Acceleration of the noise suppression component of the DUCHAMP source-finder.

Scott James Badenhorst

Dissertation presented in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in the Department of Computer Science

UNIVERSITY OF CAPE TOWN

December 2014

Supervised by

M. Kuttel

S. Blyth

Abstract

The next-generation of radio interferometer arrays - the proposed Square Kilometre Array (SKA) and its precursor instruments, The Karoo Array Telescope (MeerKAT) and Australian Square Kilometre Pathfinder (ASKAP) - will produce radio observation survey data orders of magnitude larger than current sizes. The sheer size of the imaged data produced necessitates fully automated solutions to accurately locate and produce useful scientific data for radio sources which are (for the most part) partially hidden within inherently noisy radio observations (source extraction). Automated extraction solutions exist but are computationally expensive and do not yet scale to the performance required to process large data in practical time-frames.

The DUCHAMP software package is one of the most accurate source extraction packages for general (source shape unknown) source finding. DUCHAMP's accuracy is primarily facilitated by the *à trous* wavelet reconstruction algorithm, a multi-scale smoothing algorithm which suppresses erratic observation noise. This algorithm is the most computationally expensive and memory intensive within DUCHAMP and consequently improvements to it greatly improve overall DUCHAMP performance. We present a high performance, multithreaded implementation of the *à trous* algorithm with a focus on 'desktop' computing hardware to enable standard researchers to do their own accelerated searches. Our solution consists of three main areas of improvement: single-core optimisation, multi-core parallelism and the efficient out-of-core computation of large data sets with memory management libraries. Efficient out-of-core computation (data partially stored on disk when primary memory resources are exceeded) of the *à trous* algorithm accounts for 'desktop' computing's limited fast memory resources by mitigating the performance bottleneck associated with frequent secondary storage access. Although this work focuses on 'desktop' hardware, the majority of the improvements developed are general enough to be used within other high performance computing models.

Single-core optimisations improved algorithm accuracy by reducing rounding error and achieved a 4× serial performance increase which scales with the filter size used during reconstruction. Multithreading on a quad-core CPU further increased performance of the filtering operations within reconstruction to 22× (performance scaling approximately linear with increased CPU cores) and achieved 13× performance increase overall. All evaluated out-of-core memory management libraries performed poorly with parallelism. Single-threaded memory management partially mitigated the slow disk access bottleneck and achieved a 3.6× increase (uniform for all tested large data sets) for filtering operations and a 1.5× increase overall. Faster secondary storage solutions such as Solid State Drives or RAID arrays are required to process large survey data on 'desktop' hardware in practical time-frames.

Acknowledgements

This thesis is in dedication to my family and friends for their unwavering support throughout the years and to my supervisors for their infinite patience.

Plagiarism Declaration

I know the meaning of plagiarism and declare that all of the work in the document, save for that which is properly acknowledged, is my own.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To understand galaxy formation and evolution over time, all components that make up these processes need to be understood, i.e. the role of the interstellar medium (mainly hydrogen gas and dust), star formation and evolution, and environmental factors (if the galaxy exists in a low or high density environment) [89]. Radio astronomy allows for the investigation of these processes through the observation of their neutral hydrogen (HI) content, a large component of typical galaxies, which emits a distinct 21-cm wavelength (1,420 MHz) radio emission. However, this signal is produced by a low rate quantum spin flip transition of hydrogen's single electron (changing the relative spin orientation of the proton and electron) and requires large HI volumes to produce a weak and constant signal [53]. To accurately estimate HI content in galaxies, larger surveys that probe to cosmologically significant distances with highly sensitive observational instruments are required.

The traditional observation method of using single dish telescopes has reached feasibility limits in terms of size and maintenance. Both steerable and stationary single dish telescopes would be required to be infeasibly large to further improve their angular resolutions to enable the detection of small HI sources at cosmologically significant distances. These size limitations are by-passed with radio interferometry techniques which simulate a large single dish telescope of aperture size D by combining the signals (generating observations from their interference patterns) from two smaller single dish telescopes separated by a distance D. A interferometer array consists of many smaller dishes, forming many paired dish combinations and allowing the imaging of objects at different scales. Additionally, the large collective area of an array (sum total of all dish area) makes it sensitive to faint signals. The sensitivity and angular resolution of the next-generation of radio interferometry arrays such as Square Kilometre Array (SKA)[1] and its precursor instruments, The Karoo Array Telescope (MeerKAT)[2] and Australian Square Kilometre Pathfinder (ASKAP)[3], will allow for detailed ultra-wide and ultra-deep HI surveys orders of magnitude larger than current survey sizes.

The current largest HI surveys, namely the HI Parkes all sky survey (HIPASS) [8, 29, 68, 109] and the Arecibo Legacy Fast ALFA survey (ALFALFA) [39, 84], have produced image data sets in the range 10 GB - 100 GB. Whilst in the past, the detection of sources in these noisy observations was performed manually (extracted by eye), the sheer number of sources in current surveys necessitates automated solutions to source extraction [102]. These automated HI source extraction solutions are required to accurately locate and measure the properties of (parametrise) relatively weak radio sources of unknown shape and size from survey data largely

[1] https://www.skatelescope.org/
[2] http://www.ska.ac.za/meerkat/
[3] http://www.atnf.csiro.au/projects/askap/index.html

dominated by radio noise (originates from many sources including our own galaxy and man-made signals). Thus the biggest requirement of source-finders is to maximise the number of sources detected (and parametrised) and detect at low signal-to-noise ratios whilst minimising false detections. Maximising for both these metrics is a complex task. Consequently, many existing software extraction solutions are computationally expensive and take days to process the data from current surveys.

The next-generation of HI surveys conducted on the SKA precursor instruments such as The Widefield ASKAP L-band Legacy All-sky Blind Survey (WALLABY) [32] and the Deep Investigation of Neutral Gas Origins (DINGO) [32] surveys with the ASKAP array and the Looking At the Distant Universe with the MeerKAT Array (LADUMA) [46, 47] survey will produce survey data volumes 10^2 - 10^4 times larger than current surveys [32, 52]. Processing next-generation survey data is expected to take infeasibly large time-frames with current searching (source extraction) solutions. High performance computing in conjunction with fully automated source extraction is required to enable next-generation source finding in practical time-frames.

1.1 Automated HI source extraction and the role of high performance computing

Several automated source extraction packages have been developed to process noisy data sets. The most notable are the Gamma-finder [13], DUCHAMP [104], 2D-1D wavelet reconstruction [33] and the CNHI [52] source-finders, which were evaluated for potential inclusion within the software pipeline for the ASKAP telescopes large surveys [81]. These source-finders detect sources in two main ways: intensity thresholding (where observations above a flux threshold are considered valid) and statistical detections (where the statistical likelihood of potential source regions are assessed).

The efficacy of these finders is determined by completeness and reliability metrics. Completeness measures the percentage of sources detected (true source count known) whilst reliability measures how many detected sources were true sources, as noise spikes in the observation may cause spurious detections. In the Popping et al. evaluation [81] of several leading source-finders, the DUCHAMP source extraction package was found to be a robust source-finder best suited for the extraction of small point-like sources. Additional evaluation of the DUCHAMP package in Westmeier et al. [102] showed this source-finder to be adept at finding sources at low signal-to-noise ratios. An improved version of DUCHAMP (Selavy) is set to be used to process the ASKAP surveys [50]. The success of DUCHAMP lies in its preprocessing steps, specifically the *à trous* wavelet reconstruction algorithm, a multiscale (accounts for objects of varying size) noise suppression algorithm which greatly increases the completeness and reliability of the source-finder as a whole [104, 102, 81]. However, this algorithm, as in many similar source extraction processes, is computationally intensive and takes up the majority of total DUCHAMP run-time. Although current survey sizes can be processed in a few days with automated solutions, the 2-4 orders of magnitude increase in expected survey size on the next-generation of radio telescopes will necessitate high performance computing to complete computation in practically short time-frames.

Fortunately, the majority of the algorithms used in automated source extraction are well-suited for redevelopment as high performance parallel implementations. This is seen in filtering operations, such as those found in DUCHAMP's *à trous* wavelet reconstruction [103], which contain computing tasks that are performed independently for each volumetric pixel (voxel) in a data set. This independence of operation has the potential for embarrassing amounts

of parallelism and is ideal for both the cluster and distributed high performance computing solutions. Despite this suitability, few source extraction packages have parallel implementations and are largely unoptimised for even their existing sequential implementations. Many sequential procedures within *à trous* reconstruction have the potential for large performance increases with algorithm optimisation alone.

In recent years, there has been a significant rise in commodity CPU development and increases to underlying bus speeds. Modern CPUs now contain multiple processing cores (multi-core) and hyper-threading (logical computing cores) [65, 4] which allows for efficient parallel processing. Additionally, the throughput of a single core can be increased through vector instruction SIMD (Single Data Multiple Data) parallelism such as Intel's Streaming SIMD Extensions (SSE) commands [38, 43], which allow for the concurrent execution of a single operation on multiple data. This has allowed 'desktop' commodity hardware to be suitable for high-performance parallel computing.

'Desktop' computing is still attractive with respect to source detection as it allows the individual astronomer to perform their own searches through survey data. Additionally, the choice of search parameters (optimising searches to better detect sources of a certain criteria) and the source extraction package used is open to the individual. This is in contrast to the ASKAP blind surveys where output will be produced by Selavy alone and will likely keep search parameters general to maximise completeness and reliability for blind surveys (source size and shape unknown). Whilst individual searches could potentially be performed with remote access to large clusters or distributed systems, there will exist circumstances where access to these devices is limited and necessitates high performance source detection on individual 'desktop' workstations.

However, the processing throughput of 'desktop' commodity hardware will be insufficient to process next-generation HI survey data in practical time-frames. Additionally, the amount of allocated memory required to process this survey data is expected to exceed 'desktop' main memory (fast-access memory) capacity. Excess data will be required to be temporarily stored on disk (out-of-core computation) where the slow access (orders of magnitude slower than main memory access) to and from disk can bottleneck performance. The amount of allocated memory and consequently out-of-core memory transfer can be reduced by individually processing data set 'chunks' (segments or blocks) to completion. Algorithms such as *à trous* reconstruction are ill-suited to segmentation as strong dependencies exist between algorithm components and would increase the disk access required [103]. General fast out-of-core computation can be enabled by optimising the paging process (data transfer to and from the disk) with external memory management libraries. Additionally, the performance of a subset of these data management APIs can be improved when the memory access pattern is known. Popular memory management libraries include the Mmap [63][55], Boost [37] and Stxxl [25] libraries.

Algorithm optimisation in conjunction with parallel computing and memory management may be sufficient to allow 'desktop' commodity hardware to compute source extraction procedures on the next-generation of large HI surveys in practical time-frames.

1.2 Research Questions

This thesis aims to answer the following research questions:

- Can we improve the efficiency of the *à trous* wavelet reconstruction for a single core CPU?

- Can Intel CPU SSE commands facilitate SIMD execution in this algorithm and further increase performance for the single-threaded case?
- Can we accelerate these processes by utilising parallel 'desktop' multi-core CPU hardware?
- Can slow disk access on 'desktop' hardware be mitigated with memory management to allow for efficient computation of large data sets?

1.3 Aims

We aim to develop a high performance implementation of the DUCHAMP *à trous* wavelet reconstruction algorithm for high performance 'desktop' commodity hardware. This algorithm takes up 65-95% (Chapter 3) of total DUCHAMP run-time and increasing performance will significantly aid in the future development of a high performance, parallel DUCHAMP implementation. Additionally, the high computational intensity and memory use of this algorithm will allow for the assessment of 'desktop' hardware as a viable high performance computing solution for processing large HI survey data in practical time-frames.

The developed system will be assessed with respect to each of the defined research questions in isolation as well as in combination to prevent unbiased assessment of the various developed system components. The high performance components developed in our system are intended to be general enough to be easily accommodated into larger parallel computing hardware solutions despite not being the main focus of development. The memory management solution will only be intended for incorporation into systems with insufficient fast-access memory and slow secondary storage.

1.4 Approach

A high performance commodity hardware implementation of the entirety of the DUCHAMP source extraction package was not considered due to the sheer scope of this package which includes several preprocessing operations and source detection algorithms. To reduce the scope of this work we focused on improving the *à trous* wavelet reconstruction algorithm which is the most computationally intensive procedure (65-95% run-time) with the highest memory use (utilising 5 times the observational data size in allocated memory) within DUCHAMP.

Development of the system followed an iterative approach consisting of four distinct stages: a sequential C++ implementation, single core optimisation which consists of both algorithmic optimisation and the use of Intel's SSE commands, multi-core parallelism and memory management to optimise paging from disk. This iterative development allowed for strict assessment of the validity and performance contribution introduced at each stage.

Algorithm optimisation consists of reducing the amount of computation required by the filtering procedures within the *à trous* reconstruction as they comprise the majority of algorithm run-time. Intel's SSE instructions are used to further optimise computation for the single threaded computation with implementing vector instruction parallelism on a single CPU [34].

We implement multi-core CPU parallelism solutions to all components within the *à trous* wavelet reconstruction algorithm which are embarrassingly parallel. Task parallelism (the concurrent execution of algorithm components) is not considered as it is significantly hindered by strict dependencies with the algorithm.

Three popular memory management libraries, namely Mmap [63][55], Boost [37] and Stxxl [25], are assessed to determine whether these APIs are sufficient to mitigate the disk access bottleneck on 'desktop' hardware to allow efficient out-of-core computation.

Validity will be ensured by comparing our system outputs with those produced by DUCHAMP to double precision. Error that cannot be removed will render our system invalid as a scientific tool. Evaluation of performance increases will consist of the relative performance increase introduced at each stage and the absolute performance relative to DUCHAMP as the performance contributions of each stage are cumulative.

A high performance 'desktop' commodity hardware solution for the *à trous* wavelet reconstruction algorithm will be considered successfully reached when performance is increased by an order of magnitude over the original sequential DUCHAMP implementation whilst maintaining accuracy of results. Additionally, this performance should be maintained for large data set sizes which necessitate out-of-core computation.

1.5 Outline

This thesis is structured as follows:

Chapter 2 provides an introduction to neutral hydrogen research and radio astronomy observations. This is followed by current and future HI surveys, and the high performance automated source-finders required to process them. A brief description of the resources available on 'desktop' hardware is given. Filter theory relevant to this thesis is discussed.

Chapter 3 discusses the specific processes and functionality of the DUCHAMP source extraction package. The *à trous* wavelet reconstruction algorithm is described in detail, covering memory use, computation requirements and memory access patterns.

Chapter 4 covers the design specifics and limitations of the implemented system. The design methodologies and approach to parallel development and algorithm optimisation are covered. Constraints to evaluation and validation procedures are outlined.

Chapter 5 discusses the implementation and testing of the prototype system. This includes algorithms implemented, development difficulties, the data structures used, the input parameters to the test system and implementation considerations which maximised performance.

Chapter 6 reports the findings of this thesis. Results are divided into moderate sized data sets for the findings of performance improvement and extremely large data sets for the evaluation of out-of-core computation solutions. Special cases are discussed.

Chapter 7 concludes this work, covering the overall findings and limitations of this research and suggesting possible future improvements.

Chapter 2

Background

2.1 Introduction

In this chapter, we discuss the theory and motivation behind the automated detection of neutral hydrogen (HI) sources in radio astronomy observations. This background theory includes the specifics of the HI emissions, the instrumentation used and the difficulties that arise in its detection. Past and future HI surveys are discussed to highlight the extent of neutral hydrogen research and what the next generation of radio telescopes has in store for continued research in this field. A review of current automated source detection techniques and software packages is provided, with a detailed treatment of the DUCHAMP software package covered in Chapter 3. We cover the background on computing concepts and 'desktop' commodity hardware architectures used in the remainder of this thesis to accelerate automated radio source detection within the large surveys planned for the next generation of radio astronomy telescopes.

2.2 Neutral hydrogen (HI) in galaxies

To understand galaxy formation and evolution over time, all components that make up these processes need to be understood, i.e. the role of the interstellar medium (mainly hydrogen gas and dust), star formation and evolution, and environmental factors (if the galaxy exists in a low or high density environment, e.g. voids or clusters). Although a lot is known about stars through optical observations, less is known about the hydrogen content and its distribution, one of the key components of a typical galaxy [53]. Additionally, optical surveys may underrepresent low optical brightness objects such as dwarf galaxies [32]. This too may alter the constraints on predictive models and our understanding of the cosmos.

Hydrogen is found in three forms, namely atomic (HI), molecular and ionised, depending on temperature, density or the ambient radiation field [72]. By studying hydrogen we can investigate galaxy formation and the mechanics of galaxies such as star formation, as hydrogen provides the fuel to form stars [89]. Most of the molecular hydrogen (H_2) in galaxies is cold (below 20 K) [72]. At these temperatures H_2 is at ground state and cannot produce an electromagnetic (EM) emission spectrum. Rotational transitions are still relevant at these temperatures but for H_2, which is a symmetric molecule, these transitions are suppressed and do not produce emission (essentially invisible) [72]. The location of H_2 clouds can be inferred by the existence of other molecules such as carbon monoxide which are observed at radio and microwave wavelengths [72]. Ionised hydrogen exists in high temperature areas such as around hot, young stars [72].

In contrast, investigation of neutral atomic hydrogen (HI) is made possible through observation of the distinct HI 21-cm ((1,420 MHz) radio emission [106] which falls within the portion of the EM spectrum detectable by radio telescopes on Earth. This HI 21-cm emission is caused by a low rate ($2.868\text{x}10^{-15}\text{s}^{-1}$) quantum transition where the single orbiting electron of a particular hydrogen atom spin flips, changing its spin orientation with respect to the proton. This transition is mainly produced due to collisions of the gas atoms with the interstellar medium, producing, a constant but relatively weak signal [53]. Therefore HI is a good marker for detecting gas clouds [14] making it useful in the study of early galaxy evolution.

Whilst information about the spatial extent of a galaxy can be determined by observing the HI emission alone, additional information can be obtained by studying the variation of the received flux (emission) with wavelength or frequency - called a spectrum. Doppler effects caused by the relative motion of emitting bodies alter the observed wavelength (inversely proportional to frequency) by the following relation (for non-relativistic velocities):

$$\lambda_{observed} = \lambda_{emitted}(1 + \frac{v}{c}) \tag{2.1}$$

where

$c \gg v$

$v > 0$ denotes a receding body

$\frac{v}{c} = z$ which is known as redshift

This causes the emission wavelength to appear to be shifted towards the red side of the electromagnetic spectrum ($z > 0$, red-shifting) for objects moving away from the observer, and towards the blue side of the spectrum ($z < 0$, blue-shifting) for approaching objects.

The offset of a signal profile from the HI 21-cm line allows the measurement of the red-shift due to the relative motion of emitting galaxies from Universe expansion and the motion of a galaxy in a gravitational potential (e.g. a cluster). The cosmological red-shift (the part due to Universe expansion) is critical in understanding the evolution of galaxies over cosmic time. Additionally, the rotating HI content of galaxies gives rise to a blue/red-shifting smearing of the 21-cm line which results in a distinct double-horned signal (Fig 2.1) for approximately edge-on galaxy observations. This profile degrades to a Gaussian-like shape for galaxies which appear more face-on.

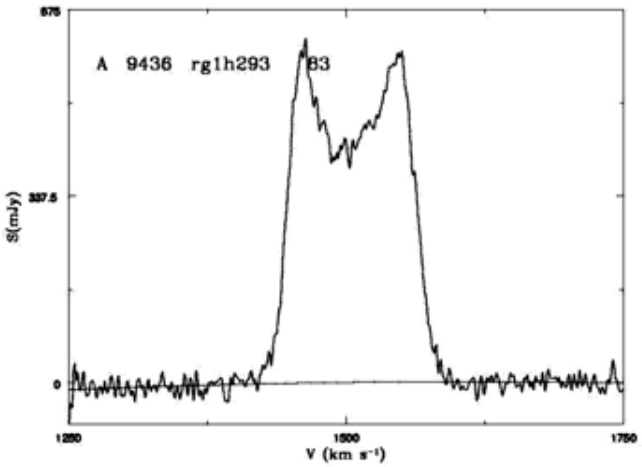


Figure 2.1: A typical HI double-horn profile of a spiral galaxy. Image Source: ALFALFA survey website[2].

In order to study the evolution of the HI content of galaxies, astronomers would like to measure observables such as the HI mass function (HIMF) and its variation with redshift, the cosmic neutral gas density (Ω_{HI}) variation with redshift and the evolution of HI properties in galaxies as a function of both redshift and environment.

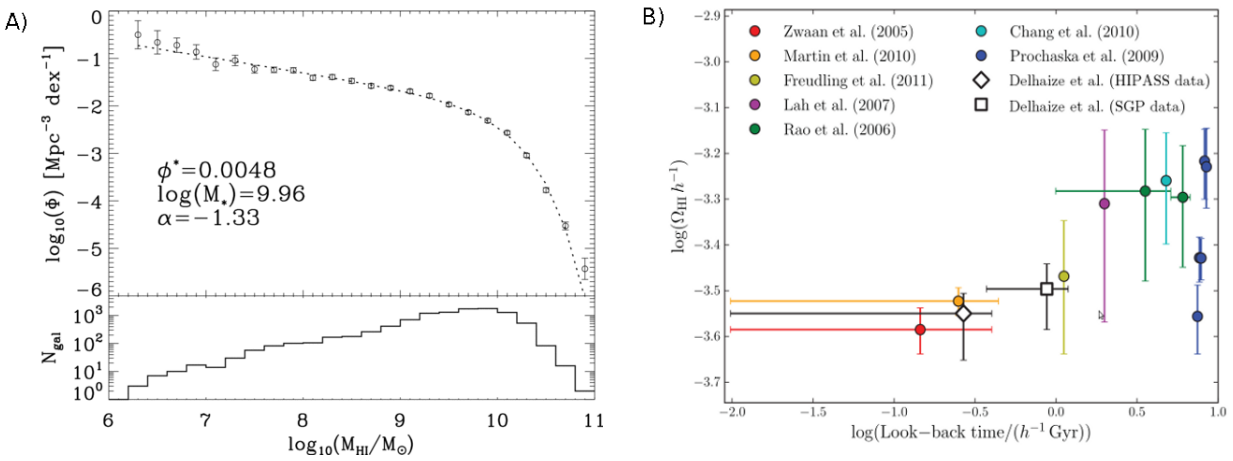


Figure 2.2: Examples of a derivation of the (A) global HI mass function using ALFALFA α.40 HI data [45] (Image sourced from [66]) and (B) cosmic neutral gas density (Ω_{HI}) constraints (Images sourced from [23]).

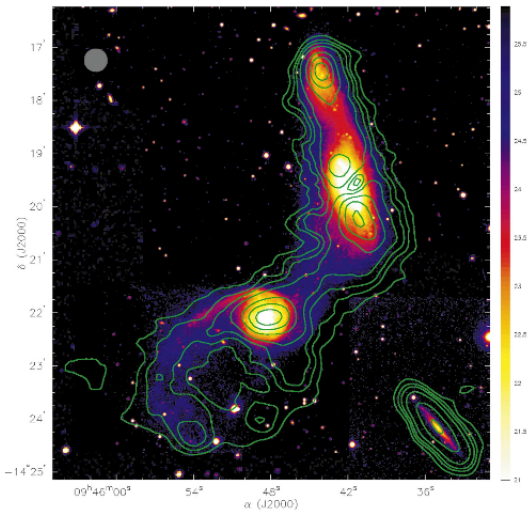


Figure 2.3: Example: Tidal HI interactions in Arp 245, overlayed as contours over an optical image (V-passband/visible spectrum). Image sourced from [30].

The HI mass function per volume (Fig 2.2 A) quantifies the relative number of galaxies within specific ranges of HI mass [47, 14]. Measuring the variation of slope, breakpoint and normalisation of this function with redshift will provide parameters required to test and constrain models of galaxy formation. Currently the HIMF has only been measured for the local universe and we do not yet know how it has evolved with cosmic time.

The cosmic neutral gas density (Fig 2.2 B) measures the total neutral gas available at different time periods. Investigating the amount of neutral gas available in galaxies as a function of cosmic time will help to shed light on the various processes driving galaxy evolution [53].

Finally, we can investigate if and when galaxy formation has been altered owing to environment conditions (such as existing in a region of low or high galaxy density). This includes mapping tidal interactions between colliding galaxies as shown in Fig 2.3.

To facilitate accurate investigation of the formation and evolution of galaxies, detailed surveys of HI to very high redshifts are required [110]. However, the majority of current studies (Chapter 2.5) are comprised of relatively local galaxies [53] as they are limited by the sensitivity and resolution of current radio telescopes. In the following sections we discuss current instrumentation and its limitations before covering the engineering advancements used in the design of the next generation of radio telescopes. These future devices are expected to be sensitive enough not only to allow ultra-deep (to high redshifts) and ultra-wide observations of HI emission, but also to allow for the probing of the epoch of reionisation [14], which marks the era when the first stars formed and began "heating" the Universe.

2.3 Radio astronomy instruments

Radio emission detection of astronomical sources is performed using various forms of extremely sensitive directional radio antennas (called radio telescopes) [32]. Two main types of antenna are utilised, namely single-dish telescopes and radio interferometer arrays. In this section we discuss the specifics of, and major differences between these devices and motivate the current large scale adoption of interferometer arrays.

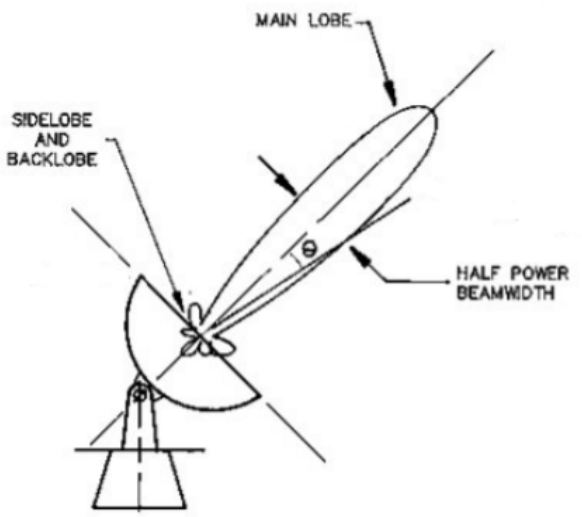


Figure 2.4: Single dish telescope overlayed with the main and side lobes of the receiver pattern. Angular resolution is equal to angle θ. Image Source: [70].

Traditionally, radio astronomy is performed using large single dish telescopes which consist of large parabolic surfaces to focus emission at a receiver (Fig 2.4) to directly detect flux values (strength of the detected signal) at specific positions (pointings) on the sky. Due to the shape and specifics of an antenna, a directional dependant pattern of reception strength is formed, known as the receiving or power pattern [106]. Radio telescopes concentrate this power pattern into the so-called main lobe or primary beam to define a small directional extent at which signal can be strongly received in order to discern the direction of the emitting body [106]. The angular extent of this beam defines the (angular) resolution of a particular radio telescope, defined as the minimum angular extent between two emitting objects to discern them as separate entities. To discern objects at high redshifts that have small angular extents, the angular resolution of the primary beam is required to be extremely small [32].

Angular resolution (θ) of a dish is proportional to wavelength and inversely proportional to aperture size.

$$\text{Angular Resolution } \theta = \frac{1.22\lambda}{D} \tag{2.2}$$

D - diameter of the radio dish.

λ - wavelength of observed emission.

Thus to attain high angular resolutions for radio waves, which are 4 to 7 orders of magnitude larger than visible light [35], the collection dish is required to have an extremely large dish diameter.

Single dish telescopes are normally steerable to increase the portion of the sky which is observable with the device. Notable large steerable radio telescopes include the 76 m Lovell telescope operated the Jodrell Bank observatory[1] in northern England, the 100 m Effelsberg telescope[2] operated by the Max Planck Institute in Germany, the CSIRO[3] 64 m Parkes telescope[4] in Australia and the 100 m Green Bank telescope[5] in the USA, the world's largest steerable radio telescope. However, steerable dish size and consequently angular resolution has reached a cost and maintenance feasibility limit ($\sim$ 100 m [35]) as a result of the sheer weight of these devices [16].

This size limit can be overcome by using a fixed dish set-up [39], such as the 305m Arecibo telescope[6] in Puerto Rico, which has a stationary collecting area and a movable receiver to allow for the observation of a sky area greater than a single pointing. This allows for telescopes to have a finer angular resolution and higher sensitivity at the cost of a greatly reduced observable sky area. However, to observe HI signals at cosmologically significant distances, an even finer angular resolution is necessitated which would require an infeasibly large dish size even for fixed dish solutions.

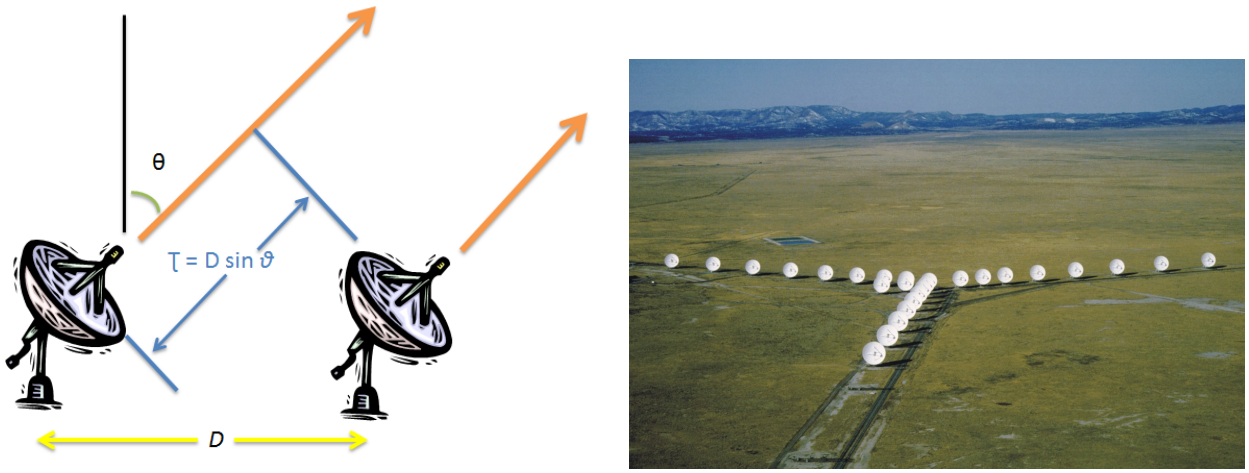


Figure 2.5: A) A single radio interferometry baseline. τ is the extra propagation distance and consequently time taken for signal to reach the more distant antenna. B) The Karl G. Jansky Very Large Array (VLA) in North America (http://www.vla.nrao.edu/).

[1] www.jb.man.ac.uk/
[2] www.mpifr-bonn.mpg.de/8964/effelsberg
[3] The Commonwealth Scientific and Industrial Research Organisation
[4] www.parkes.atnf.csiro.au/
[5] https://science.nrao.edu/facilities/gbt/
[6] www.naic.edu/index_scientific.php

The angular resolution limitations of the single dish approach can be improved upon with radio interferometry. Radio interferometry is a technique of superimposing electromagnetic waves from two or more observing devices in order to produce interference patterns which provide information about the incoming radio emission waves [21]. However, the timing within these devices must be precise as they are required to compensate for the difference in propagation times for emission to arrive at each antenna (Fig 2.5 A). Signal received by the antenna nearest to the source is delayed by the difference in propagation time to the furthest antenna, namely $t = c.(D.sin\theta)$, where D equals the distance between the antenna and θ equals the angle between the antenna pointing and a line perpendicular to the Earth. The clear advantage over single dishes is that the angular resolution of a radio interferometer array is equal to that of a theoretical single dish (Equation 2.2) with an aperture diameter equal to the longest distance (D) between the interferometer elements (known as a baseline). This allows the simulation of a large single dish telescope with relatively small single dish elements by placing elements further apart. Additionally, the field-of-view achieved with radio interferometers is much larger than its single dish counterparts as the field of view is proportional to the wavelength observed and inversely proportional to individual dish diameters which are typically relatively small. Field-of-view specifically refers to the area observed with a single pointing of the telescope.

$$\text{Field of View (primary beam)} = \frac{1.22\lambda}{d}$$

where d is equal to the diameter of the interferometer dishes.

Thus interferometer arrays can reach both superior angular resolution and field-of-view with a relatively small dish size which reduces the cost of construction and maintenance. Notable large interferometers (completed) include the Janksy Very Large Array[7] (JVLA) (Fig 2.5) in the USA with 27 $\times$ 25 m dishes (max. baseline 36 km), the Giant Metrewave Radio Telescope[8] (GMRT) in India with 30 $\times$ 45 m wires dishes (max. baseline 25 km) and the Australia Telescope Compact Array[9] (ATCA) with 6 $\times$ 25 m dishes (max. baseline 6 km).

However, there are some disadvantages and difficulties that arise with the radio interferometer approach. Unlike single dish telescopes that measure the flux of the sky directly at a specific pointing, radio interferometers record interference patterns by correlating the signals from each telescope pair in a correlator to produce the so-called UV-plane observation, a Fourier transform of the sky brightness distribution [105, 99]. The accuracy of the final image is proportional to the number of UV-plane samples (baselines) that are recorded which increases by $(\frac{(N-1)N}{2})$ with the number of antennas (N) [35]. It is not necessary to cover the entire array area with telescopes to improve UV coverage; instead the techniques of aperture synthesis (through Earth's rotation) and multi-frequency synthesis (only available to radio continuum observations) are necessitated to fill the UV-plane. However, the processing required to generate both the UV-plane and the final image (inverse Fourier transform of the UV-plane) are extremely computationally expensive.

Current instruments are limited to approximately $z \sim 0.2$ (back in cosmic time $\sim$ 2 billion years) for 21cm line observations in reasonable observing time [1]. To study the role and evolution of HI in galactic processes over significant ranges of cosmic time, higher sensitivities and smaller angular resolutions are required. This has motivated for the next-generation of radio interferometric arrays: the Australian Square Kilometre Array Pathfinder (ASKAP)[10] consisting of 36 $\times$ 12-m dishes [22, 32], the Karoo Array Telescope (MeerKAT)[11] consisting of

[7] www.vla.nrao.edu/

[8] gmrt.ncra.tifr.res.in/gmrt_hpage/GMRT/intro_gmrt.html

[9] www.narrabri.atnf.csiro.au/

[10] http://www.atnf.csiro.au/projects/askap/index.html

[11] http://www.ska.ac.za/meerkat/

64 × 13.5-m dishes [20], and the Square Kilometre Array (SKA)[12] which will consist of 3600 × 15-m dishes and low frequency antennas split across both the ASKAP and MeerKAT locations. Both the ASKAP situated in Western Australia and MeerKAT situated in the Northern Cape, South Africa are designed to be precursors to the SKA.

The extreme size of the SKA array will result in an unprecedented amount of data with 10^{18} bits per second [21] pre-correlator with exaflops of computations and petabits/s of I/O. The angular resolution possible with the SKA will be very high due to very long expected baselines as SKA dishes will extend up through Africa. Additionally, SKA's expected sensitivity will allow the observation of HI emission to very high redshifts ($1.5 < z < 2$) [16].

Radio interferometry techniques alone do not solve all the problems associated with radio emission observations. In the next section, we discuss some of the problems and difficulties that arise when observing radio emissions and how many of those problems are exacerbated when observing the relatively weak HI emission.

2.4 Detection of neutral hydrogen

Observation of HI bodies can be difficult as signal strength is relatively weak and can be hidden and distorted by the high prevalence of radio noise in an observation. In this section we discuss the primary types (and mitigation schemes) of noise and signal distortion affecting HI detection: atmospheric effects, interstellar medium, signal smearing and terrestrial noise.

Atmospheric effects describe signal distortion and obfuscation as a result of the properties of the Earth's upper atmosphere which cause refraction, scattering and absorption of radio waves. The radio window refers to the range of radio frequencies for which the Earth's atmosphere is transparent. This range approximately extends from a wavelength size of 20 m to 0.2 mm where signals with wavelength sizes larger and smaller than this range are scattered in the ionosphere and absorbed by water vapour and oxygen in the troposphere respectively [106]. Although absorption can be lessened by selecting arid locations for instruments, the effects of oxygen absorption and the ionosphere scattering cannot be mitigated from the Earth's surface and require observation from satellites.

The refraction effects of the atmosphere refer to signal distortion caused by variations in the atmosphere's refractive index [88] and atmospheric turbulence in both the ionosphere [88] and troposphere [61]. In contrast to scattering and absorption, refraction effects can be mitigated by using one or more calibration methods which attempt to estimate position, phase (interferometer specific) and gain distortions in an observation. These calibration methods include the observation of known bright sources in the field-of-view [95], self-calibration (or auto-calibration methods) which refers to the more general case where some source parameters are unknown [75][95], and information from orbiting satellites [88]. Additionally, optical survey data may be used to cross reference positions when emitting bodies have both an optical and radio component.

The detection of relatively weak radio emissions (common for higher redshift objects) is significantly hindered and limited by human-made radio frequency interference (RFI). These signals primarily originate from broadcasting of television and communication signals but radio telescopes are sufficiently sensitive to be affected by thermal noise generated by electrical devices such as spark plugs, electric fences and orbiting satellites [7]. This RFI may even originate from the observation site itself as the electronics in computing and signal transfer are capable

[12] https://www.skatelescope.org/

of radio emission which necessitates the RFI shielding of these components [7]. This motivates the placement of instruments in radio quiet areas [7] and monitoring of RFI so as to choose quiet times or observe in radio quiet frequency bands [36] as well as the implementation of radio astronomy protected bands which cannot be utilised for other purposes [57].

Three common RFI mitigation techniques are rejection in the temporal domain, frequency rejection (or flagging) and spatial filtering [36]. By monitoring signals over time the identification of short-lived RFI signals can be removed from an observation [36]. Frequency flagging refers to the process where noise frequency bands are suppressed. These noise bands are either known RFI signals or detected through the real-time spectrum monitoring of RFI using directional antennas [36, 7]. Spatial filtering uses the principle of differences in the direction-of-arrival of valid and RFI emissions [36]. For single dish antennas this is performed with a separate antenna pointing away from the field of view of the primary antenna to allow for a type of adaptive interference cancellation of unwanted sources [7, 36] using the different signal arrival times at each dish [57], as single dishes alone cannot differentiate signal direction [7]. This technique works best for strong RFI and where spectral information is not useful [7]. In the case of radio interferometers this interference cancellation/minimisation can be performed between multiple radio interferometry elements [36] as extended baselines between elements allow for the identification of localised RFI [7].

Signal smearing results from the blue/red-shifting of a rotating HI gas body (discussed above). For stronger sources this can aid identification through the double-horn/Gaussian profile that is produced. However, signal smearing of weaker sources can impede detection as the signal is spread across multiple frequency detection bins, decreasing the signal to noise in a single channel.

The next generation of radio interferometers will attempt to detect extremely weak signals in both the local and distant universe. These instruments are expected to be over 100 times more sensitive than current devices [85] which will allow the detection of fainter objects at cosmologically significant scales.

2.5 HI Surveys

In recent years, various large surveys have successfully carried out the mapping of HI in galaxies for extended regions of the sky. In this section we provide an overview of some of the large HI surveys which have been completed or are still ongoing and the sheer increase in HI information that will be provided with future surveys on the next generation of telescopes. Additionally, we compare past and future surveys to highlight the large increase in imaged data produced by these devices; i.e. the data sets from which HI sources are extracted. The size of these data sets is defined by the area of the sky surveyed and the resolution of the survey, angular resolution (pixel size) and the size and range of the spectral bins. The exponential increase in imaged data complicates the process of source extraction. Traditionally, source extraction has been performed manually (by eye) and required trained individuals to extract valid sources from noisy data sets. However, current surveys have already reached sizes that necessitate semi-automated processing, with future surveys expected to require fully automated source extraction solutions in conjunction with high performance computing.

2.5.1 Past and ongoing large HI surveys

There have been many HI galaxy surveys. The most notable of these surveys, in terms of source counts, are the HI Parkes all sky survey (HIPASS) [8, 68, 109, 29] and Arecibo Legacy Fast ALFA survey (ALFALFA)[39, 84, 40, 49]. The HIPASS survey represents the largest HI survey to date in terms of sky coverage. ALFALFA is considered larger in terms of the number of sources that were detected in the survey.

The HIPASS survey utilised the CSIRO's Parkes radio telescope (64-m diameter parabolic dish) located in Australia, to scan the entire Southern sky and partially scan the Northern sky up to +25 deg. To reach full sensitivity, the observation area was scanned 5 times to $z \sim 0.042$ using 1024 velocity bins [68] with channel separation 13.2 km.s^{-1} ($z = 0$). The HI Jodrell ALL sky survey (HIJASS) [56] completes the HIPASS survey with plans to completely observe the rest of the Northern sky. Since the Parkes telescope cannot see more Northern declinations due to its location in the Southern Hemisphere, the HIJASS survey was carried out on the 76 m Lovell telescope in the Jodrell Bank observatory in northern England [60].

The HIPASS survey has resulted in thousands of detected sources, namely 4315 detected sources in the region declination $<$ +2 deg [68] and a further 1003 sources for the range +2 $<$ declination $<$ 25 deg [107] detected purely on their HI content. Although originally 387 separate data sets were used to cover the southern sky, the entirety of the surveys was recently combined by Jurek [44] into a single cube of 12 GB (dimensions 1721 $\times$ 1721 $\times$ 1025 voxels). The sheer size of this data set prevented extraction of sources using traditional manual methods and required semi-automated techniques to extract sources of interest.

The Arecibo Legacy Fast ALFA survey (ALFALFA) [39, 84], currently the biggest HI survey in terms of source count and surveyed to a deeper redshift than HIPASS, was conducted with the Arecibo telescope. This survey has achieved 8 times the sensitivity and 4 times the angular resolution of the HIPASS survey. The L-band feed array installed at the Arecibo telescope allowed over 7000 deg^2 of the sky to be observed. Each data cube covers an area of 2.4° by 2.4° with a voxel separation of 1 arcminute and 1024 spectral channels [84]. At double precision these cubes are 162 MB in size. To cover the full range of 7000 deg^2 and redshift range of -2000–17,900km.s^{-1} [84], 4862 of such cubes will be needed and will take up 0.75 TB. To date the largest catalogue release has been the α.40 HI source catalogue with 15,855 sources found in a sky coverage of 2800 deg^2 which equates to 29 times more sources found per square degree than HIPASS [45].

Although the HI content identified through these surveys is extremely significant, more detailed study of the role and evolution of HI in galactic processes over cosmologically significant distances (and time) necessitates both larger surveys and observations to higher redshifts. In the next section we describe the future HI surveys planned for the next-generation of radio telescopes and the estimated number of sources that are expected to be detected. These surveys will be significantly larger than current HI surveys and will necessitate high performance computation to process large data for both the instrument itself and automated source detection, as manual extraction is already infeasible for current survey sizes.

2.5.2 Future HI surveys

Future HI surveys will be conducted on the Square Kilometre Array (SKA) and its precursor instruments, MeerKAT, ASKAP and APERTIF[13]. These surveys will be significantly larger

[13] http://www.astron.nl/general/apertif/apertif

than current HI surveys in terms of the amount of data generated and will necessitate high performance computation for processing and analysis. Automated source detection will be needed since manual extraction is already infeasible for current survey sizes.

The Widefield ASKAP L-band Legacy All-sky Blind Survey (WALLABY) and the Deep Investigation of Neutral Gas Origins (DINGO) are two planned HI surveys for the ASKAP facility [32]. The WALLABY survey is an ultra-wide survey which plans to detect HI to a red-shift of $z \sim 0.26$ for approximately 75% of the sky. This survey is expected to produce 1200× 782 GB data cubes (dimensions $3600 \times 3600 \times 16200$ voxels) which equates to nearly 1 PB of Stokes I imaging data. This imaging data is expected to yield over 500 000 detected galaxies [32]. Additional data products produced by this survey, such as catalogues, beam images, post-stamp images, etc., will increase the total data storage required by WALLABY to well beyond 1PB.

The DINGO survey will be an ultra-deep survey conducted to two different depths and will only cover approximately 60 deg^2 of the sky [32]. "DINGO deep" will probe five fields out to $z \sim 0.26$ with 500 hours of integration per field. A data cube equal in size to the WALLABY (782 GB) will be generated for each of the fields, totalling ~ 4 TB of imaged data. "DINGO ultra-deep" will probe the red-shift range of $0.1 < z < 0.43$ across two fields with 2500 hours of integration per field. Although fewer sources (10^5 galaxies) are expected to be found in this data than in the WALLABY survey, it will provide deeper information on the role and evolution of HI over significant time-frames.

The MeerKAT array [20] will be conducting its own ultra deep investigation: the Looking At the Distant Universe with the MeerKAT Array (LADUMA) survey [46, 47]. This ultra-deep survey will provide the first complete view of galaxy evolution for $0 < z < 1.4$ over roughly 5deg^2. Similarly to DINGO, the first phase will consist of observing a single pointing (1000 hours) to observe to a depth of $z \sim 0.6$ which is expected to provide thousands of direct detections [46]. The second phase will involve the observation of HI to $z = 1.4$ (4000 hours of observation time). Over 2 TB of imaged data will be created for this survey.

The survey sizes expected for future HI surveys and the image data they produce are orders of magnitude larger than current surveys increasing the amount of work required to extract valid sources out of noisy data sets. Current survey sizes have already necessitated that semi-automated methods be used instead of manual extraction (by eye) as the time taken to process the data is infeasibly long. This has motivated the need for fully automated solutions. In the next section we discuss some of the leading automatic solutions and how these techniques may be insufficient to process surveys in practical time-frames and will likely be required to be used in conjunction with high performance computing solutions.

2.6 Automated Source Detection

Automated detection in blind surveys arises from the need to extract HI sources from image survey data that are orders of magnitude larger than previous surveys. It is no longer feasible to have a manual component in source extraction to complete this process in practical time-frames. Several source-finders have been developed in response to this need, the quality of a source-finder being defined by two metrics: completeness and reliability [81, 101]. Completeness refers to the measure of how many sources present in the data set were found. Reliability is the measure of the number of found sources that are actually sources, as noise has the potential to be detected as a source.

The current generation of source-finders fall into two main categories: finders that rely on

intensity and thresholding techniques and finders that utilise statistical likelihoods to determine true sources. We evaluate these source-finders on their computational requirements and their completeness and reliability metrics which are reported in Jurek [52] and Popping et al. [81]. These evaluations are by no means exhaustive but give an approximation of source-finder performance [81] for specific parameter space.

2.6.1 Gamma-finder

The Gamma-Finder package [13] is a Java application which uses a source extraction technique based on the statistical Gamma test [91]. By estimating the noise variance in a continuous data set, a Gamma signal-to-noise ratio is calculated and used as a threshold to assess if an object is a detection [81] or background noise. Although essentially a basic peak over threshold detector, this finder has been shown to detect sources at fainter fluxes than other source-finders [13]. A potential drawback is that the software only produces the position and strength of sources and provides no additional parameters [13]. Nonetheless, this does not disqualify its use as it could potentially be used in conjunction with a source parametrisation package.

Evaluation of this source-finder [81] showed that it was the better finder for objects with a narrow width and strong flux. Overall, this source extractor showed good performance in completeness for high fluxes and significant completeness in the low fluxes.

2.6.2 CNHI

The CNHI source-finder [52] uses a different technique to intensity thresholding which attempts to compensate for the weakness of intensity thresholding in detecting large low flux signals in the presence of noise. Data cubes are seen as collections or bundles of spectra [81]. A Kuiper test [90] is performed to determine the probability that a test region of contiguous voxels and the rest of the spectra come from the same distribution of voxel flux values. This is succeeded by the Lutz real time analysis algorithm [64] to combine spectra which have a low probability of being noise into distinct objects to form a detection.

Evaluation of this algorithm [81] indicated the CNHI algorithm performed well for extracting model galaxies with moderate spatial profiles and objects with a low flux intensity. However, CNHI performed poorly when extracting point sources and had a poor reliability as it is difficult to eliminate false positives from the source list. Additionally, this approach requires fine tuning as the test region cannot be small or it will invalidate the statistical constraints of the Kuiper test.

2.6.3 DUCHAMP

Despite using an intensity thresholding scheme, the DUCHAMP source package [104] is a robust source-finder. This robustness is provided through several preprocessing operations which improve the reliability and completeness of results. This includes inversion operations to search for negative features, removal of large scale structures such as continuum ripples, spatial and spectral smoothing as well as complete wavelet reconstruction using the *à trous* algorithm [108] to reliably reduce noise in the observation [32]. It is a multiscale algorithm which allows the removal of artefacts such as RFI, instrument error and continuum ripples of varying sizes

within the observation. However, the performance of the *à trous* algorithms comes at the cost of high memory use and computation.

Unlike other source-finders, the intensity threshold attempts to reduce the number of false positives statistically in detection by using the False Discover Rate (FDR) algorithm. This allows for a good balance between a high completeness and the reliability which is in contrast to more stringent thresholds such as the Bonferoni threshold [9, 69] (further discussed in Section 3.2) which ensures maximum reliability but severely reduces the number of sources found. This software forms the core of the remainder of this thesis and a more detailed discussion of the components and algorithms within DUCHAMP will be covered in Chapter 3.

In the Popping et al. evaluation [81], the DUCHAMP source-finder is shown to perform best when extracting point sources compared to the competing source extraction packages. However, this finder was also shown to be incomplete for small fluxes and suffered when parametrising sources. Further evaluation of this source-finder in Westmeier et al. [102] showed that it is adept at finding sources at low signal to noise ratios. An improved version of DUCHAMP (Selavy) is set to be used within the ASKAP to produce its scientific outputs [50].

2.6.4 2D-1D wavelet reconstruction

The 2D-1D wavelet reconstruction finder [33] is a multidimensional wavelet denoising scheme similar to DUCHAMP. However, unlike the DUCHAMP package, this algorithm implements a two phase reconstruction which consists of a 2D wavelet reconstruction procedure to smooth all the channels maps and a 1D reconstruction procedure to smooth all spectra. This approach is to account for the significant differences between the spatial and spectral sizes of sources. This separated method allows for the multiscale *à trous* algorithm to easily remove specific artefacts which affect the spatial and spectral domains at different scales. Additionally, this algorithm is well suited to the extraction of anisotropic (directionally biased) sources.

The Popping evaluation [81] shows similar results to DUCHAMP when extracting point sources. In general, this source-finder has high reliability but suffers as the false detections in the final source list are difficult to eliminate. These elimination difficulties result from the majority of false detections being similar to true detections in terms of flux value and size [33]. Furthermore, although 3D curvelets [108] are better suited to anisotropic source extraction, they are computationally expensive [33]. In contrast, the 2D-1D reconstruction is a computationally light algorithm with moderately good anisotropic extraction results [33].

Reliability in the 2D-1D wavelet reconstruction can be further improved by adjusting clipping thresholds in order to fine-tune the algorithm for specific observations. However, this is also true for many of the ASKAP survey competing algorithms.

2.7 Computational Requirements

Despite the performance advantages of automated source detection software, the large data volumes expected from HI surveys on SKA precursor instruments cannot be processed in practical time-frames without large amounts of computing power. Computing resource requirements are generally met with the implementation of specialised hardware, clusters or distributed systems. However, situations arise where access to large computing solutions may be limited and necessitates the use of computing hardware that is more accessible such as 'desktop' workstations.

'Desktop' hardware consists of low cost standardised computing components used primarily for small scale computing problems. However, the development of high speed multi-core CPUs and general purpose graphics devices has enabled 'desktop' hardware to compete as a high performance computing solution. In this section we discuss the specifics of high performance CPUs and how certain resources can be exploited to increase computational performance. Additionally, we discuss the limitations of 'desktop' hardware's memory hierarchy, covering both caching and out-of-core computation issues. We discuss potential mitigation techniques to reduce the poor performance associated with out-of-core computation.

However, implementations of source-finders that exploit these technologies may not be sufficient to facilitate high performance source extraction as the algorithms within the majority of source-finders are largely unoptimised. Performance can be further increased by optimising or reducing the computational complexity of these algorithms. We discuss the theory behind Separable Filtering, an optimisation technique which can dramatically reduce the computational complexity of filtering operations (with certain criteria) commonly used in computer vision and consequently source extraction.

2.7.1 CPU Technologies

Recent developments in CPU technology have allowed CPU performance to increase despite the existence of several limiting factors such as the power wall (the exponential increase in power required to increase clock speed) and memory wall (the relatively slow data transfer between CPU cache and physical memory). In this section, we discuss CPU technologies discussed and used within the remainder of this thesis to maximise CPU performance, namely instruction level parallelism, multithreading, simultaneous multithreading, vector processing and multi-core CPUs.

Instruction Level Parallelism (ILP) is the overlapping execution of several computation instructions at once [73]. Commonly ILP is exploited in three ways: pipelining, out-of-order execution and branch prediction. Pipelining exploits the fact that a single CPU instruction consists of several stages which are computed in-order on independent parts of the CPU circuitry [73]. The different stages of multiple instructions are able to be computed concurrently on these independent CPU circuitry parts, which increases CPU throughput. Out-of-order instruction execution allows for the next instruction to be processed whilst a previous instruction is stalled on a costly memory access and no dependencies exist between these instructions [73]. This attempts to ensure that all CPU clock cycles are used to compute some work and the CPU does not lie idle. Branch prediction extends ILP by assuming the state of a particular branch condition whilst that condition is evaluated in order to save clock cycles (if predicted correctly) [73].

Multithreading CPU functionality allows a single CPU to schedule several pieces of work (threads of execution) from the same process or several processes to allow the interleaving of work (out-of-order execution) [4]. When a thread being processed stalls on a relatively costly memory access and cannot execute further instructions out-of-order, the CPU is free to schedule and process another thread in the interim. However, originally this meant only one thread could be in the CPU pipeline at a given time. Whilst this increases the use of CPU computational resources, the use of multiple threads can result in a race condition. Race conditions arise when two threads perform order dependant operations on the same data out-of-order [4]. Order dependence is not ensured by CPU thread scheduling and it is the task of software developer to organise thread access to shared data.

Simultaneous Multithreading (SMT) CPU functionality improves on ILP and multithreading by combining these technologies [73]. Independent threads can be executed in the same pipeline stage by duplicating the portion of a CPU register memory that stores the state of a thread [65]. Thus a SMT single processor contains more than one logical CPU core which better schedules work for the compute components of the CPU to decrease the number of wasted clock cycles and increase computation throughput. Hyper-threading is Intel's implementation of SMT processing which has the potential to increase multithreading parallel performance by between 4-30% [65, 93] on a single CPU core.

Vector processing CPU functionality [34] allows for the concurrent execution of Single Instruction Multi Data (SIMD) instructions where the same instruction (or set of instructions) is concurrently applied to multiple data points stored in a 1D array (vector). Whilst this has the potential to increase performance it requires the addition of supplementary chip architecture. Intel's Streaming SIMD Extensions (SSE) is facilitated by utilising unique 128-bit processor registers (XMM registers) [38] which hold either 4 single precision or 2 double precision floating point variables as "packed" elements [43]. Instructions executed on a single packed element are applied to all the floating point numbers contained within. The majority of compilers contain functionality for automatic vectorisation (conversion of iterative instructions to vector instructions). However, auto-vectorisation is a difficult problem and compilers may not completely vectorise a piece of software [38]. To fully make use of vectorisation functionality it may be necessary to manually use SSE instructions [3, 10]. The Intel C++ compiler provides a semi-manual approach to vectorisation with SSE intrinsics (SSE instruction sets) to reduce implementation complexity [43, 38]. In general, low-level SSE development in Assembly [43, 11] results in the highest performance but requires large amounts of development time.

Multi-core CPUs are multiple central processing units (cores) on a single device with a shared memory hierarchy [4] (unified shared memory architecture with multithreading). The multithreading capabilities of a multi-core system improves on a single core architecture as independent threads of execution are computed concurrently on each of the available cores. Additionally, each core in this architecture benefits from SMT parallelism and contains the functionality for vector instruction parallelism. The unified shared memory architecture indicates that each core has access to some global address space not distributed across a network: commonly primary memory and higher levels of cache memory (small, fast memory on the processor) [73]. Consequently, memory access is more intuitive to the software developer as it uses a single address and the proximity of this memory to the CPU (not distributed across a network) ensures fast access. However, scaling shared memory architectures is difficult as the addition of extra cores dramatically increases the communication traffic required to keep the multiple cores' caches synchronized.

Multithreading parallelism on multi-core architectures has the potential to greatly improve software performance. However, this requires that the considered software is well-suited to parallel implementations (lack of dependencies). To utilise the benefits of multithreading, the software developer is required to manually launch/delete threads and ensure thread safety to prevent synchronization errors (race conditions) in a particular process [4]. Additionally, multiprocessors introduce core/load balancing performance concerns as thread work load must be balanced between cores in order to achieve optimal parallel performance. However, manual thread management with low level APIs, such as PThreads (A POSIX thread standard)[62, 4], is time consuming and difficult to optimise. This can be simplified with high level thread management APIs such as OpenMP [4, 18, 38]. OpenMP is a multi-platform API for multiprocessor, shared memory architectures that simplifies thread management via instruction sets that are interpreted by the compiler (pragmas) to implement multithreading. Whilst this shortens development time, the developer is required to ensure thread safety and balance work

load between threads [4, 38]. Maximising parallel performance is still a complex task despite the thread management simplifications that OpenMP provides.

2.7.2 Out-of-core computation

Out-of-core computation occurs when the memory allocated by a program (the working set) is larger than physical memory (in-core computation) and must be temporarily stored on secondary storage. Virtual memory abstracts this memory addressing across physical memory and swap partitions on secondary storage to simulate a large homogeneous memory space. This allows large problems to be processed on systems with limited physical memory, such as 'desktop' workstations. However, disk access is still orders of magnitude slower than physical memory access. Frequent access to disk can therefore result in a performance bottleneck while computation waits for data to be transferred (transfer bound).

An alternative to the standard paging protocols for handling large data is the use of file-backed memory mapping [55]. A file-backed mapping is a byte-to-byte association of virtual memory space with a file on disk. This scheme has several advantages over the standard disk paging and standard file I/O. Memory mapping does not require the standard file I/O's intermediate step of copying first into the standard I/O buffer before copying into the supplied data structure [63] as the file-backing is its own page cache. File size is not constrained by the size of allocated swap space and physical memory but by that of secondary storage. Additionally, memory mapping allows for fragmented file mappings and does not require the existence of a large free contiguous region on disk. In contrast, swap partitions in Linux are contiguous [100] which allows for better disk access but prevents scaling to accommodate larger memory use.

Memory mapping schemes implement demand paging [82] by default which results in conservative physical memory use but has the initial disadvantage of data completely residing on disk at the time of mapping creation. An initial populating of physical memory is required before computation can begin on memory mapped data structures. However, actual memory mapping implementations often improve on demand paging by prefetching data through custom paging schemes.

Memory mapped data structures completely abstract access across the memory hierarchy [63] and are often indexed and utilised in the same manner as STL structures. Minimal redevelopment is required to integrate mapped structures into existing applications. This is in contrast to standard file I/O commands where the user must implement their own buffering scheme.

Three memory management libraries are considered over the remainder of this thesis, namely the Mmap, Boost.Interprocess and Stxxl libraries. We briefly introduce these three libraries, as follows.

MMAP The Linux system kernel facilitates memory mapping through the Mmap() system call [63, 55]. This memory management library is the simplest out of the three considered libraries. A Mmap mapping made with a user created file on disk functions as a normal C/C++ array. To ensure consistency with its file-backing, changes to the paged-in portion of the memory map must be flushed to disk. Optimised memory managed STL data structures and multithreading are not supported.

Disk access with a Mmap memory-mapped data structure can be optimised using Madvise() [63, 54] which allows the developer to specify paging advice to the kernel. Madvise calls include general advice for memory access patterns as well as the manual control of paging. When

normal paging is requested the kernel will readahead a moderate amount of pages off disk. Sequential paging advice increases readahead and releases memory soon after the page is used. Random paging disables readahead as it serves no real purpose with randomly accessed data.

Manual control of the paging scheme is facilitated with WillNeed and DontNeed. WillNeed specifies which portion of a mapped file is likely to be used next and prefetches from disk. DontNeed dumps all resources allocated to a range of pages (changes are lost if not flushed back).

Boost.Interprocess Boost.Interprocess [37] is part of the Boost C++ libraries and is primarily used to facilitate communication between two separate processes. This communication occurs through a file-backed memory mapped region and can be accessed as a normal C/C++ array. Additionally, all Boost data structures are compatible with Boost memory mapping and additionally supports multithreading to these mapped structures.

Boost.interprocess calls are platform independent, making Boost the easiest memory-mapping library to port between Mac, Linux and Windows systems. However, in contrast to Mmap, no functionality was found within Boost.Interprocess to manually optimise the paging scheme used.

Stxxl: Standard Template Library for large datasets The Stxxl library is a standard template library (STL) which has been optimised to handle large data [25]. Several Stxxl versions of common data structures and STL algorithms (searching/sorting) are implemented within this library. Stxxl memory management is implemented by using a dynamically growing memory mapped region on one or more disks which facilitates software RAID (redundant array of independent disks) [6].

Available optimisation functionality is significantly larger than that of Boost and Mmap. This includes the definition of how many blocks of memory and the size of these blocks to be paged into physical memory at any given time [24]. Additionally, several RAID disk assignment and paging strategies are supported.

2.7.3 Separable filtering

Digital image processing (signal processing) is widely utilised for the problem of astronomical source extraction to remove and/or emphasise features in observational data. A common use case involves filtering to reduce noise and spurious detections in an observation to highlight true sources and aid in the extraction, identification and parametrisation of these objects [84, 28, 67]. This is accomplished by convolving finite discrete filters ($A \in \mathbb{N}$ dimensions) with data (B dimensions $\geq A$), here the filter responses generated form the new image or data. However, convolution with finite discrete filters can be computationally expensive for large multi-dimensional filters.

Fortunately, the computation required to calculate multi-dimensional filter responses can be drastically reduced with separable filtering techniques [31, 87, 71] if the filter is separable. Separability is defined as a Q dimensioned filter being able to be produced from the outer product (or convolution) of $1 < X \leq Q$ filters [31]. For a given filter that meets these conditions there are often multiple decompositions. However, for image processing we ideally want a filter that can decompose into its simplest components, Q 1D filter components.

Filter separability and the associative property of convolution [31] allows a convolution operation with a Q dimensioned separable filter F and data D to be decomposed into a convolution with filter F's separate components in any order. Assuming the filter is perfectly separable:

$$\begin{aligned} D * F &= D * (f_1 * f_2 * f_3 * ... f_Q) && f_i \text{ is a separable component of } F \\ &= (D * f_1) * (f_2 * f_3 * ... f_Q) && Q \text{ is the number of dimensions in } F \\ &\bullet \\ &\bullet \\ &\bullet \\ &= (D * f_1 * f_2 * f_3 ... f_{Q-1}) * (f_Q) \end{aligned}$$

This equivalence allows the separate filter components to be applied to the data in any order and still produce the same results of applying the original Q dimensioned filter.

The advantage of separable filtering is that the number of floating point operations required to compute the convolution operation is now significantly reduced [31]. For the normal convolution procedure, the filter is size $F_{size} = \prod_{i=1}^{Q} X_i$ where X_i is equal to the size of i-th filter dimension [87]. This filter is applied to each voxel in the data set of size N and consequently computation is proportional to $\mathrm{O}(N \prod_{i=1}^{Q} X_i)$.

In contrast, separable filtering convolves the Q 1D separated filter components (each of of size X_i) with the data in Q separate filter passes. Thus the number of floating point operations is proportional to $O(NX_i)$ for a single pass and $\mathrm{O}(N \prod_{i=1}^{Q} X_i)$ for the entire separable convolution process [87]. Therefore the ratio at which operation complexity is reduced (theoretical performance increase) with the implementation of separable filtering is:

$$\text{Operation Reduction Factor} = \frac{\prod_{i=1}^{Q} X_i}{\sum_{i=1}^{Q} X_i} \qquad Q, X \in \mathbb{N}$$

If the filter is uniform in all directions (isotropic), the reduction factor simplifies to:

$$\text{Operation Reduction Factor} = \frac{X^Q}{QX} \qquad Q, X \in \mathbb{N}$$

Therefore, for isotropic filters, the theoretical performance increase facilitated by separable filtering implementations increases exponentially with the number of filter dimensions and polynomial growth to the Qth degree with filter extent.

Chapter 3

DUCHAMP

The data produced by the planned HI surveys on the next-generation of radio telescopes is expected to be 10^3 to 10^6 times larger than current survey sizes (Chapter 2.5.2). Automated procedures are required to extract objects of interest (sources) from this noisy radio data as a manual approach would require infeasibly large time-frames. Several source-finders have been developed in response to the need of automated source extraction (Chapter 2.6). In the source-finder evaluations in Popping et al. [81], DUCHAMP emerged as a robust source-finder that is best suited for point source extraction and detecting source profiles at low signal-to-noise ratios [102]. This motivated DUCHAMP's further development into the Selavy source-finder which will form a part of the ASKAP HI survey software pipeline [58]. However, the procedures within DUCHAMP (as is the case with many source-finders) are computationally heavy and memory intensive. High performance computing (HPC) solutions are required to scale the performance of DUCHAMP source extraction to allow the planned HI surveys to be processed in practical time-frames.

In this chapter, we discuss the specifics of the DUCHAMP source extraction package version 1.1.13 and the third party software required in its operation. This covers the various components (functions) of this system and the algorithms that each component implements. We expand on the *à trous* wavelet reconstruction algorithm, the most computationally heavy and memory intensive algorithm within DUCHAMP and the focus of the remainder of this thesis. Inefficient and parallelisable components within the *à trous* algorithm are identified to contrast the redevelopment of these components discussed in subsequent chapters.

3.1 Overview of the DUCHAMP software pipeline

The DUCHAMP software was developed as an automated 3D source-finder to extract sources from radio spectral cubes [103, 104]. However, the package is sufficiently generalised to additionally handle 2D data, such as continuum observations, and 1D data for processing a single spectrum. DUCHAMP is developed primarily in C++, with the majority of the functions using templating (precision is equal to the input data), but utilises several low level ANSI-C libraries, discussed below.

The strength of this software lies in its ability to suppress noise in an observation via pre-processing in order to improve the reliability and completeness metrics (Chapter 2.6) of the succeeding source finding algorithm. This source finding algorithm uses intensity threshold techniques which make no assumptions on source shape and are well-suited for blind surveys,

where position, shape and size are unknown. DUCHAMP can statistically define the threshold used to reduce the number of false positives during source detection. This allows for a good balance between completeness and reliability, as maximising for one metric often reduces the other. However, thresholding approaches come at the cost of source parametrisation reliability as the entire extent of sources may be missed if a portion of the emission lies beneath the intensity threshold [102]. This is partially rectified with the DUCHAMP functionality to grow found sources to a second lower threshold but is still inferior to parametrisation in alternative source-finders.

The DUCHAMP package consists of seven main phases (Fig 3.1) of operation: loading inputs, optional pre-processing, searching (thresholding), source merging, undo pre-processing, parametrisation and generation of output. The majority of these phases consist of multiple algorithm alternatives which can be selected by the user to configure and fine-tune the source extraction process. In the first phase, data is first read from a file and the relevant parameters that govern the execution specifics are set. Optional preprocessing implements procedures to both suppress noise/features in the data set to improve source detection and reduce memory use (under specific conditions). This pre-processing includes the computationally heavy *à trous* wavelet reconstruction algorithm which is the focus of this thesis. The Searching step involves defining a threshold which is used to test for a voxel significance as true source emission and eliminate background noise. Significant voxels are then merged into complex 3D objects (in the case of spectral cubes) which are further validated with respect to size in order to eliminate small high-intensity noise spikes. This is followed by the undoing of preprocessing to ensure that the correct source parameters are measured by DUCHAMP (e.g. flux). Finally, parametrisation of detected sources measures all relevant source properties and outputs them to the source catalogue along with graphical outputs. We discuss the specifics of each DUCHAMP component, as follows.

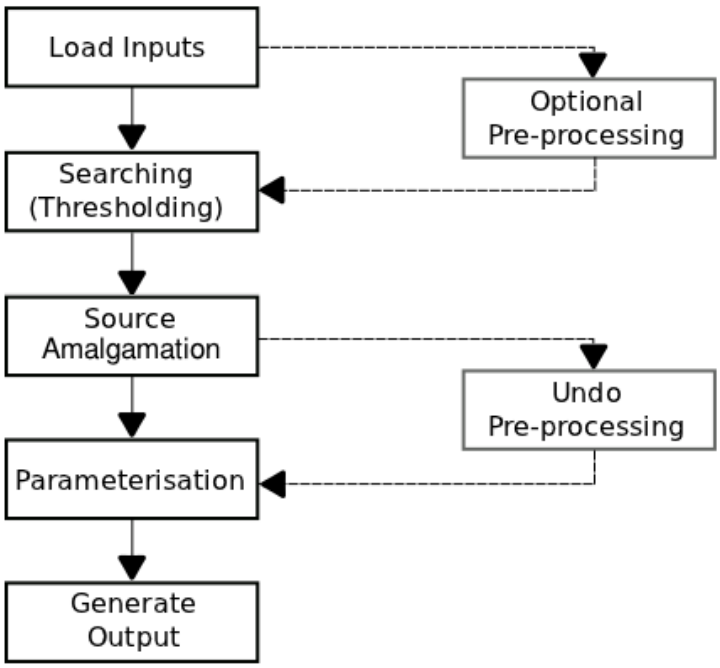


Figure 3.1: Overview of the DUCHAMP source extraction software pipeline. Figure adapted from source: Whiting, M., DUCHAMP: a 3D source-finder for spectral-line data, 2012 [104].

3.1.1 Loading Input

This DUCHAMP component handles the loading of image data and the reading of the parameter file that defines the variables and algorithms used in execution. Image data is usually in the form of a 3D Flexible Image Transport System (FITS) [98] cube or similar format that conforms to the FITS standard. However, DUCHAMP is general enough to accept 1D spectral and 2D continuum data in this format. The FITS format is the most commonly used format in astronomy and is supported by many of the source-finders reviewed in Chapter 2.6. However, we note that the use of FITS is in decline as currently it does not scale to the size and complexity of modern astronomical data sets [92].

To read observational data from the FITS file, the DUCHAMP package uses the CFITSIO library [76, 78] which abstracts the format specifics. The multi-threaded loading functionality of CFITSIO is not used by the sequential DUCHAMP package v1.1.13. At present, CFITSIO limits file size to approximately 6 TB (2^{31} 2880 byte FITS file blocks) [77]. This maximum file size may restrict FITS file use in future SKA surveys but is sufficient for the ultra-deep and ultra-wide surveys on ASKAP and MeerKAT where the largest spectral data cube expected is 782 GB (WALLABY survey) [52]. Metadata concerning the considered observation, such as beam size and correlation between voxels, is accessed in the FITS file header using the WCSLIB library [15]. I/O performance with CFITSIO and WCSLIB is a trivial contribution to the total run-time of the DUCHAMP system as data/metadata is only read once.

DUCHAMP allows for preprocessed data, discussed below, to be accepted as input to prevent recalculation of computationally expensive noise suppression algorithms when performing multiple searches with different search criteria. These criteria include stricter/looser thresholds and changes to the allowed minimum distance between two objects to consider them the same object to refine searching for faint or fragmented source profiles.

3.1.2 Preprocessing

Preprocessing is an optional DUCHAMP component that precedes the DUCHAMP source extraction functionality. This component contains a variety of procedures to improve both the computational performance and the effectiveness of the source extraction process [104, 103]. Procedures which increase computational performance attempt to exclude regions of data in order to both reduce memory use and decrease the number of voxels that require processing. In contrast, the majority of procedures which improve source extraction effectiveness add significant amounts of computation. These procedures suppress noise and large scale features in the data to improve the completeness and reliability metrics of the succeeding source extraction procedures. However, these methods only suppress noise and do not remove it completely.

System performance is improved using the Channel Exclusion and Trimming procedures. Channel Exclusion reduces the amount of computation by defining a continuous range of channels (or frequency slices) to be ignored. This is useful when a large number of sources are expected in a frequency range of non-interest which would slow down the subsequent $O(N^2)$ source-finding amalgamation procedures, discussed below. This function is commonly used to ignore the Milky Way emission bands. The Cube Trimming procedure minimises memory use when data contains substantial padding information. This padding data (blanked voxels) ensures that non-cubic observations are stored correctly in the FITS spectral data cube (of a particular size). Such blanked voxel edges are common in interferometric data where multiple circular beams have been combined into a single mosaic. Trimming removes the majority of this padding to

reduce memory use and the number of voxel validity checks required throughout the remaining software pipeline.

Procedures to improve source finding completeness and reliability include spectral baseline removal, spectral smoothing, spatial smoothing and wavelet reconstruction. We note that only the spatial smoothing is templated and potentially executes at double precision whilst the remaining procedures are cast to single precision. Spectral baseline removal uses the wavelet reconstruction algorithm, discussed below, to remove large bright spectral features to potentially improve the detection of faint objects. The significant features at the two largest scales (size of object considered) are removed temporarily and replaced in the final steps of the DUCHAMP pipeline.

Spatial smoothing uses a 2D elliptical Gaussian kernel (size and orientation defined by the user) to smooth each channel map of the input data set separately. Similarly, spectral smoothing is implemented by convolving with a Hanning filter [94], a simple averaging filter to smooth each spectrum individually. The width of this filter (size defined by the user) defines the degree of smoothing that occurs. Both of these procedures are effective at suppressing noise when the respective spatial and spectral extents (approximate) of the sources of interest are known [104]. However, both smoothing techniques focus on a single object size which makes them unsuitable for noise suppression in blind surveys where source extent is potentially diverse.

The wavelet reconstruction procedure improves on the previous two smoothing techniques by implementing the *à trous* wavelet reconstruction algorithm. The algorithm is considered to be a vital component [81, 102] in improving DUCHAMP source extraction's completeness and reliability. This algorithm consists of an *à trous* wavelet decomposition where features at each scale are thresholded for significance. Significant features generate the reconstructed/smoothed spectral cube where noise in the data has been suppressed. DUCHAMP implements 1D, 2D and 3D reconstruction options to cater for multi-dimensional input data. The significance threshold is defined using the noise statistics of the data set which can be measured with either normal or robust statistical measures. The mean and standard deviation (normal statistics) are a computationally light way of measuring the noise level and spread in the data but are easily biased/distorted by bright sources. Robust statistics, namely the median and Median Absolute Deviation from the Median (MADFM) [80], are less sensitive to outliers but require a partial sort of the data which is usually more computationally expensive than standard statistics. However, the computational costs are relatively small in comparison to the wavelet decomposition itself which is both computationally and memory intensive (especially in 3D reconstruction) and is the main cause for the *à trous* algorithm taking up the majority of run-time in DUCHAMP (Chapter 6.1.3). A more detailed discussion of the specifics of the *à trous* wavelet reconstruction algorithm is given in Chapter 3.3.

3.1.3 Searching (thresholding)

DUCHAMP's Searching component extracts (or separates) sources from noisy data; background noise will still exist after the optional noise smoothing preprocessing algorithms. However, preprocessing is useful in decreasing the number of false detections (Type I error) caused by noise and the number of valid sources missed (Type II error). The DUCHAMP search algorithm tests each individual voxel's flux value against an intensity threshold to categorise the voxel as a likely source or background noise. These simple voxel detections are built into complex structures in subsequent stages of the DUCHAMP pipeline, discussed below. The per voxel thresholding used in DUCHAMP is in contrast to source finding techniques such as statistical detections and match-filtering which test the likelihood that a particular region is a valid source.

The DUCHAMP package specifies three methods to determine threshold: namely, flux value threshold, signal to noise threshold and the False Discovery Rate. The flux value is specified by the user and is an absolute threshold. The signal to noise ratio threshold uses a multiple (user defined) of the noise spread in the data and consequently partially adjusts to the noise properties specific to that observation. The noise spread is determined by the image statistics using either normal or robust statistics, discussed above.

The False Discovery Rate (FDR) [9] algorithm is a fully automated solution that defines a statistical threshold that controls the amount of false discoveries (Type I error) during searching. Defining higher thresholds reduces Type I error in the searching process but can dramatically increase the number of fainter sources missed (Type II error). The FDR algorithm constrains Type I error whilst keeping Type II error low, effectively finding a good balance between these error metrics. Additionally, Type I error is reduced in the later stages of DUCHAMP through further validation procedures, discussed below. The specifics of the FDR algorithm are discussed in Chapter 3.2.

3.1.4 Source Amalgamation

The source amalgamation component of DUCHAMP forms 3D source objects from the individual "source" voxels detected in the preceding searching component. This occurs in two stages: a connected-components scan, which searches either the spatial or spectral domains for connected voxels, and a further merging procedure to allow for the formation of 3D objects and to merge nearby objects based on user criteria. Faint source edges are then detected and added to these objects before undergoing further validation procedures.

DUCHAMP implements two connected-components algorithms which search either the spatial or spectral domains for connected voxels. The spatial algorithm is a Lutz 2D connected-components [64] procedure, a raster scan that builds up 2D connectivity by scanning each row in a channel map and checking adjacency with objects on the previously scanned row. This generates all possible 2D objects in a particular channel map and is applied to each channel map separately which makes this procedure highly parallelisable. Spectral scanning uses a similar process but only considers individual spectra and has the potential to initially find many distinct objects which slows down the subsequent merging procedure.

Complex 3D objects are built up by merging 1D spectral objects or 2D spatial objects that are in close proximity or adjacent to one another. Close proximity is defined by the user as the maximum distance allowed between two source objects (in both the spatial and spectral domain) for them to be considered as the same object. Merging follows a two-pass procedure: an initial check when an object is added to the list of objects which is followed by a more rigorous $O(N^2)$ comparison and merging process [104]. Due to this algorithm's computational complexity, heavily populated observations can be extremely computationally expensive.

The detected source objects then have the option to "grow" down to a lower secondary threshold (user defined absolute threshold or signal-to-noise ratio). This procedure allows for the valid faint edges of a source to be detected which improves source parametrisation. If the growing procedure is performed it is followed by an additional merging operation.

Finally, the object list undergoes a pruning procedure which eliminates all objects that are insufficiently large in both spatial and spectral extent. This process removes any bright single voxel objects which are generally spurious detections of noise spikes. This pruning further reduces the Type I error in detection to below that statistically enforced in the False Discovery Rate algorithm.

3.1.5 Parametrisation and Output

The final step in the DUCHAMP pipeline is parametrisation where the useful properties of the detected sources are calculated. This component requires the reversal of certain preprocessing procedures before execution, namely Cube Trimming, Inversion and Baseline removal. The exact techniques used to parametrise all detected sources are discussed in Whiting 2010 [103]. The variables that are calculated for each source are:

- Peak position - location of the source object described as the voxel with the highest flux value.
- Average position - the central position of the source, furthest away from all source edges.
- Centroid position - flux weighted average for all voxels in each dimension.
- Source size - the extent of the source in the spatial and spectral directions.
- Number of voxels - the number of voxels that make up the detection.
- Velocity Width - the full velocity width of the detection, aka its extent in velocity units.
- Noise characteristics - the spread and level of noise before and after preprocessing.
- Peak Flux - the peak flux over a source.
- W50 - width of source at 50% peak flux.
- W20 - width of source at 20% peak flux.
- Total Flux - the sum of all flux values in a particular source object.

Additional outputs include a FITS cube that contains the detection mask of detected sources and preprocessed data. The latter is to avoid having to redo computationally heavy preprocessing when performing source extraction with different variables to fine-tune the process.

Graphical outputs include both a spatial 0th moment map which shows the channels of each object and spectral plots which displays the entire spectrum of each detection at all spatial coordinates and the 0th moment of that spectrum. The actual detection within the spectrum is indicated.

3.2 False Discovery Rate Threshold

In this section we discuss the specifics of the False Discovery Rate (FDR) algorithm and motivate its use in the Searching (thresholding) step of the DUCHAMP pipeline by contrasting this algorithm to the common statistically defined threshold alternatives. In the strictest sense, statistical thresholding is a multiple hypothesis test where each voxel is tested to fit the model to some probability [9]. A voxel intensity is tested against the null hypothesis to determine whether it fits the Gaussian noise model; a rejection of the null hypothesis classifies a voxel as a likely source. However, when performing this test, the presence of noise can result in both Type I error (false discoveries) and Type II error (the non detection of valid sources) [9, 104]. Defining a high threshold reduces the probability of a false discovery but may result in many valid sources being missed. The FDR algorithm finds a reasonable balance between these metrics. However, we note that the DUCHAMP FDR implementation is computed at single precision only.

The four common statistical alternatives to FDR are: 2σ, 3σ, Family Wise Error Rate (FWER) and the Bonferoni threshold [9, 69]. In the 2σ approach, a voxel is compared against a threshold of 2σ, where σ is the measure of noise spread in the data, to constrain the probability of false detection to approximately 5% (p-value = 0.05). The 3σ approach is a stricter threshold which reduces the probability of false detections further. However, the probability of error in these techniques grows with the number of voxels in a particular data set [69]. The large data sets expected in HI surveys will likely result in Type I error that is unacceptably high.

The FWER and Bonferoni methods prevent the error increase seen in 2σ and 3σ by adjusting to data size. The Bonferoni threshold corresponds to a p-value of $\frac{\alpha}{N}$ where N is data set size. This threshold is particularly strict and results in the non-detection of many valid sources (Type II error). The FWER threshold attempts to control Type I error while maintaining a small probability of Type II error by constraining the probability of finding one false source to α. Unfortunately this technique can be considered too strict. The FDR is similar to FWER as it constrains Type I error to a set α instead of trying to reduce it [69]. However the FDR threshold [69] results in a lower probability of Type II error than the FWER (under specific conditions, equal otherwise) and Bonferoni algorithms. Additionally, the FDR algorithm can easily account for correlation between voxels which is common in radio observations.

The algorithm to calculate the threshold is as follows [103]:

1. Convert all voxel intensity values to their corresponding p-values. This is the probability of getting a voxel intensity of this magnitude assuming the null hypotheses that the voxel is not a source is true.
2. Order the p-values in ascending order.
3. If the data is correlated we define the normalisation constant C_N as $C_N = \sum_{i=1}^{CR}(i^{-1})$ where CR is defined as the product of Beam size (B) and the number of correlated channels (C). Beam Size is calculated from the major and minor beam axis information found in the FITS header file. If the data is uncorrelated CN is set to 1.
4. α is set to p-value the desired error rate (user defined). The default value is set at 0.05.
5. Calculate $d = max\{j : Pj < \frac{j\alpha}{C_N N}\}$. D is equal to the maximum index (j) from the ordered p-value list where the stated inequality holds. Pd (position D on the ordered list) is the p-value threshold at which we reject the hypothesis that a voxel belongs to the background and is therefore a source voxel.
6. The p-value Pd is converted to an intensity threshold value. All voxel intensities are tested against this threshold. Voxel intensities exceeding this threshold are considered source voxels.

The majority of computation time is concentrated to Steps 2 and 4. Although the procedure to convert the p-value threshold (Step 3) to an intensity threshold is brute force, it is only calculated once and the amount of computation required does not scale with problem size. The dependencies between algorithm steps and within each step prevent parallelism.

3.3 The *à trous* Wavelet Reconstruction algorithm

In this section, we discuss the specifics of DUCHAMP's *à trous* wavelet reconstruction algorithm. This discussion will include identification of areas suitable for parallelism and the memory use of this algorithm. The *à trous* wavelet reconstruction algorithm [102] improves

the reliability and completion metrics for DUCHAMP source extraction by smoothing noise in observational data. This smoothing procedure considers a range of scales (size of object considered) to ensure effective noise suppression in blind surveys where potentially detected sources sizes are unknown. However, we note that the DUCHAMP implementation of this algorithm is computed at single precision only.

The reconstruction algorithm uses the discrete *à trous* wavelet decomposition (or transform) [86] which convolves low pass filter banks with the data to extract information (wavelet coefficients) at different scales. These wavelet coefficients, or filtered values, are thresholded for significance in order to discard erroneous structures at every scale. The range of scales considered can be defined by the user, with a default minimum scale of 1 and a maximum scale of $S = \log_2(d_{small}) - 1$ where d_{small} is defined as the size of the smallest dimension of the considered data set. To retrieve all the significant features within the data, it is necessary to repeat this process on the input data until only noise remains.

Three versions of the *à trous* wavelet reconstruction are implemented in DUCHAMP, namely Spectral (1D), Spatial (2D) and 3D reconstruction. The most commonly used in HI source extraction is 3D reconstruction as it smooths spectral cubes containing objects with a 3D extent. These three reconstruction versions are very similar and only differ by the number of dimensions in the low pass filter used to convolve the data at each stage. The filters used in this convolution must adhere to specific criteria [86] in order to generate a valid *à trous* decomposition. The filters provided in DUCHAMP are a B-spline filter ($\frac{1}{16}$, $\frac{1}{4}$, $\frac{3}{8}$,$\frac{1}{4}$, $\frac{1}{16}$), a Haar filter (0, $\frac{1}{2}$, $\frac{1}{2}$) and triangle filter ($\frac{1}{4}$, $\frac{1}{2}$, $\frac{1}{4}$) for 1D reconstruction.

To generate the higher dimension filters used in 2D reconstruction, the DUCHAMP implementation convolves a single 1D filter with its transpose to produce a filter with one extra dimension. This is repeated to generate the 3D reconstruction filter which is defined formally as $F_{3D} = F * F^T * F^{T'}$ where F^T and $F^{T'}$ are transposes of the row vector V where the resultant 1D filters are column aligned and spectral aligned respectively.

Multi-scale convolution (Fig 3.2) is facilitated by "growing" the filter for each increase in scale by increasing the distance between filter elements to ($2^{scale-1}$), with a distance of 1 indicating adjacent voxels.

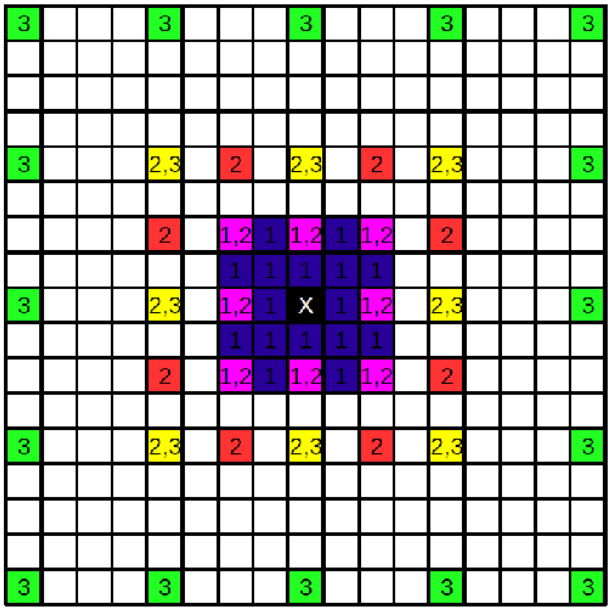


Figure 3.2: Memory access pattern of the 2D convolution filter for Scales 1-3. Figure taken from source: S.Westerlund. Analysis of the parallelisation of the DUCHAMP algorithm, ICRAR. 2009 [101].

Algorithm 3.1 describes the *à trous* wavelet reconstruction algorithm steps, covered in detail in both the DUCHAMP Users Guide [103] and Westerlund [101] assessment paper. Fig 3.3 is a graphical representation of this algorithm which shows the dependencies between algorithm components and their procedure type classification, discussed below. The steps within the *à trous* wavelet reconstruction algorithm can be classified into five procedure types: load, update, statistics, convolution and miscellaneous. Load operations refer to the creation and population of data structures whilst update operations refer to procedures that update the relevant data structures after filtering (includes significance thresholding). Statistic procedures refer to the statistical calculations required to estimate noise (spread and level) in the data, discussed above. The robust statistical measures, namely the median and Median Absolute Deviation from the Median (MADFM), are used to define noise spread and level by default. The convolution procedure refers to the multi-scale filtering process, the memory access pattern of which is further discussed below. Miscellaneous procedures refers to all operations that do not fit in the above categories. For the remainder of this thesis, *à trous* wavelet reconstruction components will be referred to by these types to differentiate between memory access patterns and avoid discussion for every algorithm component when an overview is sufficient.

Algorithm 3.1. *à trous* wavelet reconstruction algorithm.

1. Set the array which will hold the final reconstruction to 0 (reconstruction array)
2. Calculate the initial noise statistics using the input array, which originally holds the observation data.
3. Convolve input array with the chosen filter to calculate the convolved array.
4. Calculate the wavelet coefficients as the difference between the convolved and input arrays.
5. Calculate a threshold to test wavelet coefficient significance.
6. Add wavelet coefficients to the reconstructed array if they are above the threshold.
7. Increase filter scale by increasing the separation between filter elements.
8. Repeat procedure from Step 4 using the convolved array as the input array. Stop when the maximum number of scales to be computed is reached.
9. Add the remaining data in the convolved array to the reconstructed array. This provides the so called "DC offset" as the wavelet coefficients (Step 4) have a zero mean.
10. Calculate the spread statistics (standard deviation) of the calculated difference between the reconstructed array and input array (residual array).
11. Repeat algorithm from Step 3 until the residual spread between algorithm iterations (Steps 3-10 representing a single iteration) is below a small value specified in DUCHAMP. The algorithm is run for a minimum of 2 iterations.

The main operation within *à trous* reconstruction is convolution (Algorithm 3.1, Step 3) which requires the calculation of filter responses, the sum of all filter values which have been multiplied with the underlying data. The central filter element is required to be centred at each voxel to calculate that particular voxel's filter response. Consequently, computational requirements

are proportional to data set size and the number of elements in the filter passed over the data and thus 3D filtering is expected to be computationally expensive. In the case of the B3-Spline filter (125 elements), 125 filter operations are required per voxel for a single convolution pass. A convolution pass is required for every scale within each iteration (repetition) of the reconstruction algorithm which repeats until all significant structure is found. This results in reconstruction taking up to 95% [101] of total run-time. A lower bound of approximately 65% of total run-time was achieved in our preliminary timing (Chapter 6.1.3) of DUCHAMP for observations highly populated with sources, as the succeeding source amalgamation procedure is ($O(N^2)$).

Edge cases in the convolution procedure occur when a portion of the filter extends beyond the limits of the data set (centre of the filter is within the data set limits). These cases are handled by reflecting the filter position across the edge of the spectral cube so that valid spectral data is accessed.

The high run-times of the convolution procedure are partially caused by the inefficient memory access pattern of filter response calculations. Filter elements are adjacent for the first scale considered and consequently access the underlying data linearly for each row of the filter which results in near optimal cache use. However, for scales larger than 1, filter elements are separated by increasingly large increments (Fig 3.2) which causes the data accessed by a row of filter elements to be split across several cache lines. This access pattern results in non-optimal cache use which degrades performance.

The threshold (Algorithm 3.1, Step 5) used to test wavelet coefficient significance is generated using the median of the produced wavelet coefficients, the original noise spread, a user defined signal-to-noise ratio (SNR) and a variable to account for the increased correlation of voxels at progressively higher scales. The SNR threshold, if well-chosen, can suppress the random noise in the observation data set to a large degree [104].

To avoid using additional memory to store the residual array (Algorithm 3.1, Step 10), modified statistical procedures are used. These procedures use both the reconstructed and input array during calculation instead of defining an additional data structure to store the difference between them.

The algorithm is repeated (Algorithm 3.1, Step 14) to ensure that all significant structure is removed from the observation data and added to the final reconstructed array. The algorithm stops when the spread of the remaining noise in the observation data does not change significantly.

For all steps within Algorithm 3.1, non-valid voxels (padding or corrupted observations) are ignored by producing a Good Voxel Boolean mask against which all voxels are checked.

The amount of allocated memory required to run the *à trous* wavelet reconstruction is relatively high when compared to the rest of the DUCHAMP components. This algorithm requires 4 single precision data structures (includes input data) each with the same number of elements as the input spectral data cube. Consequently, the memory allocated for one data structure will be equal to and half that of single and double precision FITS spectral data respectively. The modified statistics used in Step 10 of Algorithm 3.1 prevent the normal statistical procedures from requiring an additional data structure (residual array). However, to calculate robust statistics whilst preserving the data, an additional temporary data structure equal in size to the input data structure is required. Additionally, the Good Voxel Boolean mask required to check validity requires a byte of memory for every voxel in the data set. If we ignore the relatively few allocated variables required by this algorithm we can estimate the data used in the *à trous* wavelet reconstruction algorithm. The memory use for the single precision *à trous*

wavelet reconstruction and a hypothetical double precision implementation is given as follows:

$$\begin{aligned}\text{Memory Use (single precision)} &= MainStructures + GoodVoxelMask + Statistics_{temp} \\ &= 4(N * sizeof(float)) + N + (N * sizeof(float)) \\ &= 17N(+4N) \text{ bytes} \\ \text{Memory Use (double precision)} &= 33N(+8N) \text{ bytes}\end{aligned}$$

Thus the single precision DUCHAMP implementation processing a data set of 4 GB (10^9 voxels) requires $\sim$ 20 GB of allocated memory. Similarly, a data set with the same number of elements (10^9 voxels) computed at double precision would require $\sim$ 38 GB of allocated memory. The memory use of this implementation of the *à trous* wavelet reconstruction algorithm cannot be further reduced.

The *à trous* wavelet reconstruction algorithm, as a whole, is not perfectly suited to parallelism. Data dependencies (Fig 3.3) exist between the majority of the algorithm steps which prohibits parallel execution of tasks. The only procedures that could execute in parallel are the significance thresholding and the "Update data cube values using wavelets" steps (Fig 3.3). The latter operation is a extra update procedure (computationally light) that arises from DUCHAMPs

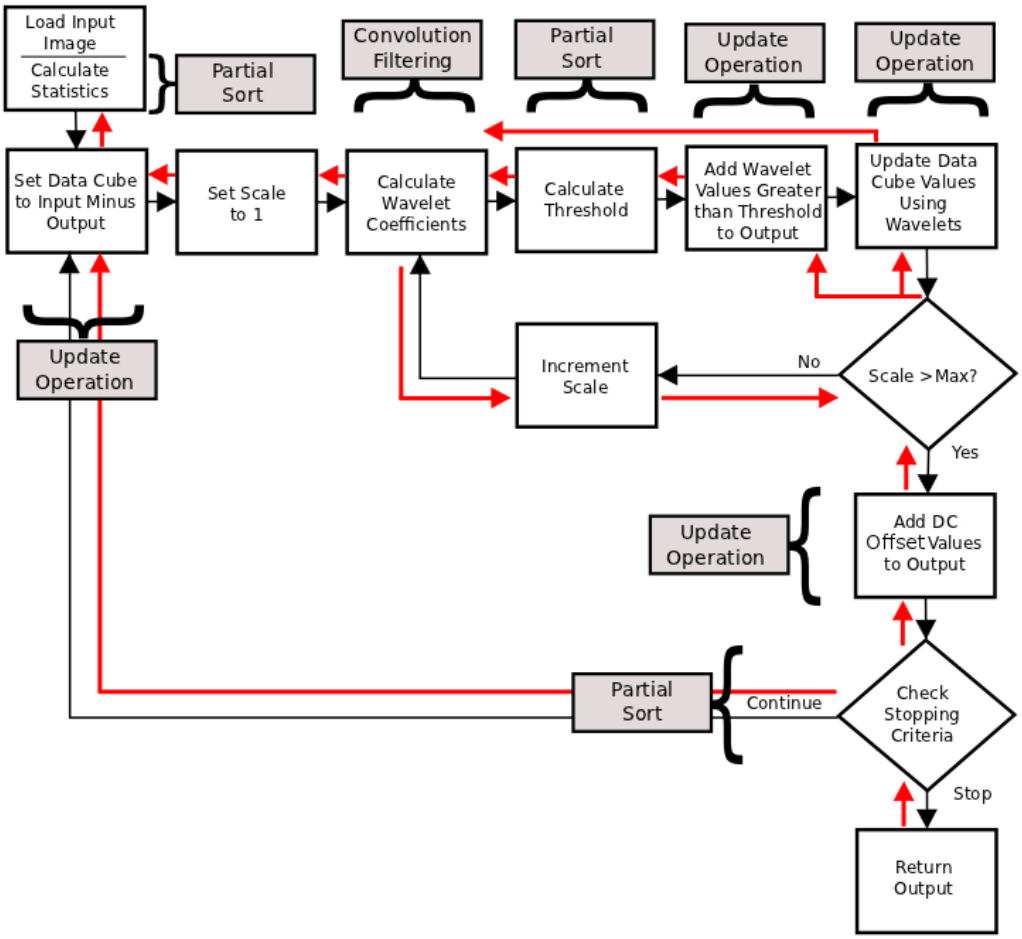


Figure 3.3: Overview of the *à trous* wavelet reconstruction algorithm. Dependencies between components are indicated via red arrows. Components are classified into five classes of procedure based on their memory access pattern; load, convolution, Update, statistics and miscellaneous. Figure adapted from source: S.Westerlund. Analysis of the parallelisation of the DUCHAMP algorithm, iVec Research Internships, ICRAR. 2009 [101].

implementation of Algorithm 3.1 in order to avoid using an extra data structure. Task parallelism at this point would result in trivial parallel performance increases. However, we note that both the Update and convolution procedures involve the execution of one or more floating point operations per voxel. Consequently, these procedure types are embarrassingly parallel and have the potential to achieve large performance increases with parallelism.

Chapter 4

Design

4.1 Introduction

Traditionally, extraction of astronomical source emission from noisy observational data has been performed manually. However, as survey size grew this approach quickly became infeasible and required the implementation of automated source-finding solutions in order to process large past and on-going HI surveys in practical time-frames.

The future HI ultra-deep and ultra-wide surveys proposed for the SKA precursor devices are expected to generate data sets in the Terabyte-Petabyte size range; orders of magnitude greater than those produced by current surveys. Currently, automated source-finding solutions will take infeasibly large time-frames to locate and parametrise radio sources within these large quantities of expected survey data. High-performance computing solutions in conjunction with automated source-finding software is required to produce scientific outputs in practical time-frames.

Automated source extraction software packages, such as Selavy which will form part of the ASKAP software pipeline, are being developed for use in high-performance supercomputing solutions in order to process large survey data in practical time-frames. However, institution run processing may be limiting as it does not allow individual astronomers to perform their own searches with differing search parameters and prohibits alternative source extraction packages from being used. Whilst this limitation can be overcome by remote access to large clusters or distributed systems, there will exist circumstances where access to these devices is limited. An alternative to both these options is the use of high performance personal 'desktop' hardware. The available computational resources on a modern CPU have the potential to perform high performance source extraction.

The aim of this dissertation is to assess whether low-cost, high-performance parallel 'desktop' hardware can scale to the computational needs of source extraction procedures processing large ultra-deep and ultra-wide HI survey data. High performance computing on 'desktop' hardware requires the efficient use of its relatively few computational resources in conjunction with optimisations to the original source extraction algorithms. In addition, the memory allocation (the working set of memory) required for large data problems often exceeds the relatively small amount of main memory available on 'desktop' hardware and requires that a portion of the data is stored on disk (out-of-core computation) and pages into memory when necessary. This disk access is slower by orders of magnitude than main memory access and can throttle overall system performance. This potential performance bottleneck had to be overcome in order for computation of large data sets to be feasible.

The assessment of high performance source-finding on 'desktop' hardware was conducted by attempting to accelerate DUCHAMP's *à trous* wavelet reconstruction algorithm and extending the developed systems functionality to handle out-of-core computation efficiently. This multi-scale wavelet smoothing algorithm algorithm was selected for performance enhancement as it is a vital component of DUCHAMP and greatly improves the reliability and completeness of the DUCHAMP (and potentially other) source extraction packages by suppressing noise in the observational data. Additionally, this algorithm is both computationally heavy (contributing 65-95% of total DUCHAMP run-time, Chapter 3.3) and memory intensive (utilising 5 times the observational data set size in allocated memory).

Our task was to develop a high performance *à trous* reconstruction implementation through the optimisation of specific algorithms within the reconstruction procedure and the optimal use of CPU hardware resources, specifically SIMD instruction sets (Chapter 2.7.1) and multi-core parallelism. Algorithm optimisation included the introduction of Separable Filtering techniques (Chapter 2.7.3) to reduce the computational complexity of the convolution filtering procedure which takes up the majority of *à trous* reconstruction run-time (Chapter 3.3). The SIMD instruction sets (Chapter 2.7.1) of modern CPU processors exploited the inherently parallel components of the *à trous* reconstruction algorithm through vector instruction parallelism. This SIMD parallelism improves single core performance by concurrently executing a single instruction on multiple data to increase CPU throughput. We further extended the exploitation of the *à trous* algorithm's embarrassing parallelism by developing a multi-core CPU parallel implementation. Both the performance increase with increased number of computing cores utilised and the performance contributions of hyper-threading technology were investigated.

The prototype was additionally required to overcome the disk access bottleneck of the 'desktop' hardware memory hierarchy by efficiently managing the high memory use of the *à trous* wavelet reconstruction algorithm. This performance can generally be lessened by dividing the data into segments and processing each segment individually, reducing the overall amount of disk transfer. However, strong dependencies in the *à trous* algorithm between voxels in filtering operations (dependence grows with increased filter scale) and global definition of survey noise (Chapter 3.3) would significantly increase memory use and memory transfer to disk. Instead three popular external memory management (out-of-core computation) libraries, namely Mmap, Boost and Stxxl, were integrated into the prototype system design to mitigate the problem of slow out-of-core computation. These libraries allow extremely large working sets of memory (the memory allocated to a process), that exceed the size of physical memory and swap space, to be addressed by defining dedicated swap space separate to operating system-managed swap space Furthermore, these libraries have the potential to amortise the cost of disk access and increase I/O performance by utilising alternative paging schemes. System performance was to be optimised both for relatively small data sets, with memory use completely in-core, and larger data sets, with significant out-of-core memory allocation.

Although the focus of this work is high performance source extraction on 'desktop' hardware only, our developed system components are intended to be general enough for use in larger parallel computing solutions. The memory management solution was only intended for systems with insufficient fast-access memory and slow secondary storage.

Thorough performance profiling and validation testing was required in order to assess system improvements and to ensure that changes to the DUCHAMP implementations did not affect output. Testing was restricted to 'desktop' hardware to assess system performance on hardware with relatively small amounts of main memory.

In this chapter, we further discuss the goals of this dissertation and the prototype system designed to accomplish these objectives. We define constraints for prototype system development

to reduce the scope of this work. We describe our approach to system design and provide an overview of DUCHAMP functionality included in the prototype test system. We specify the design decisions and techniques used to optimise performance for each of the prototype system's main modules (encapsulated components). Furthermore, we describe the design of testing procedures for both validation and performance assessment.

4.2 Goals

The main aim of this work was to develop a high performance version of the DUCHAMP *à trous* wavelet reconstruction algorithm which would dramatically increase computational performance and handle large data volumes efficiently on 'desktop' hardware. This high performance prototype would evaluate whether redevelopment of the DUCHAMP source extraction package and the use of high-end 'desktop' hardware is sufficient to process large HI observational data in practical time-frames. System design included multiple approaches to achieve this goal. To assess the contribution of each of these design approaches or techniques, we defined four main research questions:

- Can we improve the efficiency of the *à trous* wavelet reconstruction for a single core CPU?
- Can Intel CPU SSE commands facilitate SIMD execution in this algorithm and further increase performance for the single-threaded case?
- Can we accelerate these processes by utilising parallel 'desktop' multi-core CPU hardware?
- Can slow disk access on 'desktop' hardware be mitigated with memory management to allow for efficient computation of large data sets?

The goals of this work would be reached successfully when the prototype system's performance is an order of magnitude greater than the original DUCHAMP implementation and system performance scales linearly with an increase in CPU cores. Additionally, the memory management solutions used to mitigate slow out-of-core computation would be considered successful when performance is comparable between in-core and out-of-core data sets.

4.3 Assumptions and Constraints

The scope of this work was potentially very large, and consequently development and assessment could have taken infeasibly long time-frames. In order to reduce the scope of this work and prototype design, we constrained the prototype design, as follows.

System design was restricted to a subset of the DUCHAMP software system, discussed below.This functionality subset included the *à trous* wavelet reconstruction algorithm, used for noise suppression within the DUCHAMP software, which was the only algorithm considered for further development and evaluation in this work. All other functionality was included either to correctly produce input/output for this algorithm or to assist in validation.

All system input parameters to the developed algorithms were kept consistent with the default DUCHAMP values when possible. All output generated by the prototype system was required to be identical (up to double precision) to the output generated by the DUCHAMP *à trous* wavelet reconstruction algorithm or proven to produce more accurate results.

The input data for system testing was restricted to radio spectrum observational data (astronomical in origin) recorded by radio telescopes or interferometers. These observations were stored as spectral data cubes in the FITS file format [42, 79, 98], a current industry standard which the DUCHAMP software takes as input. Input data set size was restricted to 10^9 voxels (3.7 GB in single precision). This size was sufficient to test out-of-core computation due to the high memory use of the *à trous* wavelet reconstruction algorithm. Testing with data sets larger than 10^9 voxels would have caused assessment (timing) to take infeasibly long time-frames.

The prototype system procedures were required to keep all observational data in the image space. Although fast and accurate noise techniques exist for noise mitigation in the UV-plane, we did not consider them in this work. High-performance improvements were restricted to the algorithms within the DUCHAMP source extraction package to ensure valid results and fair comparisons of performance.

The prototype system was required to operate on personal computing systems, running typical 'desktop' commodity hardware. This hardware system was expected to have a multi-core CPU with SSE2 functionality, have at least 2 GB of main memory and possess a relatively large amount of secondary storage ($>$ 100 GB). Smaller hardware systems were considered infeasible for any large scale computation and subsequent testing.

Parallel distributed computing models were not considered, as this work is restricted to stand-alone workstation systems only.

The developed system was required to run on the Unix-compatible operating system. This design decision was motivated by the prevalent use of Linux and Mac OS systems in the Astronomy research community.

Run-times for the various procedures within the *à trous* wavelet reconstruction algorithm are on the order of seconds to hours. Therefore, millisecond timing was assumed to be sufficient for accurate timing.

4.4 Approach and general design decisions

Development of the prototype system followed an iterative approach where functionality and consequently complexity were added in four distinct stages. This would reduce project risks associated with scoping issues, as a functional system for assessment and research would be obtained even if later stages were abandoned. Furthermore, this iterative development would allow for strict assessment of the validity and performance contributions introduced at each stage of development. The four stages comprised of:

1. A sequential CPU implementation of the considered DUCHAMP algorithms.
2. Optimisation for the single core CPU through algorithm redesign and better use of hardware resources.
3. Development of multithreaded CPU implementations to facilitate parallelism and reduce computation time.
4. Integration of memory management libraries in order to optimise prefetching of data from disk and enable efficient computation of large data sets.

Both the single core optimisation and memory management stages entailed the development of several competing algorithms in an attempt to maximise system performance. The performance of the competing algorithms were compared to determine the optimal solution at each stage.

However, it was difficult to predict if the use of the locally optimal procedures would result in the best overall system performance. Testing included the evaluation of all possible combinations of developed procedures to determine optimal overall system performance.

The design approach and testing procedures for each of four stages of development are detailed below.

Sequential CPU: The subset of DUCHAMP functionality (Chapter 4.5) considered for inclusion in the prototype system, was reproduced directly. Double precision copies of all DUCHAMP functionality computed at single precision only were developed. These functions were used to comparatively test system accuracy as divergence of single precision results in the latter development phases may represent an increase or decrease in system accuracy.

This development stage was validated against the DUCHAMP system to determine if the removal of functionality not considered for inclusion in the prototype system, or incorrect implementation, had affected the accuracy of the system. Initial profiling of DUCHAMP and our sequential CPU algorithms allowed for the identification of computationally-heavy components and possible bottlenecks in the system. Additionally, this assessment provided a baseline of performance against which subsequent improvements were compared.

Single core optimisations:

Many existing automated source extraction algorithms are developed with the principles of correctness and accurate solutions outweighing that of system performance. Consequently, algorithm computational performance is often sub-optimal and further development is required to make these algorithms computationally efficient and to make better use of available hardware resources. A single core optimised *à trous* wavelet reconstruction solution was developed in two parts: Separable Filtering techniques and Intel's SIMD instruction sets.

The *à trous* wavelet reconstructions' convolution procedure is computationally expensive and constitutes up to 95% of algorithm run-time. We attempted to significantly reduce the number of floating point operations (per voxel) in the convolution procedure by implementing Separable Filtering techniques (Chapter 2.7.3). This technique replaces the large 3D filter convolution pass with three separate convolution processes, with each convolution pass utilising a single 1D filter. Three competing Separable Filtering techniques were developed to determine optimal memory use for Separable Filtering convolution. The design specifics of the Separable Filtering variations are discussed in Chapter 4.6.

Intel's SIMD instruction set, specifically the packed functions of SSE2 (Chapter 2.7.1), were selected to exploit all parallel SIMD components of the *à trous* wavelet reconstruction algorithm. The SSE 2 instruction sets' packed floating point operations enables single instructions to be executed concurrently on multiple data to achieve parallelism performance increases on the single core CPU. The SSE2 instruction set was specifically selected because it supported both by AMD and Intel processors [34]. This eliminated potential hardware dependencies within the prototype system.

Multithreaded CPU implementations:

Multithreaded implementations of parallelisable DUCHAMP procedures were developed to fully utilise multi-core CPU architectures and improve system performance. A comprehensive list of the parallelisable components within the *à trous* wavelet reconstruction algorithm is discussed in Chapter 3. We specifically utilise multi-core CPU architectures to facilitate parallel execution as these devices are standard in modern 'desktop' hardware. Additionally, these devices are well-suited to parallelise the large amount of branching execution found in the *à trous* wavelet reconstruction algorithm.

We selected OpenMP [19, 18], a platform-independent industry standard, to provide multi-threading functionality. This avoided hardware dependencies for the developed system and provide predictable performance for differing hardware set-ups. Dependencies between and within the *à trous* reconstruction procedures were eliminated to increase the number of parallelisable components and maximise parallel performance. We aimed to develop a parallel solution which would scale performance linearly to the maximum number of cores available on any particular multi-core system.

All parallel solutions were evaluated to determine how well performance scaled with an increase in the numbers of physical cores (and logical cores if applicable) and higher thread counts used. These results were reported relative to each procedures optimised single-threaded implementation and their corresponding single-threaded DUCHAMP procedure to allow for the assessment of performance scaling with thread count and the total performance increases relative to DUCHAMP.

Memory management integration:

Three popular external memory management libraries (Chapter 2.7.2), namely Mmap, Boost and Stxxl, were integrated into the prototype system to mitigate the slow disk access associated with the out-of-core computation of large data sets. The inclusion of these libraries did not require redevelopment of the preceding performance-enhancing solutions in the previous developmental phases, discussed above. Performance of these memory management libraries was optimised by utilising any available fine-tuning parameters. Further memory management design details are discussed below (Chapter 4.5).

Evaluation consisted of pairing these memory management libraries with all preceding performance-enhancing solutions to determine which library (and associated paging scheme) is best suited for improving the out-of-core performance of the *à trous* wavelet reconstruction algorithm. Additionally, the performance improvements achieved through the fine-tuning of these libraries were isolated in order to investigate how performance changed with parameter selection and different paging schemes.

4.5 System Design

In this section, we describe the subset of DUCHAMP v1.1.13 functionality (Chapter 3) and all additional system components that were included in the prototype system. The subset of DUCHAMP functionality that we incorporated into the system explicitly excludes the latter source extraction stages (Chapter 3.1.4) of source amalgamation, rejection and parametrisation as they constitute a relatively small portion of total run-time for relatively sparse observations. Additionally, all optional lightweight preprocessing functionality, that further improves the effectiveness of source extraction or increases computation performance, was excluded from the prototype system. These optional operations are excluded as their potential improvements are highly variable and depend on the specifics (noise properties and shape) of a particular observation. This includes a data set trimming procedure which reduces memory use when data sets are only partially occupied, and baseline removal which improves results if the data set contains large scale structures such as continuum ripples.

System components were restructured into a modular framework (Figure 4.1) to better encapsulate functionality. The phases indicated in Figure 4.1 correspond to the four stages of development, discussed above, and indicate the level of development each system component receives. Although DUCHAMP provides several procedural options within each system compo-

nent, we only incorporate the most computational expensive procedures which either provide the best reliability and completeness metrics, or fully automate a portion of system functionality. Details pertaining to the actual implementation of these components are discussed in Chapter 5.

Memory management control procedures were structured into a separate system component as their implementation was independent of source extraction algorithm specifics. This is in contrast to the single core optimisation and multi-core implementations which were incorporated within the noise suppressing *à trous* wavelet reconstruction algorithm.

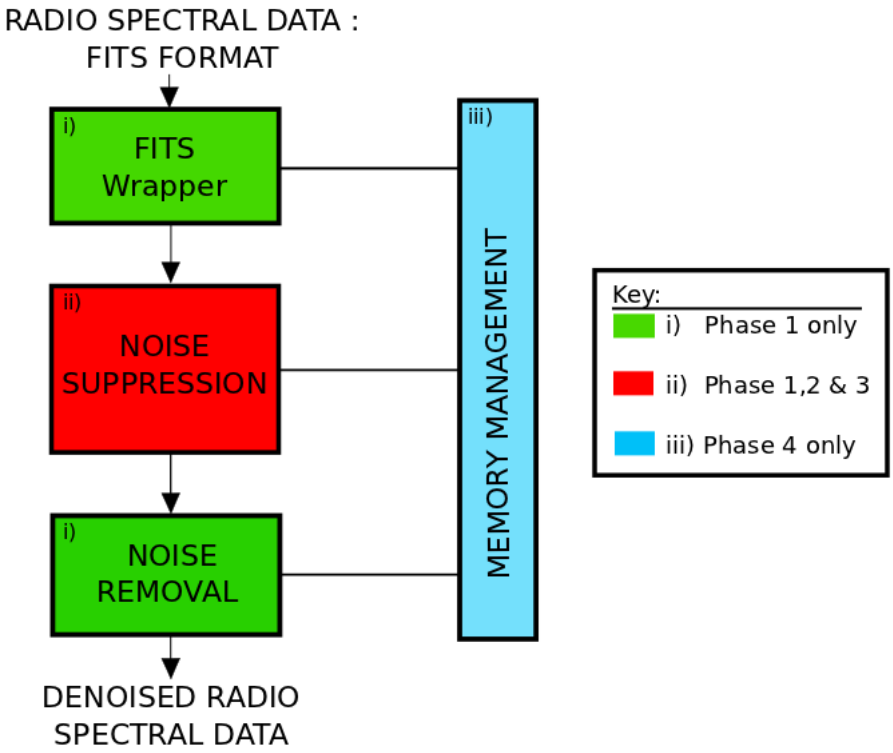


Figure 4.1: Overview of the components within the prototype system.

The prototype system modules and the DUCHAMP functionality each would implement was defined, as follows:

FITS Wrapper: The FITS file reader abstracts the use of the CFITSIO library, a platform independent C library for performing I/O with the Flexible Image Transfer System (FITS) data format. Although a C++ interface exists (CCFits), it was considered an unnecessary further abstraction from the intuitive ANSI-C routines.

The main task of this module was to efficiently access the raw FITS format data and convert it to real image voxel values for use in the prototype system. Additional metadata stored in the FITS format's headers, which describes various properties of the observation, is accessed directly using CFITSIO routines. This is in contrast to using the World Coordinate System (WCS) library used by the DUCHAMP software package to abstract the access and storing of metadata. The majority of the variables defined in the WCS are not required for our system and adding the WCSLIB package would unnecessarily increase complexity. Default values in the case of missing information are kept consistent with the DUCHAMP implementation.

The CFITSIO library was not considered for further development as the loading of data comprised a relatively small portion of DUCHAMP's total run-time.

Noise Suppression:

The DUCHAMP package implements several noise suppression procedures, namely 2D Gaussian kernel spatial filtering, spectral smoothing with a Hanning filter [94] and the *à trous* wavelet reconstruction algorithm. We selected the *à trous* wavelet reconstruction algorithm for development, as it is the most effective, in terms of noise suppression, out of these procedures. Additionally, this algorithm is computationally intensive and memory expensive making it an ideal benchmark for high-performance astronomical computing solutions and techniques for the efficient handling of large data volumes on 'desktop' hardware. The two alternative de-noising algorithms were excluded due to their simplicity.

Whilst the DUCHAMP package provides wavelet reconstruction over 1 - 3 dimensions, our system would only implement the 3D filter convolution variant in order to reduce the amount of development required. The 3D procedure is the most suitable for suppressing noise in HI radio spectral data input as the sources of interest should extend in both spacial and spectral dimensions. Additionally, this procedure is both the most computationally expensive wavelet reconstruction variant and has the greatest potential for increased performance.

Noise Removal:

The noise removal module consisted of the False Discovery Rate (FDR) algorithm for determining the threshold used to separate valid sources from background noise. The two alternate algorithms, the flux value and Signal to Noise ratio threshold (Chapter 3.1.3), were not selected due to their simplicity and dependence on user specified input. The FDR thresholding algorithm statistically defines a threshold that controls the number of false positives when extracting sources from noisy observational data. Although this algorithm can be considered computationally expensive we did not consider it for performance enhancement due to time restrictions. Instead this algorithm was to be used to further validate the accuracy of the *à trous* wavelet reconstruction. The valid source data sets produced after thresholding would be compared against their DUCHAMP counterparts. This was to ensure that deviations in output resulting from floating point arithmetic inaccuracies are identified and eliminated.

Memory management:

This component consisted of integrating the three memory management libraries; Boost, Stxxl and Mmap into the prototype system. We would compare these libraries on their ability to mitigate the slow disk access bottleneck that arises from out-of-core computation. We utilised the simplest data structure provided by each of the three integrated libraries in order to ensure comparable assessment, discussed later in Chapter 5.4. These basic data structures allowed for simpler integration with *à trous* wavelet algorithm which exclusively uses arrays to store all data. The set-up and control procedures for these libraries are encapsulated into a separate module to minimise the modification of the *à trous* wavelet reconstructions source code and to simplify changing the memory management scheme used. Multithreaded memory management functionality was not included in the prototype system as only the Boost and Stxxl libraries possessed this functionality.

4.6 Separable Filtering

Convolution is the most computationally expensive component of the *à trous* algorithm. In this procedure, data observational data is convolved with a scalable 3D filter to extract features of a specific size. The number of floating point operations (per voxel) in this procedure can be significantly reduced by implementing Separable Filtering.

To simplify convolution with Separable Filtering, the 3D filters within DUCHAMP must be able to be expressed as the convolution of two or more simpler filters (separability) (Chapter 2.7.3). All DUCHAMP filters with two or more dimensions meet this criterion of separability, as higher order filters are generated at run-time by convolving two or more 1D filters. Separable Filtering decomposes the 3D filter convolution found in DUCHAMP into a three pass convolution algorithm which uses separated 1D filters orientated in the row, column, and spectral directions respectively.

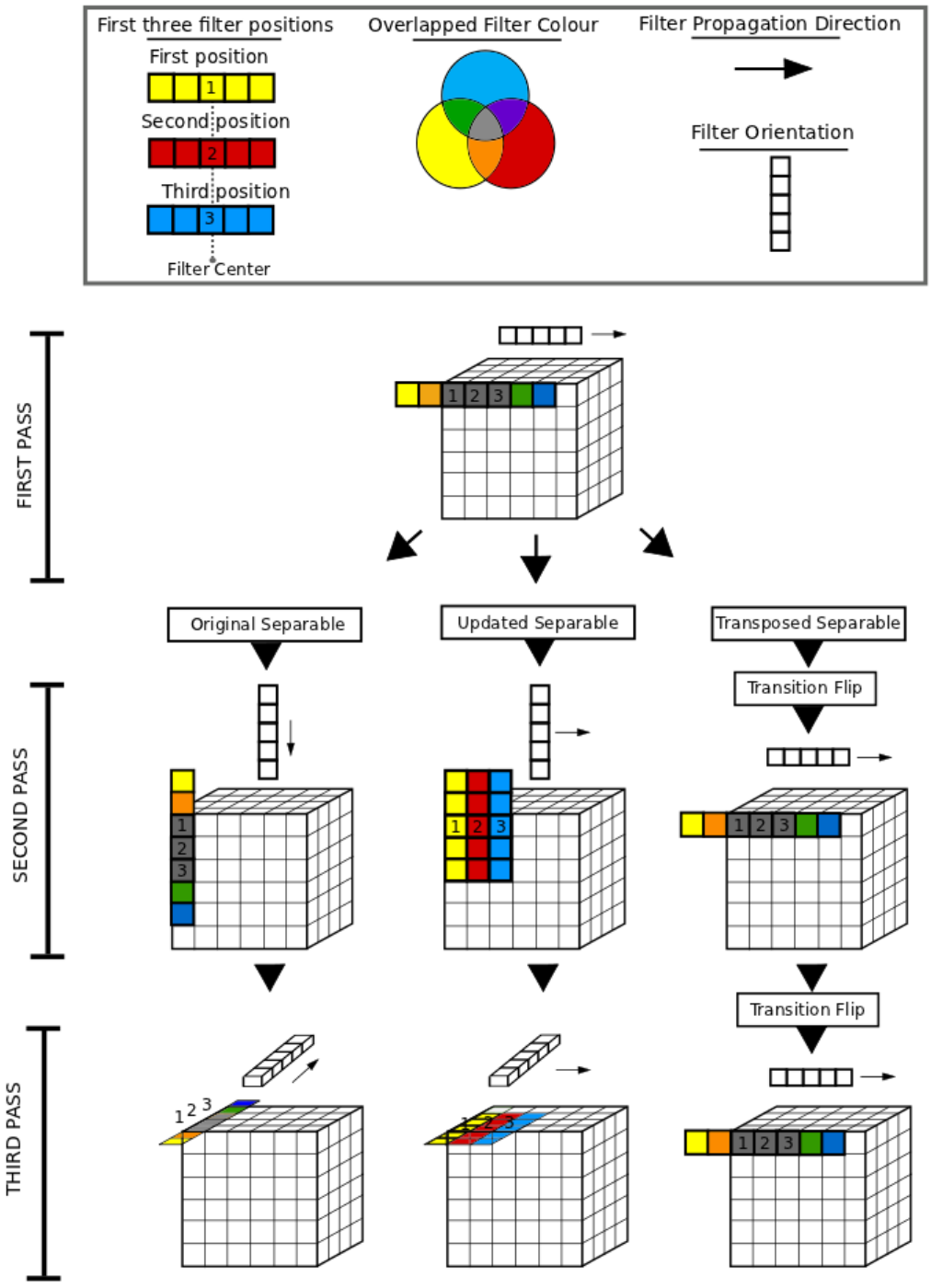


Figure 4.2: The Separable Filtering algorithm variants. Shown are the filter orientations and direction of propagation for each convolution for all three separable filter algorithm variants. The direction of propagation is shown by the first three filter placements labelled first (yellow), second(red) and third (blue). The number of the filter placement is used to indicate the central filter position.

Separable Filtering should theoretically increase performance by a factor of $\frac{X^3}{3X}$, where X is the size of any particular filter dimension (all DUCHAMP filters have equal dimensions). The two filters implemented in DUCHAMP, namely the 3D Haar of size 3 and B3-Spline filter of size 5, result in a theoretical speed-up of $\frac{3^3}{3(3)} = 3$ and $\frac{5^3}{3(5)} = 8.\dot{3}$ respectively. This reduction in operations holds for all filter scales utilised in the multi-scale *à trous* reconstruction as the number of filter elements (and convolution operations) is constant. The only disadvantage to this algorithm is the increased memory required to store the intermediate output (filter responses) between each of the separable filter passes.

The filter responses generated for each of the separated filtering passes are independent of the direction at which their respective 1D filter is propagated over the data. This allows for the development of multiple Separable Filtering convolution variants which produce distinct memory access patterns as a result of differing filter propagation directions. Three Separable Filter variants were designed (Fig 4.2), which vary only with respect to their memory access patterns, in order to determine the propagation directions that would result in optimal memory use. All three variants share the same first convolution pass method whereby a row-orientated filter moves linearly through memory and should result in optimal memory use. However, the last two passes of each algorithm differ dramatically, as discussed below. We have assigned working titles to these Separable Filtering implementations to intuitively distinguish them, namely, Original Separable, Transposed Separable and Updated Separable Filtering.

Original Separable Filtering (column 1) is the most intuitive approach and mimics the manual human approach to convolving data with a filter. This algorithm propagates each of the filters over the data in the same direction as the filters orientation. The row, column and spectra-aligned filters are moved over the data in the row (first pass), column (second pass) and spectral direction (third pass) respectively. This memory access pattern is very inefficient and causes poor performance for the Original Separable algorithm. Consequently this algorithm was used as a lower limit benchmark against which the competing Separable Filtering procedures were compared.

The Updated Separable algorithm (column 2) propagates all three filters (with their respective row, column and spectral orientations) linearly through memory for each of the separate filter passes. This algorithm was expected to have efficient cache use (using the entirety of paged in cache lines) despite the column- and spectral-aligned filters of the last two filter passes accessing memory in strides.

For the the Transpose Separable algorithm (column 3) the data cube is transposed after each pass. This was to allow the subsequent separable convolution passes to utilise a row-aligned filter and propagate this filter linearly through memory. Therefore after the first filter pass, the data in the column direction is aligned in the row direction and the corresponding column-orientated filter is replaced with a row-aligned filter. This is followed by another transpose operation which maps the spectra-aligned data to a row alignment for the final convolution pass. Although this would optimise the convolution read operations, there is a trade-off as the convolution writes which transpose the data are strided and consequently inefficient.

4.7 Evaluation and validation

The main objective of this work was to achieve large performance increases for the DUCHAMP *à trous* wavelet reconstruction process. Precise timing was required in order to accurately evaluate the system and determine the exact speed-ups attained through performance-enhancing development. Additionally, the *à trous* wavelet reconstruction algorithm had to maintain valid

and accurate output throughout the development process in order to ensure its use as a scientific tool. The evaluation and validation procedures used for the sequential CPU, optimal single core, multithreaded CPU and memory management development phases are discussed above. In this section, we discuss the additional design concerns for the testing procedures used to evaluate and validate our high performance prototype.

Run-times for the various procedures in the *à trous* wavelet reconstruction algorithm are of the order of seconds to hours. Thus millisecond timing is assumed to be sufficient for accurate timing.

Changes to the *à trous* algorithm was likely to produce rounding error unless all operations and effective operational order are kept consistent. This is caused by the finite accuracy of floating point arithmetic implemented on computing devices. Additionally, this error, which is generally small, was expected to compound during the multiple iterations of *à trous* reconstruction and could result in substantial error in the final output. The output of the direct functionality port which comprises the first stage of development, discussed above, was to be validated against DUCHAMP to produce the exact same output from all implemented procedures in order to ensure system accuracy. Operational order in later stages of development was to be kept consistent to reduce the likelihood of FP error. The test system was validated for both single and double precision calculations. System output was considered valid if it was either identical (up to the precision level used) to the output generated by the DUCHAMP *à trous* wavelet reconstruction algorithm or proven to produce more accurate results.

System accuracy and performance improvements were tested with a comprehensive list of data sets. This list should adequately cover a range of data set sizes, starting from small completely in-core data sets to data sets that are computed mostly out-of-core. Real observational data was used to cover the full range of data set sizes when possible, with simulated data sets covering any significant gaps in this range. These range gaps could be filled by merging several data cubes into a single cube. However, simulated data is simpler to produce and its use does not change the measured run-times for the *à trous* wavelet reconstruction procedures as they are dependant on data set size. Simulated data could not be used for accuracy validation as the randomly generated flux values would produce a completely random distribution of low and high values which would result in an unrealistic amount of rounding error.

Test hardware, to conform to the assumptions detailed in Chapter 4.3, would include a high end multi-core CPU with moderate amounts of main memory (> 2 GB) and possess a relatively large amount of secondary storage (> 100 GB) to assess the performance improvements of parallelism and the mitigation of slow out-of-core computation.

4.8 Summary

In this chapter, the design considerations for the development of an optimal implementation of DUCHAMP's noise suppression algorithm (*à trous* wavelet reconstruction) on 'desktop' hardware are discussed. The most pertinent among these, are the considerations for increases in computational performance through algorithm redesign and optimal use of multi-core CPU hardware, and slow out-of-core computation mitigation with memory management libraries. It was considered vital that we evaluate all combinations of the designed performance-enhancing procedures to determine optimal overall performance of this algorithm. Additionally, validation was a key concern to ensure the use of this software as a scientific tool. The implementation of this system and the results of the succeeding evaluation are discussed in Chapters 5 and 6 respectively.

Chapter 5

Implementation

In this chapter we discuss the implementation and testing of the software system design detailed in Chapter 4. The purpose of the system was to assess whether our improvements to the DUCHAMP implementation of the *à trous* wavelet reconstruction algorithm are sufficient to process large HI survey data in practical time-frames using 'desktop' hardware only. Additionally, we discuss the implementation of supporting functionality required to handle the FITS input data and aid in testing. The testing implementation included a standard implementation of DUCHAMP v1.1.13 to provide comparative evaluation baselines for both performance increases and accuracy. The software dependencies of both systems and test hardware specifics are given.

The development of this system was carried out in four implementation phases: sequential CPU, single core optimisation, multithreaded CPU and memory management. The design and approach to the development of each phase is outlined in Chapter 4.4 whilst the notable implementation specifics of each are covered in this chapter. The sequential CPU phase covers the direct reproduction of required DUCHAMP functionality within our system. Additionally, this phase lists the software dependencies (and references to installation instructions) required for both the standard implementation of DUCHAMP and our improvements to the *à trous* wavelet reconstruction algorithm, discussed in later implementation phases.

Single core optimisation refers to the optimal implementation of the three Separable filtering techniques (discussed in Chapter 4.6) and Intel's SIMD Instruction sets (discussed in Chapter 4.4). The Separable filtering techniques were used to improve the 3D convolution procedure within *à trous* wavelet reconstruction and identify the most optimal memory access pattern in this improved convolution. Intel's SIMD Instruction sets, specifically SSE2 packed instruction commands, were used to implement vector instruction parallelism on a single CPU core to increase performance. The multithreaded CPU implementation phase discusses the port of our sequential implementation to a parallel multi-core CPU implementation and the multithreading API used. The memory management implementation phase refers to the implementation and fine-tuning of three popular memory management libraries, namely Mmap, Boost and Stxxl, within our system which were evaluated on their ability to facilitate optimal out-of-core computation. All assumptions and difficulties encountered during implementation are discussed.

Although 'desktop' computing is emphasised, our implemented high performance system components are general enough to be accommodated into larger parallel computing hardware solutions. Memory management was intended for systems with insufficient fast-access memory and slow secondary storage. This chapter concludes with the implementation specifics of evaluation procedures and test data used to validate system accuracy and precisely measure performance

improvements of our improved *à trous* wavelet reconstruction algorithm.

5.1 Sequential CPU implementation

The focus of this research was the performance improvement of the *à trous* wavelet reconstruction algorithm within DUCHAMP. We implemented a sequential CPU test bed system which reproduced the DUCHAMP subset outlined in Chapter 4.7. This implementation further increased the encapsulation present in DUCHAMP's class and object structure to allow for simpler development and testing. Data structure management (creation, loading, deletion) was separated into a memory management module. This allowed for later implementation of the memory management data structures while avoiding changes to the rest of the test system.

Sequential CPU implementation allowed for an initial validation of the test system to ensure the algorithms were correctly replicated and that the removal of the majority of the DUCHAMP functionality did not invalidated system correctness. Initial profiling confirmed no significant run-time differences exist between the original DUCHAMP software and our system. This equivalence provided a valid baseline of comparison for system improvements.

In the remainder of this section, we discuss the specifics of our sequential CPU implementation. We discuss third party software required to run/test both DUCHAMP and our system. The system architecture (Chapter 4.5) is discussed, focussing on the directly ported DUCHAMP implementations (as a reference for later development) and our changes to these algorithms. The specifics of the Testing and Timing components are covered separately in Chapter 5.5.

5.1.1 Software Dependencies

Table 5.1 lists and briefly describes the required third party software for the execution of both DUCHAMP and our implemented test system. Installation procedures (beyond default) for CFITSIO and all three memory management libraries are covered below.

Name	Description	Used By
g++ (v4.4.5)	GNU C++ compiler (Optimisation O2 used for both systems)	DUCHAMP Test System
CFITSIO (v3.3.5)	Handles IO interactions with the FITS file format.	DUCHAMP Test System
WCSLIB	Handles World Coordinate System metadata. Maps voxel position to true sky coordinates.	DUCHAMP
PGPLOT	Graphics subroutine library used to display the graphical output of DUCHAMP.	DUCHAMP
OpenMP (v4.4.5)	Platform independent CPU API used to implement multithreading on multi-core CPUs.	Test System
Mmap	Memory mapping support available within Linux operating systems.	Test System
Boost Interprocess	Memory mapping support within Boost library. Used primarily for interprocess communication.	Test System
Stxxl (v1.3.1)	Memory mapping support with RAID capability.	Test System
DS9	Visualisation tool for FITS data.	Test System

Table 5.1: Test System and Original DUCHAMP Software requirements.

5.1.2 System Architecture

In this section we discuss the final system architecture (Fig 5.1) and its sequential CPU implementation. Further system overviews are not required as subsequent development was constrained within the Noise Suppression and Memory management components.

A brief overview of each implemented system component is given followed by a more detailed discussion of the highlighted system components in Fig 5.1. This includes the implementation specifics of both the *à trous* wavelet reconstruction (Chapter 3.3) and FDR thresholding (Chapter 3.2) algorithms.

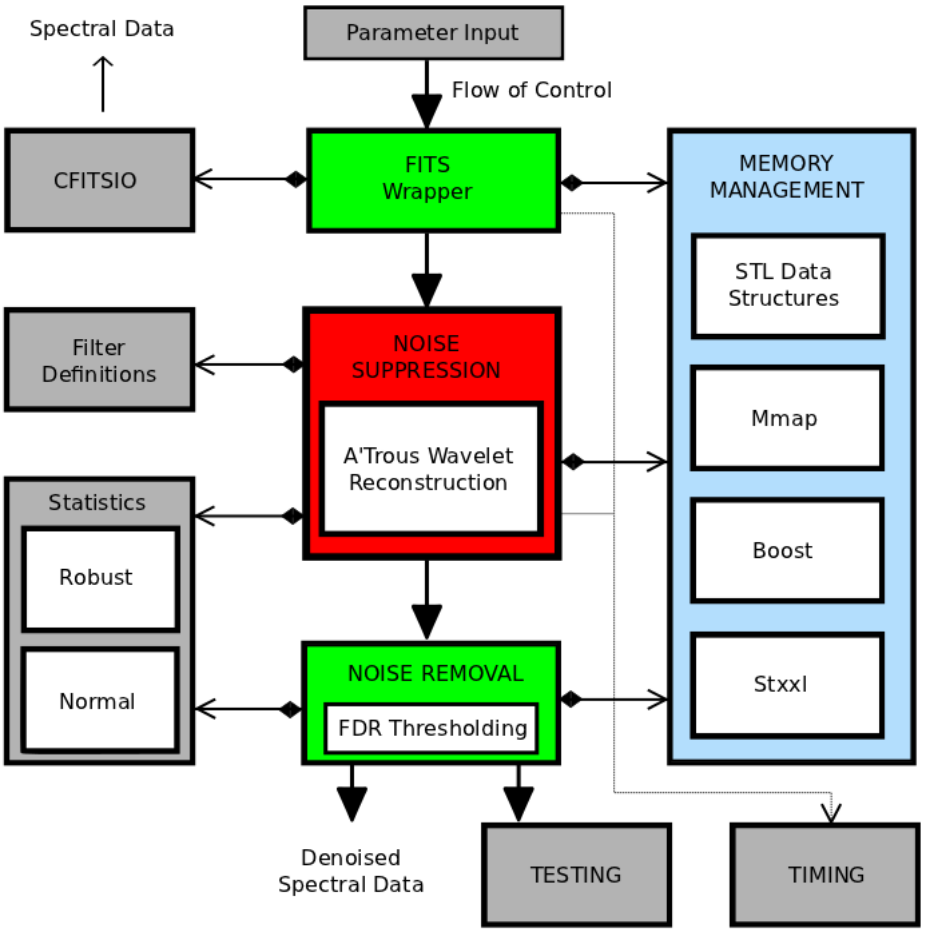


Figure 5.1: Overview of the implemented prototype system components.

The implemented system components (Fig 5.1) are as follows:

Parameter Input: Loads parameters used to define the statistical technique, filter and memory management component used during execution. Additionally, this component can define a data subset for use in the remainder of system execution.

Filter Definitions: Stores the definition of the available filters in DUCHAMP (B-Spline, Haar, Triangle) and the correlation factors for these filters at different scales. B3-Spline used by default.

CFITSIO: A platform independent C library for performing I/O with the Flexible Image Transfer System (FITS) data format.

FITS Wrapper: Abstracts the use of the CFITSIO library. Loads only the data subset requested in "Parameter Input". Loads the entire data set by default. Retrieves metadata

variables stored in the FITS header.

Noise Suppression: Implements the *à trous* wavelet reconstruction algorithm (Chapter 3.3). This algorithm uses a multiscale wavelet decomposition and significance thresholding to smooth noise in the input data. This preprocessing step improves the completeness and reliability of the succeeding source finding algorithm. Only 3D reconstruction is implemented as it is more suitable for processing 3D radio observational data.

Noise Removal: Implements the False Discovery Rate algorithm (Chapter 3.2) which defines a threshold that separates background noise from likely source signals. This threshold is statically determined to control false discoveries (inherently caused by noise) whilst detecting the majority of true sources. This component is used to further process the output of *à trous* wavelet reconstruction algorithm to enable further validity testing.

Memory management: Handles the management of both STL data structures and other data structures used by the Mmap, Boost and Stxxl memory management libraries. This includes the setting of fine-tuning parameters within these libraries which are used to optimise memory management. Only STL data structures are implemented in the Sequential CPU implementation phase. Place holders were created for the later implementation of memory management (Chapter 5.4).

Statistics: Implements both the Normal and Robust statistical estimators used in DUCHAMP. Normal statistics use the mean and standard variation to define the middle and spread of a data set. Robust statistics define the middle and spread of the data with the median and Median Absolute Deviation from the Median (MADFM) respectively. Although robust statistics are more computationally expensive, they are less sensitive to outliers.

Testing: Implements validity checks which verify whether the output produced by the test system is identical to or more accurate than the output produced by DUCHAMP.

Timing: Generates accurate (to the millisecond) timing data for each system component.

5.1.2.1 FITS Wrapper and CFITSIO

The CFITSIO library was compiled with the following flags to allow for the creation and reading of large FITS files: -D_FILE_OFFSET_BITS=64 and -D_LARGEFILE_SOURCE [77]. Large files use 64-bit addressing and required all 32-bit CFITSIO calls to be replaced with their "type long" equivalent. Maximum file size is restricted by CFITSIO to 2^{31} FITS data records (2880-byte)$\approx$ 6TB [77] which is larger than the data set sizes expected from the WALLABY and DINGO surveys (782 GB), and orders of magnitude larger than those expected from the LADUMA ultra-deep survey.

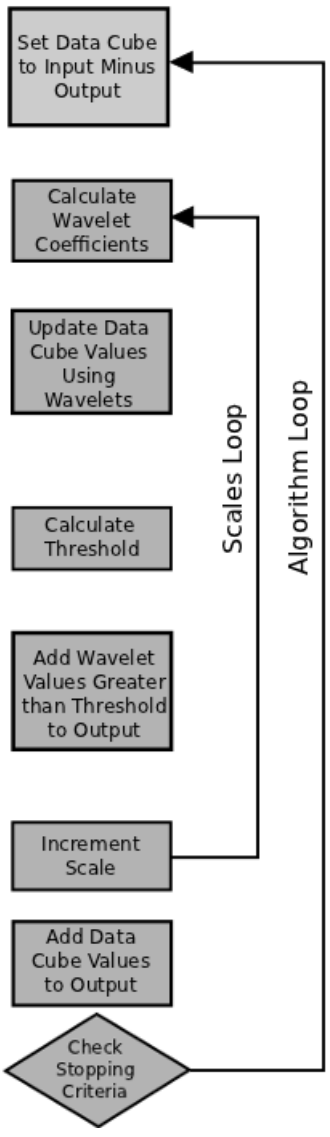
Performance improvements to I/O operations accessing FITS data were only considered to reduce computation time during testing and were not evaluated separately. CFITSIO file access was optimised by reading/writing in multiples (greater than 3) of FITS containers units (2880 bytes) to bypass CFITSIO's internal buffering [77]. Data is read once and passed to the memory management component for storage in the appropriate memory management data structure. Metadata access was simplified by using CFITSIO routines to directly access the metadata subset relevant to our system instead of using WCSLIB, a third party library used in DUCHAMP to access and store metadata. All default variable values are kept consistent with their DUCHAMP counterparts.

5.1.2.2 *à trous* wavelet reconstruction

A truncated version of the DUCHAMP *à trous* wavelet reconstruction algorithm (Chapter 3.3) was implemented. This version did not alter run-time of reconstruction under the assumption that data sets are fully populated and contain no padding information. Corrupt or invalid voxels are still expected. For ultra-wide and ultra-deep surveys, it is expected that the majority (if not all) of data sets are fully populated as blind surveys consider the entire survey space. This is in contrast to focused observations which only consider a few celestial objects. The pseudo code representation of our *à trous* wavelet reconstruction algorithm is shown in Algorithm 5.1.

Algorithm 5.1. A'Trous wavelet reconstruction with 3D convolution.

```
1: Create data structures and generate 3D filter from specified 1D filter.
2: OriginalSpread ← CalculateSpread(input[])
3: filter[] ← defineFilter(filterChoice)
4: do
5:     spacing ← 1
6:     oldSigma ← newSigma
7:     for n = 0 → TotalVoxels do
8:         coefficients[N]= input[N]-output[N]
9:     end for
10:
11:    for scale = 0 → numberOfScales do
12:        3DConvolution(wavelet[], coefficients[], GVM[], spacing)
13:
14:        for N = 0 → TotalVoxels do
15:            coefficients[N] ← coefficients[N] − wavelet[N]
16:        end for
17:
18:        if scale ≥ minScale then
19:            middle ←calculateMiddle(wavelets[])
20:            threshold ←calculateThreshold(middle, originalSigma)
21:
22:            for N = 0 → TotalVoxels do
23:                if wavelet > threshold then
24:                    output[N] ← output[N] + wavelet[N]
25:                end if
26:            end for
27:
28:        end if
29:        spacing ← spacing ∗ 2
30:    end for
31:    for n = 0 → TotalVoxels do
32:        output[N] ← output[N] + coefficients[N]
33:    end for
34:
35:    newSpread ← CalculateResidualSpread(input[], output[])
36: while CheckContinuationCriteria()
```



We only implemented 3D reconstruction as this functionality was most suitable for suppressing noise in HI radio spectral data as the sources of interest extend in both spacial and spectral

dimensions. Additionally, this procedure is significantly more computationally expensive than 1D and 2D reconstruction and had greater need of performance enhancement to process noisy data in practical time-frames.

We removed the Blank Voxel Trimming (flagging) functionality from the DUCHAMP implementation of *à trous* reconstruction. Removal of padding information can result in a row or column extent of a single voxel which can cause edge case handling (discussed below) procedures to infinitely loop. Furthermore, under the assumption of fully populated data sets, trimming serves no purpose as there is no existing padding data to be removed. Removal of trimming eliminates the need to store edge limits for each row, column and spectrum in the data set.

Declaration of data structures and memory allocation within the *à trous* algorithm was abstracted through calls to the memory management API (Chapter 4.5). This allowed the memory management library used to be easily changed and tested. Only STL data structures libraries were used in the sequential CPU implementation.

Algorithm 5.1 line 8 sets the input for particular iterations of the reconstruction algorithm. This input (stored in the coefficients array) is the difference between the original input data and the current state of the reconstructed output. For the first iteration, the coefficients array is equal to the original input. However, in subsequent iterations, the coefficients array stores residual data which may hold additional structure.

Algorithm 5.1 line 12 calls the function shown in Algorithm 5.2. This function accepts the wavelet and coefficients arrays used in the convolution calculation and the GoodVoxelCheck (GVM) array to flag and ignore corrupt voxels. This function (Algorithm 5.2) calculates filter responses for a single convolution pass (lines 4-6). The inner loops (lines 12, 19, 26) step through filter elements to calculate filter responses for each voxel. The loop counter (line 32) used to index the filter values in the DUCHAMP implementation is retained. Calculating filter positions from the inner loop indexes would remove loop dependencies within the inner loops. However, this was unnecessary for the level of parallelism implemented in Chapter 5.3. For every element in the filter, checks are required (lines 14, 21, 28) to determine whether each of the three coordinates are valid positions within the data set (in-bounds). If out-of-bounds, one or more of these coordinates are reflected across their respective edge of the data set to calculate a filter response over a valid position. Once all checks have been completed, the filter response for one filter element and its corresponding valid coefficient array position is calculated (line 35). Finally, each filter response is subtracted (line 36) from the same position in the wavelet array (originally coefficients array) to calculate the wavelet coefficients.

After the calculation of the convolution operation, the remainder of Algorithm 5.1 is as follows.

Line 15: The coefficient array is not updated within the convolution function (line 12) and is updated separately by subtracting the wavelet coefficients (even non significant responses). This differs from the description of the *à trous* algorithm in Chapter 3.3 but is necessitated to reduce memory use.

Line 24: Adds all significant structure to the reconstructed array (output array).

Line 29: The scale is increased and the convolution and thresholding procedure repeated.

After convolving and thresholding for all scales within a single iteration, the remaining coefficient array (line 32) is added to the reconstruction array as a DC offset. This is to account for the zero mean of the wavelet coefficients added in line 24.

DUCHAMP's modified statistics calls are used to calculate the noise properties of the residual data (line 35) which eliminates the need for an extra data structure. For robust modified

Algorithm 5.2. : 3D Discrete Convolution

```
1: spatialSize ← xLimit * yLimit
2: position ← 0
3:
4: for z = 0 → zLimit do
5:     for y = 0 → yLimit do
6:         for x = 0 → xLimit do
7:             filterPos ← 0
8:
9:             if !GoodVoxelCheck(x, y, z, GoodVoxelMask) then
10:                Wavelets[x, y, z] ← 0
11:            else
12:                for zoffset = −filterHalfWidth → filterHalfWidth do
13:                    CPZ ←Calculate convolve position(z, zoffset, spacing)
14:                    if OutofBounds(CPZ) then
15:                        CPZ ← Reflection(CPZ)
16:                    end if
17:                    CPZ ← CPZ * spatialSize
18:
19:                    for yoffset = −filterHalfWidth → filterHalfWidth do
20:                        CPY ←Calculate convolve position(y, yoffset, spacing)
21:                        if OutofBounds(CPY) then
22:                            CPY ← Reflection(CPX)
23:                        end if
24:                        CPY ← CPY * yDimensionSize
25:
26:                        for xoffset = −filterHalfWidth → filterHalfWidth do
27:                            CPX ←Calculate convolve position(x, xoffset, spacing)
28:                            if OutofBounds(CPX) then
29:                                CPX ← Reflection(CPX)
30:                            end if
31:                            CP ← CPZ + CPY + CPX
32:                            filterPos ← filterPos + 1
33:
34:                            if GoodVoxelCheck(CP, GoodVoxelMask) then
35:                                filterResponse ← (Coefficients[CP] * filter[filterPos])
36:                                Wavelets[x, y, z] ← Wavelets[x, y, z] − filterResponse
37:                            end if
38:                        end for
39:                    end for
40:                end for
41:            end if
42:
43:        end for
44:    end for
45: end for
```

statistics, the temporary data structure copy, used to preserve the data during sorting, is populated directly with the difference between the input and output arrays. For normal modified statistics, the difference between respective voxels in the input and output arrays are calculated

during summation in both the mean and standard deviation calculations.

The CheckContinuationCriteria (Algorithm 5.1 line: 36) checks the stopping condition for *à trous* wavelet reconstruction iterations with:
$((double)fabs(\text{oldSigma} - \text{newSigma})/\text{newSigma} > \text{reconstruction tolerance})$
This function determines whether reduction in noise spread between two algorithm iterations is small, indicating that the majority of the source structure has been removed from the input data. If the change in noise spread is larger than the reconstruction tolerance the algorithm repeats. It was necessary to cast the absolute value maths calls to double precision to achieve the same output (accurate to double precision) as the DUCHAMP implementation.

5.1.2.3 False Discovery Rate threshold algorithm

The False Discovery Rate algorithm implemented in DUCHAMP was directly reproduced in our test system (Algorithm 5.3).

Algorithm 5.3. : False Discovery Rate Threshold algorithm.

1: **for** $x = 0 \rightarrow xSize$ **do**
2: **for** $y = 0 \rightarrow ySize$ **do**
3: **for** $z = 0 \rightarrow zSize$ **do**
4: $position \leftarrow (z * xSize * ySize) + (y * xSize) + x$
5: $orderedP[count++] \leftarrow$getPValue$(array[position], middle, spread)$
6: **end for**
7: **end for**
8: **end for**
9: Stable_sort$(orderedP)$
10: $N \leftarrow BeamSize * NumberCorrelatedChannels$
11: **for** $psfCounter = 0 \rightarrow N$ **do**
12: $cN \leftarrow cN + 1/psfCounter$
13: **end for**
14:
15: **for** $loopCounter = 0 \rightarrow N$ **do**
16: **if** $orderedP[loopCounter] < \frac{alpha*(loopCounter+1)}{cN*count}$ **then**
17: $max \leftarrow loopCounter$
18: **end if**
19: **end for**
20: $pValueThreshold \leftarrow orderedP[max]$
21:
22: **do**
23: $zStat \leftarrow zStat + deltaZ$;
24: $current = 0.5*$error_function$(zStat/\sqrt{2}) - pValThreshold$;
25:
26: **if** $(initial * current) < 0$ **then**
27: $zStat \leftarrow zStat - deltaZ$
28: $deltaZ \leftarrow \frac{deltaZ}{2}$
29: **end if**
30: **while** $deltaZ > tolerance$
31: $threshold \leftarrow (zStat * spread) + middle$

This algorithm was used to determine a statistical threshold which constrains the percentage

of false positives during the thresholding procedure which "searches" for likely source voxels. Performance improvement and subsequent performance evaluation was not carried out as this algorithm was only used to produce a noiseless data set to further validate test system correctness.

Line 1-8: Converts all voxel intensity values to their corresponding p-values.

Line 9: Sort the p-values in ascending order.

Line 10-13: Calculate the correlation normalisation constant cN.

Line 15-19: Calculate $d = max\{j : Pj < \frac{j\alpha}{C_N N}\}$.

Line 20: The p-value at position d is the threshold at which we reject the hypothesis that a voxel belongs to the background and is therefore a source voxel.

Line 26-32: The p-value at position d is converted into an intensity threshold.

5.2 Single Core Optimisations

The single core optimisation implementation phase was concerned with improving the performance of the sequential *à trous* wavelet reconstruction algorithm. This was accomplished through improvements to algorithm efficiency and exploiting vector instruction parallelism in order to optimise performance for a single CPU core. We improved algorithm efficiency by implementing separable filtering to reduce the computational complexity of the 3D convolution process. Utilisation of Intel's streaming SIMD extensions (SSE) commands facilitated parallel performance increases with the concurrent execution of identical floating point operations on a single CPU core. In this section we discuss the implementation specifics of these two approaches to single core optimisation.

5.2.1 Separable filtering

Separable filtering (Chapter 2.7.3) decomposes the DUCHAMP 3D convolution into a three pass convolution procedure. Each convolution uses a 1D filter orientated in the row, column, and spectral directions respectively. Each have size X, where X is the size of any 3D filter dimension (all DUCHAMP filters are isomorphic). This reduces the number of floating point convolution operations required per voxel from X^3 to $3X$ and theoretically increases performance by $\frac{5^3}{3(5)} = 8.\dot{3}$ for the largest filter in DUCHAMP, the B3-Spline.

Implementation of separable filtering does result in two negative attributes. An extra intermediate array, equal in size to the input data set, is required to store intermediate values between the three separate filter passes. This extra memory use decreases the size of the data set computed completely in-core. Furthermore, separable filtering's multi-pass procedure introduces non-mitigatable differences in output precision relative to 3D convolution. The causes and extent of this floating point arithmetic difference are discussed in Chapter 6.1.1.3. However, we note this precision difference represents an increase in output accuracy. To ensure the outputs of all three separable variants were identical their respective arithmetic operational orders were kept uniform.

Three implementation variants of separable filtering were developed, namely the Original, Transposed and Updated Separable implementations. These variants differ only in their respective memory access pattern and are discussed in detail in Chapter 4.6. We discuss the

Algorithm 5.4. : Original Separable Convolution

```
1: procedure X Pass
2:     for z = 0 → zLimit do
3:         for y = 0 → yLimit do
4:             for x = 0 → xLimit do
5:
6:                 if !GoodVoxelCheck(x, y, z) then
7:                     Wavelets[x, y, z] ← 0
8:                 else
9:                     ConvolveRowOrientatedFilter(x, y, z, Coefficients[], Wavelets[])
10:                end if
11:            end for
12:        end for
13:    end for
14: end procedure
15:
16: procedure Y Pass
17:    for z = 0 → zLimit do
18:        for x = 0 → xLimit do
19:            for y = 0 → yLimit do
20:
21:                if !GoodVoxelCheck(x, y, z) then
22:                    Wavelets[x, y, z] ← 0
23:                else
24:                    ConvolveColumnOrientatedFilter (x, y, z, Wavelets[], Intermediate[])
25:                end if
26:            end for
27:        end for
28:    end for
29: end procedure
30:
31: procedure Z Pass
32:    for y = 0 → yLimit do
33:        for x = 0 → xLimit do
34:            for z = 0 → zLimit do
35:
36:                if !GoodVoxelCheck(x, y, z) then
37:                    Wavelets[x, y, z] ← 0
38:                else
39:                    ConvolveSpectralOrientatedFilter(x, y, z, Intermediate[], Wavelets[])
40:                end if
41:            end for
42:        end for
43:    end for
44: end procedure
```

implementation of these variants by providing a detailed discussion on the Original Separable variant followed by highlighting key differences of the two remaining variants.

Algorithm 5.4 shows the Original Separable variant implementation. This variation passes the row, column and spectral aligned filters of the three separate filter passes in the same direction as filter orientation. This is accomplished by changing the controlling loops in the First (line 2-4), Second (line 17-19) and Third (line 32-34) pass to step the filters in three

Algorithm 5.5. : Row Orientated Filter Convolution

```
1: procedure ConvolveRowOrientatedFilter(x, y, z, Coefficients[], Wavelets[])
2:     for xoffset = −filterHalfWidth → filterHalfWidth do
3:         CPX ←Calculate convolve position(x, xoffset, spacing)
4:         if OutofBounds(CPX) then
5:             CPX ← Reflection(CPX)
6:         end if
7:         CP ← (z ∗ spatialSize) + (y ∗ xLimit) + CPX
8:         filterPos ← xoffset + filterHalfWidth
9:         if GoodVoxelCheck(CP) then
10:            Wavelets[x, y, z] ← Wavelets[x, y, z] − (Coefficients[CP] ∗ filter[filterPos])
11:        end if
12:    end for
13: end procedure
```

Algorithm 5.6. : Column Orientated Filter Convolution

```
1: procedure ConvolveColumnOrientatedFilter(x, y, z, Wavelets[], Intermediate[])
2:     for yoffset = −filterHalfWidth → filterHalfWidth do
3:         CPY ←Calculate convolve position(y, yoffset, spacing)
4:         if OutofBounds(CPY) then
5:             CPY ← Reflection(CPY)
6:         end if
7:         CP ← (z ∗ spatialSize) + (CPY ∗ xLimit) + x
8:         filterPos ← yoffset + filterHalfWidth
9:         if GoodVoxelCheck(CP) then
10:            Intermediate[x, y, z] ← Intermediate[x, y, z] − (Wavelets[CP] ∗ filter[filterPos])
11:        end if
12:    end for
13: end procedure
```

Algorithm 5.7. : Channel Orientated Filter Convolution

```
1: procedure ConvolveSpectralOrientatedFilter(x, y, z, Intermediate[], Wavelets[])
2:     for zoffset = −filterHalfWidth → filterHalfWidth do
3:         CPZ ←Calculate convolve position(z, zoffset, spacing)
4:         if OutofBounds(CPZ) then
5:             CPZ ← Reflection(CPZ)
6:         end if
7:         CP ← (CPZ ∗ spatialSize) + (y ∗ xLimit) + x
8:         filterPos ← zoffset + filterHalfWidth
9:         if GoodVoxelCheck(CP) then
10:            Wavelets[x, y, z] ← Wavelets[x, y, z] − (Intermediate[CP] ∗ filter[filterPos])
11:        end if
12:    end for
13: end procedure
```

separate directions. Additionally, each pass calls a different filter response function (lines 9, 24 and 39) for row (Algorithm 5.5), column (Algorithm 5.6) and spectral (Algorithm 5.7) aligned 1D filter response calculations.

Separable filtering reduced loop nesting relative to DUCHAMP's 3D convolution as each 1D filters elements only vary in one dimension. This simplified edge case handling as the filter response position only needs checking against two edges for each separate pass. Additionally this simplification reduces the number of floating point operations required for edge case checking by a factor greater than $\frac{X^3}{3X}$. DUCHAMP's nested loop filter response calculation results in $2(X^3+X^2+X)$ filter edge checks required per voxel, while our separable filtering implementation only requires $6X$ checks per voxel (true for all separable filtering variants).

The Update Separable algorithm variant was the simplest to implement. Here, the row, column and spectral orientated filters were propagated linearly through the data in each respective filter pass. To accomplish this, the second (line 19-21) and third (line 34-36) pass controlling loops in the Original Separable filtering variant (Algorithm 5.4) were kept identical to the first pass controlling loop (4-6). The filter response functions used in the Original Separable variant are kept the same.

The Transpose Separable algorithm variant was implemented to optimise the 1D filter convolutions by transposing the data between filter passes to ensure that all memory reads were linear. The controlling loops were kept identical as with the Update Separable convolution. However, each filter pass only calls the row orientated filter response function. The Transpose operation is carried out within Algorithm 5.5 line 10 by writing the results of the filter responses to a different index, the required transposed address.

As mentioned above, the *à trous* wavelet reconstruction algorithm is sensitive to floating point rounding error. Changes to the operation order can result in large changes to the final output. Operation order is maintained in both the Original and Updated Separable algorithms. However, operation order can be violated within the Transpose Separable algorithm as column and spectral-aligned data can both be transposed to be row-aligned in two ways, the standard transpose and the reflection of that transpose. To maintain accuracy, we implement specific transpose operations (Figure 5.2) to preserve operation order.

The First Pass Transpose transposes the data set anticlockwise on the Z-axis to transform the column-aligned data into a row alignment. Thus (X,Y,Z) of the output from the first filter pass is mapped to (Y, XLIMIT-X, Z). Additionally, the array limits or dimension sizes for row and columns are also exchanged. The relevant limit exchanges for the second and third transpose operations are not given but are implied by their respective mapping operations. The Second Pass Transpose transposes the data anticlockwise on the Y-axis to transform the spectral-aligned data into a row alignment. Thus (X,Y,Z) of the output from the second filter pass is mapped to (Z, Y, XLIMIT-X). The Third Pass Transpose simply corrects the data alignment to its original alignment state by reversing all operations. Thus (X,Y,Z) is mapped to (Z, XLIMIT-X, YLIMIT-Y).

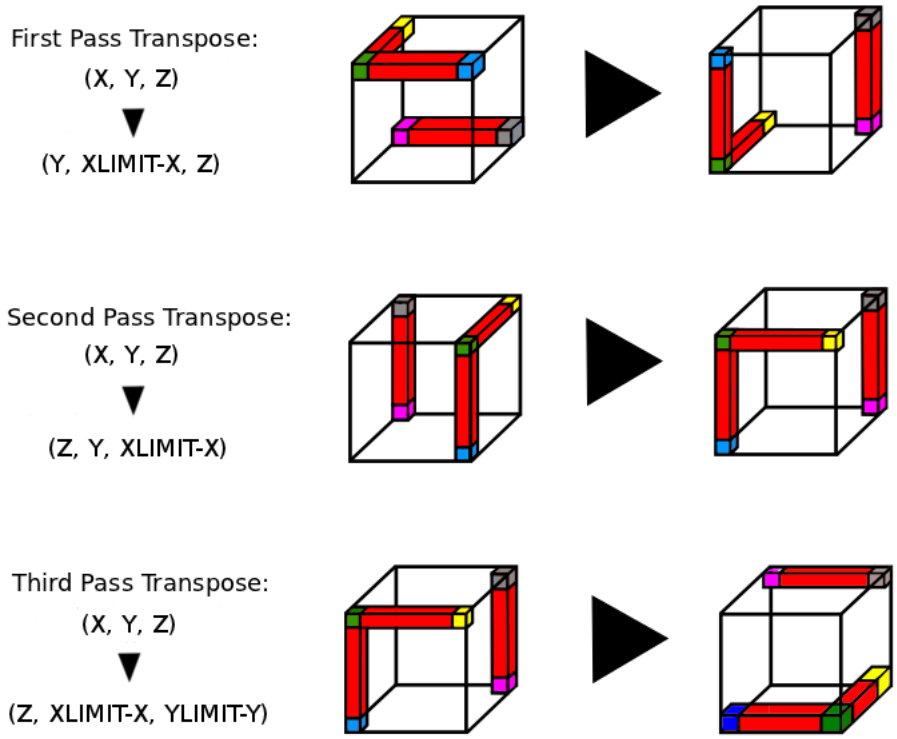


Figure 5.2: Transpose mappings operations required by Transpose Separable algorithm. These mappings ensure consistent operation order with respect to the alternative separable filtering algorithms.

5.2.2 SSE Commands Implementation

SIMD execution on a single CPU core was implemented using Intel's SSE commands[1]. We limited the set of instructions used to SSE2 and lower to avoid hardware dependencies as later SSE releases are not fully supported by other CPU brands [34]. To speed up computation, "packed" operations were used to concurrently compute 2 (double precision) or 4 (single precision) identical floating point operations. When data set size is not a multiple of the SSE unit, the small remainder was calculated sequentially.

We attempted to apply SSE to the parallelisable components of the *à trous* wavelet reconstruction algorithm, namely the convolution and update data procedures (Chapter 3.3). However, preliminary testing showed SSE to be hindered by the branching operations in the convolution procedure (Algorithm 5.2 lines 14, 21 and 28) and the update operations (Algorithm 5.1 line 24) that tests voxel values for significance. Only the update procedures in lines 15 and 32 were suitable for SSE implementations as they contained no branching.

SSE makes use of 128-bit XMM registers [34] to facilitate packed operations. We convert the source and destination arrays used by both Update procedures to SSE registers by defining the following:

$$\_\_\text{m128d*}\ \text{pSrc1} = (\_\_\text{m128d*})\ \text{pArray}$$
$$\_\_\text{m128*}\ \text{pDest} = (\_\_\text{m128d})\ \text{pResult}$$

The m128 and m128d data types are used for single and double precision packed operations respectively. SSE performance is improved by ensuring the starting address of data is 16-byte

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

aligned i.e. the memory address is divisible by 16. This alignment was ensured by using _aligned_malloc:

float* array = (float*)_aligned_malloc(SIZE*sizeof(float), 16);

Two SSE Intrinsics were used to execute a single concurrent packed operation, namely ADD and SUBTRACT. The suffixes ps and pd refer to packed single and packed double respectively.

* pArray1 = (_mm_sub_ps (pArray1, pArray2)
* pDestin = (_mm_add_pd (pArray1, pArray2)

These intrinsic operations are applied by looping through the data (as normal). However, the m128 pointer propagates in multiples of 16-bytes which results in only $\frac{\text{DATA SET SIZE}}{4}$ and a $\frac{\text{DATA SET SIZE}}{2}$ loops required to completely apply the packed commands to the data for single and double precision respectively.

5.3 Parallel implementations

The third implementation phase consisted of porting our already improved *à trous* wavelet reconstruction algorithm to a multi-core CPU implementation. This was accomplished by using the platform independent multithreading CPU library OpenMP to execute sections (task level parallelism) of code or operations (loop level parallelism) in parallel. The *à trous* wavelet reconstruction algorithm does not contain many tasks that can run in parallel (Chapter 3.3). However, many of the tasks themselves are embarrassingly parallel, namely the convolution procedures and all Update operations. However, unlike SSE parallelism, multi-core CPU parallelism can better handle branching and is suitable for the parallelisation of all the update and convolution procedures. The focus of parallel development was the convolution procedure as it takes up approximately 95% of algorithm run-time (preliminary testing results Chapter 6) .

The filter response functions, both separable and 3D, within the convolution procedure are executed per voxel with no dependencies between them. We implement multi-core parallelism within our system via OpenMP by executing filter response functions in parallel. In the DUCHAMP 3D convolution implementation, the OpenMP for loop parallelisation pragma (#pragma omp parallel for) was placed around the voxel indexing in Algorithm 5.2 lines 4-6. The flat addressing and edge case handling required that 7 variables which control position during the filter response calculation be made private, increasing memory use slightly. In the Separable Filtering convolution implementations (Algorithm 5.4), parallelism was implemented separately for each filter pass by placing the "parallel for" pragma around the lines 2, 17 and 32. However, each filter pass required only a few private variables, reducing the amount of concurrent memory use.

Finer grain parallelism between elements of a particular filter response calculation (Algorithm 5.2, lines 12, 19 and 26) was not considered as this would require separate threads for every voxel in the data set. Thread creation overhead would exceed the benefits of parallelism at this level of parallelism. Coarser grained parallelism was not possible owing to the dependencies between procedures (Chapter 3.3).

The Update operations (without SSE) (Algorithm 5.1 line 15,24 and 32) were similarly parallelised using the 'parallel for' pragma. However, SSE implementations of these procedures required segmenting data into sequential blocks, each processed by one thread. This ensured contiguous memory necessary for efficient SSE computation.

Optimal thread count to be launched will be determined during Testing (Chapter 5.5) with the findings presented in Chapter 6.

5.4 External memory management library implementation

Memory management solutions were required to bypass the slow disk access bottleneck during out-of-core computation to facilitate the processing of large data sizes on 'desktop' hardware in practical time-frames. In this section we discuss the specifics of implementing three popular memory management libraries, namely Mmap, Boost.Interprocess and Stxxl (Chapter 2.7.2). This discussion covers the implementation of the data structures, mapping operations and performance tuning procedures for each library. We do not discuss the installation procedures of these libraries as both Boost and Stxxl are well documented and Mmap is a POSIX compliant set of system calls which forms part of the Linux Operating System.

All three of the implemented libraries use file-backed memory mapping to facilitate memory management. Both the Mmap and Boost libraries require a bin file be created and grown to the required size with standard C/C++ file I/O. We created a separate file for each data structure used in the *à trous* wavelet reconstruction algorithm. Stxxl defines a mapped region over multiple disks, to facilitate software RAID, using a .stxxl file. However, we define only one Stxxl mapped disk region with disk=/media/diskName/stxxl,2G,syscall to allow for fair comparison with Boost and Mmap. The syscall flag allowed for direct I/O transfers on user memory pages [12]. Although file size can grow automatically, we defined a starting file size of 2 GB to avoid the majority of the auto-sizing overheads.

To map the created file-backings with Boost.Interprocess we defined the following:

```
Create new file mapping
file_mapping mapName(FileName, read_write);

Map the entire file region whilst defining read-write permissions
mapped_region regionName(mapName, read_write);

Get the address and size of the mapped region
Boost_Map = (Data Type*)regionName.get_address();
map_size = region.get_size();
```

To map the created file-backings with Mmap we defined the following:

```
Mmap_map = (Data Type*)mmap64(0, FILESIZE, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_POPULATE, FileName, 0);
```

Mmap64 is used to map large files with 64-bit addressing and is not an indication of a double precision map. MAP_PRIVATE ensures that the mapping can only be accessed by a single process. MAP_POPULATE prefaults the mapping to cause a read-ahead in the file, thus partially loading the mapped file into memory.

Mmap_map and Boost_map data structures function as a normal C/C++ array but have the advantage of managed data paging. Although Mmap_map required flushing changes made on paged-in memory to ensure consistency with its file-backing. The Boost_map structure is automatically made consistent with its file-backing.

Stxxl implements its own versions of STL data structures which automatically make use of its available file-backings. We utilised the stxxl::vector data structure for Stxxl memory management as its behaviour is closest to that of a standard array. This custom vector required several variables be defined. We selected the recommended default settings, namely:

```
stxxl::VECTOR_GENERATOR<DataType,NumberOfBlocksInPage=4,PagesInCache=8,
BlockSize=(2xx10124x1024)>::result vector_type
```

The stxxl::vector setup variables were the only performance tuning variables found within Stxxl. Attempts to optimise Stxxl were unsuccessful as no tuning combination was found which resulted in an *à trous* wavelet algorithm run-time less than 6 times greater than the competing memory management implementations. The Stxxl implementation was subsequently abandoned. It is unclear if software RAID (multiple disks) was required to achieved competitive performance or whether the memory access patterns used in the *à trous* algorithm were ill-suited for this library.

Our Mmap implementation was performance tuned using Madvise (Chapter 2.7.2) system calls. These tuning calls allowed for optimisation by specifying to the operating system the likely paging scheme required at specific points within the *à trous* algorithm. The Update operations in the *à trous* wavelet reconstruction algorithm were set to use the Linear Paging Scheme which aggressively reads-ahead and frees pages soon after use. The Statistics components used Random Access paging which minimises paged read-ahead as it is not particularly useful and frees up disk resources. The convolution procedure used Normal Access paging which keeps the standard Operating Systems paging scheme. No performance tuning functionality was found in the Boost.Interprocess library.

Initial profiling of our memory managed implementations showed that replacing all data structures with memory managed equivalents reduced system performance significantly. Optimal performance was achieved only when the coefficient, wavelet and wavelet copy (separable filtering only) arrays were mapped. The Input, Output arrays and the temporary data structures used during data sorting were not mapped. This is covered in more detail in the profiling results (Chapter 6.3) and is only mentioned here to highlight the limit imposed on data set size due to this partial mapping strategy. The maximum data set size that can be processed is $\frac{1}{3}$(physical memory + swap space on disk).

5.5 Testing

Precise timing was required in order to accurately evaluate our implemented system and determine the exact speed-ups attained through performance-enhancing development. Additionally, it was critical that any changes to the *à trous* wavelet reconstruction algorithm did not invalidate its use as a scientific tool.

Validity testing ensured all outputs were identical to their DUCHAMP counterparts up to double precision or were shown to increase system accuracy (Chapter 6.1.1.3). We do not discuss the actual testing procedures used as comparative testing is trivial. In the remainder of this section we discuss the performance testing procedures, and hardware and software specifics used to accurately time our implemented system improvements. Additionally, we discuss the data sets used in both performance and validity testing.

5.5.1 Hardware and Software Testing specifics

The Test hardware specifics are:
Processor: Intel Core i7-2600 CPU @ 3.40 GHz, Quad-core, Hyper-threading
Physical Memory: 2x 4 GB DIMM DDR3 (8 GB total)
Graphics Card: Not Applicable
Secondary Storage: 1 TB Seagate Barracuda 7200 RPM (Average Read Rate 102.7 MB/s)
Motherboard: Asus P8H67

The Operating System specifics are:
Linux Distribution: Ubuntu 11.04
Swap space: 14 GB

All DUCHAMP variables besides those listed below were left as default.
reconDim = 3
snrRecon = 5.
snrCut = 3.
alphaFDR = 0.05
FDRNumCorChan = 3
filterCode= 1 (uses the $5 \times 5 \times 5$ B3-Spline filter)

5.5.2 Performance Testing

The POSIX API gettimeofday [26] was used to generate timestamps expressed as seconds and microseconds since the Epoch [27]. Calculating the difference between time stamps generated accurate time measurements for all the components in the *à trous* wavelet reconstruction algorithm. We note the microsecond accuracy of this clock is not guaranteed as the Time Stamp Counters (used to facilitate microsecond accuracy) on each core of a multi-core system may be slightly out of sync [5]. However, this does not occur in practise as the Intel CPU in our test system uses synced Time Stamp Counters. Additionally, accurate microsecond timing was found to be unnecessary as all timing results were on the order of milliseconds to minutes for the smallest data sets and minutes to hours on the largest tested data sets.

Timing biases which result from Operating System services and background processes taking up processor time were minimised through batch testing and terminating non critical system services (including XServer). Batch testing ran 20 iterations of each version of the *à trous* wavelet reconstruction algorithm detailed in this chapter. The average of the lowest 5 recorded timing measurements for each algorithm component was considered the official time. A Batch number greater than 20 was infeasible for the larger data sets as a single batch had the potential to run for a week or more.

5.5.3 Test data sets

The set of test data consisted of both real and synthesised radio spectral data (3D FITS files). Real data was obtained from the The HI Nearby Galaxy Survey (THINGS)[2] [97]. However, the real data sets are concentrated in the size range of 230-520 MB with a few data sets in the 700-1000 MB range. Synthesised data populated with white noise (simpler to produce than real data composites) was generated to fill the gap between these two ranges and to produce

[2] http://www.mpia-hd.mpg.de/THINGS/Overview.html

the largest test data which exceeded the maximum size of the data contained in the THINGS survey. The largest data set created was a single-precision 1000 million voxel cube (3.7 GB) which results in a total memory use of 18.5 GB within the *à trous* algorithm.

Data sets were divided into in-core (small) and partially out-of-core (large) data sets. In-core data sets, computed with the *à trous* algorithm, only allocate a memory size that can be completely held in physical memory. In contrast, out-of-core problems will require a portion of allocated memory be held in swap space on disk. The threshold between these two data sets occurs at a data set size of approximately 324 million single-precision voxels. In-core data sets were used to profile the high performance computing improvements implemented in our system. Both in-core and out-of-core data was used in the evaluation of the implemented memory management schemes.

5.6 Summary

In this chapter, the implementation specifics of all improvements and redevelopment of the DUCHAMP *à trous* wavelet reconstruction algorithm were discussed. Three separable filtering convolution variants were implemented to reduce the computational complexity of the DUCHAMP 3D convolution implementation. The implementation of separable filtering required higher memory use but has the advantage of increased floating point arithmetic accuracy. The three separable convolution variants differ between each other by their respective memory access patterns.

Vector instruction parallelism could only be implemented efficiently with SSE2 for SIMD when *à trous* wavelet reconstruction contained no execution branching. In contrast, multi-core CPU parallelism was simple to implement with OpenMP pragmas. The superior branch handling of multi-core CPUs allowed for all SIMD *à trous* wavelet reconstruction components to be parallelised.

Three memory management libraries, namely Mmap, Boost and Stxxl, were used and fine-tuned to improve out-of-core computation. However, the Stxxl implementation was abandoned due to poor performance results. A partial memory mapping strategy where only certain data structures were mapped was implemented to increase performance. However, this decreased the maximum data set that could be processed to $\frac{1}{3}$(physical memory + swap space on disk).

The hardware and software specifics and variable definitions used during testing are discussed. Finally, we discussed the source and size range of data sets used during performance profiling.

Chapter 6

Results

The *à trous* wavelet reconstruction algorithm greatly improves the reliability and completeness of the DUCHAMP automated source extraction procedure. However, the *à trous* wavelet reconstruction algorithm is computationally expensive and memory intensive, and does not scale to the performance required to process the large data sets expected in the next generation of ultra-wide and ultra-deep HI surveys in practical time-frames.

'Desktop' hardware was considered for a high performance computing solution which could potentially scale to the computational requirements of large scale source extraction. Efficient use of relatively small amount of computational resources on 'desktop' hardware in conjunction with improvements to the *à trous* wavelet reconstruction were required in order to achieve this goal. In addition, the out-of-core disk access performance bottleneck that arises from the allocated memory exceeding 'desktop' hardwares' relatively small physical memory resources had to be overcome in order for the computation of large scale data sets to be feasible.

In this chapter, we report on the evaluation of our improved DUCHAMP *à trous* wavelet reconstruction algorithm. These performance improvements were achieved with a combination of algorithm redesign, efficient use of the memory hierarchy, Intel's SSE commands and multi-core CPU parallelism. To facilitate the computation of large data sets, we assess the potential of memory management libraries for mitigation of the disk access bottleneck of 'desktop' hardware by comparison of the performance of three currently popular memory management libraries: Mmap, Boost and Stxxl.

The *à trous* wavelet reconstruction algorithm components are categorised into five procedure types: load, convolution, update, statistics and miscellaneous. Load procedures create and populate data structures. The convolution procedure consists of multiscale filtering processes which make up the majority (up to 98%) of the *à trous* wavelet reconstruction algorithm run-time. Update procedures update the relevant data structures after the execution of the convolution procedure. Statistics procedures estimate the noise spread and level used to define the thresholds that are used to test feature significance and the algorithms end condition. Miscellaneous procedures are categorised as operations involved in the the creation, setting and retrieval of control variables and other data management tasks.

The three procedure types that contribute the most to total run-time, in ascending order, are the update, statistics and convolution procedures. The update and convolution procedures formed the majority of development in the prototype system. The DUCHAMP statistics procedures already use the highly optimised STL nth element partial sort algorithm and were not considered for further development.

We present the results in three sections: single core optimisation, multithreading and memory management. In each section, we discuss separately data sets with spatial dimensions which are powers of two larger than 512 (hereafter referred to as Power 2 data sets) and data sets with spatial dimensions that are not powers of two (Regular data sets). This is to highlight the significant performance differences between these two data set types for the DUCHAMP 3D convolution and our improved convolution implementations.

6.1 Optimal Serial Implementations

Here we investigate the performance advancements specific to a single thread of execution in order to isolate the computational improvement obtained with algorithm improvements and the use of single-threaded CPU hardware instructions to facilitate vector instruction parallelism. Algorithmic improvement refers to the reduction of DUCHAMP 3D convolution computational complexity with several alternative convolution implementations which employ separable filtering techniques for reduced computational complexity and better memory access patterns. Vector instruction parallelism for a single core CPU is facilitated through SSE2 commands (Chapter 2.7.1) which enables concurrent floating point operation execution on a single CPU core. Only the SSE improvements to Update procedures are discussed as the convolution component of the *à trous* reconstruction algorithm is poorly suited to SSE parallelism owing to its extensive divergent execution (branching).

In addition, we assess divergent output between separable filtering and DUCHAMP 3D convolution which results from floating point arithmetic rounding error. Finally, we discuss the contribution of these improved serial implementations to total run-time.

6.1.1 Convolution

The graphs in Fig 6.1 compare run-times and relative performance of the original serial DUCHAMP implementation (blue line) with our alternative convolution (iterative) implementations: a 3D filter implementation (red line) and three separable filtering methods; the Original Separable algorithm (green line), the Transposed Separable algorithm (yellow line) and the Updated Separable algorithm (pink line). We compare the average recorded run-time for a single convolution procedure or a set of three convolution passes in the case of separable filtering; both with respect to data set size. Total convolution run-time is not used for comparison as it is proportional to the number of convolutions performed for each algorithm iteration (derived from data set dimension) and the number of iterations the algorithm performed (dependent on the data set's noise properties) for each data set. These variables are not consistent across the test data set range. All testing was performed using a B3-Spline filter of size $5 \times 5 \times 5$.

The 3D filter implementation is a direct port (approximate) of the DUCHAMP *à trous* reconstruction algorithm. This implementation is used to establish whether the performance of our *à trous* wavelet reconstruction algorithm was comparable with the DUCHAMP implementation to ensure fair comparison with further developments. The Original, Transposed and Updated Separable algorithms (Chapter 5.2.1) each employ a 3-pass convolution procedure with each pass using a 1D filter which row-, column- and spectra-aligned for the first, second and third filter pass respectively. However, the direction of filter propagation for each algorithm differs which results in distinct memory access patterns.

The Original Separable algorithm propagates the row-, column- and spectra-aligned filters over

the data in the row (first pass), column (second pass) and spectral directions (third pass) respectively. The Updated Separable algorithm propagates all three filters (with their different orientations) linearly through memory for each of the filter passes. For the Transpose Separable algorithm, the data cube is transposed after each filter pass so the subsequent passes need only propagate a row-aligned filter linearly through memory. This is enabled by aligning data (with a transposition operation) in the column direction to the row direction after the first pass and repeating this process with data in the spectral direction after the second pass.

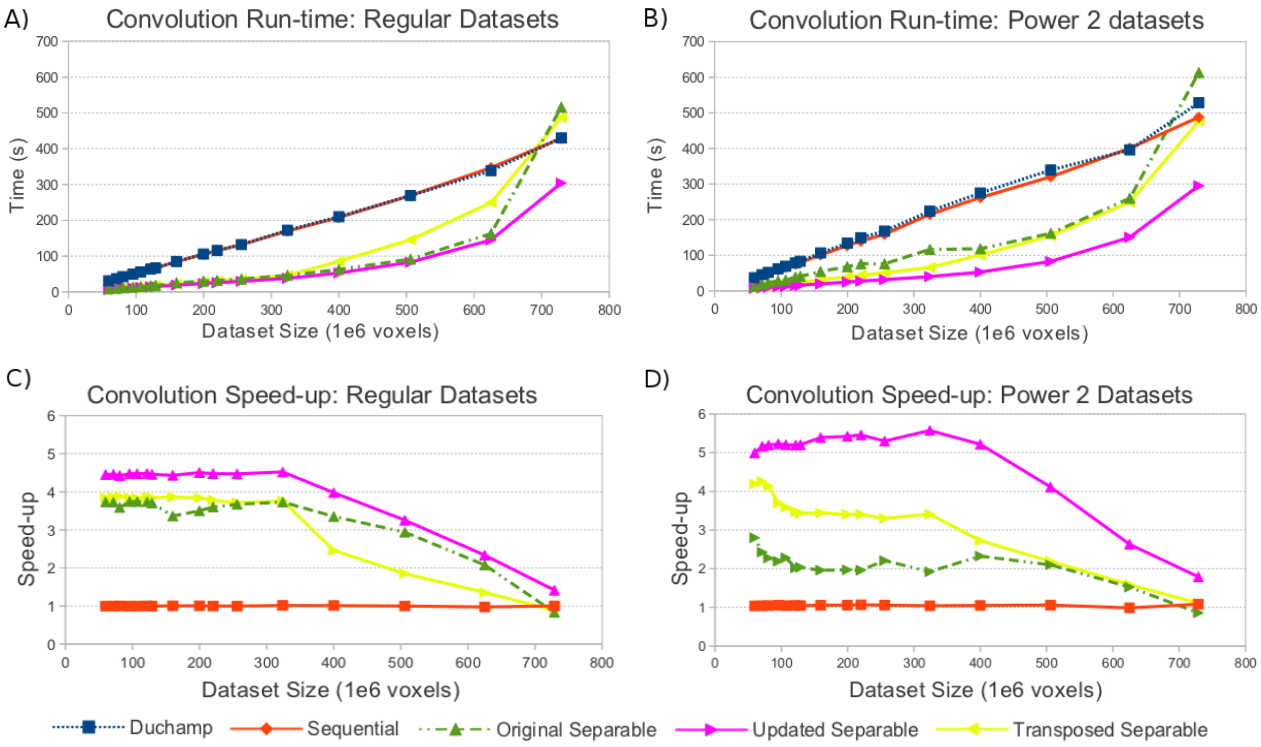


Figure 6.1: Performance of the convolution component of *à trous* Wavelet for all sequential filtering algorithm implementations for Regular (A & C) and Power 2 (B & D) data sets, showing run-times for the DUCHAMP algorithm (blue line), our 3D sequential implementation (red line), our Original (green line), Transposed (yellow line) and Updated (pink line) Separable Filtering algorithm. All algorithms implement single thread execution. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive. Robust statistics are used to estimate noise.

6.1.1.1 Convolution Results For Regular Data sets

Figure 6.1 (A & C) shows that our 3D implementation (red line) has similar performance to the DUCHAMP counterpart (blue line). The Separable Filtering implementations all substantially reduce computation time by decreasing the total operations per voxel (Chapter 5.2.1) when convolving the data from the 3D filter size (the product of its dimensions M, N and O) to the sum of these dimensions (M+N+O). However, this operation reduction comes at the cost of multiple required passes through the data and extra memory allocation to store intermediate results between filter passes. In the remainder of this section we discuss the performance of each serial separable filtering implementations relative to each other and DUCHAMP 3D convolution. We discuss in-core and out-of-core performance separately. Partial out-of-core computation begins at 400 million voxels for DUCHAMP 3D convolution. In contrast, separable filtering convolution begins out-of-core computation at 324 million voxels ($\sim$ 90% main memory use) as a result of its 20% larger allocation of memory.

The Original Separable filtering implementation (Figure 6.1 C, green line) achieves an average

performance increase of 3.7× (relative to DUCHAMP 3D convolution) for small data sets which are processed completely in-core. This is lower than the theoretical performance improvement of 8.3× expected for the $5 \times 5 \times 5$ test filter (Chapter 5.2.1). Fig 6.2 C (run-time per 1e6 voxels) shows that the Original Separable filtering implementation's inefficiency is largely constrained to the last two separable filter passes, which have respective run-times 20-22% and 20-60% larger than the first pass. The near uniformity of the first and second pass relative run-time results (in-core) indicates that run-time for these passes is increasing linearly with data set size: an optimal outcome. Additionally, this performance uniformity indicates that data set dimension has little or no effect on the relative run-time of these two passes. Similarly the majority of third pass results (Fig 6.2 C) are uniform, excluding the 159, 200 and 220 million voxel data sets which all have significantly larger spectral extents and relative run-time increases. The inefficiency of the second and third pass are explained through their memory access patterns. The second and third pass use column- and spectra-aligned filters which are propagated in the column and spectral directions respectively. This results in a strided memory access pattern which does not use the majority of the elements paged in cache lines (row-aligned blocks of memory) and consequently, poor performance. The relative increase in third pass run-time, when computing the 159, 200 and 220 million voxels data sets, is unique to this algorithm and filter pass which indicates that propagating a spectral-aligned filter in the spectral direction is particularly inefficient. Additionally, this poor performance is proportional to the extent of the spectral dimension. In contrast to the second and third pass, the first pass linearly propagates a row-aligned filter through memory which results in both efficient memory access and cache reuse.

Optimal computation was expected with Transpose Separable filtering (Fig 6.1 C, yellow line) as the data set is transposed between filter passes to allow for each 1D convolution pass to be computed with only the linear propagation of row-orientated filters. This was shown in the first pass of the Original Separable algorithm to be the most optimal memory access pattern. However, only an average performance increase of 3.8× is achieved with this algorithm for in-core data sets. Performance is limited by the average run-times (Fig 6.2 E) for the first and second filter passes are respectively 10% and 22% larger than expected. These performance decreases are associated with the additional computation required to compute the new transposed flattened array address for each filter response. This cannot be a significant contribution to run-time as the third filter pass of this algorithm is relatively efficient, only averaging 2% slower than the efficient first filter pass of the Original Separable convolution. It is likely that the first and second pass are affected by the memory access pattern of the write operations within the transpose operation which are strided in memory. Finally, we note that unlike the Original Separable convolution, run-time increases for all three passes are perfectly linear with data set size and are not a function of the data set dimension.

Maximum performance is achieved with the Update Separable filtering method (Fig 6.1 C, pink line), with a 4.5× average performance increase over DUCHAMP for in-core data sets. The memory access of the latter two passes, which use column- and spectra-aligned 1D filters respectively, is strided across memory which should cause poor performance. However, the row propagation of these filters allows for efficient cache use as the respective nth elements of every filter placement are linear in memory. This optimal cache use allows the run-times of the latter two filter passes (Fig 6.2 G) to almost equal that of the efficient row-orientated linear filter passes (first pass); thus an optimal memory access pattern is nearly achieved. However, the multiple passes required for this algorithm and the non-zero time required to cache items from memory, prevent this algorithm reaching the theoretical limit of a 8.3× performance increase.

An exponential drop in performance is experienced by both the DUCHAMP and Separable Filtering implementations when memory use begins to exceed the limits of physical memory.

The 20% higher memory use of the Separable Filtering algorithm causes this drop-off to occur at 324 million voxels ($\sim$ 90% main memory use; the remaining $\sim$ 10% of memory use is allocated to other system processes). Performance drop-off for the DUCHAMP implementation begins at the significantly larger data set size of 400 million voxels. Additionally, the rate at which DUCHAMP performance decreases is less than that experienced by any of the Separable Filtering implementations.

The rate of performance drop-off for out-of-core performance is not consistent between the Separable Filtering implementations. The Updated Separable convolution (Fig 6.1 A, pink line) has the lowest rate of performance degradation (Fig 6.1 A & C, pink line) with a near linear drop-off with the increase of out-of-core memory allocation.

The Original Separable algorithm's performance drop-off (Fig 6.1 A & C, green line) is slight for the first partial out-of-core data sets (324 -506 million voxels). However, beyond 506 million voxels the rate of performance drop-off increases greatly. At the 729 million voxel mark (168% memory use), where approximately a third of allocated memory is required to be on disk (specific to the test system), computational performance drops lower than the serial DUCHAMP implementation. Fig 6.2 C shows that the latter two filter pass run-times exponentially increase at a larger rate than the optimal first pass. This behaviour is caused by the propagation of filters in the column and spectral directions spanning more pages of memory which requires frequent disk access to page the required data into memory. However, due to small spectral extents, run-time for the third pass is kept low between 324 and 625 million voxels. This effect is similar to the larger third pass run-times achieved for the 159, 200 and 220 million voxel data sets with their relatively large spectral extents. A large increase in third pass run-time is achieved at 729 million voxels as computation is largely out-of-core and the spectral strided memory access pattern requires frequent disk access.

The initial rate of performance degradation is greatest for the Transposed Separable convolution (Fig 6.1 A & C, yellow line) with run-time 20-50% greater than the Original Separable convolution between 400 and 625 million voxels. In Fig 6.2 E the run-time increase for the second filter passes is significantly larger than that of the other two filter passes. The transpose mapping operations in this pass are strided in the spectral direction and consequently, write operations are not sequential on disk and require more pages of data to be transferred into memory, increasing the amount of slow disk access. Also, any read-ahead buffering the operating system could perform is likely to have little or no effect on improving run-time.

6.1.1.2 Convolution Results for Power of 2 data sets

Power 2 data sets are characterised as data sets with dimensions which are powers of two larger than 512. Two categories of data sets were tested, data sets with spatial dimensions 1024×1024 (henceforth referred to as 1024 data sets) and 2048×2048 (henceforth referred to as 2048 data sets). Both categories share the property that all rows of data begin at memory addresses which are separated by a multiple of page size (4 KB) and are considered page-aligned in memory.

The convolution run-times for all tested algorithms are significantly larger when computing Power 2 data sets (Fig 6.1 B & D) than the run-times achieved when processing regular data sets. The run-times for DUCHAMP convolution (blue line) and our 3D filter implementation (red line) are both increased by 36.2% on average. For the Separable Filtering algorithms, average run-times for completely in-core data sets increases by 152%, 43.4% and 5.4% for the Original (green line), Transpose (yellow line) and Updated (pink line) Separable algorithms respectively. Small variations in timing are seen among all of the competing convolution algo-

rithms (excluding Updated Separable) between 1024 and 2048 data sets (discussed below).

Figure 6.2: Relative convolution run-times (per 1e6 voxels) for the 3D Filter (A & B), Original Separable (C & D), Transpose Separable (E & F) and Updated Separable (G & H) algorithms for both Regular (first column) and Power 2 (second column) data sets. Uniform performance indicates a linear run-time increase with an increase in data set size. Run-times for the three separable algorithms are shown in terms of their three pass components. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

Analysis of the individual passes for each of the Separable Filtering algorithms (Fig 6.2), when convolving Power 2 data sets, shows that the performance drops experienced by each separable filter pass are not uniform for each algorithm. Fig 6.2 D shows that the relative increase in in-core run-time is greatest for the Original Separable algorithm's second and third pass, where memory access is strided across columns and spectra respectively. However, in this instance, the memory addresses that are accessed by the strided filter elements are page-aligned in memory. This page-alignment occurs between both certain filter elements within a single filter response calculation and certain respective elements of succeeding filter response calculations. Additionally, these run-times differ between the 1024 and 2048 data sets, with run-time increases of 48.7% (second pass) and 255.0% (third pass) for 1024 data sets, and 105.6% (second pass) and 162.5% (third pass) for 2048 data sets. The cause of this variation is discussed below.

The Transpose algorithm shows a relative in-core run-time increase for the first and third filter pass (Fig 6.2 F). For 1024 data sets, a large performance drop of 93.4% is achieved for the third pass and smaller performance drop of 8% for the first pass. In contrast, the 2048 data sets experience a large performance drop of 75% in the first pass and a smaller performance drop of 24% in the third pass. These performance drops correspond to situations where transposed mapping operations result in page-aligned writes to memory, since these writes coincide with required page-aligned reads from memory in order to update those data values.

In contrast to the behaviour of the Transpose and Original Separable filtering algorithms, the Updated Separable algorithms' run-time increases uniformly by 8.1% for the last two filter passes (Fig 6.2 H). No significant difference is seen between 1024 and 2048 data sets. In this separable convolution variant, page-alignment only occurs between certain filter elements in the calculation of a filter response. The linear propagation of each filter (despite orientation) ensures that respective elements of succeeding filter response calculations are linear in memory, and are not separated by a multiple of page size (page-alignment).

The apparent decrease in performance for page-aligned memory access is likely caused by page-aligned cache conflict misses [83]. Cache is organised into sets each of which can contain a maximum number (generally 4, 8 or 16) [83] of cache lines (blocks of memory paged in from physical memory). The relatively small number of available cache sets are mapped from the much larger main memory space by modulating the address of a block of memory (or higher level of cache) over the total number of cache sets (a hardware specific power of 2) [74]. This results in multiple memory blocks mapping to the same cache set. When a cache set is fully populated a cache line must be evicted when caching in a new cache line (cache conflict load) [38]. If this eviction occurs prematurely (data in cache still in use) the data must be cached again when next accessed (cache conflict miss). Frequent cache conflict misses can significantly reduce computational performance [38].

The rows of both 1024 and 2048 Power 2 data sets are page-aligned and consequently column and spectra strided memory access results in frequent mapping to the same cache way. This should result in frequent cache conflicts throughout the entire cache hierarchy (to different degrees) and consequently causes poor performance. Additionally, 2048 data sets' rows are aligned with every second page instead of the one-page alignment of 1024 data sets. Strided memory accesses for 1024 and 2048 Power 2 data sets are therefore separated by different powers of 2 which affects the the number of cache conflicts and explains the apparent performance difference between data set types.

Two types of alignment are possible here: cache conflicts within a single array, as discussed above, and cache conflicts between arrays where a page-aligned data structure immediately succeeds another in memory. The latter has the potential for alignment when the corresponding elements in each data structure are accessed consecutively and can occur during convolution

calculations. However, it is dismissed as a strong cause of performance degradation (in isolation) as no significant performance drop is recorded for the first filter passes of both the Original and Updated Separable convolution algorithms computing Power 2 data sets. Cache conflicts within a single array, specifically within the frequently accessed coefficient and wavelet arrays, or a similar caching effect is the probable cause of slow-down. This can be mitigated by inserting padding data between rows to misalign the data at the expense of slightly larger memory use and more complicated array addressing functions [17, 51, 96]. However, this is unnecessary as the Updated Separable convolution implementation largely mitigates the problem of frequent cache conflicts.

Each of the competing convolution algorithms are affected to a different extent by cache conflicts as a result of their distinct memory access patterns. In the case of the Original Separable algorithm (Fig 6.2, D), filter access in the column and channel direction for the last two filter passes is page-aligned and is likely to alias to the same cache line. In contrast, the Transpose Separable algorithm (Fig 6.2, F) reads all data in memory linearly and should be exempt from cache conflicts. The slow-downs recorded for the third and first passes, for 1024 and 2048 data sets, result from page-aligned writing to the transposed addresses (updating values) to obtain the next array orientation. The cache conflicts occur during the page-aligned read operations required to bring the value, stored at each transposed address, into a CPU register so it may be updated.

The column and channel strided memory access in the last two filter passes of the Updated Separable algorithm (Fig 6.2, H) results in a power of 2 caching effect within the coefficient array. However, this caching effect is greatly reduced as cache conflicts can only occur between the elements of a single filter response. Additionally, the extent of cache conflicts effect on performance is likely hidden to some degree by this implementation's efficient use of cache memory.

The performance decreases in the DUCHAMP 3D convolution (Fig 6.2) are difficult to analyse as the convolution process is performed with a large 3D filter which makes it difficult to isolate the performance contributions of row-, column- and spectra-strided memory access. However, this 3D filter moves linearly through memory which should result in optimised (to some degree) cache use, the contributions of which cannot be easily distinguished.

The relatively higher increase in run-time for the Original Separable and Transpose algorithms (relative to DUCHAMP) reduces their respective relative performance increases from 3.7× and 3.8× to 2.2× and 3.6× respectively. In contrast, the Update Separable algorithms' efficient use of cache mitigates the problem of frequent cache conflicts to a large extent and results in a relatively small run-time increase (relative to DUCHAMP) when computing Power 2 data sets which increases the relative performance of this algorithm from an average of 4.5× to 5.2×.

For Power 2 data sets larger than 324 million voxels, the performance for all separable filtering convolution variants decreases rapidly with data set size as the extent of out-of-core computation increases. The out-of-core Power 2 run-times achieved for the Updated and Transposed Separable variants are near identical to their Regular data set run-times. This run-time similarity between data set types is caused by out-of-core slow disk access dominating run-time. In contrast, the Original Separable variants' run-time is slightly worse for out-of-core Power 2 data sets and converges on the Transposed Separable variants run-time. At 729 million voxels, the performance increases between Regular and Power 2 data sets largely disappears as convolution computation is completely transfer bound.

6.1.1.3 Floating Point Arithmetic Error

Accuracy of system output was ensured at all stages of development by comparing the produced output voxels with the corresponding voxel outputs generated by DUCHAMP. The output for the majority of DUCHAMP *à trous* procedures and our improved implementations were exactly equal. However, differences between the outputs of the DUCHAMP and Separable Filtering algorithms arise (Chapter 5.2.1) from the differing number of convolution floating point operations and the inherent rounding error for floating point (FP) arithmetic [41]. We note these output differences are almost undetectable after a single iteration of the *à trous* reconstruction algorithm. However, the differences between output grows with the compounding of FP arithmetic error over multiple convolution scales and algorithm iterations. In this section, we measure the significance of this difference and determine if this difference represents a loss or gain in accuracy relative to the DUCHAMP implementation. To determine this, we compare the single precision (SP) outputs of both algorithms against both of the corresponding double precision (DP) outputs. To ensure realistic measurements, we restrict tested data sets to real data only.

Datasets	RMSD (between methods)		RMSD SP Separable. (diff. accuracy.)		RMSD SP Original. (diff. accuracy)	
	SP (xE-10)	DP (xE-20)	DP Sep (xE-10)	DP Orig (xE-10)	DP Sep (xE-10)	DP Orig (xE-10)
DD0 154	2.92	4.22	0.601	0.601	2.54	2.54
NGC 4736	1.51	2.44	0.233	0.233	1.44	1.44
NGC 5194	1.61	3.70	0.424	0.424	1.69	1.69
NGC 7793	4.48	7.27	1.12	1.12	4.18	4.18
NGC 4214	3.79	8.92	1.23	1.23	3.20	3.20
NGC 7331	2.98	3.36	0.511	0.511	2.72	2.72
NGC 5236	4.47	9.97	0.763	0.763	4.41	4.41
KELL-IF14	2.58	3.97	0.584	0.584	2.46	2.46
NGC 2403	2.81	3.77	0.843	0.843	2.90	2.90

Table 6.1: Quantification of error between the sequential 3D algorithm and Separable Filtering algorithm for both single (SP) and double (DP) precision execution.

We use the root mean square deviation (RMSD) to measure the variation between outputs. The RMSD is a measure of standard deviation between the values of two data sets and is defined as:

$$\text{RMSD} = \sqrt{\frac{\sum_{i=1}^{n} |x_{SepFilter,i} - x_{Duchamp,i}|^2}{n}}$$

Table 6.1 shows the RMSD between the DUCHAMP algorithm using 3D convolution and our improved system using separable filtering convolution. For single precision computation, the RMSD (between methods) is shown to be very small with deviation measures for all data sets to the order of 1e-10. Double precision computation shows a decrease in deviation to the order of 1e-20, which is consistent with the increase in precision. Although these RMSD values are small, we cannot predict the increase in rounding error for larger data sets where the increased number of convolution scales and algorithm iterations could further compound error. For the Separable Filtering convolution procedures to be useful, the RMSD values measures should represent a reduction (or be equivalent) in rounding error relative to DUCHAMP. To establish precision loss or gain with the implementation of separable filtering, the single precision output of each algorithm is compared against both double precision outputs.

The Separable Filtering algorithm achieves an approximately equal RMSD measure (Table 6.1, RMSD SP Separable) for both comparisons with double precision outputs. Similarly, the single precision DUCHAMP algorithm (Table 6.1, RMSD SP Original) also achieves approximately

equal RMSD measures against both double precision procedures. However, the RMSD of the Separable Filtering algorithm (when compared to DP) is on average an order of magnitude smaller than the respective DUCHAMP RMSD values. This indicates that the rounding error produced with separable filtering is significantly less than 3D DUCHAMP convolution and that accuracy is gained with our implementations. This reduced rounding error results from the significantly fewer convolution floating point operations required for this algorithm. The discrepancy is likely to hold at double precision but this cannot be tested without computation at a higher level of precision which is not available on the current test system.

6.1.2 SSE optimisation results

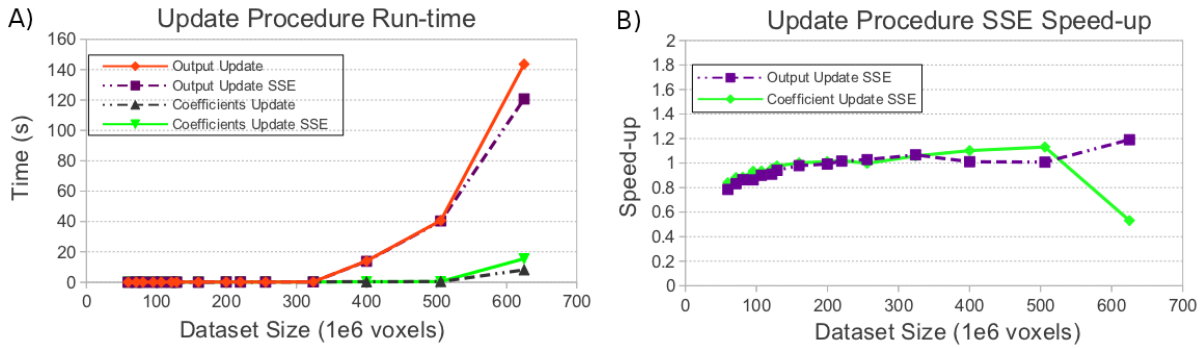


Figure 6.3: Performance increases for the SSE implementations of Coefficient and Output Update procedures using quad-packed instructions. The results shown are the average times recorded (A) and the relative speed-ups (B) for a single execution of the coefficient update (green line) and output update (purple line) procedures against their respective DUCHAMP procedure. Performance is consistent between Regular and Power 2 data sets.

Optimisation of the serial *à trous* algorithm with Intel's Streaming SIMD extensions (SSE) was constrained to the non-branching Update procedures only. Convolution and branching Update procedures are poorly suited for SSE parallelism owing to their extensive divergent execution (branching). The statistics procedures are considered optimal and are not considered for algorithm redesign (Chapter 2.6.3).

Update procedures consist of linear passes through multiple data structures whilst testing and performing floating point operations between them. (Chapter 2.6.3). The optimisation of these procedures is achieved with the Intel's SSE2 commands only, specifically the quad-pack (single precision) and dual-pack (double precision) floating point operations. These SIMD operations allow for the concurrent execution of a single command on multiple data on a single CPU.

Two Update procedures (Fig 6.3 A & B), namely, the Coefficient (green line) and the Output (purple line) Update procedures, are ported to an SSE implementation. No significant performance differences exist between Regular and Power 2 data sets and are not discussed separately. The SSE implementations performance for both algorithms is only 80% of that achieved with DUCHAMP for 59 million voxels (the smallest data set) and increases linearly until equivalent with DUCHAMP's performance at 200 million voxels. This results from large SSE register loading overheads relative to the low run-times of Update procedures processing small data sets.

For larger data sets, performance grows slowly with increases in data set size, reaching 1.3× and 1.1× for the Coefficient and Output Update procedures respectively. Although these performance increases are small in terms of run-time for a single procedure, the overall time decrease to the *à trous* algorithm is significant as these operations are performed frequently.

The Coefficient Update performance degrades significantly for data sets larger than 506 millions voxels (out-of-core), dropping to half the performance of DUCHAMP as run-time is effected by the significantly larger disk access time. In contrast, a performance increase from 1.1× to 1.2× is achieved with the Output Update procedure for 506 and 625 million voxel data sets respectively. The Output Update procedures higher performance results from immediately succeeding the Feature Update procedure (discussed below). Both these procedures access the same data structure which ensures that that a portion of the out-of-core memory (percentage unknown) required by the Coefficient Update procedure has already been paged from disk after the Feature Update procedure executes. The timing results (both Update procedures) for the largest tested data set of 729 million voxels are omitted from Figure 6.3 as they are an order of magnitude larger than the rest of the tested data sets.

A performance increase of 2-3× is expected with the implementation of quad-packed SSE commands. The large disparity between the theoretical and actual performance increases achieved with SSE is likely caused by the low number of operations performed compared to the amount of memory read into cache and the number of operations required to pack the SSE registers. Additionally, the DUCHAMP version is expected to be near optimal from compiler optimisations as update operations are simple, computationally light procedures.

6.1.3 Total Serial Run-times

Performance improvements to the convolution and Update procedures significantly contribute to the reduction of overall run-time for the *à trous* wavelet reconstruction. In this section, we discuss the significance of these improvements by discussing their contributions along-side the contributions of the components not considered for improvement with respect to the overall speed-up of the reconstruction algorithm.

Total run-time for the *à trous* wavelet reconstruction algorithm (Fig 6.4) consists of the following components: convolution (green region), noise estimation with statistical measures (yellow area), update procedures (red region), loading operations (blue region) and miscellaneous components (brown region). Miscellaneous components refers to all intermediate and set-up operations for the other components. Only the update and convolution procedures are considered for redesign. Miscellaneous, load operations and the statistical components are considered optimal, the latter using the STL nth element partial sort algorithm which has a linear complexity on average. All results reported are specific to each data set, as significant variability in run-time is caused by differing scales (dimension dependant) and iterations (noise dependant) computed. Consequently, results only approximately indicate computation time as a function of data set size and may differ substantially between Regular and Power 2 data sets.

For the unoptimised DUCHAMP algorithm (Fig 6.4, A) computing Regular in-core data sets, the convolution component comprises on average 89% of total run-time. This increases to an average of 92% for Power 2 data sets as convolution run-time increases from frequent cache conflicts. For data sets larger than 324 million voxels, both data set types experience an exponential increase in run-time as a result of slow out-of-core computation, achieving run-times of 1.8 and 2.4 hours at 729 million voxels for Regular and Power 2 data sets respectively. The contribution of convolution to total run-time decreases to 70% as the contributions of Update and random access Statistics procedures increases at a larger rate for out-of-core computation. The statistical procedures perform particularly poorly (order of magnitude run-time increase) as random access to data stored on disk is particularly inefficient.

All *à trous* wavelet algorithm implementing Separable Filtering convolution (Fig 6.4, C-H) have

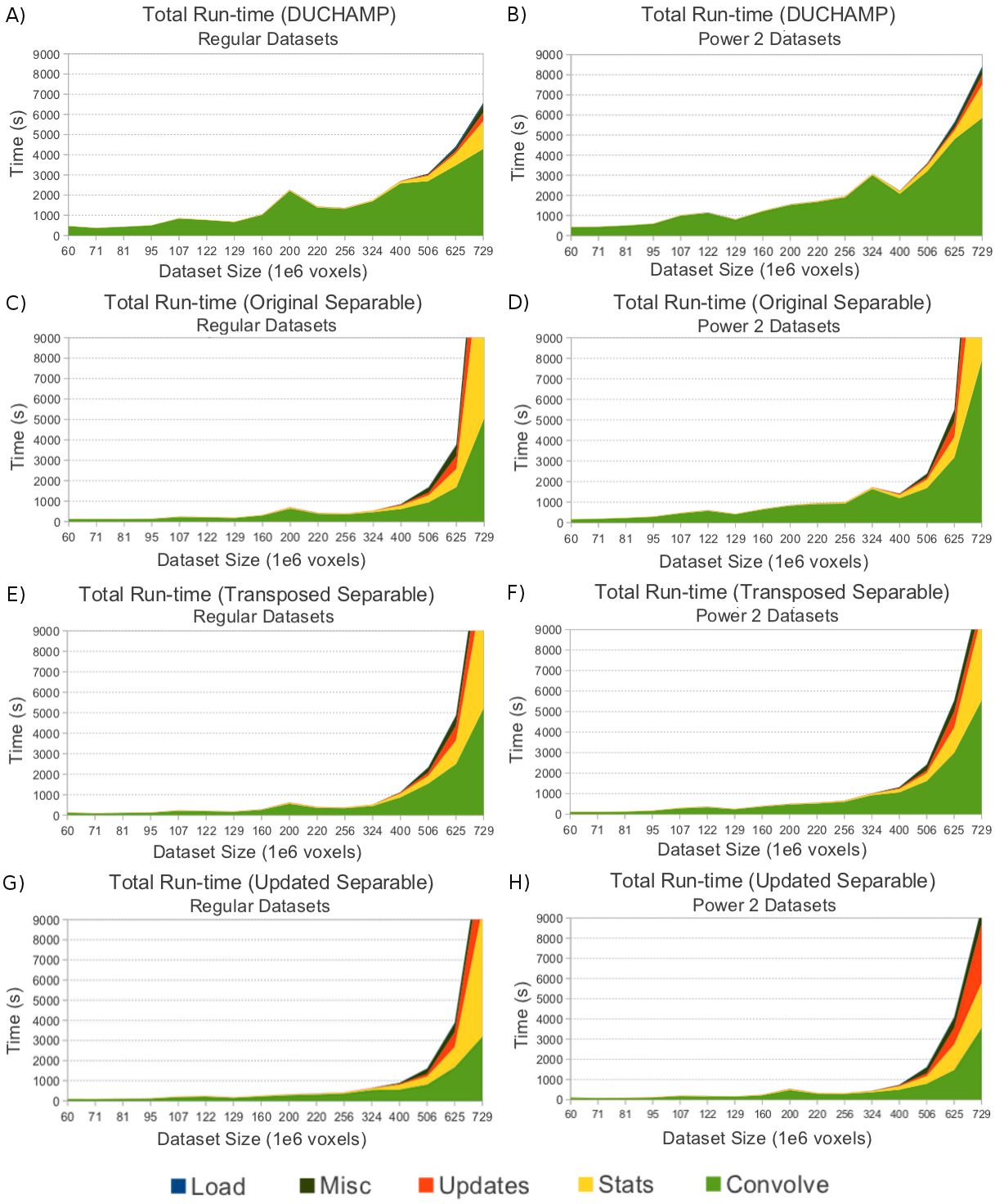


Figure 6.4: Total run-times for the improved sequential *à trous* wavelet reconstruction algorithm implementations. These run-times are deconstructed into their component procedures to identify their specific contributions. Results are shown for the DUCHAMP (A & B), Original Separable (C & D), Transpose Separable (E & F) and Updated Separable (G & H) algorithms for both Regular and Power 2 data sets. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

dramatically decreased run-times for in-core data sets in comparison to the DUCHAMP implementation. These decreases are proportional to the reduced convolution run-times achieved with Separable Filtering techniques. However, the 20% higher memory use of the Separable Filtering convolution components increases the extent of out-of-core computation for these implementations. Subsequently, out-of-core performance decreases at a faster rate with an increase

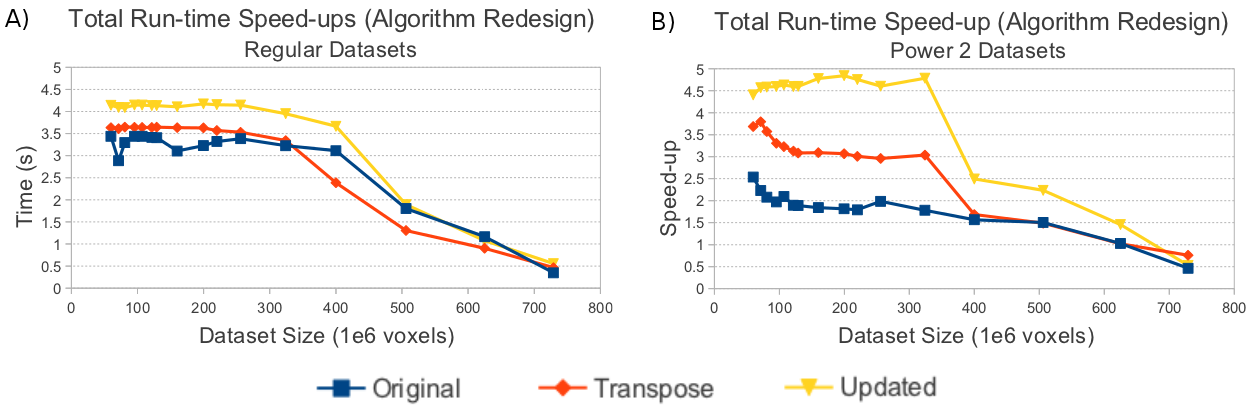


Figure 6.5: Run-time speed-ups for the sequential *à trous* wavelet reconstruction algorithm. Shown are Original Separable (blue line), Transpose Separable (red line) and Updated Separable (yellow line) for both Regular (A) and Power 2 (B) data sets. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

in data set size when compared to the unoptimised DUCHAMP implementation. Similarly to DUCHAMP, a large variation in performance is seen between Regular and Power 2 data sets as each convolution procedure causes frequent cache conflicts to some degree. The SSE implementations for the Update procedures are disabled for data sets smaller than 200 million voxels as they are detrimental to run-time (Chapter 6.1.2). For in-core data sets larger than 200 million, the SSE implementations for the Update procedures only slightly reduce the run-time contribution of these components.

The reduction in run-time for our improved single core *à trous* wavelet algorithm computing in-core data sets represents significant performance increases (Fig 6.5) over the original DUCHAMP implementation. The wavelet reconstruction algorithms implementing the Original (blue line), Transposed (red line) and Updated Separable (yellow line) convolution and SSE Update procedures result in average speed-ups of 3.3×, 3.6× and 4.1× respectively for regular in-core data sets. This is reduced to 2× and 3.3× for the reconstruction algorithms implementing the Original and Transposed Separable convolution for Power 2 data sets, owing to the greater impact of caching effects on their convolution implementations relative to DUCHAMP. The reconstruction algorithm implementing Updated Separable convolution achieves a Power 2 data set performance increase of 4.6× speed-up as it not affected (or the penalty is mitigated) by frequent cache conflicts. However, all speed-ups are reduced to near DUCHAMP performance for out-of-core data sets.

6.1.4 Summary

Updated Separable convolution performs the best out of all tested convolution variants by reducing the number of floating point operations required per voxel and efficiently utilising cache memory. Performance increases of 4.5× and 5.2× are achieved for in-core Regular and Power 2 data sets respectively relative to DUCHAMP. The relatively larger Power 2 performance results from a memory access pattern that minimises the number of cache conflicts experienced relative to DUCHAMP. Additionally, separable filtering convolution produces less rounding error than DUCHAMP 3D convolution and is consequently slightly more accurate. This results from the reduced number of floating point operations used by separable methods. However, separable filtering performance comes at the expense of greater memory use and as such, performance rapidly drops to below DUCHAMP performance for data sets larger than 324 million voxels (out-of-core). This is in contrast to the DUCHAMP 3D implementation which only begins out-

of-core computation at 400 million voxels. At 729 million voxels, separable filtering performance is completely transfer bound by slow disk access (out-of-core computation).

The implementation of Intel's SSE commands, to exploit the SIMD nature of the Update procedures, does not improve performance significantly. Performance increases range from 1.1× to 1.3× for data sets between 255 and 625 million voxels. Smaller data sets are negatively effected by SSE overheads.

Overall, the single-threaded *à trous* wavelet reconstruction performance is increased by 4.1× and 4.6× for Regular and Power 2 data sets respectively.

6.2 Multi-core parallelism

The *à trous* wavelet reconstruction algorithm is well-suited for fine-grained parallelism (Chapter 3.3) on multi-core CPUs. Specifically, the Update and convolution procedures are embarrassingly parallel and have a large potential for performance increases on parallel architectures. The majority of these procedures contain branching, which makes them better suited to multi-core CPU parallelism than to alternative parallel solutions, such as the GPU. In this section, we discuss the specific performance increases attained by porting these *à trous* procedures to a multi-core implementation using OpenMP on an i7 4-core CPU with hyper-threading. Algorithmic performance increases for all parallelised procedures are reported relative to their respective single-threaded implementations to isolate the specific contribution of multi-core parallelism.

We focus primarily on the parallelisation of the implemented convolution procedures (Fig 6.6), namely the original 3D Sequential algorithm (DUCHAMP) and the three competing Separable Filtering algorithms: Original Separable, Transposed Separable and Updated Separable. In addition, we discuss the parallel improvements to the Update procedures without the single-threaded SSE command optimisations considered above. Preliminary testing shows that the combination of these parallel paradigms bottlenecks CPU performance, causing performance to drop below that of the original DUCHAMP implementation.

6.2.1 Parallelised convolution procedures

Despite the significant performance improvements achieved with Separable Filtering, the convolution component of the *à trous* reconstruction still makes up the majority of total run-time. Both the DUCHAMP and Separable Filtering convolution implementations are embarrassingly parallel as a result of independent filtering operations (Chapter 2.6.3) and are ideal for fine-grained parallelism on multi-core CPUs. In this section, we discuss the performance improvements gained from porting the competing convolution algorithms to a multi-core parallel solution using OpenMP. We assess performance scaling with thread count on a quad-core CPU with hyper-threading.

The graphs in Fig 6.6 compare the results of the parallel DUCHAMP implementation (A & B) against our three parallel Separable Filtering methods: the Original Separable algorithm (C & D), the Transposed Separable algorithm (E & F) and the Updated Separable algorithm (G & H). Results for both Regular and Power 2 data sets are discussed to isolate the effects of cache conflicts on parallelism.

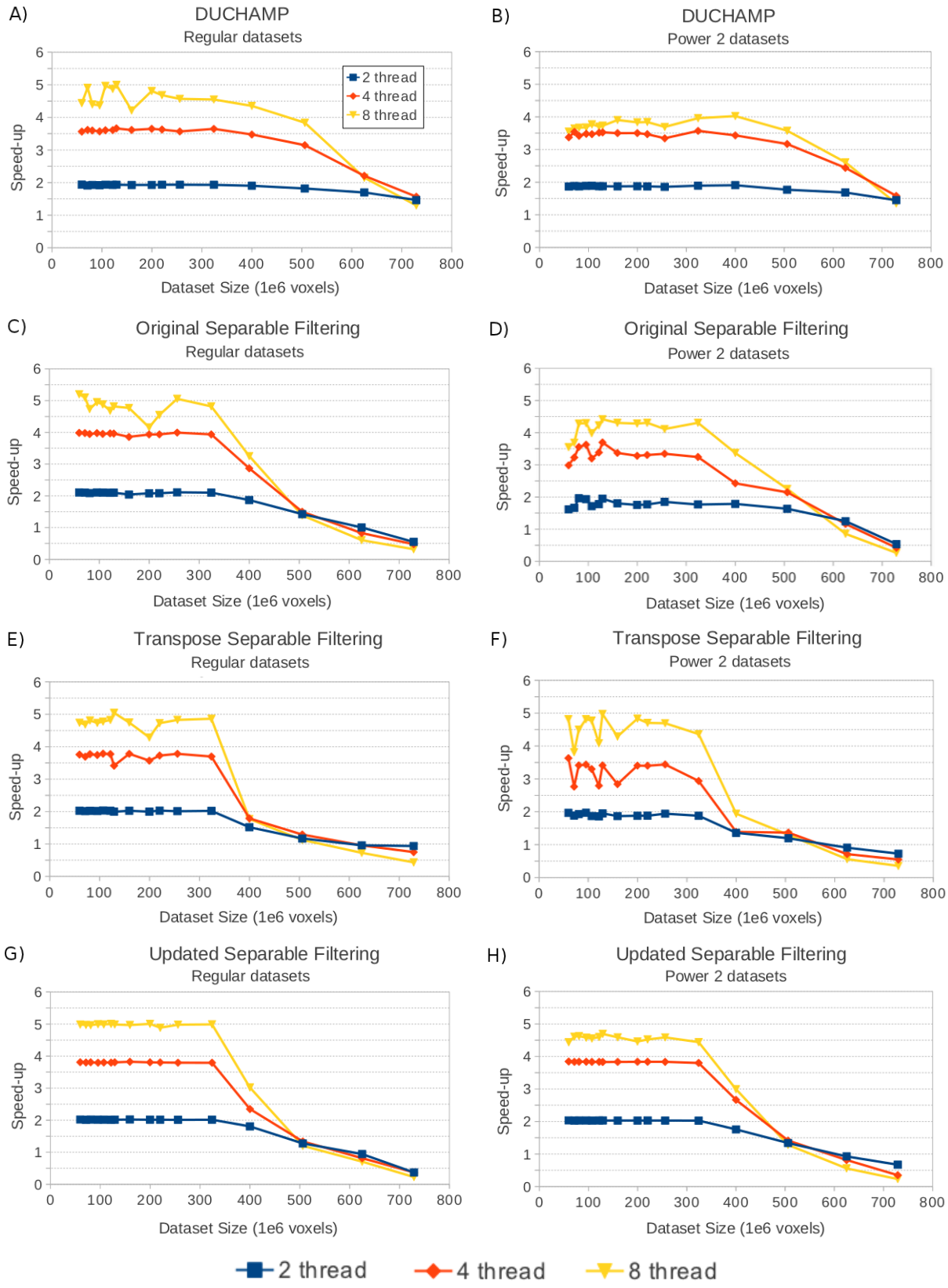


Figure 6.6: Multi-core performance increases for the convolution component of *à trous* wavelet reconstruction for both Regular and Power 2 data sets. This figure indicates the performance increases achieved with OpenMP for the DUCHAMP (A & B), the Original Separable (C & D), Transposed Separable (E & F) and Updated Separable (G & H) convolution algorithms. All speed-ups are reported relative to each algorithm's single-threaded implementation. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

6.2.1.1 DUCHAMP:

For the DUCHAMP algorithm, typical performance improvements are shown (Fig 6.6, A & B) for the number of utilised threads. For Regular data sets (Fig 6.6 A) the 2-thread (blue line) and 4-thread (red line) implementations achieve an average performance increase of 1.92× and 3.6× respectively. This is near the theoretical maximum improvement expected for 2 and 4 executing cores. However, linear performance scaling is prevented by parallel thread overheads.

The 8-thread implementation (Fig 6.6, row 1, yellow lines) achieves approximately 4.6× performance improvement, an average increase of 28% over the 4-thread implementation, as a result of hyper-threading on a 4-core CPU [93]. However, the performance for small data sets is highly variable, the average performance increase of 4-threads ranging between 16-32% . Similar behaviour is seen for both the Original and the Transposed Separable algorithms. This variation is suspected to be caused by an interplay of hyper-threading core scheduling and sub-optimal cache use.

Launching 16 threads for all convolution algorithms (not shown) results in over-scheduling of the CPU and worse performance than the 8-threaded case. This over-scheduling relationship will continue to grow with greater allocations of threads.

For Power 2 data sets (Fig 6.6 B), performance increases over the serial implementation are slightly smaller, achieving speed-ups of 1.87× and 3.5× for 2 and 4-threaded implementations respectively. This disparity in performance likely results from higher thread counts increasing the number of cache conflict stalls. Additionally, the 8-threaded performance drops significantly from 4.6× to 3.7× which may indicate that hyper-threading cannot fully optimise performance when cache is constantly evicted by conflict stalls. This behaviour similarly occurs for all competing Separable Filtering algorithms to various degrees, excluding the Updated Separable algorithm which mitigates this power of two caching effect.

Out-of-core computation again results in poor performance for all thread counts, with performance dropping off significantly for data sets larger than 400 million voxels. This algorithm uses relatively less memory than the Separable Filtering convolution which reduces the rate of performance drop-off with data set size. For the largest data set of 729 million voxels, performance of all threads converges on a 1.5× increase over the single thread performance. However, this is expected to decrease to match single thread performance for larger data sets.

6.2.1.2 Original Separable:

The parallel performance improvements achieved for Original Separable convolution (Fig 6.6, C & D) over its single thread implementation are larger than the corresponding DUCHAMP results. For Regular data sets (A), parallelism with OpenMP achieves average performance of 2.1×, 3.9× and 4.8× for the 2, 4 and 8-threaded implementations respectively for in-core data sets. Super-linear and near linear speed-ups are achieved for 2 and 4-threads respectively which exceeds the theoretical maximum performance increase from the number of cores utilised. This good performance could result from the amortisation of the relatively computationally heavy edge-cases when filter element placement extends beyond the edge of the data set. In the multithreaded solution where one or more threads are hindered by edge-case handling, convolution continues with the remaining threads which partially hides the edge-case run-time cost. Alternatively, it could be caused by the cache effect [48, 59] where the cache miss rate decreases as the accumulated caches of multiple processors can store more of the working set, significantly reducing memory access times.

The 8-thread implementation again shows the expected increased performance over the 4-thread implementation and the small variations in performance which likely result from hyper-threading scheduling and strided memory access.

Power 2 data set performance is seen to be less than that of Regular data sets, with average speed-ups of 1.8×, 3.4× and 4.1× for the 2, 4 and 8-threaded implementations respectively. However, all thread implementations experience a drop in performance for the 60, 71, 106 and 121 million voxel data sets. This is unlike the performance variability seen for Regular data sets that strongly effects the 8-threaded implementation only. The cause of this Power 2 variation could not be determined. However, similar drops in performance are seen for Transposed Separable algorithm and not for the Updated algorithm which suggests that the performance disparity is caused by a power of 2 caching effect (cache conflict misses).

Performance for data sets larger than 324 million voxels drops off sharply with the proportional increase in data stored on disk for out-of-core computation. Higher thread counts cause performance to drop at increased rates as a result of threads accessing disk more frequently. Performance for all thread counts converges to a performance increase of 1.5× at 625 million voxels, at which higher thread numbers again hinder performance with parallel scattered disk access.

6.2.1.3 Transpose Separable:

The Transpose Separable algorithm achieves average performance increases of 2×, 3.7× and 4.8× with 2, 4 and 8-threads respectively for the in-core computation of regular data sets. This constitutes a drop in performance relative to the Original Separable algorithm of approximately 4%, 7% and 8.5%. Performance is hindered by the inefficient strided read/writes required to transpose the data despite the optimal linear memory reads. However, linear speed-ups are still reached. Although we expect that this is caused by the benefits of amortising computationally expensive edge-cases, it could also be caused by the cache effect where the accumulated caches of multiple processors can store more of the working set and significantly reduce memory access times.

In-core Power 2 data sets achieve average performance increases of 1.9×, 3.2× and 4.6× for 2, 4 and 8-threaded implementations. The difference in performance between Regular and Power 2 data sets is much smaller than Original Separable algorithm as only one of the convolution passes is affected by frequent cache conflicts. Unlike Regular data sets where performance is relatively uniform, the 4 and 8-threaded implementations computing Power 2 data sets experience drops in performance of up to 20% at 71, 121 and 159 million voxel data sets. This is similar to the Original Separable algorithm's response for relatively small Power 2 data sets which are expected to be associated with power of 2 caching effects (cache conflict misses) and multi-thread core scheduling.

Out-of-core computation for the multi-threaded Transposed Separable algorithm is particularly inefficient as multiple threads are performing strided read/write operations to disk. Convolution performance begins decreasing sharply for data sets (Regular and Power 2) larger than 324 million voxels. The rate of this drop-off is greater than the similar performance drops for the DUCHAMP and Original Separable algorithms with multiple threads intensifying the penalties of scattered read/writes to disk. Consequently, convergence in performance for all thread numbers occurs approximately at 400 million voxels which is the smallest convergence point amongst all competing multi-threaded convolution algorithms.

6.2.1.4 Updated Separable:

The Updated Separable algorithm achieves near perfect speed-ups for all thread implementations, achieving average performance increases of 2×, 3.8× and 5× for 2, 4 and 8-threads respectively. Additionally, the variation in performance across all data sets is minimal, making this implementation the most predictable out of all the convolution procedures. The 2 and 4-threaded implementations exhibit higher performance than expected from parallelism, as thread overheads should limit performance. These increases are similar to the super-linear performance seen at low thread counts for the Original and Transposed Separable algorithms. The cause of this larger than expected performance increase is associated with the benefits of amortising the relatively computationally expensive edge-case or the multi-core caching effect. The 8-threaded implementation achieves a performance increase of 25% over the 4-threaded implementation. This falls into the range of the speed-up expected from Intel's hyper-threading technology.

The performance for Power 2 data sets is only reduced by 1-10% for all thread counts which results in average performance increases of 2×, 3.8× and 4.6× for 2, 4 and 8-threads respectively. This variation is significantly less than all competing convolution algorithms as a result of fewer cache conflicts and optimal cache use, discussed above. Additionally, parallel performance is near uniform across all in-core data sets.

Larger data sets show a rapid linear drop-off in performance beyond 324 million voxels, proportional to the increase in out-of-core memory use. Parallel performance for all thread implementations converges at 506 million voxels, smaller converging point than the Original Separable algorithm. At this point disk access is necessitated with multiple threads increasing the effect of the slow disk access bottleneck on performance.

6.2.2 Total performance of the parallel convolution procedure

In the previous two sections, we separately discussed the performance contributions of Separable Filtering and parallelism using multi-core CPUs. Parallel improvements were discussed relative to their single-threaded implementations to assess how performance scales with larger thread counts. This did not show the combined effect of Separable Filtering and parallelism. In this section, we discuss the performance increases achieved with parallel Separable Filtering implementations relative to the single-threaded DUCHAMP implementation.

The DUCHAMP implementation (Fig 6.7, A & B) is the parallel port of the original DUCHAMP convolution procedure. This has been previously discussed above and is included for comparative reasons only. The best parallel performance for this convolution procedure is seen with the 8-thread implementation (green line) which achieves performance increases of 4.6× and 3.7× speed-up for the Regular and Power 2 data sets respectively. The smaller performance increases for Power 2 data sets results from frequent cache conflict stalls hindering computation. However, as a result of increased out-of-core computation, performance for all thread counts drops to the same level as the single-threaded implementation for data sets larger than 400 million voxels.

The Original Separable Filtering convolution (Fig 6.7, C & D) has a significantly reduced runtime because of the combined effects of Separable Filtering (Fig 6.1) and parallelism (Fig 6.6, C & D). This results in an average convolution performance improvement of 3.7×, 7.6×, 14.5× and 17.6× for 1, 2, 4 and 8-threaded implementations respectively for in-core regular data sets. This algorithm was the most effected by cache conflicts out of all the Separable Filtering

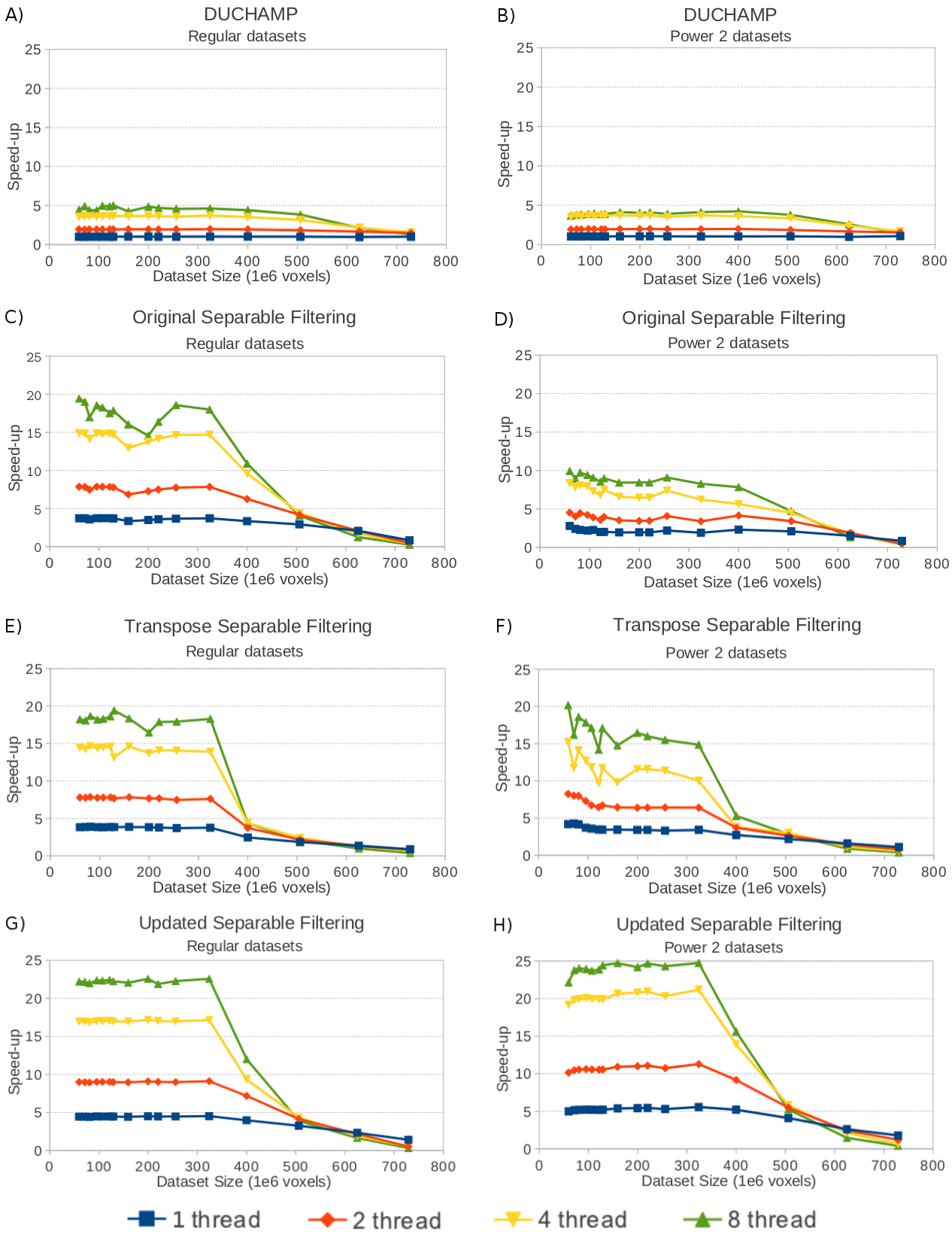


Figure 6.7: Total parallel performance results for the convolution component of *à trous* wavelet reconstruction for both Regular and Power 2 data sets. Results indicate the combined effect of algorithmic improvement and multi-core parallelism. Shown are the performance increases for the DUCHAMP (A & B), the Original Separable (C & D), Transposed Separable (E & F) and Updated Separable (G & H) algorithms. All speed-ups are reported relative to the DUCHAMP single-threaded implementation. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

implementations which results in Power 2 data sets only achieving 2.2×, 4×, 7.3× and 9× for the 1, 2, 4 and 8-thread implementations respectively. Performance of this algorithm for all thread counts drops dramatically for data sets larger than 324 million voxels, converging with the single-threaded performance at 625 million voxels. This corresponds to the increased disk access required for partial out-of-core computation. The performance contributions from higher thread counts and algorithmic improvements are small relative to this computation bottleneck. Performance continues to drop for larger data sets until it drops below the performance of DUCHAMP at 729 million voxels.

The Transpose Separable algorithm (Fig 6.7, E & F) has similar performance to the Original Separable algorithm for the majority of regular data sets, achieving 3.8×, 7.7×, 14.7× and 18.2× for the 1, 2, 4 and 8-thread implementations. In contrast to the Original Separable implementation, only one of the algorithm's separable filter passes (to a large degree) is effected by frequent cache conflicts. Consequently, results for Power 2 data are slightly more significant with speed-ups of 3.7×, 7×, 11.8× and 16.6× for the 1, 2, 4 and 8-threaded implementations respectively. However, run-time for both data set types increases exponentially for data sets larger than 324 million voxels and performance drops below a 5× performance increase for all thread counts at 400 million voxels. This rapid drop in performance corresponds to greater out-of-core disk access than the competing Separable Filtering algorithms owing to the scattered writes required for transpose mapping operations.

The Updated Separable algorithm (Fig 6.7, G & H) achieves the highest and most uniform speed-ups of all the tested convolution algorithm variants. Although similar performance gains to the other Separable Filtering algorithms are achieved with parallelism, the actual convolution procedure has a significantly better memory access pattern. This results in an average convolution speed-up of 4.5×, 9×, 17× and 22.3× for 1, 2, 4 and 8-threaded implementations respectively, for in-core regular data sets. This algorithm is only acutely affected by power of two caching effects and achieves significantly larger relative performance increases with respect to the DUCHAMP implementation, which is *not* exempt from these caching effects. This results in average performance increases of 5.3×, 10.7×, 20.2× and 24× for the 1, 2 ,4 and 8-threaded implementations. However, this algorithm's superior performance does not prevent the performance drop caused by the increase in out-of-core computation for data sets larger than 324 million voxels, dropping below the performance of DUCHAMP at 729 million voxels.

6.2.3 Parallelised Update procedures

Update procedures are the third largest contribution to total *à trous* run-time. In this section, we discuss the performance improvements achieved by porting these Update procedures to a parallel multi-core solution. We only consider parallel implementations of the original Update procedure and do not consider parallel implementations of the previous SSE implementations, as the combination of these paradigms results in poor computational performance. We assess the performance scaling of these procedures with increases in threads of execution on a quad-core CPU with hyper-threading.

We evaluate the parallel implementations of the Coefficient Update (Fig 6.8, A & B), the Output Update (Fig 6.8, C & D) and the Feature Update (Fig 6.8, E & F) procedures. There is no significant difference between the results of Regular and Power 2 data sets so results are reported for the general case. The Features Update algorithm thresholds voxel values are thresholded for significance before performing floating point operations. This made it unsuitable for SSE but it still ports well to a CPU multithreading solution.

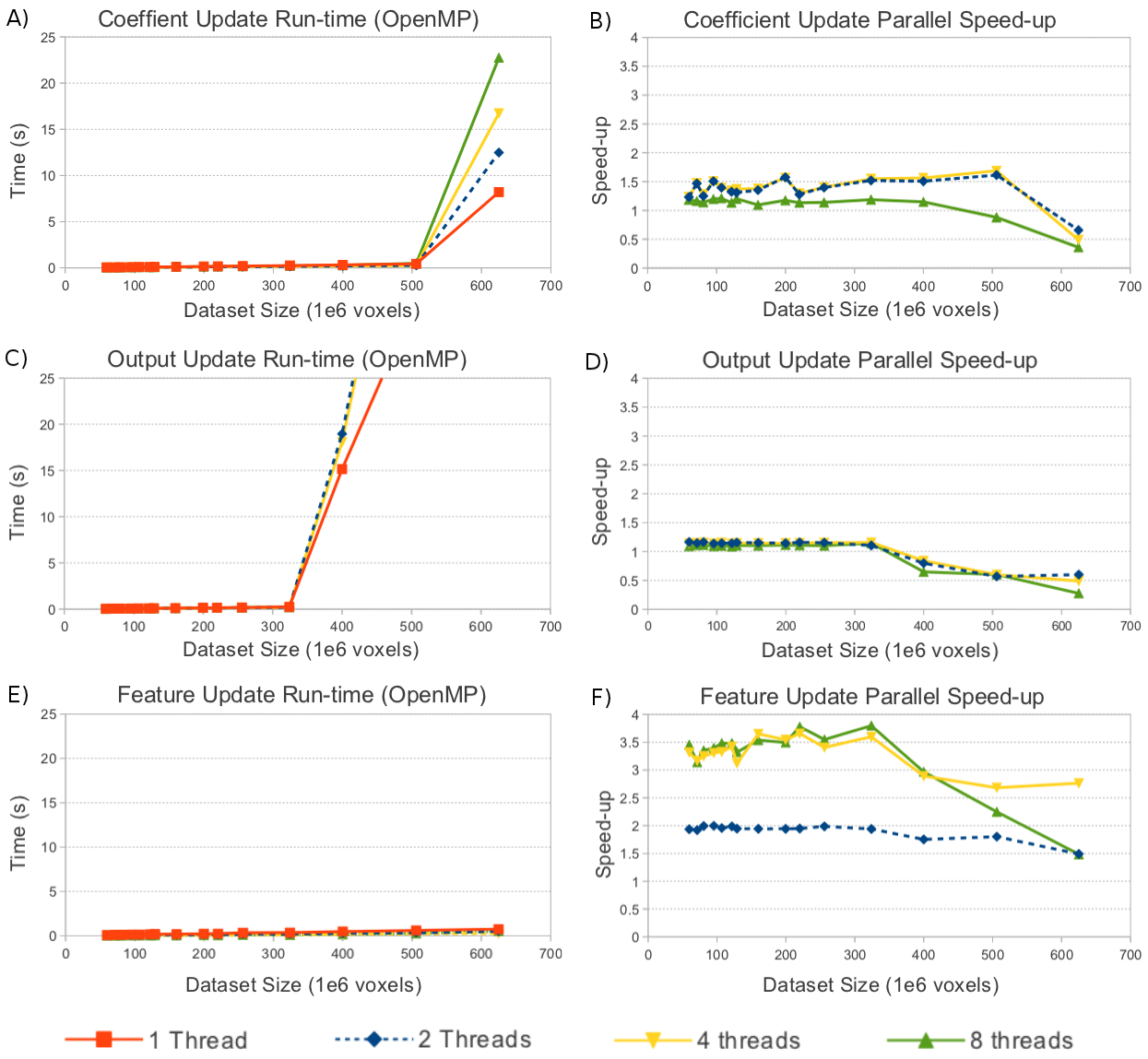


Figure 6.8: Run-times and relative performance for Update procedures parallel implementations. The results shown are the run-times and relative performance increases achieved with increased thread counts for the Coefficient Update (A & B), Output Update (C & D) and Feature Update (E & F) procedures. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

The Output Update procedure (C & D) does not perform well, with average performance increases of 1.15×, 1.14× and 1.1× with 2, 4 and 8-threads respectively. Similarly, Coefficient Update performance (A & D) increases average of 1.4×, 1.41× and 1.21× for the respective thread counts. Thread management overheads and the accessing of multiple data structures in relation to a computationally light procedure hides any small possible speed-ups as run-time is dominated by memory transfer. The performance for 8-threads is below that reached with 2-threads as more threading overheads are introduced with higher thread counts. For large data sets, performance for both Output and Coefficient Update procedures drops off sharply compared to Feature Updates, averaging the performance of the DUCHAMP procedure at 625 million voxels.

The Feature Update algorithm (E & F) shows substantial parallel performance increases which are more in line with what is expected from multi-core parallelism. The procedure achieves an average speed-up of 2×, 3.4× and 3.5× for the 2, 4 and 8-thread implementations respectively. Unlike the previous two procedures, the Feature Update algorithm primarily accesses only

one data structure to test for significant voxel values. Only significant voxels require extra computation and additional data structure access. This reduces the amount of memory that is required to be paged into cache and prevents this algorithm from being transfer bound between cache and physical memory.

6.2.4 Total Performance improvement of the *à trous* reconstruction algorithm

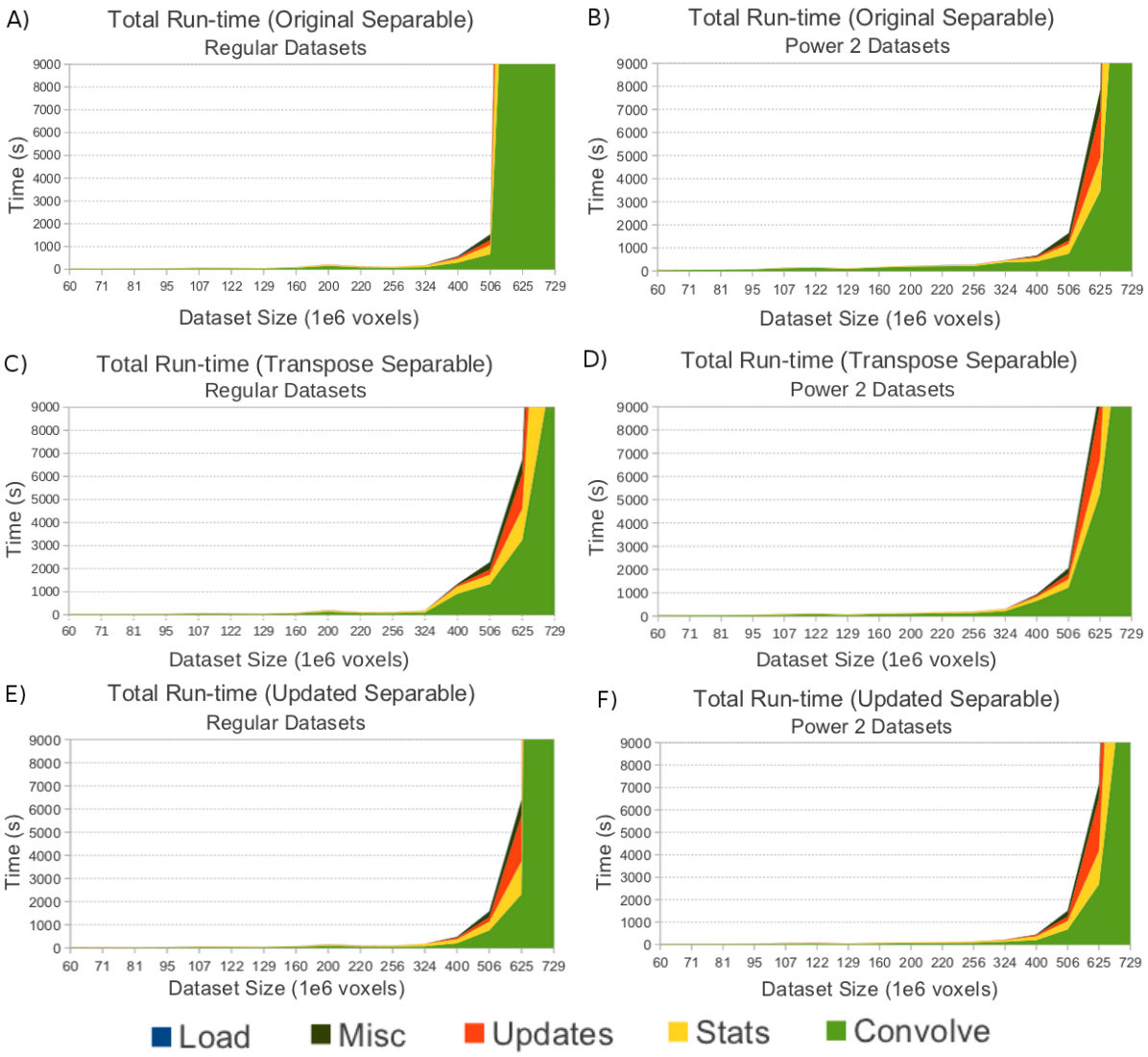


Figure 6.9: Total run-times for the improved *à trous* wavelet reconstruction algorithm implementations including parallelism and algorithmic redesign. These run-times are deconstructed into their component procedures to identify their specific contributions. Results are shown for the Original Separable (A & B), Transpose Separable (C & D), Updated Separable (E & F) algorithms for both Regular and Power 2 data sets. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

In this section, we discuss the total performance increase of the *à trous* wavelet reconstruction algorithm. We include the parallelised implementations of all the Separable Filtering convolutions and Update procedures. The SSE implementations for Updates procedures are not considered as we have discussed previously that the results with parallelism alone result in better performance. We show the results for the best performing multi-threaded implementation and analyse the run-time that each component adds to the final run-time.

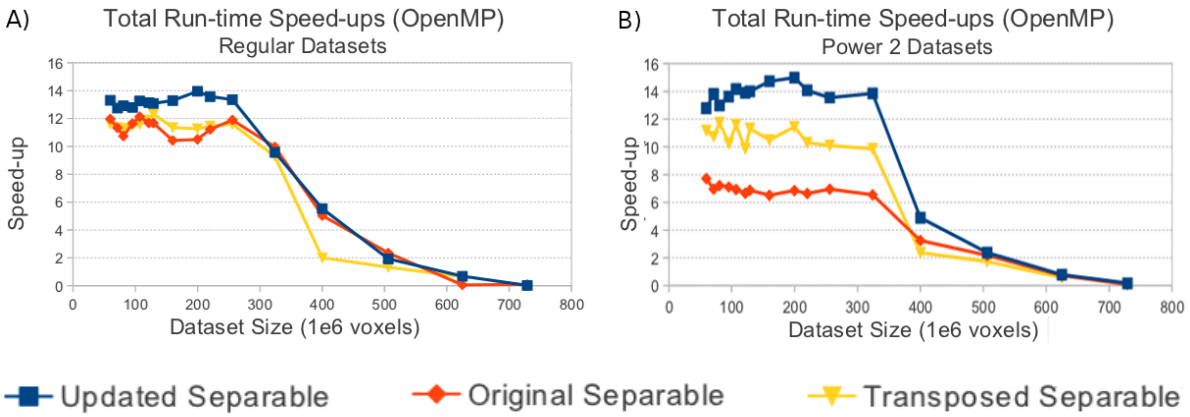


Figure 6.10: Total performance increases for the improved *à trous* wavelet reconstruction algorithm implementations including parallelism and algorithmic redesign. Performance increases are shown for the Original Separable (red line), Transpose Separable (yellow line), Updated Separable (blue line) algorithms relative to DUCHAMP for both Regular (A) and Power 2 (B) data sets. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

Total run-time for the *à trous* wavelet reconstruction algorithm (Fig 6.9) consists of the contribution from the convolution (green region), statistics (yellow region), update (red region), loading (blue region) and miscellaneous procedures (brown region), discussed above.

The timing results for in-core computation of the Original Separable (Fig 6.9, A & B), Transposed Separable (Fig 6.9, C & D) and Updated Separable implementations (Fig 6.9, E & F) for the *à trous* wavelet reconstruction are highly correlated with the convolution procedure's times as it constitutes the majority of total run-time. The results for the convolution component using 8-threads (discussed above) for Regular data sets are significant, with 17.6×, 18.2× and 22.3× speed-ups for the Original, Transposed and Updated Separable implementations. In contrast the speed-ups for Power 2 data sets are 9×, 16.6× and 24× for each algorithm as the extent of frequent cache conflicts is uniform across each algorithm. This reduces the convolutions procedure's contribution to total run-time from approximately 85-89% to 58-65% (Fig 6.11) for Regular data sets and 58-80% for Power 2 data sets. The best performing algorithm, Updated Separable Filtering, is responsible for the lowest contribution (58%) to total run-time (Fig 6.11, E & F).

Certain Update procedures perform poorly for high thread counts as a result of thread overheads and core over-scheduling. Best performance for the Coefficient (1.4×), Output (1.15×) and Feature Update (3.5×) procedures is achieved with the 2, 4 and 8-threaded implementations respectively. The cause of best performance for each algorithm varying with thread number is discussed above. Although these performance increases are relatively small, the Updates procedure's contribution to the *à trous* algorithm's total run-time decreases to only 3.5% for in-core data sets.

Statistical and miscellaneous procedures are not considered for optimisation. Their combined contribution to final run-time increases to an average of 30% for Regular data sets and 28% for Power 2 data sets. These components restrict the total performance increase possible for the *à trous* wavelet reconstruction component.

Total speed-ups for the entire *à trous* wavelet reconstruction algorithm (Fig 6.10) for in-core data sets are as follows: The Original Separable implementation averages a speed-up of 11.3× for Regular data sets and 6.9× for Power 2 data sets. The Transpose Separable implementation performs slightly better with an average speed-up of 11.4× and 10.7× for Regular and Power 2 data sets. The Updates Separable implementation has the best results with an average

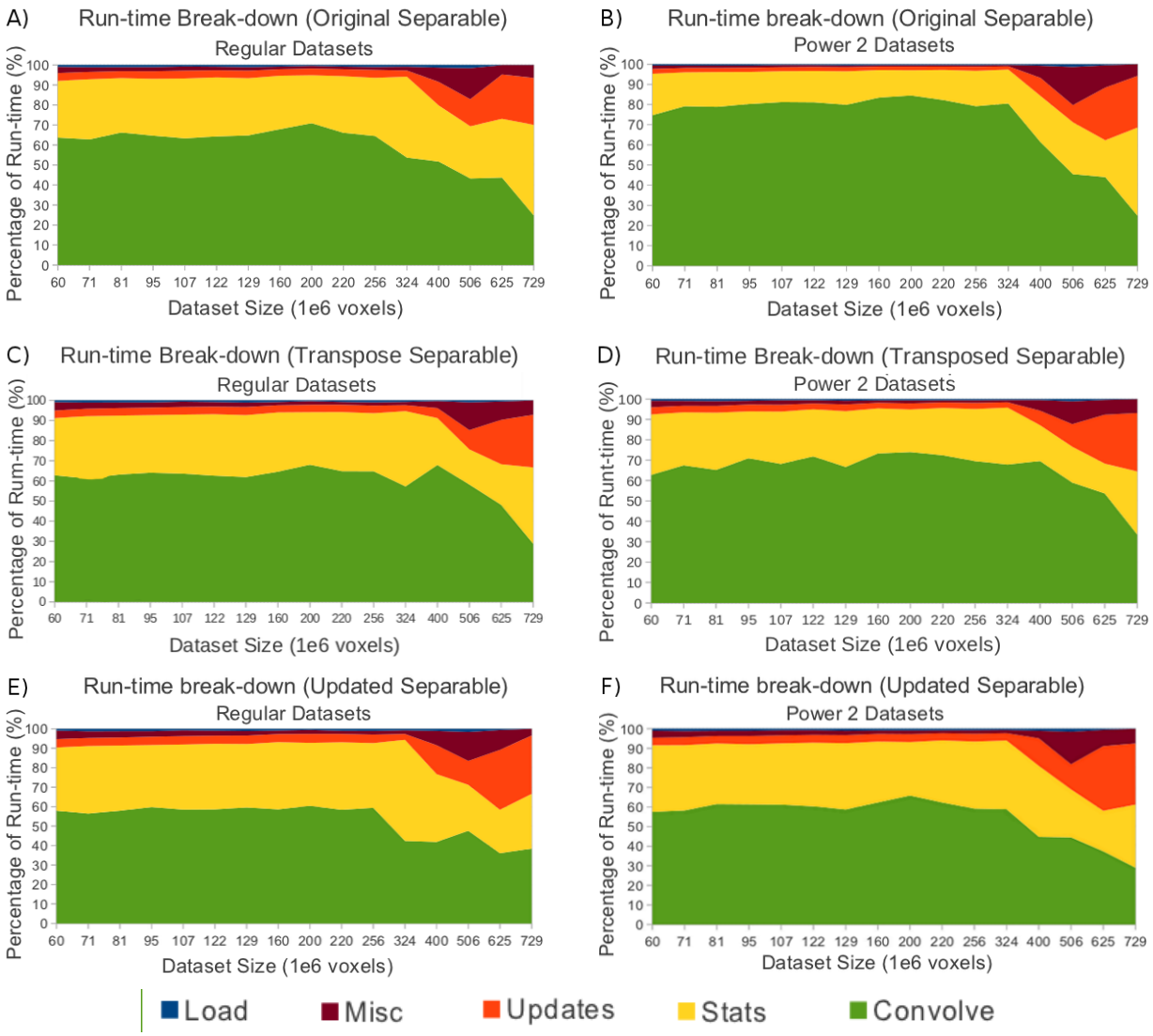


Figure 6.11: Percentage break-down for the improved *à trous* wavelet reconstruction algorithm implementations total run-times. These run-times are deconstructed into their component procedures to identify their specific contributions. Results are shown for the Original Separable (A & B), Transpose Separable (C & D), Updated Separable (E & F) algorithms for both Regular and Power 2 data sets. All speed-ups are reported relative to the DUCHAMP single-threaded implementation. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

of 13× speed-up for Regular data sets. The performance for Power 2 data set computation achieves an average of 13.9× speed-up relative to DUCHAMP but actually attains comparable run-times to Regular data set computation. Despite the implemented algorithmic improvements and parallelism, out-of-core computation is still transfer bound by slow disk access and performance decreases sharply after 324 million voxels. The effect of slow out-of-core computation is significantly increased with multi-threading as it significantly increases disk access. For all Separable Filtering convolutions computing the largest data sets, performance drops below that of the single-threaded DUCHAMP implementation due to the higher memory use. We mitigate the effect of disk access performance bottleneck by introducing memory management in the succeeding section.

6.2.5 Summary

The *à trous* wavelet reconstruction is well-suited for fine grain parallelism on multi-core CPUs. All convolution implementations scale with the increase in threads of execution for Regular data sets. The Updated Separable Filtering algorithm achieves superior results with the implementation of parallelism, achieving a parallel performance increase of 2×, 3.8× and 5× for 2, 4 and 8-threads respectively. This results in an overall convolution speed-up, relative to the single-threaded DUCHAMP implementation of 9×, 17× and 22.3× for 2, 4 and 8-threaded implementations. Additionally, the performance of Updated Separable algorithm for the computation of Power 2 data sets does not drop significantly as this algorithm is only slightly affected by cache conflict stalls, achieving parallel performance increases of 2×, 3.8× and 4.6×. This is in contrast to the large performance drops experienced by the DUCHAMP and competing Separable Filtering algorithms. Relative to the DUCHAMP solution computing Power 2 data sets, overall convolution performance increases to 10.7×, 20.2× and 24× for the 2, 4 and 8-threaded implementations.

Only Update procedures without SSE parallelism were ported to multi-core parallel implementations as the combination of these two parallel paradigms resulted in poor performance. The performance increases with multi-core parallelism are superior to that achieved with SSE. However, despite this improved performance, the Coefficient and Output Update procedures are too computationally lightweight relative to the required amount of memory transfer and result in a maximum performance improvement of 1.4× (2-threads) and 1.15× (4-threads) respectively. In contrast, the Feature Update procedure scales well with an increase in threads of execution, achieving 2×, 3.4× and 3.5× for the 2, 4 and 8-thread respectively as a result of significantly reduced memory access.

Overall, the *à trous* wavelet reconstruction algorithm (using 8-threads) implementing the Updated Separable convolution achieves the best results with an average performance increase of 13× and 14× for Regular and Power2 in-core data sets respectively. Further speed-ups are prevented by the serial Statistics procedures which are not considered for performance improvement in this work. However, this is a significant result and sufficiently increases performance to process data sets an order of magnitude larger than current data sets in practical time-frames.

6.3 Large data sets

Future surveys on the SKA precursor instruments, MeerKAT and ASKAP, and the SKA itself are expected to produce observational data sets that are exponentially larger than current surveys. In this work we considered the computational improvement of the *à trous* wavelet reconstruction algorithm on a 'desktop' commodity system, discussed above, in order to meet the computational requirements for processing large data sets in practical time-frames. These improvements alone are not sufficient for improving the computational performance of large data sets on 'desktop' hardware. The *à trous* wavelet reconstruction algorithm is extremely memory intensive and consequently the allocated memory for large data sets exceeds 'desktop' hardware's relatively small amount of physical memory. Additionally, the subsequent disk access to retrieve this data during computation is slower in orders of magnitude than physical memory access. This slow disk access can bottleneck system performance despite our implemented computational improvements. This potentially limits the data set size than can be processed in practical time-frames.

In this section, we assess the mitigation of slow out-of-core computation for the *à trous* wavelet

reconstruction algorithm with the implementation of memory management. Memory management attempts to mitigate the penalty of slow disk access by overriding the operating system's paging scheme (Chapter 2.7.2). Furthermore, it allows for memory use to extend beyond physical memory and swop space by defining file-backed swop space. Two commonly used memory management libraries are assessed: Boost and Mmap. The Stxxl library was excluded from this assessment, as preliminary testing showed run-times approximately 6× larger than the competing alternatives.

Memory management is applied to only three of the data structures used in the *à trous* wavelet reconstruction algorithm (Chapter 5.4) as preliminary testing showed that this partial mapping maximised performance. The two memory management schemes are assessed in combination with only our improved system which uses the Updated Separable convolution procedure. This improved system was selected as it achieved the highest and most predictable (low variability) in-core performance Additionally, the filters used in this procedure are propagated linearly through memory which optimises the use of paged in data and reduces disk access. The reduced disk access of this improved system makes it the most appropriate for out-of-core memory managed computation.

6.3.1 Memory-managed convolution

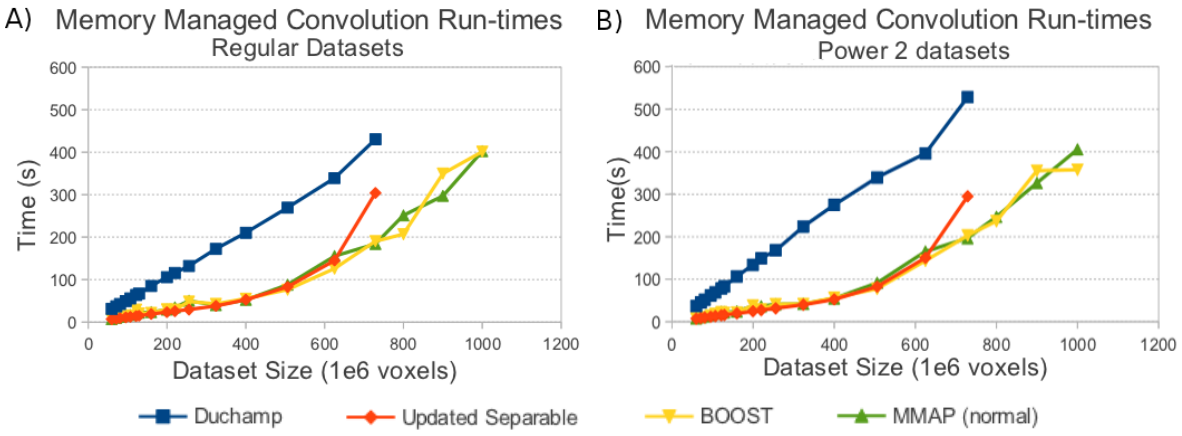


Figure 6.12: Run-times for the memory management implementations of the Updated Separable convolution procedure. Results are given for the Boost (yellow line) and Mmap (green line) implementations for both Regular (A) and Power 2 (B) data sets. Results are compared against the single threaded DUCHAMP (blue line) and Update Separable (red line) convolution procedures. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

In this section, we compare the Boost and the Mmap (normal paging scheme) implementations of the single-threaded Updated Separable filter convolution procedure. The Mmap Sequential and Random paging schemes are not shown. Sequential paging performance results are negligibly different from Normal paging and Random paging is best suited for completely random access patterns (Chapter 5.4). We only consider single-threaded memory-mapped implementations as parallelism decreased the performance of the memory-managed implementations significantly (results not shown). In Fig 6.12, the Boost (yellow line) and Mmap using normal paging (green line) are compared against the original DUCHAMP convolution implementation (blue line) and the single-threaded Update Separable algorithm without memory management (red line).

Run-times (Fig 6.12 A) for Boost and Mmap implementations are on average 32% larger than the Updated Separable implementation for completely in-core Regular data sets (smaller than

400 million voxels). This discrepancy is caused by overheads from data management and a slow first data access, as all memory mapped data is initially stored completely out-of-core. However, despite these overheads, performance is on average 3.6× better than DUCHAMP's convolution implementation. For data sets greater than 400 million voxels, the benefits of memory management outweigh its overheads, causing run-time for both Mmap and Boost implementations to increase linearly (approximately) with data set size up to 1 billion voxels (size 3.7 GB, allocated memory 18.5 GB). This linear run-time increase for both libraries contrasts the exponential run-time increase of out-of-core Updated Separable filtering convolution (red line) without memory management which results in a 2.3× performance increase relative to DUCHAMP at 729 million voxels (the largest data set used for comparison). Small performance differences are seen between the memory management libraries for larger data sets, achieving a maximum variation of 17.7%. However, the best performing algorithm alternates depending on data set size as a result of the different paging schemes implemented in each memory management scheme.

For Power 2 data sets (Fig 6.12 B), the performance of both the Boost and Mmap implementations for in-core data sets is approximately equal to the performance achieved with Regular data sets. This results from the Updated Separable algorithm's memory access pattern not causing frequent cache conflict stalls. However, for out-of-core data sets (larger than 400 million voxels) the performance of these libraries is superior to Regular data set performance as the variable performance between data sets is significantly reduced. The run-times of the Boost and Mmap implementations are both on average 9.5% larger than the best performing library computing Regular data sets. The cause of this performance increase could not be determined.

6.3.2 Memory-managed Update procedures

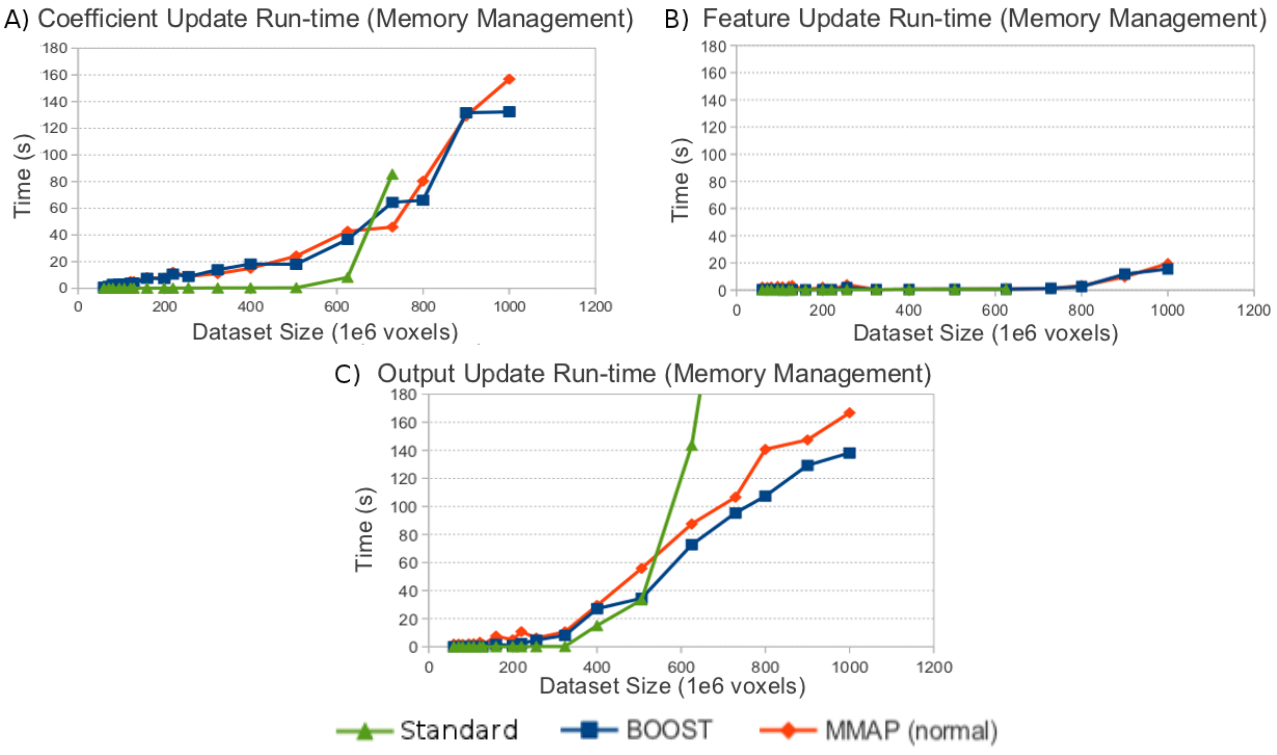


Figure 6.13: Run-times for the memory management implementations of the Floating Point Update procedures. Results are given for the Boost (blue line) and Mmap (red line) implementations of the Coefficient Update (A), Feature Update (B) and the Output Update (C) procedures. Results are compared against the standard implementation of the Floating Point Update (green line) procedures. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

The Update procedures in the *à trous* wavelet reconstruction algorithm only comprise an average of 2.4% of total run-time for in-core computation. However, run-time was shown (Section 6.3) to exponentially increase for partially out-of-core computation. Consequently, these procedures require that the disk access bottleneck is reduced in order to facilitate the computation of large data sets in practical time-frames. In this section we compare the performance of the standard Update procedures (Fig 6.13, green line) against the memory-managed Boost (blue line) and the Mmap (sequential) (red line) implementations. The DUCHAMP procedures results are not shown as performance is negligibly different from our standard Update procedures. Multithreaded implementations are not considered as preliminary testing indicated that memory-management was ill-suited to multi-core parallelism.

The Mmap and Boost implementations of the Coefficient Update procedure (Fig 6.13 A) achieve a computational performance 44.1× slower on average than the standard implementations for data sets smaller than 625 million voxels. This results from the relatively large data management overheads associated with memory management and the relatively large amount of memory transfer required for this computationally light process. However, for out-of-core data sets the run-time for both libraries increases linearly with data set size and surpasses the exponentially decreasing performance of the standard Coefficient Update at 729 million voxels (largest comparative data set). Larger data sets are not considered for comparison testing as procedures without memory management take an infeasibly large amount of run-time. The best performing memory management library alternates between data sets as a result of their unique paging schemes. The Boost and Mmap implementations' run-time continues to increase linearly with data set size up to the largest tested size of 1 billion voxels which indicates that slow disk access mitigation is successful to a large degree. However, memory-management only improves the disk access bottleneck and does not remove it completely. The best run-times achieved for 1 billion voxels (using Boost) are still 153.8× larger than the theoretical in-core computation time.

The standard Feature Update procedure's (Fig 6.13 B) performance is only affected slightly by out-of-core computation. The Feature Update procedure is immediately preceded by the Coefficient procedure, and both procedures access the same data structures. This increases the likelihood that a large portion of this data structure is already paged into memory and forgoes a large number of disk accesses. Similarly, the run-times for the memory-managed implementations are kept small but are still are on average 6.5× larger than the standard implementation up to 625 million voxels. For out-of-core data sets larger than 729 million voxels, performance of the memory-managed implementations decreases rapidly, dropping to 13.2× slower than a theoretical in-core implementation at 1 billion voxels. This only results in a negligibly small run-time of 20 seconds and consequently we consider memory management to be partially successful up to 729 million voxels.

The standard Output Update procedures (Fig 6.13 C) is disproportionately affected by out-of-core computation, and performance decreases exponentially for data set sizes larger than 324 million voxels. This is again in contrast to the Boost and Mmap implementations which show a linear increase in run-time for data sets larger than 324 million voxels. The standard implementations' performance is bypassed by both memory-managed implementations at approximately 506 million voxels. However, for this procedure, the Boost implementation performs on average 26% better than Mmap for out-of-core computation. This difference likely results from the differing paging schemes and the set of pages in memory at the start of this procedure. Boost achieves a 3.6× better performance than the standard implementation at 729 million voxels. The Boost and Mmap implementations' run-time continues to increase linearly with data set size up to the largest tested size of 1 billion voxels, mitigating the disk access bottleneck to a large degree. However, the performance of the memory-managed implementations is still

187.4× larger (using Boost) than the theoretical in-core computation time at 1 billion voxels.

6.3.3 Total run-times with Memory Management

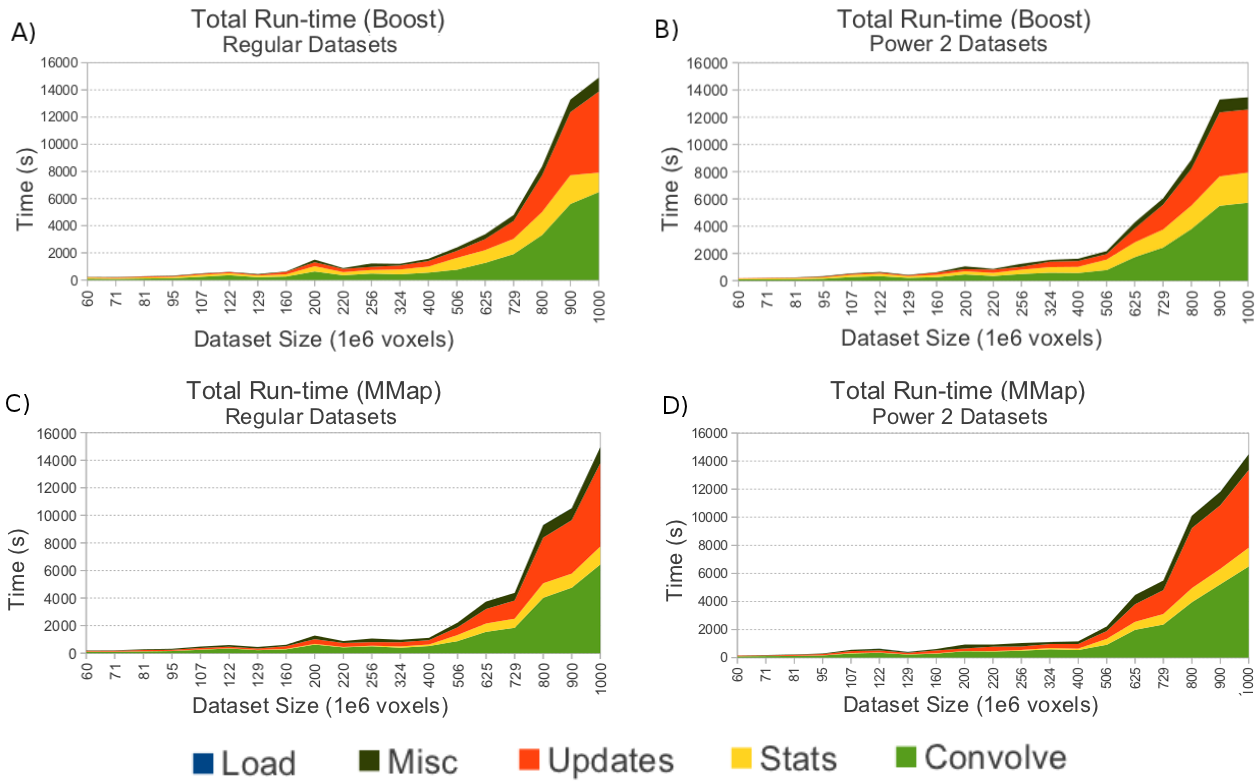


Figure 6.14: Total run-times for the memory-managed *à trous* wavelet reconstruction algorithm. These run-times are deconstructed into their component procedures to identify their specific contributions. Results are shown for the Boost (A & B) and Mmap (C & D) implementations for both Regular and Power 2 data sets. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

The Mmap and Boost memory-management solutions are not successful in completely mitigating the slow disk access bottleneck for all the procedures within the *à trous* wavelet reconstruction algorithm. These libraries do mitigate the exponential performance decrease for out-of-core computation and achieve significant performance increases relative to out-of-core procedures without memory management. However, the majority of procedures still produce run-times orders of magnitude larger than their respective theoretical in-core computation times. The main exception to this is the convolution procedure (the main contributor to run-time) where memory-management is largely successful in mitigating the disk access bottleneck. In this section, we discuss the performance (Fig 6.14) of the memory-managed convolution (green region), update (red region) and statistics (yellow region) procedures and their contribution to total run-time. The loading (blue region) and miscellaneous (brown region) procedures are not discussed as their contributions to total run-time are relatively small. Finally, we compare the overall performance of memory-managed solutions against the DUCHAMP implementation to assess whether memory management mitigation is sufficient to allow for scalable out-of-core computation. Only single-threaded solutions are discussed as multi-threaded solutions in conjunction with memory management result in extremely poor performance.

Statistic procedures are not considered for memory management as the random access memory pattern utilised by these procedures performed poorly with both the Mmap and Boost libraries.

However, the majority of the remaining *à trous* algorithms data structures implement memory management and are consequently stored mostly out-of-core when not in use. This provides a passive benefit to the statistical procedures' performance for data sets larger 523 million voxels as the data structure used for partial sorting is mostly in-core. This results in a performance increase of 3.8× and 5.64× at 729 million voxels for the Mmap (Fig 6.14, A & B) and Boost (Fig 6.14, C & D) implementations respectively. Consequently, the Mmap and Boost statistical procedures' contribution to total run-time is kept small, contributing an average of 12.3% and 22.2% respectively for out-of-core computation.

The convolution run-time is significantly reduced for larger Regular data sets in both the Boost (Fig 6.14, A) and the Mmap (Fig 6.14, C) implementations. The relatively high ratio of computation to disk access of separable filtering allowed asynchronous loading to reduce the disk access bottleneck and retain the performance contributions of our implemented improvements (to a large degree) when computing out-of-core. This results in a 2.3× performance improvement for both libraries at 729 million voxels, the largest data set used in comparison testing. The difference in run-time for Power 2 data sets (Fig 6.14, B and D) is small as the Updated Separable convolution procedure mitigates the majority of cache conflict stalls. The percentage contribution of all convolution procedures to total run-time for out-of-core computation is approximately equal for both of the memory management implementations, averaging 41% for data sets larger than 400 million voxels.

The disk access bottleneck is largely mitigated for out-of-core computation with the Mmap Feature Update (discussed above) and consequently this procedure's contribution to total run-time is negligibly small. However, the Mmap Coefficient and Output Update procedures, despite their simplicity, achieve run-times 153.8× and 187.4× larger than their respective theoretical in-core computed procedures. This causes the total run-time for all Mmap update procedures to be 170.2× larger than an equivalent in-core solution. Similar performance is achieved for most of the Boost implementations of these procedures. However, the Boost Output Update procedure performs on average 26% better than its Mmap counterpart which reduces the average total run-time contribution of the Boost procedures by 12.2%. Although these run-times are significantly better than standard out-of-core computation, they are still large enough to effectively double total run-time.

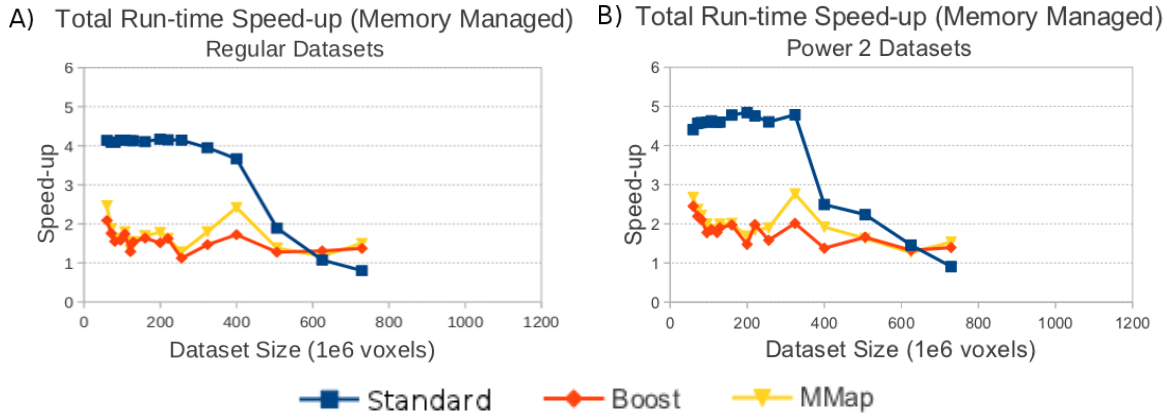


Figure 6.15: Total performance increases for the memory-managed *à trous* wavelet reconstruction algorithm. These run-times are deconstructed into their component procedures to identify their specific contributions. Results are shown for the Boost (red plot) and Mmap (yellow) implementations total performance increase relative to DUCHAMP for both Regular (A) and Power 2 (B) data sets. The results for the in-core performance of our improved *à trous* reconstruction implementation without memory management (blue plot) are shown for comparison. Testing was performed on a Intel i7-2600, 8GB RAM, 7200rpm hard-drive.

The graphs in Fig 6.15 show the overall performance increases achieved over DUCHAMP with memory management solutions to the out-of-core computation of the *à trous* wavelet reconstruction algorithm. The Mmap (red plot) and Boost (yellow plot) implementations are seen to perform similarly for all data sets except for the first partially out-of-core data sets of 324 and 400 million voxels where Mmap achieves a temporary performance increase of 31% and 38% over Boost for Regular and Power 2 data sets respectively. The in-core performance of both libraries averages 1.63× and 2× for Regular (Fig 6.15 A) and Power 2 (Fig 6.15 B) data sets respectively. This is substantially less than the in-core performance of the standard implementation (blue plot) which uses separable convolution techniques and SSE commands to achieve a 4.1× and 4.6× performance increase over DUCHAMP for Regular and Power 2 data sets respectively. This poor performance results from the overheads of creating and managing memory management data structures.

The advantages of memory management are seen for data sets larger than 625 million voxels where the performance of both the Boost and Mmap implementations exceed that of the single-threaded *à trous* algorithm (Update Separable convolution). Both memory-managed implementations are shown to mitigate the disk access penalty for large data sets, retaining an increased performance over DUCHAMP. However, performance for out-of-core computation decreases for both Boost and Mmap implementations, averaging 1.52× and 1.51× for the Regular and Power 2 data sets respectively. This poor performance is mainly attributed to the poor performance of the memory-managed Update procedures as both the convolution and statistics procedures achieved significant decreases to run-time. However, the relative performance increases are expected to rise for data sets larger than 729 million voxels. This increase would not be attributed to memory-management as run-time is shown to increase linearly in Fig 6.14 with an increase in data set size. Rather, the standard DUCHAMP performance is expected to exponentially drop with an increase in out-of-core computation, increasing the relative performance of Boost and Mmap.

The disk access bottleneck is not completely mitigated for any of the *à trous* wavelet reconstruction procedures. This limits attempts at multithreading solutions (results not shown) which in combination with slow disk access and limited physical memory results in thrashing and consequently poor performance. Therefore despite the large performance increases achieved for in-core computation with algorithm redesign and multi-core parallelism (13× performance increase), memory-management is insufficient in carrying over these performance gains to out-of-core computation.

6.3.4 Summary

We assessed both the Mmap and Boost memory management libraries on their ability to mitigate the slow disk access which results from out-of-core computation for large data sets. Both libraries performed well with the single-threaded Updated Separable convolution procedure and achieved linear performance scaling with increases in data set size. The convolution procedure's high ratio of computation to disk access allows memory mapping to asynchronously draw in memory and effectively mitigate the transfer bottleneck. However, the simple single-threaded Update procedures are completely transfer bound and run-time is significantly larger than an in-core solution despite the significant reduction to the disk access bottleneck. Additionally, the random memory access pattern of the Statistical procedures performs poorly with memory management. However, statistical procedures without memory management are computed mostly in-core when the majority of the *à trous* data structures implement memory management, significantly reducing run-time for out-of-core data sets. Multi-threaded solutions perform poorly

with memory management for all procedures and thread counts.

The slow out-of-core computation for the *à trous* wavelet reconstruction is only partially improved. Whilst memory management solutions allow performance to scale linearly with increase in data set size, performance is only 1.51-1.52× that of DUCHAMP single-threaded implementation. This could not be improved with parallelism and falls extremely short of the in-core 13× speed-up achieved with algorithmic improvement and parallelism on a quad-core CPU.

Chapter 7

Conclusions

The goal of this dissertation was to determine whether the computationally heavy *à trous* wavelet reconstruction algorithm, implemented in the DUCHAMP source extraction package, could be improved to facilitate source extraction for large scale HI surveys in practical time-frames on 'desktop' commodity hardware. High performance 'desktop' hardware was used to bypass the access constraints that can arise when performing personalised searches on clusters and large distributed computing solutions. Although 'desktop' computing is emphasised, our developed high performance system components are general enough to be accommodated into computing solutions. The memory management component, which allows large data computation on 'desktop' hardware, is intended for systems with insufficient fast-access memory and slow secondary storage. Testing was performed on a 'desktop' system with a quad-core Intel i7-2600 CPU, 8GB DDR3 RAM and 7200rpm hard-drive. Convolution performance testing was performed using the largest (and default) filter used in DUCHAMP, the $5 \times 5 \times 5$ B3-Spline filter.

We draw the conclusions of this thesis with respect to our specified research questions:

Can we improve the efficiency of the *à trous* wavelet reconstruction for a single core CPU?

Algorithmic efficiency was successfully improved in the *à trous* wavelet reconstruction algorithm by replacing the *à trous* algorithm's standard 3D convolution procedure ($\sim 95\%$ of total algorithm run-time) with our three Separable Filtering convolution procedures (which varied by memory access pattern only). The Updated Separable convolution maximised performance with a 4.5× average performance increase over DUCHAMP's 3D convolution for in-core computation. This performance resulted from the efficient cache use when propagating the row, column and spectral-aligned separable 1D filters linearly through memory in each of their respective filter passes. However, a 4.5× average performance increase is only 54% of the theoretical separable filtering performance ($5 \times 5 \times 5$ filter) as our implementation required three complete passes through the data (as opposed to one). Similar performance increases are expected with the application of this convolution optimisation technique in the general case of multi-dimensional filtering procedures using separable filters. Additionally, this algorithm significantly minimised the effect of frequent cache conflicts when computing Power 2 data sets (rows and columns are aligned to a multiple of page size). The number of cache conflicts for the Updated Separable algorithm was minimised as page-alignment only occurred between the elements of single filter response calculation and not between adjacent filter response calculations. In contrast, cache conflicts caused a significant drop in DUCHAMP convolution performance as its memory access pattern resulted in frequent page-aligned memory access and consequently numerous premature

cache evictions. This increased relative convolution performance to 5.2× that of DUCHAMP when computing Power 2 data sets.

The competing separable convolution methods, namely Original and Transposed Separable Filtering, perform relatively poorly as a result of the inefficient memory access patterns of at least one filter pass. Propagating the column (second pass) and spectral-aligned (third pass) 1D filters in the same direction as filter alignment (Original Separable Filtering method) resulted in large strides between memory accesses and non-optimal use of cache memory, and achieved a 3.7× performance increase only. The Transposed Separable Filtering method, which transposes the data and filters to ensure all memory reads are linear, was expected to achieve the highest performance increase. However, this algorithm's performance increase was limited to only 3.8× because of the extra transpose operations, and subsequent inefficient writes to memory, required to transpose the data between filter passes. Additionally, the memory access patterns of both convolution algorithms resulted in frequent page-aligned access for Power 2 data sets which resulted in highly variable, poor performance.

All Separable Filtering convolution methods are well-suited to high-accuracy source extraction problems, as floating point error is reduced by an order of magnitude relative to DUCHAMP 3D convolution. However, the 20% increase in memory allocation of Separable Filtering convolution limited the data set size which could be computed completely in-core, for which maximum performance was maintained, to only 1.2 GB (15% size of main memory, $\sim$ 90% main memory use). In contrast, the DUCHAMP performance drop-off only begins at a data set size of 1.5 GB (19% size of main memory). Nonetheless, the performance of all the Separable Filtering convolutions was still superior to DUCHAMP convolution, despite partially out-of-core computation, up to a data set size of 2.3 GB (29% size of main memory, 179% allocated memory). However, with further increases to the amount of out-of-core computation performed, run-time for both the DUCHAMP and Separable Filtering convolutions exponentially increased with the number of disk accesses required.

The algorithm efficiency and accuracy of the *à trous* reconstruction algorithm's convolution was successfully improved with our Updated Separable Filtering algorithm at the expense of a 20% increase in memory use.

Can Intel CPU SSE commands facilitate SIMD execution in this algorithm and further increase performance for the single-threaded case?

The introduction of SSE commands to facilitate SIMD execution (vector instruction parallelism) to further improve performance for the single-threaded case was largely unsuccessful. The majority of the procedures within the *à trous* wavelet reconstruction algorithm contains execution branching which prevented efficient SSE implementation; only the Update procedures were suitable for SSE parallelism. However, these procedures had a light computational load per unit of memory transfer which limited the SSE performance increase to between 1.1-1.3×, dropping to 0.5-1.2× for out-of-core data sets.

Overall single-threaded performance of the improved *à trous* wavelet reconstruction (using both Separable Filtering convolution and SSE Update procedures) achieved a performance increase of 4.1× and 4.6× for Regular and Power 2 in-core data sets respectively. SSE's contribution to this performance increase is small and should be considered a fine-tuning optimisation only. Whilst the single-threaded *à trous* wavelet reconstruction performance increases achieved are significant, they are insufficient to compute ultra-wide and ultra-deep HI surveys in practical time-frames. However, further performance increases are obtained by incorporating parallel computing, discussed below. Additionally, although not covered in this thesis, these algorithms are expected to be easily included (if necessary) in distributed computing solutions to

DUCHAMP source extraction.

Can we accelerate these processes by utilising parallel 'desktop' multi-core CPU hardware?

The *à trous* wavelet reconstruction algorithm is well-suited to multi-core CPU parallelism. All implemented Separable Filtering convolution algorithms scaled well with an increase in the number of threads and logical cores used (quad-core CPU). However, the performance of the multithreaded Original and Transposed Separable solutions exhibited unstable performance with performance drops occurring for certain data set sizes. The Updated Separable convolution algorithm exhibited a uniform performance increase and near optimal scaling with an increase in the physical cores utilised and hyper-threading for all in-core Regular and Power 2 data sets. Relative to the single-threaded DUCHAMP implementation, the Updated Separable convolution achieved a 9×, 17× and 22.3× performance increase for 2, 4 and 8-threaded implementation respectively for in-core regular data sets on a quad-core CPU. The poor performance of the DUCHAMP convolution for Power 2 data sets increased the relative performance increase of the Updated Separable convolution to 10.7×, 20.2× and 24× for 2, 4 and 8-threaded implementation respectively. The 22.3×-24× convolution performance increase achieved with 8-threads on a 4-core CPU is substantial with the potential of reducing convolution computation time from days to hours. However, this performance is limited to systems with sufficient physical memory space (6× the size of the computed data set) as even partially out-of-core problems decreases performance significantly as a result of slow disk access. Additionally, multithreading exacerbates this situation as the rate of disk access increases with thread count.

The Update procedure's SSE implementations are not suitable for multi-core CPU parallelism. Update procedures achieved the greatest performance improvements with multithreading alone. Update procedures that accessed two data structures per voxel were limited to a maximum parallel performance increase between 1.15-1.4× despite using 8-threads. Update procedures that accessed one data set (which are in the minority) for the majority of computation (second data structure access conditional), by contrast, scaled with the number of threads and achieved a 3.5× performance increase with 8-threads.

Overall *à trous* wavelet reconstruction performance with both algorithmic improvement and multi-core parallelism (8-threads) achieved a 13× (13.8× for Power 2 data sets) performance increase over the standard DUCHAMP implementation. Performance was restricted significantly by the relatively large run-time contribution of the *à trous* statistical procedures which were not considered for optimisation in this work. However, this performance increase still represents an order of magnitude increase over the single-threaded DUCHAMP implementation, which is a significant performance improvement to the largest component (with respect to run-time) in the DUCHAMP source extraction package. On a computing system with sufficiently large memory space (6× the size of the computed data set), large scale HI survey data could be computed in a practical time-frame, reducing days of computation to mere hours. Similarly, the in-core processing time of smaller data volumes (in the 0.1 - 10 GB range) is reduced to the order of minutes. The *à trous* wavelet reconstruction algorithm can be further improved with a multi-core solution for statistical procedures and higher end 'desktop' hardware with more available CPU cores. However, to meet the goals of this research, it was necessary to maintain high performance for out-of-core computation to account for the relatively small amount of physical memory on 'desktop' systems.

Can slow disk access on 'desktop' hardware be mitigated with memory management to allow for efficient computation of large data sets?

The disk access bottleneck which limits out-of-core performance was only partially mitigated

with the implementation of our memory management solutions. Although the Boost and Mmap memory management solution showed approximately equal performance, Boost and Mmap performed marginally better on Power 2 and Regular data sets respectively. Slow disk access was mitigated to a large degree for procedures with high computational intensities per unit of memory paged as it allowed enough time for a substantial amount of asynchronous paging (prefetching) of data to take place. This enabled high performance to be partially maintained for single-threaded Updated Separable convolutions with a uniform 3.6× performance increase relative to DUCHAMP up to the largest tested data set of 3.7 GB (memory use 22 GB). Additionally, run-time increased linearly with data set size, indicating that the performance achieved will likely be maintained for even larger data sets. However, the out-of-core multithreading implementation of Updated Separable convolution was significantly hindered by parallel access to disk with performance decreasing rapidly with higher thread counts and larger data sets. The lack of efficient parallelism limited memory managed convolution to only a small fraction of in-core performance.

The memory managed, computationally light Update procedures, despite having superior performance to out-of-core computation without memory management, experienced an order of magnitude decrease in performance for both single and multithreaded (to a greater extent) implementations. Additionally, the random access memory pattern of statistical procedures with memory management resulted in an exponential increase in its run-time. Statistical procedures were slightly improved by defining their data structures in-core without memory management. Poor multithreading performance and the exponential increase in run-time for both Update and Statistical procedures reduced the overall performance of our *à trous* wavelet reconstruction over DUCHAMP from 13× (in-core) to only 1.51× (out-of-core). Memory mapping solutions in isolation were insufficient in mitigating the disk access bottleneck in order to facilitate the processing of large HI observational data sets on 'desktop' hardware in practical amounts of time. However, we note that slow out-of-core computation may be improved with solutions that segment large data cubes into manageable quantities or through the implementation of faster secondary storage such as parallel RAID access or Solid State Drives (SSD).

In conclusion, the implementation of multi-core parallelism with 8-threads on a 4-core CPU in conjunction with separable filtering techniques significantly improves the performance of DUCHAMP's *à trous* wavelet reconstruction algorithm by 13× for in-core data sets. This performance improvement reduces the run-time required to perform DUCHAMP noise suppression on HI data volumes by an order of magnitude and significantly reduces total DUCHAMP run-time. Parallel convolution performance is expected to linearly scale (approximately) with the number of CPU cores available and will significantly increase the overall *à trous* wavelet reconstruction algorithm's parallel performance. However, this performance will be limited by the current lack of a multithreaded solution for *à trous* Statistical procedures. SSE intrinsics are a fine-tuning operation for the *à trous* algorithm's single-threaded case only and will produce negative results when combined with multi-core parallelism.

'Desktop' hardware has inadequate memory resources to process large HI data sets completely in-core and requires efficient out-of-core computation to maintain the achieved in-core performance increases. However, memory management with magnetic drives as secondary storage was insufficient to facilitate efficient out-of-core computation. Faster secondary storage such as parallel RAID access or Solid State Drives (SSD) are required to allow 'desktop' hardware to process large HI data sets in practical time-frames.

The memory management component was specifically intended for systems with insufficient fast-access memory and slow secondary storage. Although 'desktop' computing is emphasised, our implemented high performance system components are general enough to be accommodated into larger parallel computing hardware solutions such as clusters.

7.1 Future work

The potential of high performance source extraction on 'desktop' hardware is extensive and this dissertation only represents a fraction of the work that can be done.

'Desktop' hardware optimisations have not been exhausted. Low level cache optimisation of all *à trous* reconstruction procedures and parallel implementations of the statistics procedure could further improve performance. Memory use in the Separable Filtering implementation of *à trous* wavelet reconstruction can be reduced when implementing robust statistics as the temporary data structures used to sort data could be used to store the intermediate values generated between separable filter passes. This will increase the data set size that can be computed efficiently in-core.

Faster forms of secondary storage can be implemented to further reduce (or potentially eliminate) the problem of slow disk access. Two main affordable commodity options exist, namely parallel disk access with RAID and the use of solid state drives (SSD). Additional hardware solutions to further increase performance include the GPU, which is well-suited to the problem of lightweight thread-level parallelism. The implementation of an asynchronous CPU-GPU solution has the potential to dramatically increase the performance of *à trous* wavelet reconstruction.

Finally, we note that optimisation must be extended to the entirety of the DUCHAMP package in order for it to be an efficient source extraction tool for use with the next generation of ultra-deep and ultra-wide HI surveys.

Bibliography

[1] Abdalla, F., and Rawlings, S. Probing dark energy with baryonic oscillations and future radio surveys of neutral hydrogen. *Monthly Notices of the Royal Astronomical Society 360*, 1 (2005), 27–40.

[2] Adams, B. The 21 cm line: Why observe it? `https://alfalfasurvey.wordpress.com/2009/01/17/the-21-cm-line-why-do-we-care/`, 2009. Accessed 2014-03-24.

[3] Agulleiro, J., Garzón, E., Fernández, J., et al. Vectorization with SIMD extensions speeds up reconstruction in electron tomography. *Journal of Structural Biology 170*, 3 (2010), 570–575.

[4] Akhter, S., and Roberts, J. *Multi-core programming*, vol. 33. Intel press Hillsboro, 2006.

[5] AMD Staff. AMD64 Architecture Programmer's Manual. volume 3. `http://www.boost.org/doc/libs/1_43_0/doc/html/interprocess.html`, Sept 2007 revision 3.14.

[6] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2012.

[7] Baan, W. A. RFI mitigation in Radio Astronomy. In *Proceedings of RFI mitigation workshop PoS(RFI2010)* (2010), vol. 1, p. 32.

[8] Barnes, D., Staveley-Smith, L., De Blok, W. e. . a., Oosterloo, T., Stewart, I., Wright, A., Banks, G., Bhathal, R., Boyce, P., Calabretta, M., et al. The HI Parkes All Sky Survey: Southern observations, calibration and robust imaging. *Monthly Notices of the Royal Astronomical Society 322*, 3 (2001), 486–498.

[9] Benjamini, Y., and Hochberg, Y. Controlling the False Discovery Rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological) 57* (1995), 289–300.

[10] Bernabé, G., García, J. M., and González, J. Reducing 3D wavelet transform execution time through the streaming SIMD extensions. In *Proceedings of Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing* (2003), IEEE, pp. 49–56.

[11] Bernabé, G., García, J. M., and González, J. Reducing 3D fast wavelet transform execution time using blocking and the streaming SIMD extensions. *Journal of VLSI signal processing systems for signal, image and video technology 41*, 2 (2005), 209–223.

[12] Bingmann, T. Stxxl 1.4-dev. `http://stxxl.sourceforge.net/tags/master/index.html`, Dec. 2013-2014. Accessed 2014-10-22.

[13] Boyce, P. J. GammaFinder: A Java application to find galaxies in astronomical spectral line data cubes. Master's thesis, School of Computer Science. Cardiff University, 2003.

[14] Briggs, F. H. Neutral Hydrogen in the Universe. In *Proceedings of The New Cosmology* (June 2005), M. Colless, Ed., pp. 147–164.

[15] Calabretta, M. WCSLIB and PGSBOX. *Astrophysics Source Code Library 1* (2011), ascl:1108.003.

[16] Carilli, C., and Rawlings, S. Science with the Square Kilometer Array: motivation, key science projects, standards and assumptions. *New Astronomy Reviews 48* (2004), 979–984.

[17] Cepeda, S. Optimization and Performance Tuning for Intel Xeon Phi Co-processors, Part 2: Understanding and Using Hardware Events. `http://goo.gl/D4TS9Q`, 2012. Accessed 2014-03-20.

[18] Chapman, B., Jost, G., and Van Der Pas, R. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008.

[19] Dagum, L., and Menon, R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE 5*, 1 (1998), 46–55.

[20] Davidson, D. B. MeerKAT and SKA phase 1. In *Proceedings of 10th International Symposium on Antennas, Propagation & EM Theory (ISAPE)* (2012), IEEE, pp. 1279–1282.

[21] Davidson, D. B. Potential technological spin-offs from MeerKAT and the South African Square Kilometre Array bid. *South African Journal of Science 108*, 1-2 (2012), 1–3.

[22] DeBoer, D. R., Gough, R. G., Bunton, J. D., Cornwell, T. J., Beresford, R. J., Johnston, S., Feain, I. J., Schinckel, A. E., Jackson, C. A., Kesteven, M. J., et al. Australian SKA pathfinder: A high-dynamic range wide-field of view survey telescope. *Proceedings of the IEEE 97*, 8 (2009), 1507–1521.

[23] Delhaize, J., Meyer, M., Staveley-Smith, L., and Boyle, B. Detection of HI in distant galaxies using spectral stacking. *Monthly Notices of the Royal Astronomical Society 433* (2013), 1398–1410.

[24] Dementiev, R. Stxxl 1.4.0 Tutorial. `http://stxxl.sourceforge.net/tags/1.4.0/tutorial.html`, 2013. Accessed 2014-03-20.

[25] Dementiev, R., Kettner, L., and Sanders, P. Stxxl: Standard Template Library for XXL Data Sets. *ESA'05 Proceedings of 13th annual European conference on Algorithms* (2005), 640–651.

[26] die.net Staff. gettimeofday(2) Man-pages. `http://linux.die.net/man/2/gettimeofday`, 2014. Accessed 2014-10-22.

[27] die.net Staff. time(2) Man-pages. `http://linux.die.net/man/2/time`, 2014. Accessed 2014-10-22.

[28] Dong, F., Pierpaoli, E., Gunn, J. E., and Wechsler, R. H. Optical Cluster Finding with an Adaptive Matched-Filter Technique: Algorithm and Comparison with Simulations. *The Astrophysical Journal 676*, 2 (2008), 868–879.

[29] Doyle, M. T., Drinkwater, M., Rohde, D., Pimbblet, K., Read, M., Meyer, M., Zwaan, M., Ryan-Weber, E., Stevens, J., Koribalski, B., et al. The HIPASS catalogue–III. Optical counterparts and isolated dark galaxies. *Monthly Notices of the Royal Astronomical Society 361*, 1 (2005), 34–44.

[30] Duc, P.-A., Brinks, E., Springel, V., Pichardo, B., Weilbacher, P., and Mirabel, I. Formation of a tidal dwarf galaxy in the interacting system Arp 245 (NGC 2992/93). *The Astronomical Journal 120*, 3 (2000), 1238–1264.

[31] Dudgeon, D. E., and Mersereau, R. M. *Multidimensional Digital Signal Processing*. Prentice Hall Professional Technical Reference, 1990.

[32] Duffy, A. R., Meyer, M. J., Staveley-Smith, L., Bernyk, M., Croton, D. J., Koribalski, B. S., Gerstmann, D., and Westerlund, S. Predictions for ASKAP neutral hydrogen surveys. *Monthly Notices of the Royal Astronomical Society 426*, 4 (2012), 3385–3402.

[33] Flöer, L., and Winkel, B. 2D–1D Wavelet Reconstruction as a Tool for Source Finding in Spectroscopic Imaging Surveys. *Publications of the Astronomical Society of Australia 29*, 3 (2012), 244–250.

[34] Fog, A. Optimizing software in C++ : An optimization guide for Windows, Linux and Mac platforms. `http://www.agner.org/optimize/`, Aug 2014. Accessed 2015-01-20.

[35] Frey, S., and Mosoni, L. A short introduction to radio interferometric image reconstruction. *New Astronomy Reviews 53*, 11 (2009), 307–311.

[36] Fridman, P., and Baan, W. RFI mitigation methods in radio astronomy. *Astronomy and Astrophysics 378*, 1 (2001), 327–344.

[37] Gaztañaga, I. Boost (C++ libraries): Boost.Interprocess. `http://www.boost.org/doc/libs/1_43_0/doc/html/interprocess.html`, Last revised: May 02, 2010. Accessed 2014-01-14.

[38] Gerber, R., Bik, A., Smith, K., and Tian, X. *The Software Optimization Cookbook Second Edition. High Performance Recipes for IA 32 Platforms*. Intel Press, 2005.

[39] Giovanelli, R., Haynes, M. P., Kent, B. R., Perillat, P., Saintonge, A., Brosch, N., Catinella, B., Hoffman, G. L., Stierwalt, S., Spekkens, K., et al. The Arecibo Legacy Fast ALFA Survey. I. Science goals, survey design, and strategy. *The Astronomical Journal 130*, 6 (2005), 2598–2612.

[40] Giovanelli, R., Haynes, M. P., Kent, B. R., Saintonge, A., Stierwalt, S., Altaf, A., Balonek, T., Brosch, N., Brown, S., Catinella, B., et al. The Arecibo Legacy Fast ALFA Survey. III. HI source catalog of the northern virgo cluster region. *The Astronomical Journal 133*, 6 (2007), 2569–2583.

[41] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR) 23*, 1 (1991), 5–48.

[42] Hanisch, R., Farris, A., Greisen, E., Pence, W., Schlesinger, B., Teuben, P., Thompson, R., and Warnock, A. Definition of the flexible image transport system (FITS). *Astronomy and Astrophysics 376* (2001), 359–380.

[43] Hassaballah, M., Omran, S., and Mahdy, Y. B. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal 51*, 6 (2008), 630–649.

[44] Hassan, A., Fluke, C. J., and Barnes, D. G. Interactive visualization of the largest radioastronomy cubes. *New Astronomy 16*, 2 (2011), 100–109.

[45] Haynes, M. P., Giovanelli, R., Martin, A. M., Hess, K. M., Saintonge, A., Adams, E. A., Hallenbeck, G., Hoffman, G. L., Huang, S., Kent, B. R., et al.

The Arecibo Legacy Fast ALFA Survey: The α. 40 HI source catalog, its characteristics and their impact on the derivation of the HI Mass Function. *The Astronomical Journal 142*, 5 (2011), 170.

[46] Holwerda, B., and Blyth, S. Trumpeting the Vuvuzela: The deepest HI observations with MeerKAT. In *Proceedings of ISKAF2010 Science Meeting* (2010), vol. 1, p. 68.

[47] Holwerda, B., Blyth, S.-L., and Baker, A. Looking at the distant universe with the MeerKAT Array (LADUMA). *Proceedings of International Astronomical Union 7*, S284 (2011), 496–499.

[48] Hong, C., Chen, D., Chen, W., Zheng, W., and Lin, H. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of 19th international conference on Parallel architectures and compilation techniques* (2010), ACM, pp. 217–226.

[49] Huang, S., Haynes, M. P., Giovanelli, R., and Brinchmann, J. The Arecibo Legacy Fast ALFA Survey: The Galaxy Population Detected by ALFALFA. *The Astrophysical Journal 756*, 2 (2012), 113.

[50] Huynh, M., Hopkins, A., Norris, R., Hancock, P., Murphy, T., Jurek, R., and Whiting, M. The completeness and reliability of threshold and false-discovery rate source extraction algorithms for compact continuum sources. *Publications of the Astronomical Society of Australia 29*, 3 (2011), 229–243.

[51] Ishizaka, K., Obata, M., and Kasahara, H. Cache optimization for coarse grain task parallel processing using inter-array padding. In *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed., vol. 2958 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 64–76.

[52] Jurek, R. The Characterised Noise HI source finder: Detecting HI galaxies using a novel implementation of matched filtering. *Publications of the Astronomical Society of Australia 29*, 3 (2012), 251–261.

[53] Kanekar, N., and Briggs, F. 21-cm absorption studies with the Square Kilometer Array. *New Astronomy Reviews 48*, 11 (2004), 1259–1270.

[54] Kerrisk, M. Madvise(2) Man-pages. `http://man7.org/linux/man-pages/man2/madvise.2.html`, 2014. Accessed 2014-10-29.

[55] Kerrisk, M. Mmap(2) Man-pages. `http://man7.org/linux/man-pages/man2/mmap.2.html`, 2014. Accessed 2014-03-20.

[56] Kilborn, V. The HI Jodrell All-Sky Survey (HIJASS). In *Proceedings of Seeing Through the Dust: The Detection of HI and the Exploration of the ISM in Galaxies* (2002), vol. 276 of *Astronomical Society of the Pacific Conference Series*, p. 80.

[57] Kocz, J., Briggs, F., and Reynolds, J. Spatial filtering using a multibeam receiver. In *Proceedings of RFI Mitigation Workshop PoS(RFI2010) proceedings* (2010), vol. 1, p. 32.

[58] Koribalski, B. S. Source Finding and Visualisation. *Publications of the Astronomical Society of Australia 29*, 3 (2012), 213–213.

[59] Kosec, G., Depolli, M., Rashkovska, A., and Trobec, R. Super linear speedup in a local parallel meshless solution of thermo-fluid problems. *Computers & Structures 133* (2014), 30–38.

[60] Lang, R. H., Boyce, P. J., Kilborn, V. A., Minchin, R. F., Disney, M. J., Jordan, C. A., Grossi, M., Garcia, D. A., Freeman, K. C., Phillipps, S., et al. First results from the HI Jodrell All Sky Survey: inclination-dependent selection effects in a 21-cm blind survey. *Monthly Notices of the Royal Astronomical Society 342*, 3 (2003), 738–758.

[61] Lay, O. P., and Halverson, N. W. The impact of atmospheric fluctuations on degree-scale imaging of the cosmic microwave background. *The Astrophysical Journal 543*, 2 (2000), 787–798.

[62] Lewis, B., and Berg, D. J. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., 1998.

[63] Love, R. *Linux System Programming: Talking directly to the kernel and C library*. O'Reilly Media, 2013.

[64] Lutz, R. An algorithm for the real time analysis of digitised images. *The Computer Journal 23*, 3 (1980), 262–269.

[65] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal 6*, 1 (2002), 4–15.

[66] Martin, A. M., Papastergis, E., Giovanelli, R., Haynes, M. P., Springob, C. M., and Stierwalt, S. The Arecibo Legacy Fast ALFA Survey. X. The HI Mass Function and from the 40% ALFALFA survey. *The Astrophysical Journal 723*, 2 (2010), 1359–1374.

[67] Melin, J.-B., Bartlett, J., and Delabrouille, J. Catalog extraction in sz cluster surveys: a matched filter approach. *Astronomy and Astrophysics 459* (2006), 341–352.

[68] Meyer, M., Zwaan, M., Webster, R., Staveley-Smith, L., Ryan-Weber, E., Drinkwater, M., Barnes, D., Howlett, M., Kilborn, V., Stevens, J., et al. The HIPASS catalogue–I. Data presentation. *Monthly Notices of the Royal Astronomical Society 350*, 4 (2004), 1195–1209.

[69] Miller, C. J., Genovese, C., Nichol, R. C., Wasserman, L., Connolly, A., Reichart, D., Hopkins, A., Schneider, J., and Moore, A. Controlling the False-Discovery Rate in astrophysical data analysis. *The Astronomical Journal 122*, 6 (2001), 3492–3505.

[70] Napier, P. The primary antenna elements. In *Proceedings of Synthesis Imaging in Radio Astronomy II* (1999), vol. 180 of *Astronomical Society of the Pacific Conference Series*, pp. 37–56.

[71] Narendra, P. M. A separable median filter for image noise smoothing. *IEEE Transactions on Pattern Analysis and Machine Intelligence 3* (1981), 20–29.

[72] Nummelin, A. Observations of interstellar molecules. `http://heasarc.gsfc.nasa.gov/docs/software/fitsio/c/c_user/cfitsio.html`, 2007. Accessed 2014-12-1.

[73] Patterson, D. A., and Hennessy, J. L. *Computer organization and design, Third Edition: The Hardware/Software Interface*. Elsevier, 2007.

[74] Patterson, D. A., and Hennessy, J. L. *Computer organization and design: The hardware/software interface*. Newnes, 2013.

[75] Pearson, T., and Readhead, A. Image formation by self-calibration in radio astronomy. *Annual review of astronomy and astrophysics 22* (1984), 97–130.

[76] Pence, W. CFITSIO, v2. 0: a new full-featured data interface. In *Proceedings of Astronomical Data Analysis Software and Systems VIII* (1999), vol. 172 of *Astronomical Society of the Pacific Conference Series*, pp. 487–489.

[77] Pence, W. CFITSIO User's Reference Guide: An Interface to FITS Format Files for C Programmers. Version 3.2. `http://heasarc.gsfc.nasa.gov/docs/software/fitsio/c/c_user/cfitsio.html`, 2010. Accessed 2011-04-20.

[78] Pence, W. D. CFITSIO: A FITS File Subroutine Library. Astrophysics Source Code Library, Oct. 2010.

[79] Pence, W. D., Chiappetti, L., Page, C. G., Shaw, R., and Stobie, E. Definition of the Flexible Image Transport System (FITS), version 3.0. *Astronomy and Astrophysics 524* (2010), A42.

[80] Pham-Gia, T., and Hung, T. The mean and median absolute deviations. *Mathematical and Computer Modelling 34*, 7 (2001), 921–936.

[81] Popping, A., Jurek, R., Westmeier, T., Serra, P., Flöer, L., Meyer, M., and Koribalski, B. Comparison of potential ASKAP HI survey source finders. *Publications of the Astronomical Society of Australia 29*, 03 (2012), 318–339.

[82] Randy Kath, M. D. N. T. G. Managing Memory-Mapped Files. `http://msdn.microsoft.com/en-us/library/ms810613.aspx`, 1993. Accessed 2011-04-30.

[83] Rivera, G., and Tseng, C.-W. Compiler optimizations for eliminating cache conflict misses. Tech. Rep. CS-TR-3819, Dept. of Computer Science, University of Maryland, 1998.

[84] Saintonge, A. The Arecibo Legacy Fast ALFA Survey. IV. Strategies for Signal Identification and Survey Catalog Reliability. *The Astronomical Journal 133*, 5 (2007), 2087–2098.

[85] Schaubert, D., Boryssenko, A., Van Ardenne, A., de Vaate, J. B., and Craeye, C. The Square Kilometer Array (SKA) antenna. In *Proceedings of IEEE International Symposium on Phased Array Systems and Technology, 2003.* (2003), IEEE, pp. 351–358.

[86] Shensa, M. The Discrete Wavelet Transform: wedding the A'Trous and Mallat algorithms. *IEEE Transactions on Signal Processing 40*, 10 (1992), 2464–2482.

[87] Smith, S. W. *Digital Signal Processing: A practical guide for engineers and scientists.* Newnes, Boston, 2003.

[88] Spoelstra, T. T. The influence of ionospheric refraction on radio astronomy interferometry. *Astronomy and Astrophysics 120* (1983), 313–321.

[89] Springob, C. M., Haynes, M. P., and Giovanelli, R. Morphology, environment, and the HI mass function. *The Astrophysical Journal 621*, 1 (2005), 215–226.

[90] Stamatis, D. *Essential statistical concepts for the quality professional.* CRC Press, 2012.

[91] Stefánsson, A., Končar, N., and Jones, A. J. A note on the Gamma test. *Neural Computing & Applications 5*, 3 (1997), 131–133.

[92] Thomas, B., Jenness, T., Economou, F., Greenfield, P., Hirst, P., Berry, D., Bray, E., Gray, N., Muna, D., Turner, J., et al. Significant Problems in FITS Limit Its Use in Modern Astronomical Research. In *Proceedings of Astronomical Data Analysis Software and Systems XXIII* (2014), vol. 485 of *Astronomical Society of the Pacific Conference Series*, pp. 351–354.

[93] Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H., and Su, E. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal 6*, 1 (2002), 36–46.

[94] Tompkins, W. J. Biomedical digital signal processing. *Prentice Hall* (1993).

[95] Van Der Tol, S., Jeffs, B. D., and van der Veen, A.-J. Self-calibration for the LOFAR radio astronomical array. *Signal Processing, IEEE Transactions on 55*, 9 (2007), 4497–4510.

[96] Vera, X., Llosa, J., and González, A. Near-optimal padding for removing conflict misses. In *Languages and Compilers for Parallel Computing*. Springer, 2005, pp. 329–343.

[97] Walter, F., Brinks, E., De Blok, W., Bigiel, F., Kennicutt Jr, R. C., Thornley, M. D., and Leroy, A. Things: The hi nearby galaxy survey. *The Astronomical Journal 136*, 6 (2008), 2563.

[98] Wells, D., Greisen, E., and Harten, R. FITS-A flexible image transport system. *Astronomy and Astrophysics Supplement Series 44* (1981), 363–370.

[99] Wells, D. C. The VLBA correlator—Real-Time in the Distributed ERA. In *Astronomical Data Analysis Software and Systems II* (1993), vol. 52, pp. 267–276.

[100] Welsh, M. *Running Linux*. O'Reilly Media, Inc., 2003.

[101] Westerlund, S. Analysis of the parallelisation of the DUCHAMP algorithm. Tech. rep., iVec Research Internships, International Centre for Radio Astronomy Research (ICRAR), 2009. Available at `http://www.icrar.org/__data/assets/pdf_file/0006/1750866/stefan_westerlund_ivec_report.pdf`.

[102] Westmeier, T., Popping, A., and Serra, P. Basic Testing of the Duchamp Source Finder. *Publications of the Astronomical Society of Australia 29*, 3 (2012), 276–295.

[103] Whiting, M. Source Detection with DUCHAMP: A User's Guide. Tech. rep., 2010. Available at `http://www.atnf.csiro.au/people/Matthew.Whiting/Duchamp/downloads.php`.

[104] Whiting, M. T. DUCHAMP: a 3D source finder for spectral-line data. *Monthly Notices of the Royal Astronomical Society 421*, 4 (2012), 3242–3256.

[105] Wiaux, Y., Puy, G., Boursier, Y., and Vandergheynst, P. Spread spectrum for imaging techniques in radio interferometry. *Monthly Notices of the Royal Astronomical Society 400*, 2 (2009), 1029–1038.

[106] Wilson, T. L., Rohlfs, K., and Hüttemeister, S. *Tools of Radio Astronomy*, vol. 86. Springer, 2009.

[107] Wong, O., Ryan-Weber, E., Garcia-Appadoo, D., Webster, R., Staveley-Smith, L., Zwaan, M., Meyer, M., Barnes, D., Kilborn, V., Bhathal, R., et al. The Northern HIPASS catalogue–data presentation, completeness and reliability measures. *Monthly Notices of the Royal Astronomical Society 371*, 4 (2006), 1855–1864.

[108] Zhang, B., Fadili, J. M., and Starck, J.-L. Wavelets, ridgelets, and curvelets for Poisson noise removal. *IEEE Transactions on Image Processing 17*, 7 (2008), 1093–1108.

[109] Zwaan, M., Meyer, M., Webster, R., Staveley-Smith, L., Drinkwater, M., Barnes, D., Bhathal, R., De Blok, W., Disney, M., Ekers, R., et al. The HIPASS catalogue–II. Completeness, reliability and parameter accuracy. *Monthly Notices of the Royal Astronomical Society 350*, 4 (2004), 1210–1219.

[110] Zwaan, M., Staveley-Smith, L., Koribalski, B., Henning, P., Kilborn, V., Ryder, S., Barnes, D., Bhathal, R., Boyce, P., de Blok, W., et al. The 1000 Brightest HIPASS Galaxies: The HI Mass Function and OmegaHI. *The Astronomical Journal 125*, 6 (2003), 2842–2858.