

ABSTRACT

Asynchronous Image Reconstruction

Zongqi Ritchie Cai, Ph.D.

Mentor: Keith Schubert, Ph.D.

Algebraic Reconstruction Technique, also known as ART, is the go-to method for medical image reconstruction. For more than 50 years, since the original paper about ART is published, various reconstruction methods based on ART emerged for faster runtime performance. These methods are designed to suit different types of hardware. However, none of these methods is ART equivalent. In this work, I introduce a new implementation method that is ART equivalent, has a very fast runtime performance, and is very scalable on today and future hardware. It opens a brand new door to how we should implement ART in the future.

Asynchronous Image Reconstruction

by

Zongqi Ritchie Cai, B.S., M.S.

A Dissertation

Approved by the Department of Electrical and Computer Engineering

Kwang Y. Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Philosophy

Approved by the Dissertation Committee

Keith Schubert, Ph.D., Chairperson

Robert J Marks II, Ph.D.

Reinhard Schulte, M.D.

Jeffrey S. Olafsen, Ph.D.

Accepted by the Graduate School

December 2021

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

Copyright © 2021 by Zongqi Ritchie Cai

All rights reserved

TABLE OF CONTENTS

LIST OF FIGURES	vii
LIST OF TABLES	ix
ACKNOWLEDGMENTS	xi
DEDICATION	xii
1 Introduction	1
1.1 Projection Methods	1
1.1.1 Algebraic Reconstruction Technique	2
1.1.2 Block Iterative Algorithm	2
1.1.3 String Averaging	8
1.1.4 Summary	11
1.2 Concurrency	11
1.2.1 Concurrency vs Parallelism	11
1.2.2 Processes	12
1.2.3 Locks	15
1.2.4 Data Structures	16
1.2.5 Concurrencies in ART Algorithms	17
2 Theory	19
2.1 Asynchronous Art Algorithm	19
2.2 *	21
2.3 Theoretical Speedup	24

2.3.1	Data Distribution	24
2.3.2	Time Complexity	27
3	Implementation	28
3.1	Dataset	28
3.2	Sorting	32
3.3	Index Tree	32
3.4	Language and Environment	32
3.5	Async Grid	36
3.5.1	Channels	37
3.5.2	Blocking vs Parking	37
3.5.3	Async Node	38
3.5.4	Grid Generation	39
3.5.5	Grid Operation	47
4	Experiment	54
4.1	Introduction	54
4.2	Hardware	54
4.3	Dataset	54
4.4	Experiments	55
4.4.1	CTP404 Sensitometry Phantom	56
4.4.2	George Phantom 2.5mm thickness	57
4.4.3	George Phantom 1mm thickness	58
5	Discussion	62
5.1	Performance	62

5.1.1	Theoretical Speedup	62
5.1.2	Timing Data	62
5.2	Scalability	66
5.2.1	Scale Over Multi-cores	66
5.2.2	Scale Over Network	66
5.3	Improvements	67
6	Conclusion	70
6.1	Future Direction	71
	APPENDICES	72
	APPENDIX A Testing Scripts	73
A.1	CTP404 Phantom	73
A.2	George 2.5mm	75
A.3	George 1mm	77
	APPENDIX B CTP404 Phantom History Counts	79
	APPENDIX C George 2.5mm History Counts	81
	APPENDIX D George 1mm History Counts	83
	BIBLIOGRAPHY	88

LIST OF FIGURES

1.1	ART projection paths	3
1.2	BIP algorithm convergence	5
1.3	Cimmino project	6
1.4	SAP path 1	9
1.5	SAP path 2	9
1.6	SAP path 3	10
1.7	Average of three paths	10
1.8	Process states	13
1.9	Immutable Data Structure	17
1.10	Concurrency in ART	18
1.11	Concurrency in BIP	18
1.12	Concurrency in SAP	18
2.1	ART concurrency stack	20
2.2	Experimental data distribution per block	25
3.1	3D images	29
3.2	Data Format	30
3.3	Art Concurrency Blocks	31
3.4	Data Format	33
3.5	Index tree structure	34
3.6	Java thread life cycle	35
3.7	Reconstruction steps for different types of threads.	40

3.8	Async nodes connection in a grid	41
3.9	Blocks and layers of the grid	42
3.10	Iterators	45
3.11	Head and tail region in the grid	46
3.12	Format used by sent messages	48
3.13	Testing a grid with 16 slices and depth = 5	50
3.14	Grid with slices = 16, depth = 5	51
3.15	Threaded version of async reconstruction	53
4.1	Reconstruction of CTP404 phantom 2mm thickness, iteration 6 . . .	57
4.2	Reconstruction of George phantom with 2mm thickness, iteration 6	59
4.3	Reconstruction of George phantom with 1mm thickness, iteration 6	61
5.1	Block partition across multiple machines	68
5.2	Thread abstraction layer	68

LIST OF TABLES

2.1	Theoretical Speed up for sample dataset	26
4.1	Configurations of computers used for performance testing.	54
4.2	Dataset differences	55
4.3	CTP404 dataset history distribution over slice depth	56
4.4	George 2.5mm dataset history distribution over slice depth	58
4.5	George 1mm dataset history distribution over slice depth	60
5.1	Theoretical Speed up for CTP404 dataset	63
5.2	Theoretical Speed up for George 2.5mm dataset	63
5.3	Theoretical Speed up for George 1mm dataset	64
5.4	Timing for tests performed for dataset CTP404 on pb005	65
5.5	Timing for tests performed for dataset CTP404 on lm001	65
5.6	Timing for tests performed for dataset george 2.5mm on pb005	65
5.7	Timing for tests performed for dataset george 2.5mm on lm001	65
5.8	Timing for tests performed for dataset george 1mm on pb005	65
5.9	Timing for tests performed for dataset george 1mm on lm001	65
B.1	History count for CTP404 dataset slice 0-4	79
B.2	History count for CTP404 dataset slice 5-9	79
B.3	History count for CTP404 dataset slice 10-14	80
C.1	History count for george 2.5mm dataset slice 0-4	81
C.2	History count for george 2.5mm dataset slice 5-9	81
C.3	History count for george 2.5mm dataset slice 10-14	82

C.4	History count for george 2.5mm dataset slice 15-19	82
C.5	History count for george 2.5mm dataset slice 20-24	82
D.1	History count for george 1mm dataset slice 0-6	83
D.2	History count for george 1mm dataset slice 7-13	84
D.3	History count for george 1mm dataset slice 14-20	84
D.4	History count for george 1mm dataset slice 21-27	85
D.5	History count for george 1mm dataset slice 28-34	85
D.6	History count for george 1mm dataset slice 35-41	86
D.7	History count for george 1mm dataset slice 42-48	86
D.8	History count for george 1mm dataset slice 49-55	87
D.9	History count for george 1mm dataset slice 56-62	87

ACKNOWLEDGMENTS

First and foremost, I would like to thank my mother for her unconditional love and support, without which I certainly would not have finished this work.

I am also extremely grateful to my aunt, who made working toward Ph.D. possible by bringing me overseas to study abroad when I was 16. Ph.D. certainly wasn't on the map at that time, but her support made it a possibility down the road. She is undoubtedly the world's greatest aunt for sure!

To my great friend, mentor, and advisor Dr. Schubert, I would like to say thank you for always being a light in the darkness. Whenever I'm lost and depressed, talking to you has always given me some hope and direction. I greatly appreciate your understanding and the freedom you've provided to me. I think I could only do what I have done under your guidance.

Last but not least, I would like to express my deep and sincere appreciation for the efforts Brian Sitton has provided in finishing this work. Without the access and technical support to those computing nodes he made available, I would not have come close to finishing this work.

DEDICATION

To my Dad.

CHAPTER ONE

Introduction

The subject of this dissertation is computing medical image reconstruction using iterative projection methods efficiently on modern many-core CPU architecture by exploring data concurrency. This chapter will examine projection methods used for reconstructions and their implementations, techniques used when implementing multi-threaded programs, and the proposed computation technique for this type of reconstruction. Reviews of projection methods and their implementations are presented in section 1.1. An overview of some implementation techniques for multi-threaded programs is in section 1.2. Finally, an overview of the proposed computation technique is explained in section 1.3.

1.1 Projection Methods

This section covers some brief overviews of current iterative projection methods for image reconstructions in pCT. These algorithms uses projections onto sets while relying on the general principle that when a family of (usually closed and convex) sets is present, then projections onto the given individual sets are easier to perform than projection onto other sets (intersections, image sets under some transformation, etc.) that are derived from the given individual sets. This is definitely the case in pCT reconstruction, where the sets to be projected on in the iterative process are the hyperplane H_i defined by the i -th row of the $m \times n$ linear system $Ax = b$, namely,

$$H_i = \{x \in \mathfrak{R}^n \mid \langle a^i, x \rangle = b_i\} \quad \text{for } i = 1, 2, \dots, m \quad (1.1)$$

Where \mathfrak{R}^n is Euclidean n -dimensional space, a_i is the i -th row of A , b_i is the i -th element of b . In pCT, a_i^j correspond to the length of intersection of the i -th proton

history with the j -th voxel, x is the the unknown relative electron density image vector and b_i is the integral relative electron density corresponding to the energy lost by the i -th proton along its path.

1.1.1 Algebraic Reconstruction Technique

Given x^k , x^{k+1} can be computed by

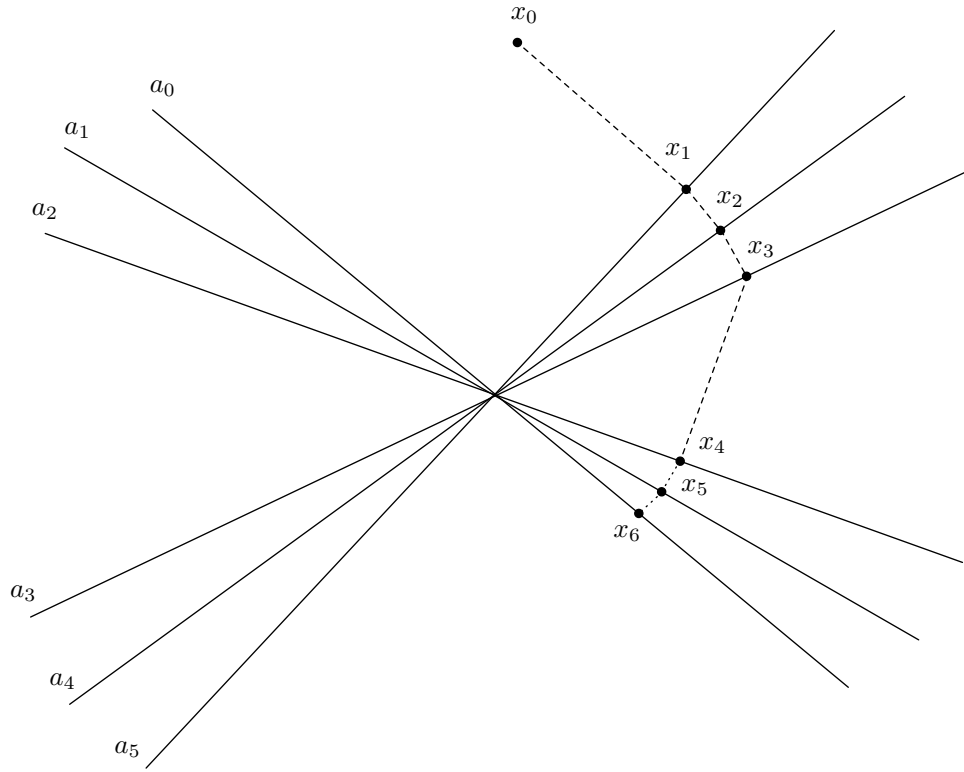
$$x^{k+1} = x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \quad (1.2)$$

where a_i is the i -th row of A and b_i is the i -th element of b . Initial value, $x^0 \in \mathfrak{R}^n$, can be a random valued vector. It is considered a full iteration when all the rows of A are used once. Typically, it takes multiple iterations to converge to a acceptable result. Although not specified, the order of a_i used can greatly impact the speed of the convergence. Figure 1.1 demonstrates that two very different ordering of the same set of data can produce drastically different result in one iteration.

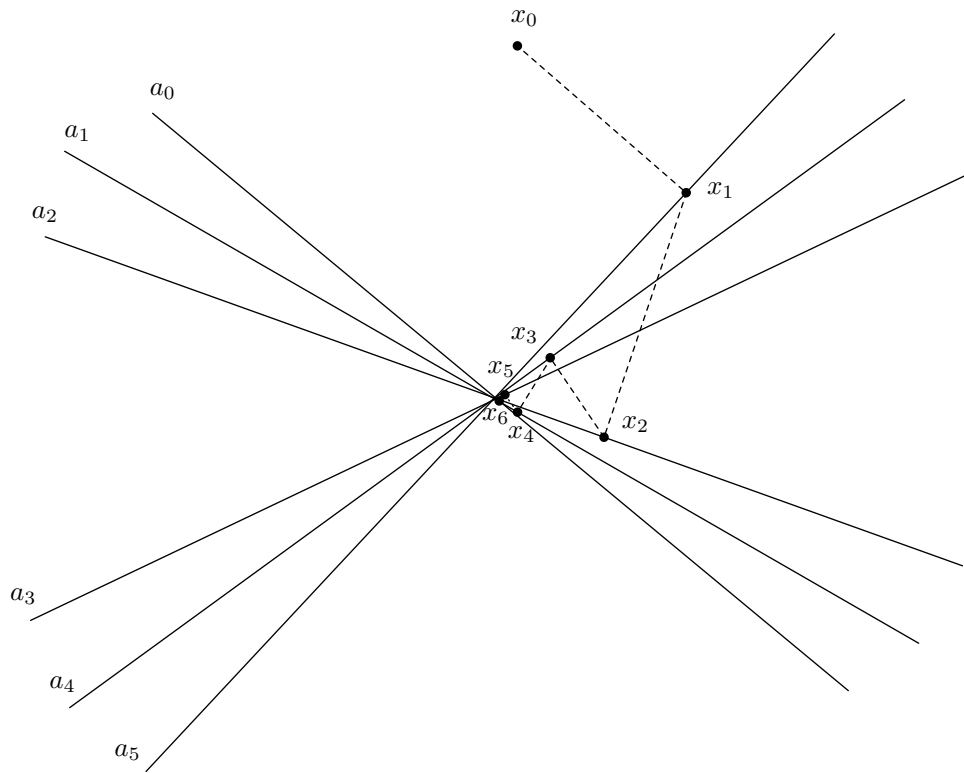
This method is also known as ART and is considered to be the standard projection method for pCT image reconstruction. All other methods reviewed here are either derived from it for computational efficiency purpose, or very closely related. The mathematical description of the algorithm is very sequential since x^{k+1} depends on previous result x^k . However, this flaw could also be considered as an advantage. Since every projection is using the result of all previous projections, x^k has maximized the information gain on given data.

1.1.2 Block Iterative Algorithm

Block iterative projection methods, also known as BIP, divides all the hyperplanes available into blocks. For each projection step, BIP projects all the hyperplanes in one block at the same time. The number of blocks, their size, and the assignments of the hyperplanes H_i to the blocks may all vary, provided that the weights attached to



(a) *sub-optimal projection sequence*



(b) *a much better projection sequence*

Figure 1.1: ART projection paths

the hyperplanes fulfills the condition of constituting a *fair* sequence, which is defined as follows,

Let $I = 1, 2, \dots, m$, and let $H_i | i \in I$ be a finite family of hyperplanes with nonempty intersection $H = \cap_{i \in I} H_i$. Denoting the nonnegative ray of the real line by \mathbf{R}_+ , introduce a mapping $w : I \rightarrow \mathbf{R}_+$, called a weight vector, with the property $\sum_{i \in I} w(i) = 1$. A sequence $\{w^k\}_{k=0}^\infty$ of weight vectors is called *fair* if, for every $i \in I$, there exists infinity many values for k for which $w^k(i) > 0$.

The general form of BIP algorithm can be express in the following form:

$$x^{k+1} = x^k + \lambda_k \left(\sum_{i \in I(k)} w^k(i) \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \right) \quad (1.3)$$

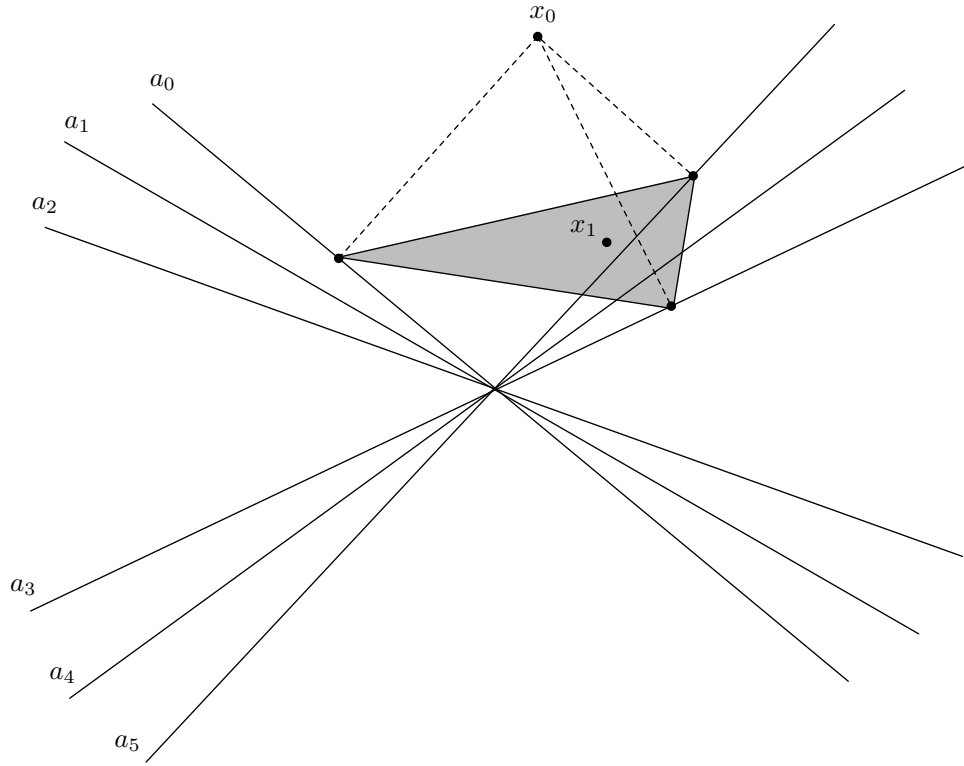
The advantage of BIP is that multiple hyperplanes can be used for computing x^{k+1} instead of just 1 in ART. The projections of these hyperplanes can be calculated at the same time using available hardware features to speed up the computation for each iteration. The convergence of BIP algorithm can be shown in Figure 1.2.

Although each iteration is calculated faster, the rate of convergence tends to be inferior to the ART algorithm. This is because the projection of each hyperplanes used in Equation 1.3 does not take other hyperplanes in the same step into consideration, because they are computed in parallel.

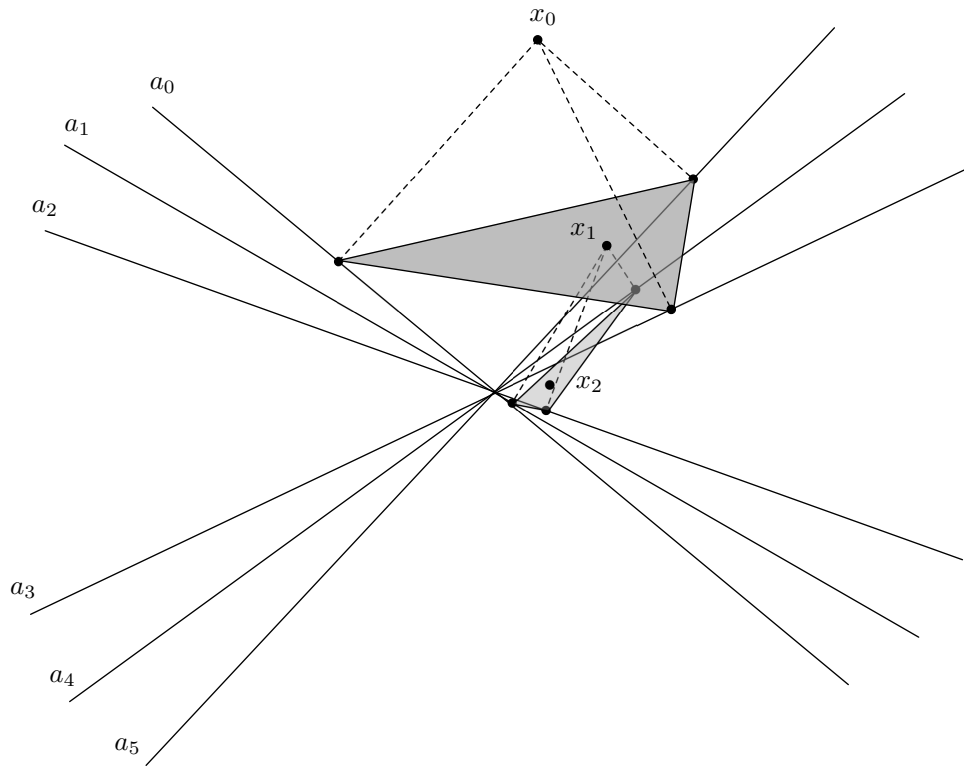
The following sections will discuss several variations of BIP algorithm. ART can be considered as a special case of BIP where each group contains only one hyperplane.

1.1.2.1 Cimmino. Like ART, Cimmino is another extreme variation of BIP where all hyperplanes are grouped into one single block.

$$x^{k+1} = x^k + \frac{1}{m} \sum_{i=1}^m \left(\lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \right) \quad (1.4)$$



(a) BIP block 1



(b) BIP block 2

Figure 1.2: BIP algorithm convergence

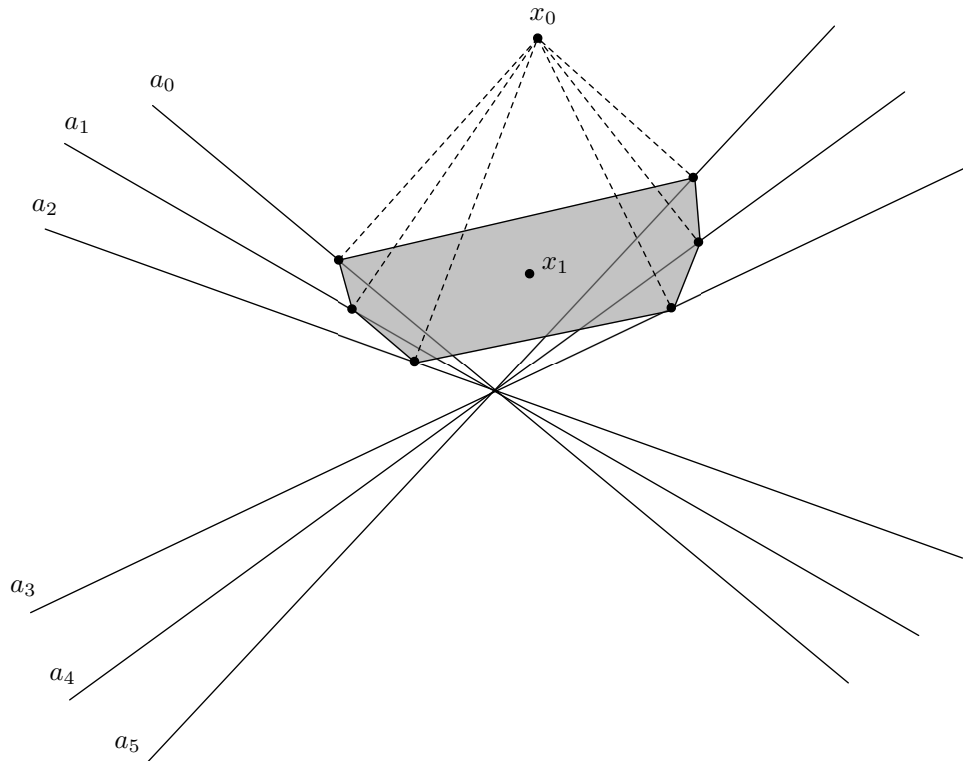


Figure 1.3: Cimmino project

Figure 1.3 show how Cimmino algorithm converges. Theoretically, if there is available hardware resources available, all projections can be computed simultaneously before summing altogether. However, this makes the result at the end of each iteration less accurate. It takes more iterations to converge to a desired result.

1.1.2.2 Block Iterative Component Averaging. The block-iterative component averaging (BICAV) algorithm is a special variant of BIP that incorporates component-related weighting in the vectors w_k . BICAV also differs in the method of projection onto the individual hyperplanes, making use of generalized oblique projections, as opposed to orthogonal projections. For a detailed discussion of the consequences of this on the projection algorithm see [1]. The iterative step in BICAV is

defined as follows:

$$x^{k+1} = x^k + \lambda_k \sum_{i \in I_t(k)} \frac{b_i - \langle a_i, x^k \rangle}{\sum_{l=1}^n s_l^{t(k)} (a_i^l)^2} a_i^i \quad (1.5)$$

where $\{s_l^t\}_{l=1}^n$ is the number of non-zero element $a_i^l \neq 0$ in the l -th column of the t -th block of the matrix A given by

$$a_t = \begin{pmatrix} a^{t_1} \\ a^{t_2} \\ \vdots \\ a^{t_{m(t)}} \end{pmatrix} \quad (1.6)$$

and $\{\lambda_k\}_{k=0}^\infty$ is a sequence of user-determined relaxation parameters.

1.1.2.3 The Diagonally Relaxed Orthogonal Projections Algorithm. DROP, also known as Diagonally Relaxed Orthogonal Projections, is a component averaging technique that makes use of orthogonal projections onto hyperplanes rather than the generalized oblique projections employed in the BICAV algorithm. It is proposed in [2].

$$x^{k+1} = x^k + \lambda_k U_{t(k)} \left(\sum_{i \in I_t(k)} w^k(i) \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \right) \quad (1.7)$$

where $U_{t(k)} = \text{diag}(\min(1, 1/s_l^t))$ with s_l^t as defined in BICAV and $\{\lambda_k\}_{k=0}^\infty$ is a sequence of user-determined relaxation parameters.

Both the DROP and BICAV algorithms are computationally more expensive than the BIP method because of the need to calculate the s_l^t 's prior to any image updates. However, it is the goal of component-dependent weighting to markedly improve the initial convergence pattern of the algorithm, which may compensate for time spent on extra calculations.

1.1.2.4 *Simultaneous Algebraic Reconstruction Technique.* [3] developed a block-iterative technique called simultaneous algebraic reconstruction technique (SART). The authors suggested the use of SART with blocks, which the authors called “subsets”, made up of image projection rays from a single projection angle and in doing so, found that SART was able to deal well with noisy data. The algorithm was developed in such a way that it was equally applicable to subsets, or blocks, of any composition as it was to subsets comprised of rays from a single projection angle. This block-iterative form, called ordered subsets simultaneous algebraic reconstruction technique (OS-SART) by [], is as follows.

$$x^{k+1} = x^k + \lambda_k \left(\frac{1}{\sum_{i \in I_t(k)} 1} \right) \sum_{i \in I_t(k)} \frac{b_i - \langle a_i, x^k \rangle}{\sum_{j=1}^n a_j^i} a_j^i \quad (1.8)$$

where $\{\lambda_k\}$ is a sequence of user-determined relaxation parameters.

1.1.3 String Averaging

Similar to BIP, SAP arrange all hyperplanes into blocks, here we call string. However, for each blocks, SAP perform serial ART algorithms. The results of each blocks then averaged to produce a final result.

$$y^{j+1} = y^j + \lambda_j \frac{b_i - \langle a_i, y^j \rangle}{\|a_i\|^2} a_i^T \quad (1.9)$$

$$x^{k+1} = \sum_{i=1}^M w_i y^i \quad (1.10)$$

The convergence behavior is shown in Figure 1.4, Figure 1.5, Figure 1.6 and Figure 1.7.

1.1.3.1 *The Component-Averaged Row Projections Algorithm.* This algorithm, also known as CARP, is developed by []. It differs with SAP in the way

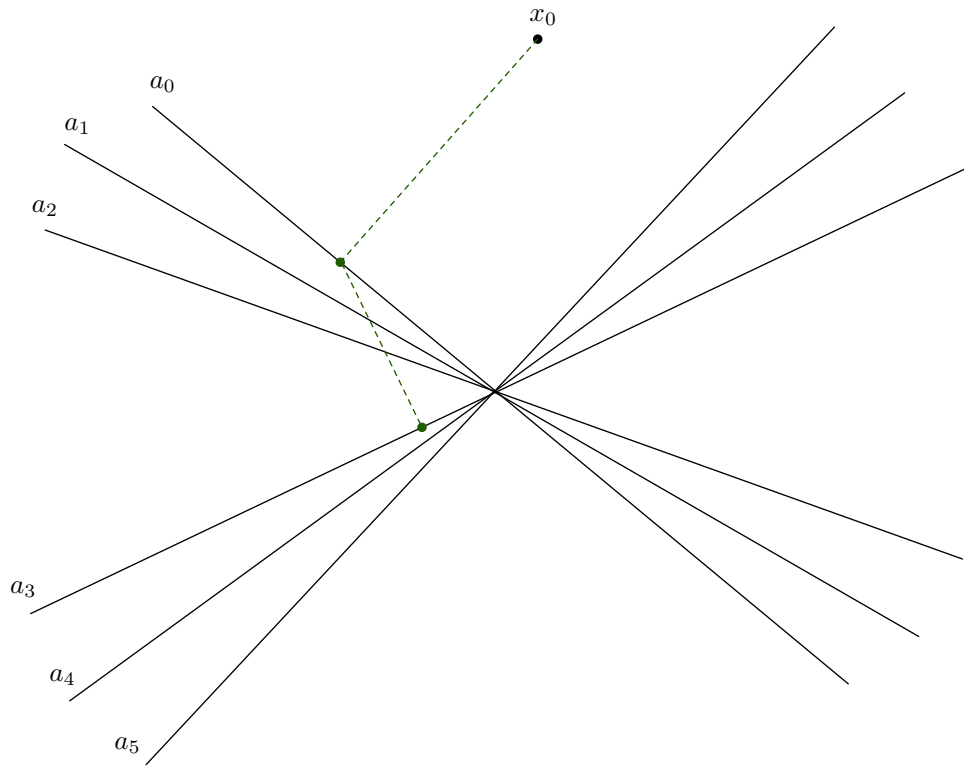


Figure 1.4: SAP path 1

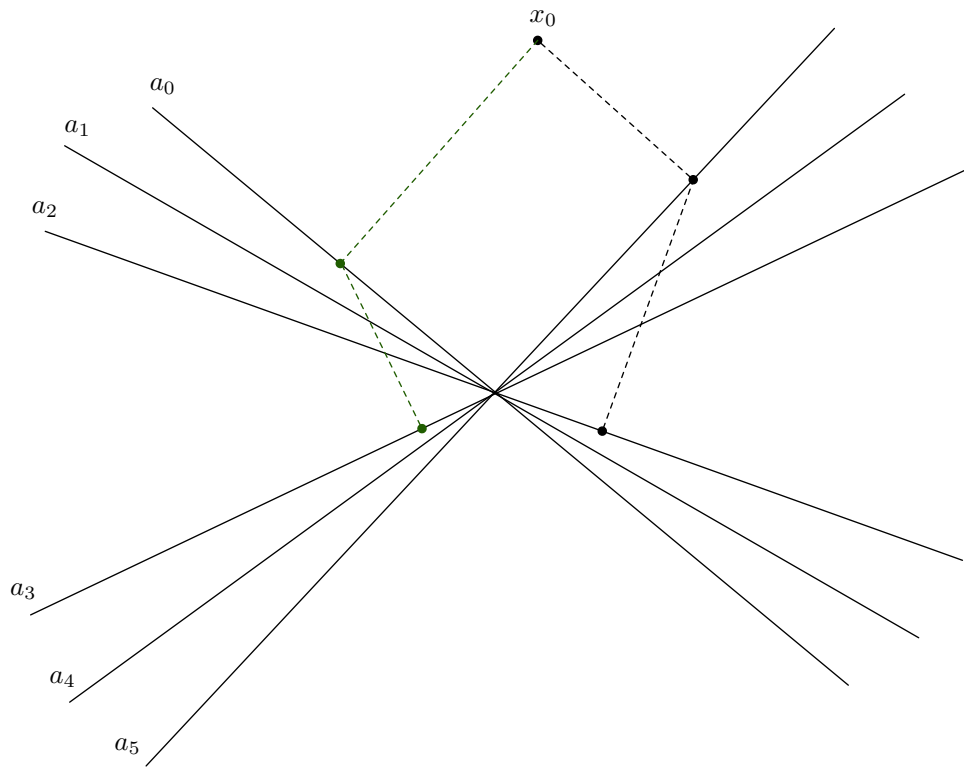


Figure 1.5: SAP path 2

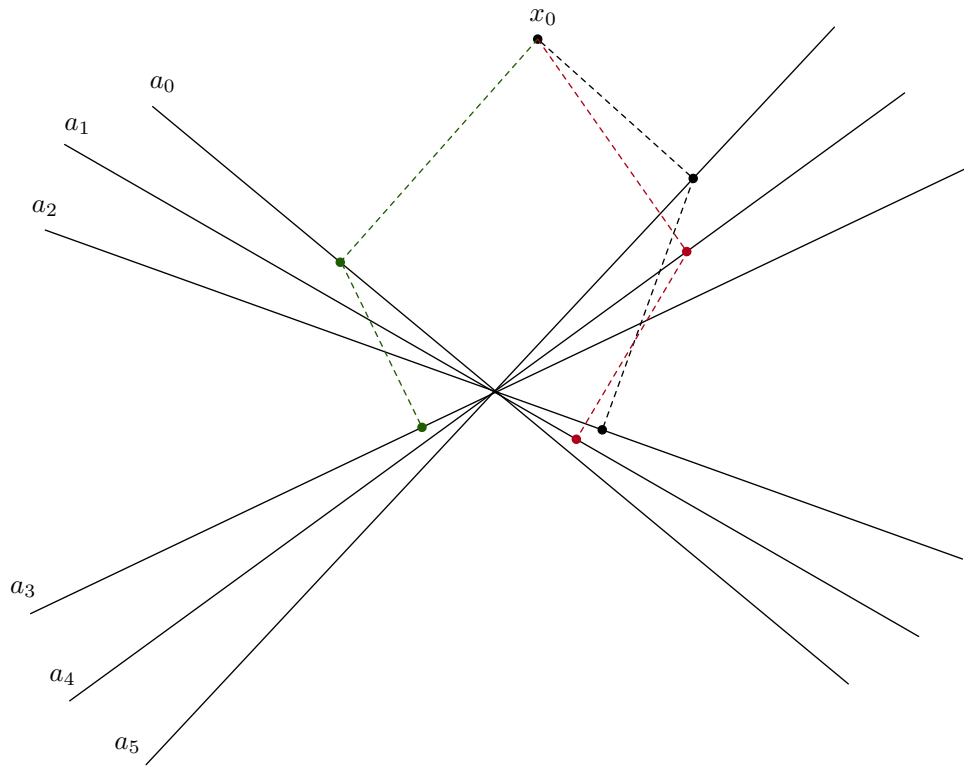


Figure 1.6: SAP path 3

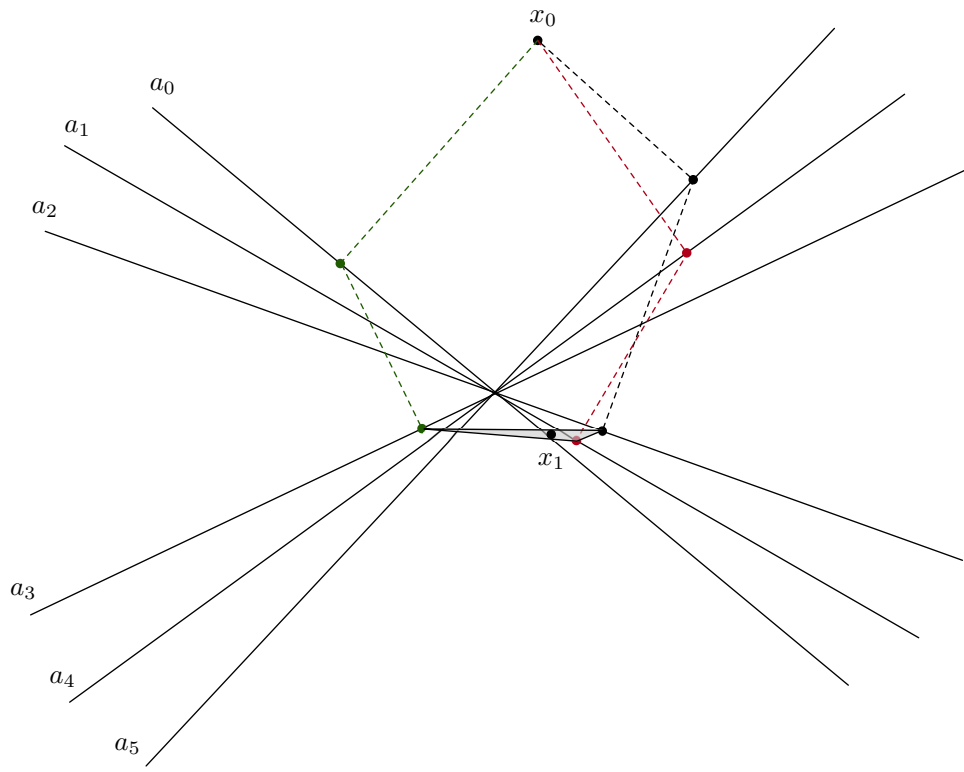


Figure 1.7: Average of three paths

averaging different y^i . Instead of weighted average in Equation 1.10, CARP uses the following scheme:

$$x_j^{k+1} = \frac{1}{s_j^t} \sum_{t=1}^M y_j^t \quad (1.11)$$

where s_j^t is the number of strings which contain at least one equation with a nonzero coefficient of x_j .

1.1.4 Summary

Overall, ART is the basic version of the projection methods here, while others are derived from ART and are suitable for different platforms. For example SAP is suitable for running on multi-core machines due to its long “string” block, while BIP is more suitable on modern GPU due to its multi-projecting feature. However, ART still have a advantage over both of these class of methods because every projection of ART take in all previous projection into consideration. So if done correctly, ART should be able converge faster in one iteration.

1.2 Concurrency

Concurrency is composition of independently executing processes, it is about dealing with a lot of things at once. These processes may or may not be related, and may or may not be running simultaneously. Concurrency gives you a way to structure program into independent pieces, and then you have to coordinate those pieces. To make that work, you need some form of communication. Communications are essential in a concurrent structure, in order for concurrency to work, you must have communication.

1.2.1 Concurrency vs Parallelism

Parallelism is simultaneous execution of multiple things (processes), possibly related, possibly not, it is about doing a lot of things at once. In terms of runtime

performance, parallelism determines how much faster a program can execute when more computing resources are available. This is very important because CPU clock speed has not improved much over the last decade. However, the number of cores on both CPU and GPU has increased dramatically. So the more computing resources a program can use effectively and efficiently, the more performance it can obtain from modern day hardware.

Concurrency and parallelism are related but separate ideas. One is about structure, and the other is about execution. Concurrency is a way to structure things so that you can, maybe, use parallelism to do a better job. Parallelism is not a goal of concurrency. Concurrency's goal is a good structure.

A good way to solve a problem is to break the problem down into independent components that you can separate and get right then compose and solve the problem altogether. Don't worry about parallelism. If the concurrency is right, the parallelism is just a free variable we can adjust.

1.2.2 Processes

A good example of concurrency is the processes running in a computer. In a running computer, usually there are hundreds of running processes. There are much more running processes than available CPU. Each process has a virtual memory space that it can see and address to. On a 64-bit operating system, regardless of the size of the real memory, each process has about 256TB (48-bits) of virtual memory space. This space is split into pages of 4K bytes and mapped into the physical memory. Each process can only access what is in their memory space, it cannot see what's outside its memory space. Therefore, each process is virtually independent of each other, the concurrency here is pretty simple. Since only one process can run at a time per CPU, each process is also associated with a process state for OS to manage. Figure 1.8 shows how processes transit between different states. When a process is loaded in ready to run in the main memory, there are three important states:

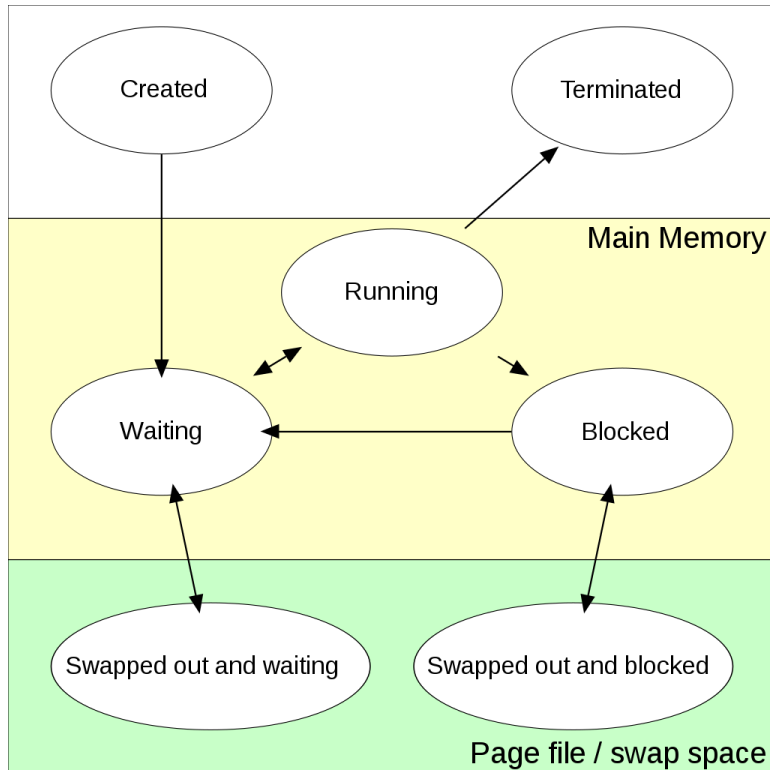


Figure 1.8: Process states

Running Only running processes can be in this state.

Blocked Processes are waiting on resources to be available in order to continue to run.

Waiting Processes is ready to start or continue to run.

1.2.2.1 Threads. Conceptually, a thread is an independent worker dedicated to work on one or a sequence of tasks. In a more computer science technical term, a thread is referring to a sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. Threads is one of the most fundamental tool implement parallelism in concurrency. One thread means there is only one thread of execution of instruction, which indicates all the instructions are issued in sequential order. There still may be some instructions are executed at the same time or out of order due to some optimizations by compiler or accelerations by the hardware, but for the most part they are sequential.

On the other hand, multi-threaded means there are multiple threads of execution are happening.

1.2.2.2 Context Switch. Context switch is referring to the activities taking place when a process transit from running to not running, either blocked or waiting. During context switch, it usually involves swapping register states, invalidates caches lines and if two threads, and if two threads that are swapping belongs to two different processes, it also means flushing the translation lookahead buffers(TLB) which is used to translate between virtual memory address and physical memory address.

1.2.2.3 Parking. Parking is term used to describe context switch between “green threads”. Green threads, also called “light weight thread”, are threads or tasks managed by a runtime environment. To avoid, expensive OS context switch, the runtime swapping in and out tasks on a running OS thread. When a green thread is swapped out of a running OS thread, it’s called parking.

1.2.2.4 Process vs Thread. A process is an instance of a computer program that is being executed by the operating system. It sits in the memory with other components, such as kernel and library codes, in a virtual memory space that runs from 0 to the maximum limit supported. We call this memory space ”virtual” because it does not represent the actual physical memory location. The virtual memory is mapped to physical memory by the operating system. Every program has the same virtual memory layout, e.g., kernel space, text (instructions), data, stack, heap, etc., and they can only access data or instructions in their own virtual memory space. There are several ways to communicate between processes, but they all have to go through series of system calls and quite expensive, definitely not something that should be done frequently. Especially on a multi or many core system, since there is only one kernel running, too many system calls for different programs or processes may become a bottleneck for them in terms of performances.

Operating system's process management is a good example of concurrency model. There are many processes are being executed at the same time, only a few running depending number processors available.

On the other hand, there are threads. Threads are different segments of code from the same virtual addressing space but have their own execution stack. Each thread can run on a different CPU core independently, and since they all share the same addressing space, they can understand each other's data much easier and more efficiently. For example, an object can be shared between threads without copying by simply passing the reference around.

1.2.3 Locks

In computer science, a lock is a mechanism that guards access to a resource with mutable state. The goal of this mechanism is the achieve mutual exclusion, so that no two threads are modifying a resources at the same time. Modifying a resources by multiple threads can cause unpredictable result. Typically, a lock is declared together with a resource it's trying to guard. Every thread that that tries to use the resource must try to acquire the lock first. As soon as the task on the resource is finish, the lock is released. When acquiring the lock, if the lock is already acquired by another thread, the thread acquiring the lock need to wait for the lock to be available.

There are various wait behaviors. A thread can simply loop until the lock becomes available. This is called spinlock, it is often used when the tasks on the resources are known to take very short time to finish. Alternatively, if the operation time on the resource is long or unknow, the thread acquiring the lock will be suspended and put into a wait queue. This behavior is called blocking and the lock is called monitor, first create by [4].

In concurrent programming, locks are used every where, implicitly or explicitly to make sure the consistency and correctness of the program running with multiple

threads. Synchronization is an example of using locks implicitly, [5] discusses synchronization with locks in extensive details.

1.2.4 Data Structures

1.2.4.1 Immutable Data Structure. Locks are only used when there is potential race condition. One way to avoid race condition is to eliminate write. If a variable state is not modified during its life time, nothing changes, therefore it is impossible to have inconsistency unless there is a hardware failure. Immutable data structure is a way to make all variables declared immutable even if you try to modify it. An example of this can be illustrated using a linked list with Figure 1.9. Variable *A* is an immutable linked list with value of 1, 2, 3, 4, 5 as shown in the figure. There is not write operation that can modify the list itself; However, there are operations that can give you a new list with desired effects. Calls to update the first element will return a new list with head updated without modifying original list. So the request of updating 1 to 100 on variable *A* will return a new variable, let's call it *B* shown in Figure 1.9. Part of *B* and *A* are shared, while the first element of the two lists are distinct, and the most importantly variable *A* is untouched. So the update request can come from a different thread, both threads get to keep their own version of the data with minimal modification. The concept can be extend from this simple linked list to tree which is used in implementing most immutable data structures.

The obvious draw back of immutable data structure is that they are not as fast as there are much more overhead when updating. Also read speed is also a bit slower due the much more complicated structure can have performance impact when navigating the data structure.

1.2.4.2 Concurrent Data Structures. Another commonly used data structure is so called synchronized data structure. They are thread safe version of single

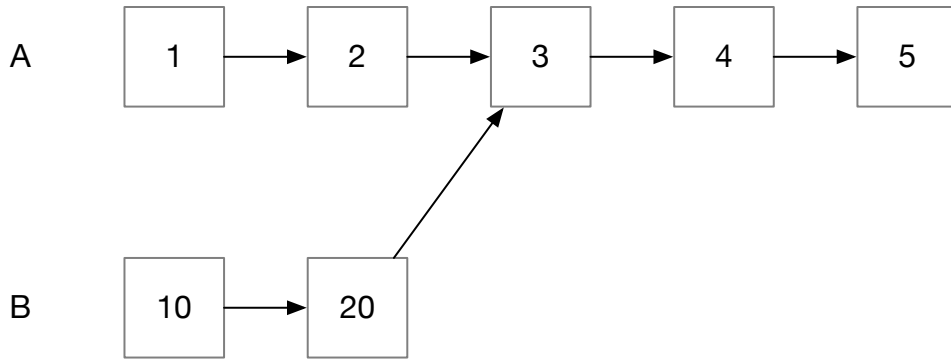


Figure 1.9: Immutable Data Structure

threaded data structure. For example, on JVM, *java.util.concurrent* contains a lot of concurrent data structures corresponding to the ones that can be found in *java.util*.

1.2.4.3 Channel. Channel is used when passing data between threads, first introduced in [6]. It has the following features

- Basic behavior of a channel is that of a queue, first in first out. Only the item at the beginning of the channel is available for access.
- There is a customizable but limited capacity on each channel.
- To use an item, it must be removed from the channel.
- Any thread that has the channel can put or take from the channel. Put means adding a new item to the channel, take means remove an item from the channel.
- If a thread tries to put a new item into a channel that is already full, or try to take from an empty channel, the thread will block. It will continue when another thread performs a take or put so that the blocked operation can continue.

The communication between threads is a form of message passing.

1.2.5 Concurrency in ART Algorithms

As an example, Figure 1.10, Figure 1.11 and Figure 1.10. Illustrates concurrency nature in ART, BIP and SAP respectively. Bubbles in these figures, can be

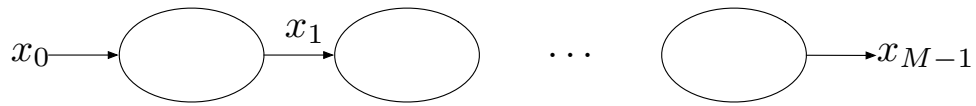


Figure 1.10: Concurrency in ART

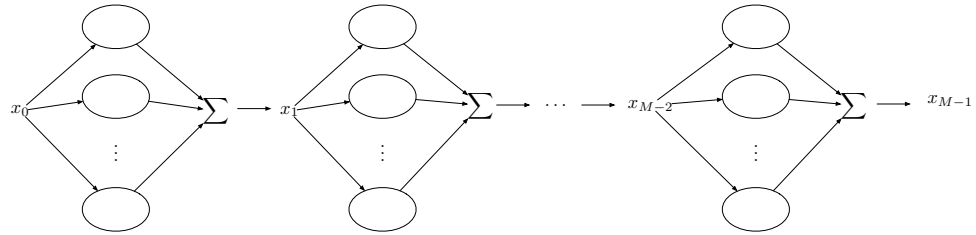


Figure 1.11: Concurrency in BIP

considered as a thread, while arrows indicates workflow. Although ART is a sequential algorithm, you can still use concurrency to describe its workflow nature.

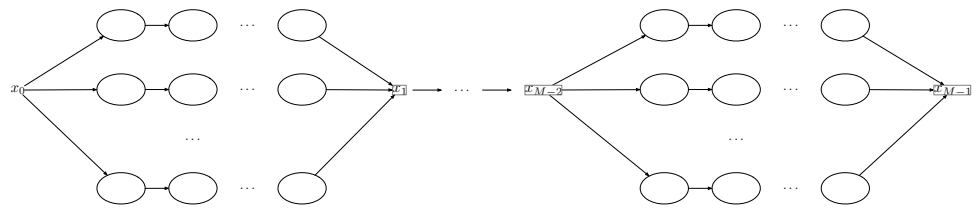


Figure 1.12: Concurrency in SAP

CHAPTER TWO

Theory

2.1 *Asynchronous Art Algorithm*

We are proposing a implementation of ART algorithm by exploring concurrency features in PCT dataset. The proton histories used in PCT dataset should always travel consecutive 3D image slices, i.e. there should be no skips, no slice se- quence such as 1, 2, 4. For those histories that do skip slices, they should be considered as erroneous, and not used for reconstruction. We can then group all the histories based on length and stack them up as shown in Figure 2.1, and perform reconstruction based on the order indicated in the same figure as well.

In Figure 2.1, $|a|$ is the length of a proton history in term of 3D image slices, i.e. the number of slices a proton beam has traveled. Arrows indicate the order reconstructions. For example, both $a[0]$ and $a[1]$ have an arrow points to $a[0, 1]$. This indicates that in order to perform reconstruction using data from histories that goes through slice 0 and 1, reconstruction using histories from slice 0 and 1 must be completed first. For blocks that are on the same rows in Figure 2.1, they are not depend on each other, reconstructions using these blocks can be performed in any order relative to each other. During run time, a dedicated thread of reconstruction for all the blocks will be launched simultaneously. Each thread will need to resolve dependency first before begin reconstruction on its data block. The final result will be collected from the threads handling $|a| = 1$.

So the algorithm can be described with the two parts:

- (1) Setup
 - (a) Sort all the histories by their history length in terms of 3D image slices.
 - (b) Stack them as shown in Figure 2.1 and setup the dependencies.

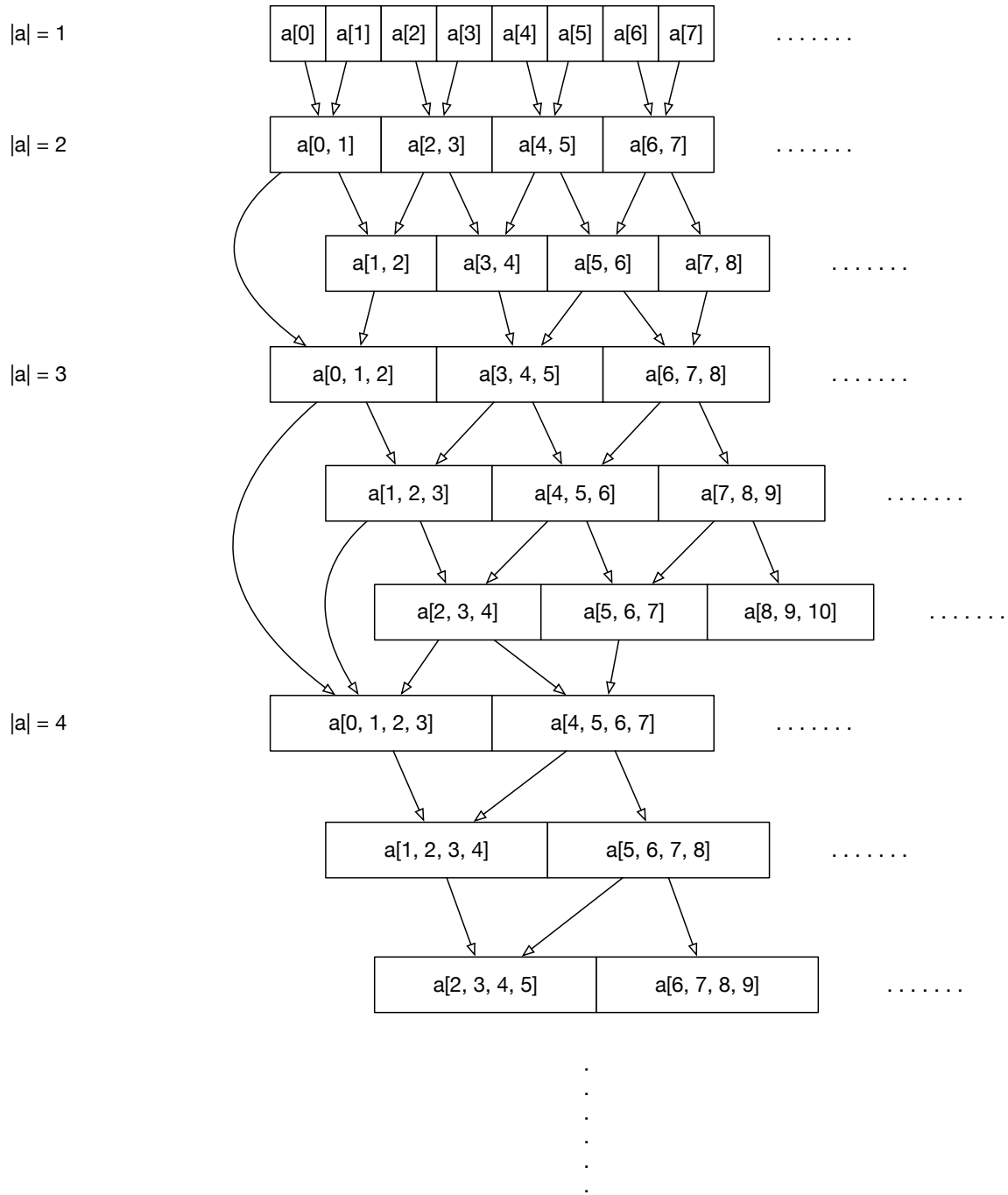


Figure 2.1: ART concurrency stack

- (c) Launch one reconstruction thread per block
- (2) Threads
- (a) Wait for only threads based on the block's dependencies to finish.
 - (b) Once dependencies have finished their reconstructions, proceed with local reconstruction using only the data block assigned to this thread.
 - (c) Back to step 2a for more iterations until the desired number of iterations have completed.

2.2 *

Proof In this section, we are going to show that the asynchronous art implementation is strictly equivalent to sequential art implementation.

Algebraic reconstruction technique (ART):

$$x^{k+1} = x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \quad (2.1)$$

We propose an order of $\{a_0, a_1, \dots, a_{n-1}\}$ for ART, where parts of the set can be projected out of order.

Lemma 1 *Any two adjacent rows in the order of projection, can be computed independently if the two rows are orthogonal to each other.*

Proof:

$$\begin{aligned} x^{k+1} &= x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \\ x^{k+2} &= x^{k+1} + \lambda_{k+1} \frac{b_{i+1} - \langle a_{i+1}, x^{k+1} \rangle}{\|a_{i+1}\|^2} a_{i+1}^T \\ &= x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T + \lambda_{k+1} \frac{b_{i+1} - \langle a_{i+1}, x^{k+1} \rangle}{\|a_{i+1}\|^2} a_{i+1}^T \end{aligned} \quad (2.2)$$

where

$$\begin{aligned}\langle a_{i+1}, x^{k+1} \rangle &= \left\langle a_{i+1}, x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \right\rangle \\ &= \langle a_{i+1}, x^k \rangle + \left\langle a_{i+1}, \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T \right\rangle\end{aligned}$$

Since

$$\lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} = K$$

where K is a scalar

$$\langle a_{i+1}, x^{k+1} \rangle = \langle a_{i+1}, x^k \rangle + K \langle a_{i+1}, a_i^T \rangle$$

If a_i and a_{i+1} are orthogonal to each other, then

$$\begin{aligned}\langle a_i, a_{i+1}^T \rangle &= 0 \\ \langle a_{i+1}, x^{k+1} \rangle &= \langle a_{i+1}, x^k \rangle + K0 = \langle a_{i+1}, x^k \rangle\end{aligned}$$

So, Equation 2.2 becomes

$$x^{k+2} = x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T + \lambda_{k+1} \frac{b_{i+1} - \langle a_{i+1}, x^k \rangle}{\|a_{i+1}\|^2} a_{i+1}^T \quad (2.3)$$

Now if we split x^k into x_1^k and x_2^k , so that

$$\begin{aligned}
x_1^k + x_2^k &= x^k \\
\langle x_1^k, x_2^{kT} \rangle &= 0 \\
\langle a_i, x_1^k \rangle &= \langle a_i, x^k \rangle \\
\langle a_i, x_2^k \rangle &= 0 \\
\langle a_{i+1}, x_2^k \rangle &= \langle a_{i+1}, x^k \rangle \\
\langle a_{i+1}, x_1^k \rangle &= 0
\end{aligned}$$

Then Equation 2.3 becomes

$$\begin{aligned}
x^{k+2} &= x_1^k + x_2^k + \lambda_k \frac{b_i - \langle a_i, x_1^k \rangle}{\|a_i\|^2} a_i^T + \lambda_{k+1} \frac{b_{i+1} - \langle a_{i+1}, x_2^k \rangle}{\|a_{i+1}\|^2} a_{i+1}^T \\
&= \left(x_1^k + \lambda_k \frac{b_i - \langle a_i, x_1^k \rangle}{\|a_i\|^2} a_i^T \right) + \left(x_2^k + \lambda_{k+1} \frac{b_{i+1} - \langle a_{i+1}, x_2^k \rangle}{\|a_{i+1}\|^2} a_{i+1}^T \right) \quad (2.4)
\end{aligned}$$

In this form we can calculate x^{k+2} by calculating projection of x_1^k to a_i and x_2^k to a_{i+1} independently and then sum them up. ■

Lemma 2 *Any group of adjacent rows in the order of projection, can be computed out of order if they are mutually orthogonal to each other.*

Proof:

If a group of adjacent rows are mutually orthogonal to each other, that means any two rows are orthogonal to each other as well. Any permutation of order of execution of this group can be obtained by sequence of swapping two adjacent rows, thus the resulting permutation should also be the same as any other permutation of this group. ■

Theorem 1 *Any group of projections can be executed independently or in any order if they are mutually orthogonal to each other.*

Proof: Since any two adjacent groups are mutually orthogonal to each other, using Lemma 2, we can shuffle the group order by swapping two group at a time. We shall obtain any order we want this way. Thus, every group can be executed independently. ■

2.3 Theoretical Speedup

2.3.1 Data Distribution

The theoretical speedup can be measured based on data distribution over different layers. Figure 2.2 shows how a sample dataset could distribute over various layers. The speedup for each layer in Figure 2.2 can be assumed as n , where n is the number of concurrent blocks on that layer. But the contribution of speed up from that layer depends on that percentage of histories on that layer. Thus, the speed up of Figure 2.2, can be can be computed as follows:

$$\begin{aligned}
 \text{speedup} &= 0.3442 \times 15 \\
 &\quad + 0.2446 \times 8 + 0.2317 \times 7 \\
 &\quad + 0.0599 \times 5 + 0.0458 \times 5 + 0.0438 \times 4 \\
 &\quad + 0.0089 \times 4 + 0.0078 \times 3 + 0.0070 \times 3 + 0.0062 \times 3 \\
 &= 9.5
 \end{aligned}$$

So roughly 9.5 times speedup without considering communication overhead. For convenience, Figure 2.2 will be shown as Table 2.1.

However, this “theoretical speedup” is simply an estimation for the potential of concurrent underlying structure. The basic assumption when calculate this speedup is that in each layer, data are distributed evenly across the blocks in that layer. The true distribution varies and can be observed from the history count data in the

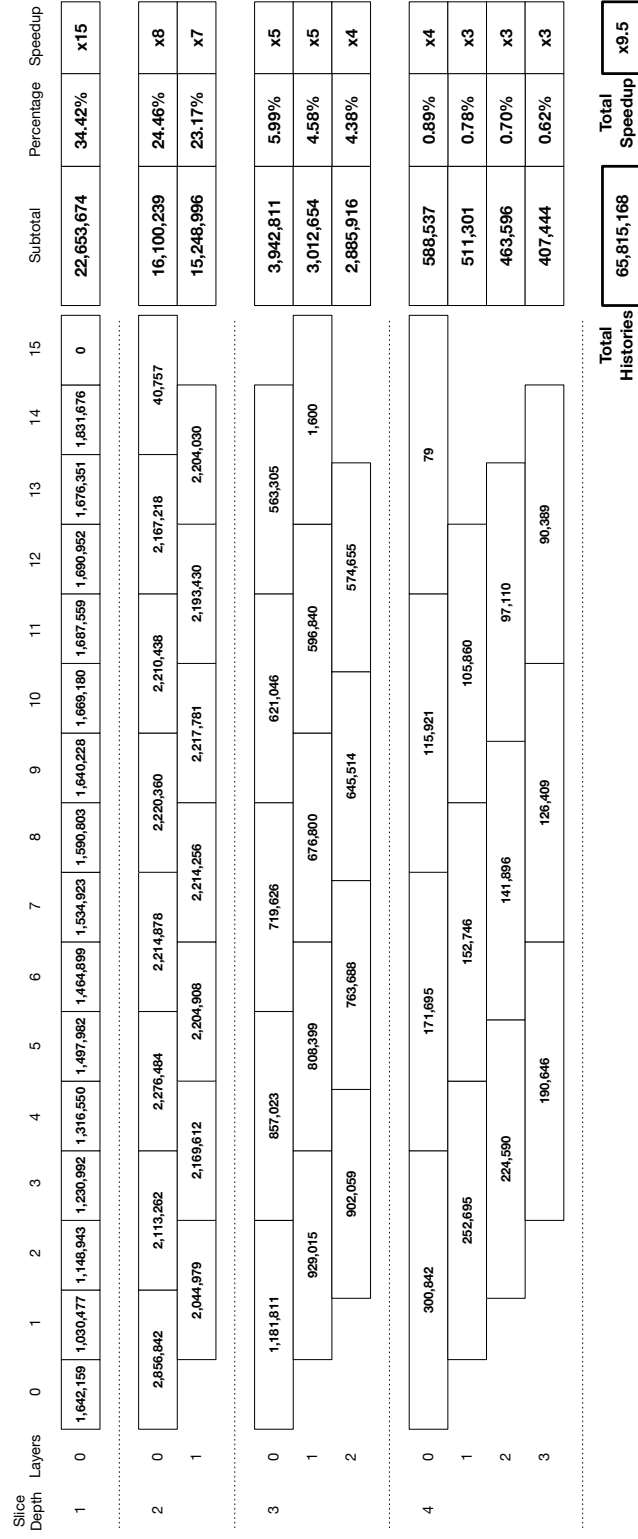


Figure 2.2: Experimental data distribution per block

Table 2.1: Theoretical Speed up for sample dataset

layer index	subtotal	%	blocks	speedups
(1 0)	22,653,674	34.42%	15	5.16
(2 0)	16,100,239	24.46%	8	1.96
(2 1)	15,248,996	23.17%	7	1.62
(3 0)	3,942,811	5.99%	5	0.30
(3 1)	3,012,654	4.58%	5	0.23
(3 2)	2,885,916	4.38%	4	0.18
(4 0)	588,537	0.89%	4	0.04
(4 1)	511,301	0.78%	3	0.02
(4 2)	463,596	0.70%	3	0.02
(4 3)	407,444	0.62%	3	0.02
speedup				9.5

Appendix sections. Another implied assumption is that every history’s projection computation takes the same amount of time. This might not be the case, histories that have different depths might vary quite a bit. The major overhead that could prevent the real speedup to be anywhere close to the theoretical speedup would be the communication delay and context switch between threads, this delay is on the Operation System level, very hard to predict. Also, as dataset gets larger and larger, the depth of the async grid could get deeper as well. The deeper the grid, the narrower the bottleneck at the bottom of the grid gets, which limit the performance as well.

On the plus side, the algorithm is inherently asynchronous because the progress between layers are not fully synchronous. For every block of computation to start, it does not need to wait for the previous layer to fully finish, it only needs to wait for related blocks from that layer. This asynchronous nature balances out with some of the overheads.

Overall, “theoretical speedup” is a summary of the potential of a grid configuration.

2.3.2 Time Complexity

For a linear ART computation, assuming each slice has n voxels, time complexity should take the following factor into considerations:

- According to [7], it is estimated that we needed approximately 20 histories per voxels in order to construct a reasonable good image. Therefore, the histories we have per slices is roughly Kn , where K is a constant and $K > 20$.
- Since each history is a line through a slice, the number of voxels is roughly in the order of \sqrt{n} .

Since each projection of ART

$$x^{k+1} = x^k + \lambda_k \frac{b_i - \langle a_i, x^k \rangle}{\|a_i\|^2} a_i^T$$

is depending on the length of the history, so each projection calculation is $O(n^{\frac{1}{2}})$. A full iteration of ART requires linearly iterate and compute projection of all histories, the time complexity of a full iteration should be $O(n^{\frac{1}{2}} \times Kn \times M)$ where M is a constant representing the number of slices. So the time complexity of ART should be $O(n^{\frac{3}{2}})$.

CHAPTER THREE

Implementation

3.1 Dataset

ART can be considered as an iterative solver of a system of linear equations $Ax = b$, where

- A is a sparse $m \times n$ matrix whose rows are proton histories.
- x represent the voxels in the reconstructed 3D image, arranged as a vector
- b is a vector of corresponding proton energy detected for each proton history, i.e. rows in matrix A .

Since we are reconstructing a 3D image, each history of a proton is a path the proton has travel the in the 3D space. The history is represented by a series of voxels. Figure 3.1, and Figure 3.2 demonstration how a 3D path is marked, seralized and stored in a sparse format. The example demonstrate a history that passes through only 1 slice. It's also very common for a history to pass through multiple consecutive slices.

We group the histories by their starting index and the length in terms of number of slices it traveled through and put them into an index tree as shown in Figure 3.5. The index tree is essentially a two dimensional structure with first dimension is mapped the the starting index of histories and the second dimension is mapped to the slice length the each history. Every entry is a group of histories with the same length and starting index. This way a block can easily be referenced with minimal search time.

Figure 3.3 is an implementation of the concurrency illustrated in Figure 2.1 with total number of slices equals to 16 with maximum history length equals 5.

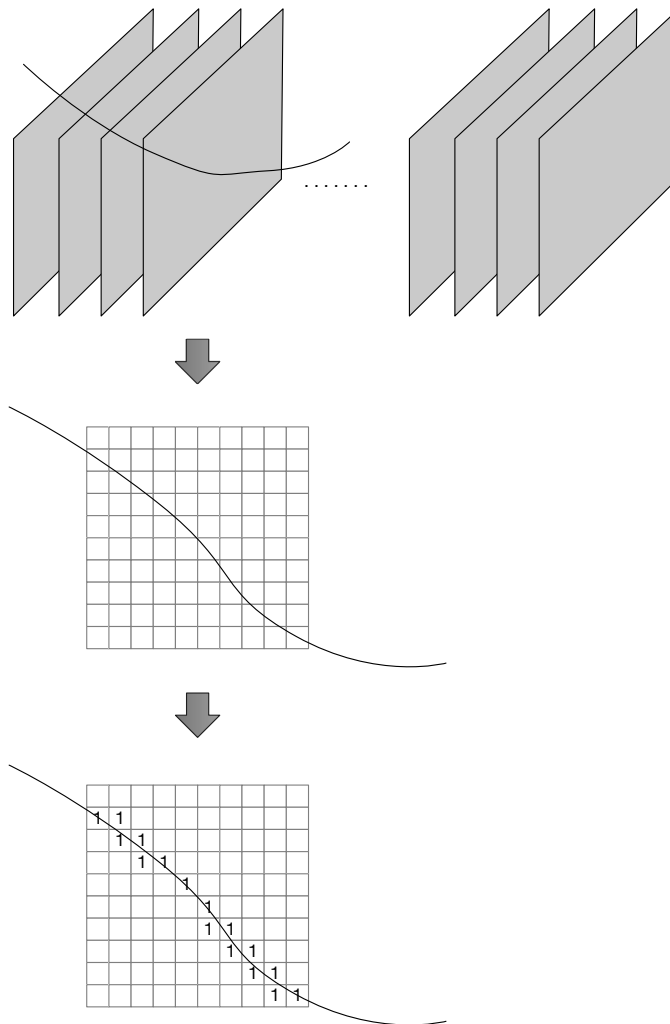



Figure 3.1: 3D images

1	1								
	1	1							
		1	1						
				1					
					1				
						1	1		
							1	1	
								1	1
									1

 **Serialize**

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Value											1	1										1	1		

Index	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
Value								1	1											1					


Index	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74
Value						1										1	1								

Index	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99
Value		1	1										1	1										1	1

 **Sparse Format**

Local index:

10	11	21	22	32	33	44	55	65	66	76	77	87	88	98	99
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

 + slice offsets = + 3 x 10 x 10

Global index:

310	311	321	322	332	333	344	355	365	366	376	377	387	388	398	399
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Figure 3.2: Data Format

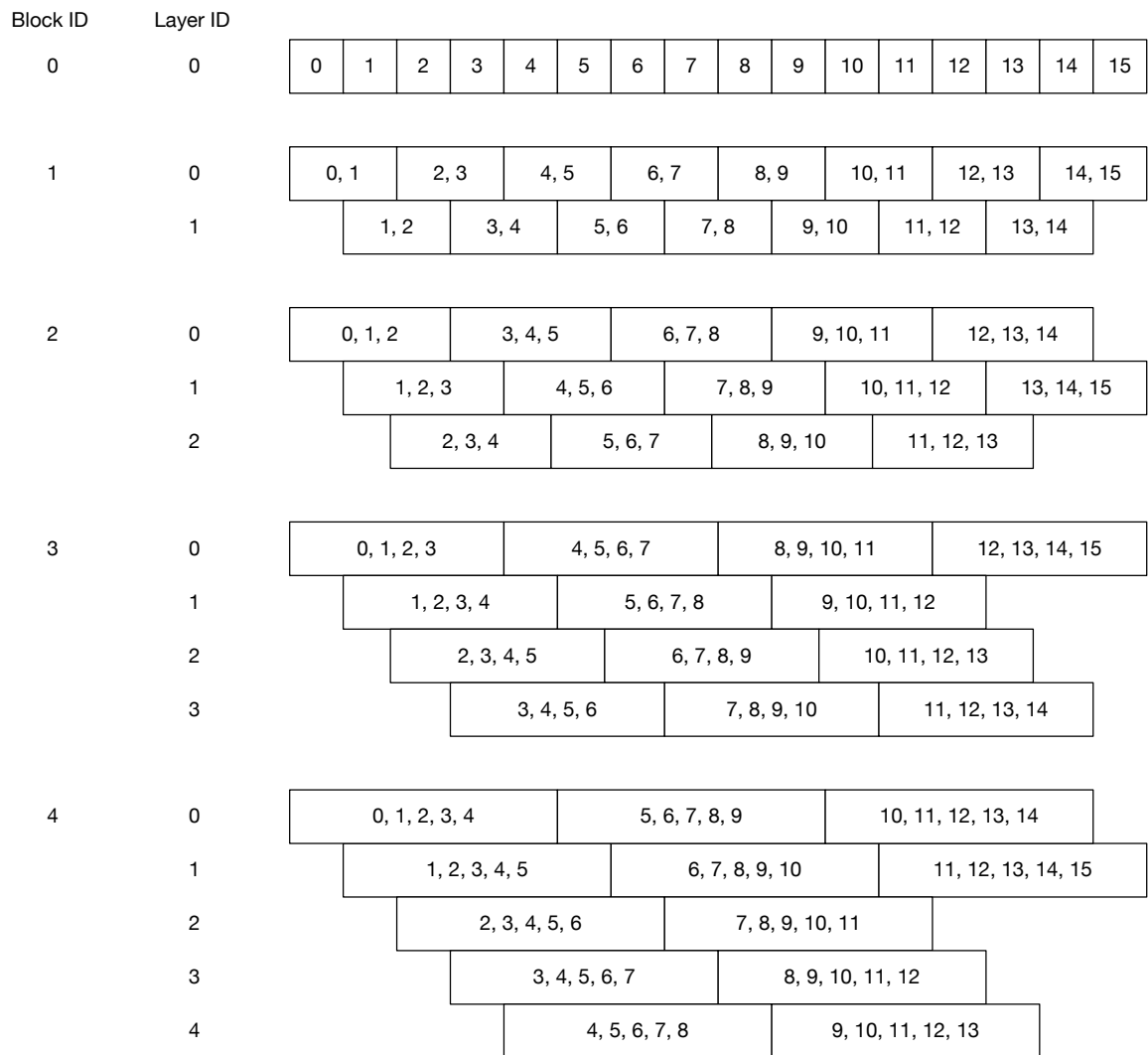


Figure 3.3: Art Concurrency Blocks

The max history length can go as deep as necessary but in our case 5 is enough to reconstruct a very good quality image.

Every block shown in Figure 3.3 is an asynchronous node used to construct the asynchronous grid. Every node must has at least following properties:

- Slices the node represent.
- A channel that accept data from upstream nodes
- A channel that can send data to downstream nodes.
- A channel that can send out final result.

3.2 *Sorting*

For testing this algorithm, before the reconstruction starts, dataset must be loaded from disk to memory and organized into a index tree which will be described in Section 3.3. Sorting structure is described in Figure 3.4. One thread is dedicated reading from disk and dispatch read data into a work queue. Worker threads queue on the work queue to fetch and work on tagging raw dataset.

3.3 *Index Tree*

The CTP404 dataset was used in this work is used in previous work [8], provided by Dr. Paniz Karbasi. They are stored in files in local storage. Dataset are loaded into the memory before reconstruction starts to simulate a real life scenario since, realistically, data will always be passed in memory rather than from disk IO.

The final final format of the data sorted in memory is shown in Figure 3.5. All histories are grouped by their starting slice index and the length of the history in term of slices. Within each group, i.e. histories that have the same starting index and number of slices, data are stored as an unsorted vector.

3.4 *Language and Environment*

The choice of language to implementation is Clojure. There are several reasons for this choice.

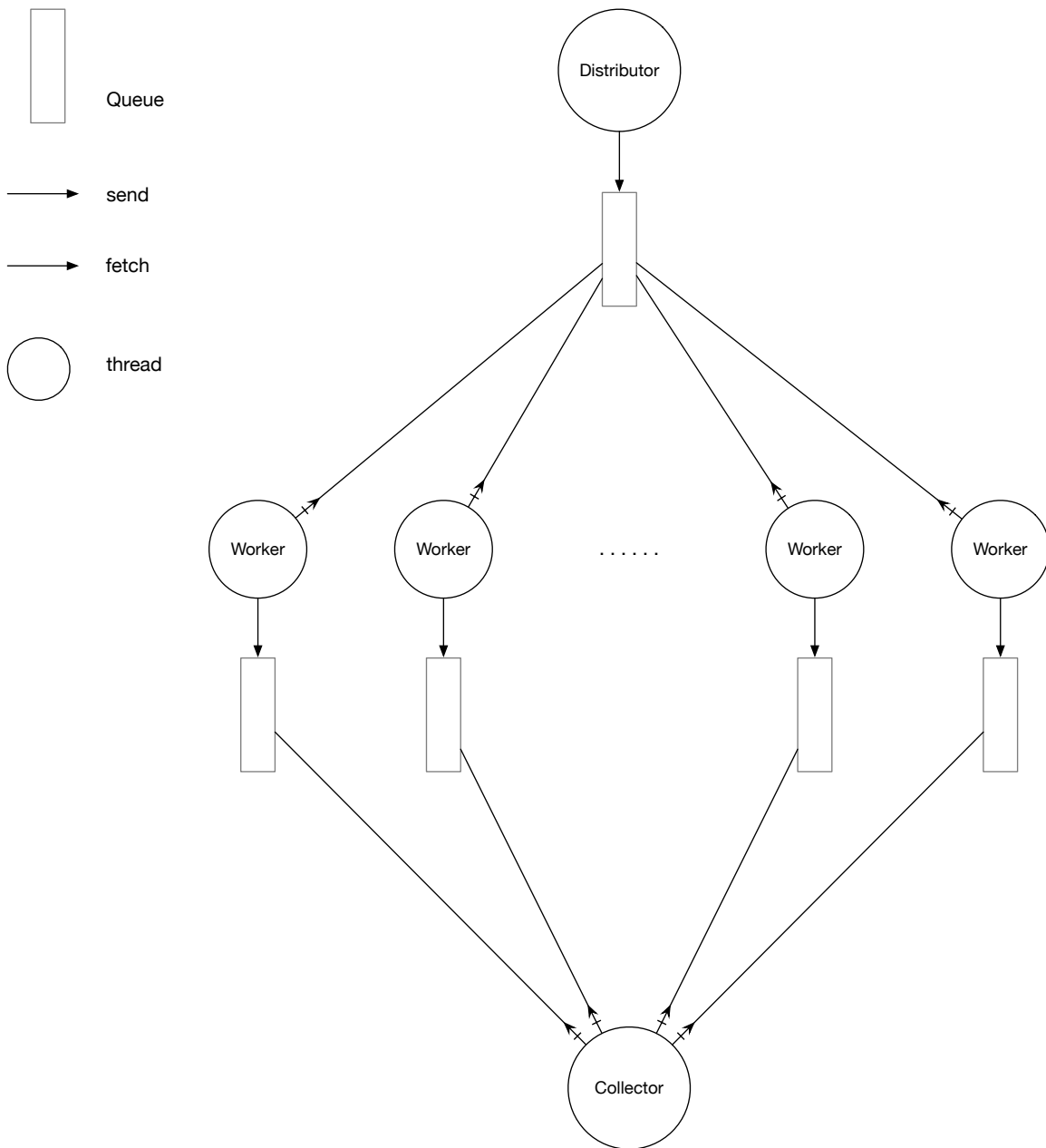


Figure 3.4: Data Format

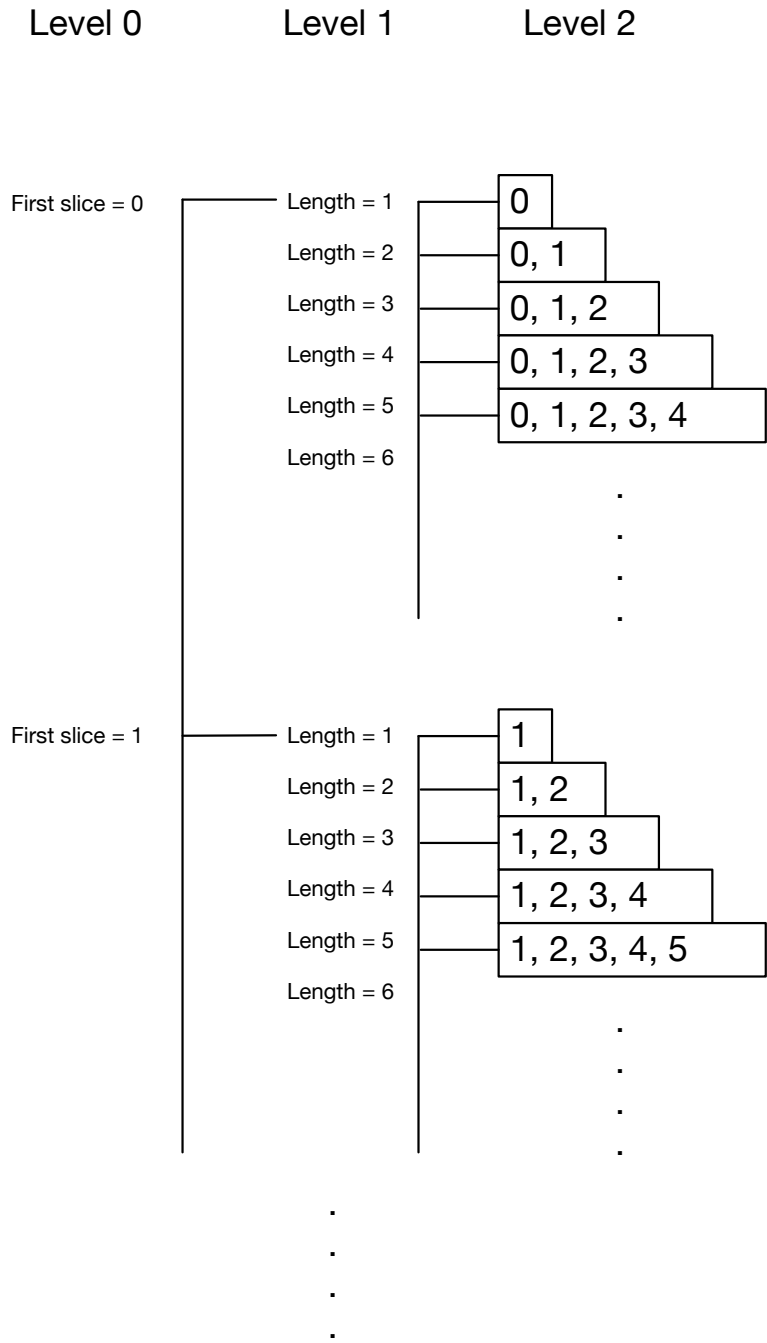


Figure 3.5: Index tree structure

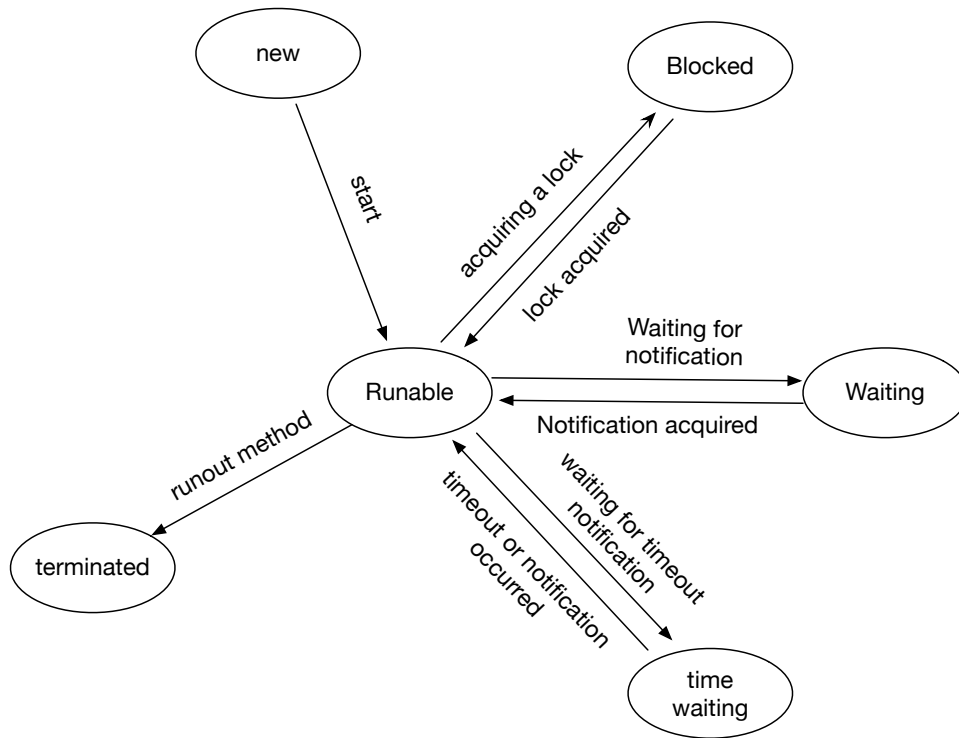


Figure 3.6: Java thread life cycle

First, Clojure is a language that runs on Java Virtual Machine (JVM). JVM has a very sophisticated system that handles multithreading. For instance, every object in JVM is implicitly associated with a monitor lock which can be used for synchronization. Also, JVM has a very advanced thread management system. JVM threads have a number of states, similar to the OS processes, except they are managed by the JVM instead of the Operating System. JVM state transitions are shown in Figure 3.6.

Apart from the multithreading features, JVM also provides a very diverse and versatile ecosystem. There are a lot of useful libraries available on JVM. Everything that can run on JVM can also be used in Clojure as well.

Second, Clojure is designed with concurrency in mind. Its fundamental container data structures are all immutable, which means any write operation on a immutable data structure is operated on top of the original data without modifying them. For example, when you are trying to add items to a immutable list, you will

get reference to a new list combining the original list and the new item, and the original list remain untouched. This model makes sharing data between threads are much more simpler.

Third, Clojure is one of the few languages that has very good support for CSP model. CSP stand for Communicating Sequential Processes, it's a mathematical process calculus for concurrency modeling. It is described by Tony Hoare in 1978. The basic idea behind CSP is that there are a number of independent processes that each execute some ordered sequence of steps. These processes can communicate with each other by sending or receiving messages over channels. When a process wants to read a message from a channel, it blocks until a message is available, then it consumes the message and moves on. A process can also place a message on a channel either synchronously or asynchronously. By using communication over channels, multiple processes can synchronize such that one process waits for a specific input from another before proceeding. The "process" referred in the concurrency model is an abstract conceptual idea, in an implementation they can be either Operating System processes or threads of a running program. In our case, we are using threads provided by the JVM environment for the process.

Fourth, Clojure provides a very dynamic developing environment making developing algorithms with large dataset much easier. With Clojure, I'm able to start up a developing environment on a remote server that has large enough memory to handle the dataset I'm experimenting and load the dataset. I can then connect to the server remotely from my much less powerful laptop and send updated algorithm to run on server remotely. Without this feature, I would have been much more difficult to progress the development of this algorithm.

3.5 *Async Grid*

To setup the concurrency we have described in our algorithm, we are going to setup a grid that represent the connections or dependencies in the graph.

3.5.1 Channels

A channel is an important concept in CSP. It is a conduit that can carry a value to one CSP process to another CSP process. In CSP, a *process* is a fundamental object. It is simply an anonymous (unnamed) piece of code that can execute a number of steps in order, potentially with its own control flow. Code in a *process* always runs synchronously - that is, the process will not proceed on to the next step until the previous step completes. So the CSP process is essentially a thread in our implementation here. And channels are used to communicate between threads. A put action is when a thread is trying to put a data onto the channel. A take action is when a thread is trying to take, or consume, a data from a channel. When a thread is trying to put but the channel is full or trying to take when the channel is empty, the thread can either be blocked or parked.

3.5.2 Blocking vs Parking

There are two types of threads in Clojure's core.async library. core.async thread create threads from JVM's CachedThreadPool, while go thread create threads from JVM's FixedThreadPool. CachedThreadPool create new threads as needed, but will reuse previously constructed threads when they are available. FixedThreadPool reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.

CachedThreadPool is used for performance intensive threads, e.g. threads that are doing reconstructions; while FixedThreadPool is used for background utility threads, doing works such as multiplexing values from one channel to other channels.

3.5.3 Async Node

An async node is a data structure that hold some essential data for a thread to execute independently without doing any global lookup or request. It includes the following attributes:

- List of slice indices: Indicate which slices this block / thread is handling.
- Key: A keyword generated from list of slice indices. It is used as a identifier for the data sent downstream. Can also be used as key for the look up table in the grid.
- Unused: Slices that are used by downstream nodes.
- Input channel: A channel a thread will listen on in order to get partially reconstructed data segment from feeding threads.
- Output channel: A channel a thread will send its partially reconstructed result to downstream threads
- Result channel: A channel that head threads will send their final reconstructed segment out.
- Mux: A multiplexing object that will move data from output channel to downstreams' input channels.
- Downstreams: Keep track of nodes that are listening on output channel, used for setting up connections.
- Upstreams: Keep track of nodes input channel is listening to.
- Local offsets map: A hashmap that maps keys of upstream nodes to relative position need to be used for local reconstruction
- Global Offset: Offset information used to copy data from and to global data.
- Type: There are two types of async node:
 - * Head: These are nodes control the start and finish of reconstruction. They will grab data segment from global data based on 'Global Offset',

initiate the reconstruction, then propagate partially reconstructed data to other nodes.

- * body: These are nodes continuously applies reconstructions as long as there are data sent.

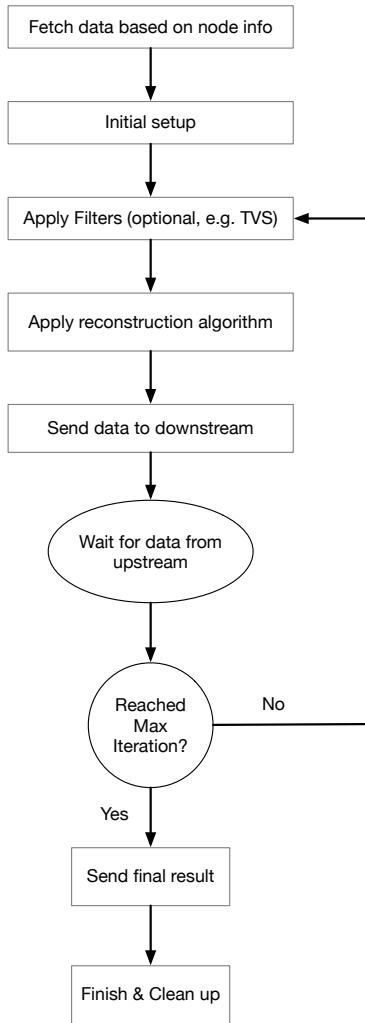
When reconstruction starts, each node will be assigned with a thread created from `CachedThreadPool`. Each thread will follow a slightly different steps based on their type as shown in Figure 3.7. Both type will fetch data from global index tree using keys derived from their “list of slice indices”, and shuffle them first before reconstruction starts. The head nodes will apply reconstruction algorithm before sending data to the downstreams, while body nodes will have to wait for the data from upstream before applying first iteration of reconstruction. The head nodes will also apply additional filters at the end of each iteration, while the body nodes do not. At the end of the reconstruction, the final results will be collected from head nodes.

3.5.4 Grid Generation

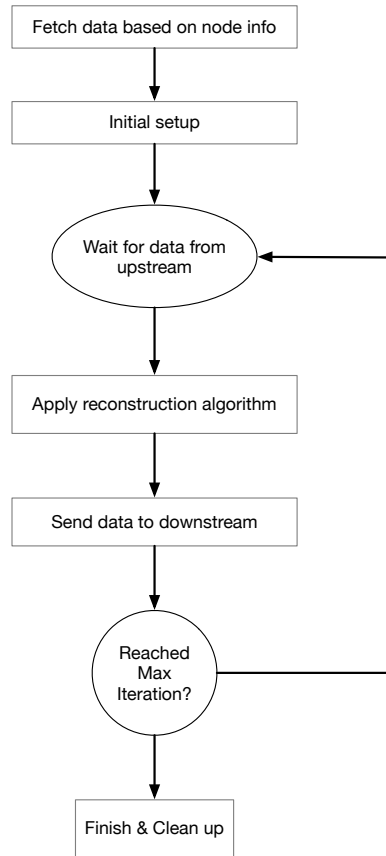
The grid is a interconnected async node. Each node will have a set of upstream nodes and downstream nodes. A node receives data from upstream nodes and send data to downstream nodes. There are many possible configurations of the grid, we choose the one that is described in Figure 3.8 where arrows indicate the direction of data flow.

When generating the grid, there are two major steps. First, we generate all the nodes we are going to use; then we iterate through all the nodes to make sure they are all connected in a desired way.

3.5.4.1 Grid Nodes Generation. Nodes are generated block by block. Each block contains all the nodes that covers the same number of continuous slices. Within each block, there are 1 or more rows. Each row contains maximum nodes that have no overlapping slices. An expected pattern is shown in Figure 3.9.



(a) Head Node Reconstruction Steps



(b) Body Node Reconstruction Steps

Figure 3.7: Reconstruction steps for different types of threads.

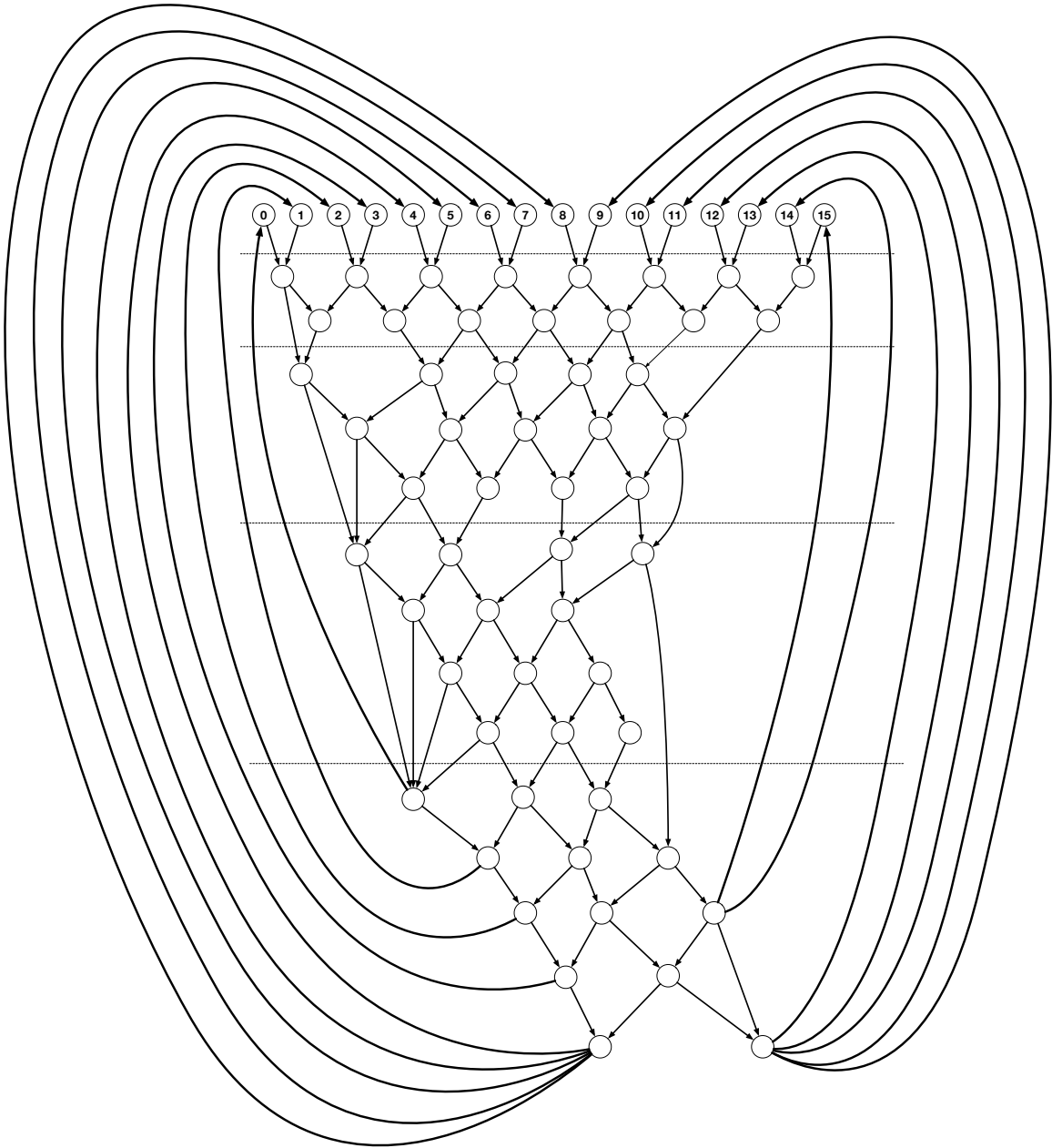


Figure 3.8: Async nodes connection in a grid

Block ID	Layer ID																
0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0, 1	2, 3	4, 5	6, 7	8, 9	10, 11	12, 13	14, 15								
	1	1, 2	3, 4	5, 6	7, 8	9, 10	11, 12	13, 14									
2	0	0, 1, 2	3, 4, 5	6, 7, 8	9, 10, 11	12, 13, 14											
	1	1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12	13, 14, 15											
	2	2, 3, 4	5, 6, 7	8, 9, 10	11, 12, 13												
3	0	0, 1, 2, 3	4, 5, 6, 7	8, 9, 10, 11	12, 13, 14, 15												
	1	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12													
	2	2, 3, 4, 5	6, 7, 8, 9	10, 11, 12, 13													
	3	3, 4, 5, 6	7, 8, 9, 10	11, 12, 13, 14													
4	0	0, 1, 2, 3, 4	5, 6, 7, 8, 9	10, 11, 12, 13, 14													
	1	1, 2, 3, 4, 5	6, 7, 8, 9, 10	11, 12, 13, 14, 15													
	2	2, 3, 4, 5, 6	7, 8, 9, 10, 11														
	3	3, 4, 5, 6, 7	8, 9, 10, 11, 12														
	4	4, 5, 6, 7, 8	9, 10, 11, 12, 13														

Figure 3.9: Blocks and layers of the grid

The pattern can be generated using snippet from Listing 3.1.

Listing 3.1: Generating all slice sequences

```
(let [length 16]
  (for [block-size (range 1 6)]
    (for [start-idx (range 0 block-size)]
      (partition block-size (range start-idx 16))))))
```

Two functions used here:

(range start end) returns a sequence of number from start (inclusive) to end (exclusive)

(partition n coll) returns a sequence of lists of partitions, n items each, without overlapping.

The exact result of Listing 3.1 is shown in Listing 3.2. Comparing it to Figure 3.9 they are identical in term of grouped slice indices.

Listing 3.2: Generated sequence

```
((((0) (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15)))
 ((0 1) (2 3) (4 5) (6 7) (8 9) (10 11) (12 13) (14 15))
 ((1 2) (3 4) (5 6) (7 8) (9 10) (11 12) (13 14)))
 ((0 1 2) (3 4 5) (6 7 8) (9 10 11) (12 13 14))
 ((1 2 3) (4 5 6) (7 8 9) (10 11 12) (13 14 15))
 ((2 3 4) (5 6 7) (8 9 10) (11 12 13)))
 ((0 1 2 3) (4 5 6 7) (8 9 10 11) (12 13 14 15))
 ((1 2 3 4) (5 6 7 8) (9 10 11 12))
 ((2 3 4 5) (6 7 8 9) (10 11 12 13))
 ((3 4 5 6) (7 8 9 10) (11 12 13 14)))
 ((0 1 2 3 4) (5 6 7 8 9) (10 11 12 13 14))
 ((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15))
 ((2 3 4 5 6) (7 8 9 10 11))
 ((3 4 5 6 7) (8 9 10 11 12))
 ((4 5 6 7 8) (9 10 11 12 13)))
```

A slightly different pattern can be generated using Listing 3.3, and result is shown in Listing 3.4

Listing 3.3: Generating all slice sequences

```
(let [length 16]
  (for [block-size (range 1 6)
        start-idx (range 0 block-size)]
    (partition block-size (range start-idx 16))))
```

Listing 3.4: Alternative sequence

```
((((0) (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15)))
 ((0 1) (2 3) (4 5) (6 7) (8 9) (10 11) (12 13) (14 15))
 ((1 2) (3 4) (5 6) (7 8) (9 10) (11 12) (13 14))
 ((0 1 2) (3 4 5) (6 7 8) (9 10 11) (12 13 14))
 ((1 2 3) (4 5 6) (7 8 9) (10 11 12) (13 14 15))
 ((2 3 4) (5 6 7) (8 9 10) (11 12 13))
 ((0 1 2 3) (4 5 6 7) (8 9 10 11) (12 13 14 15)))
```

```

((1 2 3 4) (5 6 7 8) (9 10 11 12))
((2 3 4 5) (6 7 8 9) (10 11 12 13))
((3 4 5 6) (7 8 9 10) (11 12 13 14))
((0 1 2 3 4) (5 6 7 8 9) (10 11 12 13 14))
((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15))
((2 3 4 5 6) (7 8 9 10 11))
((3 4 5 6 7) (8 9 10 11 12))
((4 5 6 7 8) (9 10 11 12 13))

```

Either format would work, with format in Listing 3.2 much easier to query for a particular row when testing and examining async node data manually.

In order to generate async nodes instead of just lists of slice indices, we simply just apply a function that generate async from given slice indices. Snippet is shown in Listing 3.5

Listing 3.5: Generating all async nodes

```

(let [length 16]
  (for [block-size (range 1 6)]
    (for [start-idx (range 0 block-size)]
      (map newAsyncNode
           (partition block-size (range start-idx 16))))))

```

3.5.4.2 Grid Connection Setup. There are a few basic rules when connecting nodes in the grid

- (1) Every node do not connect to any other nodes in the same row, since they are not supposed to have any overlapping slices.
- (2) Every node only try to connect with nodes from rows above it, a.k.a upstream, with the exception of top row when making loop back connections.

Since there is no previous row to connect to for the first row, we start making connection from the second row. When making connection for each row, we keep track of three variables:

- An iterator to the current row
- An iterator to the parent row which is the row immediately above current row
- An iterator to a stack of previous rows

Whenever we finished with a row, that row will be pushed onto a stack. So when working on the next row, the parent row for that row is always the first row on

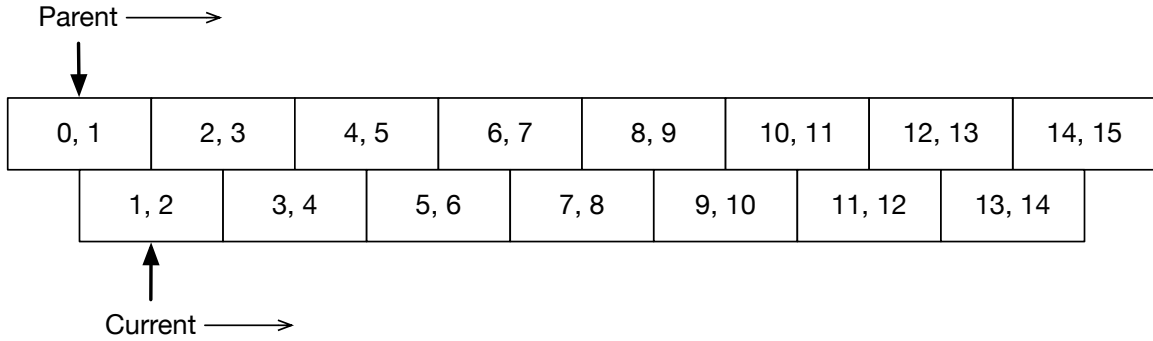


Figure 3.10: Iterators

stack. The current row iterator and parent row iterator relationship can be shown in Figure 3.10. The basic idea is to try to make connections between parent node and current node, if success and current node is not full yet, move the parent node; if the current node is full, or connection failed, move the current node. They keep moving until current row iterator reaches the end. Both iterators are moving from left to right as indicated in the figure.

When a downstream node (receiving node) is making connection with an upstream node (sending node), the following things will happen:

- Determine continuous overlapping slices between downstream node's slices and upstream node's unused slices. If there is none, no connection will be made and nothing will be changed in neither nodes.
- Overlapping slices will be added to upstream node's downstream map, and downstream node's upstream map.
- Downstream node's input channel will be added to upstream node's output multiplexer.
- Overlapping slices will be removed from upstream node's unused list.

One of the challenges when making connections are when there are gaps of coverages between rows, as shown in Figure 3.11, at both beginning and end of some rows. So when traversing through a row when making connections, we classify the whole path into three sections:

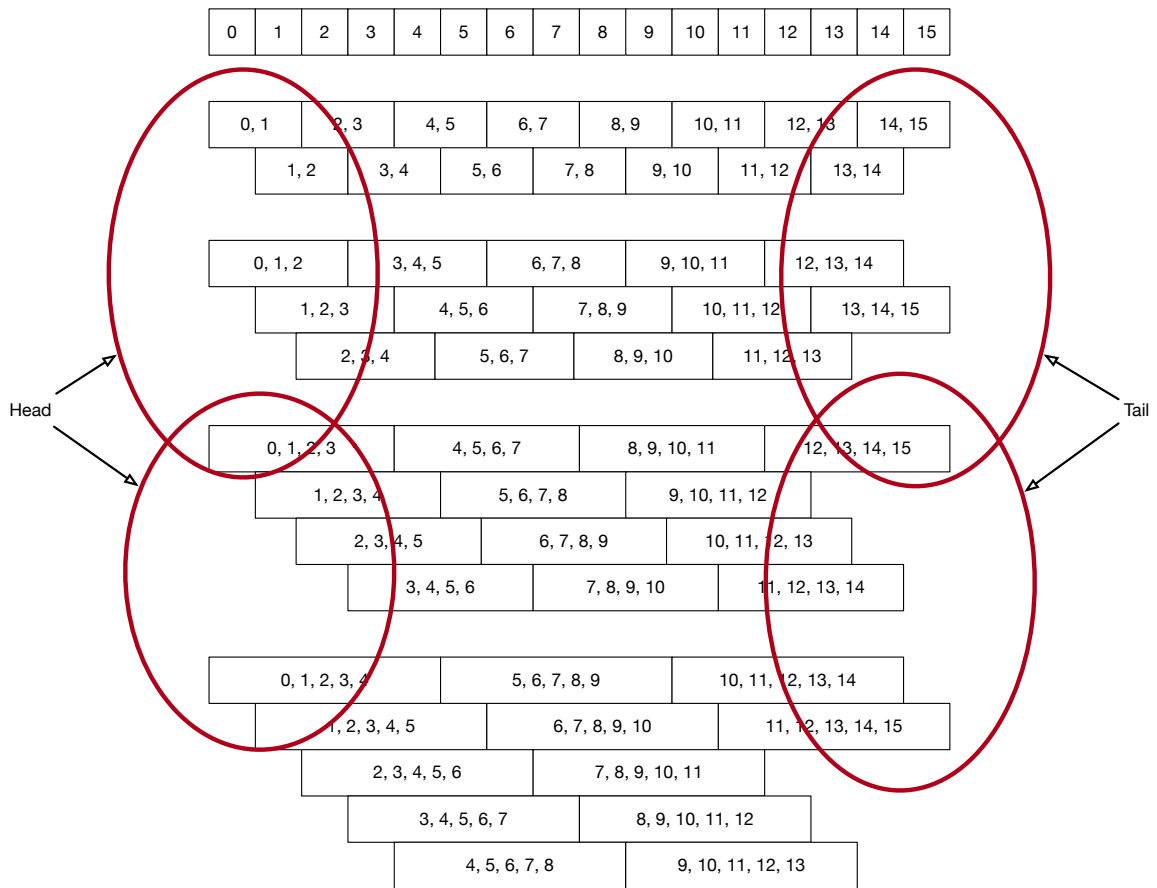


Figure 3.11: Head and tail region in the grid

Head At the beginning of each row, after making connection with the first parent node, the current node would traverse through the stack of previous rows to try to make connections with skipped slices.

Body This is the “normal” operation, where current node and parent node try to make connection, if anyone is saturated, iterator will move to the next item.

Tail At the end of each row, if parent nodes are depleted, and the current node is still not yet saturated, the current node would traverse through the stack of previous rows again to try to make connections with skipped slices at the end of each previous rows until the current node is fully saturated.

The head and tail regions are marked with red circle in Figure 3.11.

At the end, nodes from the last block containing slices that have not yet used by any down stream will connect with corresponding head node from the very first row of the grid.

3.5.5 Grid Operation

3.5.5.1 Message Format and Protocol. The general format of messages sent, shown in Figure 3.12, is a tag followed by message content. Tag is used to specify the type. With the current setup, there are two types. One is used to specify the source of the message, the other is used to signal downstream thread that the upstream thread has stopped. Also, as shown in Figure 3.12, case 1 is only used for sending data between async nodes, and case 3 is only used for head nodes to send result for additional threads to collect. Case 3 is used in both inter-nodes and external communications.

3.5.5.2 Asynchronous Operation. The grid purpose is to allow different nodes work on different part of the dataset concurrently without breaking any data dependency rules by creating a dependency network and sharing local data with only relevant nodes. Sharing node send its data as a vector of two elements. The first

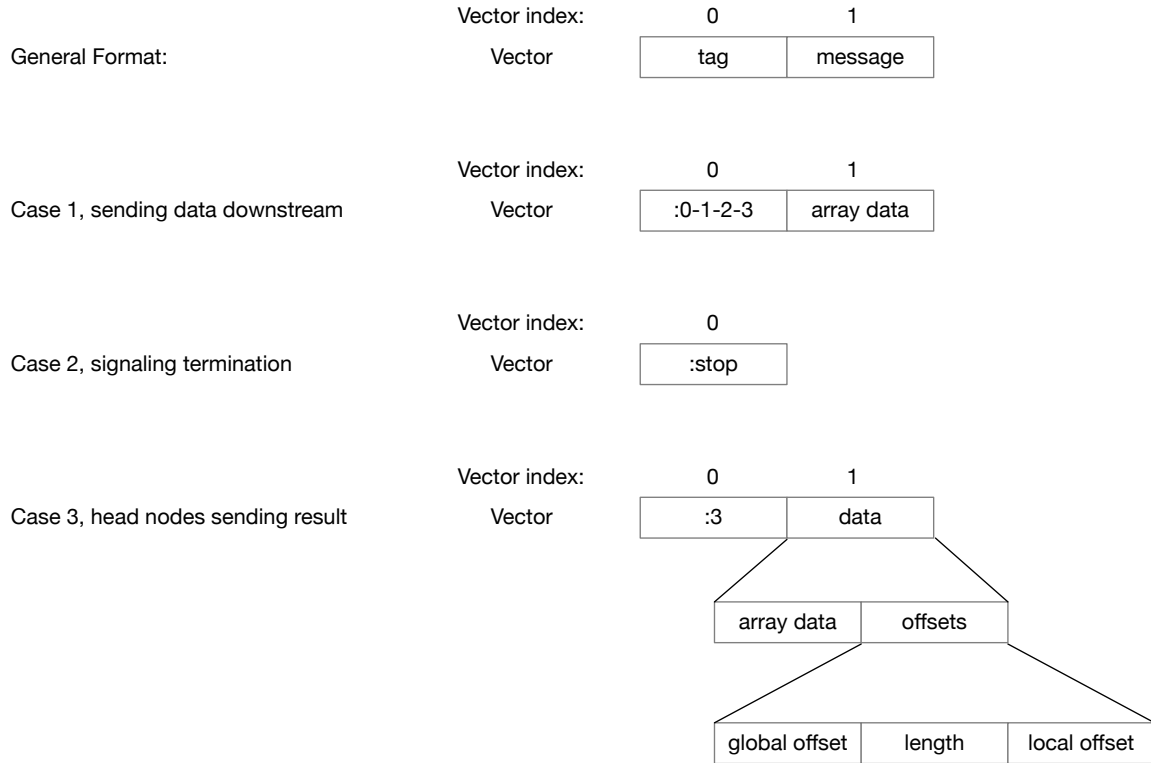


Figure 3.12: Format used by sent messages

element is the key of the sending node, and the second element is the reference of shared data. When downstream node, i.e. receiving node, receives the data, it will need to copy the section of data needed to local data. Since the data is an array, the copy usually involves the following parameters:

Length The number of elements should be copied from source array to destination array.

Source offset From which index source array should be copied.

Destination offset To which index of destination array source array should be copied.

Since we know the slice coverage of each node, we can calculate these parameters right after all the nodes are connected. The length is simply the number of overlapping slices times the number of pixels of each slice, the source offset is difference between the first index of overlapping slices and the first index of upstream

node slices, and the destination offset is the difference between the first index of overlapping slices and the first index of downstream node slices. These information are stored in a map on downstream nodes where key is the key of the upstream node and the value to the a vector of source offset, length and destination offset. With this setup, whenever we get an value from an upstream node, we just need to do a quick lookup with that upstream node's key, and the value we get can be used to call java's "arraycopy" function to copy received data to local array segment.

3.5.5.3 Grid Connection Verification. One of the simplest operation we can do with the grid is to verify the connectivity and consistency. For a grid with n slices, a zero array with length n is passed to the grid.

- (1) First layer of the grid will fetch the array data based on its index, and operate on it. Then pass the result to is downstream nodes defined by the grid connectivity.
- (2) Threads corresponding to each block will wait for data defined by their upstream connection. Once all data are collected, combine them, and operate on them. Results are send to downstreams.
- (3) Once First layer of the grid got the result back, they will combine their results as a new array and returns.

The operation used for testing could be as simple as "+1" to each element.

Figure 3.13 illustrate testing the a grid described in Figure 3.11 by adding 1 to a zero array, and expected result.

3.5.5.4 Blocking Asynchronous Reconstruction. The general process of asynchronous reconstruction using async grid works as follows:

- (1) All the histories are added to a index tree that is sorted based on their starting slice and history length as shown in Figure 3.5.

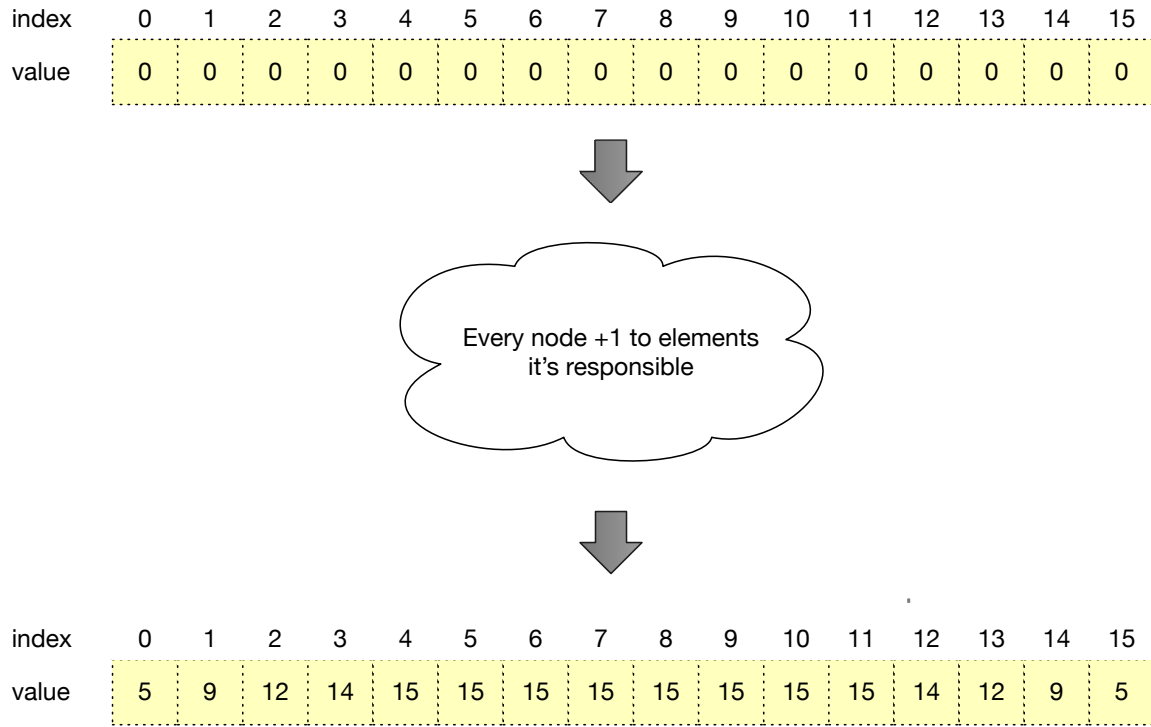


Figure 3.13: Testing a grid with 16 slices and depth = 5

- (2) Setup async grid based on the dimension of the dataset. For example, for a dataset with 16 slices, one possible setup is shown in Figure 3.14.
- (3) Launch “Head Node” threads, as seen in Figure 3.7 (a), for nodes in the first layer of the grid.
- (4) Launch “Body Node” threads, as seen in Figure 3.7 (b), for the rest of the node son the grid.
- (5) Launch a separate collector thread to collect final result from “Head Nodes” threads.

This version of algorithm will be referred as “Blocking” for the fact that majority of the running threads are blocked due to nodes dependency. There is no limits on the number of threads a process can have in the OS, but too many blocking threads will make Operation System’s scheduling task more difficult and it will add to the overall overhead time.

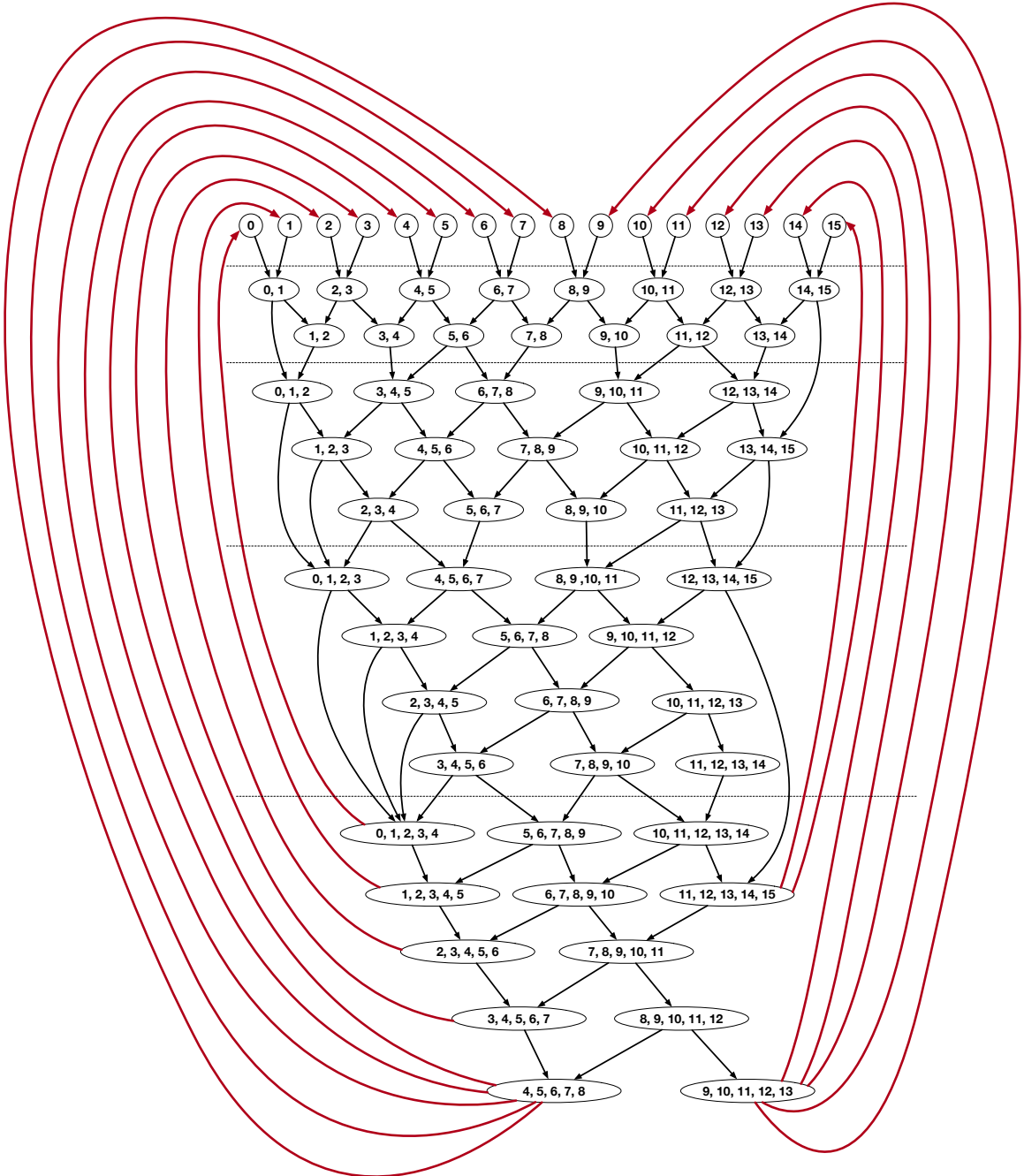


Figure 3.14: Grid with slices = 16, depth = 5

3.5.5.5 Threaded Asynchronous Reconstruction. One way, potentially could ease the scheduling mess for the OS is to combine some head and body threads into one thread. In Figure 3.14, threads, or blocks, that have dependencies vertically will never run at the same time. One strategy to combine threads, is based on their corresponding first slice id, for example, thread 0, (0, 1), (0, 1, 2) and so on, can be combined into one thread. This is shown in Figure 3.15, in which threads that have the same starting slice id have the same color, it's an indication that they are combined into one thread. When combined into one thread, there no need to communicate through channel. The data can be simply passed through memory.

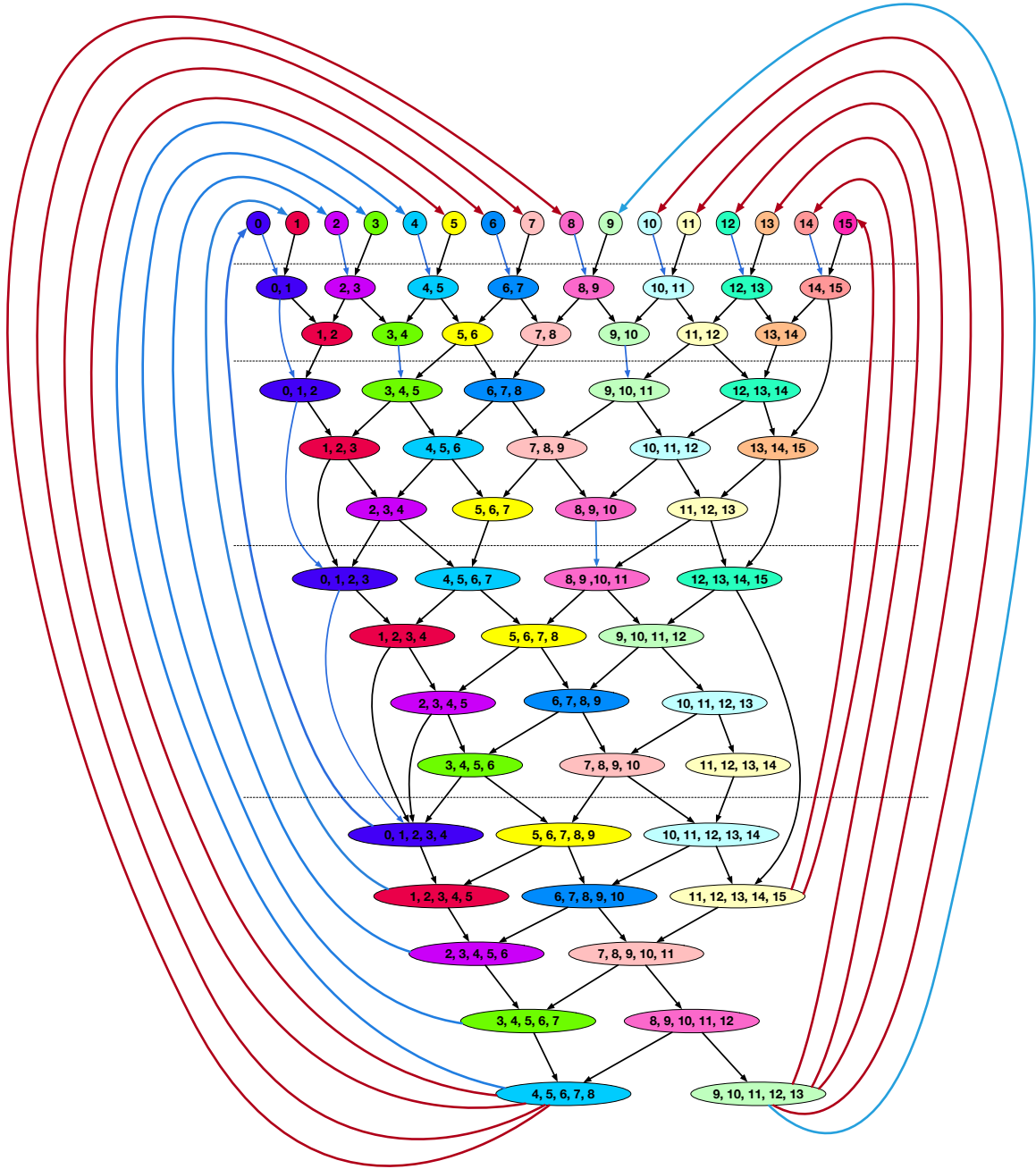


Figure 3.15: Threaded version of async reconstruction

CHAPTER FOUR

Experiment

4.1 Introduction

This chapter describes the experiments used to measure the time performance of the proposed algorithm. Experiments are run on two different hardwares over three different datasets to demonstrate the performance and how the performance scales over different data size and hardware configurations.

4.2 Hardware

The two machines used for these experiments are listed in Table 4.1. The major difference between **pb005** and **lm001** is the number of CPUs. **lm001** has almost twice number of CPUs than **pb005**. Although **lm001** has significant more memory than **pb005**, runtime memory is kept under 240GB. So the additional memory did not have any impact on the performance.

4.3 Dataset

Two different phantoms are used in experiments. They are

- CTP404 Sensitometry Phantom
- George Phantom

Table 4.1: Configurations of computers used for performance testing.

Machine Name	pb005	lm001
CPU Model	Xeon E5-2690 v2	Xeon Gold 6140
Base Frequency	3.0 GHz	2.30 GHz
Max Turbo Frequency	3.6 GHz	3.7 GHz
Physical Cores	20	36
L3 Cache	25 MB	24.75 MB
Memory Size	252 GB	755 GB
Memory bus speed	1866 MT/s (DD3)	2666 MT/s (DDR4)

Table 4.2: Dataset differences

Dataset	Thickness	Number of Slices	Number of Histories
CTP404	2.5 mm	16	66,123,545
George 2.5mm	2.5 mm	34	122,381,720
George 1mm	1 mm	64	123,832,929

There are 3 different datasets used for experiments, they are listed as follows

CTP404 A scan of CTP404 phantom, histories are computed with 2.5mm slice thickness.

George 2.5mm A scan of George phantom, histories are computed with 2.5mm slice thickness.

George 1mm This is the same scan of George 2.5mm dataset, histories are computed with 1mm slice thickness.

The scan of CTP404 phantom is much smaller in both dimension and the number of histories. By comparing CTP404 phantom dataset with george phantom 2.5mm thickness history set, as shown in Table 4.2, we can say that the george phantom 2.5mm dataset is about twice as big as CTP404 2.5mm dataset. On the other hand, we can see that george phantom 1mm dataset is about the same size as george phantom 2.5mm dataset in term of number of histories, while has almost twice the amount of slices. These datasets should provide a good indication of how the algorithm scales as dataset increases.

4.4 Experiments

On each computing node, the following procedure is used to run the performance test

- (1) Start a Clojure runtime
- (2) Load a dataset
- (3) Setup an async grid using configuration specific to each dataset.
- (4) Run sequential version using the grid for 5 times.

Table 4.3: CTP404 dataset history distribution over slice depth

Depth	History count	%	Accumulated count	acc %
1	22,653,674	34.26%	22,653,674	34.26%
2	31,349,235	47.41%	54,002,909	81.67%
3	9,841,381	14.88%	63,844,290	96.55%
4	1,970,878	2.98%	65,815,168	99.53%
5	268,949	0.41%	66,084,117	99.94%
6	29,719	0.04%	66,113,836	99.99%
7	4,874	0.01%	66,118,710	99.99%
8	2,044	0.00%	66,120,754	100.00%
9	1,222	0.00%	66,121,976	100.00%
10	758	0.00%	66,122,734	100.00%
11	443	0.00%	66,123,177	100.00%
12	231	0.00%	66,123,408	100.00%
13	101	0.00%	66,123,509	100.00%
14	32	0.00%	66,123,541	100.00%
15	4	0.00%	66,123,545	100.00%
16	0	0.00%	66,123,545	100.00%

- (5) Run blocked version, (see Section 3.5.5.4), using the grid for 5 times.
- (6) Run threaded version, (see Section 3.5.5.5) using the grid for 5 times.
- (7) Exit the Clojure runtime.

The exact procedure is used for every dataset on every computing node.

4.4.1 CTP404 Sensitometry Phantom

4.4.1.1 Setup. The proton history counts of dataset CTP404 across different starting slice and slice depth are listed in appendix Tables B.1, Table B.2 and Table B.3. A summary of those tables are shown in Table 4.3.

Due to the thickness of the slice is 2.5mm, slice depth of 4 means there are up to 10mm deviation from the original intended direction. Although some deviation is necessary to obtain some cross slice information, too much of is an indication of noisy data. So a cutoff point is set at slice depth 4. Also notice that at slice depth 5, the amount of data to be included is less than 1%. With this cutoff point, there are still 99.53% histories included. The cut-off point can be seen in table 4.3.

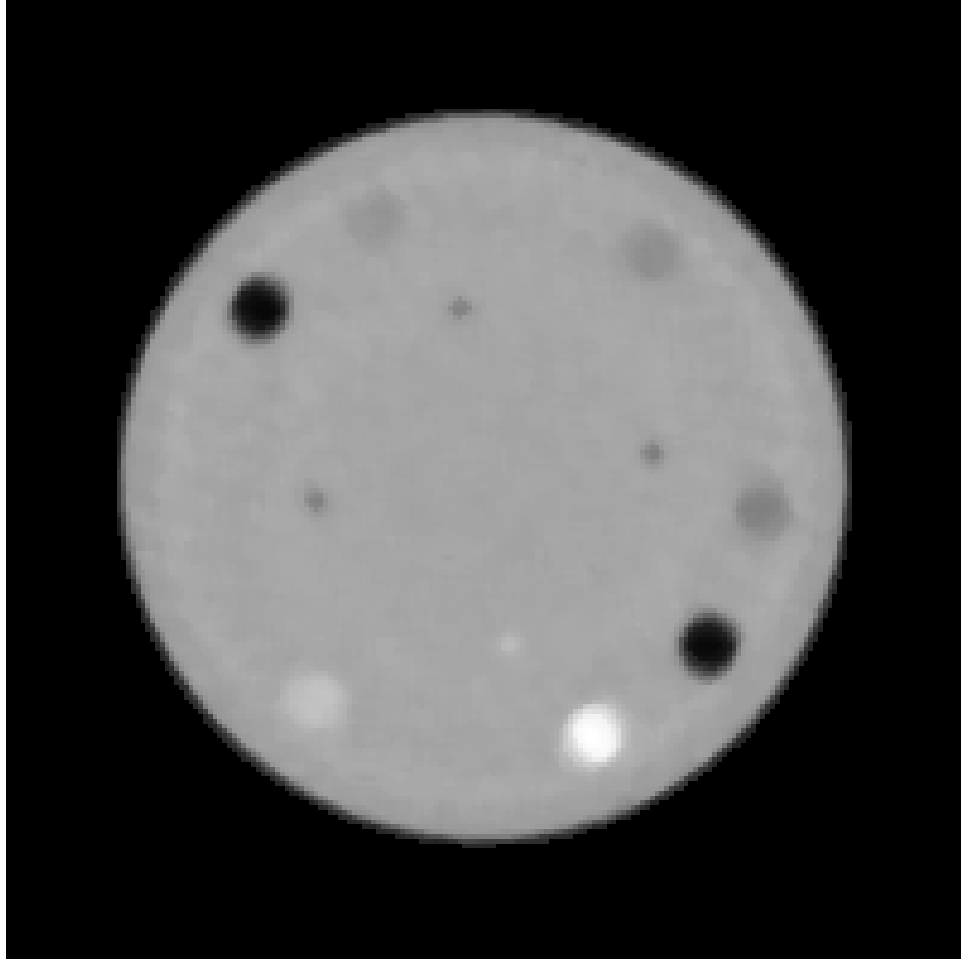


Figure 4.1: Reconstruction of CTP404 phantom 2mm thickness, iteration 6

4.4.1.2 Reconstructed Image. The reconstruction image after 6 iterations is shown in Figure 4.1. All the features of the phantom can be identified. The detailed quality is not of concern here, since appropriate parameters can be adjusted to obtain images with much better quality.

4.4.2 George Phantom 2.5mm thickness

4.4.2.1 Setup. The proton history counts of dataset CTP404 across different starting slice and slice depth are listed in appendix Tables C.1, Tables C.2, Tables C.3, Tables C.4 and Tables C.5. A summary of those tables are shown in table Tables 4.4.

Table 4.4: George 2.5mm dataset history distribution over slice depth

Depth	History count	%	Accumulated count	acc %
1	39,809,087	32.53%	39,809,087	32.53%
2	58,399,290	47.72%	98,208,377	80.25%
3	19,608,018	16.02%	117,816,395	96.27%
4	4,013,844	3.28%	121,830,239	99.55%
5	512,695	0.42%	122,342,934	99.97%
6	37,322	0.03%	122,380,256	100.00%
7	1,446	0.00%	122,381,702	100.00%
8	18	0.00%	122,381,720	100.00%
9	0	0.00%	122,381,720	100.00%

The distribution is very similar to dataset CTP404. A cutoff point is also set at slice depth 4. The number of histories include for computation in terms of percentage is very similar to that of CTP404 dataset, also turns out to be 99.55%. This is probably due to the fact that both datasets are computed with 2.5mm thickness.

4.4.2.2 Reconstructioned Image. Reconstructed image of 12th slice after 6 iterations is shown in Figure 4.2. All the important features are clearly identifiable in the image.

4.4.3 George Phantom 1mm thickness

4.4.3.1 Setup. The proton history counts of dataset CTP404 across different starting slice and slice depth are listed in appendix Tables D.1, Tables D.2, Tables D.3, Tables D.4, Tables D.5, Tables D.6, Tables D.7, Tables D.8 and Tables D.9. A summary of those tables are shown in Tables 4.5.

Due to the thickness of each slice is 1mm, the slice depth of proton histories tends to be much deeper in terms of number of slices. Although it's tempting to set the slice depth to be 10 so that slice depth in terms of mm is the same as George 2.5mm dataset, Table 4.5 shows that histories with depth of 9 is less than 1% (still less than 1% even after adding depth of 10). Also, at slice depth of 8, there are more



Figure 4.2: Reconstruction of George phantom with 2mm thickness, iteration 6

Table 4.5: George 1mm dataset history distribution over slice depth

Depth	History count	%	Accumulated count	acc %
1	13,566,575	10.96%	13,566,575	10.96%
2	36,215,913	29.25%	49,782,488	40.20%
3	28,174,841	22.75%	77,957,329	62.95%
4	19,405,078	15.67%	97,362,407	78.62%
5	12,258,448	9.90%	109,620,855	88.52%
6	7,108,926	5.74%	116,729,781	94.26%
7	3,813,738	3.08%	120,543,519	97.34%
8	1,883,364	1.52%	122,426,883	98.86%
9	851,217	0.69%	123,278,100	99.55%
10	354,999	0.29%	123,633,099	99.84%
11	134,812	0.11%	123,767,911	99.95%
12	45,933	0.04%	123,813,844	99.98%
13	14,068	0.01%	123,827,912	100.00%
14	3,853	0.00%	123,831,765	100.00%
15	918	0.00%	123,832,683	100.00%
16	195	0.00%	123,832,878	100.00%
17	48	0.00%	123,832,926	100.00%
18	3	0.00%	123,832,929	100.00%
19	0	0.00%	123,832,929	100.00%

histories included than george 2.5mm dataset at depth of 4. So depth of 8 for this dataset should be sufficient for this dataset.

4.4.3.2 Reconstructed Image. A reconstructed middle slice, slice #38, after 6 iterations is shown in Figure 4.3.

4.4.3.3 Timing. Performance data are shown in Table 5.8 and Table 5.9. On pb005, the speedup over sequential algorithm is up to 16, while on lm001 speedup is about 10-11, which is also pretty impressive. Part of the reason that pb005, a machine with less core and memory bandwidth can achieve much more speed up is probably due the face that the concurrent version of reconstruction algorithm is close to its maximum potential on that machine. This also indicates that on lm001, it can take on a much larger dataset without having too much performance impact.



Figure 4.3: Reconstruction of George phantom with 1mm thickness, iteration 6

CHAPTER FIVE

Discussion

5.1 Performance

5.1.1 Theoretical Speedup

Base on the cut-off depths described in Section 4.4.1.1, Section 4.4.2.1 and Section 4.4.3.1, together with history count data from Appendix B, Appendix C and Appendix D, speedup info are extracted shown in Table 5.1, Table 5.2 and Table 5.3.

5.1.2 Timing Data

The timing data from experiments on the three datasets are shown in Table 5.4, Table 5.5, Table 5.6, Table 5.7, Table 5.8 and Table 5.9

Note that the difference between “blocked” and “threaded” version is not big, but for the most of the result, they do beat out the “blocked” version. The “threaded” algoirthm’s timing data do have a a very stable timing result, while “blocked” version’s timing data varies quite a bit. This improvement is mostly likely due to less threads to be scheduled by the OS. Overall the difference between the two versions are not huge.

The dataset size from CTP404 to Geroge phantom 2.5mm is almost doubled, with the same grid depth, but the time increase is only about $(71.62 - 55.43)/55.43 = 0.29$, which is 29%. So this algorithm does seem to scale well with problem size. Also when the dataset is broken down into more slices, as seen from George 2.5mm to George 1mm, the timing does gets better. Although not by much, only $(71.62 - 60.13)/60.13 = 0.19$, 19%. The small increase is due to the increase in the depth of the grid, as well as different data distribution over the grid.

Table 5.1: Theoretical Speed up for CTP404 dataset

layer index	subtotal	%	blocks	speedups
(1 0)	22653674	34.42%	15	5.16
(2 0)	16100239	24.46%	8	1.96
(2 1)	15248996	23.17%	7	1.62
(3 0)	3942811	5.99%	5	0.30
(3 1)	3012654	4.58%	5	0.23
(3 2)	2885916	4.38%	4	0.18
(4 0)	588537	0.89%	4	0.04
(4 1)	511301	0.78%	3	0.02
(4 2)	463596	0.70%	3	0.02
(4 3)	407444	0.62%	3	0.02
speedup				9.5

Table 5.2: Theoretical Speed up for George 2.5mm dataset

layer index	subtotal	%	blocks	speedups
(1 0)	39809087	32.68%	25	8.17
(2 0)	29516558	24.23%	13	3.15
(2 1)	28882732	23.71%	12	2.84
(3 0)	6624510	5.44%	8	0.43
(3 1)	7002776	5.75%	8	0.46
(3 2)	5980732	4.91%	8	0.39
(4 0)	1141320	0.94%	6	0.06
(4 1)	1083321	0.89%	6	0.05
(4 2)	820886	0.67%	6	0.04
(4 3)	968317	0.79%	5	0.04
speedup				15.6

Table 5.3: Theoretical Speed up for George 1mm dataset

layer index	subtotal	%	blocks	speedups
(1 0)	13566575	11.08%	62	6.87
(2 0)	17983653	14.69%	32	4.70
(2 1)	18232260	14.89%	31	4.62
(3 0)	9449985	7.72%	21	1.62
(3 1)	9180979	7.50%	21	1.57
(3 2)	9543877	7.80%	20	1.56
(4 0)	4847964	3.96%	16	0.63
(4 1)	4867948	3.98%	15	0.60
(4 2)	4699673	3.84%	15	0.58
(4 3)	4989493	4.08%	15	0.61
(5 0)	2381373	1.95%	12	0.23
(5 1)	2552439	2.08%	12	0.25
(5 2)	2409686	1.97%	12	0.24
(5 3)	2489053	2.03%	12	0.24
(5 4)	2425897	1.98%	12	0.24
(6 0)	1114473	0.91%	10	0.09
(6 1)	1359706	1.11%	10	0.11
(6 2)	1280376	1.05%	10	0.10
(6 3)	1223912	1.00%	10	0.10
(6 4)	1068627	0.87%	10	0.09
(6 5)	1061832	0.87%	9	0.08
(7 0)	525082	0.43%	9	0.04
(7 1)	565009	0.46%	9	0.04
(7 2)	528424	0.43%	8	0.03
(7 3)	530334	0.43%	8	0.03
(7 4)	548112	0.45%	8	0.04
(7 5)	564999	0.46%	8	0.04
(7 6)	551778	0.45%	8	0.04
(8 0)	178049	0.15%	8	0.01
(8 1)	254212	0.21%	7	0.01
(8 2)	264364	0.22%	7	0.02
(8 3)	271022	0.22%	7	0.02
(8 4)	259018	0.21%	7	0.01
(8 5)	241960	0.20%	7	0.01
(8 6)	215590	0.18%	7	0.01
(8 7)	199149	0.16%	7	0.01
speedup				25.5

Table 5.4: Timing for tests performed for dataset CTP404 on pb005

algorithm	run 1	run 2	run 3	run 4	run 5	Average	Speedup
sequential	374.01s	392.72s	392.18s	385.93s	389.88s	386.94s	1.0
blocked	58.46s	57.56s	54.40s	53.48s	53.86s	55.55s	7.0
threaded	53.82s	54.12s	53.88s	54.19s	53.97s	54.00s	7.1

Table 5.5: Timing for tests performed for dataset CTP404 on lm001

algorithm	run 1	run 2	run 3	run 4	run 5	Average	Speedup
sequential	410.32s	341.22s	332.89s	322.01s	322.56s	345.80s	1.0
blocked	56.41s	55.83s	56.35s	54.56s	53.04s	55.24s	6.3
threaded	55.27s	53.77s	57.35s	54.24s	56.53s	55.43s	6.2

Table 5.6: Timing for tests performed for dataset george 2.5mm on pb005

algorithm	run 1	run 2	run 3	run 4	run 5	Average	Speedup
sequential	861.19s	853.65s	860.00s	861.40s	858.13s	858.87s	1.0
blocked	78.79s	77.59s	83.43s	71.62s	71.73s	76.63s	11.2
threaded	76.33s	70.74s	70.85s	69.51s	70.31s	71.55s	12.0

Table 5.7: Timing for tests performed for dataset george 2.5mm on lm001

algorithm	run 1	run 2	run 3	run 4	run 5	Average	Speedup
sequential	726.65s	638.74s	652.73s	643.70s	643.80s	661.12s	1.0
blocked	75.12s	74.23s	73.56s	71.63s	69.85s	72.88s	9.0
threaded	69.70s	71.96s	70.85s	71.89s	73.71s	71.62s	9.2

Table 5.8: Timing for tests performed for dataset george 1mm on pb005

algorithm	run 1	run 2	run 3	run 4	run 5	Average	Speedup
sequential	1092.69s	1089.82s	1103.35s	1090.50s	1120.40s	1099.35s	1.0
blocked	71.25s	70.62s	65.95s	69.69s	66.68s	68.84s	16.0
threaded	69.81s	67.24s	66.70s	66.24s	67.25s	67.45s	16.3

Table 5.9: Timing for tests performed for dataset george 1mm on lm001

algorithm	run 1	run 2	run 3	run 4	run 5	Average	Speedup
sequential	711.15s	654.76s	655.19s	650.07s	682.13s	670.66s	1.0
blocked	62.68s	60.14s	58.05s	75.72s	60.97s	63.51s	10.6
threaded	62.08s	60.60s	59.42s	59.92s	58.64s	60.13s	11.2

All reconstruction ran in these tests are with 6 iterations. 6 iterations used because in [8], it is shown that with right parameters, it's possible to get a good quality image with 6 iterations. Even though, in this setup, ART was applied instead of DROP, which was used in [8], this asynchronous method mathematically does not conflict with existing reconstruction algorithms. This asynchronous algorithm can be integrated with any existing reconstruction.

Sorting histories into index tree is not part of the timing test. The reason for this is that sorting existing data into index tree involves a lot unnecessary overheads. The sorting can be incorporated with proton history preprocessing which requires further testing.

5.2 Scalability

5.2.1 Scale Over Multi-cores

The asynchronous nature of the algorithm makes it flexible over number of cores. With 63 slices of data, 20 cores is able to do almost as well as 36 cores compute node. This is due to the fact that the top section of the grid only gets run simultaneously only during the first iteration when they are starting almost at the same time. By the end of the first iteration, due to dependencies, threads progress differently. Threads resides on the outside area of the grids tends to progress faster. So the throughput is not determined by the number of slices, which is the nubmer of blocks on the top of the grid, but the number of blocks of the last layers of the grid. This suggest that if we can find out exactly how many data we can trim off without impact the quality of the images reconstructed, we can probably improved the performance by having a shallower grid.

5.2.2 Scale Over Network

The method we have described and demonstrated here is very flexible and scalable. Let's assume we have a total 32 slices to reconstruct and each machine can

only work on 16 slices at a time. Full partitions of 32 slices are shown in Figure 5.1. White blocks have all the slices on machine A, gray blocks have all the blocks on machine B while the yellow blocks have slices on both machines. In this situation, we can assign all the white blocks to machine A, gray blocks to machine B. Both white and yellow blocks are normal grid blocks but on different machines. Yellow blocks can be treated as an abstract block. They will be inserted to both grids on machine A and machine B to ensure normal blocks function as normal, i.e. abstract blocks will be connected as upstream or downstream, and normal blocks that are connected to them are expecting data sent from them or will send them their data. This configuration can be shown in Figure 5.2. The difference between a normal block and an abstract block is that a normal block will work on the reconstruction algorithm for the slices it's assigned to, while abstract blocks may or may not. For every pair of abstract blocks, only one of them needs to work on the actual reconstruction, the other will be mostly responsible for data transmission. Take block 13-14-15-16 shown in Figure 5.2 for example:

- (1) Block 13-14-15-16 on machine B will send the missing data from slice 16 to machine A
- (2) Block 13-14-15-16 on machine A work on the reconstruction
- (3) Block 13-14-15-16 on machine A send the result to machine B

5.3 Improvements

The implementation is done in JVM environment for the ease of exploring some of the concurrent features of the data and the algorithm. However, if implemented in a language with better runtime performance, the timing result should be much better.

This algorithm does not conflict with existing reconstruction algorithms such as BIP or SAP. Rather, it can be combined with BIP or SAP to have even better performance.

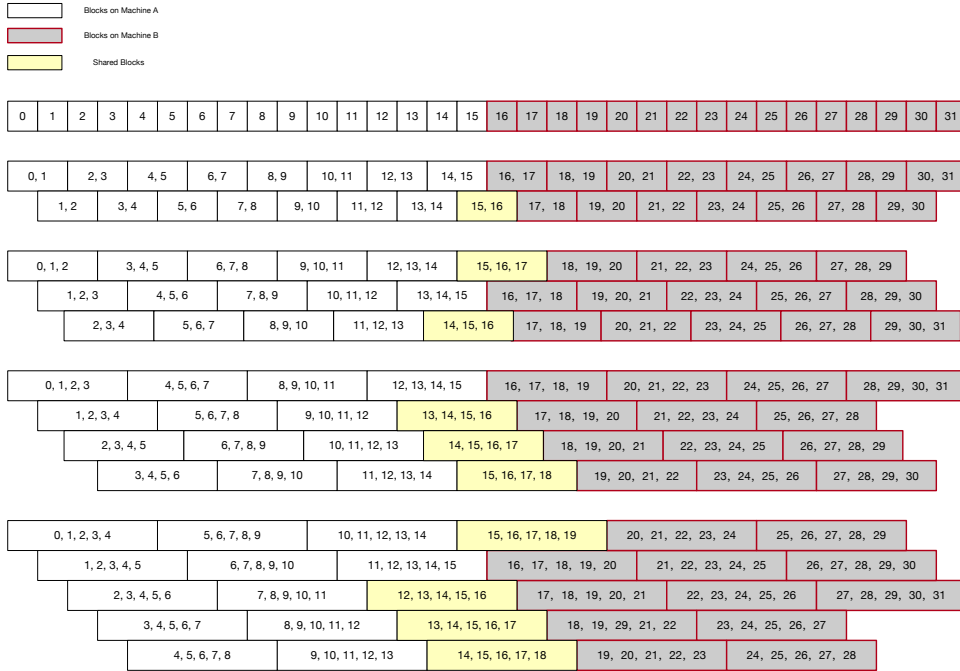


Figure 5.1: Block partition across multiple machines

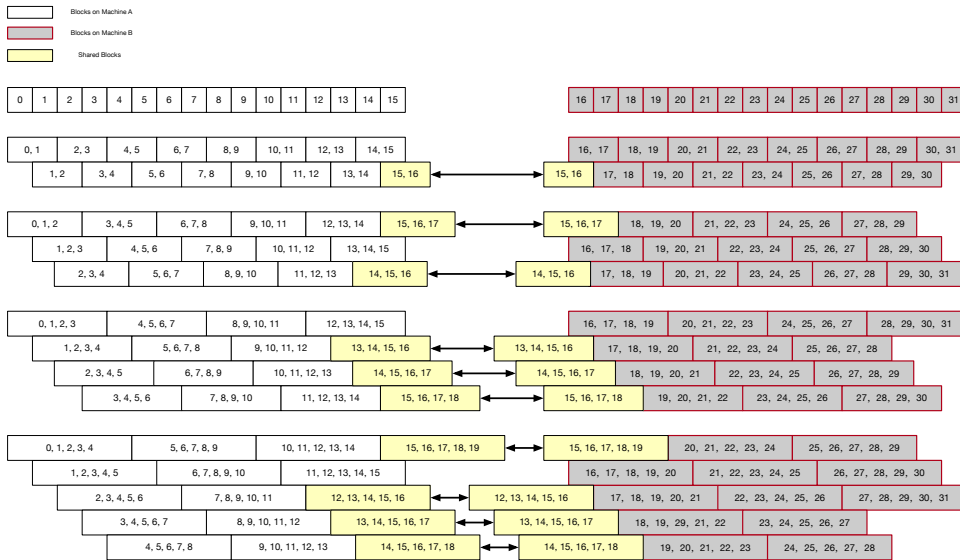


Figure 5.2: Thread abstraction layer

During the test, computing nodes's hyperthreading feature is turned on. Turning them off, might help to have better performance since running two threads on a single core could impact cache performance.

CHAPTER SIX

Conclusion

In this work, we have introduced a new algorithm that can be used for medical image reconstruction using projection methods, such as ART. This algorithm is capable of generating significant speedup for multi-sliced 3D image reconstruction over its sequential counterpart. Our testing data suggest that it can move the reconstruction time frame into clinical usage requirements. The speedup mostly comes from exploring the nature of data concurrency in this type of image reconstruction.

The algorithm is asynchronous in nature due to the architecture is based on data concurrency. The asynchronosness allows many parts of the reconstruction to run at the same time. Modern-day hardware is moving toward a many-core future. On the machine we have tested this algorithm on, there are a total of 36 physical CPU cores, while we were only able to use 15 of them at most due to the size of the dataset. If the dataset were to double the number of slices and preserve the average number of history per voxel, this algorithm is able to scale with the dataset without any time increase. Also, due to the advance in hardware, 36 CPU per is a relatively small size. As of this dissertation is written, 96 CPU computing machine with two sockets, i.e. 96 cores are composed of two 48 core CPUs, is available. Amazon's AWS is providing such computing resources. AMD currently has a 64 core desktop CPU on the market and has a 96 core CPU coming out the next year. ARM also released its road map for the next couple of years that includes 192 core computing nodes. All this information means that we can scale our reconstruction from the current 15 slices to 100 slices with very little time loss on JVM. For more than 100 slices, we have suggested an asynchronous communication scheme that can scale across multiple JVM

or compute nodes with minimum time loss while still maintaining the the structure and consistency of the algorithm.

The structure of the algorithm also allows for different projection methods to be used. In our experiments, we have used only the original ART projection method. For even more speedup, a string averaging variant can be used. It would give more threads per block, thus decrease the time of each iteration.

Overall, this new algorithm introduced a new approach to 3D medical image reconstruction. So far, we have demonstrated that CPU implementation of this suggested method can come very close to the fastest GPU-based reconstruction method with more flexibility and scalability.

6.1 Future Direction

A larger dataset across multiple compute nodes still needs to be studied. The effects of asynchronous communication on the timing are unclear at this point. Also, due to the asynchronous and structural constraints, one thread per slice is only needed at the very beginning of the reconstruction. From then on, only half of the threads are needed. So it is interesting to see that as the number of slices becomes more significant than the number of CPUs cores, how the overall computation time would be affected. This could show even more potential and advantages of the asynchronous design.

APPENDICES

APPENDIX A

Testing Scripts

A.1 CTP404 Phantom

Listing A.1: Testing script for CTP404 Dataset

```
(defn generate-lambdas
  [^long layers f]
  {:pre [(and (int? layers) (pos? layers))]}
  (into (sorted-map)
        (mapv (fn [^long i]
                (let [i (unchecked-inc i)] [i (f i)])
              (range layers))))

(def dataset-name "exp_CTP404")
(def base-dir (format "datasets/%s" dataset-name) #_"datasets/
George-02272019-Run23-interpolated/output-2.5mm/B_1280000_L_1.000000")
(def dataset-config {:min-len 75
                    :batch-size 100000
                    :style :new
                    :count? false
                    :global? false
                    :phantom "CTP404"
                    :thickness "2.5mm"})

(if (.isDirectory (clojure.java.io/file base-dir))
    (do (timbre/info (format "Loading %s dataset ... " dataset-name))
        (def data_CTP404 (with-open [dataset ^PCTDataset (pct.data.io/newPCTDataset
base-dir "MLP_paths_r=1.bin" "WEPL.bin")]
                        {:x0 (.x0 dataset)
                         :rows (.rows dataset)
                         :cols (.cols dataset)
                         :slice-count (.slices dataset)
                         :slice-offset (pct.data.io/slice-offset dataset)
                         :index (pct.data.io/load-dataset dataset
dataset-config)})))
        (timbre/info (format "Path [%s] does not exist or is not a folder." base-dir)))

(def x0 (:x0 data_CTP404))
(def ^{:tag 'long} _rows (long (:rows data_CTP404)))
(def ^{:tag 'long} _cols (long (:cols data_CTP404)))
(def ^{:tag 'long} offset (* _rows _cols))
(def ^{:tag 'long} slices 16)
(def samples [7 8 9])
(def history-index (:index data_CTP404))
(def ^{:tag 'long} depth 4)
(def regions test/George-regions)
(def recon-opts {:iterations 6
                 :lambda (generate-lambdas 10 (fn [i] (/ 0.00025 (double i))))
                 :tvs? true
                 :tvs-alpha 0.75
                 :tvs-N 5
                 :dump true
                 :grid-depth depth
                 :dataset dataset-config})

(def ^{:tag 'int} test-runs 5)
(def recon-type :blocked)

(spec/def ::recon-types #{:blocked :threaded :sequential})
(spec/def ::sample-range-check (spec/coll-of (spec/and pos? #(< % slices))))

(defn run
  ([] (run test-runs recon-type))
  ([^long n] (run n recon-type))
  ([^long test-runs recon-type]
   {:pre [(int? test-runs)]})
```

```

        (spec/valid? ::recon-types recon-type)
        (spec/valid? ::sample-range-check samples)]]
(let [s (format "Running test for %s x%d" recon-type test-runs)]
  (println s)
  (timbre/info s))
(def grid nil)
(def result nil)
(loop [i (int 0)]
  (when (< i test-runs)
    (cond
      (= recon-type :blocked)
      (do (def result-folder (format "results/%s/blocked" dataset-name))
          (def grid (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                  :connect? true :slice-offset offset))
          (def result (recon/async-art-blocked grid history-index x0 recon-opts
          )))
        (= recon-type :threaded)
        (do (def result-folder (format "results/%s/threaded" dataset-name))
            (def grid (-> (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                          ) :connect? true :slice-offset offset)
                      (.trim-connections)))
            (def result (recon/async-art-threaded grid history-index x0
            recon-opts)))
        (= recon-type :sequential)
        (do (def result-folder (format "results/%s/sequential" dataset-name))
            (def grid (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                  :connect? true :slice-offset offset))
            (def result (recon/seq-art grid history-index x0 recon-opts)))
        :else
        (do (timbre/info (format "Unknown recon-type: [%s]" recon-type))
            (def result nil)))
      (when result
        (let [stats (->> result
                      (mapv (fn [[k v]]
                            (when (int? k)
                              [k (test/series-stats (first v) [_rows _cols
                              slices] regions samples)]))
                      (remove nil?)
                      (into (sorted-map))
                      (#(assoc % 0 (test/series-stats x0 [_rows _cols slices]
                      regions samples))))]
              (pct.data.io/save-result result _rows _cols slices recon-opts
              {:type :all
               :folder result-folder
               :result {:samples samples :stats stats}}))]
          (recur (unchecked-inc-int i))))))
;; (run 5 :blocked)
;; (run 5 :threaded)
;; (run 5 :sequential)

```

A.2 George 2.5mm

Listing A.2: Testing script for George 2.5mm Dataset

```
(defn generate-lambdas
  [^long layers f]
  {:pre [(and (int? layers) (pos? layers))]}
  (into (sorted-map)
        (mapv (fn [^long i]
                (let [i (unchecked-inc i)] [i (f i)]))
              (range layers))))

(def dataset-name "george-2.5mm")
(def base-dir (format "datasets/%s" dataset-name) #_"datasets/
George-02272019-Run23-interpolated/output-2.5mm/B_1280000_L_1.000000")
(def dataset-config {:min-len 75
                    :batch-size 100000
                    :style :new
                    :count? false
                    :global? false
                    :phantom "george"
                    :thickness "2.5mm"})

(if (.isDirectory (clojure.java.io/file base-dir))
    (do (timbre/info (format "Loading %s dataset ... " dataset-name))
        (def data_george_2p5mm (with-open [dataset ^PCTDataset (pct.data.io/
newPCTDataset base-dir "MLP_paths_r=0.bin")]
                                {:x0 (.x0 dataset)
                                 :rows (.rows dataset)
                                 :cols (.cols dataset)
                                 :slice-count (.slices dataset)
                                 :slice-offset (pct.data.io/slice-offset dataset)
                                 :index (pct.data.io/load-dataset dataset
dataset-config)})))

        (timbre/info (format "Path [%s] does not exist or is not a folder." base-dir)))

    (def x0 (:x0 data_george_2p5mm))
    (def ^{:tag 'long} _rows (long (:rows data_george_2p5mm)))
    (def ^{:tag 'long} _cols (long (:cols data_george_2p5mm)))
    (def ^{:tag 'long} offset (* _rows _cols))
    (def ^{:tag 'long} slices 25)
    (def samples (range 12 17))
    (def history-index (:index data_george_2p5mm))
    (def ^{:tag 'long} depth 4)
    (def regions test/George-regions)
    (def recon-opts {:iterations 6
                    :lambda (generate-lambdas 10 (fn [i] (/ 0.00025 (double i))))
                    :tvs? true
                    :tvs-alpha 0.75
                    :tvs-N 5
                    :dump true
                    :grid-depth depth
                    :dataset dataset-config})

    (def ^{:tag 'int} test-runs 5)
    (def recon-type :blocked)

    (spec/def ::recon-types #{:blocked :threaded :sequential})
    (spec/def ::sample-range-check (spec/coll-of (spec/and pos? #(< % slices)))))

(defn run
  ([] (run test-runs recon-type))
  ([^long n] (run n recon-type))
  ([^long test-runs recon-type]
   {:pre [(int? test-runs)
          (spec/valid? ::recon-types recon-type)
          (spec/valid? ::sample-range-check samples)]}
    (let [s (format "Running test for %s x%d" recon-type test-runs)]
      (println s)
      (timbre/info s))
    (def grid nil)
    (def result nil)
    (loop [i (int 0)]
      (when (< i test-runs)
        (cond
         (= recon-type :blocked)
         (do (def result-folder (format "results/%s/blocked" dataset-name))
```

```

      (def grid (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                :connect? true :slice-offset offset))
      (def result (recon/async-art-blocked grid history-index x0 recon-opts
      ))))
(= recon-type :threaded)
(do (def result-folder (format "results/%s/threaded" dataset-name))
    (def grid (-> (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                  )) :connect? true :slice-offset offset)
      (.trim-connections)))
    (def result (recon/async-art-threaded grid history-index x0
      recon-opts)))

(= recon-type :sequential)
(do (def result-folder (format "results/%s/sequential" dataset-name))
    (def grid (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
              :connect? true :slice-offset offset))
      (def result (recon/seq-art grid history-index x0 recon-opts)))

:else
(do (timbre/info (format "Unknown recon-type: [%s]" recon-type))
    (def result nil)))
(when result
  (let [stats (->> result
                 (mapv (fn [[k v]]
                        (when (int? k)
                          [k (test/series-stats (first v) [_rows _cols
                                                           slices] regions samples)]))
                       (remove nil?)
                       (into (sorted-map))
                       (#(assoc % 0 (test/series-stats x0 [_rows _cols slices]
                                                         regions samples))))]
        (pct.data.io/save-result result _rows _cols slices recon-opts
          {:type :all
           :folder result-folder
           :result {:samples samples :stats stats}}))
      (recur (unchecked-inc-int i))))))

;; (run 5 :blocked)
;; (run 5 :threaded)
;; (run 5 :sequential)

```


A.3 George 1mm

Listing A.3: Testing script for George 1mm Dataset

```
(defn generate-lambdas
  [^long layers f]
  {:pre [(and (int? layers) (pos? layers))]}
  (into (sorted-map)
        (mapv (fn [^long i]
                (let [i (unchecked-inc i)] [i (f i)]))
              (range layers))))

(def dataset-name "george-1mm")
(def base-dir (format "datasets/%s" dataset-name) #_"datasets/
George-02272019-Run23-interpolated/output-2.5mm/B_1280000_L_1.000000")
(def dataset-config {:min-len 75
                    :batch-size 100000
                    :style :new
                    :count? false
                    :global? false
                    :phantom "george"
                    :thickness "1mm"})

(if (.isDirectory (clojure.java.io/file base-dir))
    (do (timbre/info (format "Loading %s dataset ..." dataset-name))
        (def data_george_1mm (with-open [dataset ^PCTDataset (pct.data.io/
newPCTDataset base-dir "MLP_paths_r=0.bin")]
                              {:x0 (.x0 dataset)
                               :rows (.rows dataset)
                               :cols (.cols dataset)
                               :slice-count (.slices dataset)
                               :slice-offset (pct.data.io/slice-offset dataset)
                               :index (pct.data.io/load-dataset dataset
dataset-config)})))
        (timbre/info (format "Path [%s] does not exist or is not a folder." base-dir)))

    (def x0 (:x0 data_george_1mm))
    (def ^{:tag 'long} _rows (long (:rows data_george_1mm)))
    (def ^{:tag 'long} _cols (long (:cols data_george_1mm)))
    (def ^{:tag 'long} offset (* _rows _cols) )
    (def ^{:tag 'long} slices 64)
    (def samples (range 30 40))
    (def history-index (:index data_george_1mm))
    (def ^{:tag 'long} depth 8)
    (def regions test/George-regions)
    (def recon-opts {:iterations 6
                    :lambda (generate-lambdas 10 (fn [i] (/ 0.00025 (double i))))
                    :tvs? true
                    :tvs-alpha 0.75
                    :tvs-N 5
                    :dump true
                    :grid-depth depth
                    :dataset dataset-config})

    (def ^{:tag 'int} test-runs 5)
    (def recon-type :blocked)

    (spec/def ::recon-types #{:blocked :threaded :sequential})
    (spec/def ::sample-range-check (spec/coll-of (spec/and pos? #(< % slices)))))

    (defn run
      ([] (run test-runs recon-type))
      ([^long n] (run n recon-type))
      ([^long test-runs recon-type]
       {:pre [(int? test-runs)
              (spec/valid? ::recon-types recon-type)
              (spec/valid? ::sample-range-check samples)]}
        (let [s (format "Running test for %s x%d" recon-type test-runs)]
          (println s)
          (timbre/info s))
        (def grid nil)
        (def result nil)
        (loop [i (int 0)]
          (when (< i test-runs)
            (cond
              (= recon-type :blocked)
              (do (def result-folder (format "results/%s/blocked" dataset-name))
```

```

      (def grid (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                :connect? true :slice-offset offset))
      (def result (recon/async-art-blocked grid history-index x0 recon-opts
      )))
(= recon-type :threaded)
(do (def result-folder (format "results/%s/threaded" dataset-name))
    (def grid (-> (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
                  )) :connect? true :slice-offset offset)
      (.trim-connections)))
    (def result (recon/async-art-threaded grid history-index x0
      recon-opts)))

(= recon-type :sequential)
(do (def result-folder (format "results/%s/sequential" dataset-name))
    (def grid (pct.async.node/newAsyncGrid slices (range 1 (inc depth))
              :connect? true :slice-offset offset))
      (def result (recon/seq-art grid history-index x0 recon-opts)))

:else
(do (timbre/info (format "Unknown recon-type: [%s]" recon-type))
    (def result nil)))
(when result
  (let [stats (->> result
                (mapv (fn [[k v]]
                        (when (int? k)
                          [k (test/series-stats (first v) [_rows _cols
                                                            slices] regions samples)]))
                    (remove nil?)
                    (into (sorted-map))
                    (#(assoc % 0 (test/series-stats x0 [_rows _cols slices]
              regions samples))))]
        (pct.data.io/save-result result _rows _cols slices recon-opts
          {:type :all
           :folder result-folder
           :result {:samples samples :stats stats }}}}
    (recur (unchecked-inc-int i))))))

;; (run 5 :blocked)
;; (run 5 :threaded)
;; (run 5 :sequential)

```

APPENDIX B

CTP404 Phantom History Counts

Table B.1: History count for CTP404 dataset slice 0-4

depth	slice 0	1	2	3	4
1	1,642,159	1,030,477	1,148,943	1,230,992	1,316,550
2	2,856,842	2,044,979	2,113,262	2,169,612	2,276,484
3	1,181,811	929,015	902,059	857,023	808,399
4	300,842	252,695	224,590	190,646	171,695
5	50,851	41,961	32,638	27,295	23,005
6	6,210	4,381	3,599	2,998	2,526
7	795	639	559	505	462
8	287	262	235	265	220
9	161	168	171	170	169
10	131	104	144	108	108
11	77	86	99	91	87
12	48	62	58	62	1
13	27	31	42	1	0
14	17	15	0	0	0
15	4	0	0	0	0
16	0	0	0	0	0

Table B.2: History count for CTP404 dataset slice 5-9

depth	slice 5	6	7	8	9
1	1,497,982	1,464,899	1,534,923	1,590,803	1,640,228
2	2,204,908	2,214,878	2,214,256	2,220,360	2,217,781
3	763,688	719,626	676,800	645,514	621,046
4	152,746	141,896	126,409	115,921	105,860
5	19,763	19,031	16,528	14,473	12,543
6	2,374	2,287	1,919	1,789	1,590
7	486	454	484	450	40
8	240	222	288	25	0
9	177	193	13	0	0
10	159	4	0	0	0
11	3	0	0	0	0
12	0	0	0	0	0

Table B.3: History count for CTP404 dataset slice 10-14

depth	slice 10	11	12	13	14
1	1,669,180	1,687,559	1,690,952	1,676,351	1,831,676
2	2,210,438	2,193,430	2,167,218	2,204,030	40,757
3	596,840	574,655	563,305	1,600	0
4	97,110	90,389	79	0	0
5	10,816	45	0	0	0
6	46	0	0	0	0
7	0	0	0	0	0

APPENDIX C

George 2.5mm History Counts

Table C.1: History count for george 2.5mm dataset slice 0-4

depth	slice 0	1	2	3	4
1	1,081,058	592,589	674,088	765,324	845,562
2	1,907,515	1,602,711	1,741,590	1,836,658	1,878,442
3	1,019,720	1,051,590	1,092,503	1,083,198	1,073,829
4	312,601	368,148	382,656	363,161	342,514
5	45,124	72,406	75,281	70,359	52,024
6	2,404	7,327	8,445	6,413	1,123
7	35	378	503	115	67
8	0	7	1	1	0

Table C.2: History count for george 2.5mm dataset slice 5-9

depth	slice 5	6	7	8	9
1	956,191	1,029,353	1,119,870	2,112,609	1,228,058
2	2,014,972	2,079,991	2,192,622	3,157,349	2,559,367
3	1,043,761	825,561	1,284,260	1,003,066	955,706
4	249,941	93,984	288,890	193,615	184,932
5	9,352	16,992	37,885	20,210	18,889
6	894	2,260	2,141	1,162	1,018
7	13	103	72	38	39
8	3	1	3	0	0

Table C.3: History count for george 2.5mm dataset slice 10-14

depth	slice 10	11	12	13	14
1	1,651,904	1,747,187	1,831,990	1,911,923	1,979,197
2	2,680,669	2,704,703	2,729,118	2,738,467	2,738,757
3	902,951	852,751	805,054	759,610	720,412
4	164,907	145,035	127,270	113,215	101,080
5	15,528	12,903	10,971	9,170	7,830
6	813	640	542	399	309
7	21	17	9	10	5
8	0	0	1	0	0

Table C.4: History count for george 2.5mm dataset slice 15-19

depth	slice 15	16	17	18	19
1	2,037,981	2,068,774	2,085,402	2,094,368	2,083,990
2	2,722,657	2,718,509	2,693,248	2,675,493	2,644,146
3	687,370	660,356	642,092	625,924	618,529
4	92,112	84,297	80,819	78,028	79,119
5	7,054	6,276	5,896	5,916	6,058
6	308	253	245	301	325
7	5	2	6	8	0
8	1	0	0	0	0

Table C.5: History count for george 2.5mm dataset slice 20-24

depth	slice 20	21	22	23	24
1	2,058,314	2,013,123	1,957,086	1,883,763	1,999,383
2	2,608,122	2,568,175	2,520,825	2,605,006	80,178
3	617,814	621,977	651,651	8,333	0
4	81,023	86,266	231	0	0
5	6,571	0	0	0	0
6	0	0	0	0	0

APPENDIX D

George 1mm History Counts

Table D.1: History count for george 1mm dataset slice 0-6

depth	slice 0	1	2	3	4	5	6
1	0	86,801	51,791	56,376	57,506	61,631	67,973
2	121,642	285,603	230,390	250,046	264,480	270,611	283,112
3	82,040	378,826	266,466	283,459	301,416	308,898	310,546
4	62,260	378,072	267,947	267,582	277,081	289,932	290,252
5	35,811	307,974	229,689	232,137	230,430	235,491	243,874
6	16,969	213,626	174,622	177,208	172,565	174,041	178,860
7	6,283	133,732	120,576	120,045	118,723	116,936	119,557
8	1,889	70,844	73,219	73,582	72,441	73,000	73,538
9	454	27,539	38,754	39,178	39,924	41,602	39,730
10	95	7,957	16,433	19,201	20,281	20,600	20,433
11	5	1,856	5,708	8,828	8,735	9,144	9,523
12	1	354	1,630	3,042	3,500	3,748	3,863
13	0	46	338	969	1,210	1,342	1,512
14	0	1	63	264	383	472	426
15	0	0	11	52	106	145	136
16	0	0	0	13	18	37	38
17	0	0	0	3	8	4	4
18	0	0	0	0	0	0	0

Table D.2: History count for george 1mm dataset slice 7-13

depth	slice 7	8	9	10	11	12	13
1	73,685	79,618	84,442	89,215	93,265	97,389	107,468
2	297,746	309,880	324,858	337,911	337,908	366,718	381,597
3	315,170	326,113	333,040	334,783	339,337	361,711	366,963
4	293,638	293,741	293,688	291,453	296,282	309,939	308,294
5	243,084	240,618	235,761	232,457	238,857	239,646	237,089
6	179,794	178,035	171,778	170,366	171,275	169,063	166,440
7	120,862	116,248	115,043	112,342	111,383	109,179	104,710
8	71,359	70,825	69,317	66,952	65,814	62,514	57,416
9	39,790	39,064	37,015	36,889	34,796	32,394	20,093
10	19,933	19,389	19,148	17,724	16,459	10,549	3,184
11	8,889	8,835	8,438	7,845	5,163	1,445	727
12	3,737	3,348	3,348	2,264	711	369	577
13	1,301	1,192	984	305	146	187	198
14	447	323	95	80	63	68	34
15	121	35	26	23	20	12	4
16	9	11	6	4	4	0	0
17	2	9	0	2	0	1	0
18	1	0	0	0	0	0	0

Table D.3: History count for george 1mm dataset slice 14-20

depth	slice 14	15	16	17	18	19	20
1	110,541	116,049	122,265	127,043	134,481	137,760	197,312
2	390,950	401,828	411,511	427,216	438,171	440,601	838,474
3	368,045	375,472	382,978	387,475	387,727	533,666	878,047
4	307,648	309,964	308,786	303,254	321,362	517,547	650,350
5	235,493	231,358	221,084	199,144	230,094	388,892	327,468
6	161,221	152,042	117,512	97,866	146,502	206,983	184,562
7	97,416	66,048	41,904	55,350	81,852	106,866	100,261
8	37,592	18,048	18,683	30,494	42,971	54,087	48,672
9	6,937	6,381	11,706	12,654	20,800	25,818	21,112
10	2,294	4,241	3,941	5,744	10,885	9,794	8,465
11	1,575	1,572	1,137	2,775	4,288	3,374	3,074
12	563	267	500	1,318	1,285	1,129	1,026
13	109	57	203	431	362	315	326
14	12	26	115	130	96	94	82
15	0	15	24	22	24	17	19
16	1	10	4	3	3	10	5
17	2	0	0	0	0	3	0
18	0	0	1	0	1	0	0

Table D.4: History count for george 1mm dataset slice 21-27

depth	slice 21	22	23	24	25	26	27
1	269,692	215,854	149,511	201,735	212,991	220,850	227,530
2	671,346	392,340	521,640	604,861	622,902	638,205	650,630
3	590,766	367,448	482,683	501,091	506,423	513,536	512,025
4	362,474	302,700	365,627	360,460	358,851	355,933	356,434
5	239,867	209,954	243,007	232,706	227,363	226,915	220,028
6	138,983	129,760	145,162	136,260	134,186	128,837	126,027
7	75,604	69,879	77,040	72,746	68,501	67,651	64,463
8	37,022	32,952	37,447	33,653	32,403	30,920	29,140
9	15,845	14,144	15,399	14,233	13,567	12,369	12,159
10	6,424	5,377	5,851	5,422	4,980	4,741	4,168
11	2,162	1,777	2,100	1,778	1,800	1,552	1,396
12	706	590	618	574	508	488	442
13	238	145	200	153	145	146	112
14	62	41	42	57	35	45	31
15	16	7	8	6	6	6	10
16	3	1	1	1	2	0	1
17	2	2	1	0	0	1	1
18	0	0	0	0	0	0	0

Table D.5: History count for george 1mm dataset slice 28-34

depth	slice 28	29	30	31	32	33	34
1	237,132	241,363	250,354	256,739	261,831	272,019	274,392
2	662,593	671,627	683,735	693,022	703,816	714,112	718,507
3	521,990	519,172	521,896	527,955	524,723	529,308	534,848
4	352,676	349,775	350,541	345,135	341,971	345,917	335,754
5	217,845	214,360	210,258	207,891	203,438	198,677	195,619
6	121,910	118,363	115,353	113,073	107,592	105,433	103,372
7	61,165	60,122	57,267	53,917	52,612	50,666	48,496
8	27,724	26,390	24,958	23,867	22,448	21,600	20,486
9	11,043	10,412	9,990	9,006	8,815	8,248	7,670
10	4,024	3,769	3,552	3,312	3,168	2,975	2,715
11	1,328	1,220	1,161	1,157	993	949	853
12	399	368	327	312	289	261	243
13	121	111	100	79	77	63	71
14	23	14	20	22	11	16	18
15	9	1	1	3	1	3	3
16	2	0	1	1	0	0	0
17	0	0	0	0	0	1	0
18	0	0	0	0	0	0	0

Table D.6: History count for george 1mm dataset slice 35-41

depth	slice 35	36	37	38	39	40	41
1	278,302	291,109	288,392	298,249	304,757	299,916	314,481
2	732,469	737,126	739,667	750,709	748,899	760,759	760,276
3	525,581	535,164	528,526	524,786	536,254	521,922	535,384
4	334,699	335,391	327,221	330,939	322,294	319,908	330,008
5	193,133	190,220	188,809	181,778	182,132	181,103	177,784
6	100,088	99,894	94,276	92,690	93,003	88,178	88,412
7	47,161	44,609	43,386	42,925	41,106	39,910	39,619
8	19,089	18,722	17,966	17,009	16,619	16,312	15,277
9	7,338	6,958	6,558	6,306	6,075	5,745	5,654
10	2,546	2,401	2,277	2,156	2,146	2,024	1,946
11	781	789	728	671	712	632	571
12	218	193	209	187	180	163	203
13	56	46	40	39	53	47	41
14	12	7	18	14	5	8	5
15	4	2	3	2	2	1	2
16	0	0	2	1	1	0	0
17	0	1	0	0	1	0	0
18	0	0	0	0	0	0	0

Table D.7: History count for george 1mm dataset slice 42-48

depth	slice 42	43	44	45	46	47	48
1	303,096	307,761	316,488	308,379	321,262	311,170	310,353
2	749,659	765,980	761,331	762,986	764,668	754,726	761,454
3	530,415	518,682	532,675	519,263	523,035	523,793	513,423
4	316,431	316,449	317,429	311,619	317,681	307,380	308,273
5	176,330	172,777	172,841	172,763	168,348	167,384	169,291
6	86,518	85,165	86,065	82,883	81,939	82,923	82,075
7	37,663	37,756	37,145	36,293	36,360	35,823	36,350
8	15,171	14,789	14,405	14,728	14,006	14,525	14,199
9	5,325	5,400	5,296	5,332	5,098	5,232	5,335
10	1,891	1,739	1,744	1,799	1,680	1,830	1,809
11	686	580	598	567	612	603	634
12	201	180	164	168	169	194	191
13	58	31	45	40	66	52	60
14	4	6	14	9	12	8	9
15	1	1	0	3	1	3	1
16	0	0	1	0	0	1	0
17	0	0	0	0	0	0	0

Table D.8: History count for george 1mm dataset slice 49-55

depth	slice 49	50	51	52	53	54	55
1	318,318	305,259	310,432	303,797	302,372	305,859	289,986
2	755,031	746,721	746,145	736,190	739,020	727,817	715,005
3	518,607	509,544	510,014	509,584	500,553	504,566	492,406
4	311,088	305,949	306,960	301,757	304,341	303,708	300,440
5	167,476	168,221	163,556	165,240	168,061	167,489	165,947
6	82,566	82,219	80,661	82,230	83,702	82,979	83,597
7	36,121	36,424	36,063	36,716	37,575	37,968	38,322
8	14,277	14,645	14,521	14,540	15,438	15,535	17,284
9	5,308	5,617	5,407	5,595	5,857	6,247	4
10	1,791	1,946	1,902	1,954	2,190	0	0
11	589	635	592	670	0	0	0
12	196	199	213	0	0	0	0
13	62	38	0	0	0	0	0
14	11	0	0	0	0	0	0
15	0	0	0	0	0	0	0

Table D.9: History count for george 1mm dataset slice 56-62

depth	slice 56	57	58	59	60	61	62
1	293,070	284,276	280,599	276,956	262,933	262,008	272,391
2	711,625	699,614	695,862	678,282	661,411	670,867	21,025
3	496,819	485,895	482,322	480,247	525,910	13,429	0
4	300,972	293,414	301,375	349,062	6,970	0	0
5	166,069	166,940	197,679	2,698	0	0	0
6	84,533	95,894	795	0	0	0	0
7	42,807	191	0	0	0	0	0
8	40	0	0	0	0	0	0
9	0	0	0	0	0	0	0

BIBLIOGRAPHY

- [1] Y. Censor, D. Gordon, and R. Gordon, “Bicav: a block-iterative parallel algorithm for sparse systems with pixel-related weighting,” *IEEE Transactions on Medical Imaging*, vol. 20, no. 10, pp. 1050–1060, 2001.
- [2] Y. Censor, T. Elfving, G. T. Herman, and T. Nikazad, “On diagonally relaxed orthogonal projection methods,” *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 473–504, 2008. [Online]. Available: <https://doi.org/10.1137/050639399>
- [3] A. Andersen and A. Kak, “Simultaneous algebraic reconstruction technique (sart): A superior implementation of the art algorithm,” *Ultrasonic Imaging*, vol. 6, no. 1, pp. 81–94, 1984. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0161734684900087>
- [4] C. A. R. Hoare, “Monitors: An operating system structuring concept,” *Commun. ACM*, vol. 17, no. 10, p. 549–557, oct 1974. [Online]. Available: <https://doi.org/10.1145/355620.361161>
- [5] E. Gomez, K. E. Schubert, and R. Cai, “A model for entropy of parallel execution,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 555–560.
- [6] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978. [Online]. Available: <https://doi.org/10.1145/359576.359585>
- [7] M. Witt, B. E. Schultze, R. W. Schulte, K. E. Schubert, and E. Gomez, “A proton simulator for testing implementations of proton CT reconstruction algorithms on GPGPU clusters,” in *Proceedings of the IEEE Nuclear Science Symposium & Medical Imaging Conference (NSS/MIC) 2012*, 2012, pp. 4329–4334.
- [8] P. Karbasi, R. Cai, B. Schultze, H. Nguyen, J. Reed, P. Hall, V. Giacometti, V. Bashkirov, R. Johnson, N. Karonis, J. Olafsen, C. Ordonez, K. E. Schubert, and R. W. Schulte, “A highly accelerated parallel multi-gpu based reconstruction algorithm for generating accurate relative stopping powers,” in *2017 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2017, pp. 1–4.

- [9] R. Gordon, R. Bender, and G. T. Herman, “Algebraic reconstruction techniques (art) for three-dimensional electron microscopy and x-ray photography,” *Journal of Theoretical Biology*, vol. 29, no. 3, pp. 471–481, 1970. [Online]. Available: [https://doi.org/10.1016/0022-5193\(70\)90109-8](https://doi.org/10.1016/0022-5193(70)90109-8)
- [10] S. Penfold, R. Schulte, Y. Censor, V. Bashkirov, S. McAllister, K. Schubert, and A. Rosenfeld, “Block-iterative and string-averaging projection algorithms in proton computed tomography image reconstruction,” 2010.
- [11] P. Karbasi, B. Schultze, V. Giacometti, T. Plautz, K. Schubert, R. Schulte, and V. Bashkirov, “Incorporating robustness in diagonally-relaxed orthogonal projections method for proton computed tomography,” *2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, pp. 1–4, 2015.
- [12] M. Jiang and G. Wang, “Convergence of the simultaneous algebraic reconstruction technique (sart),” *IEEE Transactions on Image Processing*, vol. 12, no. 8, pp. 957–961, 2003. [Online]. Available: <https://doi.org/10.1109/tip.2003.815295>
- [13] Y. Censor and E. Tom, “Convergence of string-averaging projection schemes for inconsistent convex feasibility problems,” *Optimization Methods and Software*, vol. 18, no. 5, pp. 543–554, 2003. [Online]. Available: <https://doi.org/10.1080/10556780310001610484>
- [14] K. E. Schubert, R. Cai, E. Gomez, and P. J. Boston, “Using extremophile behavior to identify biological targets of opportunity,” in *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 9 2017, p. nil. [Online]. Available: <https://doi.org/10.1109/smc-it.2017.13>
- [15] H. Heaton and Y. Censor, “Asynchronous sequential inertial iterations for common fixed points problems with an application to linear systems,” *Journal of Global Optimization*, vol. 74, no. 1, pp. 95–119, 2019. [Online]. Available: <https://doi.org/10.1007/s10898-019-00747-4>
- [16] L. Elsner, I. Koltracht, and M. Neumann, “Convergence of sequential and asynchronous nonlinear paracontractions,” *Numerische Mathematik*, vol. 62, no. 1, pp. 305–319, 1992. [Online]. Available: <https://doi.org/10.1007/bf01396232>
- [17] R. Wilson, “Radiological use of fast protons,” *Radiology*, pp. 487–491, 1946.

- [18] J. Hu, X. Zhao, and F. Wang, “An extended simultaneous algebraic reconstruction technique (e-sart) for x-ray dual spectral computed tomography,” *Scanning*, vol. 38, no. 6, pp. 599–611, 2016. [Online]. Available: <https://doi.org/10.1002/sca.21306>
- [19] A. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*, ser. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2001, ch. 7, pp. 275–296. [Online]. Available: https://books.google.com/books?id=Z6RpVjb9_lwC
- [20] Paul Alcorn, “Arm details neoverse v1 and n2 platforms, new mesh design,” 2021, [Online; accessed 3-July-2021]. [Online]. Available: <https://www.tomshardware.com/news/arm-details-neoverse-v1-and-n2-platforms-new-mesh-design>
- [21] Hassan Mujtaba, “Amd epyc genoa cpu platform detailed – up to 96 zen 4 cores, 192 threads, 12-channel ddr5-5200, 128 pcie gen 5 lanes, sp5 ‘lga 6096’ socket,” 2021, [Online; accessed 3-July-2021]. [Online]. Available: <https://wccfttech.com/amd-epyc-genoa-cpu-platform-detailed-up-to-96-zen-4-cores-12-channel-ddr5-5200-sp5-lga-6096-socket/>
- [22] Chris Bergey, SVP and GM, Infrastructure Line of Business, Arm, “Transforming compute for next-generation infrastructure,” 2021, [Online; accessed 3-July-2021]. [Online]. Available: <https://www.arm.com/company/news/2021/04/transforming-compute-for-next-generation-infrastructure>
- [23] Y. Censor, D. Gordon, and R. Gordon, “Component averaging: an efficient iterative parallel algorithm for large and sparse unstructured problems,” *Parallel Computing*, vol. 27, no. 6, pp. 777–808, 2001. [Online]. Available: [https://doi.org/10.1016/s0167-8191\(00\)00100-9](https://doi.org/10.1016/s0167-8191(00)00100-9)
- [24] D. Gordon, “The cimmino-kaczmarz equivalence and related results,” *Applied Analysis & Optimization*, vol. 2, no. 2, pp. 253–370, 2018. [Online]. Available: <http://yokohamapublishers.jp/online2/opaa/vol2/p253.html>
- [25] D. Fischer, “Paradoxes in parallel processing (abstract),” in *Proceedings of the 1990 ACM annual conference on Cooperation - CSC '90*, - 1990, p. nil. [Online]. Available: <https://doi.org/10.1145/100348.100462>
- [26] L. Gubin, B. Polyak, and E. Raik, “The method of projections for finding the common point of convex sets,” *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 6, pp. 1–24, 1967. [Online]. Available: [https://doi.org/10.1016/0041-5553\(67\)90113-9](https://doi.org/10.1016/0041-5553(67)90113-9)
- [27] L. Bregman, “The method of successive projections for finding a common point of convex sets,” *Soviet Mathematics Doklady*, vol. 6, pp. 688–692, 1965.

- [28] H. Kong and J. Pan, “An improved ordered-subset simultaneous algebraic reconstruction technique,” in *2009 2nd International Congress on Image and Signal Processing*, 2009, pp. 1–5.
- [29] G. Wang and M. Jiang, “Ordered-subset simultaneous algebraic reconstruction techniques (os-sart),” *Journal of X-ray Science and Technology*, vol. 12, no. 3, pp. 169–177, 2004.
- [30] C. Tschalär, “Straggling distributions of extremely large energy losses,” *Nucl. Instrum. Methods*, vol. 61, pp. 141–156, 1968.
- [31] International Commission on Radiation Units and Measurements, “Stopping powers and ranges for protons and alpha particles,” *ICRU Report*, vol. 49, 1993.
- [32] W. R. Leo, *Techniques for Nuclear and Particle Physics Experiments*, 2nd ed. Springer, 1994.
- [33] B. Schaffner and E. Pedroni, “The precision of proton range calculations in proton radiotherapy treatment planning: experimental verification of the relation between CT-HU and proton stopping power,” *Phys. Med. Bio.*, vol. 43, pp. 1579–1592, 1998.
- [34] Y. Chen, E. Gomez, R. F. Hurley, Y. Nie, K. E. Schubert, and R. W. Schulte, “Accurate proton beam localization,” in *Proceedings of The 2012 International Conference on Bioinformatics and Computational Biology (BIO-COMP’12)*, 2012, pp. 213–217.
- [35] C.-A. C. Fekete, P. Doolan, M. F. Dias, L. Beaulieu, and J. Seco, “Developing a phenomenological model of the proton trajectory within a heterogeneous medium required for proton imaging,” *Phys. Med. Bio.*, vol. 60, no. 13, pp. 5071–5082, 2015. [Online]. Available: <http://stacks.iop.org/0031-9155/60/i=13/a=5071>
- [36] G. N. Hounsfield, “Method of and apparatus for examining a body by radiation such as X or gamma radiation,” *U.S. Patent and Trademark Office*, no. US 3919552, Nov 1975.
- [37] —, “Method of and apparatus for examining a body by radiation such as X or gamma radiation,” U.S. Patent 3919552, Nov 11, 1975.
- [38] G. Coutrakon, J. Hubbard, J. Johanning, G. Maudsley, T. Slaton, and P. Morton, “A performance study of the loma linda proton medical accelerator,” *Med. Phys.*, vol. 21, pp. 1691–1701, 1994.

- [39] M. Bruzzi, N. Blumenkrantz, J. Feldt, J. Heimann, H. F.-W. Sadrozinski, A. Seiden, D. Williams, V. A. Bashkirov, R. W. Schulte, D. Menichelli, M. Scaringella, G. Cirrone, G. Cuttone, N. Randazzo, V. Sipala, and D. L. Presti, "Prototype tracking studies for proton CT," *IEEE Transactions on Nuclear Science*, vol. 54, pp. 140–145, Feb 2007.
- [40] G. Cuttone, G. Cirrone, G. Candiano, F. D. Rosa, G. Russo, N. Randazzo, V. Sipala, S. L. Nigro, D. L. Presti, J. Feldt, J. Heimann, H. F.-W. Sadrozinski, A. Seiden, D. Williams, V. A. Bashkirov, R. W. Schulte, M. Bruzzi, and D. Menichelli, "Monte Carlo studies of a proton computed tomography system," *IEEE Transactions on Nuclear Science*, vol. 54, pp. 1487–1491, Oct 2007.
- [41] J. Missaghian, R. F. Hurley, V. A. Bashkirov, B. Colby, V. Rykalin, S. Kachigiun, D. Fusi, R. W. Schulte, M. F. Martinez, H. F.-W. Sadrozinski, and S. N. Penfold, "Beam test results of a csi calorimeter matrix element," *JINST*, vol. 5, p. P06001, 2010.
- [42] R. F. Hurley, R. W. Schulte, V. A. Bashkirov, G. Coutrakon, H. F.-W. Sadrozinski, and B. Patyal, "The phase i proton CT scanner and test beam results at llumc," *Trans. Am. Nucl. Soc.*, vol. 106, pp. 63–66, 2012.
- [43] R. F. Hurley, R. W. Schulte, V. A. Bashkirov, A. Wroe, A. Ghebremedhin, H. F.-W. Sadrozinski, V. Rykalin, G. Coutrakon, P. Koss, and B. Patyal, "Water-equivalent path length calibration of a prototype proton CT scanner," *Med. Phys.*, vol. 39, pp. 2438–2446, 2012.
- [44] R. W. Schulte, V. A. Bashkirov, R. P. Johnson, H. F.-W. Sadrozinski, and K. E. Schubert, "Overview of the llumc/ucsc/csusb phase 2 proton CT project," in *Trans. Am. Nucl. Soc.*, vol. 106, 2012, pp. 59–62.
- [45] H. F.-W. Sadrozinski, R. P. Johnson, S. Macafee, A. Plumb, D. Steinberg, A. Zatserklyaniy, V. A. Bashkirov, R. F. Hurley, and R. W. Schulte, "Development of a head scanner for proton CT," *Nucl. Instrum. Methods Phys. Res. A*, vol. 699, pp. 205–210, 2013.
- [46] M. Scaringella, M. Bruzzi, M. Bucciolini, M. Carpinelli, G. A. P. Cirrone, C. Civinini, G. Cuttone, D. L. Presti, S. Pallotta, C. Pugliatti, N. Randazzo, F. Romano, V. Sipala, C. Stancampiano, C. Talamonti, E. Vanzi, and M. Zani, "A proton computed tomography based medical imaging system," *Journal of Instrumentation*, vol. 9, no. 12, p. C12009, 2014. [Online]. Available: <http://stacks.iop.org/1748-0221/9/i=12/a=C12009>

- [47] V. A. Bashkirov, R. P. Johnson, H. F.-W. Sadrozinski, and R. W. Schulte, “Development of proton computed tomography detectors for applications in hadron therapy,” *Nucl. Instrum. Methods Phys. Res. A*, vol. 809, pp. 120–129, 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0168900215009274>
- [48] T. E. Plautz, V. A. Bashkirov, V. Giacometti, R. F. Hurley, R. P. Johnson, P. Piersimoni, H. F.-W. Sadrozinski, R. W. Schulte, and A. Zatserklyaniy, “An evaluation of spatial resolution of a prototype proton CT scanner,” *Med. Phys.*, vol. 43, no. 12, pp. 6291–6300, Dec 2016. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5097050/>
- [49] R. P. Johnson, V. A. Bashkirov, G. Coutrakon, V. Giacometti, P. Karbasi, N. T. Karonis, C. Ordoñez, M. Pankuch, H. F.-W. Sadrozinski, K. E. Schubert, and R. W. Schulte, “Results from a prototype proton-CT head scanner,” *Conference on the Application of Accelerators in Research and Industry, CAARI 2016, 30 October - 4 November 2016, Ft. Worth, TX, USA*, Jul 2017. [Online]. Available: <https://arxiv.org/pdf/1707.01580>
- [50] R. P. Johnson, V. A. Bashkirov, L. DeWitt, V. Giacometti, R. F. Hurley, P. Piersimoni, T. E. Plautz, H. F.-W. Sadrozinski, K. E. Schubert, R. W. Schulte, B. E. Schultze, and A. Zatserklyaniy, “A fast experimental scanner for proton CT: Technical performance and first experience with phantom scans,” *IEEE Transactions on Nuclear Science*, vol. 63, pp. 52–60, 2016.
- [51] R. W. Schulte, S. N. Penfold, J. Tafas, and K. E. Schubert, “A maximum likelihood proton path formalism for application in proton computed tomography,” *Med. Phys.*, vol. 35, pp. 4849–4856, Nov 2008.
- [52] V. A. Bashkirov, R. W. Schulte, G. Coutrakon, B. Erdelyi, K. Wong, H. F.-W. Sadrozinski, S. N. Penfold, A. B. Rosenfeld, S. A. McAllister, and K. E. Schubert, “Development of Proton Computed Tomography for Applications in Proton Therapy,” in *Application of Accelerators in Research and Industry: Twentieth International Conference*, F. D. McDaniel and B. L. Doyle, Eds., vol. AIP Conference Proceedings Volume 1099. Fort Worth (Texas): American Institute of Physics, Aug 10-15 2008, pp. 460–463, iSBN: 978-0-7354-0633-9.
- [53] S. N. Penfold, A. B. Rosenfeld, R. W. Schulte, and K. E. Schubert, “A more accurate reconstruction system matrix for quantitative proton computed tomography,” *Med. Phys.*, vol. 36, no. 10, pp. 4511–4518, Oct 2009.
- [54] S. A. McAllister, K. E. Schubert, R. W. Schulte, and S. N. Penfold, “General purpose graphics processing unit speedup of integral relative electron density calculation for proton computed tomography,” in *Proceedings of the IEEE High Performance Medical Imaging Workshop 2009*, 2009.

- [55] K. N. Kutulakos and S. M. Seitz, “A theory of shape by space carving,” in *Proceedings of the Seventh International Conference on Computer Vision (ICCV)*, 1999, pp. 307–314.
- [56] —, “A theory of shape by space carving,” *International Journal of Computer Vision*, vol. 38, no. 3, pp. 199–218, Marr Prize Special Issue 2000.
- [57] S. Vedula, S. Baker, S. Seitz, and T. Kanade, “Shape and motion carving in 6d,” in *Proceedings of Computer Vision and Pattern Recognition Conference (CVPR)*, 2000.
- [58] W. Niem, “Robust and fast modelling of 3D natural objects from multiple views,” in *SPIE Proceedings Image and Video Processing*, vol. 2182, no. II, 1994, pp. 388–397.
- [59] —, “Error analysis for silhouette-based 3D shape estimation from multiple views,” in *Proceedings of International Workshop on Synthetic-Natural Hybrid Coding and Three-Dimensional Imaging*, 1997, pp. 143–146.
- [60] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, Nov 1986.
- [61] B. E. Schultze, M. Witt, K. E. Schubert, and R. W. Schulte, “Space carving and filtered back projection as preconditioners for proton computed tomography reconstruction,” in *Proceedings of the IEEE Nuclear Science Symposium & Medical Imaging Conference (NSS/MIC) 2012*, 2012, pp. 4335–4340.
- [62] B. E. Schultze, M. Witt, Y. Censor, K. E. Schubert, and R. W. Schulte, “Performance of hull-detection algorithms for proton computed tomography reconstruction,” in *Infinite Products of Operators and Their Applications*, ser. Contemporary Mathematics, S. Reich and A. Zaslavski, Eds., vol. 636. American Mathematical Society, 2015, pp. 211–224.
- [63] J. Radon, “Über die bestimmung von funktionen durch ihre intergralwerte langsgewisser mannigfaltigkeiten (on the determination of functions from their integrals along certain manifolds),” *Berichte Saechsische Akademie der Wissenschaften*, vol. 29, pp. 262–277, 1917.
- [64] R. Bracewell and A. Riddle, “Inversion of fan beam scawns in radio astronomy,” *Astrophysics Journal*, vol. 150, pp. 427–434, 1967.
- [65] G. Ramachandran and A. Lakshminarayanan, “Three dimensional reconstructions from radiographs and electron micrographs: Application of convolution instead of Fourier transforms,” *Proc. Natl. Acad. Sci.*, vol. 68, pp. 2236–2240, 1971.

- [66] A. Lakshminarayanan, “Reconstruction from divergent ray data,” Department of Computer Science, State University of New York at Buffalo, Tech. Rep., 1975.
- [67] L. Feldkamp, L. Davis, and J. Kress, “Practical cone-beam algorithms,” *J. Opt. Soc. Am.*, vol. A1, pp. 612–619, 1984.
- [68] L. Shepp and B. Logan, “The Fourier reconstruction of a head section,” *IEEE Transactions on Nuclear Science*, vol. NS-21, pp. 21–43, 1974.
- [69] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*. New York: IEEE Press, 1988.
- [70] Y. Censor, T. Elfving, G. T. Herman, and T. Nikazad, “On diagonally relaxed orthogonal projection methods,” *SIAM J. Sci. Comput.*, vol. 30, no. 1, pp. 473–504, 2008. [Online]. Available: <https://doi.org/10.1137/050639399>
- [71] G. T. Herman and R. Davidi, “Image reconstruction from a small number of projections,” *Inverse Problems*, vol. 24, no. 4, p. 045011, 2008. [Online]. Available: <http://stacks.iop.org/0266-5611/24/i=4/a=045011>
- [72] S. N. Penfold, R. W. Schulte, Y. Censor, V. A. Bashkirov, S. A. McAllister, K. E. Schubert, and A. B. Rosenfeld, “Block-iterative and string-averaging projection algorithms in proton computed tomography image reconstruction,” in *Biomedical Mathematics: Promising Directions in Imaging, Therapy Planning and Inverse Problems*, Y. Censor, M. Jiang, and G. Wang, Eds., The Huangguoshu International Interdisciplinary Conference. Madison, WI, USA: Med. Phys., 2010, pp. 347–367.
- [73] S. N. Penfold, “Image Reconstruction and Monte Carlo Simulations in the Development of Proton Computed Tomography for Applications in Proton Radiation Therapy,” Ph.D. dissertation, University of Wollongong, Australia, 2010.
- [74] S. N. Penfold and Y. Censor, “Techniques in iterative proton CT image reconstruction,” *Sensing and Imaging*, vol. 16, no. 1, Oct 2015. [Online]. Available: <https://doi.org/10.1007/s11220-015-0122-3>
- [75] M. Bruzzi, C. Civinini, M. Scaringella, D. Bonanno, M. Brianzi, M. Carpinelli, G. Cirrone, G. Cuttone, D. L. Presti, G. Maccioni, S. Pallotta, N. Randazzo, F. Romano, V. Sipala, C. Talamonti, and E. Vanzi, “Proton computed tomography images with algebraic reconstruction,” *Nucl. Instrum. Methods Phys. Res. A*, vol. 845, pp. 652–655, May 2017, Proceedings of the Vienna Conference on Instrumentation 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0168900216304454>

- [76] B. E. Schultze, P. Karbasi, V. Giacometti, T. E. Plautz, K. E. Schubert, and R. W. Schulte, “Reconstructing highly accurate relative stopping powers in proton computed tomography,” in *Proceedings of the IEEE Nuclear Science Symposium & Medical Imaging Conference (NSS/MIC) 2015*, Oct 2015, pp. 1–3.
- [77] R. P. Johnson, “Review of medical radiography and tomography with proton beams,” *Reports on Progress in Physics*, vol. 81, no. 1, p. 016701, 2018. [Online]. Available: <http://stacks.iop.org/0034-4885/81/i=1/a=016701>
- [78] R. Davidi, G. T. Herman, and Y. Censor, “Perturbation-resilient block-iterative projection methods with application to image reconstruction from projections,” *International Transactions in Operational Research*, vol. 16, no. 4, pp. 505–524, 2009. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1475-3995.2009.00695.x>
- [79] G. T. Herman, E. Garduño, R. Davidi, and Y. Censor, “Superiorization: An optimization heuristic for medical physics,” *Med. Phys.*, vol. 39, no. 9, pp. 5532–5546, 2012. [Online]. Available: <http://dx.doi.org/10.1118/1.4745566>
- [80] Y. Censor, “Weak and strong superiorization: Between feasibility-seeking and minimization,” *Analele Stiintifice ale Universitatii Ovidius Constanta, Seria Matematica*, vol. 23, pp. 41–54, Oct 2014.
- [81] Y. Censor, R. Davidi, and G. T. Herman, “Perturbation resilience and superiorization of iterative algorithms,” *Inverse problems*, vol. 26, p. 65008, Jun 2010.
- [82] Y. Censor, R. Davidi, G. T. Herman, R. W. Schulte, and L. Tetrushvili, “Projected subgradient minimization versus superiorization,” *Journal of Optimization Theory and Applications*, vol. 160, no. 3, pp. 730–747, Mar 2014. [Online]. Available: <https://doi.org/10.1007/s10957-013-0408-3>
- [83] Y. Censor, G. T. Herman, and M. Jiang, “Superiorization: Theory and applications,” Special Issue of *Inverse Problems*, vol. 33, no. 4, p. 040301, 2017. [Online]. Available: <http://stacks.iop.org/0266-5611/33/i=4/a=040301>
- [84] A. Chambolle, V. Caselles, M. Novaga, D. Cremers, and T. Pock, “An introduction to total variation for image analysis,” in *Theoretical Foundations and Numerical Methods for Sparse Recovery*, De Gruyter, 2010.
- [85] Y. Censor, “Can linear superiorization be useful for linear optimization problems?” *Inverse Problems*, vol. 33, no. 4, p. 044006, 2017. [Online]. Available: <http://stacks.iop.org/0266-5611/33/i=4/a=044006>

- [86] T. Humphries, J. Winn, and A. Faridani, “Superiorized algorithm for reconstruction of CT images from sparse-view and limited-angle polyenergetic data,” *Phys. Med. Bio.*, vol. 62, no. 16, pp. 6762–6783, 2017. [Online]. Available: <http://stacks.iop.org/0031-9155/62/i=16/a=6762>
- [87] E. Helou, M. Zibetti, and E. Miqueles, “Superiorization of incremental optimization algorithms for statistical tomographic image reconstruction,” *Inverse Problems*, vol. 33, no. 4, p. 044010, 2017. [Online]. Available: <http://stacks.iop.org/0266-5611/33/i=4/a=044010>
- [88] E. Garduño and G. T. Herman, “Computerized tomography with total variation and with shearlets,” *Inverse Problems*, vol. 33, no. 4, p. 044011, 2017. [Online]. Available: <http://stacks.iop.org/0266-5611/33/i=4/a=044011>
- [89] Q. Yang, W. Cong, and G. Wang, “Superiorization-based multi-energy CT image reconstruction,” *Inverse Problems*, vol. 33, no. 4, p. 044014, 2017. [Online]. Available: <http://stacks.iop.org/0266-5611/33/i=4/a=044014>
- [90] O. Langthaler, “Incorporation of the superiorization methodology into biomedical imaging software,” Sep 2014, 76 pages. [Online]. Available: <https://static1.squarespace.com/static/559921a3e4b02c1d7480f8f4/t/585c49d6725e25be085071c7/1482443225511/Langthaler.pdf>
- [91] B. Prommegger, “Verification and evaluation of superiorized algorithms used in biomedical imaging: Comparison of iterative algorithms with and without superiorization for image reconstruction from projections,” Oct 2014, 84 pages. [Online]. Available: <https://static1.squarespace.com/static/559921a3e4b02c1d7480f8f4/t/585c49bc8419c2c4f892861b/1482443201138/Prommegger.pdf>
- [92] C. Havas, “Revised implementation and empirical study of maximum likelihood expectation maximization algorithms with and without superiorization in image reconstruction,” Oct 2016, 49 pages. [Online]. Available: https://static1.squarespace.com/static/559921a3e4b02c1d7480f8f4/t/596c97aad1758e1c6808c0fa/1500288944245/Havas+Clemens_615.pdf
- [93] S. N. Penfold, R. W. Schulte, Y. Censor, and A. B. Rosenfeld, “Total variation superiorization schemes in proton computed tomography image reconstruction,” *Med. Phys.*, vol. 37, pp. 5887–5895, 2010.
- [94] D. Butnariu, R. Davidi, G. T. Herman, and I. G. Kazantsev, “Stable convergence behavior under summable perturbations of a class of projection methods for convex feasibility and optimization problems,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 1, no. 4, pp. 540–547, Dec 2007.

- [95] Y. Censor, “Superiorization and perturbation resilience of algorithms: A bibliography compiled and continuously updated,” <http://math.haifa.ac.il/yair/bib-superiorization-censor.html>.
- [96] U. Linz, *Ion Beam Therapy: Fundamentals, Technologies, and Clinical Applications*, ser. Biological and Medical Physics, Biomedical Engineering. Springer, 2012, vol. 320.
- [97] R. R. Wilson, “Radiological use of fast protons,” *Radiology*, vol. 47, no. 5, pp. 487–491, 1946, PMID: 20274616.
- [98] A. M. Cormack, “Representation of a function by its line integrals, with some radiological applications,” *Journal of Applied Physics*, vol. 34, no. 9, pp. 2722–7, Sep 1963.
- [99] —, “Representation of a function by its line integrals, with some radiological applications. ii,” *Journal of Applied Physics*, vol. 35, no. 10, pp. 2908–13, Oct 1964.
- [100] A. Cormack and A. Koehler, “Quantitative proton tomography: preliminary experiments,” *Phys. Med. Bio.*, vol. 21, no. 4, pp. 560–569, 1976. [Online]. Available: <http://stacks.iop.org/0031-9155/21/i=4/a=007>
- [101] K. M. Hanson, J. N. Bradbury, T. M. Cannon, R. L. Hutson, D. B. Laubacher, R. Macek, M. A. Paciotti, and C. A. Taylor, “The application of protons to computed tomography,” *IEEE Transactions on Nuclear Science*, vol. 25, no. 1, pp. 657–660, Feb 1978.
- [102] K. M. Hanson, “Proton computed tomography,” *IEEE Transactions on Nuclear Science*, vol. 26, no. 1, pp. 1635–1640, Feb 1979.
- [103] K. M. Hanson, J. N. Bradbury, T. M. Cannon, R. L. Hutson, D. B. Laubacher, R. J. Macek, M. A. Paciotti, and C. A. Taylor, “Computed tomography using proton energy loss,” *Phys. Med. Bio.*, vol. 26, no. 6, pp. 965–983, 1981. [Online]. Available: <http://stacks.iop.org/0031-9155/26/i=6/a=001>
- [104] G. Dedes, L. D. Angelis, S. Rit, D. Hansen, C. Belka, V. A. Bashkirov, R. P. Johnson, G. Coutrakon, K. E. Schubert, R. W. Schulte, K. Parodi, and G. Landry, “Application of fluence field modulation to proton computed tomography for proton therapy imaging,” *Phys. Med. Bio.*, vol. 62, no. 15, pp. 6026–6043, 2017. [Online]. Available: <http://stacks.iop.org/0031-9155/62/i=15/a=6026>

- [105] G. Dedes, R. P. Johnson, M. Pankuch, N. Detrich, W. M. A. Pols, S. Rit, R. W. Schulte, K. Parodi, and G. Landry, “Experimental fluence modulated proton computed tomography by pencil beam scanning,” *Med. Phys.*, vol. 45, pp. 3287–3296, May 2018.
- [106] S. Agostinelli, J. Allison, K. Amako *et al.*, “Geant4 - a simulation toolkit,” *Nucl. Instrum. Methods Phys. Res. A*, vol. 506, no. 3, pp. 250–303, 2003. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0168900203013688>
- [107] H. Jiang and H. Paganetti, “Adaptation of geant4 to Monte Carlo dose calculations based on CT data,” *Med. Phys.*, vol. 31, no. 10, pp. 2811–2818, 2004. [Online]. Available: <https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.1796952>
- [108] V. Giacometti, V. A. Bashkirov, P. Piersimoni, S. Guatelli, T. E. Plautz, H. F.-W. Sadrozinski, R. P. Johnson, A. Zatserklyaniy, T. Tessonier, K. Parodi, A. B. Rosenfeld, and R. W. Schulte, “Software platform for simulation of a prototype proton CT scanner,” *Med. Phys.*, vol. 44, no. 3, pp. 1002–1016, 2017. [Online]. Available: <http://dx.doi.org/10.1002/mp.12107>
- [109] A. Mustafa and D. Jackson, “The relation between x-ray CT numbers and charged particle stopping powers and its significance for radiotherapy treatment planning,” *Phys. Med. Bio.*, vol. 28, no. 2, pp. 169–176, Feb 1983.
- [110] N. Matsufuji, H. Tomura, Y. Futami, H. Yamashita, A. Higashi, S. Minohara, M. Endo, and T. Kanai, “Relationship between CT number and electron density, scatter angle and nuclear reaction for hadron-therapy treatment planning,” *Phys. Med. Bio.*, vol. 43, pp. 3261–3275, 1998.
- [111] A. Smith, “Vision 20/20: proton therapy,” *Med. Phys.*, vol. 36, pp. 556–568, 2009.
- [112] H. Paganetti, “Range uncertainties in proton therapy and the role of Monte Carlo simulations,” *Phys. Med. Bio.*, vol. 57, no. 11, pp. R99–R117, May 2012.
- [113] C. Civinini, D. Bonanno, M. Brianzi, M. Carpinelli, G. Cirrone, G. Cuttone, D. L. Presti, G. Maccioni, S. Pallotta, N. Randazzo, M. Scaringella, F. Romano, V. Sipala, C. Talamonti, E. Vanzi, and M. Bruzzi, “Proton computed tomography: iterative image reconstruction and dose evaluation,” *Journal of Instrumentation*, vol. 12, no. 01, p. C01034, 2017. [Online]. Available: <http://stacks.iop.org/1748-0221/12/i=01/a=C01034>

- [114] R. I. MacKay, “Image guidance for proton therapy,” *Clin. Oncol. (R. Coll. Radiol.)*, vol. 30, no. 5, pp. 293–298, 2018. [Online]. Available: <http://iopscience.iop.org/article/10.1088/1361-6633/aa8b1d/meta>
- [115] U. Schneider and E. Pedroni, “Proton radiography as a tool for quality control in proton therapy,” *Med. Phys.*, vol. 22, no. 4, pp. 353–363, Apr 1995. [Online]. Available: <https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.597470>
- [116] A. C. Society, “Cancer facts and figures 2013,” American Cancer Society Atlanta, Tech. Rep., 2013.
- [117] A. S. for Therapeutic Radiology (ASTRO), “Fast facts about radiation oncology,” <https://www.astro.org/News-and-Media/Media-Resources/FAQs/Fast-Facts-About-Radiation-Therapy/Index.aspx>, Nov 2012.
- [118] C. T. Rueden, J. Schindelin, M. C. Hiner, B. E. DeZonia, A. E. Walter, E. T. Arena, and K. W. Eliceiri, “Imagej2: Imagej for the next generation of scientific image data,” *BMC Bioinformatics*, vol. 18, no. 1, p. 529, Nov 2017. [Online]. Available: <https://doi.org/10.1186/s12859-017-1934-z>
- [119] K. M. Crowe, T. F. Budinger, J. L. Cahoon, V. P. Elischer, R. H. Huesman, and L. L. Kanstein, “Axial scanning with 900 mev alpha particles,” *IEEE Transactions on Nuclear Science*, vol. 22, no. 3, pp. 1752–1754, Jun 1975.
- [120] G. Poludniowski, N. M. Allinson, and P. M. Evans, “Proton radiography and tomography with application to proton therapy,” *The British Journal of Radiology*, vol. 88, p. 20150134, Jun 2015. [Online]. Available: <https://doi.org/10.1259/bjr.20150134>
- [121] K. M. Hanson, J. N. Bradbury, R. A. Koeppe, R. J. Macek, D. R. Machen, R. Morgado, M. A. Paciotti, S. A. Sandford, and V. W. Steward, “Proton computed tomography of human specimens,” *Phys. Med. Bio.*, vol. 27, no. 1, pp. 25–36, Jan 1982. [Online]. Available: <https://doi.org/10.1088/0031-9155/27/1/003>