When  $f_{bw} = 1$ , each center frequency bin maps to a single Fourier bin. Otherwise, two or more Fourier bins will be associated to one center frequency bin.

The number of center frequencies  $c$  used in beamforming is given by

$$c = \left\lfloor \frac{f}{f_{bw}} \right\rfloor. \quad (2.7)$$

The center frequency bin indices  $c_j$  are simply elements of the ordered set  $\{0, \dots, c-1\}$  which we will call  $C$ .

Let each center frequency bin be mapped to its corresponding lower and upper Fourier frequency bins by

$$\text{fbinlo}(c_j) = c_j f_{bw} \quad \forall c_j \in C, \quad (2.8)$$

$$\text{fbinhi}(c_j) = (c_j + 1) f_{bw} - 1 \quad \forall c_j \in C. \quad (2.9)$$

---

<sup>1</sup>Other forms of replica vectors exist. For instance, spherical wave replica vectors anticipate a signal arrival from a point source at some non-infinite range.

where  $F$  is an ordered set of indices  $\{0, \dots, f-1\}$ . The actual center frequency value for center frequency bin  $c_j$  can be computed as

$$\text{cfreq}(c_j) = \frac{1}{2} (\text{ffreq}(\text{fbinhi}(c_j)) - \text{ffreq}(\text{fbinlo}(c_j))) \quad \forall c_j \in C \quad (2.10)$$

where  $\text{ffreq}()$  is a function that maps a Fourier bin index to the corresponding Fourier frequency in Hz, and may be expressed as

$$\text{ffreq}(f_j) = f_j(\Delta f) = \frac{f_j f_s}{n_{FFT}} \quad \forall f_j \in F. \quad (2.11)$$

Also, the Fourier frequency bin  $f_j$  may be mapped to its corresponding center frequency bin  $c_j$  by

$$\text{cfbin}(f_j) = \left\lfloor \frac{f_j}{f_{bw}} \right\rfloor \quad \forall f_j \in F. \quad (2.12)$$

## 2.3 Adaptive Beamforming

An adaptive beamformer, in general, adapts its weights to the data, specifically by taking advantage of the assumptions that signals of interest are semi-stationary, and that noise will tend to be manifested as non-stationary processes across all sensors of the array. A cross-spectral density matrix (CSM) represents the covariance of each sensor against all other sensors, and is formed by the outer-product (cross-spectra) of a single vector of complex FFT spectra and its conjugate-transpose:

$$R_{f_j t_i} = x_{f_j t_i} x_{f_j t_i}^H \quad (2.13)$$

where  $x$  is an  $n \times 1$  complex vector from some Fourier frequency bin  $f_j$  at some time sequence  $t_i$ . Note that the *true* cross spectral density matrix for a given point in time and frequency,

$$R_{true} = E [x x^H], \quad (2.14)$$

is not precisely known[10], and can only be estimated through integration of CSM snapshots over time and frequency.

The estimate is centered at a center frequency bin  $c_j$ . The subsequently derived adaptive weights will be applied to all Fourier frequency bins which contributed to the CSM estimate, at some sequence within the time integration block.<sup>2</sup>

The number of samples integrated along the frequency axis is a user-defined parameter known as the CSM *bandwidth*. Since CSMs are computed at center frequency bins, this bandwidth parameter is equal to  $f_{bw}$ . Assuming that all CSM frequency bands are disjoint and of fixed bandwidth, the center frequencies generated are characterized the same as center frequencies for CBF described above<sup>3</sup>.

For consistency, we will refer to center frequencies in the context of CBF as well as adaptive beamforming. And while there is no averaging of cross-spectra is needed for CBF,  $f_{bw}$  is still required in order to specify the frequency separation of the CBF weights.

For time averaging, let us consider only sliding block averaging.<sup>4</sup> A sliding block average is parameterized by the sliding block duration,  $t_{blk}$ , in units of Fourier time sequences, which characterizes each CSM estimate's time integration. Within the sliding block structure lie the the most recent  $t_{blk}$  frequency-integrated CSMs for a given center frequency bin. These CSMs, integrated across frequency, are then integrated uniformly across the length of the sliding block every time the weights are to be recomputed to form the estimate given by

<sup>2</sup>Usually weights are applied to the most recent Fourier sequence, but they may be optimally centered within the time integration block if some latency can be tolerated.

<sup>3</sup>These relations fail when individual CSM frequency bands are allowed to overlap.

<sup>4</sup>Other methods of CSM time averaging exist, such as  $\alpha$ -based exponential.

$$R_{c_j t_i} = \sum_{t_i=0}^{t_{blk}-1} \sum_{f_j=f_{binlo}(c_j)}^{f_{binhi}(c_j)} R_{f_j, t_i}. \quad (2.15)$$

The product  $f_{bw} t_{blk}$  is known as the *time-bandwidth* product of the CSM, and represents the total number of samples that provide support to the CSM estimate before it is decomposed of its eigenstructure.

A complex Hermitian eigenvalue decomposition (EVD) yields [4]:

$$R = U \Lambda U^H \quad (2.16)$$

where  $\Lambda$  is a diagonal matrix of eigenvalues  $\lambda_j$  of  $R$ , and the columns  $u_j$  of  $U$  are the orthonormal eigenvectors of  $R$ , such that  $U^H U = I$ . The inverse of the covariance matrix  $R$  is needed when constructing adaptive weights:

$$R^{-1} = U \text{diag} \left( \frac{1}{\lambda_j} \right) U^H. \quad (2.17)$$

### 2.3.1 MVDR

The source of many forms of adaptive beamforming is the Minimum Variance Distortionless Response (MVDR) beamformer. It has been shown that minimum power output subject to the unity constraint of Eq. (2.6) in the look direction characterized by the replica vector  $v$  takes the form of

$$w = \frac{R^{-1} v}{v^H R^{-1} v} \quad (2.18)$$

Substituting Eq. (2.17) into Eq. (2.18) gives basic MVDR weights,

$$\begin{aligned} w &= \frac{U \text{diag} \left( \frac{1}{\lambda_j} \right) U^H v}{v^H U \text{diag} \left( \frac{1}{\lambda_j} \right) U^H v} \\ &= \frac{U \text{diag} \left( \frac{1}{\lambda_j} \right) U^H v}{\sum_{j=0}^{n-1} \left( \frac{1}{\lambda_j} \right) |u_j^H v|^2}. \end{aligned} \quad (2.19)$$

### 2.3.2 DMR

Developed by Abraham and Owsley [3], dominant mode rejection (DMR) is a derivative of the MVDR algorithm that employs only the most dominant  $D$  eigenvalues and eigenvectors of  $R$ . By using only a subset of the eigenvalues, the CSM may be estimated though shorter frequency and time integration. Let us define the an approximation to  $R$  that only uses the largest  $D$  eigenvalues and eigenvectors; a white noise parameter  $\epsilon$  is injected along the diagonal (also known as diagonal loading) for robustness and to facilitate inversion:

$$\hat{R}_D = U_D \Lambda_D U_D^H + \epsilon I_n \quad (2.20)$$

where  $U_D$  is an  $n \times D$  matrix containing the  $D$  eigenvectors corresponding to the most dominant  $D$  eigenvalues,  $\Lambda_D$  is a  $D \times D$  diagonal matrix of the most dominant eigenvalues, and  $I_n$  is an  $n \times n$  identity matrix.

By inverting  $\hat{R}_D$  (whose derivation can be found in [19]) and substituting into Eq. (2.18), we obtain the DMR weights vector for the look direction  $l_k$ ,

$$\begin{aligned}
w &= \frac{v - U_D \text{diag} \left( \frac{\lambda_j}{\lambda_j + \epsilon} \right) U_D^H v}{n - v^H U_D \text{diag} \left( \frac{\lambda_j}{\lambda_j + \epsilon} \right) U_D^H v} \\
&= \frac{v - U_D \text{diag} \left( \frac{\lambda_j}{\lambda_j + \epsilon} \right) U_D^H v}{n - \sum_{j=0}^{D-1} \frac{\lambda_j}{\lambda_j + \epsilon} |u_j^H v|^2}.
\end{aligned} \tag{2.21}$$

However, the DMR weights equation is numerically unstable when  $\epsilon$  is small relative to  $\lambda_j$  and  $u_j^H v \approx 1$  for all  $j$ . An  $\epsilon$  chosen near zero causes the summation term to remain nearly  $n$ , which in turn causes the denominator to take on a value near zero, possibly leading to imprecise values for the weight vector. To remedy this, we rewrite Eq. (2.21) as

$$\begin{aligned}
w &= \frac{v - U_D \text{diag} \left( \frac{\lambda_j}{\lambda_j + \epsilon} \right) U_D^H v}{n - \sum_{j=0}^{D-1} \frac{\lambda_j}{\lambda_j + \epsilon} |u_j^H v|^2} \\
&= \frac{v - U_D \left( I_D - \text{diag} \left( \frac{\epsilon}{\lambda_j + \epsilon} \right) \right) U_D^H v}{n - \sum_{j=0}^{D-1} \left( 1 - \frac{\epsilon}{\lambda_j + \epsilon} \right) |u_j^H v|^2} \\
&= \frac{v - U_D \left( U_D^H v - \text{diag} \left( \frac{\epsilon}{\lambda_j + \epsilon} \right) U_D^H v \right)}{n - \sum_{j=0}^{D-1} \left( |u_j^H v|^2 - \frac{\epsilon}{\lambda_j + \epsilon} |u_j^H v|^2 \right)} \\
&= \frac{v - U_D U_D^H v + U_D \text{diag} \left( \frac{\epsilon}{\lambda_j + \epsilon} \right) U_D^H v}{n - \sum_{j=0}^{D-1} |u_j^H v|^2 + \sum_{j=0}^{D-1} \frac{\epsilon}{\lambda_j + \epsilon} |u_j^H v|^2}
\end{aligned} \tag{2.22}$$

Instability problems in the weight computation for small values of  $\epsilon$  can be avoided by using using Eq. (2.22), which is algebraically equivalent to the original Eq. (2.21).

Methods of choosing appropriate white noise  $\epsilon$  exist, and vary in computational complexity. Some methods such as the White Noise Gain Constraint described by Cox *et al* [8] select a unique  $\epsilon$  for each look direction beamformed. For simplicity, we use a method provided by Westwood that defines  $\epsilon$  independently of look direction as a fraction of the average omni-directional power received across all beamformed sensors [19]:

$$\epsilon = \frac{\mu}{n} \sum_{j=0}^{n-1} R_{jj}. \tag{2.23}$$

where  $\mu$  is a user-defined parameter typically on the interval [0.001,0.1].

## Chapter 3

# Algorithm Derivation

### 3.1 Notation

To aid in our derivation of algorithms and their correctness, we present a set of notational conventions used throughout this section [5].

Statements indicate a change of value to a variable or set of variables such that the state of the program has changed. Predicates assert the state of the variables in the program between statements. For example, the statement

$$v := \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

assigns the  $2 \times 1$  vector  $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$  to the vector variable  $v$ . So, after this statement we can indicate the state of  $v$  with the predicate

$$\left\{ v = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right\}.$$

Specifically, the state of variables before a statement  $S$  is known as the statement's *precondition* while the state of variables after  $S$  is referred to as the statement's *postcondition*. The *Hoare triplet*  $\{P_1\}S\{P_2\}$  is true if and only if  $P_2$  is true after the execution of  $S$ , given that  $P_1$  is true before the statement  $S$ .

We will also refer to *loop-invariants*, which are predicates found within loops that hold true upon entering the loop, during the loop's iteration, and also upon exiting the loop. A loop *guard* is a predicate that determines whether the body of the loop will be executed at the beginning of each iteration.

The following sections will deal heavily with vectors, matrices, and multi-dimensional matrices. We would like to meaningfully indicate both information about the number of dimensions present in the matrix as well as the origin of the vector or matrix if it is a reference to subset of the larger matrix. Thus, we will use the following rules and notations when referring to matrix variables:

- All vectors are column-oriented, and will be labeled as lowercase letters, such as  $v$ ,  $b$ , or  $x$ .
- "Full" matrices will be indicated by uppercase letters<sup>1</sup> with no subscripts. A full matrix is a matrix that is not a reference into a larger matrix. Examples of instances of full matrices that will occur frequently in the derivations are  $V$ ,  $B$ , and  $X$ .
- A vector, denoted by some lowercase letter, will always be a subset of its parental full matrix that is referenced by the equivalent uppercase letter. So,  $v$  is some vector within the full matrix  $V$ .

---

<sup>1</sup>Note that uppercase letters are used to identify other constructs as well, such as predicates and sets of indices. Context and previous usages of the letter should be enough to identify the type (ie: matrix, predicate, set, etc.)

- A submatrix is a matrix of at least two dimensions that contains at least one fewer dimension than its corresponding full matrix. Most occurrences of submatrices will be two-dimensional, where their parental full matrix is three-dimensional (except in the case of the DMR weights, which are four-dimensional).
- Subscripts will help denote the location of a vector or submatrix within its corresponding full matrix. For example,  $X_{t_i}$  is the  $i$ th submatrix along the  $t$  dimension of the full matrix  $X$ , and  $v_{f_j t_i}$  is the  $j$ th vector along the  $f$  dimension of the submatrix  $V_{t_i}$ , which is the  $i$ th submatrix along the  $t$  dimension of the full matrix  $V$ .
- In variables with multiple indexing subscripts, lower dimensional indices will appear nearest to the variable while the higher dimensional indices will appear furthest from the variable name. Given an  $n \times 1$  vector  $x_{f_j t_i}$ , one can infer that the full matrix to which  $x$  belongs is  $n \times f \times t$ .

## 3.2 Correctness of loops

Gunnels *et al* suggests the following approach to the formal derivation of a loop [13] [15]:

1. Determine a loop-invariant  $P_{inv}$ .
2. Determine a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies that the desired operation has been computed.
3. Determine the initialization  $S_I$  so that  $P_{inv}$  holds true before entering the loop.
4. Determine the repetend  $S_B$  so that eventually  $G$  is false and the loop-invariant  $P_{inv}$  is maintained.

This recipe for derivation not only facilitates developing loops from scratch, but also yields loops whose correctness is verified. As a template for loop derivation, we will use the following layout of the aforementioned statements and predicates to ensure loop correctness [5].

```

{ $P_{pre}$ }
{ $S_I$ }
{ $P_{inv}$ }
while  $G$  do
  { $P_{inv} \wedge G$ }
   $S_B$ 
  { $P_{inv}$ }
enddo
{ $P_{inv} \wedge \neg G$ }
{ $P_{post}$ }

```

Every loop has a precondition and postcondition denoted  $P_{pre}$  and  $P_{post}$ , which are predicates that describe variable states before and after execution of the loop. The loop invariant,  $P_{inv}$ , is a predicate which holds true for each iteration of the loop. The loop guard,  $G$ , is some statement that describes the condition necessary for the loop to continue such that  $P_{inv} \wedge \neg G$  is true when the loop terminates. The loop initialization and body statements are abbreviated  $S_I$  and  $S_B$ , respectively.

## 3.3 Sequential CBF

Using the notations given in the previous section, and the template described above, we will derive the CBF algorithm.

Let  $\hat{P}_{pre}$  and  $\hat{P}_{post}$  denote the precondition and postcondition, respectively, of the entire program, while  $P_{pre}$  and  $P_{post}$  will refer to the precondition and postcondition of some subset of the program such as a statement, compound statement, or loop, depending on context. Thus far we know only  $\hat{P}_{pre}$  and  $\hat{P}_{post}$ :

$$\hat{P}_{pre} : \{\text{inputOK}_{CBF}\}$$

$$\hat{P}_{post} : \left\{ b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in T \right\}$$

where  $\text{inputOK}_{CBF}$  is true if and only if

$$X \text{ is } n \times f \times t \wedge$$

$$n \geq 1 \wedge f \geq 1 \wedge t \geq 1 \wedge l \geq 1 \wedge f_{bw} \geq 1 \wedge$$

$$\{(\theta, \phi)\} \neq \emptyset.$$

evaluates to true,  $F$  is the set of indices  $\{0, \dots, f-1\}$ , and  $T$  is the set of indices  $\{0, \dots, t-1\}$ . We wish to derive a program,  $S_{CBF}$ , that satisfies

$$\{\text{inputOK}_{CBF}\}$$

$$S_{CBF}$$

$$\{b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in T\}$$

Let us begin by considering Eq. (2.1). The matrix  $W_{c_j t_i}$  is  $n \times l$  and  $x_{f_j t_i}$  is  $n \times 1$ , thus the matrix-vector multiplication  $W_{c_j t_i}^H x_{f_j t_i}$  is well-defined and produces a  $l \times 1$  vector,  $b_{f_j t_i}$  containing the beamformed spectra in each of the  $l$  look directions at each Fourier frequency bin  $f_j$  for each time sequence  $t_i$ .

Recall that the weights matrix  $W$  is data independent for CBF. Furthermore, the CBF weight matrix does not vary with  $t_i$ . So,

$$W_{c_j t_i} = W_{c_j t_h} = W_{c_j} \forall c_j \in C, \forall t_i, t_h \in T$$

and thus we may simplify our discussion henceforth by removing the time sequence dimension and referring to the  $n \times l \times c$  CBF weights matrix as  $W$ .

### 3.3.1 Derivation of inner loop over Fourier frequency bin to beamform

The core beamforming computation of the CBF algorithm for one time sequence  $t_i$  is described by the statement which assigns the vector  $b_{f_j}$  with the result of the matrix-vector multiplication of  $W_{c_j}^H$  and  $x_{f_j}$  for all  $f$  Fourier frequency bins:

$$b_{f_j t_i} := W_{c_j}^H x_{f_j t_i}$$

where center frequency bin  $c_j$  is a function of  $f_j$ . For the remainder of this subsection, we will focus on derivation of the CBF algorithm with respect to Fourier frequency bin, and so we will treat the  $t_i$  index for  $B$  and  $X$  as a *fixed* arbitrary time sequence index without any universal quantification. Thus, any occurrence of the  $t_i$  index should be interpreted as describing the variable in question as having a time sequence dimension equal to 1. Instead of removing the index from this inner loop derivation altogether, we maintain consistent indexing so the work done here better integrates with the derivation of the outer time sequence loop in Sec. 3.3.2.

Because we will continue derivation with respect to a single time sequence  $t_i$ , we reason that the inner Fourier frequency loop's precondition is

$$P_{pre} : \{ \text{uninitialized}(B_{t_i}) \wedge \text{computed}_{\text{CBF}}(W) \}$$

where  $B_{t_i}$  is a reference to a single  $l \times f$  submatrix of  $B$ ,  $\text{uninitialized}(B_{t_i})$  is a predicate that is true if and only if the submatrix  $B_{t_i}$  has not yet been initialized, and  $\text{computed}_{\text{CBF}}(W)$  is a predicate that is true if and only if

$$W_{c_j} = \frac{1}{n} \exp \left[ i \frac{2\pi c \text{freq}(c_j)}{c_{avg}} (A_{xyz} - C_{ref})^T P_{dc} \right] \quad \forall f_j \in F. \quad (3.1)$$

which is derived by substituting Eq. (2.4) into Eq. (2.5). The predicate  $\text{computed}_{\text{CBF}}(W)$  is included in the inner loop's precondition to ensure that the weights for all center frequency bins are ready to be used in the computation. The inner loop postcondition equals the program's postcondition, sans the universal quantifier over  $T$ .

$$P_{post} : \{ b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in F \},$$

Note that the beamformed spectra vector  $b_{f_j t_i}$  depends only on  $W_{c_j}$  and  $x_{f_j t_i}$  for all  $f_j \in F$ . So, once a column vector from each of  $B_{t_i}$  and  $X_{t_i}$  are processed, they need not be accessed again for future computation. We will also assume that the column vectors of  $B_{t_i}$  and  $X_{t_i}$  are processed with an ascending index  $f_j$  (ie: the data vectors are processed upward in frequency). Let us partition  $B_{t_i}$ ,  $X_{t_i}$ , and  $W$  each into two matrices along the  $f$  dimension characterized by those column vectors which have been processed and those which have not yet been processed:

$$\begin{aligned} B_{t_i} &\rightarrow \left( B_{proc} \parallel B_{unproc} \right) \\ X_{t_i} &\rightarrow \left( X_{proc} \parallel X_{unproc} \right) \\ W &\rightarrow \left( W_{proc} \parallel W_{unproc} \right) \end{aligned}$$

where  $B_{proc}$  is  $l \times f_{proc}$ ,  $B_{unproc}$  is  $l \times f_{unproc}$ ,  $X_{proc}$  is  $n \times f_{proc}$ ,  $X_{unproc}$  is  $n \times f_{unproc}$ ,  $W_{proc}$  is  $n \times l \times c_{proc}$ ,  $W_{unproc}$  is  $n \times l \times c_{unproc}$ ,  $f = f_{proc} + f_{unproc}$ , and  $c = c_{proc} + c_{unproc}$ .

Combining these partitionings with the postcondition  $P_{post}$  we find that

$$b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in \{0, \dots, f_{proc} - 1\} \quad (3.2)$$

$$b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in \{f_{proc}, \dots, f - 1\}. \quad (3.3)$$

These newfound equations can be interpreted as follows: at any given time in the inner loop, Eq. (3.2) implies that beamforming on the vector  $x_{f_j t_i}$  for all  $j \in \{0, \dots, f_{proc} - 1\}$  has completed with the results residing in the corresponding vectors  $b_{f_j t_i}$ ; likewise, Eq. (3.3) describes how the beamforming will eventually take place for Fourier frequency bins  $j \in \{f_{proc}, \dots, f - 1\}$  that have not yet been processed. It is important to notice that the matrix partition  $B_{unproc}$  is always uninitialized since it has not yet been updated with corresponding values of  $W_{c_j}^H x_{f_j t_i}$  for all  $j \in \{f_{proc}, \dots, f - 1\}$ .

It is more natural to describe the loop-invariant in terms of computations that have taken place rather than computations that have yet to take place, so the state described by Eq. (3.2) will serve as our loop-invariant  $P_{inv}$ :

$$P_{inv} : \left( b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in \{0, \dots, f_{proc} - 1\} \wedge \text{uninitialized}(b_{f_j t_i}) \quad \forall f_j \in \{f_{proc}, \dots, f - 1\}. \right) \quad (3.4)$$

where the predicate  $\text{uninitialized}(b_{f_j t_i})$  is true if and only if all elements of the column vector  $b_{f_j t_i}$  have not yet been initialized.

Given the loop-invariant in (3.4), we must derive a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies the loop postcondition  $P_{post}$ :

**Repartition**

$$\begin{aligned} \left( \begin{array}{c} B_{proc} \\ X_{proc} \end{array} \parallel \begin{array}{c} B_{unproc} \\ X_{unproc} \end{array} \right) &\rightarrow \left( \begin{array}{c} B_{FL} \\ X_{FL} \end{array} \parallel \begin{array}{c} b_{f_j t_i} \\ x_{f_j t_i} \end{array} \mid \begin{array}{c} B_{FR} \\ X_{FR} \end{array} \right) \end{aligned}$$

**Repartition if**  $f_{proc} \bmod f_{bw} = 0$

$$\left( \begin{array}{c} W_{proc} \\ \end{array} \parallel \begin{array}{c} W_{unproc} \\ \end{array} \right) \rightarrow \left( \begin{array}{c} W_{CL} \\ \end{array} \parallel \begin{array}{c} W_{c_j} \\ \end{array} \mid \begin{array}{c} W_{CR} \\ \end{array} \right)$$

**where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j}$  **is**  $n \times l$ ,  
 $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .

$\{Q_{bu}\}$

$S_u$

$\{Q_{au}\}$

**Continue with**

$$\begin{aligned} \left( \begin{array}{c} B_{proc} \\ X_{proc} \end{array} \parallel \begin{array}{c} B_{unproc} \\ X_{unproc} \end{array} \right) &\leftarrow \left( \begin{array}{c} B_{FL} \\ X_{FL} \end{array} \mid \begin{array}{c} b_{f_j t_i} \\ x_{f_j t_i} \end{array} \parallel \begin{array}{c} B_{FR} \\ X_{FR} \end{array} \right) \end{aligned}$$

**Continue with if**  $(f_{proc} + 1) \bmod f_{bw} = 0$

$$\left( \begin{array}{c} W_{proc} \\ \end{array} \parallel \begin{array}{c} W_{unproc} \\ \end{array} \right) \leftarrow \left( \begin{array}{c} W_{FL} \\ \end{array} \mid \begin{array}{c} W_{c_j} \\ \end{array} \parallel \begin{array}{c} W_{CR} \\ \end{array} \right)$$

**where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j}$  **is**  $n \times l$ ,  
 $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .

Figure 3.1: Layout for body of CBF loop over Fourier frequency bin

$$(P_{inv} \wedge \neg G) \Rightarrow (P_{post}).$$

Consider the  $G$ :  $f_{proc} \neq f$ . The predicate  $P_{inv} \wedge \neg G$  implies that  $f = f_{proc}$ ,  $B = B_{proc}$ , and so  $b_{f_j t_i} = W_{c_j}^H x_{f_j t_i}$  for all  $j \in F$ , which describes the same state described by the loop's postcondition. Thus the predicate  $G$  can serve as our loop guard in the conjunction that implies completion of the loop:

$$(P_{inv} \wedge \neg G) \Rightarrow (b_{f_j t_i} = W_{f_j}^H x_{f_j t_i} \forall f_j \in F).$$

Next we must find what loop initialization statement  $S_I$  must execute to make  $P_{inv}$  hold true before the loop begins execution. Such a statement does not perform any computation, but rather simply performs the partitioning necessary to move the program into a state that satisfies the loop-invariant. Consider the statement  $S_I$ ,

**Partition**

$$\begin{aligned} B_{t_i} &\rightarrow \left( \begin{array}{c} B_{proc} \\ X_{proc} \\ W_{proc} \end{array} \parallel \begin{array}{c} B_{unproc} \\ X_{unproc} \\ W_{unproc} \end{array} \right) \\ X_{t_i} &\rightarrow \left( \begin{array}{c} X_{proc} \\ \end{array} \parallel \begin{array}{c} X_{unproc} \\ \end{array} \right) \\ W &\rightarrow \left( \begin{array}{c} W_{proc} \\ \end{array} \parallel \begin{array}{c} W_{unproc} \\ \end{array} \right) \end{aligned}$$

**where**  $B_{proc}$  **is**  $l \times 0$ ,  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ ,  
 $f_{proc} = 0$ ,  $f_{unproc} = f$ ,  $c_{proc} = 0$ , **and**  $c_{unproc} = c$

The statement initialization partitions the matrices  $B_{t_i}$ ,  $X_{t_i}$ , and  $W$  and establishes the processed partitions of  $B_{t_i}$  and  $X_{t_i}$  as being empty, meaning no computation has occurred.

After the initialization statement  $S_I$ , all of  $B_{t_i}$  is uninitialized and due to the precondition, the CBF weight matrix is computed and resides in  $W$ , thus  $P_{inv}$  is true.

Now that a loop-invariant, loop guard, and initialization statement are known, we may derive the inner loop body. Computation should move the program toward a state in which  $G$  is false. The idea is to let  $X_{proc}$ ,  $B_{proc}$ , and  $W_{proc}$  expand as the algorithm progresses until  $X_{t_i} = X_{proc}$ ,  $B_{t_i} = B_{proc}$ , and  $W = W_{proc}$ , implying that  $X_{unproc}$  is  $n \times 0$ ,  $B_{unproc}$  is  $l \times 0$ , and  $W_{unproc}$  is  $n \times l \times 0$ . We begin deriving the loop body in Fig. (3.1) by repartitioning the matrices to outline the progress made by the core update statement  $S_u$ .

The purpose of this repartitioning is to expose individual column vectors of  $B_{t_i}$  and  $X_{t_i}$ , and submatrices of  $W$ , which have not yet been processed. However,  $W$  may not need be repartitioned for every iteration of the loop. Since several Fourier frequency bins may map to the same center frequency bin, we will only expose a new submatrix of weights if the computation is moving on to process Fourier bins that associate with a new center frequency. From Eq. (2.12) we see that as Fourier frequency bin increases, the bins become associated with the next higher center frequency bin when  $f_{proc}/f_{bw} = \lfloor f_{proc}/f_{bw} \rfloor$ . In other words, since the center frequencies are separated by  $f_{bw}$ , a new center frequency's weights should be extracted when  $f_{proc} \bmod f_{bw} = 0$ . So, we denote the repartitioning of  $W$  as a conditional with the **Repartition if** statement. The **Continue with if** statement is based on a similar conditional, though here we wish to merge  $W_{c_j}$  with the other previously processed weight submatrices when the next Fourier bin belongs to a new center frequency bin. This happens when the next iteration will need the weight submatrix computed at the next higher center frequency bin, which occurs when  $(f_{proc} + 1) \bmod f_{bw} = 0$ .

In the repartitioning notation, a single line separates a newly-exposed column vector from its originating matrix, while the double lines more firmly separate the processed and unprocessed partitions of the matrix in question.

Notice that the column vectors and submatrices extracted from the unprocessed partitions of  $B_{t_i}$ ,  $X_{t_i}$ , and  $W$  are reassigned to their respective processed partitions after the inner loop's core update.

Let us abbreviate the ordered set of indices  $j \in \{0, \dots, f_{proc} - 1\}$  as  $F_L$  and the ordered set of indices  $j \in \{f_{proc} + 1, \dots, f - 1\}$  as  $F_R$ . Also, let us denote the set of indices  $c \in \{0, \dots, c_{proc} - 1\}$  as  $C_L$  and  $j \in \{c_{proc} + 1, \dots, c - 1\}$  as  $C_R$ . After the loop body's first repartitioning,

$$( B_{proc} = B_{F_L} \parallel B_{unproc} = ( b_{f_{proc}t_i} \mid B_{F_R} ) ) \quad (3.5)$$

$$( X_{proc} = X_{F_L} \parallel X_{unproc} = ( x_{f_{proc}t_i} \mid X_{F_R} ) ) \quad (3.6)$$

$$( W_{proc} = W_{C_L} \parallel W_{unproc} = ( W_{c_{proc}} \mid W_{C_R} ) ) \quad (3.7)$$

where  $B_{F_L}$  is  $l \times f_{proc}$ ,  $B_{F_R}$  is  $l \times (f_{unproc} - 1)$ ,  $X_{F_L}$  is  $n \times f_{proc}$ ,  $X_{F_R}$  is  $n \times (f_{unproc} - 1)$ ,  $W_{C_L}$  is  $n \times l \times c_{proc}$ ,  $W_{C_R}$  is  $n \times l \times (c_{unproc} - 1)$ ,  $b_{f_{proc}t_i}$  is  $l \times 1$ ,  $x_{f_{proc}t_i}$  is  $n \times 1$ , and  $W_{c_{proc}}$  is  $n \times l$ . Note that the repartitioning of  $W$  only occurs when  $f_{proc} \bmod f_{bw} = 0$ .

The predicates  $Q_{bu}$  denotes the program state after the inner loop body's first repartitioning but before the core update statement has executed. Here,  $Q_{bu}$  is derived by substituting the matrices equated in Eq. (3.5) and (3.6) into the loop-invariant  $P_{inv}$ :

$$Q_{bu} : \left( \begin{array}{c} B_{proc} = ( b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \forall f_j \in F_L ) \wedge \\ B_{unproc} = \left( \begin{array}{c|c} \text{uninitialized}(b_{f_{proc}t_i}) & \text{uninitialized}(b_{f_j t_i}) \\ \hline & \forall f_j \in F_R \end{array} \right) \end{array} \right).$$

Similarly, the predicate  $Q_{au}$  corresponds to the state of the program after the core update statement  $S_u$  but before the loop body's second repartitioning,

$$( B_{proc} = ( B_{F_L} \mid b_{f_{proc}t_i} ) \parallel B_{unproc} = B_{F_R} ) \quad (3.8)$$

$$( X_{proc} = ( X_{F_L} \mid x_{f_{proc}t_i} ) \parallel X_{unproc} = X_{F_R} ) \quad (3.9)$$

$$( W_{proc} = ( W_{C_L} \mid W_{c_{proc}} ) \parallel W_{unproc} = W_{C_R} ) \quad (3.10)$$

which merges the newly-processed column vectors  $b_{f_j t_i}$  and  $x_{f_j t_i}$  with the other processed column vectors residing in  $B_{proc}$  and  $X_{proc}$ , respectively. Note that the most recently extracted submatrix of  $W$  is only merged with the previously processed weight submatrices when  $(f_{proc} + 1) \bmod f_{bw} = 0$ .

The predicate  $Q_{au}$  should describe a state after the update of  $B_{t_i}$  in which  $P_{inv}$  still holds true. More simply,  $Q_{au}$  should be equal to  $P_{inv}$  except that it should also exhibit the appropriate update of the two column vectors which were just processed:

$$Q_{au} : \left( B_{proc} = \left( \begin{array}{l} b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \\ \forall f_j \in F_L \end{array} \middle| b_{f_{proc} t_i} = W_{c_{proc}}^H x_{f_{proc} t_i} \right) \wedge \right. \\ \left. B_{unproc} = (\text{uninitialized}(b_{f_j t_i}) \forall f_j \in F_R) \right).$$

To find the core update statement  $S_u$  of the loop body  $S_{CBF}$ , we observe the state  $Q_{bu}$  of the program before  $S_u$  with the state  $Q_{au}$  of the program after  $S_u$ . Specifically, the interesting portions of the state transition can be reduced to

$$\{Q_{bu} : (\text{uninitialized}(b_{f_{proc} t_i}))\} \\ S_u \\ \{Q_{au} : (b_{f_{proc} t_i} = W_{c_{proc}}^H x_{f_{proc} t_i})\}.$$

From this, we infer that the appropriate update for  $S_u$  is

$$b_{f_{proc} t_i} := W_{c_{proc}}^H x_{f_{proc} t_i}.$$

After the newly-processed column vectors  $b_{f_{proc}}$  and  $x_{f_{proc}}$  are merged with their respective processed partitions, we refer to them as officially “processed”, and thus there exists an implicit statement after the second partitioning that updates the values of  $f_{proc}$  with  $f_{proc} + 1$  and  $f_{unproc}$  with  $f_{unproc} - 1$ . A similar update is made to  $c_{proc}$  and  $c_{unproc}$  as part of the **Continue with if** statement.

### 3.3.2 Derivation of outer loop over time sequence

With the CBF inner loop over Fourier frequency bin  $f_j$  derived, we proceed to derive the CBF outer loop over time sequence  $t_i$ .

Recall the core beamforming computation introduced at the beginning of Sec. 3.3.1.

$$b_{f_j t_i} := W_{c_j}^H x_{f_j t_i}.$$

Here, we will focus on the outer loop of the CBF algorithm, which will iterate over time sequence  $t_i$ . For this derivation subsection, we will denote the beamformed spectra vectors  $b_{f_j t_i}$  for all  $f \in F$  at a some time sequence  $t_i$  as a beamformed spectra matrix  $B_{t_i}$ . Similarly, the Fourier spectra vectors  $x_{f_j t_i}$  for all  $f \in F$  at some time sequence  $t_i$  will be denoted by the Fourier spectra matrix  $X_{t_i}$ . Since the CBF weight matrix  $W$  does not change with time, we may safely omit it from the matrix partitionings for this section.

We reason that the outer time sequence loop’s postcondition equals the program’s postcondition under the argument that all useful beamforming-related computation will cease after the time sequence loop terminates. We do not expect, however, that  $P_{pre} = \hat{P}_{pre}$  since the outer loop precondition contain additional predicates which were not met at the time the program began execution. Specifically, the full weights matrix  $W$  must be computed as in Eq. (3.1) before the outer loop begins iterating. The  $P_{pre}$  and  $P_{post}$  for the outer loop over time sequence are

$$P_{pre} : \{ \text{uninitialized}(B) \wedge \text{computed}_{CBF}(W) \} \\ P_{post} = \hat{P}_{post} : \{ b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in T \},$$

where  $\text{uninitialized}(B)$  is true if and only if the submatrices  $B_{t_i}$  for all  $t_i \in T$  have not yet been initialized.

Note that the beamformed spectra submatrix  $B_{t_i}$  depends only on  $W$  and  $X_{t_i}$  for all  $t_i \in T$ . Thus, once a submatrix from each of the full matrices  $B$  and  $X$  are processed, they need not be accessed again for future computation. As with the inner loop over Fourier frequency, we assume the submatrices are processed with ascending index  $t_i$  (ie: the data sequences are processed forward in time). Let us partition the full matrices  $B$  and  $X$  each into two matrices along the  $t$  dimension characterized by those submatrices which have been processed and those which have not yet been processed.

$$\begin{aligned} B &\rightarrow \left( B_{proc} \parallel B_{unproc} \right) \\ X &\rightarrow \left( X_{proc} \parallel X_{unproc} \right) \end{aligned}$$

where  $B_{proc}$  is  $l \times f \times t_{proc}$ ,  $B_{unproc}$  is  $l \times f \times t_{unproc}$ ,  $X_{proc}$  is  $n \times f \times t_{proc}$ ,  $X_{unproc}$  is  $n \times f \times t_{unproc}$ , and  $t = t_{proc} + t_{unproc}$ .

Substituting these partitionings into the outer loop postcondition  $P_{post}$  we find that

$$b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in \{0, \dots, t_{proc} - 1\} \quad (3.11)$$

$$b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in \{t_{proc}, \dots, t - 1\}. \quad (3.12)$$

So, at any given time in the outer loop, Eq. (3.11) implies that beamforming on the vector  $x_{f_j t_i}$  for all  $f_i \in F$  and all  $t_i \in \{0, \dots, t_{proc} - 1\}$  has completed with the results residing in the corresponding vectors  $b_{f_j t_i}$ ; Similarly, Eq. (3.12) describes how the beamforming will eventually take place for time sequences  $t_i \in \{t_{proc}, \dots, t - 1\}$  that have not yet been processed. It is important to notice that at any point in the scope of the outer loop, the matrix partition  $B_{unproc}$  is uninitialized since it has not yet been updated with corresponding values of  $W_{f_j}^H x_{f_j t_i}$  for all  $t_i \in \{t_{proc}, \dots, t - 1\}$ .

The state described by Eq. (3.11) will serve as our outer loop-invariant  $P_{inv}$ :

$$P_{inv} : \left( \begin{array}{l} \text{computed}_{CBF}(W) \wedge \\ b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in \{0, \dots, t_{proc} - 1\} \wedge \\ \text{uninitialized}(b_{f_j t_i}) \quad \forall f_j \in F, \forall t_i \in \{t_{proc}, \dots, t - 1\} \end{array} \right) \quad (3.13)$$

where the predicate  $\text{uninitialized}(b_{f_j t_i})$  is true if and only if all elements of the column vector  $b_{f_j t_i}$  have not yet been initialized.

Given the loop-invariant in (3.13), we must derive a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies the loop postcondition  $P_{post}$ :

$$(P_{inv} \wedge \neg G) \Rightarrow (P_{post}).$$

Consider the  $G$ :  $t_{proc} \neq t$ . The predicate  $P_{inv} \wedge \neg G$  implies that  $t = t_{proc}$ ,  $B = B_{proc}$ , and thus  $b_{f_j t_i} = W_{c_j}^H x_{f_j t_i}$  for all  $f_j \in F$ ,  $t_i \in T$ , which describes the same state described by the outer loop's postcondition. Thus the predicate  $G$  can serve as the outer loop guard in the conjunction that implies completion of the loop:

$$(P_{inv} \wedge \neg G) \Rightarrow (b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in T).$$

Next we must find what loop initialization statement  $S_I$  must execute to make  $P_{inv}$  hold true before the loop begins execution. Such a statement will should include provisions for the weight computation, and also perform the partitioning necessary to move the program into a state that satisfies the loop-invariant. Consider the statement  $S_I$ ,

$S_{CBFweights}$

**Partition** 
$$\begin{aligned} B &\rightarrow \left( B_{proc} \parallel B_{unproc} \right) \\ X &\rightarrow \left( X_{proc} \parallel X_{unproc} \right) \end{aligned}$$

**where**  $B_{proc}$  **is**  $l \times f \times 0$ ,  $X_{proc}$  **is**  $n \times f \times 0$ , **and**  $t_{proc} = 0$ .

The initialization statement contains a sub-statement  $S_{CBFweights}$  which computes the replica matrices defined in Eq. (2.4). Equation (2.5) shows that CBF weights are then straightforward to compute from the replica matrices. Note that weight matrices must be computed for each center frequency which will be beamformed (though in the case of CBF, this means we will compute a weight matrix for each Fourier

### Repartition

$$\begin{aligned} \left( \begin{array}{c|c} B_{proc} & B_{unproc} \end{array} \right) &\rightarrow \left( \begin{array}{c|c|c} B_{T_L} & B_{t_i} & B_{T_R} \end{array} \right) \\ \left( \begin{array}{c|c} X_{proc} & X_{unproc} \end{array} \right) &\rightarrow \left( \begin{array}{c|c|c} X_{T_L} & X_{t_i} & X_{T_R} \end{array} \right) \\ \text{where } B_{t_i} \text{ is } l \times f, & X_{t_i} \text{ is } n \times f, \text{ and } t_i = t_{proc}. \end{aligned}$$

$\{Q_{bu}\}$

$S_u$

$\{Q_{au}\}$

### Continue with

$$\begin{aligned} \left( \begin{array}{c|c} B_{proc} & B_{unproc} \end{array} \right) &\leftarrow \left( \begin{array}{c|c|c} B_{T_L} & B_{t_i} & B_{T_R} \end{array} \right) \\ \left( \begin{array}{c|c} X_{proc} & X_{unproc} \end{array} \right) &\leftarrow \left( \begin{array}{c|c|c} X_{T_L} & X_{t_i} & X_{T_R} \end{array} \right) \\ \text{where } B_{t_i} \text{ is } l \times f, & X_{t_i} \text{ is } n \times f, \text{ and } t_i = t_{proc}. \end{aligned}$$

Figure 3.2: Layout for body of CBF loop over time sequence

frequency). Refer to Sec. 3.3.3 for the derivation of this loop. For now, we will assume the following precondition and postcondition for  $S_{CBFweights}$ :

$$\begin{aligned} &\{\text{inputOK}_{CBF} \wedge \text{uninitialized}(V) \wedge \text{uninitialized}(W)\} \\ &\quad S_{CBFweights} \\ &\{\text{computed}_{CBF}(W)\}. \end{aligned}$$

The predicate  $S_{CBFweights}$  prepares the CBF weights and stores them in the matrix  $W$  for later use within the loop body.

Given the program state transition that occurs over  $S_{CBFweights}$ , and the partitioning specified above, both the outer loop precondition and loop-invariant are satisfied.

Now that a loop-invariant, loop guard, and initialization statement are known, we may derive the outer loop body. Computation should move the program toward a state in which  $G$  is false. The idea is to let  $X_{proc}$  and  $B_{proc}$  expand as the algorithm progresses until  $X = X_{proc}$ ,  $B = B_{proc}$ , implying that  $X_{unproc}$  and  $B_{unproc}$  are  $n \times f \times 0$  and  $l \times f \times 0$ , respectively. We begin deriving the loop body in Fig. (3.2) by repartitioning the matrices to outline the progress made by the outer loop core update statement  $S_u$ .

The purpose of this repartitioning is to expose individual submatrices across the time sequence dimension of  $B$  and  $X$  which have not yet been processed. Here, a single line separates a newly-exposed submatrix from its originating matrix, while the double lines more firmly separate the processed and unprocessed partitions of the full matrix in question.

Notice that the submatrices extracted from the unprocessed partitions of  $B$  and  $X$  are reassigned to their respective processed partitions after the outer loop's core update.

Let  $T_L$  denote the ordered set of indices  $t_i \in \{0, \dots, t_{proc} - 1\}$  and  $T_R$  denote the ordered set of indices  $t_i \in \{t_{proc} + 1, \dots, t - 1\}$ . After the first repartitioning,

$$\left( \begin{array}{c|c} B_{proc} = B_{T_L} & B_{unproc} = \left( \begin{array}{c|c} B_{t_{proc}} & B_{T_R} \end{array} \right) \end{array} \right) \quad (3.14)$$

$$\left( \begin{array}{c|c} X_{proc} = X_{T_L} & X_{unproc} = \left( \begin{array}{c|c} x_{t_{proc}} & X_{T_R} \end{array} \right) \end{array} \right) \quad (3.15)$$

where  $B_{T_L}$  is  $l \times f \times t_{proc}$ ,  $B_{T_R}$  is  $l \times f \times (t_{unproc} - 1)$ ,  $X_{T_L}$  is  $n \times f \times t_{proc}$ ,  $X_{T_R}$  is  $n \times f \times (t_{unproc} - 1)$ ,  $B_{t_i}$  is  $l \times f$ , and  $X_{t_i}$  is  $n \times f$ .

The predicate  $Q_{bu}$  denotes the program state after the outer loop body's first repartitioning but before the core update statement has executed. Here,  $Q_{bu}$  is derived by substituting the matrices equated in Eq. (3.14) and (3.15) into the loop-invariant  $P_{inv}$ :

$$Q_{bu} : \left( \begin{array}{c} \text{computed}_{\text{CBF}}(W) \wedge \\ B_{proc} = \left( b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in T_L \right) \wedge \\ B_{unproc} = \left( \text{uninitialized}(B_{t_{proc}}) \mid \text{uninitialized}(B_{t_i}) \right. \\ \left. \forall t_i \in T_R \right) \end{array} \right).$$

Similarly, the predicate  $Q_{au}$  corresponds to the state of the program after the core update statement  $S_u$  but before the loop body's second repartitioning,

$$( B_{proc} = ( B_{T_L} \mid B_{t_{proc}} ) \parallel B_{unproc} = B_{T_R} ) \quad (3.16)$$

$$( X_{proc} = ( X_{T_L} \mid X_{t_{proc}} ) \parallel X_{unproc} = X_{T_R} ) \quad (3.17)$$

which merges the newly-processed submatrices  $B_{t_i}$  and  $X_{t_i}$  with the other processed submatrices residing in  $B_{proc}$  and  $X_{proc}$ , respectively. So,  $Q_{au}$  should describe a state after the update of  $B$  in which  $P_{inv}$  still holds true. More simply,  $Q_{au}$  should be equal to  $P_{inv}$  except that it should also exhibit the appropriate update of the two submatrices which were just processed:

$$Q_{au} : \left( \begin{array}{c} \text{computed}_{\text{CBF}}(W) \wedge \\ B_{proc} = \left( \begin{array}{c} b_{f_j t_i} = W_{c_j}^H x_{f_j t_i} \mid b_{f_j t_{proc}} = W_{c_j}^H x_{f_j t_{proc}} \\ \forall f_j \in F, \forall t_i \in T_L \mid \forall f_j \in F \end{array} \right) \wedge \\ B_{unproc} = \left( \text{uninitialized}(B_{t_i}) \forall t_i \in T_R \right) \end{array} \right).$$

To find the core update statement  $S_u$  of the outer loop body, we observe the state of the program  $Q_{bu}$  before the outer loop update statement with the state of the program  $Q_{au}$  after the update. The interesting portions of the state transition can be reduced to

$$\left\{ Q_{bu} : \left( \text{uninitialized}(B_{t_{proc}}) \wedge \text{computed}_{\text{CBF}}(W) \right) \right\}_{S_u} \\ \left\{ Q_{au} : \left( b_{f_j t_{proc}} = W_{c_j}^H x_{f_j t_{proc}} \forall f_j \in F \right) \right\}.$$

A comparison of  $Q_{bu}$  and  $Q_{au}$  of the outer loop body with  $P_{pre}$  and  $P_{post}$  of the inner loop from Sec. 3.3.1 reveals that for  $t_i = t_{proc}$ ,  $Q_{bu}$  satisfies the inner loop precondition while  $Q_{au}$  satisfies the inner loop postcondition. From this, we infer that the core update statement for the outer loop is the entirety of the inner loop over center frequency:

$$S_u = S_{inner}.$$

After the newly-processed submatrices  $B_{t_i}$  and  $X_{t_i}$  are merged with their respective processed partitions, we refer to them as officially “processed”, and thus there exists an implicit statement after the second partitioning that updates the values of  $t_{proc}$  with  $t_{proc} + 1$  and  $t_{unproc}$  with  $t_{unproc} - 1$

### 3.3.3 Derivation of loop over center frequency bin to compute replica vectors

The portion of the CBF algorithm derived in Sec. 3.3.2 confirms that the CBF weight submatrices must be computed prior to outer loop over time sequence. In this section, we will derive the statement  $S_{CBFweights}$ . The vast majority of this derivation covers the more general problem of deriving a loop that will compute the plane wave replica vectors described in Eq. (2.3) and (2.4). From these equations, we see that the creation of the replica matrices for each center frequency  $c_j$  is performed through a matrix-matrix multiply, a scaling by the center frequency's wavenumber, and then an invocation of Euler's Identity for each matrix element to arrive at the final complex spectra:

$$V_{c_j} := \exp \left[ i \frac{2\pi \text{cfreq}(c_j)}{c_{avg}} (A_{xyz} - C_{ref})^T P_{dc} \right]$$

Note the absence of any  $t_i$  index in the replica equations, because replica vectors do not vary with time sequence. For the remainder of this section, we will refer to  $V_{c_j}$  as the  $j$ th  $n \times l$  submatrix of the full matrix  $V$ .

It is worth pointing out here that the  $n \times l$  replica submatrices generated for use in CBF are instantiated for each Fourier frequency bin  $f_j$  in the Fourier frequency band. So in Eq. (2.3) and (2.4), the center frequency  $f_{c_j} = f_j$  for all  $f_j \in F$  when CBF is performed. However, a DMR beamformer (and all other MVDR-based beamformers) must average the cross-spectra across adjacent Fourier frequency bins and time sequences, thus the DMR algorithm needs only to compute weights, and by proxy replica vectors, for each center frequency bin  $c_j$ . Here, we will use  $V_{c_j}$  to refer to the  $n \times b$  submatrix of the full replica matrix  $V$  that corresponds to center frequency  $c_j$ .

A reasonable precondition and postcondition for a loop that constructs the CBF weight matrix were given in the derivation of the CBF outer loop over time sequence in Sec. 3.3.2. When the replica computation is separated from the CBF weight normalization, we have

$$S_{CBFweights} : \left\{ \begin{array}{c} \{ \text{inputOK}_{CBF} \wedge \text{uninitialized}(V) \wedge \text{uninitialized}(W) \} \\ S_{replicas} \\ \{ \text{computed}(V) \wedge \text{uninitialized}(W) \} \\ S_{CBFnorm} \\ \{ \text{computed}_{CBF}(W) \}. \end{array} \right\}$$

where the CBF weights matrix  $W$  will be computed through the unity normalization after the loop in  $S_{replicas}$ . This breakdown of the  $S_{CBFweights}$  statement mirrors the separation of functionality of Eq. (2.3) and (2.5). For much of the remainder of this section, we will work to derive a generalized loop to create the replica matrix  $V$ .

To specify the replica loop precondition, we only need to include predicates that concern the replica computation. Thus, we will omit out those from the replica loop precondition those predicates concerning the weights matrix  $W$  and readdress the CBF weights issue at the end of this section. The replica loop precondition is

$$P_{pre} : \{ \text{inputOK}_{CBF} \wedge \text{uninitialized}(V) \}$$

while the postcondition matches the description of Eq. (2.3):

$$P_{post} : \{ \text{computed}(V) \}.$$

The predicate `computed` is true if and only if

$$V_{c_j} = \exp \left[ i \frac{2\pi \text{cfreq}(c_j)}{c_{avg}} (A_{xyz} - C_{ref})^T P_{dc} \right] \forall c_j \in C$$

where  $C$  is the set of indices  $\{0, \dots, c-1\}$ .

The replica submatrix  $V_{c_j}$  depends only on the wavenumber and the result matrix-matrix product  $(A_{xyz} - C_{ref})^T P_{dc}$  of the array sensor coordinates and the direction cosines. Once a submatrix from  $V$  is processed, it need not be accessed again for the remainder of the loop. We also assume that the submatrices of  $V$  are processed in ascending index  $c_j$ . Let us partition  $V_{c_j}$  into a matrix along the  $c$  dimension characterized by those submatrices which have been processed and those which have not yet been processed:

$$V_{c_j} \rightarrow ( V_{proc} \parallel V_{unproc} )$$

where  $V_{proc}$  is  $n \times l \times f_{proc}$ ,  $V_{unproc}$  is  $n \times l \times f_{unproc}$ , and  $c = c_{proc} + c_{unproc}$ .

Notice that since the term  $(A_{xyz} - C_{ref})^T P_{dc}$  remains static throughout the replica computation, it need not be partitioned. In fact, this matrix term may be precomputed prior to the replica loop over center frequency. Let us refer to the  $n \times l$  matrix product of this term as  $Z$ .

Substituting this partitioning into the postcondition  $P_{post}$  we find that

$$V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right] \quad \forall c_j \in \{0, \dots, c_{proc} - 1\} \quad (3.18)$$

$$V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right] \quad \forall c_j \in \{c_{proc}, \dots, c - 1\}. \quad (3.19)$$

At any given time in the inner loop, Eq. (3.18) implies that replica computation for the submatrix  $V_{c_j}$  for all  $c_j \in \{0, \dots, c_{proc} - 1\}$  has completed. Equation (3.19) describes how the replica matrix computation will eventually take place for center frequency bins  $c_j \in \{c_{proc}, \dots, c - 1\}$  that have not yet been processed. Notice that the matrix partition  $V_{unproc}$  is always uninitialized since it has not yet been updated with corresponding values of  $\exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right]$  for all  $c_j \in \{c_{proc}, \dots, c - 1\}$ .

As before, we will describe the loop-invariant in terms of computations that have already taken place, so the state described by Eq. (3.18) will serve as our loop-invariant  $P_{inv}$ :

$$P_{inv} : \left( \begin{array}{l} V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right] \quad \forall c_j \in \{0, \dots, c_{proc} - 1\} \wedge \\ \text{uninitialized}(V_{c_j}) \quad \forall c_j \in \{c_{proc}, \dots, c - 1\}. \end{array} \right) \quad (3.20)$$

where the predicate  $\text{uninitialized}(V_{c_j})$  is true if and only if all elements of the submatrix  $V_{c_j}$  have not yet been initialized.

Given the loop-invariant in (3.20), we must derive a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies the loop postcondition  $P_{post}$ :

$$(P_{inv} \wedge \neg G) \Rightarrow (P_{post}).$$

Consider the  $G$ :  $c_{proc} \neq c$ . The predicate  $P_{inv} \wedge \neg G$  implies that  $c = c_{proc}$ ,  $V = V_{proc}$ , and thus  $V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right]$  for all  $c_j \in C$ , which describes the same state described by the loop's postcondition. Thus the predicate  $G$  can serve as our loop guard in the conjunction that implies completion of the loop:

$$(P_{inv} \wedge \neg G) \Rightarrow \left( V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right] \quad \forall c_j \in C \right).$$

Next we must find what loop initialization statement  $S_I$  must execute to make  $P_{inv}$  hold true before the loop begins execution. Consider the statement  $S_I$ ,

$$Z := (A_{xyz} - C_{ref})^T P_{dc}$$

$$\text{Partition } V \rightarrow ( V_{proc} \parallel V_{unproc} )$$

**where**  $V_{proc}$  **is**  $n \times l \times 0$ ,  $c_{proc} = 0$  **and**  $c_{unproc} = c$ .

First, we precompute  $Z$  for notational conciseness, and so that unnecessary computations are not repeated. Then  $S_I$  partitions the matrix  $V$  and establishes that the processed partition as being empty, meaning no computation has occurred.

After the initialization statement  $S_I$ , all of  $V$  is uninitialized and the weight matrices for all center frequency bins are computed and reside in  $V$ , thus  $P_{inv}$  is true.

Now we derive the the replica loop body. Computation should move the program toward a state in which  $G$  is false. The idea is to let  $V_{proc}$  expand as the algorithm progresses until  $V = V_{proc}$  implying that  $V_{unproc}$  is  $n \times l \times 0$ . We begin deriving the loop body in Fig. (3.3) by repartitioning the matrices to outline the progress made by the core update statement  $S_u$ .

**Repartition**

$( V_{proc} \parallel V_{unproc} ) \rightarrow ( V_{LC} \parallel V_{c_j} \mid V_{RC} )$   
**where**  $V_{c_j}$  **is**  $n \times l \times 1$ , **and**  $c_j = c_{proc}$ .  
 $\{Q_{bu}\}$   
 $S_u$   
 $\{Q_{au}\}$

**Continue with**

$( V_{proc} \parallel V_{unproc} ) \leftarrow ( V_{LC} \mid V_{c_j} \parallel V_{RC} )$   
**where**  $V_{c_j}$  **is**  $n \times l \times 1$ , **and**  $c_j = c_{proc}$ .

Figure 3.3: Layout for body of replica loop over center frequency bin

The purpose of this repartitioning is to expose individual submatrices of  $V$  which have not yet been processed. Here, a single line separates a newly-exposed submatrix from its originating matrix, while the double lines more firmly separate the processed and unprocessed partitions of the matrix in question.

Notice that the submatrices extracted from the unprocessed partition of  $V$  is reassigned to its respective processed partition after the replica loop's core update.

Let us abbreviate the ordered set of indices  $c_j \in \{0, \dots, c_{proc} - 1\}$  as  $LC$  and the ordered set of indices  $c_j \in \{c_{proc} + 1, \dots, c - 1\}$  as  $RC$ . After the first repartitioning,

$$( V_{proc} = V_{LC} \parallel V_{unproc} = ( V_{c_{proc}} \mid V_{RC} ) ) \quad (3.21)$$

where  $V_{LC}$  is  $n \times l \times c_{proc}$ ,  $V_{RC}$  is  $n \times l \times (c_{unproc} - 1)$ , and  $V_{c_j}$  is  $n \times l \times 1$ . The predicates  $Q_{bu}$  denotes the program state after the replica loop body's first repartitioning but before the core update statement has executed. Here,  $Q_{bu}$  is derived by substituting the matrix equated in Eq. (3.21) into the loop-invariant  $P_{inv}$ :

$$Q_{bu} : \left( \begin{array}{l} B_{proc} = ( V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right] \forall c_j \in LC ) \wedge \\ B_{unproc} = \left( \text{uninitialized}(V_{c_{proc}}) \mid \text{uninitialized}(V_{c_j}) \right) \forall c_j \in RC \end{array} \right)$$

Similarly, the predicate  $Q_{au}$  corresponds to the state of the program after the core update statement  $S_u$  but before the loop body's second repartitioning,

$$( V_{proc} = ( V_{LC} \mid V_{c_{proc}} ) \parallel V_{unproc} = V_{RC} ) \quad (3.22)$$

which merges the newly-processed submatrix  $V_{c_j}$  with the other processed submatrices residing in  $V_{proc}$ . So,  $Q_{au}$  should describe a state after the update of  $V$  in which  $P_{inv}$  still holds true. More simply,  $Q_{au}$  should be equal to  $P_{inv}$  except that it should also exhibit the appropriate update of the submatrix that was just processed:

$$Q_{au} : \left( \begin{array}{l} B_{proc} = \left( \begin{array}{l} V_{c_j} = \exp \left[ i \frac{2\pi c_{freq}(c_j)}{c_{avg}} Z \right] \\ \forall c_j \in LC \end{array} \mid \begin{array}{l} V_{c_{proc}} = \\ \exp \left[ i \frac{2\pi c_{freq}(c_{proc})}{c_{avg}} Z \right] \end{array} \right) \\ \wedge B_{unproc} = ( \text{uninitialized}(V_{c_j}) \forall c_j \in RC ) \end{array} \right)$$

To find the core update statement  $S_u$  of the replica loop body, we observe the state  $Q_{bu}$  of the program before  $S_u$  with the state  $Q_{au}$  of the program after  $S_u$ . Specifically, the interesting portions of the state transition can be reduced to

$$\{ Q_{bu} : ( \text{uninitialized}(V_{c_{proc}}) ) \}$$

$$\{ Q_{au} : ( V_{c_{proc}} = \exp \left[ i \frac{2\pi c_{freq}(c_{proc})}{c_{avg}} Z \right] ) \}.$$

From this, we infer that the appropriate update for  $S_u$  for the replica loop body is

$$V_{c_{proc}} := \exp \left[ i \frac{2\pi \text{cfreq}(c_{proc})}{c_{avg}} Z \right].$$

After the newly-processed submatrix  $V_{c_{proc}}$  is merged with its respective processed partition, we refer to it as officially “processed”, and thus there exists an implicit statement after the second partitioning that updates the values of  $c_{proc}$  with  $c_{proc} + 1$  and  $c_{unproc}$  with  $c_{unproc} - 1$

Recall the breakdown of  $S_{CBFweights}$  given at the beginning of this section. The work done in this section up until now has dealt with the derivation of  $S_{replica}$ , abstracting the computation from its place in the CBF algorithm wherever possible. The remaining statement to complete the derivation of  $S_{CBFweights}$  is  $S_{CBFnorm}$  whose precondition and postcondition is

$$\begin{aligned} & \{ \text{computed}(V) \wedge \text{uninitialized}(W) \} \\ & \quad S_{CBFnorm} \\ & \{ \text{computed}_{CBF}(W) \}. \end{aligned}$$

Notice that before  $S_{CBFnorm}$ , the replica matrix  $V$  is fully computed, while the weight matrix  $W$  is still uninitialized. From Eq. (2.5), we conclude that  $S_{CBFnorm}$  is

$$W := \frac{1}{n} V.$$

### 3.3.4 Final sequential algorithm for CBF

```

Partition   $V \rightarrow ( V_{proc} \parallel V_{unproc} )$ 
  where  $V_{proc}$  is  $n \times l \times 0$ , and  $c_{proc} = 0$ 
 $Z := (A_{xyz} - C_{ref})^T P_{dc}$ 
while  $c_{proc} \neq c$  do
  Repartition
     $( V_{proc} \parallel V_{unproc} ) \rightarrow ( V_{LC} \parallel V_{c_j} \mid V_{RC} )$ 
    where  $V_{c_j}$  is  $n \times l$ , and  $c_j = c_{proc}$ 
     $V_{c_{proc}} := \exp \left[ i \frac{2\pi \text{cfreq}(c_{proc})}{c_{avg}} Z \right].$ 
  Continue with
     $( V_{proc} \parallel V_{unproc} ) \leftarrow ( V_{LC} \mid V_{c_j} \parallel V_{RC} )$ 
    where  $V_{c_j}$  is  $n \times l$ , and  $c_j = c_{proc}$ .
enddo
 $W := \frac{1}{n} V$ 
Partition   $B \rightarrow ( B_{proc} \parallel B_{unproc} )$ 
              $X \rightarrow ( X_{proc} \parallel X_{unproc} )$ 
  where  $B_{proc}$  is  $l \times f \times 0$ ,  $X_{proc}$  is  $n \times f \times 0$ , and  $t_{proc} = 0$ 
while  $t_{proc} \neq t$  do
  Repartition
     $( B_{proc} \parallel B_{unproc} ) \rightarrow ( B_{TL} \parallel B_{t_i} \mid B_{TR} )$ 
     $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{TL} \parallel X_{t_i} \mid X_{TR} )$ 
    where  $B_{t_i}$  is  $l \times f$ ,  $X_{t_i}$  is  $n \times f$ , and  $t_i = t_{proc}$ 
  Partition   $B_{t_i} \rightarrow ( B_{proc} \parallel B_{unproc} )$ 
              $X_{t_i} \rightarrow ( X_{proc} \parallel X_{unproc} )$ 
              $W \rightarrow ( W_{proc} \parallel W_{unproc} )$ 
    where  $B_{proc}$  is  $l \times 0$ ,  $X_{proc}$  is  $n \times 0$ ,  $W_{proc}$  is  $n \times l \times 0$ , and  $c_{proc} = 0$ 
  while  $f_{proc} \neq f$  do
    Repartition

```

$$\begin{aligned} & \left( \begin{array}{c} B_{proc} \parallel B_{unproc} \\ X_{proc} \parallel X_{unproc} \end{array} \right) \rightarrow \left( \begin{array}{c} B_{FL} \parallel b_{f_j t_i} \mid B_{FR} \\ X_{FL} \parallel x_{f_j t_i} \mid X_{FR} \end{array} \right) \\ \text{Repartition if } f_{proc} \bmod f_{bw} = 0 & \\ & \left( \begin{array}{c} W_{proc} \parallel W_{unproc} \\ \end{array} \right) \rightarrow \left( \begin{array}{c} W_{CL} \parallel W_{c_j} \mid W_{CR} \\ \end{array} \right) \\ \text{where } b_{f_j t_i} \text{ is } l \times 1, \quad x_{f_j t_i} \text{ is } n \times 1, \quad W_{c_j} \text{ is } n \times l, \quad f_j = f_{proc} \text{ and } c_j = c_{proc}. & \\ b_{f_j t_i} := W_{c_j}^H x_{f_j t_i} & \\ \text{Continue with} & \\ & \left( \begin{array}{c} B_{proc} \parallel B_{unproc} \\ X_{proc} \parallel X_{unproc} \end{array} \right) \leftarrow \left( \begin{array}{c} B_{FL} \mid b_{f_j t_i} \parallel B_{FR} \\ X_{FL} \mid x_{f_j t_i} \parallel X_{FR} \end{array} \right) \\ \text{Continue with if } (f_{proc} + 1) \bmod f_{bw} = 0 & \\ & \left( \begin{array}{c} W_{proc} \parallel W_{unproc} \\ \end{array} \right) \leftarrow \left( \begin{array}{c} W_{FL} \mid W_{c_j} \parallel W_{CR} \\ \end{array} \right) \\ \text{where } b_{f_j t_i} \text{ is } l \times 1, \quad x_{f_j t_i} \text{ is } n \times 1, \quad W_{c_j} \text{ is } n \times l, \quad f_j = f_{proc} \text{ and } c_j = c_{proc}. & \\ \text{enddo} & \\ \text{Continue with} & \\ & \left( \begin{array}{c} B_{proc} \parallel B_{unproc} \\ X_{proc} \parallel X_{unproc} \end{array} \right) \leftarrow \left( \begin{array}{c} B_{TL} \mid B_{t_i} \parallel B_{TR} \\ X_{TL} \mid X_{t_i} \parallel X_{TR} \end{array} \right) \\ \text{where } B_{t_i} \text{ is } l \times f, \quad X_{t_i} \text{ is } n \times f, \quad \text{and } t_i = t_{proc}. & \\ \text{enddo} & \end{aligned}$$

### 3.4 Sequential DMR

Building off of the loops derived for CBF, we will continue by deriving loops that are specific to the DMR algorithm.

For DMR,  $\hat{P}_{pre}$  and  $\hat{P}_{post}$  are

$$\begin{aligned} \hat{P}_{pre} &: \{\text{inputOK}_{\text{DMR}}\} \\ \hat{P}_{post} &: \left\{ b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in T \right\} \end{aligned}$$

where  $\text{inputOK}_{\text{DMR}}$  is true if and only if

$$\text{inputOK}_{\text{CBF}} \wedge t_{blk} \geq 1 \wedge k \geq 1 \wedge D \geq 1 \wedge \mu \geq 0$$

evaluates to true. Here,  $F$  is the set of indices  $\{0, \dots, f-1\}$ , and  $T$  is the set of indices  $\{0, \dots, t-1\}$ . We wish to derive a program,  $S_{\text{DMR}}$ , that satisfies

$$\begin{aligned} & \{\text{inputOK}_{\text{DMR}}\} \\ & S_{\text{DMR}} \\ & \{b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in T\} \end{aligned}$$

Let us begin by considering Eq. (2.1). The matrix  $W_{c_j t_i}$  is  $n \times l$  and  $x_{f_j t_i}$  is  $n \times 1$ , thus the matrix-vector multiplication  $W_{c_j t_i}^H x_{f_j t_i}$  is well-defined and produces a  $l \times 1$  vector,  $b_{f_j t_i}$  containing the beamformed spectra in each of the  $l$  look directions at each Fourier frequency bin  $f_j$  for each time sequence  $t_i$ .

Unlike CBF, DMR weights are data dependent and thus vary with time sequence, so the full matrix  $W$  will be  $n \times l \times c \times t$ .

#### 3.4.1 Derivation of inner loop over Fourier frequency bin to beamform

The derivation of the inner loop over Fourier frequency bin for the DMR algorithm will be similar to the one found in the CBF derivation, except that the DMR weights matrix will contain a dimension over time sequence since DMR weights are expected to change with time.

The core beamforming computation of the DMR algorithm for one time sequence  $t_i$  is described by

$$b_{f_j t_i} := W_{c_j t_i}^H x_{f_j t_i}$$

where center frequency bin  $c_j$  is a function of Fourier frequency bin  $f_j$ . A feasible mapping for this function is given in Eq. (2.12). This relation will remain implicit for the rest of this section to reduce notational clutter. Here, we will focus on derivation of the DMR algorithm with respect to iteration over Fourier frequency bin, and so we will treat the  $t_i$  index for  $B$  and  $X$  as a *fixed* arbitrary time sequence index without any universal quantification. Thus, any occurrence of the  $t_i$  index should be interpreted as describing the variable in question as having a time sequence dimension equal to 1. We choose to leave the index present in this inner loop derivation for the same consistency reasons that we cited when leaving them in the previous inner loop derivation for CBF in Sec. 3.3.1.

Because we will continue derivation with respect to a single time sequence  $t_i$ , we reason that the inner Fourier frequency loop's precondition is

$$P_{pre} : \{ \text{uninitialized}(B_{t_i}) \wedge \text{computed}_{\text{DMR}}(W_{t_i}) \}$$

where  $B_{t_i}$  is a reference to a single  $l \times f$  submatrix of  $B$ ,  $\text{uninitialized}(B_{t_i})$  is a predicate that is true if and only if the submatrix  $B_{t_i}$  has not yet been initialized, and  $\text{computed}_{\text{DMR}}(W_{t_i})$  is a predicate that is true if and only if the DMR weights have been computed according to Eq. (2.22). This predicate is included in the precondition to ensure that the DMR weights for all center frequency bins are ready to be applied in the core beamforming computation. The inner loop postcondition is similar to the program post condition, but without the universal quantifier over  $T$ , since only one additional time sequence will have been completed by the inner loop.

$$P_{post} : \{ b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F \},$$

Note that the beamformed spectra vector  $b_{f_j t_i}$  depends only on  $W_{c_j}$  and  $x_{f_j t_i}$  for all  $f_j \in F$ . Once a column vector from each of  $B_{t_i}$  and  $X_{t_i}$  are processed, they need not be accessed again for future computation. We will also assume that the column vectors of  $B_{t_i}$  and  $X_{t_i}$  are processed with an ascending index  $f_j$  (ie: the data vectors are processed upward in frequency). Let us partition  $B_{t_i}$ ,  $X_{t_i}$ , and  $W_{t_i}$  each into two matrices along the their respective Fourier and center frequency bin dimensions characterized by those column vectors which have been processed and those which have not yet been processed:

$$\begin{aligned} B_{t_i} &\rightarrow \left( B_{proc} \parallel B_{unproc} \right) \\ X_{t_i} &\rightarrow \left( X_{proc} \parallel X_{unproc} \right) \\ W_{t_i} &\rightarrow \left( W_{proc} \parallel W_{unproc} \right) \end{aligned}$$

where  $B_{proc}$  is  $b \times f_{proc}$ ,  $B_{unproc}$  is  $b \times f_{unproc}$ ,  $X_{proc}$  is  $n \times f_{proc}$ ,  $X_{unproc}$  is  $n \times f_{unproc}$ ,  $W_{proc}$  is  $n \times l \times c_{proc}$ ,  $W_{unproc}$  is  $n \times l \times c_{unproc}$ ,  $f = f_{proc} + f_{unproc}$ , and  $c = c_{proc} + c_{unproc}$ .

Combining these partitionings with the postcondition  $P_{post}$  we find that

$$b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in \{0, \dots, f_{proc} - 1\} \quad (3.23)$$

$$b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in \{f_{proc}, \dots, f - 1\}. \quad (3.24)$$

The first equation implies that at any given time, beamforming on the Fourier vector  $x_{f_j t_i}$  for all  $j \in \{0, \dots, f_{proc} - 1\}$  has completed, and the results of this computation reside in the corresponding beamformed data vectors  $b_{f_j t_i}$ . The second equation only describes how the unprocessed data will eventually become beamformed. Note that the matrix partition  $B_{unproc}$  is always uninitialized since it has not yet been updated with corresponding values of  $W_{c_j}^H x_{f_j t_i}$  for all  $j \in \{f_{proc}, \dots, f - 1\}$ .

Since it is more natural to describe the loop-invariant in terms of computations that have taken place rather than computations that have yet to take place, we use the state described by Eq. (3.2) in our loop-invariant  $P_{inv}$ :

$$P_{inv} : \left( \begin{array}{l} b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in \{0, \dots, f_{proc} - 1\} \wedge \\ \text{uninitialized}(b_{f_j t_i}) \forall f_j \in \{f_{proc}, \dots, f - 1\}. \end{array} \right) \quad (3.25)$$

where the predicate  $\text{uninitialized}(b_{f_j t_i})$  is true if and only if all elements of the column vector  $b_{f_j t_i}$  have not yet been initialized.

Given the loop-invariant in (3.25), we must derive a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies the loop postcondition  $P_{post}$ :

$$(P_{inv} \wedge \neg G) \Rightarrow (P_{post}).$$

Consider the  $G$ :  $f_{proc} \neq f$ . The predicate  $P_{inv} \wedge \neg G$  implies that  $f = f_{proc}$ ,  $B = B_{proc}$ , and thus  $b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i}$  for all  $j \in F$ , which describes the same state described by the loop's postcondition. Thus the predicate  $G$  can serve as our loop guard in the conjunction that implies completion of the loop:

$$(P_{inv} \wedge \neg G) \Rightarrow (b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F).$$

Next we must find what loop initialization statement  $S_I$  must execute to make  $P_{inv}$  hold true before the loop begins execution. Such a statement does not perform any computation, but rather simply performs the partitioning necessary to move the program into a state that satisfies the loop-invariant. Consider the statement  $S_I$ ,

$$\begin{array}{l} \mathbf{Partition} \quad B_{t_i} \rightarrow \left( \begin{array}{l} B_{proc} \parallel B_{unproc} \\ X_{t_i} \rightarrow \left( \begin{array}{l} X_{proc} \parallel X_{unproc} \\ W_{t_i} \rightarrow \left( \begin{array}{l} W_{proc} \parallel W_{unproc} \end{array} \right) \end{array} \right) \\ \mathbf{where} \quad B_{proc} \text{ is } l \times 0, \quad X_{proc} \text{ is } n \times 0, \quad W_{proc} \text{ is } n \times l \times 0, \\ f_{proc} = 0, \quad f_{unproc} = f, \quad c_{proc} = 0, \quad \mathbf{and} \quad c_{unproc} = c \end{array} \right) \end{array}$$

The statement initialization partitions the matrices  $B_{t_i}$ ,  $X_{t_i}$ , and  $W_{t_i}$  and establishes the processed partitions of  $B_{t_i}$  and  $X_{t_i}$  as being empty, meaning no computation has occurred.

After the initialization statement  $S_I$ , all of  $B_{t_i}$  is uninitialized and due to the precondition, the DMR weight matrix is computed and resides in  $W_{t_i}$ , thus  $P_{inv}$  is true.

Now that a loop-invariant, loop guard, and initialization statement are known, we may derive the inner loop body. Computation should move the program toward a state in which  $G$  is false. The idea is to let  $X_{proc}$ ,  $B_{proc}$ , and  $W_{proc}$  expand as the algorithm progresses until  $X_{t_i} = X_{proc}$ ,  $B_{t_i} = B_{proc}$ , and  $W_{t_i} = W_{proc}$ , implying that  $X_{unproc}$  is  $n \times 0$ ,  $B_{unproc}$  is  $l \times 0$ , and  $W_{unproc}$  is  $n \times l \times 0$ . We begin deriving the loop body in Fig. (3.1) by repartitioning the matrices to outline the progress made by the core update statement  $S_u$ .

The purpose of this repartitioning is to expose individual column vectors of  $B_{t_i}$  and  $X_{t_i}$ , and submatrices of  $W$ , which have not yet been processed. Note that like the weight matrix in the CBF algorithm, the DMR weights  $W_{t_i}$  may not need be repartitioned for every iteration of the loop. Since several Fourier frequency bins may map to the same center frequency bin, we will only expose a new submatrix of weights if the computation is moving on to process Fourier bins that associate with a new center frequency. From Eq. (2.12) we see that as Fourier frequency bin increases, the bins become associated with the next higher center frequency bin when  $f_{proc}/f_{bw} = \lfloor f_{proc}/f_{bw} \rfloor$ . In other words, since the center frequencies are separated by  $f_{bw}$ , a new center frequency's weights should be extracted when  $f_{proc} \bmod f_{bw} = 0$ . So, we denote the repartitioning of  $W_{t_i}$  as a conditional with the **Repartition if** statement. The **Continue with if** statement is based on a similar conditional. Though, here we wish to merge  $W_{c_j}$  with the other previously processed weight submatrices only when the next Fourier bin belongs to a new center frequency bin. This happens when the next iteration will need the weight submatrix computed at the next higher center frequency bin, which occurs when  $(f_{proc} + 1) \bmod f_{bw} = 0$ .

In the repartitioning notation, a single line separates a newly-exposed column vector from its originating matrix, while the double lines more firmly separate the processed and unprocessed partitions of the matrix in question.

**Repartition**

$$\begin{pmatrix} B_{proc} \parallel B_{unproc} \\ X_{proc} \parallel X_{unproc} \end{pmatrix} \rightarrow \begin{pmatrix} B_{FL} \parallel b_{f_j t_i} \mid B_{FR} \\ X_{FL} \parallel x_{f_j t_i} \mid X_{FR} \end{pmatrix}$$

**Repartition if**  $f_{proc} \bmod f_{bw} = 0$

$$\begin{pmatrix} W_{proc} \parallel W_{unproc} \end{pmatrix} \rightarrow \begin{pmatrix} W_{CL} \parallel W_{c_j} \mid W_{CR} \end{pmatrix}$$

**where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j t_i}$  **is**  $n \times l$ ,  
 $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .

$\{Q_{bu}\}$

$S_u$

$\{Q_{au}\}$

**Continue with**

$$\begin{pmatrix} B_{proc} \parallel B_{unproc} \\ X_{proc} \parallel X_{unproc} \end{pmatrix} \leftarrow \begin{pmatrix} B_{FL} \mid b_{f_j t_i} \parallel B_{FR} \\ X_{FL} \mid x_{f_j t_i} \parallel X_{FR} \end{pmatrix}$$

**Continue with if**  $(f_{proc} + 1) \bmod f_{bw} = 0$

$$\begin{pmatrix} W_{proc} \parallel W_{unproc} \end{pmatrix} \leftarrow \begin{pmatrix} W_{FL} \mid W_{c_j} \parallel W_{CR} \end{pmatrix}$$

**where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j t_i}$  **is**  $n \times l$ ,  
 $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .

Figure 3.4: Layout for body of DMR loop over Fourier frequency bin

Notice that the column vectors and submatrices extracted from the unprocessed partitions of  $B_{t_i}$ ,  $X_{t_i}$ , and  $W_{t_i}$  are reassigned to their respective processed partitions after the inner loop's core update.

Let us abbreviate the ordered set of indices  $j \in \{0, \dots, f_{proc} - 1\}$  as  $F_L$  and the ordered set of indices  $j \in \{f_{proc} + 1, \dots, f - 1\}$  as  $F_R$ . Also, let us denote the set of indices  $c \in \{0, \dots, c_{proc} - 1\}$  as  $C_L$  and  $j \in \{c_{proc} + 1, \dots, c - 1\}$  as  $C_R$ . After the loop body's first repartitioning,

$$\begin{pmatrix} B_{proc} = B_{FL} \parallel B_{unproc} = ( b_{f_{proc} t_i} \mid B_{FR} ) \end{pmatrix} \quad (3.26)$$

$$\begin{pmatrix} X_{proc} = X_{FL} \parallel X_{unproc} = ( x_{f_{proc} t_i} \mid X_{FR} ) \end{pmatrix} \quad (3.27)$$

$$\begin{pmatrix} W_{proc} = W_{CL} \parallel W_{unproc} = ( W_{c_{proc} t_i} \mid W_{CR} ) \end{pmatrix} \quad (3.28)$$

where  $B_{FL}$  is  $l \times f_{proc}$ ,  $B_{FR}$  is  $l \times (f_{unproc} - 1)$ ,  $X_{FL}$  is  $n \times f_{proc}$ ,  $X_{FR}$  is  $n \times (f_{unproc} - 1)$ ,  $W_{CL}$  is  $n \times l \times c_{proc}$ ,  $W_{CR}$  is  $n \times l \times (c_{unproc} - 1)$ ,  $b_{f_{proc} t_i}$  is  $l \times 1$ ,  $x_{f_{proc} t_i}$  is  $n \times 1$ , and  $W_{c_{proc} t_i}$  is  $n \times l$ . Note that the repartitioning of  $W$  only occurs when  $f_{proc} \bmod f_{bw} = 0$ .

The predicates  $Q_{bu}$  denotes the program state after the inner loop body's first repartitioning but before the core update statement has executed. Here,  $Q_{bu}$  is derived by substituting the matrices equated in Eq. (3.26) and (3.27) into the loop-invariant  $P_{inv}$ :

$$Q_{bu} : \left( \begin{array}{c} B_{proc} = ( b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F_L ) \wedge \\ B_{unproc} = \left( \begin{array}{c} \text{uninitialized}(b_{f_{proc} t_i}) \\ \text{uninitialized}(b_{f_j t_i}) \\ \forall f_j \in F_R \end{array} \right) \end{array} \right)$$

Similarly, the predicate  $Q_{au}$  corresponds to the state of the program after the core update statement  $S_u$  but before the loop body's second repartitioning,

$$\begin{pmatrix} B_{proc} = ( B_{FL} \mid b_{f_{proc} t_i} ) \parallel B_{unproc} = B_{FR} \end{pmatrix} \quad (3.29)$$

$$\begin{pmatrix} X_{proc} = ( X_{FL} \mid x_{f_{proc} t_i} ) \parallel X_{unproc} = X_{FR} \end{pmatrix} \quad (3.30)$$

$$\begin{pmatrix} W_{proc} = ( W_{CL} \mid W_{c_{proc} t_i} ) \parallel W_{unproc} = W_{CR} \end{pmatrix} \quad (3.31)$$

which merges the newly-processed column vectors  $b_{f_j t_i}$  and  $x_{f_j t_i}$  with the other processed column vectors residing in  $B_{proc}$  and  $X_{proc}$ , respectively. Note that the most recently extracted submatrix of  $W_{t_i}$  is only merged with the previously processed weight submatrices when  $(f_{proc} + 1) \bmod f_{bw} = 0$ .

The predicate  $Q_{au}$  should describe a state after the update of  $B_{t_i}$  in which  $P_{inv}$  still holds true. More simply,  $Q_{au}$  should be equal to  $P_{inv}$  except that it should also exhibit the appropriate update of the two column vectors which were just processed:

$$Q_{au} : \left( B_{proc} = \left( \begin{array}{c|c} b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} & b_{f_{proc} t_i} = W_{c_{proc} t_i}^H x_{f_{proc} t_i} \\ \forall f_j \in F_L & \\ \hline B_{unproc} = (\text{uninitialized}(b_{f_j t_i}) \forall f_j \in F_R) & \end{array} \right) \wedge \right).$$

To find the core update statement  $S_u$  of the loop body  $S_{CBF}$ , we observe the state  $Q_{bu}$  of the program before  $S_u$  with the state  $Q_{au}$  of the program after  $S_u$ . Specifically, the interesting portions of the state transition can be reduced to

$$\left\{ \begin{array}{c} Q_{bu} : (\text{uninitialized}(b_{f_{proc} t_i})) \\ S_u \\ Q_{au} : (b_{f_{proc} t_i} = W_{c_{proc} t_i}^H x_{f_{proc} t_i}) \end{array} \right\}.$$

From this, we infer that the appropriate update for  $S_u$  is

$$b_{f_{proc} t_i} := W_{c_{proc} t_i}^H x_{f_{proc} t_i}.$$

After the newly-processed column vectors  $b_{f_{proc}}$  and  $x_{f_{proc}}$  are merged with their respective processed partitions, we refer to them as officially “processed”, and thus there exists an implicit statement after the second partitioning that updates the values of  $f_{proc}$  with  $f_{proc} + 1$  and  $f_{unproc}$  with  $f_{unproc} - 1$ . A similar update is made to  $c_{proc}$  and  $c_{unproc}$  as part of the **Continue with if** statement.

### 3.4.2 Derivation of inner loop over center frequency bin to compute DMR weights

The previous derivation loops over Fourier frequency bin to apply the DMR weights. Now, we will derive the loop which precedes it that must compute the DMR weights submatrix for a single time sequence.

Now we will derive the loop structure by focusing on the loop’s iteration over center frequency bin  $c_j$ . Again, we treat  $t_i$  as a fixed value that references a single time sequence.

We reason that the inner loop over center frequency has the precondition

$$P_{pre} : \{ \text{uninitialized}(B_{t_i}) \wedge \text{uninitialized}(W_{t_i}) \}.$$

where  $\text{uninitialized}(B_{t_i})$  is a predicate that is true if and only if the submatrix  $B_{t_i}$  has not yet been initialized, and  $\text{uninitialized}(W_{t_i})$  is true if and only if the submatrix  $W_{t_i}$  has not yet been initialized.

The weights must be updated before they can be applied to the Fourier data, implying that the loop over center frequency bin must occur before the loop over Fourier frequency bin. Therefore, the postcondition for the loop over center frequency bin will be equal to the precondition of the loop over Fourier frequency bin from Sec. 3.3.1:

$$P_{post} : \{ \text{computed}_{\text{DMR}}(W_{c_j t_i}) \forall c_j \in C \}.$$

Note that the formation of the DMR weights does not require any use of  $B_{t_i}$ , thus partitioning it is not needed for this loop over center frequency bin. However, Eq. (2.13) shows that the columns of  $X_{t_i}$  are required to form individual CSM snapshots. So, we will only partition  $X$  and  $W$ .

$$\begin{array}{l} X_{t_i} \rightarrow \left( X_{proc} \parallel X_{unproc} \right) \\ W_{t_i} \rightarrow \left( W_{proc} \parallel W_{unproc} \right) \end{array}$$

where  $X_{proc}$  is  $n \times f_{proc}$ ,  $X_{unproc}$  is  $n \times f_{unproc}$ ,  $W_{proc}$  is  $n \times l \times c_{proc}$ ,  $W_{unproc}$  is  $n \times l \times c_{unproc}$ ,  $f = f_{proc} + f_{unproc}$ , and  $c = c_{proc} + c_{unproc}$ .

Combining the partitioning of  $W_{t_i}$  with the postcondition  $P_{post}$  we find that

$$\text{computed}_{\text{DMR}}(W_{c_j t_i}) \quad \forall c_j \in \{0, \dots, c_{proc} - 1\} \quad (3.32)$$

$$(3.33)$$

Notice that the remainder of  $W_{t_i}$ , that is,  $W_{c_j t_i}$  for all  $c_j$  in  $\{c_{proc}, \dots, c-1\}$ , is always uninitialized since it has not yet been assigned the DMR weight values. The state described by Eq. (3.32) forms the basis of our loop-invariant  $P_{inv}$ :

$$P_{inv} : ( \text{computed}_{\text{DMR}}(W_{c_j t_i}) \quad \forall c_j \in \{0, \dots, c_{proc} - 1\} ) \quad (3.34)$$

Now we must derive a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies the loop postcondition  $P_{post}$ :

$$(P_{inv} \wedge \neg G) \Rightarrow (P_{post}).$$

Consider the  $G$ :  $c_{proc} \neq c$ . The predicate  $P_{inv} \wedge \neg G$  implies that  $c = c_{proc}$ ,  $W = W_{proc}$ ,  $X = X_{proc}$ , and thus  $\text{computed}_{\text{DMR}}(W_{c_j t_i})$  for all  $c_j \in C$ , which describes the same state described by the loop's postcondition. Thus the predicate  $G$  can serve as our loop guard in the conjunction that implies completion of the loop:

$$(P_{inv} \wedge \neg G) \Rightarrow (\text{computed}_{\text{DMR}}(W_{c_j t_i}) \quad \forall c_j \in C).$$

Next we must find what loop initialization statement  $S_I$  must execute to make  $P_{inv}$  hold true before the loop begins execution. This statement performs the partitioning necessary to move the program into a state that satisfies the loop-invariant. Consider the statement  $S_I$ ,

**Partition**  $X_{t_i} \rightarrow ( X_{proc} \parallel X_{unproc} )$   
 $W_{t_i} \rightarrow ( W_{proc} \parallel W_{unproc} )$   
**where**  $X_{proc}$  **is**  $l \times 0$ ,  $W_{proc}$  **is**  $n \times 0$ ,  $f_{proc} = 0$ ,  $f_{unproc} = f$ ,  $c_{proc} = 0$ , **and**  $c_{unproc} = c$

The statement initialization partitions the matrices  $X_{t_i}$  and  $W_{t_i}$ , and establishes the processed partitions of  $X_{t_i}$  and  $W_{t_i}$  as being empty, meaning no computation has occurred.

It is worth noting that after the initialization statement  $S_I$ , all of  $W_{t_i}$  is uninitialized. And since the submatrix of beamformed data  $B_{t_i}$  is not used in this loop—which precedes the loop over Fourier bin—it is also uninitialized, making the precondition is true. With  $W_{t_i}$  partitioned and completely uninitialized, the loop-invariant  $P_{inv}$  becomes true.

Now that a loop-invariant, loop guard, and initialization statement are known, we may derive the inner loop body  $S_{DMRweights}$ . Computation should move the program toward a state in which  $G$  is false. The idea is to let  $X_{proc}$  and  $W_{proc}$  expand as the algorithm progresses until  $X_{t_i} = X_{proc}$  and  $W_{t_i} = W_{proc}$ , implying that  $X_{unproc}$  is  $n \times 0$  and  $W_{unproc}$  is  $n \times l \times 0$ . We begin deriving the loop body in Fig. (3.5) by repartitioning the matrices to outline the progress made by the core update statement  $S_u$ .

Unlike previous loop derivations, we will extract several columns of Fourier data from  $X_{t_i}$  at a time rather than just one. In Fig. 3.5, we denote this collection of Fourier column vectors as an  $n \times f_{bw}$  submatrix  $X_{c_j t_i}$  where  $c_j$  is the center frequency bin to which all the Fourier column vectors belong. The reason for this multiple extraction stems from the update of the CSM sliding block history that must happen somewhere in the body of this loop. Specifically, we must integrate  $f_{bw}$  CSM snapshots over Fourier frequency before updating the block history. Thus, we will need access to all  $f_{bw}$  Fourier column vectors associated with the current center frequency bin  $c_j$ . Mappings from a center frequency bin to its lowest and highest Fourier bins are given in Eq. (2.8) and (2.9).

The repartitioning of  $W_{t_i}$  will occur along whole  $n \times l$  submatrix boundaries just as it did for the CBF derivation.

**Repartition**

$$\begin{aligned} \left( \begin{array}{c} X_{proc} \\ W_{proc} \end{array} \parallel \begin{array}{c} X_{unproc} \\ W_{unproc} \end{array} \right) &\rightarrow \left( \begin{array}{c} X_{FL} \\ W_{CL} \end{array} \parallel \begin{array}{c} X_{c_j t_i} \\ W_{c_j t_i} \end{array} \parallel \begin{array}{c} X_{FR} \\ W_{CR} \end{array} \right) \\ \text{where } X_{c_j t_i} \text{ is } n \times f_{bw} t_{blk}, \quad W_{c_j t_i} \text{ is } n \times l, \\ f_j = f_{proc} \text{ and } c_j = c_{proc}. \end{aligned}$$

$\{Q_{bu}\}$

$S_u$

$\{Q_{au}\}$

**Continue with**

$$\begin{aligned} \left( \begin{array}{c} X_{proc} \\ W_{proc} \end{array} \parallel \begin{array}{c} X_{unproc} \\ W_{unproc} \end{array} \right) &\leftarrow \left( \begin{array}{c} X_{FL} \\ W_{CL} \end{array} \parallel \begin{array}{c} X_{c_j t_i} \\ W_{c_j t_i} \end{array} \parallel \begin{array}{c} X_{FR} \\ W_{CR} \end{array} \right) \\ \text{where } X_{c_j t_i} \text{ is } n \times f_{bw} t_{blk}, \quad W_{c_j t_i} \text{ is } n \times l, \\ f_j = f_{proc} \text{ and } c_j = c_{proc}. \end{aligned}$$

Figure 3.5: Layout for body of DMR loop over center frequency bin

Notice that the column vectors and submatrices extracted from the unprocessed partitions of  $X_{t_i}$  and  $W_{t_i}$  are reassigned to their respective processed partitions after the core of the loop body.

Let us abbreviate the ordered set of indices  $j \in \{0, \dots, f_{proc} - 1\}$  as  $F_L$  and the ordered set of indices  $j \in \{f_{proc} + 1, \dots, f - 1\}$  as  $F_R$ . Also, let us denote the set of indices  $c \in \{0, \dots, c_{proc} - 1\}$  as  $C_L$  and  $j \in \{c_{proc} + 1, \dots, c - 1\}$  as  $C_R$ . After the loop body's first repartitioning,

$$\left( \begin{array}{c} X_{proc} \\ W_{proc} \end{array} = \begin{array}{c} X_{FL} \\ W_{CL} \end{array} \parallel \begin{array}{c} X_{unproc} \\ W_{unproc} \end{array} = \begin{array}{c} X_{c_{proc} t_i} \\ W_{c_{proc} t_i} \end{array} \parallel \begin{array}{c} X_{FR} \\ W_{CR} \end{array} \right) \quad (3.35)$$

$$\left( \begin{array}{c} X_{proc} \\ W_{proc} \end{array} = \begin{array}{c} X_{FL} \\ W_{CL} \end{array} \parallel \begin{array}{c} X_{unproc} \\ W_{unproc} \end{array} = \begin{array}{c} X_{c_{proc} t_i} \\ W_{c_{proc} t_i} \end{array} \parallel \begin{array}{c} X_{FR} \\ W_{CR} \end{array} \right) \quad (3.36)$$

where  $X_{FL}$  is  $n \times c_{proc} f_{bw}$ ,  $X_{FR}$  is  $n \times (c_{unproc} - 1) f_{bw}$ ,  $W_{CL}$  is  $n \times l \times c_{proc}$ ,  $W_{CR}$  is  $n \times l \times (c_{unproc} - 1)$ ,  $X_{c_{proc} t_i}$  is  $n \times f_{bw}$ , and  $W_{c_{proc} t_i}$  is  $n \times l$ .

The predicates  $Q_{bu}$  denotes the program state after the inner loop body's first repartitioning but before the loop core has executed. Here,  $Q_{bu}$  is derived by substituting the matrices equated in Eq. (3.35) and (3.36) into the loop-invariant  $P_{inv}$ :

$$Q_{bu} : \left( \begin{array}{c} W_{proc} = (\text{computed}_{\text{DMR}}(W_{c_j t_i}) \forall c_j \in C_L) \wedge \\ W_{unproc} = \left( \begin{array}{c} \text{uninitialized}(W_{c_{proc} t_i}) \\ \text{uninitialized}(W_{c_j t_i}) \\ \forall c_j \in C_R \end{array} \right) \end{array} \right).$$

Similarly, the predicate  $Q_{au}$  corresponds to the state of the program after the core update statement  $S_u$  but before the loop body's second repartitioning,

$$\left( \begin{array}{c} X_{proc} \\ W_{proc} \end{array} = \left( \begin{array}{c} X_{FL} \\ W_{CL} \end{array} \parallel \begin{array}{c} X_{c_{proc} t_i} \\ W_{c_{proc} t_i} \end{array} \right) \parallel \begin{array}{c} X_{unproc} \\ W_{unproc} \end{array} = \begin{array}{c} X_{FR} \\ W_{CR} \end{array} \right) \quad (3.37)$$

$$\left( \begin{array}{c} X_{proc} \\ W_{proc} \end{array} = \left( \begin{array}{c} X_{FL} \\ W_{CL} \end{array} \parallel \begin{array}{c} X_{c_{proc} t_i} \\ W_{c_{proc} t_i} \end{array} \right) \parallel \begin{array}{c} X_{unproc} \\ W_{unproc} \end{array} = \begin{array}{c} X_{FR} \\ W_{CR} \end{array} \right) \quad (3.38)$$

which merges the newly-processed submatrices  $X_{c_j t_i}$  and  $W_{c_j t_i}$  with the other processed column vectors residing in  $X_{proc}$  and  $W_{proc}$  respectively.

The predicate  $Q_{au}$  should describe a state after the update of  $W_{t_i}$  in which  $P_{inv}$  still holds true. More simply,  $Q_{au}$  should be equal to  $P_{inv}$  except that it should also exhibit the appropriate update of the two column vectors which were just processed:

$$Q_{au} : \left( \begin{array}{c} B_{proc} = \left( \begin{array}{c} \text{computed}_{\text{DMR}} W_{c_j t_i} \\ \forall c_j \in C_L \end{array} \parallel \text{computed}_{\text{DMR}}(W_{c_{proc} t_i}) \right) \wedge \\ B_{unproc} = (\text{uninitialized}(W_{c_j t_i}) \forall c_j \in C_R) \end{array} \right).$$

To find the core update statement  $S_u$  of the loop body  $S_{CBF}$ , we observe the state  $Q_{bu}$  of the program before  $S_u$  with the state  $Q_{au}$  of the program after  $S_u$ . Specifically, the interesting portions of the state transition can be reduced to

$$\left\{ \begin{array}{c} Q_{bu} : (\text{uninitialized}(W_{c_{proct_i}})) \\ S_u \\ Q_{au} : (\text{computed}_{\text{DMR}}(W_{c_{proct_i}})) \end{array} \right\}.$$

From this, we can see that we must move the DMR weight submatrix  $W_{c_{proct_i}}$  from the uninitialized state to the compute state specified by Eq. (2.22). So the core update statement for the DMR loop over center frequency is actually a compound statement involving secondary requisite weight computations such as those for the white noise  $\epsilon$  and the CSM eigendecomposition.

The first step in computing the weights is the update of the CSM sliding block history. This statement is the only statement within this loop which is always executed for each iteration, regardless of the weight update parameter  $k$ . Let  $R_{c_j t_i}^{blk}$  denote the block history of  $t_{blk}$  CSM frequency estimates at center frequency bin  $c_j$  at time sequence  $t_i$ . So the updating of the CSM occurs with the following precondition and postcondition:

$$\left\{ \begin{array}{c} \text{needsUpdate}(R_{c_j t_i}^{blk}) \\ S_{CSMsblk} \\ \text{isUpdated}(R_{c_j t_i}^{blk}) \end{array} \right\}$$

where the predicate  $\text{needsUpdate}(R_{c_j}^{blk})$  is true if and only if  $R_{c_j}^{blk}$  contains the previous  $t_{blk}$  frequency estimates, excluding the current time sequence  $t_i$ . That is, frequency estimates with time sequence index  $\{t_i - t_{blk}, \dots, t_i - 1\}$ . The predicate  $\text{isUpdated}(R_{c_j}^{blk})$  is true if and only if  $R_{c_j}^{blk}$  contains the frequency estimates with time sequence index  $\{t_i - t_{blk} - 1, \dots, t_i\}$ , meaning the frequency estimate for sequence  $t_i$  has replaced the one for sequence  $t_i - t_{blk}$  in the sliding block history  $R_{c_j}^{blk}$ . Notice that when  $t_{blk} \geq 2$ , the sliding block will be only partially filled for time sequences  $\{0, \dots, t_{bw} - 1\}$ . The most common way to deal with this initialization issue is to apply CBF weights to the Fourier data for those sequences. A second solution is to initialize the CSM block history to contain identity matrices whose diagonal entries are chosen according to the amount of total power desired for the initialization. After  $t_{blk}$  time sequences, the identity CSM frequency estimates will have fallen completely out of the block history.

The CSM sliding block history may be viewed, and even implemented, as an array of  $c$  circular queue data structures, each of size  $t_{blk}$ , where the type of each element in the queue is a reference to allocated memory capable of storing an  $n \times n$  complex matrix. Let us assume that the following operations are defined on the circular queue data structure:

- **push( Q, a )** The push() function will take two arguments. Here,  $Q$  is a reference to the circular queue in question and  $a$  is a reference to the element that is to be added to the queue. This function does not return any values, but rather only changes the state of  $Q$ .
- **integrate( Q )** The integrate() function will take one argument: a reference to the circular queue to be integrated. Let us define integration as a sum of all the elements of queue, which is the function's return value.

For our purposes,  $Q$  will be a reference to the CSM sliding block history for some center frequency bin denoted  $R_{c_j}^{blk}$ . (This should not be confused with  $R_{c_j t_i}^{blk}$ , which describes the *state* of the sliding block history at a particular time sequence  $t_i$ .) Also,  $a$  will be a reference to the CSM frequency estimate at center frequency bin  $c_j$  for time sequence  $t_i$  which we will denote  $R_{c_j t_i}$ . And since addition is well-defined for complex matrices, the integrate() function works given the CSM data type of our circular queue. Thus,

$$\text{push} \left( R_{c_j}^{blk}, \sum_{f_j = \text{fbinlo}(c_j)}^{\text{fbinhi}(c_j)} x_{f_j t_i} x_{f_j t_i}^H \right)$$

where  $x_{f_j t_i} x_{f_j t_i}^H$  forms a CSM snapshot at Fourier frequency  $f_j$ . These Fourier vectors are obtained from the same submatrix of Fourier vectors that was collectively extracted from the unprocessed partitions, denoted  $X_{c_j t_i}$  earlier in this section.

The next four statements should only execute when a weight update is needed. Here, the need for this weight update is expressed by including an additional guard predicate in the preconditions and postconditions of the following four DMR-related weight computation statements. We will determine the need for a weight update with the boolean expression  $G_{Wupdate}$  given by

$$t_i \bmod k = 0$$

where  $k$  is the weight update parameter whose valid range of values are the whole numbers.

The first statement to occur when  $G_{Wupdate}$  is true will be to recompute the CSM estimates at each center frequency bin by integrating the frequency estimates over the current contents of the sliding block history.

$$\left\{ G_{Wupdate} \wedge \text{isUpdated}(R_{c_j t_i}^{blk} \text{-computed}(\bar{R}_{c_j t_i})) \right\}$$

$$S_{CSMestimate}$$

$$\left\{ G_{Wupdate} \wedge \text{isUpdated}(R_{c_j t_i}^{blk} \text{computed}(\bar{R}_{c_j t_i})) \right\}$$

where  $\text{computed}(\bar{R}_{c_j t_i})$ , is true if and only if the CSM sliding block has been integrated across time to form the CSM estimate at center frequency bin  $c_j$ , with the results stored in  $\bar{R}_{c_j t_i}$ .

Since we have already defined the  $\text{integrate}()$  function on the circular queue underlying the sliding block histories at each center frequency bin, the CSM estimate computation follows easily as,

$$\bar{R}_{c_j t_i} := \text{integrate}(R_{c_j}^{blk})$$

After the CSM block history and estimate are updated for center frequency bin  $c_j$ , the  $D$  most dominant eigenvalues and their corresponding eigenvectors must be factorized from the CSM estimate. Let us denote the CSM estimate from the sliding block for a single center frequency bin at time sequence  $t_i$  as  $\bar{R}_{c_j t_i}$ . Then, the precondition and postcondition of the statement are given by

$$\left\{ G_{Wupdate} \wedge \text{computed}(\bar{R}_{c_j t_i}) \wedge \neg \text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i}) \right\}$$

$$S_{EVD}$$

$$\left\{ G_{Wupdate} \wedge \text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i}) \right\}$$

where  $\text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i})$  is true if and only if the columns of the  $n \times D$  matrix  $U_D$  contain the eigenvectors of the CSM estimate  $\bar{R}_{c_j t_i}$  and  $\Lambda$  is a diagonal matrix of the eigenvalues of  $\bar{R}_{c_j t_i}$ . Let us denote the eigenvalue decomposition of the most dominant  $D$  eigenvalues and eigenvectors from the CSM estimate  $\bar{R}_{c_j t_i}$  as

$$(U_D, \Lambda, U_D^H) := EVD(\bar{R}_{c_j t_i})$$

With the CSM estimate formed, we may now compute the white noise  $\epsilon$  by the method given in Eq. (2.23).

$$\left\{ G_{Wupdate} \wedge \text{computed}(\bar{R}_{c_j t_i}) \right\}$$

$$S_\epsilon$$

$$\left\{ G_{Wupdate} \wedge \text{epsilonOf}(\epsilon, \bar{R}_{c_j t_i}) \right\}$$

where the predicate  $\text{epsilonOf}(\epsilon, \bar{R}_{c_j t_i})$  is true if and only if  $\epsilon$  equals the trace of  $\bar{R}_{c_j t_i}$  normalized by  $\mu/n$ . So, this statement may be simply expressed as,

$$\epsilon := \frac{\mu}{n} \text{Tr}(\bar{R}_{c_j t_i})$$

Now we may derive the final statement that assembles the DMR weights. The  $n \times 1$  DMR weights vectors for each look direction may be computed for the center frequency bin  $c_j$ , according to Eq. (2.22). The precondition and postcondition for building the DMR weights submatrix for center frequency  $c_j$  at time sequence  $t_i$  are

$$\left\{ G_{W_{update}} \wedge \neg \text{computed}_{\text{DMR}}(W_{c_j t_i}) \wedge \text{epsilonOf}(\epsilon, \bar{R}_{c_j t_i}) \wedge \text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i}) \right\} \\ S_{\text{DMRweights}} \\ \left\{ G_{W_{update}} \wedge \text{computed}_{\text{DMR}}(W_{c_j t_i}) \right\}$$

where  $\text{computed}_{\text{DMR}}(W_{c_j t_i})$  is true if and only if  $W_{c_j t_i}$  contains the DMR weights vectors for all look directions, computed at center frequency bin  $c_j$ , the predicate  $\text{epsilonOf}(\epsilon, \bar{R}_{c_j t_i})$  is true if and only if  $\epsilon$  equals the trace of  $\bar{R}_{c_j t_i}$  normalized by  $\mu/n$ . Evaluating the DMR weights equation uses the replica submatrix  $V_{c_j}$ , the white noise  $\epsilon$ , the  $D$  most dominant eigenvalues  $\Lambda_D$  and their eigenvectors  $U_D$ . Each column vector within the DMR weight submatrix  $W_{c_j t_i}$  is computed as

$$w_{l_k} := \frac{v_{l_k} - U_D U_D^H v_{l_k} + U_D \text{diag}(\frac{\epsilon}{\lambda_j + \epsilon}) U_D^H v_{l_k}}{n - \sum_{j=0}^{D-1} |u_j^H v_{l_k}|^2 + \sum_{j=0}^{D-1} \frac{\epsilon}{\lambda_j + \epsilon} |u_j^H v_{l_k}|^2} \quad \forall l_k \in \{0, \dots, l-1\}$$

where  $l_k$  denotes the index of the look direction corresponding to a the  $k$ th column of the  $n \times l$  submatrices  $W_{c_j t_i}$  and  $V_{c_j}$ ,  $\lambda_j$  is the  $j$ th most-dominant eigenvalue, and  $u_j$  is the eigenvector associated with  $\lambda_j$ .

Thus, the core update statement  $S_u$  for the DMR inner loop over center frequency bin is actually a compound statement,

$$S_u : \left\{ \begin{array}{c} \left\{ \text{needsUpdate}(R_{c_j t_i}^{blk}) \right\} \\ S_{\text{CSM}_{sblk}} \\ \left\{ \text{isUpdated}(R_{c_j t_i}^{blk}) \right\} \\ \left\{ G_{W_{update}} \wedge \text{isUpdated}(R_{c_j t_i}^{blk}) \wedge \neg \text{computed}(\bar{R}_{c_j t_i}) \right\} \\ S_{\text{CSM}_{estimate}} \\ \left\{ G_{W_{update}} \wedge \text{isUpdated}(R_{c_j t_i}^{blk}) \wedge \text{computed}(\bar{R}_{c_j t_i}) \neg \text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i}) \right\} \\ S_{\text{EVD}} \\ \left\{ G_{W_{update}} \wedge \text{computed}(\bar{R}_{c_j t_i}) \wedge \text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i}) \neg \text{epsilonOf}(\epsilon, \bar{R}_{c_j t_i}) \right\} \\ S_{\epsilon} \\ \left\{ G_{W_{update}} \wedge \text{evdOf}(U_D, \Lambda_D, \bar{R}_{c_j t_i}) \wedge \text{epsilonOf}(\epsilon, \bar{R}_{c_j t_i}) \wedge \neg \text{computed}_{\text{DMR}}(W_{c_j t_i}) \right\} \\ S_{\text{DMRweights}} \\ \left\{ G_{W_{update}} \wedge \text{computed}_{\text{DMR}}(W_{c_j t_i}) \right\} \end{array} \right\}$$

The ordering of the statements specified above is required by the preconditions and postconditions associated with each statement, except for  $S_{\text{epsilon}}$ , which may precede  $S_{\text{EVD}}$ . Neither statement depends on the other, but rather they only share a flow dependence on  $S_{\text{CSM}_{estimate}}$ . Thus, their order does not affect the final weight.

Since  $G_{W_{update}}$  is present in all statements except  $S_{\text{CSM}_{sblk}}$ , we may enclose these statements with an **if then** conditional guard in the final DMR algorithm:

```

S_CSM_sblk
if G_W_update then
  S_CSM_estimate
  S_EVD
  S_epsilon
  S_DMR_weights
endif

```

After the newly-processed submatrices  $X_{c_{proc}t_i}$  and  $W_{c_{proc}t_i}$  are merged with their respective processed partitions, we refer to them as officially “processed”, and thus there exists an implicit statement after the second partitioning that updates the values of  $c_{proc}$  with  $c_{proc} + 1$  and  $c_{unproc}$  with  $c_{unproc} - 1$ .

### 3.4.3 Derivation of outer loop over time sequence

We proceed to derive the DMR outer loop over time sequence  $t_i$ . This loop will resemble the outer loop over time sequence that was derived for CBF in Sec. 3.3.2, except that the weights matrix will be included in the discussion since the adaptive weights change over time.

Recall the core beamforming computation:

$$b_{f_j t_i} := W_{c_j t_i}^H x_{f_j t_i}.$$

Here, we will focus on the outer loop of the DMR algorithm, which will iterate over time sequence  $t_i$ . For this derivation subsection, we will denote the beamformed spectra vectors  $b_{f_j t_i}$  for all  $f_j \in F$  at a some time sequence  $t_i$  as a beamformed spectra matrix  $B_{t_i}$ . Similarly, the Fourier spectra vectors  $x_{f_j t_i}$  for all  $f_j \in F$  at some time sequence  $t_i$  will be denoted by the Fourier spectra matrix  $X_{t_i}$ .

The outer time sequence loop’s postcondition equals the program’s postcondition under the argument that all useful beamforming-related computation will cease after the time sequence loop terminates. The  $P_{pre}$  and  $P_{post}$  for the outer loop over time sequence are

$$P_{pre} : \{ \text{uninitialized}(B) \wedge \}$$

$$P_{post} = \hat{P}_{post} : \{ b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in T \},$$

where  $\text{uninitialized}(B)$  is true if and only if the submatrices  $B_{t_i}$  for all  $t_i \in T$  have not yet been initialized.

Once a corresponding submatrices from each of the full matrices  $B$ ,  $X$ , and  $W$  are processed, they need not be accessed again for future computation. As with the inner loop over Fourier frequency, we assume the submatrices are processed with ascending index  $t_i$  (ie: the data sequences are processed forward in time). Let us partition the full matrices  $B$ ,  $X$ , and  $W$  each into two matrices along the  $t$  dimension characterized by those submatrices which have been processed and those which have not yet been processed.

$$B \rightarrow ( B_{proc} \parallel B_{unproc} ), \quad X \rightarrow ( X_{proc} \parallel X_{unproc} ),$$

$$\text{and } W \rightarrow ( W_{proc} \parallel W_{unproc} )$$

where  $B_{proc}$  is  $b \times f \times t_{proc}$ ,  $B_{unproc}$  is  $b \times f \times t_{unproc}$ ,  $X_{proc}$  is  $n \times f \times t_{proc}$ ,  $X_{unproc}$  is  $n \times f \times t_{unproc}$ ,  $W_{proc}$  is  $n \times l \times c \times t_{proc}$ ,  $W_{unproc}$  is  $n \times l \times c \times t_{unproc}$ , and  $t = t_{proc} + t_{unproc}$ .

Substituting these partitionings into the outer loop postcondition  $P_{post}$  we find that

$$b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in \{0, \dots, t_{proc} - 1\} \quad (3.39)$$

$$b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in F, \forall t_i \in \{t_{proc}, \dots, t - 1\}. \quad (3.40)$$

So, at any given time in the outer loop, Eq. (3.39) implies that beamforming on the Fourier vector  $x_{f_j t_i}$  with the weights submatrix  $W_{c_j t_i}$  for all  $f_i \in F$  and all  $t_i \in \{0, \dots, t_{proc} - 1\}$  has completed with the results residing in the corresponding vectors  $b_{f_j t_i}$ . Equation (3.12) describes how beamforming will eventually take place for time sequences  $t_i \in \{t_{proc}, \dots, t - 1\}$  that have not yet been processed. Notice that at any point in the scope of the outer loop, the matrix partition  $B_{unproc}$  is uninitialized since it has not yet been updated with corresponding values of  $W_{c_j t_i}^H x_{f_j t_i}$  for all  $f_j \in F$  and  $t_i \in \{t_{proc}, \dots, t - 1\}$ .

The state described by Eq. (3.39) will serve in our outer loop-invariant  $P_{inv}$ :

$$P_{inv} : \left( \begin{array}{l} \text{computed}_{\text{DMR}}(W_{t_i}) \forall t_i \in \{0, \dots, t_{proc} - 1\} \wedge \\ b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in \{0, \dots, t_{proc} - 1\} \wedge \\ \text{uninitialized}(b_{f_j t_i}) \forall f_j \in F, \forall t_i \in \{t_{proc}, \dots, t - 1\} \end{array} \right) \quad (3.41)$$

### Repartition

$$\begin{aligned}
& \left( \begin{array}{c|c} B_{proc} & B_{unproc} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} B_{T_L} & B_{t_i} & B_{T_R} \end{array} \right) \\
& \left( \begin{array}{c|c} X_{proc} & X_{unproc} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} X_{T_L} & X_{t_i} & X_{T_R} \end{array} \right) \\
& \left( \begin{array}{c|c} W_{proc} & W_{unproc} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} W_{T_L} & W_{t_i} & W_{T_R} \end{array} \right) \\
& \text{where } B_{t_i} \text{ is } l \times f, \quad X_{t_i} \text{ is } n \times f, \quad W_{t_i} \text{ is } n \times l \times f, \text{ and } i = t_{proc}. \\
& \{Q_{bu}\} \\
& S_u \\
& \{Q_{au}\}
\end{aligned}$$

### Continue with

$$\begin{aligned}
& \left( \begin{array}{c|c} B_{proc} & B_{unproc} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} B_{T_L} & B_{t_i} & B_{T_R} \end{array} \right) \\
& \left( \begin{array}{c|c} X_{proc} & X_{unproc} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} X_{T_L} & X_{t_i} & X_{T_R} \end{array} \right) \\
& \left( \begin{array}{c|c} W_{proc} & W_{unproc} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} W_{T_L} & W_{t_i} & W_{T_R} \end{array} \right) \\
& \text{where } B_{t_i} \text{ is } l \times f, \quad X_{t_i} \text{ is } n \times f, \quad W_{t_i} \text{ is } n \times l \times f, \text{ and } i = t_{proc}.
\end{aligned}$$

Figure 3.6: Layout for body of DMR loop over time sequence

where the predicate  $\text{uninitialized}(b_{f_j t_i})$  is true if and only if all elements of the column vector  $b_{f_j t_i}$  have not yet been initialized.

Given the loop-invariant in (3.41), we must derive a loop guard  $G$  such that  $P_{inv} \wedge \neg G$  implies the loop postcondition  $P_{post}$ :

$$(P_{inv} \wedge \neg G) \Rightarrow (P_{post}).$$

Consider the  $G$ :  $t_{proc} \neq t$ . The predicate  $P_{inv} \wedge \neg G$  implies that  $t = t_{proc}$ ,  $B = B_{proc}$ , and thus  $b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i}$  for all  $f_j \in F$ ,  $t_i \in T$ , which describes the same state described by the outer loop's postcondition. Thus the predicate  $G$  can serve as the outer loop guard in the conjunction that implies completion of the loop:

$$(P_{inv} \wedge \neg G) \Rightarrow (b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \quad \forall f_j \in F, \quad \forall t_i \in T).$$

Next we must find what loop initialization statement  $S_I$  must execute to make  $P_{inv}$  hold true before the loop begins execution. This statement will only perform the partitioning necessary to move the program into a state that satisfies the loop-invariant. Consider the statement  $S_I$ ,

$$\begin{aligned}
\text{Partition } & B \rightarrow \left( \begin{array}{c|c} B_{proc} & B_{unproc} \end{array} \right) \\
& X \rightarrow \left( \begin{array}{c|c} X_{proc} & X_{unproc} \end{array} \right) \\
& W \rightarrow \left( \begin{array}{c|c} W_{proc} & W_{unproc} \end{array} \right) \\
& \text{where } B_{proc} \text{ is } l \times f \times 0, \quad X_{proc} \text{ is } n \times f \times 0, \quad W_{proc} \text{ is } n \times l \times c \times 0, \text{ and } t_{proc} = 0.
\end{aligned}$$

Given the program state transition that occurs with the partitioning specified above, both the outer loop precondition and loop-invariant are satisfied.

Now that a loop-invariant, loop guard, and initialization statement are known, we may derive the outer loop body  $S_{DMRouter}$ . Computation should move the program toward a state in which  $G$  is false. The idea is to let  $X_{proc}$ ,  $B_{proc}$ , and  $W_{proc}$  expand as the algorithm progresses until  $X = X_{proc}$ ,  $B = B_{proc}$ , and  $W = W_{proc}$ , implying that  $X_{unproc}$ ,  $B_{unproc}$ , and  $W_{unproc}$  are  $n \times f \times 0$ ,  $l \times f \times 0$ , and  $n \times l \times c \times 0$  respectively. We begin deriving the loop body in Fig. (3.6) by repartitioning the matrices to outline the progress made by the outer loop core update statement  $S_{Uouter}$ .

The purpose of this repartitioning is to expose individual submatrices across the time sequence dimension of  $B$ ,  $X$ , and  $W$  which have not yet been processed. Here, a single line separates a newly-exposed submatrix from its originating matrix, while the double lines more firmly separate the processed and unprocessed partitions of the full matrix in question.

Notice that the submatrices extracted from the unprocessed partitions of  $B$ ,  $X$ , and  $W$  are reassigned to their respective processed partitions after the outer loop's core update.

Let  $T_L$  denote the ordered set of indices  $i \in \{0, \dots, t_{proc} - 1\}$  and  $T_R$  denote the ordered set of indices  $i \in \{t_{proc} + 1, \dots, t - 1\}$ . After the first repartitioning,

$$( B_{proc} = B_{T_L} \parallel B_{unproc} = ( B_{t_{proc}} \mid B_{T_R} ) ) \quad (3.42)$$

$$( X_{proc} = X_{T_L} \parallel X_{unproc} = ( x_{t_{proc}} \mid X_{T_R} ) ) \quad (3.43)$$

$$( W_{proc} = W_{T_L} \parallel W_{unproc} = ( W_{t_{proc}} \mid W_{T_R} ) ) \quad (3.44)$$

where  $B_{T_L}$  is  $l \times f \times t_{proc}$ ,  $B_{T_R}$  is  $l \times f \times (t_{unproc} - 1)$ ,  $X_{T_L}$  is  $n \times f \times t_{proc}$ ,  $X_{T_R}$  is  $n \times f \times (t_{unproc} - 1)$ ,  $W_{T_L}$  is  $n \times l \times f \times t_{proc}$ ,  $W_{T_R}$  is  $n \times l \times f \times (t_{unproc} - 1)$ ,  $B_{t_i}$  is  $l \times f$ ,  $X_{t_i}$  is  $n \times f$ , and  $W_{t_i}$  is  $n \times l \times f$ ,

The predicate  $Q_{bu}$  denotes the program state after the outer loop body's first repartitioning but before the core update statement has executed. Here,  $Q_{bu}$  is derived by substituting the matrices equated in Eq. (3.42), (3.43), and (3.44) into the loop-invariant  $P_{inv}$ :

$$Q_{bu} : \left( \begin{array}{c} \text{computed}_{\text{DMR}}(W_{t_i}) \forall t_i \in T_L \wedge \\ B_{proc} = (b_{f_j t_i} = W_{c_j t_i} x_{f_j t_i} \forall f_j \in F, \forall t_i \in T_L) \wedge \\ B_{unproc} = \left( \begin{array}{c} \text{uninitialized}(B_{t_{proc}}) \\ \text{uninitialized}(B_{t_i}) \\ \forall t_i \in T_R \end{array} \right) \end{array} \right).$$

Similarly, the predicate  $Q_{au}$  corresponds to the state of the program after the core update statement  $S_u$  but before the loop body's second repartitioning,

$$( B_{proc} = ( B_{T_L} \mid B_{t_{proc}} ) \parallel B_{unproc} = B_{T_R} ) \quad (3.45)$$

$$( X_{proc} = ( X_{T_L} \mid X_{t_{proc}} ) \parallel X_{unproc} = X_{T_R} ) \quad (3.46)$$

$$( W_{proc} = ( W_{T_L} \mid W_{t_{proc}} ) \parallel W_{unproc} = W_{T_R} ) \quad (3.47)$$

which merges the newly-processed submatrices  $B_{t_i}$ ,  $X_{t_i}$ , and  $W_{t_i}$  with the other processed submatrices residing in  $B_{proc}$ ,  $X_{proc}$ , and  $W_{proc}$  respectively. So,  $Q_{au}$  should describe a state after the update of  $B$  in which  $P_{inv}$  still holds true. More simply,  $Q_{au}$  should be equal to  $P_{inv}$  except that it should also exhibit the appropriate update of the two submatrices which were just processed:

$$Q_{au} : \left( \begin{array}{c} \text{computed}_{\text{DMR}}(W_{t_i}) \forall t_i \in T_L \wedge \\ B_{proc} = \left( \begin{array}{c} b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \\ \forall f_j \in F, \forall t_i \in T_L \end{array} \mid \begin{array}{c} b_{f_j t_{proc}} = W_{f_j t_{proc}}^H x_{f_j t_{proc}} \\ \forall f_j \in F \end{array} \right) \wedge \\ B_{unproc} = \left( \text{uninitialized}(B_{t_i}) \forall t_i \in T_R \right) \end{array} \right).$$

To find the core update statement  $S_u$  of the outer loop body  $S_{CBF_{outer}}$ , we observe the state of the program  $Q_{bu}$  before the outer loop update statement with the state of the program  $Q_{au}$  after the update. The interesting portions of the state transition can be reduced to

$$\left\{ Q_{bu} : \left( \text{uninitialized}(B_{t_{proc}}) \wedge \text{computed}_{\text{DMR}}(W_{t_i}) \forall t_i \in T_L \right) \right. \\ \left. \begin{array}{c} S_u \\ \left\{ Q_{au} : \left( b_{f_j t_i} = W_{c_j t_i}^H x_{f_j t_i} \forall f_j \in F, \forall t_i \in \{0, \dots, t_{proc}\} \right) \right\} \end{array} \right\}.$$

A comparison of  $Q_{bu}$  and  $Q_{au}$  of the outer loop body with  $P_{pre}$  of the inner weight update loop from Sec. 3.4.2 and  $P_{post}$  of the inner beamforming loop from Sec. 3.4.1 reveals that for  $t_i = t_{proc}$ ,  $Q_{bu}$  satisfies the inner weight update loop precondition while  $Q_{au}$  satisfies the inner beamforming loop postcondition. From this, we infer that the core update statement for the outer loop is a compound statement containing the

inner loop over center frequency to update the weights if necessary and the inner loop over Fourier frequency to beamform:

$$S_u : \left\{ \begin{array}{l} S_{DMRweights} \\ S_{DMRbeamform} \end{array} \right\}$$

After the newly-processed submatrices  $B_{t_i}$  and  $X_{t_i}$  are merged with their respective processed partitions, we refer to them as officially “processed”, and thus there exists an implicit statement after the second partitioning that updates the values of  $t_{proc}$  with  $t_{proc} + 1$  and  $t_{unproc}$  with  $t_{unproc} - 1$

### 3.4.4 Derivation of loop over center frequency bin to compute replica vectors

The replica matrix needed for the DMR algorithm are equivalent to those derived for use in the CBF algorithm in Sec. 3.3.3. Note that DMR does not call for the replica vectors to be normalized, as that is part of the CBF weight computation and not including in building a general replica matrix.

### 3.4.5 Final sequential algorithm for DMR

```

Partition  $V \rightarrow ( V_{proc} \parallel V_{unproc} )$ 
  where  $V_{proc}$  is  $n \times l \times 0$ , and  $c_{proc} = 0$ 
 $Z := (A_{xyz} - C_{ref})^T P_{dc}$ 
while  $c_{proc} \neq c$  do
  Repartition
     $( V_{proc} \parallel V_{unproc} ) \rightarrow ( V_{LC} \parallel V_{c_j} \mid V_{RC} )$ 
    where  $V_{c_j}$  is  $n \times l$ , and  $c_j = c_{proc}$ 
     $V_{c_{proc}} := \exp \left[ i \frac{2\pi c_{freq}(c_{proc})}{c_{avg}} Z \right]$ 
  Continue with
     $( V_{proc} \parallel V_{unproc} ) \leftarrow ( V_{LC} \mid V_{c_j} \parallel V_{RC} )$ 
    where  $V_{c_j}$  is  $n \times l$ , and  $c_j = c_{proc}$ .
enddo
Partition  $B \rightarrow ( B_{proc} \parallel B_{unproc} )$ 
   $X \rightarrow ( X_{proc} \parallel X_{unproc} )$ 
   $W \rightarrow ( W_{proc} \parallel W_{unproc} )$ 
  where  $B_{proc}$  is  $l \times f \times 0$ ,  $X_{proc}$  is  $n \times f \times 0$ ,
   $W_{proc}$  is  $n \times l \times c \times 0$ , and  $t_{proc} = 0$ 
while  $t_{proc} \neq t$  do
  Repartition
     $( B_{proc} \parallel B_{unproc} ) \rightarrow ( B_{TL} \parallel B_{t_i} \mid B_{TR} )$ 
     $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{TL} \parallel X_{t_i} \mid X_{TR} )$ 
     $( W_{proc} \parallel W_{unproc} ) \rightarrow ( W_{TL} \parallel W_{t_i} \mid W_{TR} )$ 
    where  $B_{t_i}$  is  $l \times f$ ,  $X_{t_i}$  is  $n \times f$ ,  $W_{t_i}$  is  $n \times l \times c$ , and  $t_i = t_{proc}$ 
  Partition  $X_{t_i} \rightarrow ( X_{proc} \parallel X_{unproc} )$ 
   $W_{t_i} \rightarrow ( W_{proc} \parallel W_{unproc} )$ 
  where  $X_{proc}$  is  $n \times 0$ ,  $W_{proc}$  is  $n \times l \times 0$ , and  $c_{proc} = 0$ 
while  $c_{proc} \neq c$  do
  Repartition
     $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{FL} \parallel X_{c_j t_i} \mid X_{FR} )$ 
     $( W_{proc} \parallel W_{unproc} ) \rightarrow ( W_{CL} \parallel W_{c_j t_i} \mid W_{CR} )$ 
    where  $X_{c_j t_i}$  is  $n \times f_{bw} t_{blk}$ ,  $W_{c_j t_i}$  is  $n \times l$ ,  $f_j = f_{proc}$  and  $c_j = c_{proc}$ .
  push  $( R_{c_j}^{blk}, \sum_{f_j = f_{binlo}(c_j)}^{f_{binhi}(c_j)} x_{f_j t_i} x_{f_j t_i}^H )$ 
  if  $t_i \bmod k = 0$  then

```

```

 $\bar{R}_{c_j t_i} := \text{integrate} (R_{c_j}^{blk})$ 
 $(U_D, \Lambda, U_D^H) := \text{EVD} (\bar{R}_{c_j t_i})$ 
 $\epsilon := \frac{\mu}{n} \text{Tr} (\bar{R}_{c_j t_i})$ 
 $w_{l_k} := \frac{\text{DMR}_{\text{num}}}{\text{DMR}_{\text{den}}} \forall l_k \in \{0, \dots, l-1\}$ 
where  $\text{DMR}_{\text{num}} = v_{l_k} - U_D U_D^H v_{l_k} + U_D \text{diag}(\frac{\epsilon}{\lambda_j + \epsilon}) U_D^H v_{l_k}$  and
 $\text{DMR}_{\text{den}} = n - \sum_{j=0}^{D-1} |u_j^H v_{l_k}|^2 + \sum_{j=0}^{D-1} \frac{\epsilon}{\lambda_j + \epsilon} |u_j^H v_{l_k}|^2$ 
endif
Continue with
 $(X_{proc} \parallel X_{unproc}) \leftarrow (X_{FL} \mid X_{c_j t_i} \parallel X_{FR})$ 
 $(W_{proc} \parallel W_{unproc}) \leftarrow (W_{FL} \mid W_{c_j} \parallel W_{CR})$ 
where  $X_{c_j t_i}$  is  $n \times f_{bw} t_{blk}$ ,  $W_{c_j t_i}$  is  $n \times l$ ,
 $f_j = f_{proc}$  and  $c_j = c_{proc}$ .

enddo
Partition  $B_{t_i} \rightarrow (B_{proc} \parallel B_{unproc})$ 
 $X_{t_i} \rightarrow (X_{proc} \parallel X_{unproc})$ 
 $W_{t_i} \rightarrow (W_{proc} \parallel W_{unproc})$ 
where  $B_{proc}$  is  $l \times 0$ ,  $X_{proc}$  is  $n \times 0$ ,  $W_{proc}$  is  $n \times l \times 0$ , and  $c_{proc} = 0$ 
while  $f_{proc} \neq f$  do
Repartition
 $(B_{proc} \parallel B_{unproc}) \rightarrow (B_{FL} \parallel b_{f_j t_i} \mid B_{FR})$ 
 $(X_{proc} \parallel X_{unproc}) \rightarrow (X_{FL} \parallel x_{f_j t_i} \mid X_{FR})$ 
Repartition if  $f_{proc} \bmod f_{bw} = 0$ 
 $(W_{proc} \parallel W_{unproc}) \rightarrow (W_{CL} \parallel W_{c_j} \parallel W_{CR})$ 
where  $b_{f_j t_i}$  is  $l \times 1$ ,  $x_{f_j t_i}$  is  $n \times 1$ ,  $W_{c_j t_i}$  is  $n \times l$ ,  $f_j = f_{proc}$  and  $c_j = c_{proc}$ .
 $b_{f_j t_i} := W_{c_j}^H x_{f_j t_i}$ 
Continue with
 $(B_{proc} \parallel B_{unproc}) \leftarrow (B_{FL} \mid b_{f_j t_i} \parallel B_{FR})$ 
 $(X_{proc} \parallel X_{unproc}) \leftarrow (X_{FL} \mid x_{f_j t_i} \parallel X_{FR})$ 
Continue with if  $(f_{proc} + 1) \bmod f_{bw} = 0$ 
 $(W_{proc} \parallel W_{unproc}) \leftarrow (W_{FL} \mid W_{c_j} \parallel W_{CR})$ 
where  $b_{f_j t_i}$  is  $l \times 1$ ,  $x_{f_j t_i}$  is  $n \times 1$ , and  $f_j = f_{proc}$ .

enddo
Continue with
 $(B_{proc} \parallel B_{unproc}) \leftarrow (B_{TL} \mid B_{t_i} \parallel B_{TR})$ 
 $(X_{proc} \parallel X_{unproc}) \leftarrow (X_{TL} \mid X_{t_i} \parallel X_{TR})$ 
where  $B_{t_i}$  is  $l \times f$ ,  $X_{t_i}$  is  $n \times f$ , and  $t_i = t_{proc}$ .

enddo

```

### 3.5 Data Partitioning for Parallelization

Now that sequential algorithms have been derived, we may begin discussion of how the program might partition the data, and its associated computation, across  $N$  parallel processes.

Note that “partition” in this context refers to the dividing of the data such that all parallel processes or threads involved may share the computational workload. This concept of partitioning for parallelism is similar in theory to the matrix partitioning performed in the sequential beamformer derivations, but different in motivation.

For parallelization with MPI, we will make two mild assumptions about the processing environment present at runtime. First, we will *not* assume that every processing node has access to the filesystem onto which the program will store its final results. Secondly, we will assume that the network of processing nodes

is homogeneous. That is, each MPI process will reside on a host of similar configuration of CPU type, CPU speed, main memory, network interface, etc.

Given these two assumptions, we choose to derive our parallel algorithm in terms of a synchronous client-server model of parallelism. This model relegates file I/O and data distribution responsibilities to one process, known as the server. In addition to this overhead, the server will also participate in computation. All other processes are strictly computational in behavior. The model is synchronous with respect to the data distribution and collection. That is, the client processes will receive data only when the server is ready to send it, and send data only when the server is ready to receive it. This *blocking* communication scheme, which is automatic for many MPI library calls, is chosen to provide synchronization among the clients and server. Specifically, even though no two parallel processes will perform computation on the same data, all processes must finish their computation before the results can be written to the output stream. Similarly, a client process may not begin computation until it has received its assigned data segment from the server process.

In contrast to MPI’s message-based communication model, the POSIX threads library provides mechanisms for synchronization through the use of mutexes and condition variables [17].

Ideally, we would like each parallel process to be assigned equal amounts of work. Under this scenario, each node would never have the chance to fall idle, and thus would always be performing communication or useful computations. A straightforward way to distribute the workload would involve partitioning the data into  $N$  equally-sized segments and assigning each segment to a particular node. However, given this client-server model in which the server performs a slice of the workload as well as overhead tasks, this approach will guarantee that all client processes experience some amount of idle time for each iteration of parallel data distribution and collection. For now, we will use the naïve approach. Section 5.1 contains a discussion of balancing the server’s computation and I/O with the client’s dedicated computation.

An intelligent partitioning may be found along an axis in the data that minimizes communication. To determine the best axis over which to partition, let us consider all the dimensions in our data, and their level of attractiveness.

- **Array element.** Recall from Eq. (2.1) that beamforming is expressed as a matrix-vector multiply of an  $n \times l$  weight matrix (complex conjugated and transposed) with an  $n \times 1$  vector of Fourier spectra. This can also be described as a sequence of  $l$  dot products of  $n$ -length vectors. Each dot product operation would require the presence of the data values from all  $n$  elements. A program with this partitioning would spend more time communicating the results of the dot products than it would spend performing useful computations, thus partitioning the data into subsets of array elements is infeasible. Coarser-grain parallelism is required.
- **Look direction.** Unlike splitting the array elements, partitioning the space of look directions is feasible from a computational standpoint. However, Eq. (2.22) specifies that for a center frequency  $c_j$ , all look directions are beamformed using the eigenvalues and eigenvectors decomposed from the same CSM estimate  $R_{c_j}$ . So, partitioning across look direction would require us to either duplicate CSM estimates across processes, either by redundant computations or through interprocess communication. Neither of these options is attractive.
- **Fourier time sequence.** Since the  $t$  index is the outermost dimension of all the data involved in beamforming, partitioning the data into subsets of Fourier time sequences produces the fewest number of data partitions, and thus the fewest number of data distributions across processes. However, our algorithm derivations establish that the program is to process the data forward in time. This fact, coupled with the description of the sliding block technique in Sec. 2.3 used to integrate CSM samples over time, places a temporal dependency within the flow of processing that makes partitioning over Fourier time sequences completely infeasible.
- **Fourier frequency bin.** By partitioning a single Fourier time sequence across Fourier frequency subbands, beamforming may occur for each process on a bin-by-bin basis. However, the CSM estimates of one process may need data from Fourier frequency bins outside of the process’s subband.

- **Center frequency bin.** We elude all of the problems and dependencies other partitionings by breaking up the data across center frequency. Beamforming may still occur independently at each Fourier bin, and because the data is partitioned at center frequency boundaries, each parallel process will have local access to all the data necessary to maintain its subband’s CSM estimates forward in time without needing to communicate with any other client processes. This partitioning scheme is computationally feasible, and reduces communication to the distribution of unprocessed data and subsequent collection of processed data.

Thus, we will partition the data and computations over center frequency. Henceforth, we will often refer to a particular partition of the full set of center frequency bins as a “subband”.

For simplicity, let us define the following properties of our partitioning:

- Each partition is equal in size.
- Each partition is disjoint from all other partitions.
- Each partition is ordered according to frequency bin index.
- These properties remain constant for each parallel iteration of data distribution and collection.

The first property listed above allows us to assume that  $N$  divides evenly into  $f$ . Then the set of center frequency bins for each process will be partitioning given by

$$\begin{aligned}
 C_0 &= \left\{ 0, \dots, \frac{c}{N} - 1 \right\} \\
 C_1 &= \left\{ \frac{c}{N}, \dots, \frac{2c}{N} - 1 \right\} \\
 C_2 &= \left\{ \frac{2c}{N}, \dots, \frac{3c}{N} - 1 \right\} \\
 &\vdots \\
 C_{N-1} &= \left\{ \frac{c(N-1)}{N}, \dots, c - 1 \right\}
 \end{aligned}$$

So in general, the subset of center frequency bins for parallel process  $i$  is

$$C_i = \left\{ \frac{ic}{N}, \dots, \frac{(i+1)c}{N} - 1 \right\} \tag{3.48}$$

We refer to this straightforward scheme as *naïve* partitioning.

In addition to assigning each parallel process its own subset of center frequency bins to beamform, we must inform each process of how many center frequency bins exist within its partition. Let us define the number of center frequency bins local to a parallel process  $i$  as

$$f_{local} = f_{F_i} = \max(F_i) - \min(F_i) + 1 \tag{3.49}$$

where  $\max$  and  $\min$  are functions which return the maximum and minimum elements within the given set of frequency bins, respectively. Given the partitioning of Eq. (3.48), it is easy to show that the number of center frequency bins local to each process  $c_{local} = c/N$ .

Note that the equation for the number of center frequency bins in each subband defined above does not necessarily assume equally-sized subbands. In Sec. 5.1 we will discuss the a load-balancing scheme which calls for the server to be assigned a smaller subband than the clients.

Each parallel process must also be aware of its own unique process index, or *rank*, among the set of  $N$  processes actively participating in the computation. We will denote the total number of parallel processes

as  $N$ . We will allow a process to access its rank using the function  $\text{rank}()$ , whose range is the set of integers  $\{0, \dots, N - 1\}$ . For our purposes, let the process of rank  $N - 1$  be assigned the role of the server process. We will explain why this convention was chosen in Sec. 5.1.

Now that a suitable parallel partitioning scheme and its associated semantics are known, we will propose changes to the sequential CBF and DMR algorithms necessary to carry out the parallelization. We will revisit each loop derivations common to both CBF and DMR and address how the loops will change under our developing parallelization. Most changes will come in the form of textual substitutions, and will not compromise the validity of the algorithm’s correctness. Any changes which are not textual substitutions will be explained and justified in detail.

## 3.6 Parallel CBF and DMR

### 3.6.1 Changes to loop over center frequency bin to compute replica vectors

Due to the partitioning of the set of center frequencies into multiple subbands, each process will need to build only  $\frac{1}{N}$ th of  $V$ . That is, each process need only compute the replica submatrices which correspond to the center frequencies it was assigned by the server process. Because each process will only be responsible for  $f_{local}$  Fourier frequency bins, which are associated with the process’s  $c_{local}$  center frequencies, we begin our changes to the sequential derivations by performing a textual substitution all occurrences of  $f$  with  $f_{local}$  and  $c$  with  $c_{local}$ . These substitutions apply to all occurrences, both implicit and explicit, of  $f$  and  $c$ , including those inherent in the definitions of  $F$  and  $C$ .

The previous substitution on  $f$  suggests that both  $V$  and  $W$  are now  $n \times l \times c_{local}$  matrices. To avoid confusion, let us rename  $V$  and  $W$  on each process as  $V_{local}$  and  $W_{local}$ . The concept of the full matrices  $V$  and  $W$  survives, but now only concerns the server process.

### 3.6.2 Changes to outer loop over time sequence

Unlike the changes made to the replica loop, the scope of the outer loop over time sequence will not experience a substitution involving  $f$  or  $c$ . This substitution does not occur because the outer loop does not index over Fourier frequency, and more importantly, the data variables in the outer loop such as  $B$  and  $X$  have not yet been partitioned for parallelism. Thus, there is currently no need to redefine  $B$  and  $X$  in terms of  $f_{local}$  and  $c_{local}$  in the scope of the outer loop.

A change to the outer loop more interesting than textual substitution occurs in the form of the actual parallel partitioning, data distribution, and collection.

First, observe the **Repartition** and **Continue with** statements within the scope of the outer loop. The former extracts the submatrix  $X_{t_i}$  that corresponds to the next time sequence  $t_i$  to be processed from the other unprocessed submatrices in  $X_{T_R}$ . In an implementation, this repartitioning is analogous to a statement which reads the next time sequence of Fourier data from the input stream. Likewise, the **Continue with** statement joins the submatrix  $B_{t_i}$  of newly beamformed data with the previously processed submatrices in  $B_{T_L}$ . Symmetrically, an implementation would manifest this statement as one which writes the newly-processed time sequence of beamformed data to the output stream. However, not all processes will react to the **Repartition** and **Continue with** statements with I/O. Section 3.5 specifies that only the server process will perform the I/O functions of reading incoming Fourier data and writing outgoing beamformed data. So, when viewed as a parallel algorithm, only the server process need “perform” the **Repartition** and **Continue with** statements.

In addition, since the **Repartition** and **Continue with** statements correspond to the entry and exit points for the data, the distribution of Fourier data and collection of beamformed data must occur somewhere between these two statements. Furthermore, the distribution of the Fourier data must occur before the inner loop begins processing the individual frequencies of the submatrices corresponding to a single time sequence  $t_i$ . And the collection of the beamformed data must occur after the inner loop completes processing all frequencies. From this, we can infer that the data distribution must occur immediately before

the “Partition” statement of the inner loop while the data collection must be positioned immediately after the “enddo” of the inner loop.

Let us partition the  $n \times f$  submatrix  $X_{t_i}$  into  $N$  partitions where each  $n \times f_{local}$  partition is denoted  $X_{F_p}$ , and  $p$  is the process rank. We will mark the distribution of the Fourier data as

**Partition**  $X_{t_i} \rightarrow ( X_{F_0} \mid X_{F_1} \mid \cdots \mid X_{F_{N-1}} )$   
**For processes**  $p = 0$  **to**  $N - 1$   
     **Distribute**  $X_{F_p} \rightarrow X_{local}$  **on process**  $p$

and the collection of the beamformed data as

**For processes**  $p = 0$  **to**  $N - 1$   
     **Collect**  $B_{F_p} \leftarrow B_{local}$  **on process**  $p$   
**Merge**  $B_{t_i} \leftarrow ( B_{F_0} \mid B_{F_1} \mid \cdots \mid B_{F_{N-1}} )$

It is important to be aware that the **Partition** and **Merge** statements are exclusively executed by the server process. Also, the server performs only the “sending” side of the **Distribute** statement and the “receiving” side of the **Collect** statement. Symmetry follows as each client is only aware of the “receiving” context of the **Distribute** statement and performs only the “sending” portion of the **Collect** statement.

We insert these statements into their appropriate positions within the parallel revision of the CBF and DMR algorithms in Sec. 3.6.4 and 3.6.5, respectively.

### 3.6.3 Changes to inner loop over Fourier frequency bin to beamform

We finalize our parallelization of the CBF and DMR algorithms with changes to the inner loop over Fourier frequency bin. As with the replica matrix loop, we must first apply our textual substitution of  $f_{local}$  for  $f$  and  $c_{local}$  for  $c$ . Once again, note that this change appears to have limited effects when viewed only from the summarized algorithms in Sec. 3.6.4 and 3.6.5. However, this textual substitution also applies to other areas of the algorithm such as those matrix dimensions which are not listed in the algorithm summary.

Next, consider the original **Partition** statement from the sequential CBF algorithm,

**Partition**  $B_{t_i} \rightarrow ( B_{proc} \parallel B_{unproc} )$   
 $X_{t_i} \rightarrow ( X_{proc} \parallel X_{unproc} )$   
 $W \rightarrow ( W_{proc} \parallel W_{unproc} )$   
**where**  $B_{proc}$  **is**  $l \times 0$ ,  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$

and the equivalent statement from the sequential DMR algorithm,

**Partition**  $B_{t_i} \rightarrow ( B_{proc} \parallel B_{unproc} )$   
 $X_{t_i} \rightarrow ( X_{proc} \parallel X_{unproc} )$   
 $W_{t_i} \rightarrow ( W_{proc} \parallel W_{unproc} )$   
**where**  $B_{proc}$  **is**  $l \times 0$ ,  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$

This statement was originally written to further partition the submatrices  $B_{t_i}$  and  $X_{t_i}$  into processed and unprocessed submatrices along the Fourier frequency bin dimension. However, recall that the submatrices  $B_{t_i}$  and  $X_{t_i}$  were partitioned for parallelism and distributed before the inner loop’s opening **Partition** statement. Thus, we must change the **Partition** statement to reflect that the client processes are now operating on  $B_{local}$  and  $X_{local}$ . Since  $W$  (in the CBF algorithm) and  $W_{t_i}$  (in the DMR algorithm) never exist contiguously on one process (unless  $N = 1$ ), we will rename  $W$  and  $W_{t_i}$  in the sequential algorithm to  $W_{local}$  in the parallelized version. These changes appear in the parallel CBF and DMR algorithms as

**Partition**  $B_{local} \rightarrow ( B_{proc} \parallel B_{unproc} )$   
 $X_{local} \rightarrow ( X_{proc} \parallel X_{unproc} )$   
 $W_{local} \rightarrow ( W_{proc} \parallel W_{unproc} )$   
**where**  $B_{proc}$  **is**  $l \times 0$ ,  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$

where the submatrix  $B_{local}$  is  $l \times f_{local}$ ,  $X_{local}$  is  $n \times f_{local}$ , and  $W_{local}$  is  $n \times l \times c_{local}$ .

### 3.6.4 Final parallel algorithm for CBF

**Partition**  $V \rightarrow (V_{proc} \parallel V_{unproc})$   
**where**  $V_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$   
**while**  $c_{proc} \neq c_{local}$  **do**  
    **Repartition**  
         $(V_{proc} \parallel V_{unproc}) \rightarrow (V_{LC} \parallel V_{c_j} \mid V_{RC})$   
        **where**  $V_{c_j}$  **is**  $n \times l$ , **and**  $c_j = c_{proc}$   
         $V_{c_{proc}} := \exp \left[ i \frac{2\pi c_{freq}(c_{proc})}{c_{avg}} Z \right]$ .  
    **Continue with**  
         $(V_{proc} \parallel V_{unproc}) \leftarrow (V_{LC} \mid V_{c_j} \parallel V_{RC})$   
        **where**  $V_{c_j}$  **is**  $n \times l$ , **and**  $c_j = c_{proc}$ .  
**enddo**  
 $W := \frac{1}{n} V$   
**Partition**  $B \rightarrow (B_{proc} \parallel B_{unproc})$   
         $X \rightarrow (X_{proc} \parallel X_{unproc})$   
**where**  $B_{proc}$  **is**  $l \times f \times 0$ ,  $X_{proc}$  **is**  $n \times f \times 0$ , **and**  $t_{proc} = 0$   
**while**  $t_{proc} \neq t$  **do**  
    **Repartition**  
         $(B_{proc} \parallel B_{unproc}) \rightarrow (B_{TL} \parallel B_{t_i} \mid B_{TR})$   
         $(X_{proc} \parallel X_{unproc}) \rightarrow (X_{TL} \parallel X_{t_i} \mid X_{TR})$   
        **where**  $B_{t_i}$  **is**  $l \times f$ ,  $X_{t_i}$  **is**  $n \times f$ , **and**  $i = t_{proc}$   
    **Partition**  $X_{t_i} \rightarrow (X_{F_0} \mid X_{F_1} \mid \dots \mid X_{F_{N-1}})$   
    **For processes**  $p = 0$  **to**  $N - 1$   
        **Distribute**  $X_{F_p} \rightarrow X_{local}$  **on process**  $p$   
    **Partition**  $B_{local} \rightarrow (B_{proc} \parallel B_{unproc})$   
         $X_{local} \rightarrow (X_{proc} \parallel X_{unproc})$   
         $W_{local} \rightarrow (W_{proc} \parallel W_{unproc})$   
        **where**  $B_{proc}$  **is**  $l \times 0$ ,  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$   
    **while**  $f_{proc} \neq f_{local}$  **do**  
        **Repartition**  
             $(B_{proc} \parallel B_{unproc}) \rightarrow (B_{FL} \parallel b_{f_j t_i} \mid B_{FR})$   
             $(X_{proc} \parallel X_{unproc}) \rightarrow (X_{FL} \parallel x_{f_j t_i} \mid X_{FR})$   
        **Repartition if**  $f_{proc} \bmod f_{bw} = 0$   
             $(W_{proc} \parallel W_{unproc}) \rightarrow (W_{CL} \parallel W_{c_j} \mid W_{CR})$   
            **where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j}$  **is**  $n \times l$ ,  $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .  
             $b_{f_j t_i} := W_{f_j}^H x_{f_j t_i}$   
        **Continue with**  
             $(B_{proc} \parallel B_{unproc}) \leftarrow (B_{FL} \mid b_{f_j t_i} \parallel B_{FR})$   
             $(X_{proc} \parallel X_{unproc}) \leftarrow (X_{FL} \mid x_{f_j t_i} \parallel X_{FR})$   
        **Continue with if**  $(f_{proc} + 1) \bmod f_{bw} = 0$   
             $(W_{proc} \parallel W_{unproc}) \leftarrow (W_{FL} \mid W_{c_j} \parallel W_{CR})$   
            **where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j}$  **is**  $n \times l$ ,  $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .  
    **enddo**  
    **For processes**  $p = 0$  **to**  $N - 1$   
        **Collect**  $B_{F_p} \leftarrow B_{local}$  **on process**  $p$   
    **Merge**  $B_{t_i} \leftarrow (B_{F_0} \mid B_{F_1} \mid \dots \mid B_{F_{N-1}})$   
    **Continue with**  
         $(B_{proc} \parallel B_{unproc}) \leftarrow (B_{TL} \mid B_{t_i} \parallel B_{TR})$   
         $(X_{proc} \parallel X_{unproc}) \leftarrow (X_{TL} \mid X_{t_i} \parallel X_{TR})$   
        **where**  $B_{t_i}$  **is**  $l \times f$ ,  $X_{t_i}$  **is**  $n \times f$ , **and**  $i = t_{proc}$ .

enddo

### 3.6.5 Final parallel algorithm for DMR

**Partition**  $V \rightarrow (V_{proc} \parallel V_{unproc})$   
**where**  $V_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$   
 $Z := (A_{xyz} - C_{ref})^T P_{dc}$   
**while**  $c_{proc} \neq c$  **do**  
**Repartition**  
 $(V_{proc} \parallel V_{unproc}) \rightarrow (V_{LC} \parallel V_{c_j} \mid V_{RC})$   
**where**  $V_{c_j}$  **is**  $n \times l$ , **and**  $c_j = c_{proc}$   
 $V_{c_{proc}} := \exp\left[i \frac{2\pi c_{freq}(c_{proc})}{c_{avg}} Z\right]$ .  
**Continue with**  
 $(V_{proc} \parallel V_{unproc}) \leftarrow (V_{LC} \mid V_{c_j} \parallel V_{RC})$   
**where**  $V_{c_j}$  **is**  $n \times l$ , **and**  $c_j = c_{proc}$ .  
enddo  
**Partition**  $B \rightarrow (B_{proc} \parallel B_{unproc})$   
 $X \rightarrow (X_{proc} \parallel X_{unproc})$   
 $W \rightarrow (W_{proc} \parallel W_{unproc})$   
**where**  $B_{proc}$  **is**  $l \times f \times 0$ ,  $X_{proc}$  **is**  $n \times f \times 0$ ,  
 $W_{proc}$  **is**  $n \times l \times c \times 0$ , **and**  $t_{proc} = 0$   
**while**  $t_{proc} \neq t$  **do**  
**Repartition**  
 $(B_{proc} \parallel B_{unproc}) \rightarrow (B_{TL} \parallel B_{t_i} \mid B_{TR})$   
 $(X_{proc} \parallel X_{unproc}) \rightarrow (X_{TL} \parallel X_{t_i} \mid X_{TR})$   
 $(W_{proc} \parallel W_{unproc}) \rightarrow (W_{TL} \parallel W_{t_i} \mid W_{TR})$   
**where**  $B_{t_i}$  **is**  $l \times f$ ,  $X_{t_i}$  **is**  $n \times f$ ,  $W_{t_i}$  **is**  $n \times l \times c$ , **and**  $t_i = t_{proc}$   
**Partition**  $X_{t_i} \rightarrow (X_{F_0} \mid X_{F_1} \mid \dots \mid X_{F_{N-1}})$   
**For processes**  $p = 0$  **to**  $N - 1$   
**Distribute**  $X_{F_p} \rightarrow X_{local}$  **on process**  $p$   
**Partition**  $X_{local} \rightarrow (X_{proc} \parallel X_{unproc})$   
 $W_{local} \rightarrow (W_{proc} \parallel W_{unproc})$   
**where**  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$   
**while**  $c_{proc} \neq c_{local}$  **do**  
**Repartition**  
 $(X_{proc} \parallel X_{unproc}) \rightarrow (X_{FL} \parallel X_{c_j t_i} \mid X_{FR})$   
 $(W_{proc} \parallel W_{unproc}) \rightarrow (W_{CL} \parallel W_{c_j t_i} \mid W_{CR})$   
**where**  $X_{c_j t_i}$  **is**  $n \times f_{bw} t_{blk}$ ,  $W_{c_j t_i}$  **is**  $n \times l$ ,  $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .  
**push**  $(R_{c_j}^{blk}, \sum_{f_j = f_{binlo}(c_j)}^{f_{binhi}(c_j)} x_{f_j t_i} x_{f_j t_i}^H)$   
**if**  $t_i \bmod k = 0$  **then**  
 $\bar{R}_{c_j t_i} := \text{integrate}(R_{c_j}^{blk})$   
 $(U_D, \Lambda, U_D^H) := EVD(\bar{R}_{c_j t_i})$   
 $\epsilon := \frac{\mu}{n} \text{Tr}(\bar{R}_{c_j t_i})$   
 $w_{l_k} := \frac{\text{DMR}_{\text{num}}}{\text{DMR}_{\text{den}}} \forall l_k \in \{0, \dots, l-1\}$   
**where**  $\text{DMR}_{\text{num}} = v_{l_k} - U_D U_D^H v_{l_k} + U_D \text{diag}(\frac{\epsilon}{\lambda_j + \epsilon}) U_D^H v_{l_k}$  **and**  
 $\text{DMR}_{\text{den}} = n - \sum_{j=0}^{D-1} |u_j^H v_{l_k}|^2 + \sum_{j=0}^{D-1} \frac{\epsilon}{\lambda_j + \epsilon} |u_j^H v_{l_k}|^2$   
**endif**  
**Continue with**  
 $(X_{proc} \parallel X_{unproc}) \leftarrow (X_{FL} \mid X_{c_j t_i} \parallel X_{FR})$

$( W_{proc} \parallel W_{unproc} ) \leftarrow ( W_{FL} \mid W_{c_j} \parallel W_{CR} )$   
**where**  $X_{c_j t_i}$  **is**  $n \times f_{bw} t_{blk}$ ,  $W_{c_j t_i}$  **is**  $n \times l$ ,  
 $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .

**enddo**

**Partition**  $B_{local} \rightarrow ( B_{proc} \parallel B_{unproc} )$   
 $X_{local} \rightarrow ( X_{proc} \parallel X_{unproc} )$   
 $W_{local} \rightarrow ( W_{proc} \parallel W_{unproc} )$

**where**  $B_{proc}$  **is**  $l \times 0$ ,  $X_{proc}$  **is**  $n \times 0$ ,  $W_{proc}$  **is**  $n \times l \times 0$ , **and**  $c_{proc} = 0$

**while**  $f_{proc} \neq f_{local}$  **do**

**Repertition**

$( B_{proc} \parallel B_{unproc} ) \rightarrow ( B_{FL} \parallel b_{f_j t_i} \mid B_{FR} )$   
 $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{FL} \parallel x_{f_j t_i} \mid X_{FR} )$

**Repertition if**  $f_{proc} \bmod f_{bw} = 0$

$( W_{proc} \parallel W_{unproc} ) \rightarrow ( W_{CL} \parallel W_{c_j} \mid W_{CR} )$

**where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ ,  $W_{c_j t_i}$  **is**  $n \times l$ ,  $f_j = f_{proc}$  **and**  $c_j = c_{proc}$ .

$b_{f_j t_i} := W_{c_j}^H x_{f_j t_i}$

**Continue with**

$( B_{proc} \parallel B_{unproc} ) \leftarrow ( B_{FL} \mid b_{f_j t_i} \parallel B_{FR} )$   
 $( X_{proc} \parallel X_{unproc} ) \leftarrow ( X_{FL} \mid x_{f_j t_i} \parallel X_{FR} )$

**Continue with if**  $(f_{proc} + 1) \bmod f_{bw} = 0$

$( W_{proc} \parallel W_{unproc} ) \leftarrow ( W_{FL} \mid W_{c_j} \parallel W_{CR} )$

**where**  $b_{f_j t_i}$  **is**  $l \times 1$ ,  $x_{f_j t_i}$  **is**  $n \times 1$ , **and**  $f_j = f_{proc}$ .

**enddo**

**For processes**  $p = 0$  **to**  $N - 1$

**Collect**  $B_{F_p} \leftarrow B_{local}$  **on process**  $p$

**Merge**  $B_{t_i} \leftarrow ( B_{F_0} \mid B_{F_1} \mid \cdots \mid B_{F_{N-1}} )$

**Continue with**

$( B_{proc} \parallel B_{unproc} ) \leftarrow ( B_{TL} \mid B_{t_i} \parallel B_{TR} )$   
 $( X_{proc} \parallel X_{unproc} ) \leftarrow ( X_{TL} \mid X_{t_i} \parallel X_{TR} )$

**where**  $B_{t_i}$  **is**  $l \times f$ ,  $X_{t_i}$  **is**  $n \times f$ , **and**  $t_i = t_{proc}$ .

**enddo**

## Chapter 4

# Algorithm Analysis

### 4.1 Expected Sequential Performance

In analyzing the overall performance of the sequential beamformers, let us consider the complexity of the each major portion of the sequential CBF and DMR algorithms derived in Sec. 3.3 and 3.4. To simplify the analysis, we only consider memory references on the whole – integer and floating point arithmetic is ignored.

First let us consider the complexity of creating the replica vectors. The majority of the computation lies with the dot product of the direction cosines and the sensor coordinates, which may be cast as a matrix-matrix multiply when the operation for all sensor coordinates and look directions is considered whole. The sensor coordinates are transposed to form an  $n \times 3$  matrix while the direction cosines  $P_{dc}$  form a  $3 \times l$  matrix. The  $n \times l$  product of these two matrices is instantiated for each center frequency  $c_j$ , thus the computation of the entire matrix is of complexity  $3cln$ :

$$O_{\text{replica}} = O(3cln).$$

The complexity of the statement that reads a block of Fourier time sequences may depend on system-specific factors. Let us define  $t_{\text{read}}$  as the number of Fourier time sequences read and distributed to all processes during any iteration of the outer loop, and  $n_{\text{iter}}$  as the number of iterations the beamformer will run. We have been assuming that the number of time sequences of Fourier data read during any iteration is one, meaning that the number of iterations is  $t$ . A discussion in Sec. 5.1.6 suggests that it may be desirable to read in a block of time sequences for each iteration, rather than only one. Regardless, the relationship  $t_{\text{read}}n_{\text{iter}} = t$  holds. And if we assume that the input Fourier data contains only those frequencies we wish to beamform, and spectral data for only those sensors we wish to use in beamforming, then the read statement should take the complexity of order  $fnt$ .

$$O_{\text{read}} = O(fnt_{\text{read}}n_{\text{iter}}) = O(fnt)$$

Next, for each CSM frequency bin to be integrated upon, we must build the CSM estimate. Recall from Eq. (2.13) that a single CSM is formed through an outer product of complex vectors, which in this case is  $O(n^2)$ . However, CSMs become useful only when integrated over frequency or time, preferably both.

Given  $f$  Fourier frequency bins and a CSM bandwidth of  $f_{bw}$ , the processing band will contain  $c$  center frequency bins.<sup>1</sup> For each center frequency bin, a new frequency estimate will overwrite the oldest resident within the block history, and then all  $t_{blk}$  frequency estimates within the sliding block be integrated (over time) to arrive at the final CSM estimate.

The CSM estimates, and more generally the weights, are updated according to an integral weight update parameter  $k$  which specifies how many sequences to wait before updating the weights matrix  $W$ . So with  $t$  Fourier sequences to process, and updating weights every  $k$  sequences, gives  $t/k$  weight updates.

---

<sup>1</sup> $c = \lfloor f/f_{bw} \rfloor$  when center frequency bins are non-overlapping as described in Eq. (2.7).

Thus, the cost of updating the CSM estimates for each center frequency bin for all discrete time sequences is

$$O_{CSM} = O(n^2 f_{bw} ct + n^2 t_{blk} ct) = O(cn^2 t(f_{bw} + t_{blk}))$$

The eigendecomposition is of  $O(n^3)$  and well-known. The CSM estimate for each center frequency is decomposed for each weight update.

$$O_{EVD} = O\left(cn^3 \frac{t}{k}\right)$$

By Eq. (2.23), the white noise  $\epsilon$  is computed as some normalized fraction of the trace of the CSM estimate. This value must be recomputed for each center frequency bin every time the CSM estimate changes (ie: for each  $t/k$  weight updates). Compared to other portions of the beamforming algorithm, the white noise  $\epsilon$  computation is relatively cheap:

$$O_\epsilon = O\left(cn \frac{t}{k}\right)$$

Computation of the adaptive DMR weights of Eq. (2.22) for a single look direction  $l_k$  requires several matrix-multiplies and vector operations.

Consider the numerator first. The product  $U_D^H v$  is computed (of complexity  $nD$ ) and stored for future use, which results in a vector of length  $D$ . Subsequent multiplication from the left by the diagonal  $\left(\frac{\epsilon}{\lambda_i + \epsilon}\right)$  matrix can be thought of as a vector dot-product, thus it is of complexity  $D$ . Then, the  $U_D$  term is multiplied from the left (with complexity  $nD$ ) to form a vector of length  $n$ . Computing  $U_D U_D^H v$  is easier now that  $U_D^H v$  has already been computed. Thus, evaluation of the term  $U_D U_D^H v$  is of complexity  $nD$ . The numerator is finalized with one vector subtraction and one vector addition of complexity  $n$  each.

The denominator computation begins by computing the two summation terms, each of which involves summing  $D$  dot-products of pairs of vectors of length  $n$ , so is computed with  $nD$  complexity. Both terms sum over the same indices, so they may be computed simultaneously, halving the complexity from  $2nD$  to  $nD$ . The scalar subtraction and addition together contribute constant complexity 2.

Therefore, the complexity of DMR weight update computations for all  $l$  look directions and  $c$  center frequency bins and all  $t/k$  weight updates takes the form of

$$\begin{aligned} O_{weights-comp} &= O_{weights-numerator} + O_{weights-denominator} \\ &= O\left(cl \frac{t}{k} (3nD + D + 2n)\right) + O\left(cl \frac{t}{k} (nD + 2)\right) \\ &= O\left(cl \frac{t}{k} (4nD + 2n + D + 2)\right). \end{aligned}$$

Applying the weights for each Fourier frequency bin is simply a matrix-vector multiply of  $W_{c_j t_i}^H$  and  $x_{f_j t_i}$  for all  $f$  Fourier frequency bins at all  $t$  time sequences. Given that  $W_{c_j t_i}^H$  is  $l \times n$  and  $x_{f_j t_i}$  is  $n \times 1$ , the computation of the final beamformed spectra for all  $f$  Fourier bins and  $t$  times sequences is

$$O_{weights-apply} = O(flnt).$$

The beamformed data is sent through the output stream in the final step of the main loop. Since  $t_{read} = t_{write}$ , and the product of either with  $n_{iter}$  is still  $t$ , the complexity of writing the beamformed data is

$$O_{write} = O(flt_{write} n_{iter}) = O(flt).$$

In the following summary of CBF complexity, we do not consider the complexity of computing the CBF weights because they are of the same complexity of creating replica vectors. In fact, the only reason to compute replica vectors for conventional beamforming is to build the weights once. CBF replicas would go unused after the initial weights have been computed.

The collective complexity of the conventional beamformer components is

$$\begin{aligned}
O_{CBF} &= O_{\text{replica}} + O_{\text{read}} + O_{\text{weights-apply}} + O_{\text{write}} \\
&= O(3cln) + O(fnt) + O(flnt) + O(flt) \\
&= O(3cln + fnt + flnt + flt) \\
&= O(3cln + ft(l + ln + n))
\end{aligned} \tag{4.1}$$

The complexity of the DMR beamformer shares all of the complexities of its more fundamental CBF brethren with the added algorithmic steps of maintaining and decomposing CSM estimates, finding suitable instances of the white noise  $\epsilon$ , and updating the adaptive weights.

$$\begin{aligned}
O_{DMR} &= O_{\text{replica}} + O_{\text{read}} + O_{CSM} + O_{EVD} + O_{\epsilon} + \\
&\quad O_{\text{weights-comp}} + O_{\text{weights-apply}} + O_{\text{write}} \\
&= O(3cln) + O(fnt) + O(cn^2t(f_{bw} + t_{blk})) + O(cn^3\frac{t}{k}) + O(cn\frac{t}{k}) + \\
&\quad O(cl\frac{t}{k}(4nD + 2n + D + 2)) + O(flnt) + O(flt) \\
&= O(3cln + fnt + cn^2t(f_{bw} + t_{blk})) + cn^3\frac{t}{k} + cn\frac{t}{k} + \\
&\quad cl\frac{t}{k}(4nD + 2n + D + 2) + flnt + flt) \\
&= O(3cln + t(fn + cn^2(f_{bw} + t_{blk})) + \frac{1}{k}(cn^3 + cn + \\
&\quad cl(4nD + 2n + D + 2)) + fln + fl)) \\
&= O(3cln + t(f(l + ln + n) + cn^2(f_{bw} + t_{blk})) + \frac{1}{k}(cn^3 + cn + \\
&\quad cl(4nD + 2n + D + 2)))) \\
&= O(3cln + t(f(l + ln + n) + c(n^2(f_{bw} + t_{blk})) + \frac{1}{k}(n^3 + n + \\
&\quad l(4nD + 2n + D + 2))))
\end{aligned} \tag{4.2}$$

## 4.2 Expected Parallel Performance

Analysis of the parallel CBF and DMR beamformers will use their sequential analyses as a starting point. This section also considers the cost of interprocess communication and synchronization. Once again, this analysis only considers memory references and ignores integer and floating point computations.

In light of the naïve partitioning chosen in Sec. 3.5, most adjustments to the complexities of the sequential algorithms will come in the form of a substitution that converts the computational instances of  $c$  to  $c_{\text{local}}$ , which can be simplified to  $c/N$  under the naïve partitioning. Note that portions of the complexity analysis corresponding to non-computational tasks such as I/O are unaffected under the naïve partitioning.

By the changes made in Sec. 3.6.1, the replica computation can be done entirely in parallel. Assuming that  $c_{\text{local}} = c/N$ , the complexity of building the replica matrices under either CBF or DMR algorithms is

$$O_{\text{replica}} = \left( \frac{3cln}{N} \right).$$

In Sec. 3.3.2, we established that only the server process will perform all I/O. Accordingly, this cost does not change with  $N$  since the same amount of Fourier data is read by only the server regardless of the number of parallel processes present. So the cost of reading a Fourier sequence is

$$O_{read} = O(fnt_{read}n_{iter}) = O(fnt).$$

The cost of maintaining the CSM estimates, performing the eigendecomposition, finding the white noise  $\epsilon$ , and evaluating the DMR weights equation are all directly proportional to the number of center frequency bins to beamform. Since each node's naïvely partitioned subband of center frequency bins is equal and disjoint, the complexity of the entire weight update process drops by a factor of  $N$ :

$$O_{CSM} = O\left(cn^2 \frac{t(f_{bw} + t_{blk})}{N}\right),$$

$$O_{EVD} = O\left(cn^3 \frac{t}{kN}\right),$$

$$O_{\epsilon} = O\left(cn \frac{t}{kN}\right),$$

$$O_{weights-comp} = O\left(bc \frac{t}{kN}(4nD + 2n + D + 2)\right).$$

The cost of applying the complex weights to the Fourier data depends indirectly upon the number of center frequencies beamformed. The algorithms derived of the data partitioning in Sec. 3.5 specify that each process is assigned  $f_{local}$  Fourier frequency bins. Our naïve partitioning establishes that  $f_{local} = f/N$ . So the complexity of any computational occurrence of  $f$  is reduced by  $N$ .

$$O_{weights-apply} = O\left(\frac{flt}{N}\right)$$

Once again, our relegation of all I/O to the server process implies that the server bear all complexity involved with the writing of beamformed data to the output stream. Accordingly, its complexity is unaffected by the number of parallel processes  $N$ .

$$O_{write} = O(flt).$$

Finally, we must consider the complexity of the communication and synchronization that will occur in the parallel algorithms.

Referring back to the parallel algorithms summarized in Sec. 3.6.4 and 3.6.5, we see that communication is invoked in two separate portions of the outer loop over time sequence. The first instance of communication is represented by the **Distribute** statement. As part of this statement, the server sends a block of partitioned data of size  $n \times f_{local} \times t_{read}$  to each of the  $N$  parallel processes, including itself. This cost is incurred for each of the  $n_{iter}$  iterations of the outer loop. (Note we have been assuming that only one time sequence is read and distributed for each iteration, meaning  $t_{read} = 1$  and  $n_{iter} = t$ .) Since  $Nf_{local} = f$  under the naïve partitioning and  $t_{read}n_{iter} = t$ , the complexity of the **Distribute** statement is

$$O_{distribute} = O(nf_{local}t_{read}n_{iter}N) = O(fnt)$$

The second instance of communication corresponds to the **Collect** statement. The cost associated with this statement is similar in nature to that of the **Distribute** statement. Except here, the data blocks

being returned to the server process are  $l \times f_{local} \times t_{read}$ . This cost is incurred for all  $n_{iter}$  iterations of data distribution and collection. The complexity of **Collect** statement is

$$O_{collect} = O(lf_{local}t_{read}n_{iter}N) = O(flt).$$

The sum of these two expressions gives the complexity of all communication incurred by the parallel beamformers, using a naïve partitioning of data.

$$\begin{aligned} O_{comm} &= O_{distribute} + O_{collect} \\ &= O(fnt) + O(flt) \\ &= O(fnt + flt) \\ &= O(ft(l + n)) \end{aligned}$$

Communication during data distribution and collection is incurred at both the clients and the server. Also, the cost of communication is equal for both CBF and DMR parallel algorithms, and the same is true of the cost of I/O.

Let us digress momentarily to explain that the theoretical complexities of both forms of communication lean on the assumption that the cost of partitioning, distributing, and collecting a fixed amount of total data is equal, regardless of the size of the partitions, and thus the number of clients receiving a partition. Out of context, this assumption is quite liberal. A fine-grained accounting of communication costs suggests that all communication consists of a “startup” time, and a “transfer” time. The startup time represents the overhead of initiating the interprocess connection, and is generally fixed for any instance of communication. The transfer time, however, scales proportionally to the number of units of data being transferred. So, as  $N$  increases the total amount of communication startup costs increases, thus increasing the total amount of communication costs (startup plus transfer). If  $N$  grows large enough, the cost of initiating communication will overshadow the cost of actually transferring data, which does not grow with  $N$  since the aggregate amount of data being transferred is unchanged. Despite this underlying variability in communication, we stand by our initial assumption by reasoning that the amounts of data being distributed (or collected) is large compared to the sum of the costs of initiating the communication with each parallel process.

The total complexity of the parallel CBF algorithm is

$$\begin{aligned} O_{CBF} &= O_{replica} + O_{read} + O_{weights-apply} + O_{write} + O_{comm} \\ &= O\left(\frac{3cln}{N}\right) + O(nft) + O\left(\frac{nflt}{N}\right) + O(flt) + O(ft(n+l)) \\ &= O\left(\frac{3cln}{N} + nft + \frac{fnt}{N} + flt + ft(n+l)\right) \\ &= O\left(\frac{3cln}{N} + ft\left(n + \frac{ln}{N} + l + (n+l)\right)\right) \\ &= O\left(\frac{3cln}{N} + ft\left(n\left(2 + \frac{l}{N}\right) + 2l\right)\right) \\ &= O\left(\frac{3cln}{N} + ft\left(n\left(l\left(2 + \frac{1}{N}\right) + 2\right)\right)\right) \end{aligned} \tag{4.3}$$

As in the sequential analysis, the complexity of the parallel DMR algorithm contains additional terms to represent all computations that contribute to the update of the complex weight matrices. The total complexity of the parallel DMR algorithm is

$$\begin{aligned}
O_{DMR} &= O_{replica} + O_{read} + O_{CSM} + O_{EVD} + \\
&\quad O_{\epsilon} + O_{weights-comp} + O_{weights-apply} + \\
&\quad O_{write} + O_{comm} \\
&= O\left(\frac{3cln}{N}\right) + O(fnt) + O\left(cn^2\frac{(f_{bw} + t_{blk})}{N}\right) + O\left(cn^3\frac{t}{kN}\right) + \\
&\quad O\left(cn\frac{t}{kN}\right) + O\left(lc\frac{t}{kN}(4nD + 2n + D + 2)\right) + O\left(\frac{flnt}{N}\right) + \\
&\quad O(flt) + O(ft(n+l)) \\
&= O\left(\frac{3cln}{N} + fnt + cn^2\frac{(f_{bw} + t_{blk})}{N} + cn^3\frac{t}{kN} + \right. \\
&\quad \left. cn\frac{t}{kN} + lc\frac{t}{kN}(4nD + 2n + D + 2) + \frac{flnt}{N} + \right. \\
&\quad \left. flt + ft(n+l)\right) \\
&= O\left(\frac{3cln}{N} + t\left(fl + fn + f(l+n) + cn^2\frac{(f_{bw} + t_{blk})}{N} + \right. \right. \\
&\quad \left. \left. cn^3\frac{1}{kN} + cn\frac{1}{kN} + lc\frac{1}{kN}(4nD + 2n + D + 2) + \frac{fln}{N}\right)\right) \\
&= O\left(\frac{3cln}{N} + t\left(fl + fn + f(l+n) + \frac{1}{N}\left(cn^2(f_{bw} + t_{blk}) + \right. \right. \right. \\
&\quad \left. \left. \left. cn^3\frac{1}{k} + cn\frac{1}{k} + lc\frac{1}{k}(4nD + 2n + D + 2) + fln\right)\right)\right) \\
&= O\left(\frac{3cln}{N} + t\left(2f(l+n) + \frac{1}{N}\left(flnt + cn^2(f_{bw} + t_{blk}) + \right. \right. \right. \\
&\quad \left. \left. \left. \frac{c}{k}\left(n^3 + n + l(4nD + 2n + D + 2)\right)\right)\right)\right) \tag{4.4}
\end{aligned}$$

### 4.2.1 Speedup

Amdahl's Law places a theoretical upper bound on the potential speedup of a program [16]. The maximum attainable speedup  $S^{max}$  due to parallelism as a function of number of processors  $N$  is given by

$$S^{max}(N) = \frac{1}{p/N + s} \tag{4.5}$$

where  $p$  is the fraction of the program which can be parallelized and  $s$  is the remaining fraction which must remain sequential.

Actual relative speedup  $S^{act}$  attained by a parallel program as a function of number of processors  $N$  can be measured by

$$S^{act}(N) = \frac{T(1)}{T(N)} \tag{4.6}$$

where  $T(1)$  is the runtime of the parallel program with one processor and  $T(N)$  is the runtime of the same program with  $N$  processors.

In order to find  $S^{max}$  for CBF and DMR, we must first find the fractions of the algorithms which can ( $p$ ) and cannot ( $s$ ) be parallelized. To facilitate this, let us first make a simplifying assumption that all parameters that relate to data dimensions have the same effect on the complexity of the algorithms, that is,

$c = f = l = n = t = D$ . Also, let  $k = 1$  and  $f_{bw} = t_{blk} = 1$ . Under these assumption, the complexities of the computational (ie: parallelizable) portions of the sequential CBF and DMR algorithms are reduced to

$$\begin{aligned}
O_{CBF-comp} &= O_{replica} + O_{weights-apply} \\
&= O(3n^3) + O(n^4) \\
&= O(3n^3 + n^4) \\
&\approx O(n^4) \\
O_{DMR-comp} &= O_{replica} + O_{CSM} + O_{EVD} + O_{epsilon} + O_{weights-comp} + O_{weights-apply} \\
&= O(3n^3) + O(n^4) + O(n^5) + O(n^3) + O(4n^5 + 2n^4 + n^3 + 2n^2) \\
&= O(3n^3 + n^4 + n^5 + n^3 + 4n^5 + 2n^4 + n^3 + 2n^2) \\
&= O(5n^5 + 3n^4 + 2n^3 + 2n^2) \\
&\approx O(n^5)
\end{aligned}$$

For both CBF and DMR, the complexity of the non-parallelizable overhead, consisting of I/O and communication, is

$$\begin{aligned}
O_{overhead} &= O_{read} + O_{write} + O_{comm} \\
&= O(n^3) + O(n^3) + O(2n^3) \\
&= O(n^3 + n^3 + 2n^3) \\
&= O(4n^3) \\
&\approx O(n^3)
\end{aligned}$$

The fraction of either sequential algorithm that can be parallelized can be represented as the computational term divided by the sum of the computational and overhead terms, while the fraction that cannot be parallelized is represented as the overhead term divided by the sum of the computational and overhead terms. So for CBF,  $p$  and  $s$  can be written

$$p_{CBF} = \frac{n^4}{n^4 + n^3} \quad (4.7)$$

$$s_{CBF} = \frac{n^3}{n^4 + n^3} \quad (4.8)$$

The corresponding computational and overhead fractions of the DMR algorithm are

$$p_{DMR} = \frac{n^5}{n^5 + n^3} \quad (4.9)$$

$$s_{DMR} = \frac{n^3}{n^5 + n^3} \quad (4.10)$$

Substituting the CBF fractions expressions into Eq. (4.6), we have

$$\begin{aligned}
S_{CBF}^{max}(N) &= \left( \frac{p_{CBF}}{N} + s_{CBF} \right)^{-1} \\
&= \left( \frac{n^4}{N(n^4 + n^3)} + \frac{n^3}{n^4 + n^3} \right)^{-1}
\end{aligned}$$

$$\begin{aligned}
&= \left( \frac{n^4}{N(n^4 + n^3)} + \frac{Nn^3}{N(n^4 + n^3)} \right)^{-1} \\
&= \left( \frac{n^4 + Nn^3}{N(n^4 + n^3)} \right)^{-1} \\
&= \frac{N(n^4 + n^3)}{n^4 + Nn^3} \\
&= \frac{N + N/n}{1 + N/n}
\end{aligned} \tag{4.11}$$

and for DMR, the maximum speedup is

$$\begin{aligned}
S_{DMR}^{max}(N) &= \left( \frac{p_{DMR}}{N} + s_{DMR} \right)^{-1} \\
&= \left( \frac{n^5}{N(n^5 + n^3)} + \frac{n^3}{n^5 + n^3} \right)^{-1} \\
&= \left( \frac{n^5}{N(n^5 + n^3)} + \frac{Nn^3}{N(n^5 + n^3)} \right)^{-1} \\
&= \left( \frac{n^5 + Nn^3}{N(n^5 + n^3)} \right)^{-1} \\
&= \frac{N(n^5 + n^3)}{n^5 + Nn^3} \\
&= \frac{N + N/n^2}{1 + N/n^2}
\end{aligned} \tag{4.12}$$

## 4.2.2 Efficiency

In the context of parallel computing, *efficiency* reflects how well a parallelized program can utilize parallel processes to reduce total runtime. Specifically, relative efficiency is the ratio of speedup to the number of processors. By using the maximum speedup described above, we arrive at the maximum possible efficiency  $E_{max}(N)$  as a function of the number of processors  $N$ .

$$E_{max}(N) = \frac{S_{max}(N)}{N} \tag{4.13}$$

where  $0 < E_{max}(N) < 1$ . An efficiency of 1 implies linear speedup and is usually considered unattainable in real parallel systems because it implies the presence of negligible communication and synchronization, and also perfect load-balancing.

Like speedup, actual relative efficiency may be measured through empirical means by using  $S^{act}(N)$  instead of  $S^{max}(N)$ .

$$E^{act}(N) = \frac{S^{act}(N)}{N} = \frac{T(1)}{T(N)N} \tag{4.14}$$

Using the maximum speedup for parallel CBF computed in Eq. (4.11), the upper bound for efficiency can be expressed as

$$\begin{aligned}
E_{CBF}^{max}(N) &= \frac{1}{N} \left( \frac{N + N/n}{1 + N/n} \right) \\
&= \frac{N + N/n}{N + N^2/n}
\end{aligned}$$

$$= \frac{1 + 1/n}{1 + N/n} \quad (4.15)$$

and performing the same substitution with Eq. (4.12) gives an upper bound for parallel DMR efficiency of

$$\begin{aligned} E_{DMR}^{max}(N) &= \frac{1}{N} \left( \frac{N + N/n^2}{1 + N/n^2} \right) \\ &= \frac{N + N/n^2}{N + N^2/n^2} \\ &= \frac{1 + 1/n^2}{1 + N/n^2} \end{aligned} \quad (4.16)$$

### 4.2.3 Isoefficiency

Browne describes *isoefficiency* as the “relationship between the amount of computational work to be accomplished and the number of processors” such that efficiency remains constant [7]. To determine the isoefficiency function  $K$  for the parallel CBF algorithm, let us hold efficiency, as defined in Eq. (4.13), constant:

$$E_{CBF} = \text{constant} = \frac{n^4}{N(n^4/N + n^3)} = \frac{n^4}{n^4 + Nn^3}$$

From this expression, we can observe the impact of  $N$  on the fraction as a whole. For parallel CBF  $N$  must scale proportionally to  $n$  to maintain constant efficiency, so the isoefficiency function  $K_{CBF}(n) = n$ .

Holding the efficiency of parallel DMR constant, we find that

$$E_{DMR} = \text{constant} = \frac{n^5}{N(n^5/N + n^3)} = \frac{n^5}{n^5 + Nn^3}$$

and so for parallel DMR,  $N$  must scale proportionally to  $n^2$  to maintain constant efficiency. Thus, the isoefficiency function is  $K_{DMR}(n) = n^2$ .

These isoefficiencies suggest that it will be easier to maintain a constant efficiency as  $N$  increases when performing DMR than it will be with CBF. This can be traced back to the overall lower-order complexity of the CBF algorithm. With fewer computations to perform,  $n$  must increase at least proportionally to  $N$  in order to avoid loss of efficiency with CBF. However, the higher-order computations inherent in the DMR algorithm allows that  $n$  need only increase proportionally to the square root of  $N$  to maintain a fixed level of efficiency.

# Chapter 5

## Optimizations

The parallel CBF and DMR algorithms were derived in a somewhat straightforward manner to maintain clarity and simplicity in the derivations. Now that the base parallel algorithms exist, we will discuss a range of possible optimizations. Some of these improvements address the algorithm when run in parallel while others affect both sequential and parallel instantiations. Furthermore, these optimizations fall into three categories depending on the level at which the optimization is applied: algorithm, source, and compiler. All optimizations listed here are optional and except for a few floating point compiler optimizations, these techniques do not affect the final beamformer output; rather, most attempt to improve performance or reduce the memory requirements of the algorithms' implementations.

### 5.1 Algorithm-level

Algorithmic optimizations describe ways to streamline the basic parallel algorithms derived in Sec. 3.6.4 and 3.6.5. These improvements come about from the desire to keep all parallel processes busy with useful computation or file I/O as often as possible.

#### 5.1.1 Load-balancing

The issue of *load-balancing* addresses the need to ensure that each process continuously participates in progressing through the total workload.

Consider an iteration over the outer time sequence loop of either beamformer other than the first iteration. Currently, the beamforming algorithms call for the server process to repartition, or read, the next time sequence of Fourier data only after the beamformed data from all nodes has been collected, merged, and output. Also, the parallel client processes are sent Fourier data from the server, and return beamformed data to the server, with increasing rank, such that process 0 is serviced first, then process 1, process 2, etc so that the server always serves itself last. Assuming that the host operating system schedules the processes with equally high priority, all client processes (processes 0 through  $N - 2$ ) will idle during two distinct intervals of time. First, the clients will idle with no new data to beamform as they wait for the server to finish beamforming its own subband of data. Then the server process finally finishes its beamforming and begins collecting the beamformed data from each client, but after a client has submitted its completed beamformed data, it must idle once more as the server writes the previous sequence of beamformed data and then reads the next sequence of Fourier data. We anticipate that this idling of client processes will severely degrade the efficiency of the parallel CBF and DMR algorithms. But, it seems we may ameliorate the problem given the following two steps.

1. First, we will change the parallel algorithms such that the server process is ready to refresh the client process with new Fourier data immediately after it has collected the client's previously-processed beamformed data. To do this, the server process must have read the new sequence of Fourier data prior

to collecting the previous iteration’s beamformed data. This change breaks some of the algorithm’s symmetry, and thus requires additional logic to initialize the procedure and ensure proper termination of the loop when the end of the data is reached. However, this modification transforms the two separate periods of client idling described above into one only one period. Compare Fig. 5.1 and 5.2 for illustrations on these changes in data distribution. Unfortunately, there is still no guarantee that the server will have finished (or even started) reading the next time sequence of Fourier data before process 0 is ready to submit the beamformed data from the previous data distribution and receive new work. In fact, if it is assumed that all processes progress through their equal subbands at an equal rate, the client processes will always begin idling before the server finishes processing its own subband partition.

2. The algorithms derived up to this point have assumed equal distribution of the Fourier data. This equal distribution of data simplifies many aspects of the algorithm, but has the side effect of leaving the server process with more total work to perform due to the server’s unique role as the gateway for incoming Fourier data and outgoing beamformed data. We attempt to remedy this situation by reducing the amount of computational work assigned to the server. In this case, the client processes will collectively share the difference in work between the server’s naïve subband assignment and its reduced subband. And while the client processes spend a bit more time processing the data that the server otherwise would have been responsible for, the potential exists for the server to finish reading the next Fourier sequence just as the first client process finishes beamforming the previous iteration’s data. In this ideal scenario, none of the processes would incur significant idling, and efficiency would be virtually maximized.

So how might this load-balancing be implemented? To maximize performance among homogeneous processing nodes, the amount of *total* work assigned to each node, in units of CPU time, must be equal. The total cost  $T$  of a sequential instantiation of the program is

$$T = T_{bf} + T_{io}$$

where  $T_{bf}$  is the total cost of useful beamforming and related computations, and  $T_{io}$  is the total amount of file I/O that takes place during the processing. The naïve partitioning scheme splits  $T_{bf}$  across all processes,

$$\begin{aligned} T_0 &= \frac{T_{bf}}{N} \\ T_1 &= \frac{T_{bf}}{N} \\ &\vdots \\ T_{N-1} &= \frac{T_{bf}}{N} + T_{comm} + T_{io} \end{aligned}$$

which includes  $T_{comm}$ , the added runtime cost of communication between server and clients. This situation is less than ideal when  $T_{comm}$  and  $T_{io}$  are non-negligible relative to  $\frac{T_{bf}}{N}$ . To balance the workload on each process, we introduce two adjustment factors:  $\alpha$  and  $\beta$ . This adjusted distribution of work can be viewed as

$$\begin{aligned} T_0 &= \frac{T_{bf}}{N} + \alpha \\ T_1 &= \frac{T_{bf}}{N} + \alpha \\ &\vdots \\ T_{N-1} &= \frac{T_{bf}}{N} + T_{comm} + T_{io} - \beta \end{aligned}$$

Since no new work is being introduced,

$$\beta = \alpha(N - 1) \quad (5.1)$$

must hold true. Also, we would prefer to find the ideal case where all parallel processes are assigned equal work such that

$$T_{N-1} = T_p \quad \forall p \in \{0, \dots, N - 2\}.$$

This equality assumes that each process will spend equal amounts of time beamforming given equal amounts of data to beamform. In other words, each process will progress through the computational portions of the program at the same rate.

Equating any of the expressions for the client processes' work with the server's work we have

$$\begin{aligned} \frac{T_{bf}}{N} + \alpha &= \frac{T_{bf}}{N} + T_{comm} + T_{io} - \beta \\ \alpha &= T_{comm} + T_{io} - \beta \end{aligned} \quad (5.2)$$

To solve for  $\beta$ , we use Eq. (5.1) and (5.2) to form the system of equations,

$$\begin{cases} \alpha = T_{comm} + T_{io} - \beta \\ \beta = \alpha(N - 1) \end{cases}$$

Substituting Eq. (5.2) into Eq. (5.1), we can solve for  $\beta$ ,

$$\begin{aligned} \beta &= (T_{comm} + T_{io} - \beta)(N - 1) \\ \beta &= (T_{comm} + T_{io})(N - 1) - \beta(N - 1) \\ \beta + \beta(N - 1) &= (T_{comm} + T_{io})(N - 1) \\ \beta N &= (T_{comm} + T_{io})(N - 1) \\ \beta &= \frac{(N - 1)(T_{comm} + T_{io})}{N} \end{aligned} \quad (5.3)$$

and then solve for  $\alpha$  by substituting the resulting expression into Eq. (5.1),

$$\begin{aligned} \alpha(N - 1) &= \frac{(N - 1)(T_{comm} + T_{io})}{N} \\ \alpha &= \frac{(N - 1)(T_{comm} + T_{io})}{(N - 1)N} \\ \alpha &= \frac{T_{comm} + T_{io}}{N} \end{aligned} \quad (5.4)$$

Equation (5.3) expresses how much *less* time the server should spend beamforming in terms of the total cost of communication, the total cost of file I/O, and the number of parallel processes being employed, where the amount less work is relative to a naïve partitioning. Similarly, Eq. (5.4) expresses how much *more* time the clients should spend beamforming in order to minimize the amount of idle time incurred waiting for the server to catch up to the point in the algorithm when data collection and distribution occurs.

Given  $\beta$ , one may compute the ideal number of fewer center frequency bins to be assigned to the server's subband by

$$\beta_c = \left\lceil \frac{\beta c}{T_{bf}} \right\rceil. \quad (5.5)$$

where  $\beta_c$  is the number of fewer center frequency bins to be processed by the server. As a result, the subbands are drawn out such that each of the  $N - 1$  clients will process  $\alpha_c$  additional center frequency bins, whose value is approximately

$$\alpha_c \approx \frac{c - \beta_c}{N - 1}. \quad (5.6)$$

Note that  $\alpha_c$  may not necessarily take on the same value for each client process since the fraction in Eq. (5.6) may not evaluate to a whole number. In this case, the remaining center frequency bins may be allocated to a subset of the client processes according to some heuristic, such as giving an extra center frequency bin to the lower-ranked clients.

Combining Eq. (5.5) and (5.3), we have

$$\beta_c = \left\lceil \frac{(N - 1)(T_{comm} + T_{io})c}{NT_{bf}} \right\rceil.$$

However,  $T_{bf}$ ,  $T_{comm}$ , and  $T_{io}$  are not known precisely until after a sequential instantiation of the beamforming program has completed. Though, given a modest amount of empirical data, these values may be approximated with reasonable accuracy.

From Sec. 4.1, we should see that

$$\begin{aligned} O_{comm} &= O_{distribute} + O_{collect} \\ &= O(fnt) + O(flt) \\ &= O(ft(l + n)) \\ O_{io} &= O_{read} + O_{write} \\ &= O(fnt) + O(flt) \\ &= O(ft(l + n)) \end{aligned}$$

and for sequential CBF,  $O_{bf}$  is

$$O_{bf} = O(3cln + flnt)$$

while sequential DMR implemented with a Fourier data matrix and SVD has an  $O_{bf}$  term equal to

$$O_{bf} = O(3cln + t(fln + c(nf_{bw}t_{blk} + \frac{1}{k}((f_{bw}t_{blk})^3 + n + l(4nD + 2n + D + 2))))))$$

Assuming our complexity analysis is reasonable, we can separate out the machine-specific portions of  $T_{bf}$ ,  $T_{comm}$ , and  $T_{io}$  using the results of the complexity of each term. Let us refer to the computational coefficient as  $\kappa$ , the communications coefficient as  $\delta$ , and the file I/O coefficient as  $\gamma$ , such that

$$T_{bf} \approx \kappa O_{bf} \quad (5.7)$$

$$T_{comm} \approx \delta O_{comm} \quad (5.8)$$

$$T_{io} \approx \gamma O_{io} \quad (5.9)$$

Note that these coefficients have units of time. An estimate for  $\beta_c$  may be obtained by approximating  $T_{bf}$ ,  $T_{comm}$ , and  $T_{io}$  using the the complexity analysis of each term and their respective coefficients, which may be obtained empirically for each computing platform.

In the parallel CBF and DMR implementations,  $\beta_c$  appears as an runtime input parameter. If one is given (including  $\beta_c = 0$ ), the beamforming program will use this static value. If no  $\beta_c$  is provided, then the program will attempt to estimate it using the results of the complexity analysis combined with user-provided values for  $\kappa$ ,  $\delta$ , and  $\gamma$ .

```

Partition  $B \rightarrow ( B_{proc} \parallel B_{unproc} )$ 
            $X \rightarrow ( X_{proc} \parallel X_{unproc} )$ 
while  $t_{proc} \neq t$  do
  Repartition
     $( B_{proc} \parallel B_{unproc} ) \rightarrow ( B_{TL} \parallel B_{t_i} \parallel B_{TR} )$ 
     $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{TL} \parallel X_{t_i} \parallel X_{TR} )$ 
  Partition  $X_{t_i} \rightarrow ( X_{F_0} \mid X_{F_1} \mid \dots \mid X_{F_{N-1}} )$ 
  For processes  $p = 0$  to  $N - 1$ 
    Distribute  $X_{F_p} \rightarrow X_{local}$  on process  $p$ 
     $\vdots$ 
  For processes  $p = 0$  to  $N - 1$ 
    Collect  $B_{F_p} \leftarrow B_{local}$  on process  $p$ 
  Merge  $B_{t_i} \leftarrow ( B_{F_0} \mid B_{F_1} \mid \dots \mid B_{F_{N-1}} )$ 
  Continue with
     $( B_{proc} \parallel B_{unproc} ) \leftarrow ( B_{TL} \parallel B_{t_i} \parallel B_{TR} )$ 
     $( X_{proc} \parallel X_{unproc} ) \leftarrow ( X_{TL} \parallel X_{t_i} \parallel X_{TR} )$ 
enddo

```

Figure 5.1: Original layout of data distribution and collection

### 5.1.2 Implementations of complex matrix-matrix multiply

The equation for DMR weights vectors (Eq. (2.22)) contains a matrix-vector multiplication between the complex conjugate transposed eigenvectors  $U_D^H$  and the replica vector for a single look direction and center frequency. This operation may be cast into a matrix-matrix multiply between  $U_D^H$  and the replica vectors  $V_{c_j}$  for all look directions at a single center frequency. The desire to condense computations into sets of matrix-matrix multiplications comes from previous research in this field that has yielded algorithms and implementations that deliver very good performance [9] [14]. Specifically, the ITXGEMM matrix-matrix kernel and the Goto BLAS implementation take care in moving matrix data through the cache hierarchy and avoiding Translation Look-aside Buffer (TLB) misses, respectively, to attain near-peak performance. However, these implementations are currently only available for Intel hardware. Since our computing environment uses UltraSPARC hardware, we will use need to use a more generic BLAS implementation.

Here, we provide benchmarks for the single precision complex matrix-matrix multiply routine `cgemm` included in three different implementations of the BLAS:

- the BLAS as implemented in the Sun performance library,
- the BLAS compiled from the source distribution provided by [1], and
- the BLAS as optimized by the Automatically Tuned Linear Algebra Software (ATLAS) [20].

The benchmarks were performed on matrices of varying dimensions, with the dimensions biased somewhat toward those that might be found in the matrix multiplication of  $U_D^H$  and  $V$ . Each benchmark is the sample average of 450 invocations of the routine, where each invocation uses a different portion of the test data. Figure 5.3 contains the results of these benchmarks.

Note that ATLAS is built upon the BLAS. In this case, we configured ATLAS to use the compiled BLAS implementation, not the BLAS supplied by the Sun performance library.

As expected, these benchmarks show that for the matrix sizes tested, the ATLAS implementation of the `cgemm` routine outperforms the compiled BLAS. But surprisingly, `cgemm` compiled from scratch with the Sun Fortran-77 compiler consistently outperforms the `cgemm` routine included in Sun performance library. Based on this data, we chose to use the complex matrix-matrix multiply routine as implemented by ATLAS compiled from source.

```

Partition  $B \rightarrow ( B_{proc} \parallel B_{unproc} )$ 
 $X \rightarrow ( X_{proc} \parallel X_{unproc} )$ 
Repartition
 $( B_{proc} \parallel B_{unproc} ) \rightarrow ( B_{TL} \parallel B_{t_i} \mid B_{TR} )$ 
 $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{TL} \parallel X_{t_i} \mid X_{TR} )$ 
where  $t_i = t_0$ 
Partition  $X_{t_i} \rightarrow ( X_{F_0} \mid X_{F_1} \mid \dots \mid X_{F_{N-1}} )$ 
For processes  $p = 0$  to  $N - 1$ 
  Distribute  $X_{F_p} \rightarrow X_{local}$  on process  $p$ 
while  $t_{proc} \neq t$  do
   $\vdots$ 
  if  $t_{proc} + 1 \neq t$  then
    Repartition
 $( B_{proc} \parallel B_{unproc} ) \rightarrow ( B_{TL} \parallel B_{t_i} \mid B_{TR} )$ 
 $( X_{proc} \parallel X_{unproc} ) \rightarrow ( X_{TL} \parallel X_{t_i} \mid X_{TR} )$ 
    where  $t_i = t_{proc}$ 
    Partition  $X_{t_i} \rightarrow ( X_{F_0} \mid X_{F_1} \mid \dots \mid X_{F_{N-1}} )$ 
  endif
  For processes  $p = 0$  to  $N - 1$ 
    Collect  $B_{F_p} \leftarrow B_{local}$  on process  $p$ 
    if  $t_{proc} + 1 \neq t$  then
      Distribute  $X_{F_p} \rightarrow X_{local}$  on process  $p$ 
    endif
  Merge  $B_{t_i} \leftarrow ( B_{F_0} \mid B_{F_1} \mid \dots \mid B_{F_{N-1}} )$ 
  Continue with
 $( B_{proc} \parallel B_{unproc} ) \leftarrow ( B_{TL} \mid B_{t_i} \parallel B_{TR} )$ 
 $( X_{proc} \parallel X_{unproc} ) \leftarrow ( X_{TL} \mid X_{t_i} \parallel X_{TR} )$ 
enddo

```

Figure 5.2: Optimized layout of data distribution and collection

### 5.1.3 An alternate method of obtaining CSM eigenvalues and eigenvectors

An alternate approach for obtaining the CSM eigenvalues and eigenvectors exists. This method does not call for building the cross-spectra at all, but rather uses a matrix whose columns are the  $f_{bw}t_{blk}$  Fourier data vectors integrated upon for any particular center frequency bin and for any state of the sliding block history. An SVD on this Fourier data matrix yields singular values that are equal to the square roots of the CSM estimate's eigenvalues, and left singular vectors that are equal to the CSM estimate's eigenvectors. To verify this relationship. Let  $A$  be an  $n \times m$  Fourier data matrix, where  $m$  for our purposes is the time bandwidth product  $f_{bw}t_{blk}$ . The CSM estimate  $R$  can now be expressed as  $R = AA^H$ . Recall from Eq. (2.16) that  $R$  may be decomposed with a complex Hermitian EVD such that

$$R = U\Lambda U^H$$

where  $\Lambda$  is a diagonal matrix whose diagonal entries are the eigenvalues of  $R$ , and  $U$  is an  $n \times n$  matrix whose columns are the orthogonal eigenvectors that correspond to the eigenvalues of  $\Lambda$ . The matrix  $A$  may also be decomposed with a singular value decomposition:

$$R = V\Sigma W^H$$

where  $\Sigma$  is an  $n \times m$  diagonal matrix whose diagonal entries are the singular values of  $R$ ,  $V$  is an  $n \times n$

$n$	$D$	$l$	Sun BLAS	compiled BLAS	compiled ATLAS
64	20	101	0.005020	0.003921	0.003618
64	64	101	0.005102	0.003988	0.003806
64	64	201	0.010150	0.007925	0.007098
128	20	101	0.027820	0.023300	0.014150
128	128	101	0.028040	0.023500	0.014540
128	128	401	0.111100	0.093320	0.055060
256	20	101	0.115400	0.097270	0.085290
256	256	201	0.229900	0.194600	0.110400
256	256	401	0.459300	0.388400	0.218400

Figure 5.3: Average `cgemm` runtime (in seconds) for three BLAS implementations

matrix of left singular vectors, and  $W$  is a  $m \times m$  matrix of right singular vectors. Because  $R = AA^H$ , and  $A = V\Sigma W^H$ , we have

$$\begin{aligned}
R &= AA^H \\
&= (V\Sigma W^H)(V\Sigma W^H)^H \\
&= (V\Sigma W^H)(W\Sigma^H V^H) \\
&= V\Sigma(W^H W)\Sigma^H V^H \\
&= V\Sigma(I_m)\Sigma^H V^H \\
&= V\Sigma\Sigma^H V^H \\
&= V|\Sigma|^2 V^H
\end{aligned} \tag{5.10}$$

and since diagonalization is guaranteed to be unique, we may conclude that  $U = V$  and  $\Lambda = |\Sigma|^2$ .

This approach to computing the CSM eigenvalues and eigenvectors is computationally cheaper in most cases than building the CSM estimates outright. The cost of performing the SVD for an  $n \times m$  matrix is roughly  $O(m^3)$ . So, for Fourier data matrices, the SVD takes on a complexity of

$$O_{SVD} = O((f_{bw}t_{blk})^3)$$

In contrast to a complex Hermitian EVD, whose complexity is  $O(n^3)$ , the cost of an SVD on a Fourier data matrix is proportional to the cube of the time-bandwidth product  $f_{bw}t_{blk}$  (ie: the width of the data matrix). An update of the Fourier data matrices has a complexity

$$O_{FDM} = O(cnt f_{bw}t_{blk}).$$

Compare this to the complexity of updating the CSM estimates formulated in Sec. 4.1:

$$O_{CSM} = O(cn^2 t(f_{bw} + t_{blk})).$$

As long as  $n f_{bw}t_{blk}$  remains less than  $n^2(f_{bw} + t_{blk})$ , the cost of maintaining the Fourier data matrices is less than the cost of maintaining CSM estimates. Thus, the Fourier data matrix implementation of CSMs is cost-effective when the time-bandwidth product is not large, or more precisely, when  $f_{bw}t_{blk}$  is less than  $n(f_{bw} + t_{blk})$ .

### 5.1.4 Implementations of the complex singular value decomposition

A review of the DMR complexity analysis results of Sec. 4.1 shows that the sum of all EVD operations has a complexity  $O(cn^3t/k)$  and incurs a runtime cost of approximately  $O(n^5)$  when  $c$ ,  $f$ ,  $l$ ,  $n$ ,  $t$ , and  $D$  are treated equally. Notice that under similar simplifying assumptions the cost of the DMR weight computation and application is also approximately  $O(n^5)$ . Thus, the EVD or SVD have the potential to account for a significant amount of the total cost associated with DMR beamforming.

Regardless of whether the final beamforming program uses an EVD on the CSM estimates or an SVD on the Fourier data matrices, one should consider linking the program against a vendor-optimized version of LAPACK. These library implementations are typically optimized to take advantage of features specific to the ISA of the target CPU architecture. Furthermore, the hardware vendor usually provides well-documented interfaces similar to the official LAPACK calling sequences published in [4].

To verify that the Sun implementation of LAPACK performs well, we benchmarked the SVD routine `cgesvd` included in the Sun performance library with the corresponding routine included in a version of the library compiled from the source distribution obtained from [2]. The compiled LAPACK was built using the version of the BLAS that was compiled from source. These sample averages were obtained by invoking the routine 450 times, each time on a different portion of the test data set. Values for  $n$  and the time-bandwidth product  $f_{bw}t_{blk}$  (corresponding to the number of elements in each matrix column and the number of matrix columns, respectively) were chosen so that the benchmark included SVD results for both square and rectangular “panel” matrices. (Fourier data matrices tend to take the form of panel matrices.) The results of these benchmarks are shown in Fig. 5.4.

$n$	$f_{bw}t_{blk}$	Sun LAPACK	compiled LAPACK
64	24	0.004288	0.002854
64	64	0.018848	0.013511
128	48	0.029035	0.021081
128	128	0.148219	0.105804
256	96	0.253827	0.182367
256	256	1.200401	0.869247

Figure 5.4: Average `cgesvd` runtime (in seconds) for two LAPACK implementations

This informal benchmarking shows that the `cgesvd` routine obtained from a compiled version of the LAPACK library noticeably and consistently completes in less time than the `cgesvd` routine provided in the Sun performance library for all matrix sizes tested.

It is also worth noting that the Sun performance library uses slightly different function prototypes intended to free the user from needing to allocate workspace memory. Consequently, the Sun performance library’s calling interface is not directly compatible with the “official” LAPACK interface.

We chose to link against the `cgesvd` routine included in the compiled LAPACK library. This allows increased portability with the official LAPACK interface as well as substantial performance gains against the Sun-supplied LAPACK implementation.

### 5.1.5 DMR weight computation and application

As currently drawn up, the sequential and parallel DMR algorithms first update the weights submatrices for all center frequency bins in the local processing subband. Then, the DMR weights are applied *en masse* to all Fourier data vectors belonging to the current time sequence. This method requires computing and storing an  $n \times l \times c_{local}$  matrix of complex values, and is useful when the weight update parameter is allowed to be greater than 1. If, however, a weight update were always performed for each new time sequence, (ie:  $k = 1$ ), the loops may be combined. The resulting loop structure would contain an outer loop over center frequency bin while the inner loop would iterate over the  $f_{bw}$  Fourier bins associated with each center frequency bin.

With the two loops fused, only storage for one  $n \times l$  weight submatrix is needed for any one iteration over time sequence. With the realistic parameters values of  $c = 500$ ,  $l = 201$ , and  $n = 100$ , and assuming single precision complex data, the sequential and parallel algorithms as derived earlier in this document require a total of  $500 \times 201 \times 100 \times 8$  bytes, or roughly 77 MB of storage across all parallel processes to store the weights. However, if only one center frequency bin's weights are computed and applied at a time, the DMR beamformer would only require a modest  $1 \times 201 \times 100 \times 8$  bytes, or, 157 KB of memory to house the weight matrix. And considering the scenario with  $N$  distributed memory processes (using MPI on a cluster of compute nodes, for example), the memory requirement for each parallel process is reduced even further.

But once again, this reduction in memory allocation is only feasible if the beamformer were to update the weights every sequence. Otherwise, the computational savings that the weight update parameter provides are generally more attractive than the smaller memory footprint.

### 5.1.6 Data distribution block size

In some cases, it may be desirable to partition and distribute the subbands of Fourier data for several time sequences all at once. In this scenario, the **Partition** statement of the outer loop over time sequence would extract a block of time sequences, and distribute the partitioned subbands of each time sequence to their respective parallel processes.

In computing environments with very fast interconnects between processors, this is less of an issue. But when utilizing many processors, the amount of data being worked on by each processor should be sufficiently large that the processors may compute for a reasonable amount of time before needing to submit their finished work and receive more data. By keeping data distribution and collection infrequent, the cost of initiating the communication can be amortized over a larger amount of useful computation.

## 5.2 Source-level

Source-level optimizations describe enhancements to the implementation, many of which are dependent on the C programming language. Most source-level optimizations focus on minimizing the cost of referencing memory. The optimizations listed in this section were heavily influenced by the guidelines laid out in the Portable High-Performance ANSI C Coding Methodology [6]. These optimizations were applied to both MPI and POSIX threads implementations of the parallel beamforming algorithms.

### 5.2.1 Accumulator variables

Use local variables when performing accumulating computations. This frees the compiler to assigning the accumulator variable to a CPU register, which is much more readily accessible than memory or even cache. This can greatly improve performance when the accumulator is at the center of a deeply-nested loop. The following simplified code fragment captures a situation found in several of the parallel beamformers' loops.

```
void function_name( float* x, float* y, int n )
{
    int i;

    *x = 0;
    for( i = 0; i < n; ++i )
    {
        *x += y[i];
    }
}
```

Here, `*x` will be cause `x` to be dereferenced and loaded from memory (or cache) for each iteration of the for loop. Rather, one should use a local variable for the accumulator:

```

void function_name( float* x, float* y, int n )
{
    int i;
    float temp;

    temp = 0;
    for( i = 0; i < n; ++i )
    {
        temp += y[i];
    }
    *x = temp;
}

```

This reduces the number of memory references to \*x from n to one.

## 5.2.2 Allocating multidimensional arrays

Our parallel CBF and DMR implementations use many multidimensional storage arrays to handle its intermediate data products. Since the sizes of these arrays are not known until runtime, a programmer may be tempted to allocate them statically (either globally or local to a function):

```
complex csm[N_FREQ_MAX][N_ELEM_MAX][N_ELEM_MAX];
```

Access into this array occurs using the an expression such as `csm[i][j][k]`. However, this method of allocation is inefficient. The storage requirements—due to variations in the input parameters—are likely to vary between instantiations of the parallel beamformer, possibly by several as orders of magnitude, so statically allocating an absolute maximum is less than ideal. An alternative is to allocate each dimension of the array dynamically:

```

complex*** csm;
int i, j;

csm = (complex***)malloc( n_freq * sizeof(complex**) );
for( i = 0; i < n_freq; ++i )
    csm[i] = (complex**)malloc( n_elem * sizeof(complex*) );

for( i = 0; i < n_elem; ++i )
    for( j = 0; j < n_elem; ++j )
        csm[i][j] = (complex*)malloc( n_elem * sizeof(complex) );

```

This method is memory-efficient because the amount allocated scales directly with the input parameters. Also, the programmer can still access elements of `csm` using C's indexing operator (ie: `csm[i][j][k]`). However, this method of allocation can be tedious for multidimensional arrays. Furthermore, the operating system makes no guarantee that dynamically allocated memory is contiguous. Without contiguously allocated arrays, cache performance will suffer heavily due to lack of spatial locality with successive memory references. Yet a third way of allocating the necessary storage is:

```

complex* csm;

csm = (complex*)malloc( n_freq * n_elem * n_elem * sizeof(complex) );

```

This method of allocation is simple, while allowing the programmer to utilize the contiguous region of memory however he chooses. Access into such an array for this kind of storage allocation can be performed

using an expression such as `csm[ i*n_elem*n_elem + j*n_elem + k]`, assuming that `n_freq` corresponds to the outermost dimension of the array.

In the parallel CBF and DMR implementations, all multidimensional storage arrays are allocated as contiguous regions of memory. The main disadvantage to this approach is that the programmer cannot use C's indexing brackets to index into the array's inner dimensions. However, this missing convenience is offset by the added flexibility of dimension layout and opportunity for better cache performance.

### 5.2.3 Transforming array indexing into pointer arithmetic

Replace all array subscript indexing in computationally intensive sections of the code with direct pointer dereferencing via pointer arithmetic. This reduces the amount of integer math required to evaluate the subscript expression. The following code fragment sums the first `n` elements of the `j`th row of a column-major order matrix (ie: of leading dimension `column_size`) and stores the result in the variable referenced by `x`:

```
void function_name( float* x, float* A, int column_size, int j, int n )
{
    int i;
    float temp;

    for( i = 0; i < n; ++i )
    {
        temp += A[i*column_size + j];
    }

    *x = temp;
}
```

This code can be rewritten as

```
void function_name( float* x, float* A, int column_size, int j, int n )
{
    int i;
    float temp;

    A = A + j;
    for( i = 0; i < n; ++i )
    {
        temp += *A;
        A += column_size;
    }

    *x = temp
}
```

which uses pointer arithmetic instead of array indexing. The original code performs `n` integer multiplies and `n` integer additions in the process of evaluating the subscript expression. The transformed code requires no arithmetic to evaluate the array index, and only requires one integer addition for each loop iteration to update the pointer to the correct value. The benefit of this optimization becomes greater as the subscript expression in the first fragment of code grows more complex.

## 5.3 Compiler-level

This section describes automatic performance tuning facilities used to compile the parallel implementations. These compiler options and their descriptions are quoted from the `man` page for `cc`, the C compiler included with the Sun Forte Developer 7 development tools.

- `-xlibmil`

Inlines some math library routines for faster execution.

- `-ftrap=%none`

Disables all IEEE 754 trapping modes at program startup.

- `-fns`

On some SPARC systems, the nonstandard floating point mode disables “gradual underflow”, causing tiny results to be flushed to zero rather than producing subnormal numbers. It also causes subnormal operands to be silently replaced by zero. On those SPARC systems that do not support gradual underflow and subnormal numbers in hardware, use of this option can significantly improve the performance of some programs. With this nonstandard mode enabled, floating point arithmetic may produce results that do not conform to the requirements of the IEEE 754 standard.

- `-fsimple=2`

Allows the optimizer to make simplifying assumptions concerning floating-point arithmetic. The option allows three levels of optimization: 0, 1, or 2. Level 2 permits aggressive floating point optimizations that may cause many programs to produce different numeric results due to changes in rounding. For example, `-fsimple=2` permits the optimizer to attempt replacing computations of `x/y` in a given loop where `y` and `z` are known to have constant values, with `x*z`, where `z=1/y` is computed once and saved in a temporary variable, thereby eliminating costly divide operations. However, the optimizer still is not permitted to introduce a floating point exception in a program that otherwise produces none.

- `-x04`

Specifies an advanced level of general machine code optimization. These optimizations include induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, complex expression expansion, and automatic inlining of functions contained in the same source file.

- `-xmemalign=8s1`

For memory accesses where the alignment is determinable at compile time, the compiler generates the appropriate load/store instruction sequence for that alignment of data. For memory accesses where the alignment cannot be determined at compile time, the compiler must assume an alignment to generate the needed load/store sequence. The `-xmemalign` flag specifies the maximum memory alignment of data to be assumed by the compiler in these indeterminable situations. In this case, the compiler assumes at most 8 byte alignment, and raises a signal SIGBUS when a misaligned memory access does take place.

Notice that since these compiler flags are specific to the Sun C compiler, they do not necessarily have analogs in other compilers such as the GNU or Intel C compilers.

---

<sup>1</sup>This compiler flag is required when linking to the Sun performance library.

## Chapter 6

# Performance Analysis

### 6.1 Experiment details

To test the performance of the parallel CBF and DMR algorithms, their MPI and POSIX threads implementations were tested with various numbers of MPI processes and POSIX threads, and also with various CBF and DMR input parameters.

The host computing environment is a Sun Fire™ V880 SMP server, configured with eight 750Mhz UltraSPARC™ III processors and 32 gigabytes of main memory, running the Solaris 8 operating system.

The CBF and DMR algorithms were first implemented with calls to MPI, and then modified to make calls to the POSIX threads library that mimic the blocking nature of MPI’s synchronous send, receive, and broadcast routines. The MPI-enabled CBF and DMR programs were linked against the MPICH implementation [11] [12] of MPI 1.x that uses the `ch_p4` abstract device. The POSIX threads beamformers were linked to the POSIX 1003.1c compliant `pthread` library that accompanies Sun Forte Developer 7 [18].

Timing the MPI beamformers was performed using the `MPI_Wtime()` while the `pthread` implementation was timed using the standard C library’s `time()` and `difftime()` functions. The function `MPI_Wtime()` is a high-resolution timer that returns the time since some fixed date in the past represented as a double precision floating-point value. However, `time()` and `difftime()` use the standard C library’s `time_t` structure, which can only represent the current time to the nearest second. Therefore, we expect that the performance results for the MPI CBF and DMR implementations will offer more precision than the POSIX threads results.

The parameter combinations listed in Fig. 6.1 were run through both CBF and DMR beamformers, using MPI and POSIX threads. These values were chosen because they represent reasonable input parameters for a medium-sized amount of data to run through a research-oriented beamformer. For CBF,  $t_{blk}$  and  $D$  are ignored.

Notice that aside from number of processors, only the number of elements in the sensor array  $n$  is varied. The decision to limit parameter variation to  $n$  was made consciously. Since reporting results on all parameter variations is beyond the scope of this document, we chose to vary the parameter that most heavily influences the runtime duration. According to the complexity analysis performed in Sec. 4.1,  $n$  seems to have the most influence on total runtime.

$f$	$l$	$n$	$t$	$f_{bw}$	$t_{blk}$	$D$	$N$
1200	201	{64,128,256}	100	10	4	20	{1,2,...,8}

Figure 6.1: Beamformer parameter combinations to be used in performance tests

In addition to the aforementioned parameters being run on naïvely partitioned data, we also report the results of the beamformer implementations when adjusted to perform some degree of load-balancing.

The results of the MPI and POSIX threads implementations are presented in Fig. 6.2. These results reflect a naïve partitioning scheme.

The DMR benchmarks were rerun according to the load-balancing described in Sec. 5.1.1. The program was run sequentially several times so that averages for the computational, communications, and I/O runtime coefficients  $\kappa$ ,  $\delta$ , and  $\gamma$  could be sampled. The coefficients were sampled three times for each value of  $n$  being tested. Figure 6.3 shows the coefficient samples and sample means. These values are machine-specific, and sensitive to the response time of the network file system being read from and written to. But even an approximation to these coefficients should yield improved performance over a naïve partitioning.

## 6.2 Results

Beamformer input parameters								MPI		pthreads	
$f$	$l$	$n$	$t$	$f_{bw}$	$t_{blk}$	$D$	$N$	CBF	DMR	CBF	DMR
1200	201	64	100	10	4	20	1	56.1	322.2	60.0	320.0
1200	201	64	100	10	4	20	2	46.6	174.0	42.0	173.0
1200	201	64	100	10	4	20	3	38.3	124.8	37.0	136.0
1200	201	64	100	10	4	20	4	35.9	101.4	35.0	158.0
1200	201	64	100	10	4	20	5	35.4	86.2	35.0	163.0
1200	201	64	100	10	4	20	6	35.1	79.2	34.0	169.0
1200	201	64	100	10	4	20	7	34.1	71.0	31.0	165.0
1200	201	64	100	10	4	20	8	33.3	70.1	32.0	134.0
1200	201	128	100	10	4	20	1	85.5	720.1	88.0	720.0
1200	201	128	100	10	4	20	2	58.0	378.9	58.0	376.0
1200	201	128	100	10	4	20	3	47.4	263.2	48.0	294.0
1200	201	128	100	10	4	20	4	43.0	199.3	44.0	293.0
1200	201	128	100	10	4	20	5	40.6	164.6	46.0	340.0
1200	201	128	100	10	4	20	6	38.9	141.3	47.0	278.0
1200	201	128	100	10	4	20	7	37.9	125.8	40.0	332.0
1200	201	128	100	10	4	20	8	37.4	114.2	38.0	309.0
1200	201	256	100	10	4	20	1	142.4	1981.8	144.0	1973.0
1200	201	256	100	10	4	20	2	87.7	1009.1	86.0	1015.0
1200	201	256	100	10	4	20	3	67.7	684.6	83.0	732.0
1200	201	256	100	10	4	20	4	57.2	522.6	67.0	1023.0
1200	201	256	100	10	4	20	5	51.1	412.7	63.0	988.0
1200	201	256	100	10	4	20	6	47.6	342.1	77.0	991.0
1200	201	256	100	10	4	20	7	45.6	300.9	70.0	881.0
1200	201	256	100	10	4	20	8	44.8	277.6	58.0	876.0

Figure 6.2: Parallel beamformer runtime results (in wall clock seconds) for naïve partitioning

## 6.3 Analysis

Parallel speedup and efficiency for the MPI beamformers was computed according to Eq. (4.6) and (4.14), respectively, and plotted for each value of  $n$  as a function of the number of processors  $N$ . Figure 6.5 shows these results when the data is partitioned naïvely into equal subbands. Immediately, we can see that as  $N$  increases, the MPI DMR beamformer holds its efficiency better than the MPI CBF beamformer for similar

$n$	$N$	$\kappa$	$\delta$	$\gamma$
64	1	$1.702 \times 10^{-8}$	$1.273 \times 10^{-7}$	$8.790 \times 10^{-7}$
64	1	$1.704 \times 10^{-8}$	$1.276 \times 10^{-7}$	$8.350 \times 10^{-7}$
64	1	$1.707 \times 10^{-8}$	$1.270 \times 10^{-7}$	$8.300 \times 10^{-7}$
128	1	$1.313 \times 10^{-8}$	$9.684 \times 10^{-8}$	$6.792 \times 10^{-7}$
128	1	$1.309 \times 10^{-8}$	$9.691 \times 10^{-8}$	$6.659 \times 10^{-7}$
128	1	$1.307 \times 10^{-8}$	$9.672 \times 10^{-8}$	$6.687 \times 10^{-7}$
256	1	$7.642 \times 10^{-9}$	$6.877 \times 10^{-8}$	$4.781 \times 10^{-7}$
256	1	$7.629 \times 10^{-9}$	$6.879 \times 10^{-8}$	$4.771 \times 10^{-7}$
256	1	$7.633 \times 10^{-9}$	$6.915 \times 10^{-8}$	$4.781 \times 10^{-7}$
Average		$1.259 \times 10^{-8}$	$9.768 \times 10^{-8}$	$6.656 \times 10^{-7}$

Figure 6.3: Sampled values for  $\kappa$ ,  $\delta$  and  $\gamma$

Beamformer input parameters								MPI	pthreads
$f$	$l$	$n$	$t$	$f_{bw}$	$t_{blk}$	$D$	$N$	DMR	DMR
1200	201	64	100	10	4	20	1	322.2	320.0
1200	201	64	100	10	4	20	2	172.0	173.0
1200	201	64	100	10	4	20	3	117.2	218.0
1200	201	64	100	10	4	20	4	88.1	112.0
1200	201	64	100	10	4	20	5	73.2	162.0
1200	201	64	100	10	4	20	6	65.0	170.0
1200	201	64	100	10	4	20	7	54.4	184.0
1200	201	64	100	10	4	20	8	50.0	175.0
1200	201	128	100	10	4	20	1	720.1	720.0
1200	201	128	100	10	4	20	2	375.7	375.0
1200	201	128	100	10	4	20	3	263.7	397.0
1200	201	128	100	10	4	20	4	192.2	371.0
1200	201	128	100	10	4	20	5	152.9	296.0
1200	201	128	100	10	4	20	6	129.0	264.0
1200	201	128	100	10	4	20	7	112.1	363.0
1200	201	128	100	10	4	20	8	102.9	316.0
1200	201	256	100	10	4	20	1	1981.8	1973.0
1200	201	256	100	10	4	20	2	1039.7	1047.0
1200	201	256	100	10	4	20	3	690.7	1352.0
1200	201	256	100	10	4	20	4	528.6	1005.0
1200	201	256	100	10	4	20	5	425.0	903.0
1200	201	256	100	10	4	20	6	373.9	1056.0
1200	201	256	100	10	4	20	7	316.4	909.0
1200	201	256	100	10	4	20	8	310.3	760.0

Figure 6.4: Parallel beamformer runtime results (in wall clock seconds) for load-balanced partitioning

values of  $n$ . This observation was predicted as a result of the isoefficiency analysis in Sec. 4.2.3, and can be attributed to the higher-order complexity of the DMR algorithm.

By checking the timing output from the beamformer programs we find that each experiment in Fig. 6.5 spent approximately 24 seconds performing file I/O. This result is true of both CBF and DMR results, for all values of  $n$  and  $N$  tested. Recall that the parallel beamformers must perform file I/O sequentially due to

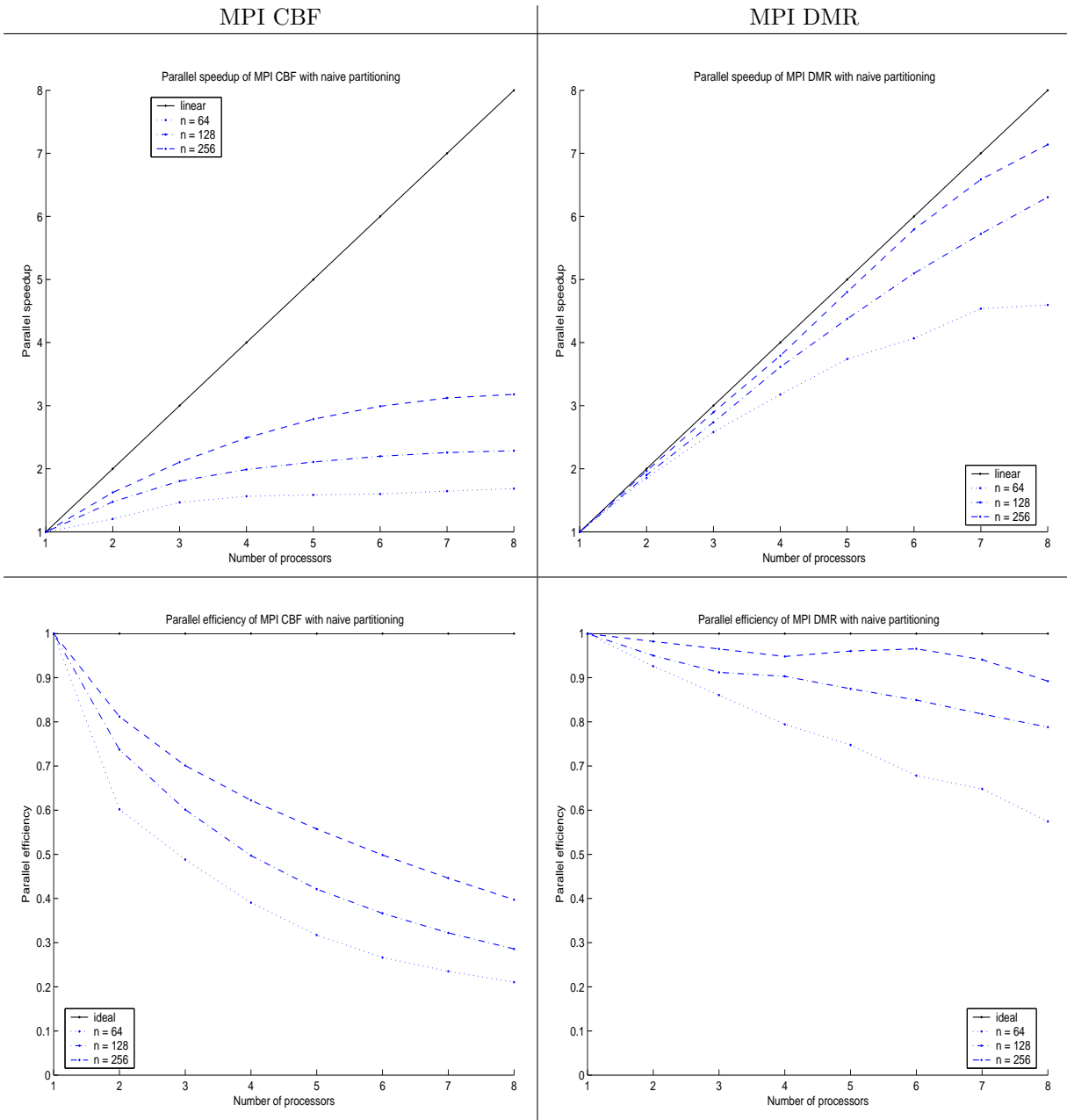


Figure 6.5: Parallel speedup and efficiency for CBF and DMR implemented with MPI with naïve data partitioning.

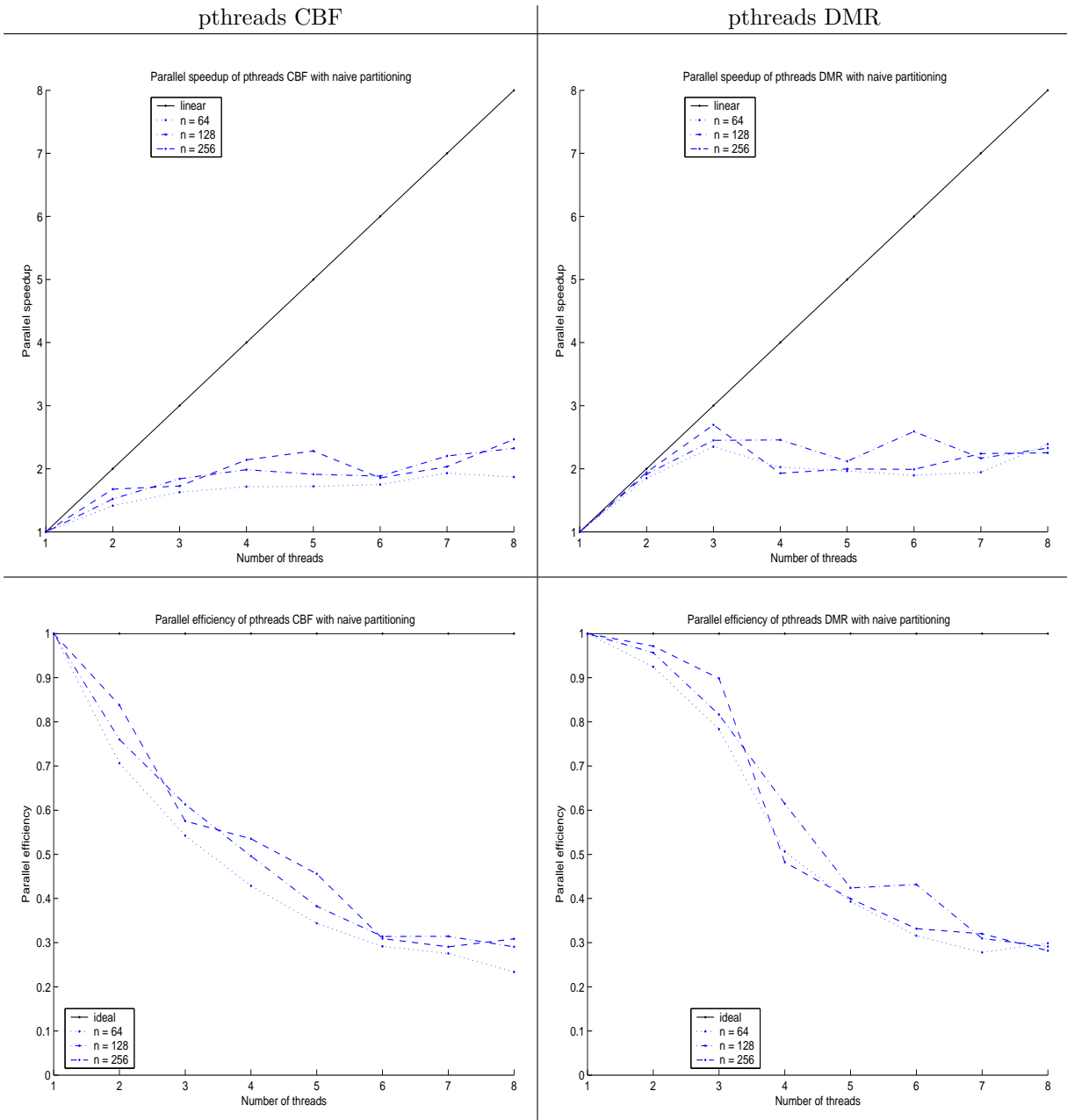


Figure 6.6: Parallel speedup and efficiency for CBF and DMR implemented with pthreads with naïve data partitioning.

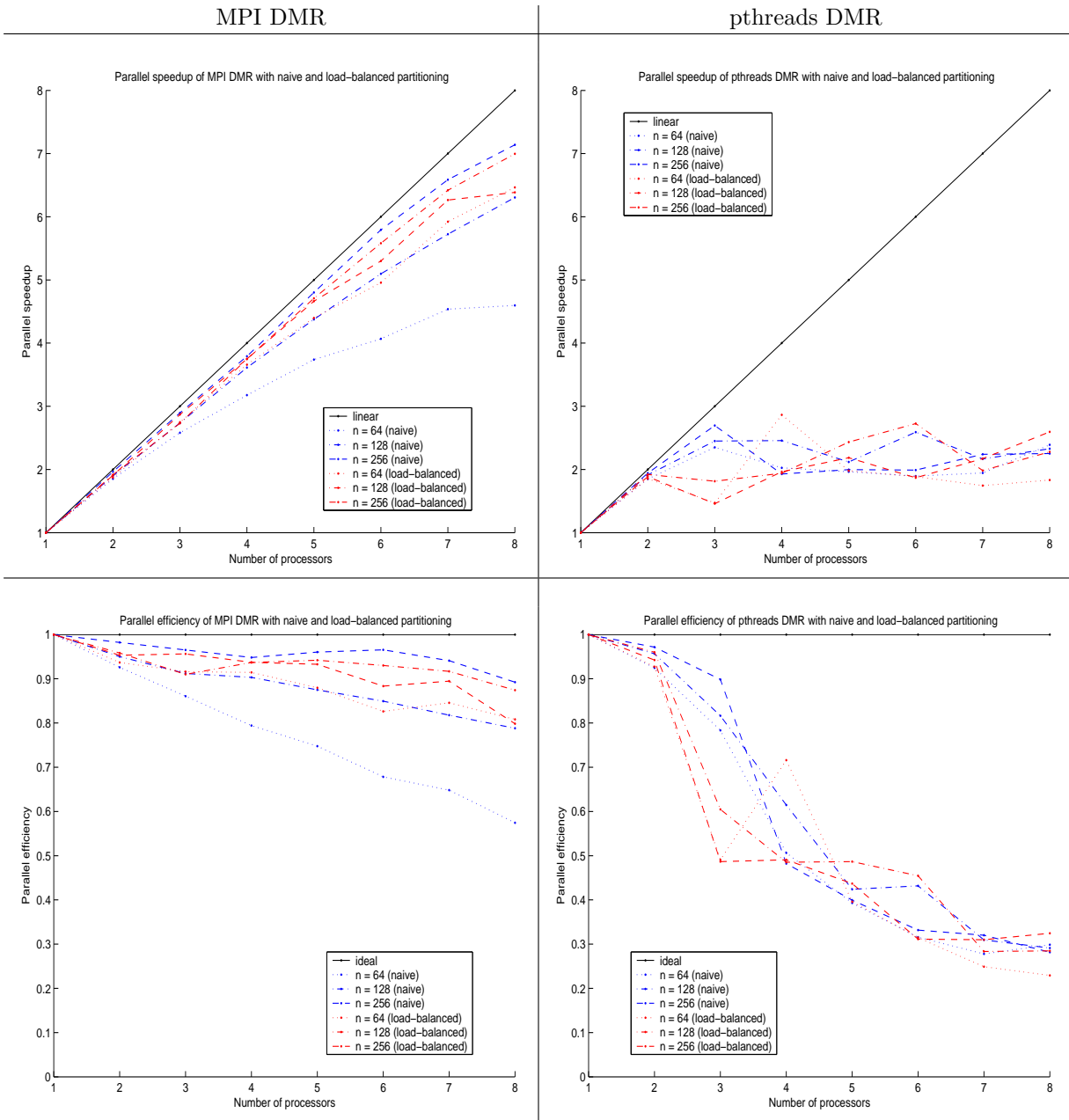


Figure 6.7: Parallel speedup and efficiency for MPI and pthreads DMR with naïve and load-balanced data partitioning.

the lack of balance between each process’s workload. That is, assuming that all processes progress through their computations at the same rate, the server will finish last, at which time it must still read the next block of Fourier data to prepare for the next data distribution. So, during this time, all client processes are idle. As  $N$  grows, the number of clients grows, meaning more CPU time is wasted while performing I/O. For large enough  $N$ , the time spent performing sequential file I/O dwarfs the amount of time spent performing parallel computations. This scenario is reflected by how quickly the CBF speedup tapers off in the CBF plots of Fig. 6.5, and agrees with Amdahl’s Law. That is, even with infinite speedup, a program with 24 seconds of sequential runtime will finish in no less than 24 seconds.

Remarkably, for the largest value of  $n$  tested (256 array elements), the MPI DMR beamformer offers near-linear speedup on parallel instantiations up to 6 processors, while a similar level of speedup is held up to only 2 processors when  $n = 128$ .

The pthreads implementations did not fair nearly as well as the MPI beamformers. Given the MPI CBF results, weak performance was expected from the pthreads implementation. Thus, the CBF speedup and efficiency in Fig. 6.6 is not surprising. However, the pthreaded DMR implementation performance languishes for values of  $N$  greater than 3. A closer look at the timing output from these beamformers reveals the source of the higher pthread DMR runtimes: communication. When the implementations were modified to use the POSIX threads library, pthread primitives and functions were used to build more complex functions that behave (and block) like the analagous routines in the MPI library. These functions rely heavily on mutex locking and condition variable waiting and signaling. It is possible that the retrofitted pthread communications layer is performing excessive or redundant synchronization. But if the pthread communications is coded correctly, then the limited performance may stem from the Sun implementation of the POSIX threads library. In either case, the wide variation of efficiencies for increasing values of  $N$  is suspicious.

When load-balancing was employed using the sampled mean values for  $\kappa$ ,  $\delta$ , and  $\gamma$ , MPI DMR performance improved for  $n = 64$  and  $n = 128$ . Furthermore, the increase in efficiency gained from load-balancing when  $n = 64$  is greater than the corresponding gain for  $n = 128$ . And for load-balanced MPI DMR with  $n = 256$ , the change in efficiency comes in the form of a loss. This suggests that there is some cross-over point after which load-balancing, as currently formulated, no longer improves performance. These results lead to the observation that load-balancing for MPI DMR provides the greatest improvement when  $n$  is relatively small, probably because the beamformer is inherently less efficient for small  $n$ , which leaves more room for improvement.

Figure 6.8 shows two screenshots of the output from the visualization tool `jumpshot` when fed timing data from the automatic logging facilities provided by the MPE library. These graphs show the state of the beamformer for the server and all clients as time progresses. Both figures show the results of an MPI DMR experiment using 4 processors. The top screenshot reflects naïvely partitioned subbands. The green regions represent periods of file I/O, the grey regions correspond to beamforming and related computations, and the gaps in processes 0 through 2 reflect periods of idle time for the clients. The bottom screenshot shows the results of conservative load-balancing. Note that the gaps of idle time for processes 0 through 2 become narrower when we attempt to load-balance the data partitioning.

The timing data from the beamformers suggests that the poor pthreads performance can only be attributed to an increase in communication costs; all other timing breakdowns show computational and file I/O costs similar to MPI timings. So, we did not expect load-balancing to improve the pthreaded DMR implementations. The speedup and efficiency graphs for the threaded DMR implementation in Figure 6.7 agree with this prediction. In fact, for some values of  $N$ , the threaded DMR with load-balancing offers lower performance than the same implementation run with naïve partitioning.

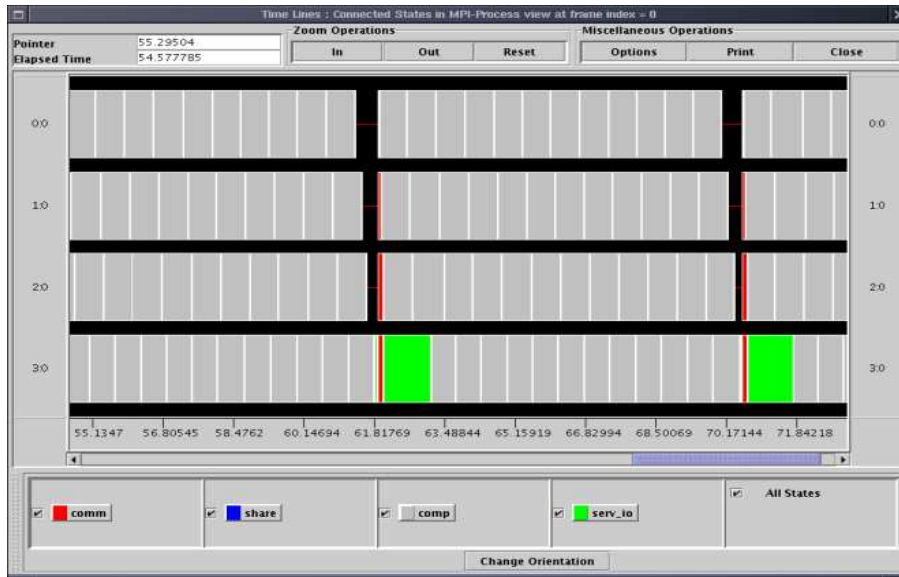
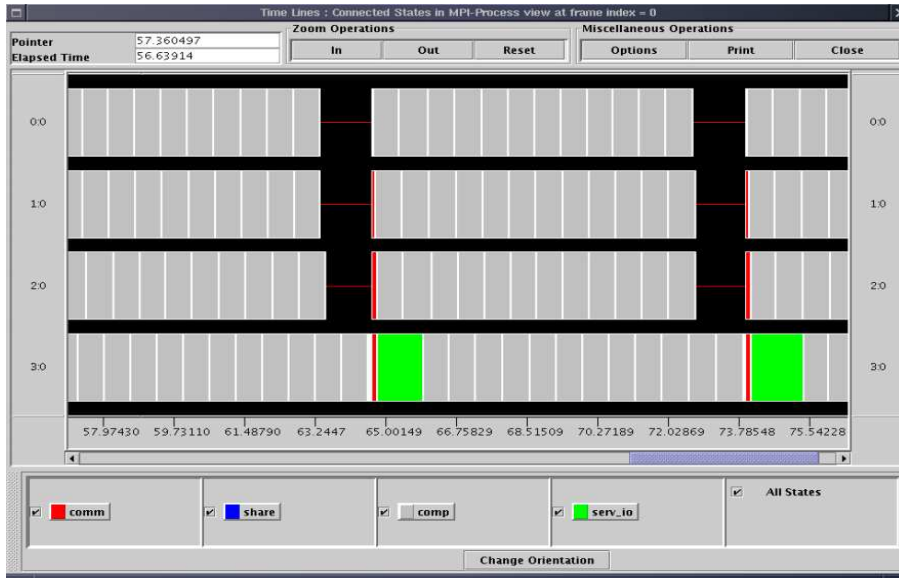


Figure 6.8: Output from `jumpshot` tool showing MPI DMR beamformer state across 4 processes as a function of time. The processing in the top window was partitioned naïvely while the data in the bottom window was partitioned with some load-balancing.

## Chapter 7

# Conclusions

Before implementing a parallel frequency-domain beamformer, we wished to formally derive the sequential algorithms from well known beamforming principles, and then parallelize the algorithm with MPI-friendly notation. The methods developed by Gunnels and van de Geijn heavily influenced and aided the derivation of each loop. Aside from helping to derive the algorithms, these formal methods yield algorithms whose correctness is verified, so we can be confident that the high-level algorithm is correct before implementation.

Two implementations of the parallel algorithms were measured for performance: one using the message passing interface and one using POSIX threads. The MPI implementation outperforms pthreads in all instances tested. This result disagrees with expectation, since POSIX threads are generally considered more efficient than message passing, even when the message passing is performed on a symmetric multiprocessor. It is worth recalling that the CBF and DMR beamformers were first implemented in MPI, and then rewritten to use POSIX threads. A ground-up rewrite made little sense because the resulting implementation would have closely resembled the MPI implementation due to the synchronous client-server model inherent in the parallel algorithm design. Perhaps a more efficient implementation would be possible if the model were not synchronous, or not strictly based on the client-server paradigm. Another cause for poor performance in the threaded implementation could stem from excessive synchronization and mutex locking. The code should be rechecked for redundant synchronization and signaling to ensure the fault does not lie with the use of the POSIX threads interface. Curiosity also begs that the Sun implementation of POSIX threads be further tested for performance anomalies.

The MPI implementation of DMR beamformer scales fairly well up to 8 processors on the Sun Fire server used for testing, and performance improves for smaller values of  $n$  when load-balancing is employed. The MPI CBF beamformer is consistently harder to keep busy with work than the DMR beamformer. Parallel CBF performs poorly with many processors because the time spent on communication and file I/O are significant relative to the amount of work that each process performs. DMR calls for more work to be done per unit of data, and thus spends a higher percentage of time performing useful computations.

Without load-balancing, the MPI DMR loses efficiency quickly as the number of processors grows large. And though the processes' subbands seemed adjustable enough, the precise amount by which to decrease the server's workload and the corresponding amount by which to increase the clients' workloads were not immediately obvious. However, expressions that describe the ideal adjustment were derived. And though the ideal adjustment is not always possible due to the granularity of the center frequency bins (ie: the smallest indivisible amount of work that can be beamformed with DMR), good workload adjustments are possible if the runtime complexity coefficients can be approximated. In fact, with just a few benchmarks, the computational, communications, and file I/O coefficients can be estimated well enough to provide reasonable load balancing.

Because jobs with fewer processors tend to be more efficient to begin with, adjusting the server workload becomes more beneficial as the number of processors increases. However, even when the workloads of each process are carefully balanced, loss of efficiency is inevitable. Still, load-balancing can significantly dampen the rate at which efficiency drops off as more processors are utilized.

It may be possible to implement an adaptive load-balancing mechanism. Such a mechanism might keep track of total runtime spent performing beamforming, communication, and I/O for some fixed number of time sequences, and then use this information to dynamically readjust the subbands to equalize the distribution of total work assigned to each process. Subsequent efforts to fine-tune the system should explore this alternative method of automatic load-balancing.

Future research should also extend the performance analysis to clusters of homogeneous compute nodes. This would allow a true test of the MPI DMR implementation's scalability. The performance of MPI DMR on clusters may be of particular interest given the price/performance issues that surround the purchase of an SMP versus an equivalent computing cluster.

# Bibliography

- [1] BLAS (Basic Linear Algebra Subprograms). <http://www.netlib.org/blas>, 2003.
- [2] LAPACK - Linear Algebra PACKage. <http://www.netlib.org/lapack>, 2003.
- [3] D. A. Abraham and N. L. Owsley. Beamforming with dominant mode rejection. In *Oceans 1990 Conference Proceedings*, pages 470–475, 1990.
- [4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [5] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Transactions on Mathematical Software*, 1996.
- [6] Jeff Bilmes, Krste Asanovic, Jim Demmel, Dominic Lam, and Chee-Whye Chin. Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. Technical report, University of California at Berkeley, August 1996. LAPACK Working Note 111.
- [7] James C. Browne. Performance of Parallel Programs. Unpublished lecture notes from CS 377 course-work, 2001.
- [8] H. Cox, R. M. Zeskind, and M. M. Owen. Robust adaptive beamforming. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(10):1365–1376, 1987.
- [9] Kazushige Goto and Robert A. van de Geijn. On Reducing TLB Missing in Matrix Multiplication. *ACM Transactions on Mathematical Software*, submitted, 2003.
- [10] David E. Grant, Mimi Z. Lawrence, and Jonathan H. Gross. Cross-spectral matrix estimation effects on adaptive beamforming. *Journal of the Acoustic Society of America*, 98:517–524, 1995.
- [11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [12] William D. Gropp and Ewing Lusk. *User's Guide for mpich, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.
- [13] J. Gunnels, G. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 2001.
- [14] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A Family of High-Performance Matrix Multiplication Algorithms. A Technical Paper Submitted to the International Conference on Computer Sciences, 2001.
- [15] John A. Gunnels and Robert A. van de Geijn. A Recipe for Deriving High-Performance Dense Linear Algebra Algorithms. Unpublished computer sciences coursework document.

- [16] Harry F. Jordan and Gita Alaghband. *Fundamentals of Parallel Processing*, chapter 2, pages 23–46. Prentice Hall, 2003.
- [17] Lawrence Livermore National Laboratory. POSIX threads programming. <http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html>, 2003.
- [18] Sun Microsystems, Inc. Multithreaded Programming Guide. <http://docs.sun.com>, 2001.
- [19] Evan K. Westwood, Jonathan H. Gross, Richard A. Gramann, Clark S. Penrod, Fredrick W. Machell, and David E. Grant. Composite beam correlation: algorithm description and a comparison of array performance. Technical Report ARL-TL-EV-96-50, Applied Research Laboratories, The University of Texas at Austin, 1996.
- [20] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. <http://math-atlas.sourceforge.net>, 2003.