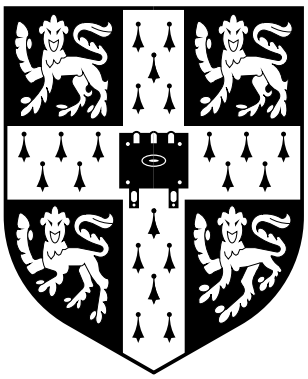
Parallelisation of greedy algorithms for compressive sensing reconstruction

David William Turner



Department of Computer Science and Technology
University of Cambridge
Selwyn College

September 2018

This dissertation is submitted for the degree of Doctor of Philosophy.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.

David Turner
September 2018

Parallelisation of greedy algorithms for compressive sensing reconstruction

David William Turner

Summary

Compressive Sensing (CS) is a technique which allows a signal to be compressed at the same time as it is captured. The process of capturing and simultaneously compressing the signal is represented as linear sampling, which can encompass a variety of physical processes or signal processing. Instead of explicitly identifying redundancies in the source signal, CS relies on the property of sparsity in order to reconstruct the compressed signal. While linear sampling is much less burdensome than conventional compression, this is more than made up for by the high computational cost of reconstructing a signal which has been captured using CS. Even when using some of the fastest reconstruction techniques, known as greedy pursuits, reconstruction of large problems can pose a significant burden, consuming a great deal of memory as well as compute time.

Parallel computing is the foundation of the field of High Performance Computing (HPC). Modern supercomputers are generally composed of large clusters of standard servers, with a dedicated low-latency high-bandwidth interconnect network. On such a cluster, an appropriately written program can harness vast quantities of memory and computational power. However, in order to exploit a parallel compute resource, an algorithm usually has to be redesigned from the ground up. In this thesis I describe the development of parallel variants of two algorithms commonly used in CS reconstruction, Matching Pursuit (MP) and Orthogonal Matching Pursuit (OMP), resulting in the new distributed compute algorithms DistMP and DistOMP. I present the results from experiments showing how DistMP and DistOMP can utilise a compute cluster to solve CS problems much more quickly than a single computer could alone. Speed-up of as much as a factor of 76 is observed with DistMP when utilising 210 workers across 14 servers, compared to a single worker. Finally, I demonstrate how DistOMP can solve a problem with a 429GB equivalent sampling matrix in as little as 62 minutes using a 16-node compute cluster.

Acknowledgements

This work was funded under an ICASE award from the Engineering and Physical Sciences Research Council, with sponsorship provided by Thales Research and Technology (TRT).

First, I would like to express my gratitude to my supervisor, Ian Wassell, for giving me the opportunity to undertake this PhD, for allowing me the freedom to find a topic I could call my own (and develop many skills along the way), for bearing with me as my PhD has taken rather longer than expected, and for all his hard work in helping me write and edit this thesis. My thanks also go to Richard Egan, Mike Newman, and Chris Firth at TRT for hosting my internship as well as for support, encouragement, and engaging discussions.

I would like to thank my research group, the Digital Technology Group, for being such a great place to work for four years. In particular, thanks to Tom for the friendship and countless caffeinated beverages which have so helped this thesis along. I am very grateful to everybody at my employer, Argon Design, who have been so understanding, accommodating, and sympathetic during the finishing of this thesis. Thanks also to Adam, for years of putting up with sharing a house with me, for teaching me so much that I have used in this PhD and beyond, and for the afternoon hacking which eventually lead to the research in this thesis.

I am indebted to my family for the unwavering support, encouragement, and confidence in my abilities. And finally, also to Katie, for too much to enumerate.

Contents

Contents i

List of figures iv

List of algorithms v

List of acronyms vi

List of symbols viii

Notation ix

1 Introduction 1

1.1 Sparsity and Compressive Sensing 1

1.2 CS reconstruction of large signals 3

1.2.1 Applications involving large signals 3

1.2.2 Reconstruction techniques for large signals 4

1.3 Parallelism and High Performance Computing 6

1.4 Motivation and research questions 7

1.5 Original contributions 8

1.6 Thesis outline 8

2 Review of CS and parallel computing literature 11

2.1 Information and sparsity 11

2.2 Compressive Sensing 14

2.2.1 Sampling 14

2.2.2 Reconstruction 15

2.2.2.1 Reconstruction by combinatorial optimisation 15

2.2.2.2 Reconstruction by convex optimisation 16

2.2.2.3 Reconstruction using greedy algorithms 17

2.2.2.4 Sparse regression 19

2.2.3 Review of CS literature 20

2.2.4 Sparse transforms . . . 22
2.2.5 Fast transforms and implicit dictionaries . . . 24
2.2.6 Applications . . . 25
2.3 High performance parallel computing . . . 26
2.3.1 Types of parallel computer . . . 26
2.3.1.1 SISD: Instruction-level parallelism . . . 27
2.3.1.2 SIMD: Data parallelism . . . 28
2.3.1.3 MIMD: Task parallelism . . . 28
2.3.2 Interconnect architecture . . . 30
2.3.3 Terminology . . . 31
2.3.4 History . . . 31
2.4 Parallelisation . . . 32
2.4.1 Performance metrics . . . 33
2.4.2 Theory . . . 33
2.4.2.1 Task-based parallelisation . . . 38
2.4.2.2 Data-based parallelisation . . . 39
2.4.3 Message Passing Interface . . . 40
2.4.3.1 MPI Primitives . . . 41
2.4.4 Hybrid-memory parallelisation . . . 42
2.4.5 Automatic Parallelisation . . . 43
2.5 Accelerated CS reconstruction . . . 44
2.5.1 GPU-accelerated reconstruction . . . 44
2.5.2 FPGAs and ASICs . . . 46
2.5.3 Distributed reconstruction for DCS . . . 47
2.5.4 Distributing convex algorithms using ADMM . . . 48
2.6 Chapter summary . . . 49

3 Parallelised greedy pursuits **51**
3.1 Matching Pursuit . . . 51
3.2 Distributed Matching Pursuit . . . 54
3.3 Orthogonal Matching Pursuit . . . 57
3.3.1 Implementing the LLSQ solver . . . 59
3.3.2 OMP with MGS updates and BS . . . 62
3.4 Distributed Orthogonal Matching Pursuit . . . 65
3.4.1 Storage and organisation . . . 65
3.4.2 Distribution of computation . . . 67
3.4.3 QA-worker algorithm . . . 67
3.4.4 R-worker algorithm . . . 70
3.5 Chapter summary . . . 74

4 Experimental results and analysis **75**
4.1 The Darwin cluster . . . 75
4.2 Implementing the algorithms . . . 76
4.3 Methodology . . . 78
4.4 Interpretation of results . . . 80
4.5 Results . . . 82
4.5.1 Multithreaded MP . . . 82
4.5.2 DistMP . . . 84
4.5.3 Multithreaded OMP . . . 90
4.5.4 DistOMP . . . 90
4.5.5 Large demonstration problem . . . 97
4.6 Comparison with Amdahl's Law . . . 99
4.6.1 DistMP . . . 99
4.6.2 DistOMP . . . 102
4.7 Chapter summary . . . 102

5 Conclusions **105**
5.1 Research questions . . . 105
5.2 Future work . . . 106
5.2.1 Hybrid parallelisation . . . 106
5.2.2 Implicit matrices . . . 107
5.2.3 Parallelisation of further greedy algorithms . . . 107
5.2.4 Multiple R-workers . . . 108
5.2.5 Distributed reconstruction with compute accelerators . . . 109
5.3 Final remarks . . . 110

Bibliography **111**

List of figures

2.1 An example of wavelet-based compression 13
2.2 Relative shapes of vectors and matrices in CS sampling 15
2.3 Illustration of Fourier and wavelet dictionary atoms 23
2.4 A simple example of instruction pipelining 28
2.5 Structure of a two-level NUMA architecture 29
2.6 Speed-up predicted by Amdahl's Law 34
2.7 Scaled-speedup predicted by Gustafson-Barsis' Law. 35
2.8 Foster's methodology for parallelisation 38
2.9 A task-dependency graph for a very simple algorithm 38

3.1 Column distribution over workers in DistMP 54
3.2 Inter-worker communications in DistMP 56
3.3 Matrix storage in DistOMP 66
3.4 Inter-worker communications in DistOMP 73

4.1 Scatter plots demonstrating variability in results 81
4.2 Multithreaded Matching Pursuit 83
4.3 Wall-time against number of workers for DistMP 87
4.4 Speed-up against number of workers for DistMP 88
4.5 Efficiency against number of workers for DistMP 89
4.6 Multithreaded Orthogonal Matching Pursuit 91
4.7 Wall-time against number of workers for DistOMP 94
4.8 Speed-up against number of workers for DistOMP 95
4.9 Efficiency against number of workers for DistOMP 96
4.10 Time taken to solve a large problem with DistOMP 98
4.11 Comparison between DistMP and Amdahl's Law predictions 101
4.12 Comparison between DistOMP and Amdahl's Law predictions 103

List of algorithms

3.1 Matching Pursuit 53
3.2 Distributed Matching Pursuit 57
3.3 Orthogonal Matching Pursuit: Concise form 59
3.4 A single iteration of column-wise Modified Gram-Schmidt QR decomposition . 61
3.5 An implementation of Back Substitution for solving LLSQ problems 62
3.6 Orthogonal Matching Pursuit: using MGS updates 64
3.7 Distributed Orthogonal Matching Pursuit: QA Worker 69
3.8 Distributed Orthogonal Matching Pursuit: R-worker 72

List of acronyms

- ADMM: Alternating Direction Method of Multipliers
- BP: Basis Pursuit
- BPDN: Basis Pursuit De-Noise
- BS: Back Substitution
- CG: Conjugate Gradients
- CPU: Central Processing Unit
- CS: Compressive Sensing
- DCT: Discrete Cosine Transform
- DFT: Discrete Fourier Transform
- DCS: Distributed Compressive Sensing
- DistMP: Distributed Matching Pursuit
- DistOMP: Distributed Orthogonal Matching Pursuit
- DWT: Discrete Wavelet Transform
- FFT: Fast Fourier Transform
- FLOPS: Floating Point Operations per Second
- FWT: Fast Wavelet Transform
- GPU: Graphics Processing Unit
- HPC: High Performance Computing
- IID: Independent and Identically Distributed
- LASSO: Least Absolute Shrinkage and Selection Operator

- LLSQ: Linear Least Squares
- MGS: Modified Gram-Schmidt
- MIMD: Multiple Instruction Multiple Data
- MISD: Multiple Instruction Single Data
- MP: Matching Pursuit
- MPI: Message Passing Interface
- MRI: Magnetic Resonance Imaging
- MSE: Mean Squared Error
- NSP: Null-space Property
- NUMA: Non-uniform Memory Access
- OMP: Orthogonal Matching Pursuit
- PRNG: Pseudo-Random Number Generator
- RIP: Restricted Isometry Property
- SIMD: Single Instruction Multiple Data
- SISD: Single Instruction Single Data
- UMA: Uniform Memory Access

List of symbols

Below I identify a number of symbols commonly used in this thesis. Some symbols may have multiple meanings depending on context.

- $\mathbf{f}$: A signal we wish to capture
- $\mathbf{x}$: Sparse representation of $\mathbf{f}$
- S: Sparsity ($S = \|\mathbf{x}\|_0$)
- $\mathbf{\Psi}$: Sparse basis ($\mathbf{x} = \mathbf{\Psi}^{-1}\mathbf{f}$)
- $\mathbf{b}$: Linear samples ($\mathbf{b} = \mathbf{A}\mathbf{x}$)
- $\mathbf{\Phi}$: Linear sampling ensemble ($\mathbf{b} = \mathbf{\Phi}\mathbf{f}$)
- $\mathbf{A}$: Equivalent sampling matrix ($\mathbf{A} = \mathbf{\Psi}\mathbf{\Phi}$)
- $\mathbf{r}$: Residual ($\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$)
- N: Number of columns in the equivalent sampling matrix
- n: Number of rows in the equivalent sampling matrix
- i: Current iteration number in an iterative algorithm
- k: Index of the dictionary column selected in each iteration of MP and similar algorithms
- $\mathbf{g}$: Selected dictionary atom ($\mathbf{g} = \mathbf{A}_k$)
- w: Index of a compute worker undertaking a parallel algorithm
- W: Number of compute workers undertaking a parallel algorithm
- $\mathbf{L}$: Local portion of $\mathbf{A}$ in DistMP and DistOMP
- $\mathbf{Q}$: Orthogonal portion of the result of a QR decomposition
- $\mathbf{R}$: Coefficients resulting from a QR decomposition
- $\mathbf{P}$: Local portion of $\mathbf{Q}$ in DistOMP

Notation

- x: A scalar
- $\mathbf{x}$: A column vector
- $\hat{\mathbf{x}}$: An estimate of $\mathbf{x}$
- $\mathbf{X}$: A matrix
- $\mathcal{A}$: A set
- $|\mathcal{A}|$: The cardinality (size) of a set
- x_i: The ith scalar element of $\mathbf{x}$
- $\mathbf{X}_i$: The ith column of the matrix $\mathbf{X}$
- $X_{i,j}$: The scalar element at the ith row and jth column of $\mathbf{X}$
- $\mathbf{x}_{\mathcal{A}}$: The elements of the vector $\mathbf{x}$ selected by the set $\mathcal{A}$
- $\mathbf{X}_{\mathcal{A}}$: The columns of the matrix $\mathbf{X}$ selected by the set $\mathcal{A}$
- $\mathbf{X}_{\mathcal{A},i}$: The elements selected by the set $\mathcal{A}$ in the ith column of $\mathbf{A}$.
- $\mathbf{X}^T$: The transpose of the matrix $\mathbf{X}$
- $\mathbf{X}^\dagger$: The Moore-Penrose pseudo-inverse of the matrix $\mathbf{X}$
- $\mathbf{0}^m$: A column-vector of zeros of length m
- $\mathbf{O}^{m\times n}$: A matrix of zeros with m rows and n columns
- $\mathbf{I}$: The square identity matrix, consisting of 1 in every element of the leading diagonal, and 0 otherwise
- $\mathbb{R}$: The set of real numbers
- $\mathbb{R}^m$: A column vector consisting of m real numbers
- $\mathbb{R}^{m\times n}$: A matrix with m rows and n columns consisting of real numbers

- $\{a \dots b\}$: The set of integers from a to b inclusive
- $[\mathbf{X}\ \mathbf{x}]$: The matrix resulting from appending a new column $\mathbf{x}$ to the right hand side of the matrix $\mathbf{X}$
- $\lfloor x \rfloor$: The floor operator; the result of rounding x down to the nearest integer
- $\lceil x \rceil$: The ceiling operator; the result of rounding x up to the nearest integer
- $a \bmod b$: The modulo operator; the remainder after dividing a by b

Throughout the text, italic characters (e.g. n, N) are used for scalar variables. Lowercase bold characters (e.g. $\mathbf{f}$, $\mathbf{x}$) are used for column vectors. Uppercase bold characters (e.g. $\mathbf{\Psi}$, $\mathbf{\Phi}$, $\mathbf{A}$) are used for matrices. Matrices and vector multiplication is indicated by matrices and vectors written sequentially, for example $\mathbf{\Phi\Psi}$ indicates the matrix-matrix multiplication, or inner product, between the matrices $\mathbf{\Phi}$ and $\mathbf{\Psi}$, and $\mathbf{Ax}$ indicates the matrix-vector product between the matrix $\mathbf{A}$ and the column vector $\mathbf{x}$.

Subscripts are used to indicate a single column or element taken from a vector or matrix. A bold character with a subscript indicates a single column taken from a matrix, for example $\mathbf{X}_i$ indicates the ith column of the matrix $\mathbf{X}$. An italic character with a subscript indicates a scalar element of a matrix or vector, for example x_i indicates the ith element of the vector $\mathbf{x}$, or X_{ij} indicates the element at the ith row and jth column of the matrix $\mathbf{X}$. Subscript indices are numbered starting from 1 as the first element.

Optimisation problems use the notation

$$\underset{x}{\text{minimise}}\ f(x) \text{ s.t. } g(x) = h.$$

Here, x is the optimisation variable, $f(x)$ is the function to minimise, and $g(x) = h$ is a constraint equation. A hat is commonly used for optimisation variables which approximate or converge to a value, for example $\hat{x}$ is an approximation to x.

We will commonly use the ℓ_0, ℓ_1 and ℓ_2 norms, denoted $\|.\|_0$, $\|.\|_1$, and $\|.\|_2$ respectively. These are defined as

$$\begin{aligned}
\|\mathbf{x}\|_0 &\stackrel{\text{def}}{=} \sum_i \left[x_i^0\right], \\
\|\mathbf{x}\|_1 &\stackrel{\text{def}}{=} \sum_i |x_i|, \text{ and} \\
\|\mathbf{x}\|_2 &\stackrel{\text{def}}{=} \sum_i \left[x_i^2\right].
\end{aligned}$$

Indices, whether indicating a worker number, iteration number, or a row or column of a matrix, all start from 1.

Chapter 1

Introduction

In the modern world we capture, communicate, and store a vast and ever increasing quantity of data. However, much of this data is actually very redundant, as evidenced by the prevalence of data compression which can often reduce the quantity of data by an order of magnitude or more without any noticeable loss. For example, audio and video compression do such a good job that they have become ubiquitous: digital audio and video are rarely stored or transmitted uncompressed. However, conventional compression relies on having perfectly captured every detail of a signal and then locating and exploiting patterns, repetitions, and redundancies in the captured data. What if we could skip the step of explicit compression and capture a signal in its pre-compressed form? Could we, at the same time, reduce the effort expended in capturing the signal or increase the detail captured for the same effort spent capturing it? Within certain constraints, Compressive Sensing (CS) enables us to do precisely this.

1.1 Sparsity and Compressive Sensing

Compressive Sensing (CS) achieves its remarkable results by relying on sparsity. The concept of sparsity can be seen as a development of Shannon's information theory [1] which describes how a discrete signal may contain much less information than the length of the signal. By distilling a signal down to its useful information we can transmit far less data and reconstruct the original signal at the receiver. Sparsity gives us a particular set of tools to do this. A sparse signal is a signal where most of its elements are zero or close to zero, meaning that the majority of the energy is concentrated in a small number of elements. Signals which are not naturally sparse might become sparse when represented in a particular basis, referred to as a sparse basis. Sparse representation can be written as $\mathbf{x} = \mathbf{\Psi}^{-1}\mathbf{f}$ where $\mathbf{f}$ is the signal, $\mathbf{\Psi}$ is the sparse basis, and $\mathbf{x}$ is the sparse representation of the signal. A simple example is that any signal which is highly periodic and relatively smooth is likely to be sparse in the Fourier domain, that is, after carrying out a Fourier transform on the signal. Through the use of more specialised or even custom-created basis transforms, we can come up with sparse representations of a wide variety

of classes of signals.

Sparsity was originally used in statistics to enhance signal modelling: For a model with a large number of parameters, allowing all parameters to vary freely is likely to result in over-fitting to training signals, but constraining the model so that only a few parameters may be non-zero (but not specifying which parameters) may produce much better results. A widely used algorithm is Least Absolute Shrinkage and Selection Operator (LASSO), popularised by Tibshirani [2]. A related topic in signal processing is sparse decomposition of signals, where observed signals are described using a small number of of *atoms* taken from an over-complete dictionary (one with fewer rows than columns). The use of over-complete bases makes decomposition more difficult but can lead to improved results [3]. Research seeking guaranteed recovery of sparse representations from known signals laid the groundwork for CS [4][5][6].

Exact reconstruction of incompletely sampled signals by exploiting a sparse representation was first demonstrated by Candès, Romberg, and Tao [7]. In CS, a signal is captured and compressed by applying linear sampling, for example the matrix-vector multiplication $\mathbf{b} = \mathbf{\Phi f}$ where $\mathbf{\Phi}$ is the sampling matrix and $\mathbf{b}$ is the vector of samples taken. In order to achieve compression, $\mathbf{\Phi}$ must have fewer rows than columns. In some applications linear sampling occurs naturally, for example Magnetic Resonance Imaging (MRI) [7], and in others it can be applied to a fully captured signal, for example by multiplying the signal vector by a random Independent and Identically Distributed (IID) Gaussian matrix which has fewer rows than columns. Linear sampling can also be applied by incompletely capturing a signal, represented by a sampling matrix consisting of a subset of the rows of the identity matrix. By changing the shape and properties of the sampling matrix we can take as few samples as possible while still being able to recover the original signal.

When attempting reconstruction we encounter an ambiguity because we have deliberately under-sampled: There are an infinite number of signals we could have observed which would produce the samples we captured. We narrow this down by finding a signal which is sparse in a particular basis and fits with our observed samples. Combining the sampling process with sparse representation, we write $\mathbf{b} = \mathbf{Ax}$, where the matrix $\mathbf{A} = \mathbf{\Phi\Psi}$ is called the *equivalent sampling matrix*. We reconstruct the signal by calculating $\hat{\mathbf{f}} = \mathbf{\Psi}\hat{\mathbf{x}}$, where $\hat{\mathbf{x}}$ is found by solving the optimisation problem

$$\underset{\hat{\mathbf{x}}}{\text{minimise}} \left\|\hat{\mathbf{x}}\right\|_1 \text{ s.t. } \mathbf{A}\hat{\mathbf{x}} = \mathbf{b}. \tag{1.1}$$

Equation 1.1 is called Basis Pursuit (BP). Previously used for sparse decomposition [8], BP was utilised for CS reconstruction by Candès, Romberg, and Tao [7]. Using BP, the authors derive conditions on the sampling matrix necessary for reconstruction. This result was generalised by Candès and Tao [9] to handle noisy signals with inexact sparsity, in the process introducing the Restricted Isometry Property (RIP) which determines whether sufficiently sparse signals can be exactly recovered with high probability following sampling by a particular sampling matrix.

While BP can be solved using modern linear and convex programming techniques, this is generally infeasible for very large problems. Furthermore, while BP is simple to state, implementing an efficient solver is far from easy. Greedy algorithms are an alternative which are simple to implement and may converge on a solution much faster than convex algorithms. They are not a new invention: The simplest greedy pursuit, namely Matching Pursuit (MP), was introduced for signal decomposition in 1993 [3]. Tropp [6] showed criteria for guaranteed convergence when using Orthogonal Matching Pursuit (OMP), another greedy algorithm, for sparse approximation and later applied the same algorithm to CS reconstruction showing promising theoretical and practical results [10].

Numerous potential applications for CS have been demonstrated. The original application of MRI [7] is particularly attractive due to Fourier sampling occurring naturally along with inherent limits on the number of samples which can be taken. Another well-known application is the single pixel camera [11], where a camera uses a micro-mirror array to sub-sample an input image which is then focused onto a single pixel sensor.

1.2 CS reconstruction of large signals

Relatively efficient algorithms have been developed for CS reconstruction but handling large signals remains problematic: Candès and Romberg [12] comment how solving large problems with unstructured sampling systems is numerically intractable due to the huge quantity of data involved, in particular explicit representation of the sampling matrix. A key issue is sampling matrix size. If we take n samples of a signal of length N then our sampling matrix will contain Nn elements. If the number of samples taken is proportional to the signal size then the sample matrix size is proportional to the square of the signal length. This is compounded when the signal concerned is an image: With a fixed sub-sampling ratio both N and n are proportional to the number of pixels in the image, so Nn is proportional to the fourth power of the side-length of the image. A typical 1000×1000 pixel image sub-sampled at 1% would therefore produce a sampling matrix with 10^{10} elements. If each element were stored as a 64-bit double precision floating point value then this corresponds to a sampling matrix size of 75 gigabytes. In addition to the quantity of memory required, each application of such a large matrix in the reconstruction algorithm would require a large number of operations leading to long reconstruction times.

1.2.1 Applications involving large signals

A number of proposed and demonstrated applications for CS involve reconstruction of such large problems. In Transmission Electron Microscopy (TEM), a beam of electrons is projected through a target onto a sensor in order to form images of the sample. Some targets degrade under bombardment with a large number of electrons, so it is desirable to be able to derive a high quality image using a limited dose of electrons. The application of CS to TEM is proposed by

Binev et al. [13] who demonstrate reconstruction of 128×128-pixel images, and Stevens et al. [14] show how in-painting techniques based on CS can be used to reconstruct megapixel images from sub-sampled TEM. Stevens et al. [15] also propose the use of CS to reduce the quantity of data involved in implementing video-rate TEM. Each 1024×1024-pixel frame would form a signal sized $N = 10^6$ and considering the whole sequence of 900 frames as a single signal would increase this to $N = 9 \times 10^8$. Each frame could be sampled and reconstructed independently but this prevents exploitation of temporal redundancy between frames: considering a number of frames together will lead to the best compression and reconstruction quality.

Electron holography operates similarly to TEM except that the structure of a 3-dimensional object is inferred from 2-dimensional images captured on a sensor. Multiple 2D images are collected by modulating the electron beam before it enters the object. Compressive holography, proposed by Brady et al. [16], uses sparse techniques to enhance the reconstruction, modelling the interaction with the beam and object as linear sampling and the shape of the object as a sparse signal. The size of the reconstruction problem can become very large because the reconstructed signal is 3-dimensional, for example Brady et al. use 10 depth-planes resulting in a single reconstruction having $N = 5 \times 10^6$ and $n = 5 \times 10^5$. Endo et al. [17] simulate a higher resolution 1024×1024-pixel sensor resulting in $N = 10^7$ and $n = 10^6$. Hahn et al. [18] extend compressive holography further, proposing compressive holographic video rather than just static images. Each 3-dimensional capture has a size of $1024 \times 1024 \times 35$, with 109 frames captured in a short sequence. Considering the full sequence as a single signal would lead to a size of $N = 4 \times 10^9$.

Radio interferometry, used for astronomical imaging, involves the use of a number of spatially distributed radio receivers to simulate one large receiver. Signals observed by pairs of receivers are compared in order to build up a picture of the sky. This setup can be modelled as linear sub-Fourier sampling and so the use of CS is attractive as a means of reconstruction. Belle, Armstrong, and Gain [19] propose the use of sparse techniques for deconvolution of astronomical images. This is demonstrated using synthetic images up to 64 megapixels ($N = 6 \times 10^7$) generated by simulating a radio-interferometer. The authors explain how existing radio interferometers produce images of less than 16 megapixels but that the upcoming Square Kilometre Array will produce much larger 10-gigapixel images. Another application of CS involves conventionally captured optical signals. Astronomical images captured from ground-level are subject to blurring from atmospheric effects. The use of CS for de-blurring these images is proposed by Fiandrotti et al. [20]. The example of de-blurring a mega-pixel image results in a problem size of $N = 10^6$ and $n = 5 \times 10^5$.

1.2.2 Reconstruction techniques for large signals

Several techniques may be used to assist with CS reconstruction of large signals. For many reconstruction algorithms the majority of the computational load takes the form of matrix

multiplication by the equivalent sampling matrix. If the sampling matrix is structured then it may be possible for the multiplication to be replaced by a fast transform operator. For example, if the sampling process involves a Fourier transform then application of the sample matrix in a reconstruction algorithm could be implemented using a Fast Fourier Transform (FFT), which is significantly faster than a general matrix-vector multiplication. Furthermore, with a structured sampling matrix it is not necessary to explicitly store the whole matrix in memory so the memory capacity required for reconstruction can be greatly reduced. In some applications (for example holography [16][17], radio interferometry [19], and MRI [7]) Fourier sampling occurs naturally and so the use of the FFT in reconstruction algorithms is an obvious choice. A structured sample matrix may also be chosen for ease of reconstruction, for example Fiandrotti et al. [20] use circulant matrices in order to reduce memory capacity requirements and accelerate reconstruction. However, enforcing the use of structured matrices to improve reconstruction speed may harm compression ratio or reconstruction quality: In general, an unstructured over-complete sparsity basis can produce a more sparse result for a given class of signal than a structured matrix. In particular, dictionary learning [21] may be used to generate an unstructured sparsity basis tailor-made to a specific class of signals.

Another common technique used for solving large CS problems is partitioning or blocking [22][23][24] where the original signal is split into multiple segments before sampling which allows each smaller segment to be independently reconstructed. In addition to reducing reconstruction time and memory requirements, if a signal is segmented in the time domain then overall latency between capture and reconstruction may be reduced. Partitioning is often carried out implicitly, for example if each frame in a video sequence is independently sampled and reconstructed, as was done by Hahn et al. [18]. Partitioning is also used explicitly in some applications, for example Stevens et al. [14] apply an in-painting algorithm to 8×8-pixel patches, and in their work on video-rate TEM [15] the image is sampled and reconstructed in patches with each patch using the same sample matrix. The main downside to partitioning is reduction of compression ratio or reconstruction quality: the larger the number of partitions a signal is split into, the greater the total number of samples required for exact recovery [24]. Furthermore, if an image is reconstructed in independent spatial partitions then the reconstruction is likely to contain visible blocking artefacts along boundaries.

Various researchers have demonstrated accelerated CS reconstruction using parallel compute or compute accelerator hardware. Graphics Processing Unit (GPU) compute is commonly used due to the prevalence of GPU hardware and the relative ease of programming using toolkits such as cuBLAS [25]. Many implementations [17][19][26][27][28] rely on structured sampling matrices where the FFT can be used instead of explicit storage and computation. A few implementations [26][29][30][31] use unstructured dictionaries but these are limited to problems of up to $N = 2 \times 10^4$. Another area of research is the use of FPGA (Field Programmable Gate Array) and ASIC (Application-Specific Integrated Circuit) accelerators [32][33][34][35]. However, this research tends to focus on very low latency reconstruction and

is limited to very small problems (typically $N < 1024$). Researchers have demonstrated solution of much larger problems using algorithms based on Alternating Direction Method of Multipliers (ADMM) [36][37] but these algorithms have differing reconstruction performance compared to well-known algorithms such as BP and OMP and the results of reconstruction depend on the size and connectedness of the compute resource used.

In this section I have shown several applications which require reconstruction of large CS problems. Megapixel-sized images are not unusual, but sampling one with CS can easily result in a sample matrix larger than can be held in memory on most computers. I then looked at the use of partitioning and implicit sampling matrices, two common strategies used for processing large CS reconstructions, and showed how each has downsides including reduced compression or reconstruction quality. High performance parallel computing, where the memory and processing power of many computers can be combined to tackle a single problem, provides an alternative strategy for solution of large CS problems.

1.3 Parallelism and High Performance Computing

Computers have exploited parallelism to increase performance for over five decades [38][39] and this has not changed in recent years. Since 2000, increases in processor clock speed have slowed and attention has turned to multi-core architectures to deliver continuing performance improvements [40]. Limits on attainable clock speed restrict the performance achievable on an individual processor core, but for problems which are amenable to parallelisation a large number of processor cores can work together to tackle a problem much more quickly than a single processor could alone. Another aspect to parallelisation is memory capacity: It is desirable to be able to hold the whole problem being tackled in memory because this is much faster (both lower latency and higher bandwidth) than resorting to disk storage. However, a limited quantity of memory can be attached to a single processor. This is limited by memory density and by the memory all needing to be physically close to the processor to meet latency requirements. A parallel computer can possess much more memory than could feasibly be attached to a single-processor computer.

Two common types of parallel processor, as categorised by Flynn's Taxonomy [41][42], are Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). SIMD processors carry out a single operation on a large quantity of data, for example vector processor super-computers [43] and GPU computing. Vector processing can give massive speed-up while being simple to program but is only applicable to certain problems, generally involving mathematical operations on large datasets. Also, the scale of vector processors is limited by the synchronisation and communication required. MIMD parallelism includes the use of multiple processor cores and multiple discrete processors. MIMD parallelism can be divided into Uniform Memory Access (UMA) and Non-uniform Memory Access (NUMA) depending whether the memory structure is uniform or heterogeneous. UMA parallel computing includes

the ubiquitous multi-core processors found in modern computers while NUMA architectures are found in servers with multiple discrete processors and in distributed memory super-computers. Efficiently programming a NUMA parallel computer requires care to ensure that data is stored locally to the processors which will be operating on it. Communication generally takes the form of explicit messages between processors, for example using the Message Passing Interface (MPI) protocol [44].

Modern High Performance Computing (HPC) generally uses distributed-memory clusters composed of a large number of standard servers with a high performance interconnect network [43][45]. Each server, referred to as a *node*, runs its own operating system and may have its own hard-disk storage. Each node separately runs a program, and the programs communicate over a network to coordinate solving a problem. The interconnect network often uses high bandwidth and low latency fibre-optic links structured as a redundantly-connected topology allowing simultaneous fast communications across multiple parts of the network. To effectively exploit a distributed memory massively parallel computer, software must be modified or specially written for this purpose [46][47]. Separation must be introduced in either tasks or data so that the problem to be solved can be split up and spread over the compute cluster. A commonly used technique is described by Foster's Methodology [48].

1.4 Motivation and research questions

In Section 1.2.1 I described several applications which motivate the use of CS with large signals. This leads to two issues: the dataset may be too large to fit in memory, and carrying out reconstruction on such a large signal could take a very long time. The latency between availability of samples and completion of reconstruction could be prohibitive in some scenarios or limit the quantity or rate of data which can be collected. In Section 1.2.2 I described how implicit dictionaries and partitioning can be used to enable solution of very large problems but how both come with downsides including reduced compression or reduced reconstruction performance. I then describe various research which has investigated hardware accelerated CS reconstruction and explain how no existing techniques enable the reconstruction of large problems using unstructured sampling matrices while giving the same results as standard reconstruction algorithms.

The goal of this thesis is to investigate whether high performance parallel computing can be applied to reconstruct very large CS problems and to solve them quickly. In particular, I will focus on CS problems using large unstructured dictionaries, where the use of implicit matrices and fast transforms is not applicable. I also aim to maintain the properties and results of standard reconstruction algorithms, in particular Matching Pursuit (MP) and Orthogonal Matching Pursuit (OMP), and to design algorithms such that the reconstruction is identical regardless of the size of structure of the computing resource. For parallelisation I will target large, homogeneous, compute clusters using message passing for communication between compute workers. By making available parallelised implementations of MP and OMP I hope to enable CS researchers

to carry out numerical experiments on larger unstructured problems than were previously feasible to reconstruct, and to create interest in high performance parallel implementations of other CS reconstruction algorithms. I will base my work around the following five questions and put forward specific answers to them in my conclusions in Chapter 5.

1. How can MP and OMP be parallelised using Message Passing Interface (MPI) on a compute cluster?
2. Can the parallel algorithms be used to solve much larger problems than would ordinarily fit in main memory?
3. Can the parallel algorithms solve problems faster than a single compute worker could alone?
4. How efficient are these parallel algorithms?
5. How do the properties of the algorithms vary with number of workers and differing CS problems?

1.5 Original contributions

The original contributions of this thesis are the development and testing of two novel algorithms, Distributed Matching Pursuit (DistMP) and Distributed Orthogonal Matching Pursuit (DistOMP), which allow the use of a HPC resource to reconstruct very large signals captured using CS while giving the same results as MP and OMP respectively. These algorithms are motivated by the occurrence of large signals in various CS applications, in particular those involving imaging, and by the reduction in compression caused by the use of partitioning and structured sample matrices, two techniques commonly used to allow reconstruction of large signals. The design and implementation of both algorithms are detailed, followed by testing to quantify the level of speed-up achieved for various sizes and sparsity of CS problem and varying numbers of compute workers.

1.6 Thesis outline

In this chapter I have given a brief introduction to the fields of CS and HPC. After an overview of CS I showed how reconstruction can involve intensive computation on very large quantities of data which may take prohibitively long to process and reviewed various applications of CS which require the use of very large signals. I also explained how modern high performance computing focuses on parallelism, culminating in large-scale clusters of standard servers. I described the intersection between these fields, suggesting how the parallelism techniques from HPC may be able to accelerate processing of CS reconstruction problems. Finally, I enumerated

five specific research questions I will attempt to answer in this thesis. The remainder of this thesis is structured as follows:

In Chapter 2 I give a detailed introduction into the material involved in the original work in this thesis. I begin with the work of Shannon on how information may be quantified and how it differs from data. I draw parallels between information theory and the mathematical concept of sparsity, and give an example of how this may be used to achieve data compression. I introduce CS as an application of sparsity to sampling theory, giving particular attention to the reconstruction process which will form the focus of later chapters. Next, I give an introduction to high performance parallel computing, discussing the types of parallel computer and their history. Finally, I discuss parallelisation, the process of modifying algorithms and programs to exploit parallel computers. I cover the theory and several performance metrics relevant to parallelisation and explain two common strategies: task parallelism and data parallelism. I also give details of MPI, the communication mechanism I use in later chapters.

In Chapter 3 I introduce the algorithms DistMP and DistOMP, both of which are my original contributions to the field. I begin with a detailed breakdown of the algorithm for MP and then show how DistMP uses data parallelism in order to distribute the sampling matrix over a large number of compute workers. Next, I give a detailed breakdown of the OMP algorithm, showing how the use of a progressive Modified Gram-Schmidt (MGS) decomposition gives a particularly efficient implementation. Finally, I present DistOMP, a parallelised alternative to OMP based on data parallelism.

In Chapter 4 I describe my evaluation and characterisation of the algorithms described in the previous chapter. I begin by describing the Darwin Cluster, upon which my experiments will be carried out, and also describe various details of the implementations I produced of the algorithms. I then present results and discussion comparing and characterising the algorithms using different numbers of compute workers and solving different shapes and sizes of problems. Finally, I show the scale of problems which can be handled by my new algorithms by using DistOMP to solve a particularly large problem.

In Chapter 5 I summarise the contributions made in this thesis and respond to the questions posed earlier in this chapter. I briefly compare my work to existing alternatives. I then detail various paths for extending my work which could be carried out in future.

Chapter 2

Review of CS and parallel computing literature

In this chapter I will review the literature concerning the topics of CS, HPC, and accelerated CS reconstruction. I begin with an introduction to Shannon's Information Theory and his demonstration of the decoupling between data and information. I then give a brief overview of the concept of sparsity and show how it is related to Information Theory and how sparsity can be used for compression. Next, I will give an introduction into CS, including descriptions of its history, the underpinning theory, and some reconstruction techniques. Aside from CS, the other key component of my work concerns HPC clusters and parallel computing. I will introduce the history and terminology of high performance parallel computing and overview some of the common architectures used. I then cover techniques used for parallelisation, the conversion of algorithms and software to exploit parallel compute resources. I will explain the underlying theory, cover some of the common techniques used for parallelisation, and introduce the metrics used to evaluate parallel compute performance which I will later apply to my own algorithms. In the final section I review related work on the topics of accelerated, parallelised, and distributed CS reconstruction techniques.

2.1 Information and sparsity

In his 1948 article, *A Mathematical Theory of Communication* [1], Claude Shannon founded the field of information theory by defining the information content of a message. The length of a message may be defined in bits as follows: Consider the message constructed from symbols, each of which may take one of a number of values (i.e., characters which can be any upper-case letter). Take the base-two logarithm of the number of values per symbol, and multiply it by the number of characters in the message. For example, using upper-case letters and spaces, the word PARIS has a size of $\log_2(27^5) = 24$ bits (rounding up). Shannon's innovation was considering the quantity of information (also referred to as information entropy) present, distinct

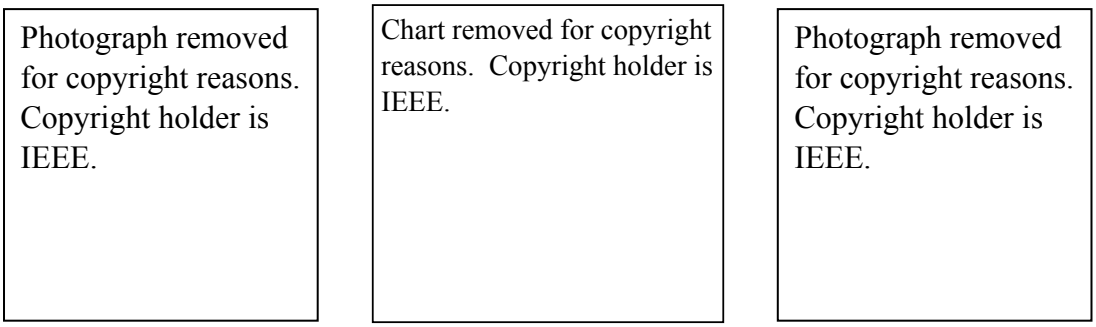
from the message size. For example, for English text, if we randomly choose a string of letters and punctuation, we are exceedingly unlikely to end up with a valid message. The information content of a message depends not on the length of the message, but on the number of valid messages we could have chosen to transmit. If there are n possibilities, the information content is defined as being $\log_2 n$ bits. Shannon showed that English text (upper-case letters and spaces) has an average information quantity of 2.14 bits per letter, much less than the 4.75 bits per letter needed to represent totally random characters. This implies we might be able to take a message of English text and, by exploiting information theory and prior knowledge about the English language, represent it in under half the amount of space. This process is conventional data compression (or source coding), where we take some data and try to represent it in as small a space as possible by removing any redundancy or exploiting prior knowledge. In comparison to compression, CS is more like taking a message where many of the characters are missing or obscured and attempting to reconstruct the original message.

CS is closely related to the theory of sparsity. A signal is referred to as sparse if it has relatively few non-zero elements compared to its size. In particular, sparsity is often not a fixed proportion of a signal's size, but depends instead on the source of the signal or how it was generated. Sparsity may be measured using the ℓ_0 norm

$$S = \|\mathbf{x}\|_0 \stackrel{\text{def}}{=} \sum_i x_i^0.$$

Relatively few signals are sparse in their natural form, however a surprising number of signals appear sparse after an appropriate transform. Referring to the original signal as $\mathbf{f}$ and its sparse representation as $\mathbf{x}$, we can write $\mathbf{f} = \mathbf{\Psi x}$, where $\mathbf{\Psi}$ is our sparse basis. Natural signals may not be perfectly sparse however transformed, that is, the majority of elements are unlikely to be precisely zero. However, it is common to find a small number of elements dominate the energy in an approximately sparse signal, while the vast majority of the elements are close to zero and have negligible energy. A compressible signal is one which is well-approximated by a small number of its largest coefficients or which has coefficients which decay with a power-law trend. It is also the case that, using an appropriate transform, the significant elements of a sparse representation of a signal often represent the important or useful parts of the signal, where the majority of small elements correspond to noise. As an example, we would expect a periodic signal to appear sparse after application of a Fourier transform (for example a Discrete Fourier Transform (DFT) or Discrete Cosine Transform (DCT)). A signal which appears periodic has the bulk of its energy in the Fourier domain at components corresponding to the frequency of periodicity and its harmonics. Any noise in the signal would, after a Fourier Transform, appear as a small magnitude spread across all frequencies. More exotic transforms than the Fourier Transform can turn surprisingly complicated (and realistic) signal structures into relatively few sparse components.

It is easy to see the structure in a highly periodic signal and therefore imagine how it is



(a) The original photograph with 1024×1024 pixels

(b) Wavelet coefficients of the photograph

(c) A reconstruction from only the 25,000 largest coefficients

Figure 2.1: An example of wavelet-based compression from Candès and Wakin [49]

probably sparse in the Fourier domain, but many less obviously structured signals are similarly amenable to sparse representation. For example, photographs are generally full of detail, containing innumerable shapes, colours and textures. It is hard to imagine that an attempt to extract the structure of the photograph and then reconstruct the original could result in a convincing replica. However, the Discrete Wavelet Transform (DWT) can do just this. As an example, Figure 2.1a shows a photograph with 1024×1024 pixels, each of which can take 256 greyscale levels. Figure 2.1b shows the value of the wavelet coefficients resulting from performing the DWT on the photograph. The wavelet coefficients are approximately sparse: the energy of the post-transform signal is concentrated in a small subset of the elements, representing the dominant structure in the image. We can further emphasize this result by forcing smaller elements in the DWT domain signal to zero so that only the 25,000 with greatest magnitude (2.4% of the elements) contain any information, then performing the inverse wavelet transform after this threshold operation. Figure 2.1c shows the result: the difference between this and the original photograph is imperceptible.

Earlier we said that regardless of its size, a message (or signal) may contain a more limited quantity of information, and showed how to calculate the information content of a message from the level of uncertainty. Conventional compression gives us tools to distil the information out of a message and transmit this as concisely as possible, followed by later expanding out the complete, original, message. The mathematics of sparsity gives us a very different tool for analysing signals, but one with a similar meaning and some parallels with information theory. Signals, or messages, rarely contain as much information as their size might imply, and the right techniques can separate the two. CS relies on signals having a sparse representation as a way to reconstruct the signals from incomplete data. However, unlike compression, we do not need to observe and process the full signal before producing the concise form: Under the right conditions, we can start with incomplete observations of a signal and perfectly reconstruct that same signal, sight unseen, even given the presence of noise in the signal or our observations of it.

2.2 Compressive Sensing

CS is a research area based on the idea of combining compression into the action of sensing, and the techniques which can be used to reconstruct an incompletely sampled signal. Essentially, in some signals the information we wish to capture is broadly spread over the whole length of the signal. This means we can capture incomplete samples from the signal without loss of information. In many cases it doesn't even matter which parts of the signal we sample: random sampling is not only possible, but has beneficial statistical properties. CS relies on the idea of mathematical sparsity to describe the quantity and form of information in a signal or piece of data, as well as using sparse techniques to reconstruct the captured signals. CS can bring advantages in a variety of scenarios. When we want to retrieve a signal from an energy-constrained sensor over a low-bandwidth channel, CS can be used to apply compression with only trivial computation needed on the sensing end (in exchange for more intensive computation required for reconstruction than conventional decompression). In some situations there are inherent limits on how many samples we may take from the signal we are trying to observe, and CS may be used to reconstruct the best possible estimate of what the original signal looked like from incomplete observations.

2.2.1 Sampling

CS relies on linear sampling to capture signals. Considering the original signal to be captured as a finite dimensioned vector, the sampling process is equivalent to a matrix-vector multiplication. Naming the original signal $\mathbf{f}$, our sampling matrix (also known as sampling ensemble) $\mathbf{\Phi}$, and our captured samples as $\mathbf{b}$, sampling is written as

$$\mathbf{b} = \mathbf{\Phi f}.$$

$\mathbf{\Phi}$ is defined as having n rows by N columns, where N is the size of our original signal $\mathbf{f}$ and n is the number of samples taken in $\mathbf{b}$. The sampling ratio, or compression ratio, is n/N. The sampling process is represented in Figure 2.2 which shows the relative shapes of the vectors and matrices.

In order to achieve useful compression we desire that the sample vector has significantly fewer elements than the original signal. This is equivalent to a sampling matrix which is wide, that is, it has significantly more columns than rows. In some applications we can implement a physical sampling process which can be represented as linear sampling in this manner. In other cases we fully sample the signal over the desired period and then apply the sampling matrix multiplication numerically in software before storing or transmitting the samples.

Not every conceivable sampling matrix leaves the possibility of reconstruction (for example $\mathbf{\Phi} = \mathbf{O}$ is a particularly degenerate case which obviously cannot be reconstructed). In the next section I will describe the several properties which allow us to predict and analyse which

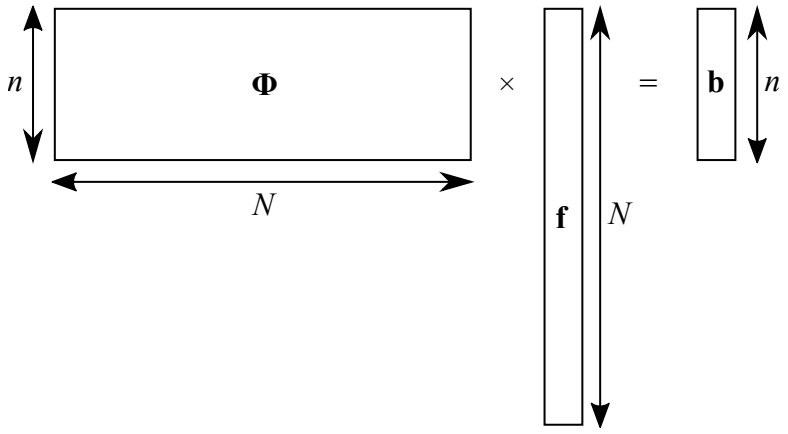


Figure 2.2: The relative shapes of the vectors and matrices involved in CS sampling.

sampling matrices and signals are likely to result in successful reconstruction.

Examples of sampling matrices are useful in demonstrating their forms and properties. Using the identity matrix as a sampling matrix ($\mathbf{\Phi} = \mathbf{I}$) is a trivial example which achieves no compression but is particularly easy to reconstruct ($\mathbf{f} = \mathbf{I}^{-1}\mathbf{b} = \mathbf{b}$). However, by taking the identity matrix and removing some of its rows we can achieve a useful sampling matrix with particular real-life significance: The sub-sampled identity matrix, produced by randomly removing rows from the $N \times N$ identity matrix until we are left with only n columns, is equivalent to the sampling process randomly selecting n of N elements of the original signal to capture.

Another class of sampling matrix is uniform dense sampling matrices. The most common is a dense n-by-N matrix of IID Gaussian random numbers. While this may be less practical to realise than the sub-sampled identity matrix, it has desirable statistical properties meaning reconstruction is possible for a wider class of target signals. A Bernoulli matrix of IID random variables taking the values ± 1 can also be used, giving similar results [50].

2.2.2 Reconstruction

2.2.2.1 Reconstruction by combinatorial optimisation

The reconstruction problem can be stated simply as solving

$$\mathbf{b} = \mathbf{\Phi}\hat{\mathbf{f}} \tag{2.1}$$

for $\hat{\mathbf{f}}$, an estimate of the original signal. However, this equation has an infinite number of solutions. Equivalently, we cannot invert $\mathbf{\Phi}$ because it has more columns than rows. We need more information to narrow down which of the infinite number of solutions is the correct one.

In CS, our prior knowledge is that a signal is sparse, or has a sparse representation. In general our signal $\mathbf{f}$ might not be sparse, but we assert that in the cases we consider, a sparse representation $\mathbf{x}$ exists such that $\mathbf{f} = \mathbf{\Psi}\mathbf{x}$. We can rewrite the reconstruction problem, Equation 2.1, as

$$\mathbf{b} = \mathbf{\Psi\Phi\hat{x}} = \mathbf{A\hat{x}}, \tag{2.2}$$

where $\mathbf{A} = \mathbf{\Psi\Phi}$ is the *equivalent sampling matrix*, or *equivalent sensing matrix*. For simplicity, I will use the equivalent sampling matrix where possible, and consider $\mathbf{x}$ as the signal we wish to reconstruct (knowing that the actual original signal $\mathbf{f}$ can be easily recovered by the matrix-vector product $\mathbf{\Psi x}$).

Using the ℓ_0-norm as a measure of sparsity, we can exploit our knowledge of the sparsity of $\mathbf{x}$ by choosing the solution to Equation 2.2 which is most sparse. This is the optimisation problem

$$\underset{\hat{\mathbf{x}}}{\text{minimise}} \left\|\hat{\mathbf{x}}\right\|_0 \text{ s.t. } \mathbf{A\hat{x}} = \mathbf{b}. \tag{2.3}$$

Unfortunately, Equation 2.3 represents a combinatorial optimisation problem which is generally intractable to solve.

2.2.2.2 Reconstruction by convex optimisation

If we replace the sparsity measure in Equation 2.3 with the ℓ_1-norm then, with overwhelming probability, the solution is identical.

$$\underset{\hat{\mathbf{x}}}{\text{minimise}} \left\|\hat{\mathbf{x}}\right\|_1 \text{ s.t. } \mathbf{A\hat{x}} = \mathbf{b}. \tag{2.4}$$

Equation 2.4 is known as BP. This problem is no longer combinatorial. So long as the signal, sampling matrix, and sparse basis are real-valued, we can recast the optimisation as a linear problem [8]:

$$\begin{aligned} \underset{\hat{\mathbf{x}},\mathbf{u}}{\text{minimise}} \sum_i u_i \text{ s.t. } \hat{x}_i - u_i &\leq 0 \; \forall i \\ -\hat{x}_i - u_i &\leq 0 \; \forall i \\ \mathbf{A\hat{x}} &= \mathbf{b}. \end{aligned}$$

We may extend Equation 2.4 by considering that our sampling process may have been contaminated with noise. Because the concepts leading to a sparse representation are tied to the underlying signal rather than our noisy samples, in addition to CS letting us reconstruct from incomplete samples, it can even help to remove noise from the observations! We model our noisy sampling process (or noisy signal) as

$$\mathbf{b} = \mathbf{\Phi f} + \mathbf{n},$$

where $\mathbf{n}$ is a vector of IID Gaussian random numbers, representing noise present in either the signal, our sampling process, or the storage or transmission of the samples. With the addition of noise, the optimisation in Equation 2.4 is unlikely to converge to the desired solution. We may extend BP to allow noisy measurements by replacing the strict equality constraint with an

inequality, where ϵ is an estimate of noise magnitude,

$$\underset{\hat{\mathbf{x}}}{\text{minimise}} \left\|\hat{\mathbf{x}}\right\|_1 \text{ s.t. } \left\|\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}\right\|_2 < \epsilon. \tag{2.5}$$

Equation 2.5 is known as Basis Pursuit De-Noise (BPDN). As with BP, assuming real-valued variables this optimisation may also be recast as

$$\begin{aligned} \underset{\hat{\mathbf{x}},\mathbf{u}}{\text{minimise}} \sum_i u_i \text{ subject to } \hat{x}_i - u_i &\leq 0 \; \forall i \\ -\hat{x}_i - u_i &\leq 0 \; \forall i \\ \left\|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\right\|_2 &< \epsilon. \end{aligned}$$

This optimisation takes the form of a quadratic cone problem, for which there are efficient numerical solvers available.

For the remainder of this thesis, I will use $\mathbf{f}$ to refer to both the original signal and an estimate of it produced by reconstruction, as well as using $\mathbf{x}$ to refer to the sparse coefficients of the original signal as well as estimates produced by reconstruction.

2.2.2.3 Reconstruction using greedy algorithms

The optimisation techniques discussed so far, which all fall under the banner of convex optimisation, are guaranteed to find the correct global solution to the BP or BPDN optimisation problems, and as we show in the next section we can ensure that the solution to the optimisation problem will be the signal we are trying to reconstruct so long as certain requirements are met. However, while modern convex optimisation solvers are efficient compared to older techniques, a significantly faster class of solvers exists: the greedy pursuits.

A *greedy* algorithm is a class of iterative solver which, at each iteration, takes the most immediately obvious step towards the solution. A greedy algorithm is effective for local optimisation, where a simple gradient descent leads to the local optimum, but may never find a preferable global optimum. For this reason greedy algorithms are well-suited to linear or convex optimisation problems where only one optimum point exists. A simple example is Newton's method for numerical equation solution. The advantage of greedy algorithms is their simplicity, which usually makes them both easy to implement and very fast to run, as each iteration involves relatively few calculations.

The greedy pursuits are a class of algorithms for CS reconstruction (or, equivalently, sparse approximation), loosely based on Matching Pursuit (MP). As would be expected for a greedy algorithm, they are much simpler to implement than a convex optimisation solver and are also much faster.

MP [3] is an iterative solver, introduced in 1993 for signal decomposition. In the ith iteration, MP decomposes the signal into the sum of at most i columns of the equivalent sampling matrix, each multiplied by a coefficient, and a residual. At each iteration MP finds the column with

the highest absolute correlation with the residual and adds that column to the sum, with its correlation as the coefficient. The algorithm is initialised by setting the residual to the signal to be approximated, in this case $\mathbf{b}$. The result of the algorithm is the vector of coefficients, $\mathbf{x}$. MP is exceedingly simple to implement, but both its performance and run-time are lacking: because the columns may in general have non-zero correlation, MP may (and often does) revisit a column: If at iteration i we select the column $\mathbf{g}_i$ then at the end of that iteration the residual $\mathbf{r}$ is orthogonal to $\mathbf{g}_i$. In a future iteration j we subtract some multiple of the column $\mathbf{g}_j$ from $\mathbf{r}$. However, if $\mathbf{g}_j$ has non-zero correlation to $\mathbf{g}_i$ then the residual is no longer orthogonal to $\mathbf{g}_i$ at the end of iteration j and we may select $\mathbf{g}_i$ again in future. Most other greedy algorithms take the same form as MP (iterative improvement of sparse approximation, each iteration correlating the residual with the dictionary) but with improvements to performance and reduced run-time.

Iterative Hard Thresholding (IHT) [51] improves on MP by allowing multiple dictionary columns to be selected in each iteration. In each iteration the correlation between the residual and dictionary is added to the current sparse approximation. A hard threshold is then applied, selecting the S coefficients with greatest magnitude and setting all others to zero. IHT has a computational cost comparable to MP in each iteration while improving on the performance of MP. Implementation requires sorting the coefficients in each iteration but is otherwise straightforward, however, reconstruction performance of IHT still falls short of many other greedy algorithms.

OMP [10], another greedy algorithm based on MP, solves the issue of MP revisiting columns. Its operation is largely similar to MP, but after selecting the column with the highest correlation to the residual the sparse estimate $\mathbf{x}$ is recalculated such that the residual is orthogonal to every column selected so far (referred to as the working set). This ensures we never select the same column twice, and so OMP often completes in far fewer iterations than MP (although each iteration requires more computation than MP). In general OMP completes in S iterations, the number of non-zero elements in the sparse approximation. This means that for very sparse signals it executes quickly, but for less sparse signals it can take a large number of iterations with each requiring an expensive orthogonalisation step. The reconstruction performance of OMP is still worse than convex optimisation but it has become widely used in CS research due to its relative simplicity and ease of implementation. The operation of MP and OMP will be explored in greater detail in Chapter 3.

Stagewise Orthogonal Matching Pursuit (StOMP) [52] improves upon the computational complexity of OMP by allowing multiple columns to be selected in each iteration. Where OMP only selects the single dictionary column with the highest magnitude correlation to the residual, StOMP selects all columns with correlation magnitude exceeding a threshold. As in OMP, the sparse estimate is then updated such that the residual is orthogonal to all selected columns. StOMP is much faster to execute than MP or OMP due to requiring fewer iterations than the level of sparsity. However, its performance still falls short of convex optimisation.

Regularized Orthogonal Matching Pursuit (ROMP) [53] also extends OMP by selecting

multiple columns in each iteration. However, instead of selecting the columns with the largest magnitude of correlation, ROMP selects columns with a similar size in order to improve reconstruction performance. It achieves this using a regularisation step where columns are grouped together with columns of similar magnitude and the group with the largest overall magnitude is selected. Exact reconstruction can be guaranteed for ROMP if the equivalent sampling matrix satisfies the RIP [54], though a greater number of samples are required compared to convex optimisation. ROMP requires the same sorting of coefficients as IHT in addition to the regularisation grouping. While these do not add significantly to the algorithm's run-time, they complicate implementation compared to a simpler algorithm such as OMP.

MP, OMP, StOMP, and ROMP all share a limitation which reduces their performance: columns added to the working set cannot be removed in a later iteration. This means that a poor choice of dictionary column selected in an early iteration will adversely effect the eventual solution. Two similar algorithms were introduced which allow columns to be removed from the working set with the aim of improving reconstruction performance. Compressive Sampling Matching Pursuit (CoSaMP) [55] initially follows the approach of StOMP: The dictionary columns with the largest correlation magnitudes are selected and added to the working set, after which the sparse estimate is updated by least squares approximation. CoSaMP then performs a pruning step at the end of each iteration where only the largest entries in the sparse estimate are retained and the remainder are set to zero (and the corresponding columns are removed from the working set). This improves reconstruction performance by allowing dictionary columns which perform poorly (by contributing little to the residual estimate) in later iterations to be removed. The pruning step adds minimal complexity and implementation difficulty over StOMP but allows improved reconstruction guarantees, comparable to those of convex optimisation. Subspace Pursuit (SP) [56] is very similar to CoSaMP. The main differences are that SP performs the orthogonalisation step a second time after pruning and has slightly weaker reconstruction guarantees than CoSaMP. One downside to both CoSaMP and SP is that they require an estimate of the sparsity of the signal to be reconstructed and an inaccurate estimate will adversely affect reconstruction performance.

2.2.2.4 Sparse regression

The problem of CS reconstruction is equivalent to a problem in statistics known as *sparse regression*. Regression is the process of varying a model's parameters to fit data, the best known example of which is least-squares fitting. Sparse regression introduces the constraint that most of the model's parameters must be zero or that the summed magnitude of the parameters must be limited, while still attempting to fit the data. In some cases this can give better results or prevent over-fitting compared to a simple least-squares fit. Another way of thinking of sparse regression is that we have a signal and we wish to find a sparse representation of it in some basis.

While sparse regression is mathematically equivalent to CS reconstruction, different terminology is used. In CS we try to reconstruct our *signal* $\mathbf{x}$ from the *samples* $\mathbf{b}$ taken using

the *equivalent sampling matrix* $\mathbf{A}$. In sparse regression, $\mathbf{b}$ is our *signal* and $\mathbf{x}$ is the *sparse representation* of the signal. $\mathbf{A}$ is the *dictionary*, where we refer to each column as an *atom*. We attempt to use a linear combination of a small number of the atoms to approximate the signal.

One particular technique used for sparse regression is the LASSO, introduced by Tibshirani in 1996 [2]. It turns out that LASSO is equivalent to OMP when the latter is stopped after a fixed number of iterations (and hence producing a result with a predetermined number of non-zero coefficients). LASSO can be stated as

$$\underset{\hat{\mathbf{x}}}{\text{minimise}} \ \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2 \ \text{subject to} \ \|\hat{\mathbf{x}}\|_1 < t,$$

where t is known as the regularisation parameter.

The viewpoint of sparse regression is well suited to greedy algorithms because they clearly work to approximate $\mathbf{b}$ using linear combinations of the columns of $\mathbf{A}$. I will, therefore, often use the terminology of sparse regression when describing greedy reconstruction algorithms. My algorithms and results are equally applicable to both CS and sparse regression.

2.2.3 Review of CS literature

The precursor for CS was sparse approximation using over-complete dictionaries, motivated by decomposition of fully-sampled signals rather than recovery of partially sampled signals. Mallat and Zhang [3] introduced the MP algorithm for sparse approximation with examples such as feature extraction from noisy signals. Although MP was observed to often work well, there was no general proof of its convergence on a sparse solution. Chen, Donoho, and Saunders [8] introduced the BP algorithm, using convex optimisation which could solve problems where MP did not converge on a sparse solution. This is demonstrated empirically but without any formal proof or constraints for convergence. In the absence of noise BP can be solved using linear programming, however in the much more common scenario of signals contaminated by noise the related BPDN algorithm must be used which requires more intensive quadratic programming to solve.

Using the BP algorithm of [8], Donoho and Huo [4] showed guaranteed recovery of a sparse solution when using a dictionary composed from a concatenated pair of bases (for example sinusoids and spikes). They prove that a signal may only have one highly sparse representation in such a pair of bases and that this representation may be calculated using convex ℓ_1 optimisation. This work is limited to the very specific scenario of signals composed from a basis pair but the theoretical framework formed the groundwork of future CS research. In particular they use the generalised uncertainty principle, which states that a signal cannot possess a sparse representation in two different bases if the bases have limited mutual coherence. Mutual coherence is defined as

$$\mu(\mathbf{\Phi}, \mathbf{\Psi}) = \sqrt{n} \max_{\substack{1 \leq k \leq n \\ 1 \leq j \leq n}} \left| \left\langle \mathbf{\Phi_k^T}, \mathbf{\Psi_j} \right\rangle \right|,$$

and takes values between 1 and $\sqrt{n}$ where $\mu = 1$ indicates maximally incoherent bases and $\mu = \sqrt{n}$ indicates maximum coherence.

Donoho and Elad [5] broadened the scope from dictionaries consisting of two concatenated bases to all over-complete dictionaries. Reconstruction is based on the null-space property: suppose a signal $\mathbf{y}$ possessed two distinct sparse decompositions $\mathbf{x}$ and $\mathbf{x}'$ in a dictionary $\mathbf{A}$ then the difference $\mathbf{x} - \mathbf{x}'$ must lie in the null-space of $\mathbf{A}$. If we can guarantee that any S-sparse $\mathbf{x} - \mathbf{x}'$ is orthogonal to the null-space of $\mathbf{A}$ then we know that any solution $\mathbf{x}$ must be unique. The authors introduce a new measurement spark$(\mathbf{A})$, defined as the cardinality of the smallest linearly dependent set of columns drawn from $\mathbf{A}$. If and only if spark$(\mathbf{A}) > 2S$ then no S-sparse $\mathbf{x}$ and $\mathbf{x}'$ could produce the same samples $\mathbf{y}$. There are three important shortcomings to this work: It requires signals to be exactly sparse, it is not robust against noise, and evaluating spark$(\mathbf{A})$ for an arbitrary matrix is computationally infeasible so guarantees are limited to dictionaries where spark can be estimated statistically.

Tropp [6] derives an exact recovery criterion which is sufficient to guarantee convergence on the optimal sparse representation with generic dictionaries when using BP or OMP. The guarantee is based on the incoherence of the equivalent sampling matrix, defined

$$M(\mathbf{A}) = \max_{\substack{1 \leq k \leq n \\ 1 \leq j \leq n \\ k \neq j}} \left| \langle \mathbf{A_k}, \mathbf{A_j} \rangle \right|.$$

Extending recovery guarantees to greedy pursuits was a significant achievement, but this work relies on exactly sparse signals and the absence of noise which significantly limits the application of recovery guarantees to real-world scenarios rather than synthetic test signals.

The use of sparse approximation to recover signals from incomplete measurements was introduced by Candès, Romberg, and Tao [7], motivated by the application of MRI where incomplete Fourier sampling occurs naturally. A lower bound necessary for recovery is derived but the analysis is only applicable to a limited set of bases, is not robust against noise, and requires exact sparsity. Candès and Tao [57] generalise this result: Two new properties named the Uniform Uncertainty Principle (UUP) and Exact Reconstruction Principle (ERP) are introduced, shown to be sufficient for exact recovery with overwhelming probability, and demonstrated to hold for both Gaussian and sub-Fourier measurement matrices. Furthermore, the analysis holds for compressible signals where, rather than being exactly sparse, the magnitude of sparse coefficients decays with a power-law curve. This extends applicability to signals which are smooth or have bounded variation. Candès and Tao [9] extend the analysis further by showing that exact reconstruction is possible with any general measurement ensemble which satisfies the UUP. The authors introduce the concept of a restricted isometry, where the operation of

multiplying a matrix by a sufficiently sparse vector is reversible. The RIP quantifies whether a matrix approximately preserves the magnitude of sparse vectors, and therefore whether it preserves the distance between sparse vectors. The Restricted Isometry Constant for a matrix $\mathbf{A}$ is defined as the smallest δ_S such that

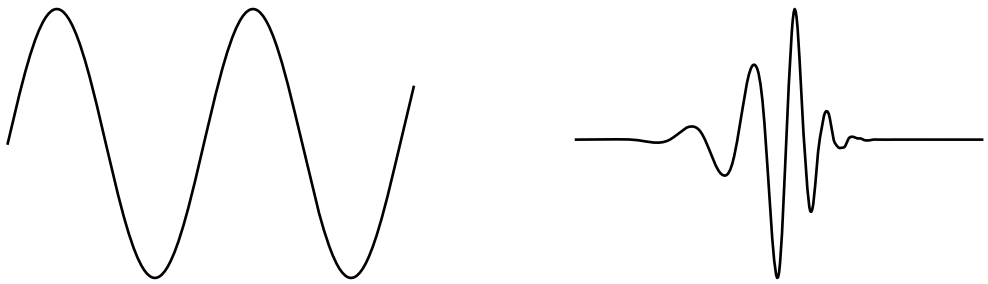
$$(1 - \delta_S)\|\mathbf{x}\|_{\ell_2}^2 \leq \|\mathbf{A}\mathbf{x}\|_{\ell_2}^2 \leq (1 + \delta_S)\|\mathbf{x}\|_{\ell_2}^2$$

holds for all S-sparse vectors $\mathbf{x}$. The matrix $\mathbf{A}$ is said to posses the RIP of order S if $\delta_S < 1$. The RIP is a strictly stronger property than the Null-space Property (NSP). Candès, Romberg, and Tao [58] extend the analysis to include signals contaminated by noise. By using BPDN stable signal recovery can be achieved for sparse and approximately sparse signals, meaning that small perturbations in the recorded samples cause only small changes to the reconstructed signal.

The RIP can be used to guarantee reconstruction when noise is present and for signals which are not exactly sparse. One shortcoming of the RIP is that it is generally computationally intractable to calculate for an arbitrary matrix (demonstrated by Tillmann and Pfetsch [59]) which can complicate sensing matrix design and analysis. Donoho, Elad, and Temlyakov [60] derive guarantees for reconstruction of noisy signals with asymptotic sparsity using BPDN and OMP but with guarantees based on the incoherence of the equivalent sampling matrix rather than the RIP. The guarantees are pessimistic in that they are weaker than those based on the RIP, but the incoherence of a sampling matrix is much easier to evaluate in practice than the restricted isometry constant. Therefore, incoherence may provide a useful tool to predict reconstruction performance when it is impossible to directly evaluate whether an equivalent sampling matrix satisfies the RIP.

2.2.4 Sparse transforms

Previously I mentioned the Discrete Fourier Transform (DFT) and Discrete Cosine Transform (DCT) as transforms which can give a sparse representation of a signal composed from relatively few sinusoidal components. These transforms can be thought of as decomposing a signal in terms of a set of atoms: finding the correct combination and proportion of these atoms to reproduce the original signal. In the case of the DFT and DCT, the atoms are sinusoids with different frequencies and phases. Thus, the DFT and DCT operate by identifying the frequency components present in a signal. The signal's representation in the Fourier domain is effectively a list of the amplitude, frequency, and phase of each sinusoid which makes up the original signal. While the DFT and DCT are effective at sparsifying signals composed from sinusoids, there are many other types of structure which may be present and which could be exploited for sparsity. The atoms of the DFT and DCT are sinusoids which exist over all of time, which makes the DFT and DCT suitable for sparsifying periodic signals. However, they have poor ability to handle local periodicity or structure in signals. The Discrete Wavelet Transform (DWT) has atoms



(a) An atom from a Fourier dictionary (b) An atom from a wavelet dictionary

Figure 2.3: An illustration of the difference between atoms in a Fourier dictionary and a wavelet dictionary

which are short, localised packets of sinusoids. Therefore the DWT is better suited to handling local periodicity in signals, and representation in the DWT domain indicates phase, frequency, and location of structures. Figure 2.3 illustrates the difference between atoms in Fourier and wavelet dictionaries.

The DFT, DCT and DWT are all isometries: the forward and reverse transforms can be applied to a signal at will without ambiguity or loss of information. Often we can achieve a more sparse representation of a signal by using an over-complete basis, a matrix with more columns than rows, so that its columns are linearly dependant. It is intuitive that over-complete dictionaries may be able to provide more sparse results: as we add more atoms to a dictionary, it is likely that signals will tend to require the composition of fewer of the dictionary atoms to be represented.

One group of overcomplete dictionaries comes from extending the wavelet concept. Wavelets are good at representing structure lying along the orthogonal axes, but poor at representation of other angles. In other words, in two dimensions wavelets compactly represent lines or edges parallel to the axes, but lines at other arbitrary angles require many more components in the wavelet domain. Several wavelet derivatives solve this problem, including curvelets and shearlets. By changing the nature of the transform and increasing the number of atoms in the dictionary, they may give a more sparse representation of certain signals (in particular images and photographs).

Dictionary learning [21] provides a technique for creating dictionaries tailor-made to compactly representing a certain class of signal, based on a set of training data (a number of representative signals). The number of atoms in the dictionary, and therefore the level of overcompleteness, may be chosen at the time of training. While dictionary learning is very compute-intensive and requires careful selection of the training data, it can produce the most sparse representations of a class of signals due to its targeted nature.

A downside to overcomplete dictionaries is that the transforms are no longer isometries.

While a signal only has one valid representation in the Fourier domain, for example, it has infinitely many representations in the curvelet or shearlet domain. This means more care is needed (and generally more computation required) when decomposing a signal in order to find a sparse representation.

2.2.5 Fast transforms and implicit dictionaries

There are several ways to represent a dictionary or basis, and correspondingly different ways in which reconstruction algorithms are implemented around them. The most intuitive is to store the dictionary as an explicit matrix in memory, having either generated it, loaded it from disk, or received it over a network. We then use the dictionary in our algorithm by performing mathematical operations on it, for example performing matrix-vector multiplications using it, calculating its norm, or calculating various decompositions. This is the most flexible implementation, as we have no limitation on choice of dictionary or on the mathematical operations we can carry out on the dictionary. The first major downside is that if the dictionary is very large, it requires that we have a large amount of main memory to store it: our quantity of main memory limits the maximum size of dictionary we can deal with. The other major downside is that this implementation may be the slowest compared to the other options. Nonetheless, this approach is the most generic and flexible, and is the only one suitable for the unpredictable and unstructured dictionaries produced by dictionary learning.

Another option is to store the dictionary implicitly. This can be applied to any matrix which is predictably and procedurally generated from an appropriate function or algorithm. With an explicit matrix we would have generated it ahead of time, stored it in memory, and accessed each element from memory when required. With an implicit matrix, we calculate each element when required instead of fetching it from memory. Since we are no longer holding the dictionary in memory, we remove the memory constraint. This technique may also be faster than using an explicit dictionary: if the generating function for each element is simple, it may be faster to calculate the element than fetch it from memory. In a distributed-memory environment, generating the matrix implicitly would save communication between nodes when they require access to part of the dictionary stored on another node. While implicit storage limits our choice of dictionary, we can still perform any desired mathematical operation on the dictionary, although operations with memory requirements proportional to dictionary size may be limited.

The final option is fast transforms. For certain structured matrices there are fast algorithms for performing matrix-vector multiplications which are asymptotically faster than performing the multiplication explicitly. Examples include the FFT (based on the DFT), the Fast Wavelet Transform (FWT) (based on the DWT), and the Fast Hadamard Transform, based on the Hadamard Transform. When appropriate, this approach will usually be the fastest to execute as well as eliminating the memory requirement of storing the dictionary explicitly. However, we are very limited in our choice of dictionary as only certain structured dictionaries have associated

fast transforms. We are also very limited in what mathematical operations we can carry out on the dictionary: usually fast transforms only allow matrix-vector multiplication by the dictionary and its transpose, i.e., calculating $\mathbf{Ax}$ and $\mathbf{A^T b}$. However, many reconstruction algorithms can be implemented using just these two operations on the dictionary.

2.2.6 Applications

A classic application of CS is MRI. The motivation is to capture the clearest image possible from a limited number of samples. The number of samples is limited because carrying out each sample takes a non-trivial amount of time, and the patient can only stay still for a small amount of time, so taking too many samples results in a blurry image. The hope is that CS can give a better reconstruction from the samples than a more naive reconstruction strategy. Another convenient property of MRI is that linear sub-sampling is inherent to the process. The physics of MRI mean that each sample taken is a set of points along a line in the Fourier transform of the image. Multiple samples are taken by changing the angle of the line. The final piece of the puzzle is a sparsifying basis. As with most real-world images, MRI images are sparse in the wavelet domain. Candès, Romberg and Tao [7] showed how CS can be used to give perfect reconstruction in a simulated MRI problem.

Another widely reported application is the single pixel camera, described by Wakin et al. [11]. A standard digital camera focuses the image onto a sensor. The sensor is comprised of many pixels arranged in a grid, each sensitive to the intensity of light. The grid of pixels on the sensor corresponds to the pixels we see in the resulting digital image. With the single pixel camera, there is only a single sensor pixel. A micro-mirror array or liquid crystal filter is used to selectively allow light through certain pixels onto the sensor. The image could be fully sampled by opening one pixel at a time and collecting all the readings. Alternatively, the pixels can be opened in combinations and the image linearly sub-sampled, and reconstructed using CS. This technique is appealing where the sensor pixels need to be large or are expensive, for example an infrared camera.

In the applications considered so far, we aim to capture and reconstruct a single signal with a sparse representation. In cases where we wish to capture several related signals, Distributed Compressive Sensing (DCS) can be used to exploit the similarities between the signals and enable further sub-sampling than if the signals were captured and reconstructed in isolation. DCS is similar to the theory of Distributed Source Coding which is concerned with the transmission rate required for joint decoding of independently coded correlated sources, for example Slepian and Wolf [61] describe noiseless decoding of two correlated sources but do not account for correlation or redundancy within each source. DCS, introduced by Duarte et al. [62], gives a new mechanism to achieve joint decoding for any number of signals while exploiting both the correlation between signals and within each signal. Two Joint Sparsity Models (JSMs) are defined which each describe a model of the structure of intra-correlation within each signal

and inter-correlation between the signals. With JSM-1, all signals possess a sparse common component and individual sparse innovation. Equivalently, every signal is sparse and the difference between any pair of signals is also sparse. Under JSM-1, reconstruction of all signals is simultaneously carried out using the modified BP described by Baron et al. [63]. A significant downside to this approach is that all signals must be reconstructed simultaneously: If the cost of reconstruction grows super-linearly with the size of signal, then DCS reconstruction may be much more costly than separately reconstructing each signal. One situation in which JSM-1 may apply is a number of sensors spaced over an area collect correlated readings. JSM-2 models the situation where the sparse representation of every signal shares the same support but the signals differ in their coefficients. Reconstruction in this case is carried out using the DCS-SOMP algorithm described by Duarte et al. [62]. An example of when JSM-2 may apply is when the same signal is captured by sensors in different locations so that the observed signals vary only in time delay and phase. In addition to reconstructing the captured signal, this setup could be used to localise the signal source.

DCS is of particular interest in Wireless Sensor Networks (WSNs) for several reasons. WSNs often seek to capture related signals or the same signal from spatially diverse locations, so the observed signals are likely to be correlated. The radio bandwidth between nodes is often limited so compression is desirable, as is independent sampling The sensor nodes usually have limited battery capacity and compute power, so simple CS sampling is desirable in comparison to the more power-hungry traditional compression encoding.

2.3 High performance parallel computing

In this section I will give an overview of high performance parallel computing in order to give crucial context, as well as motivation, to my work. I will describe the types of modern parallel computer and parallel computing architectures, including a definition of relevant terminology which will be used through the rest of this thesis. Finally, I will show how parallelism has been entwined with computing throughout its history and how this influences modern parallel computing paradigms.

2.3.1 Types of parallel computer

There is a wide variety of parallelism found in computing, varying in the underlying mechanism, how the parallelism is handled by the programmer or end user, and the performance improvements one might expect to gain. In order to impose order on this variety, a system of categorisation is needed. One of the earliest systems for categorising computer parallelism architectures is Flynn's Taxonomy, introduced by Michael Flynn in 1966 [41][42]. Flynn's Taxonomy groups architectures by the number of instruction streams and the number of data streams. The four original groups are: Single Instruction Single Data (SISD), Single Instruc-

	Single data	**Multiple data**
Single instruction	Single Instruction Single Data (SISD): Sequential processors with no parallelism	SIMD: Vector processors, GPUs
Multiple instruction	Multiple Instruction Single Data (MISD): Fault-tolerant processors	MIMD: Multi-core, multi-processor, and cluster computing

Table 2.1: Flynn's Taxonomy

tion Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). A summary of Flynn's Taxonomy is shown in Table 2.1.

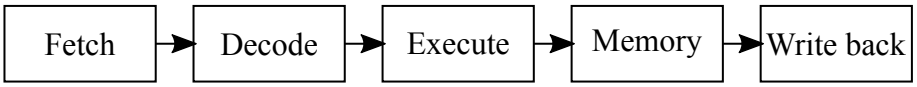
A traditional, fully sequential, computer is SISD: it processes a single sequential stream of instructions, with each instruction operating on a single piece of data. SIMD computers maintain the single sequential instruction stream but each instruction may act on a large number of pieces of data in parallel, carrying out the same operation on each piece of data. MISD computers are rarely encountered but are occasionally used for fault-tolerance, where multiple instructions are used redundantly and the results compared; we will not discuss this type of computer henceforth. MIMD computers process a number of independent instruction streams, each with its own associated data.

2.3.1.1 SISD: Instruction-level parallelism

Although SISD describes a type of computer generally considered as purely sequential, some parallelism is often still present. The main distinction is that the computer appears sequential to the programmer's and user's point of view; any parallelism is hidden internally and manifests as improved sequential performance (higher clock speed or more instructions per clock).

One near-universal form of SISD parallelism is pipelining, where each instruction is split into several stages (e.g. fetch, decode, execute, memory, and write back); Figure 2.4 shows a simple example of this. Pipelining allows the instruction clock frequency to be increased by splitting up the latency of a complete instruction. Instructions begin executing prior to the completion of earlier instructions which may include branches, so the processor must predict which path will be taken by a branch; a misprediction will cause a gap in processing while the pipeline is flushed and restarted. A great deal of effort (and silicon area) in modern high-performance processors goes into effective branch prediction in order to avoid misprediction and keep the pipeline full.

Another form of SISD parallelism (which may be combined with pipelining) is superscalar architectures: the processor is designed with duplicates of some functional units, and instructions from a single instruction stream are issued to multiple processing elements in parallel in order to improve throughput. The most common form of superscalar architecture is dynamic multiple-issue where the processor decides at run-time how to parallelise the instructions; no intervention is needed by the compiler. A less common form of SISD parallelism is static multiple-issue,



(a) Breaking down an instruction into 5 stages

Clock cycle	1	2	3	4	5	6
Instruction 1	Fetch	Decode	Execute	Memory	Write back	
Instruction 2		Fetch	Decode	Execute	Memory	Write back
Instruction 3			Fetch	Decode	Execute	Memory
Instruction 4				Fetch	Decode	Execute
Instruction 5					Fetch	Decode

(b) The flow of 5 instructions through this 5-stage pipeline

Figure 2.4: A simple example of instruction pipelining

where the compiler determines which instructions may be executed in parallel and statically encodes this information in the compiled program. One example of this is the Intel Itanium IA-64 architecture which uses Explicitly Parallel Instruction Computing (EPIC), an evolution of Very Long Instruction Word (VLIW) architecture [64][65].

2.3.1.2 SIMD: Data parallelism

This architecture is relatively easy to implement, but more difficult to program for, and significant speed-up is limited to certain amenable applications: A high number of general purpose programs cannot benefit significantly from SIMD parallelisation.

Some of the earliest supercomputers, known as Vector Processors, implemented SIMD parallelism. Memory access had quite a high latency, so fetching instructions and fetching and storing data severely limited the instruction rate which could be achieved. Throughput could be improved by having each instruction able to operate on a large amount of data. When this suited the program being written, high throughput (measured in Floating Point Operations per Second (FLOPS)) could be achieved.

SIMD parallelism also appeared in personal computers in the form of processor extensions, including Intel's MMX, SSE, and AVX families and AMD's 3DNow. While these provide limited benefit to many programs they are widely used for multimedia processing.

Finally, modern GPU accelerators have a large number of simple processing cores which all perform the same operation on a large quantity of data, thus implementing SIMD parallelism.

2.3.1.3 MIMD: Task parallelism

Finally, the most powerful and most flexible form of parallelism is MIMD, or task parallelism. Multiple independent processors each execute their own separate instruction stream, generally operating on different pieces of data. There are a broad range of types of MIMD parallel

computer. These can be most easily categorised by considering the way they interact with memory.

Shared memory parallel computers all access the same coherent memory and share a unified memory address space. Separate tasks often operate on separate areas of memory, however any communication required between the tasks will be done by accessing the same memory. Multiple simultaneous tasks accessing the same memory requires a great deal of care to maintain consistency, especially when considering the addition of multiple caches. For this reason, shared memory multiprocessors are limited in scale. Shared memory parallelism is commonly used within a single computer and is no longer used to build large super-computers.

Shared memory task parallelism can be further divided into Uniform Memory Access (UMA) and Non-uniform Memory Access (NUMA). In UMA computers every processor (or processor core) has equal access to the whole of main memory, in terms of both bandwidth and latency. Cache collisions may cause issues if two processors try to work in the same area of memory, but otherwise the program running on each processor may be unaware of the parallelism. Multi-core processors are an example of UMA shared-memory task parallelism.

In NUMA computers, processor cores and blocks of memory are grouped (for example, per processor socket). Within each group, every core has full access to that group's memory. Cores may freely access memory in another group, but bandwidth will be more limited and latency may be higher. Thus naive programs can run in a NUMA environment, but performance will benefit if programs more intelligently utilise the NUMA domains. NUMA computers are generally those with multiple discrete processors in separate sockets, where each socket is generally associated with a portion of memory. Access to another socket's memory imposes a performance penalty. Each socket may include multiple cores, so a UMA domain exists nested within the overall NUMA system. Figure 2.5 shows a two-level NUMA architecture: the 4 compute cores in each processor socket share a NUMA domain, as do the two sockets in a compute node. The greatest bandwidth and lowest latency exists between cores in a processor socket followed by the communication link between two sockets in a node, followed by the network linking the two compute nodes.

The vast majority of modern supercomputers are distributed memory systems. Each node has fast access to its own main memory and much slower access to other nodes' memories. Communication between nodes occurs by message passing (For example MPI, discussed in

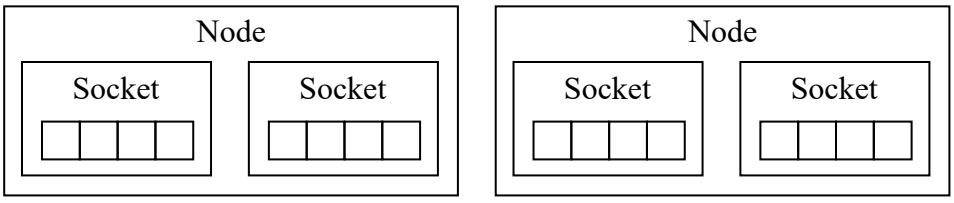


Figure 2.5: A two-level NUMA architecture resulting from two compute nodes each containing two separate processor sockets. Each socket is shown with 4 cores.

detail later), or by explicit remote memory access. Because communication between nodes is so much slower than memory access within a node, it is important that programs intelligently utilise the platform to achieve acceptable performance. Since the program must be explicitly written for the parallel platform, it utilises explicit parallel operations.

By accepting that links between nodes will be much slower than memory access, distributed memory computers unlock massive scalability. They may possess vast quantities of memory and computational power which would be impossible to implement in a shared memory system.

One may simulate a shared memory computer running on a distributed memory platform, by creating a virtual uniform address space and channelling remote memory accesses over message passing (effectively a form of automatic parallelisation). While this greatly eases the burden on the programmer, it is likely to result in much poorer performance than a program specifically written for the distributed memory environment, which can more intelligently ensure data locality and reduce communication over the slower inter-node interconnects.

2.3.2 Interconnect architecture

Most parallel computation will require communication between the individual processors throughout the computation. This communication may be over short, high quality links in a single data-centre in the scenario of cluster computing, or may utilise slow, unreliable links over the internet in the case of grid or distributed computing.

Often, the computation cannot proceed until communication completes, so communication time could become a significant barrier to high performance. An individual interconnect link has two fundamental properties: latency and bandwidth. Latency is the time taken to transfer a small message, which includes the propagation time of the medium (limited by the speed of light) and overheads in the network hardware and software. Bandwidth is the rate at which information may be transferred. The overall communication time is the latency, plus the message size divided by the bandwidth. Latency is more important for small messages, while bandwidth makes a bigger difference for large messages.

A variety of interconnection topologies may be used in cluster computing. A few important properties of the topology are:

- Number of links: The total number of links present in the topology, which determines the cost of constructing this interconnect topology.
- Bisection width: The typical communication bandwidth available for half of the nodes to simultaneously communicate with the other half. It is calculated by counting the minimum number of links which would have to be cut to divide the network into two halves, and multiplying this by the bandwidth of a single link.
- Diameter: The maximum number of links which must be traversed to communicate between two nodes, which determines the maximum latency of a communication on the

topology.

In some topologies the bandwidth and latency may differ between different pairs of nodes, so parallel programs may have to exploit knowledge of the topology and node locations to minimise communication time.

2.3.3 Terminology

- *Cluster*: A group of compute *nodes* in a single location with high performance *interconnect*, often connected to a large quantity of bulk *storage*.
- *Node*: A single computer in the cluster. It usually combines a number of compute *cores* with a quantity of main memory, and may have some local disk storage.
- *Core*: A single processor core within a compute node. Each core can run a single process or thread.
- *Socket*: A single processor. It contains one or more processor *cores*. A *node* may contain one or more processor sockets. Multi-socket systems are usually NUMA, with each socket tightly-coupled to a section of main memory.
- *Worker*: A single MPI process, usually running on a single processor core. Algorithm design is usually concerned with the workers involved, without regard for the physical nodes, processors, and cores which execute the workers.
- *Storage*: Clusters usually include a large quantity of storage for holding data input to, and output from, the programs running. Whilst of high specification and usually connected to the high performance interconnects, disk storage is still relatively slow compared to main memory access.
- *Interconnect*: The network used to connect compute nodes together.

2.3.4 History

Parallel computing may feel like a modern trend, but parallel computers have existed since the early expansion of digital computing in the 1960s: The Burroughs B5000 included asymmetric multi-processing capacity [38] and the Burroughs D825 has been called the first true symmetric multi-processing system [39].

The next step in large scale parallelism was the introduction of large-scale SIMD supercomputers. Instead of fast or parallel execution of a generic instruction stream, these computers focussed on quickly performing the same operation on large arrays of data. Notable early vector processors included the TI ASC [66], CDC STAR-100 [67] and ILLIAC IV [68], operational in 1966 and 1975 respectively. The first commercially successful vector supercomputer was the

Cray-1 [67], created in 1976, and Cray Research continued to develop vector supercomputers through the late 1970s and the 1980s.

In the 1990s, large scale cluster computers were introduced by Fujitsu [69], Intel [70], and Cray [71]. Where the previous vector machines were programmed as a single computer with large memory and fast execution of instructions, cluster computers were programmed from the point of view of individual processors, with explicit communication (and high bandwidth low latency interconnects) required to exploit parallelism. This strategy for parallel computing is typified by the MPI [44], first introduced in 1992 and still in wide use today (including by my own work).

Up until the 2000s, the increase of speed in personal computers was enabled by an increase in clock speeds [47], permitted by the shrinking of transistors on integrated circuits, and the inclusion of exponentially more transistors on each die (as described by Moore's Law). By the 2000s personal computer clock speeds had passed 1GHz and were beginning to level off. Aside from further increasing clock speed, the only remaining strategies to increase performance were increasing the number of instructions executed per clock cycle (through improved caches, pipelining, hyperthreading, and branch prediction), or introducing parallelism into personal computer processors. For many years these processors had included SIMD parallelism (for example the MMX, SSE, 3dnow, and AVX extensions to x86 [67]), but true MIMD multiprocessing was first introduced on the desktop in 2005 by the AMD Athlon 64 X2 and the Intel Pentium D.

Today, parallel computers are ubiquitous. The most basic personal computers have at least two processor cores, and high-powered desktops often include 16 processor cores or more. Even smart-phones generally have at least two processor cores, and sometimes as many as 8.

Modern supercomputing is entirely based on parallelism. Supercomputers are usually clusters, utilising vast numbers of commodity servers with a high-speed interconnect network.

2.4 Parallelisation

Most computer programs, when implemented in the obvious manner, are sequential: they are executed one part after another, with each section depending on results from the previous section. Parallel compute resources can be used to run several sequential programs at the same time, but significant work is normally needed before a sequential program can usefully exploit a parallel computer.

I begin by giving a summary of common metrics used in parallelisation. I will refer back to these when discussing the merits of various parallelisation techniques. I will then explain the theory and strategies used for effective parallelisation and finish by giving an introduction to MPI, which is used in my algorithms in the following chapter.

2.4.1 Performance metrics

- *Wall-time* is the time taken for a program to execute, hence also known as execution time. It is named because it is the time elapsed as would be measured by a simple wall-clock. Wall-time is an ideal metric to minimise if we are parallelising an algorithm to reduce how long a user has to wait for results, or if we wish to reduce the latency in a process. Wall-time is denoted as T. When comparing parallel and serial algorithms, their wall-times are denoted T_p and T_s respectively.

- *Total time* is the wall-time of a parallel algorithm multiplied by the number of workers W executing the algorithm. Total time is defined as WT_p. This measures the total amount of computation put into solving the problem and is related to the energy consumption of running the algorithm.

- *Overhead* is the difference in total time between a serial algorithm and a parallelised version, $WT_p - T_s$. Overhead is a measure of the inefficiency introduced in a parallel algorithm in exchange for a reduction in wall-time. It may be caused by communication latency, or by parallelisation necessitating a less efficient algorithm than a serial implementation.

- *Speed-up* is a dimensionless measure of the reduction in wall-time due to parallelisation, defined $S = T_s/T_p$.

- *Efficiency* is defined as speed-up divided by W. Efficiency is a useful measure of the level of overhead in a parallel algorithm.

If computation is evenly divided over W workers with no overhead then the speed-up would be W, and one might expect this to be the ideal case and hence the maximum speed-up. However, it is possible to observe super-linear speed-up greater than W (which would correspond to a negative overhead and efficiency greater than 1). This can occur for a variety of reasons (many of which are discussed by Ristov et al. [72]) but the simplest explanation is if the parallel computer has increased memory along with its compute and so allows holding the whole problem in main memory, while the corresponding serial computer would have to access the problem from disk resulting in significant slow-downs.

2.4.2 Theory

There are several reasons we might want to utilise a parallel computer. The most obvious is that we want to run the program faster, that is, we want to reduce the wall-time of a program. Another motivation is if we want our program to work on a dataset larger than we can hold in main memory of any one computer. A cluster can easily have a massive sum quantity of main

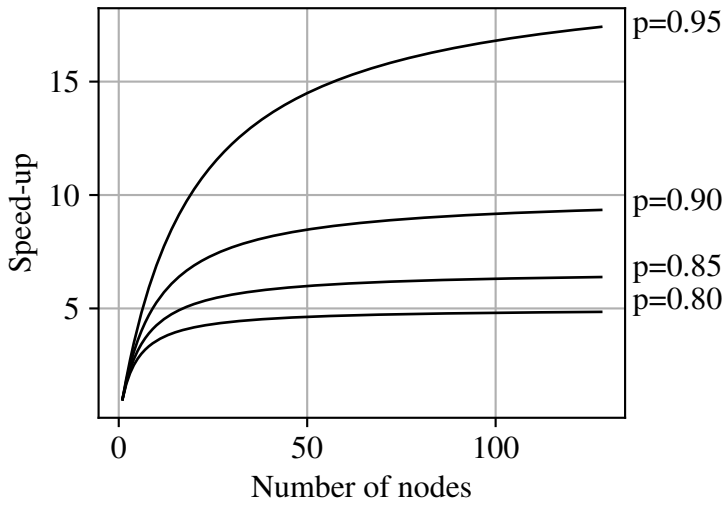


Figure 2.6: Speed-up predicted by Amdahl's Law for 1 to 128 parallel processors and four different values of parallel proportion p.

memory, and each cluster node can access its main memory much faster than if we tried to store the dataset on bulk storage (i.e., hard disk drives).

When parallelising with a goal of reducing execution time, Amdahl's Law, introduced by Gene Amdahl in 1967 [73], is a useful concept to bear in mind. It considers a hypothetical situation where an improvement in compute resources causes speed-up in a portion of a program, but no speed-up in the remainder of the program. The overall speed-up depends greatly on what proportion of the program is affected by the speed-up, in addition to the level of speed-up. Amdahl's Law is defined as

$$S = \frac{1}{1 - p + \frac{p}{s}}, \tag{2.6}$$

where S is the overall speed-up of the whole program, s is the speed-up of the portion of the program affected by our improvement in compute resource, and p is the proportion of the program benefiting from our improvement in compute resource.

Consider a hypothetical program where part of the work is perfectly parallelisable, and the other part is fully sequential or serial. When more parallel processors are tasked with executing this program the parallel portion gets faster in proportion, but the sequential portion takes the same amount of time. Thus, the speed-up from parallelisation depends not only on the number of processors used, but also on the proportion of the program which may be parallelised.

Amdahl's Law gives particularly discouraging results for massively parallel computers where speed-up is very sensitive to the proportion of the program which is parallelisable. Figure 2.6 shows a chart of predicted speed-up against number of parallel processors for problems with four different parallel proportions. With a large number of workers, small changes to the parallel proportion of the program make a massive difference to the speed-up achieved, and the maximum

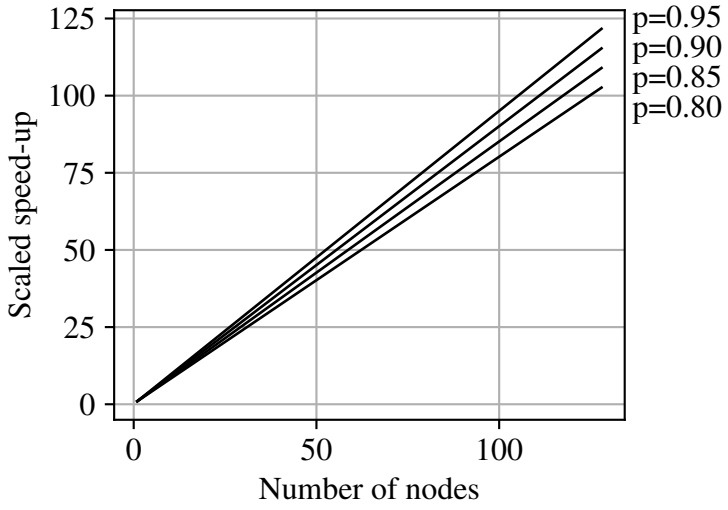


Figure 2.7: Scaled speed-up predicted by Gustafson-Barsis' Law for 1 to 128 parallel processors and four different values of parallel proportion p.

speed-up is asymptotically limited for a given program with a fixed parallel proportion.

A contrasting view is given by Gustafson [74], who observes that the size of problem submitted by users tends to grow with the number of parallel processors in a computer. In many cases a user has a choice over the size of problem they input to a program, for instance the resolution of an image to process or the grid density in finite element analysis. If the user cares more about the program's execution time than the size of problem solved, they may adjust the problem size to achieve a tolerable execution time. Furthermore, if a parallel program has overheads of fixed cost with respect to problem size, then the fraction of a program which is parallelisable grows directly with problem size. In the ideal case where program execution time is fixed and parallel fraction is proportional to number of parallel processors, this leads to predictions very different to those of Amdahl's law.

The Gustafson-Barsis Law introduces a metric called *scaled speed-up*. For the scenario of a parallel computer solving a problem the size of which is proportional to the number of parallel processors available, we can calculate how much longer a serial computer would have taken to solve the larger problem. If a serial computer would have taken 50 times longer for this larger problem, we say the scaled speed-up is 50. Using the same definitions of s and p as Equation 2.6, scaled speed-up can be calculated as

$$\text{scaled speed-up} = 1 + (s - 1)p.$$

Figure 2.7 shows a chart of scaled speed-up for the same 1 to 128 parallel processors and four parallel proportions as Figure 2.6. When many parallel processors are available, the difference between speed-up and scaled speed-up can be very large. Under Gustafson's assumptions about the use of parallel computers, scaled speed-up is much less sensitive to the parallel proportion

of a program than Amdahl's Law.

Amdahl's Law could be described as pessimistic about the utility of parallel computing, while Gustafson's theory is correspondingly optimistic. Problem size increasing with the level of parallelism available is a best-case scenario and end users may desire decreased execution time in addition to the processing of larger problems. While superficially similar to speed-up, scaled speed-up has a very different meaning and the two are not directly comparable: Speed-up describes reduced execution time as number of parallel processors increases and other variables are kept constant. Scaled speed-up is effectively describing an increase in data throughput rather than a true reduction in execution time. Furthermore, Gustafson's assumption that non-parallelisable overheads remain fixed regardless of problem size is also very optimistic: In many cases one would expect the communications overhead to also grow with increased problem sizes. Realistic results are likely to lie between the predictions of Amdahl's Law and Gustafson-Barsis' Law: massively parallel computers are likely used to solve larger problems than serial computers but problem size is not directly proportional to the number of parallel processors and serial overheads also grow with problem size.

When analysing a parallel program, we can think of overheads due to communication and coordination latency as adding to the serial part of the program which cannot be parallelised. The amount of serial overhead will limit the speed-up we can achieve by adding more processors. Therefore, reducing overhead (and therefore minimising communication) and avoiding serial bottlenecks are key to developing a parallel program which can achieve good speed-up with many processors.

The simplest programs to parallelise are known as *embarrassingly parallel* problems. A problem is embarrassingly parallel if it can be easily split up into pieces which can be processed in parallel. Each piece may be processed without communication or coordination between the elements of the parallel computer. Generally only some minimal communication is required at the start, to distribute the problem across the parallel computer, and at the end, to collect the results. A simple example would be graphics rendering, where each frame (or even small pieces of a frame) can be rendered in parallel without requiring the results from previous renderings.

In practice, most algorithms we want to parallelise are not embarrassingly parallel, so more work is needed. Various strategies exist to develop parallel implementations of an algorithm. The most effective strategy will usually depend on the specifics of the algorithm. For some algorithms it may be the case that there is no effective strategy for parallelisation.

As mentioned previously, Amdahl's Law tells us that to achieve maximum speed-up from parallelisation we need to minimise the serial portion of the program, which includes minimising the number of communications required during parallel processing. This may lead to a trade-off, where the overall most efficient algorithm is unsuitable for parallelisation, but a less efficient algorithm may be more amenable to parallelisation, which could lead to a faster parallel implementation at the expense of efficiency. Another way to minimise the impact of communication overheads is to overlap communication with processing. If the environment and algorithm

allow, the program may be able to carry out useful processing while waiting for communications to complete, effectively reducing the communication overhead. However, this may not be possible if dependencies between communication and processing give inadequate flexibility for re-ordering, or if the communication mechanism cannot be carried out asynchronously.

Another important consideration is the distribution of load between parallel workers. Assuming that parallel workers work in lock-step (no worker can proceed until all workers complete a task), effort must be made to evenly distribute work. If there is a significant disparity between the quantity of processing carried out on different workers, some workers will *stall* while waiting for others to complete, effectively introducing added overhead and reducing speed-up and efficiency.

Factors to consider from a potential parallelisation strategy include granularity and degree of concurrency. Granularity describes how many pieces the problem can be broken into. A problem exhibiting fine granularity can be broken down into many small pieces, whereas a problem with coarse granularity has less potential for being broken down. Granularity will influence how many parallel processors we can exploit: a problem with course granularity may have a restrictive limit on the number of parallel processors which can be usefully utilised. The degree of concurrency describes more precisely how many parallel processors can be utilised: it is the number of parallel activities which can be carried out in part of a program. Over the course of a program, one can quantify the maximum and average degree of concurrency, which give a good indication of how many parallel processors may usefully be exploited.

Foster's Methodology [48] is a generic technique for parallelising a program. A simple example is illustrated in Figure 2.8. The steps of Foster's Methodology are as follows:

1. *Partitioning*: Identify how the problem can be broken down into smaller pieces, each involving largely independent computation.

2. *Communication*: Identify the communication required by each of the small tasks resulting from partitioning. This includes distribution of inputs to tasks, collection of outputs from tasks, and communication required between tasks.

3. *Agglomeration*: To simplify the problem, combine tasks where no utility is gained by keeping them separate. For example, if one task must always follow another and has no other dependency, the two can probably be combined with no loss of flexibility.

4. *Mapping*: Consider how the remaining concurrent tasks can be mapped onto the parallel processors available. If the duration of each task is known beforehand then mapping can be carried out in advance. If the duration of tasks is uncertain, mapping can be carried out at run-time.

Problems are commonly partitioned using two categories of technique: task partitioning and data partitioning. The division shows some semblance to the difference between SIMD

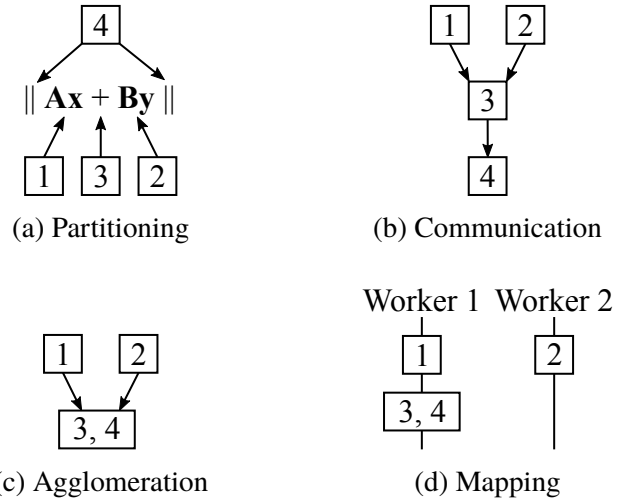


Figure 2.8: A simple example of Foster's methodology for parallelisation. The calculation is partitioned into two matrix-vector products, an addition, and a norm. The addition and norm are combined then the result is scheduled for two workers.

and MIMD parallel computers, and indeed some of techniques used are quite similar. Other partitioning and parallelisation techniques exist, including recursive, speculative, and exploratory decomposition. Furthermore, techniques may be combined when decomposing a single problem, referred to as hybrid decomposition.

2.4.2.1 Task-based parallelisation

Task-based parallelisation involves breaking up the algorithm into distinct tasks which can be carried out at the same time. For example, if we were trying to calculate $\mathbf{Ax} + \mathbf{By}$ with large matrices and vectors, we could have one worker calculate the product $\mathbf{Ax}$ and another calculate the other product, $\mathbf{By}$. One of the intermediate vectors would then have to be communicated so the addition can be carried out.

To plan a task-based parallelisation strategy it is useful to draw a task dependency graph. The algorithm is broken up into distinct tasks, with each task represented by a graph node, and an edge is drawn for each dependency where one task must be completed before another. Figure 2.9 shows a simple task-dependency graph for the example of calculating $\mathbf{Ax} + \mathbf{By}$. A task-dependency graph may contain cycles, where part of the algorithm is executed repeatedly with its output being returned to the input. This is the essence of iterative algorithms. If the task dependency graph is overall mostly linear, then a pipelining strategy may be appropriate: In the

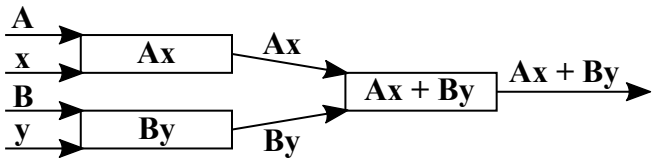


Figure 2.9: A task-dependency graph for a very simple algorithm

same way as instruction pipelining in processors, each distinct task is carried out in parallel. Each entry traversing through the algorithm takes the same amount of time, but the overall throughput can be increased. If the algorithm is broken up into more tasks, then more parallel processors can be exploited to increase throughput. However, if the time taken to complete each task becomes comparable to the time taken for communication then the overheads will begin to limit the speed-up and efficiency achievable. If the task dependency graph contains cycles (i.e., an iterative algorithm) then pipelining cannot be used, because the data needed to begin a traversal of the graph is not available until the previous traversal has completed.

If the task dependency graph has a more dense structure then it can be considered as a tree, with computation beginning at the leaves and moving towards the root node of the tree, which is the overall result of the algorithm. Adjacent items in the tree can be computed at the same time, reducing the wall-time of the overall algorithm. The level of parallelisation possible depends on the width of the tree.

It may be appropriate to decouple the different tasks from the parallel workers available, particularly if there are many more parallel tasks than processors available. In this case, a worker pool is a useful architecture. The parallel processors are dynamically assigned tasks as they occur. This means a worker can either process a single long task or a number of shorter tasks without stalling.

2.4.2.2 Data-based parallelisation

The other common parallelisation strategy is data-based parallelism. This is generally appropriate where large quantities of data are being processed, and particularly where computation done on the data is generally localised, that is, intensive processing is done on specific areas of data with minimal interaction needed from the processing of other areas of data. The size of data and level of processing locality determines how amenable an algorithm will be to data parallelism. Poor locality will introduce the need for excessive communication which introduces overheads and reduces efficiency. Larger quantities of data tend to increase the cost of processing, thus reducing the relative burden of overheads and increasing efficiency. Unlike task-based parallelism, data parallelism can be applicable to iterative algorithms.

To implement data-based parallelisation, the developer must decide whether to break down the problem based on its input data, its output data, or intermediate data. For example, developing an algorithm to search for a short string or pattern in a large set of data, input-based parallelisation would be suitable: split the large set of data over a number of workers, each of which can perform the search over its own domain and halt the algorithm if the pattern is found. For a program designed to generate a high resolution image of the Mandelbrot set, output-based parallelisation would be appropriate: break the output image into sections and assign each section to a worker, whose job it is to populate that segment with the appropriate pixel colours.

In some cases the same parallelisation structure may be apparent from more than one of the input, output, or intermediate data, if they are highly related. For example, if performing a

colour correction on an image, the colour of an output pixel depends primarily on the colour of the corresponding input pixel. Once the problem's datasets have been partitioned, compute can also be partitioned such that compute tasks are localised to the data they interact with.

2.4.3 Message Passing Interface

The Message Passing Interface (MPI) is a communication standard widely used in distributed memory compute clusters. MPI was introduced by a group of researchers in 1991–1993 in an attempt to develop a standardised communication protocol for distributed-memory cluster computing. MPI was intended not only to provide a standard interface to programmers, but also to manufacturers of interconnect networking hardware. This would allow programmers to create software which could run on any compute cluster, regardless of the underlying communication technology.

MPI provides a relatively low level interface to programmers, with the intention of providing a high performance base upon which software can be developed directly, or higher level abstractions created if needed. The lack of abstraction means the programming interface closely reflects the underlying hardware concepts, helping the programmer design the program around efficient use of communication, where a higher level abstraction might encourage more gratuitous use of communication leading to worse performance.

From the programmer's point of view, MPI's primary interface is language bindings provided in C and Fortran, both commonly used scientific computing languages. Bindings exist for a vast number of other languages, developed by third parties and based on the C or Fortran bindings. On the hardware side, MPI supports a wide array of communication and interconnect technologies including TCP/IP sockets (which are available on any standard network) and Infinniband (a group of dedicated high performance fibre interconnect standards).

MPI models the parallel computer as a number of *workers*. Generally each worker will correspond to a single process, and each node in a cluster will run as many workers as it has processor cores. Sets of workers are combined into *communicators*, which allow any worker in the communicator to communicate with any other worker in that communicator (or with the entire communicator). Multiple communicators may exist simultaneously, and each worker may simultaneously be a member of multiple communicators. Within each communicator workers are numbered, referred to as that worker's *rank*. The same worker may have a different rank in each different communicator it belongs to. To begin with, workers have access to a single communicator called *world*, which contains every worker taking part in this parallel program. Further communicators are made by creating *groups*, which can be created by duplicating or splitting a communicator. Groups may be used for organising point-to-point communications but are particularly useful for collective communications, in which every member of a communicator takes part. A worker specifies which communicator to use when executing a communication primitive.

I will now give a brief summary of the communication primitives MPI provides, in particular those which I later use in my own algorithms.

2.4.3.1 MPI Primitives

Most MPI operations involve the transfer of data. MPI standardises a number of data types (for example *integer* and *double*) in order to allow interoperability between the different programming languages and platforms, which may use differing and incompatible native types. MPI also provides for the creation of derived data types, including arrays and structures composed from the primitive data types.

Many operations are available in asynchronous or synchronous variants, as well as providing buffered or unbuffered options. Asynchronous operations allow a worker to carry out other tasks while waiting for communication to complete. Synchronous operations stall program flow until communication is complete. Asynchronous operations allow for potentially increased efficiency compared to synchronous operations by negating the communication overhead, however, asynchronous operations require more care to maintain consistency. Buffered operations immediately copy data to be transmitted by an asynchronous operation. The advantage of this is that a worker may then modify the data structure before waiting for communication to complete. The disadvantage is that the data must be copied, imposing a time and memory penalty which may be significant for large pieces of data.

In addition to being confined to a communicator, MPI messages can include a tag, used by the programmer to identify or categorise a message. Operations which send a message include a tag value to attach. Operations which receive a message may specify the tag they expect: any messages with different tags are ignored.

MPI primitives can be simply divided into three categories. Point-to-point messages involve sending a message from one worker to another. Collective operations involve any number of workers communicating. Synchronisation primitives are used to control program flow and ensure multiple workers stay synchronised where necessary.

Point-to-point operations include:

- `send`: Send a message to another worker
- `recv`: Receive a message from another worker
- `sendrecv`: Simultaneously send a message and receive a message

Collective operations include:

- `bcast`: Broadcast a message to all workers within the communicator
- `scatter`: One worker simultaneously sends a different message to every worker in the communicator

- `gather`: One worker receives a different message from every worker in the communicator, collecting up the received data.
- `allgather`: Every worker sends a different message. Every worker receives the message sent by every other worker.
- `reduce`: Every worker in the communicator presents a piece of data. All the data is combined in an operation (for example finding the maximum, finding the minimum, or calculating the sum) and one worker ends up with the result.

Synchronisation operations include:

- `barrier`: Each worker reaching the barrier halts until all workers in the communicator have reached the barrier. Then, all workers proceed simultaneously.

2.4.4 Hybrid-memory parallelisation

A modern supercomputer is rarely a purely shared-memory or distributed-memory parallel architecture. In reality, the vast majority are clusters of servers, with each server containing one or more multicore processors. Each processor socket is usually a pure UMA shared memory environment, with every core of that socket having equal access to that socket's memory. Within a multi-socket server, the server is overall a NUMA shared memory environment, with an overall shared address space and where cores in one socket may access memory attached to another socket, but with increased latency and reduced bandwidth between sockets. Finally, the cluster overall forms a distributed memory environment, where processes have no implicit access to memory on another server and must communicate via an explicit message passing interface. This overall structure is known as a hybrid architecture, where the communication properties are different depending what scale is looked at.

A hybrid-architecture parallel computer may be programmed as though it were a purely distributed-memory computer: an MPI worker is run for every processor core on every server and message passing is used for all communications, ignoring the added communication capability available between some workers. This is the simplest scheme, alternatively, hybrid parallelisation may be used: By utilising shared memory communication where available, it may be possible to improve performance, at the expense of added complexity and programmer effort.

One way to implement hybrid parallelisation is to run fewer processes than cores are available and use multithreading to exploit the extra cores. For example, one could run as many MPI workers as there are NUMA domains (i.e., processor sockets) and have each process spawn as many threads as there are cores per socket, using shared memory to communicate between the threads. As an alternative, newer versions of MPI support shared memory communication, either explicitly by letting two workers in a shared memory domain share a block of memory, or implicitly by using shared memory as the transmission medium for normal MPI messages.

What benefits may be gained from hybrid techniques depends on the nature of the algorithm being implemented. If two workers need to read a large dataset stored on one worker, sharing the memory is likely to be a great deal faster than explicitly transmitting the memory contents using messages. However, if workers only exchange occasional short messages, exploiting shared memory is less likely to give significant benefit, especially if the communication overhead is already minimal compared to actual computation time.

2.4.5 Automatic Parallelisation

Automatic parallelisation, where a program is automatically analysed to identify and implement potential parallelisation optimisations with the goal of reducing runtime, is an open research topic. It is often applied as a source code transformation: given the source code of a sequential program, the parallelisation tool outputs the source code for a new program which exploits a parallel compute resource. First, the parts of the program which consume significant time must be identified so that parallelisation effort can be targeted appropriately. Then, dependency analysis is used to determine ordering and which code can be run in parallel.

Automatic parallelisation can be simple and effective when targeting shared-memory parallel computers, and many such tools exist to use OpenMP for multi-threading. Targeting a distributed memory parallel computer is a much more difficult problem; In addition to dependency analysis, decisions must be made about how problem data is distributed across NUMA domains such as cluster nodes. Poor data distribution can cause large communication overheads to significantly reduce speed-up. A few tools exist for automated or assisted parallelisation over distributed-memory computers. PETSc [75] provides a high-performance framework including parallelisation over hybrid systems including GPUs for solution of systems modelled by partial differential equations, for example finite element analysis. CAPTools [76], and the related ParaWise, allow guided and automatic parallelisation based on OpenMP and MPI aimed at problems based on structured mesh computational mechanics. Neither of these tools would be suitable for parallelising the greedy algorithms used for CS reconstruction, which has a very different structure.

OMP2MPI [77] is not specific to a certain class of problems; it simply takes OpenMP parallelised loops and distributes the iterations to MPI workers using a master-slave architecture. High performance scientific code would usually use a maths library (such as BLAS or higher-level libraries) for vector and matrix operations rather than writing explicit loops over the elements and so loop vectorisation approaches to parallelisation would be ineffective. Explicit loops could be used instead of maths library operations in order to take advantage of loop vectorisation parallelisation but this would come at the cost of losing optimisation built-in to the maths library: Maths libraries such as BLAS reduce redundancy in operations, use SIMD processor instructions, and optimise memory access to use cache efficiently. Manual parallelisation allows the best of both worlds: after a data decomposition, optimised maths library optimisations can

be used on the portions of data. Furthermore, manual parallelisation reveals the structure present in a problem and provides information about whether a modified or different algorithm might be amenable to parallelisation where it might be more difficult to extract this information from the results of an automatic parallelisation tool.

2.5 Accelerated CS reconstruction

In this section, I will discuss the issues involved in accelerating, parallelising, and distributing CS reconstruction and evaluate existing works related to these topics. I will pay particular attention to the size of problems solved, whether the algorithms execute in a shared or distributed memory environment, and any decomposition of the problem used for parallelism. Most of these works focus on reducing reconstruction latency for problems which are practical, if slow, to reconstruct using a single CPU. A few pieces of work address reconstruction of problems which would be too large to be tractable without parallelisation or reconstruction. Finally, some work addresses distributed reconstruction, where distributed sensors collect and reconstruct signals with limited communication.

2.5.1 GPU-accelerated reconstruction

A number of works use GPUs for accelerating CS reconstruction. This is attractive because GPUs are relatively inexpensive and commonplace, and the shader architecture is well-suited to carrying out arithmetic using large vectors and matrices. Development of mathematical algorithms for GPUs is also significantly aided by libraries based on CUDA, for example cuBLAS [25], which provides GPU-accelerated implementations of common BLAS operations, and CUFFT, which provides GPU-accelerated FFT functions. GPUs also implement floating-point maths operations, simplifying implementation and potentially improving reconstruction quality. One downside of using GPU accelerators is the limited memory capacity available in comparison to a system's main memory, so very large problems may be limited by the bandwidth available for copying data between main memory and GPU memory.

A well-known GPU-accelerated CS reconstruction toolkit is GAGA (GPU Accelerated Greedy Algorithms), an implementation of several greedy algorithms based on hard thresholding presented by Blanchard and Tanner [26]. They report speed-up of up to 70x over a CPU MATLAB implementation (using automatic multithreading). Written in CUDA-C, GAGA is used to solve problems of up to $N = 1,048,576$, but only with a dictionary composed of a subset of a DCT matrix where it is not required to store the dictionary explicitly and fast transforms can be used to greatly reduce the computation required to perform matrix-vector products using the dictionary. When using a generic dictionary composed from IID random Gaussian values, they only solved up to $N = 16,384$. Several other papers report similar results using structured sub-Fourier dictionaries: Smith et al. [27] use a GPU maths library for MATLAB to

implement a split-Bregman solver, showing speed-up of 27x when reconstructing MRI images up to $N = 67,108,864$ (8192×8192 pixels) with a partial-Fourier dictionary; Borghi et al. [28] use CUDA and CUFFT to solve problems up to $N = 1,048,576$ with partial-DCT dictionaries; Belle, Armstrong, and Gain [19] use cuBLAS and CUFFT to achieve 23x speed-up reconstructing radio interferometry images up to $N = 64,000,000$ (8000×8000 pixels); and Endo et al. [17] use CUFFT to achieve 20x speed-up reconstructing holographic images up to $N = 10,485,760$ (1024×1024 pixels with 10 depth levels). No explicit decomposition of the problem is described in any of these works: The whole problem is transferred to GPU memory, vector operations or custom kernels are applied relying on the shared-memory architecture within a GPU, then the results are transferred back from the GPU. All these works rely on the use of fast transforms for structured dictionaries: none can be used on the unstructured dictionaries produced by dictionary learning, for example.

In contrast, some works consider the use of GPUs to solve problems with random or unstructured dictionaries. Here, the dictionary must be stored explicitly and fast transform operators cannot be used to speed multiplication so the problems solved are much smaller than those using sub-Fourier dictionaries. With 10% sub-sampling and double-precision floating point storage, a problem with $N = 10^5$ has a 7.5GB dictionary, similar to the memory capacity of the latest high-end GPUs. Using a dictionary too large to fit in GPU memory adds significant implementation complexity and the potential for CPU-GPU transfer bandwidth to be a bottleneck. Andrecut [29] uses cuBLAS to implement MP and achieve a 31x speed-up over CBLAS with signals up to $N = 15,000$, again with no explicit decomposition of the problem: CBLAS maths operations are simply replaced with their cuBLAS equivalents. Reliance on shared-memory computing limits the size of problem which can be solved and the lack of explicit decomposition means no real insight is gained into how the problem is partitioned for parallelisation.

Two papers describe an explicit decomposition of problems with generic random dictionaries. Fang et al. [30] use cuBLAS and CUDA to implement OMP with a 40x speed-up over a CPU implementation for signals of $N = 10^4$. The only decomposition used is a simple partitioning of matrix-vector multiplications; the overall algorithm still assumes a shared-memory environment. Kulkarni et al. [31] describe implementations of OMP for both GPU and a domain-specific many-core platform, claiming reconstruction of images up to 1024×1024 pixels, speed-up of up to 16x, and power reduction of up to 15x compared to a CPU implementation. However, they take the unusual approach of sampling and reconstructing each column of the image totally independently in addition to breaking larger images down into blocks. While this drastically reduces the size of problems to be solved and allows image columns to be processed in parallel, it also greatly reduces the compression performance and harms reconstructed image quality. For comparison with other works, this implementation is in reality equivalent to only solving relatively small problems with $N = 512$.

2.5.2 FPGAs and ASICs

Another topic of interest is the use of FPGAs (Field Programmable Gate Arrays) and ASICs (Application Specific Integrated Circuits) for solving CS problems. The goal is generally solution of small problems with very low latency or solving problems in a power-efficient manner compared to a CPU or GPU. The first ASIC or FPGA implementation of CS reconstruction appears to be that of Septimus and Steinberg [32], who implement a variation on OMP avoiding full calculation of the sparse estimate at every iteration. MGS updates to QR decomposition are used at each iteration to efficiently solve the Linear Least Squares (LLSQ) problem and the design is demonstrated through synthesis and simulation for an FPGA for a fixed problem size of $N = 128$. Yu et al. [78] use a similar approach but with some optimisations to reduce the number of square-root calculations, simulating the solution of fixed $N = 128$ problems on an FPGA platform. Stanislaus and Mohsenin [79] target fixed $N = 256$ problems, though the algorithm they describe as OMP appears to be MP so it is unclear which algorithm was actually implemented. For both of these algorithms an implicit column-wise data decomposition is used for the matrix-vector multiplication at the correlation step, but the algorithms are otherwise designed for a shared-memory environment. Quan et al. [80] implement a modified OMP algorithm named IOMP, specifically taking advantage of sub-Fourier dictionaries and using MGS updates to a QR decomposition. The efficiency of the FFT algorithm allows them to solve a larger problem of $N = 2048$.

Rather than updating a QR decomposition, Rabah et al. [33] solve problems of $N = 1024$ using a modified Cholesky decomposition in each iteration to form the Moore-Penrose pseudo-inverse and solve the LLSQ problem. A parallel inner product unit operates on 64-element blocks of data and is used in several parts of the algorithm. Calculating the full Cholesky decomposition, even with an efficient parallel implementation, seems very wasteful compared to the per-iteration MGS updates used in many other implementations of OMP and it is unclear why this decision was taken. In contrast, Polat and Kayhan [34] use an Alternate Cholesky Decomposition for LLSQ solution but only calculate an update to the decomposition in each iteration. The solver is described as "adaptable to different compression ratios" however it is only reconfigurable at the time of synthesis, after which the design is constrained to a single fixed problem size ($N = 1024$ is described in the paper). Kulkarni and Mohsenin [35] present hardware architectures for OMP and two reduced-complexity variants, reconfigurable at synthesis-time to signals from $N = 128$ to $N = 1024$. A full LU decomposition is used for the LLSQ solution in each iteration. Huang and Wang [81] introduce another OMP-variation where the Moore-Penrose pseudo-inverse is updated in each iteration. Their technique allows some task-parallelism in each iteration which is otherwise not possible due to the iterative nature of OMP. The solver is synthesized separately for both $N = 256$ and $N = 512$, but only Bernoulli dictionaries (taking values 0 or 1) are supported, which greatly reduces the complexity of the algorithm and the space required to store the dictionary at the expense of flexibility and relevance to all applications. In contrast

to the hardware reconstruction implementations mentioned so far, Chen and Zhang [82] do not implement a greedy pursuit but rather use an algorithm based on a simplified split-Bregman approach to solve the ℓ_1 problem. Their implementation can solve a single fixed problem size of $N = 16,384$ (128×128) however, as with most solvers supporting large problems, they require the use of sub-Fourier dictionaries to avoid storing the dictionary explicitly and allow the use of fast transform operations.

All of these works use fixed point arithmetic, which could potentially limit compression performance and solution accuracy. They also all target very small problems and can generally only tackle one fixed problem size determined at the time of synthesis. Various vector operations are carried out in parallel, but all still rely on the entire problem being held and processed on a single device. This is a useful strategy for low-latency reconstruction of small problems but does not scale to larger problem sizes or a distributed memory environment.

2.5.3 Distributed reconstruction for DCS

In the topic of DCS there is interest in distributed reconstruction algorithms. The standard DCS strategy is to return all samples to a central fusion centre where reconstruction can utilise full knowledge of the samples from all sensors. However, in some applications returning all samples to a fusion centre might require prohibitive amounts of communication bandwidth. Also, with a large number of sensors, reconstructing using all samples from all sensors may be a prohibitively large problem. With distributed reconstruction, each sensor primarily reconstructs the signal from the samples it recorded itself, but also conducts a limited amount of communication with some other sensors (possibly only its neighbours or nearby sensors) so as to exploit the signal correlation to improve reconstruction quality. Generally, distributed reconstruction will result in better compression performance than each sensor reconstructing from its own samples in isolation but worse performance than reconstruction at a fusion centre with full knowledge of all samples. The level of communication between sensors can be tuned to trade-off between reconstruction performance and level of communication required.

Several works demonstrate distributed reconstruction for DCS focusing on the application of distributed spectrum estimation for cognitive radios, where a number of distributed radio transceivers share bandwidth and intelligently estimate which frequencies are in use by each other (and other spectrum users) instead of following a predefined allocation or multiple access strategy. Sundman et al. [83] introduce a distributed variation of Subspace Pursuit named Distributed Predictive Subspace Pursuit (DPrSP), where in each iteration each node receives coefficient estimates from other nodes and then refines its own estimate of the coefficients. This strategy is limited to the DCS JSM-2 model which is only applicable in a limited number of situations. Sundman, Chatterjee, and Skoglund [84][85] extend this approach to other greedy pursuits and introduce a more general mixed support-set JSM. Their distributed greedy algorithms follow a similar strategy to DPrSP: In each iteration the sparse estimate is locally

refined then coefficients are exchanged with other nodes. Experiments are carried out with relatively small signals ($N = 500$). Wimalajeewa and Varshney [86] present a similar approach applied to OMP for problems of size $N = 256$. Ravazzi, Fosson, and Magli [87] describe distributed optimisation of non-linear problems, which may give applicability to a greater set of real-world problems but is also relevant for the linear DCS problems we are concerned with. The authors introduce various distributed algorithms based on Iterative Hard Thresholding which follow the same pattern of local refinement alternating with exchange of sparse estimates. As with the other investigations into distributed optimisation for DCS, only relatively small problems are solved (up to $N = 2560$). All of these algorithms focus on maximising compression and reconstruction quality while minimising communication between sensors rather than exploiting a distributed compute resource to reduce reconstruction time.

2.5.4 Distributing convex algorithms using ADMM

Most of the accelerated CS research discussed so far uses greedy pursuits. However, similar techniques are applicable to convex algorithms. One strategy to achieve this is the ADMM. ADMM is a general technique for distributed optimisation, applied to convex problems such as basis pursuit and LASSO by Boyd et al. [36]. It describes a mathematical strategy for decomposing optimisation problems for parallel solution, rather than a particular implementation of a distributed solver. Because solvers based on ADMM implement convex algorithms they may give better compression performance than techniques based on greedy algorithms. Boyd et al. also describe implementation of ADMM in a distributed computing environment using MPI, where each node stores a portion of the sparse estimate and dictionary and solves a small convex problem locally. Communications consist of global averaging of variables and synchronisation. In addition to a number of smaller problems solved on a single workstation, the authors describe the use of a distributed memory compute cluster to solve an unstructured problem with $N = 400,000$ and $n = 8,000$ utilising 80 workers spread equally over 10 machines. Utilising double precision floating point values the full dataset took over 30GB to store, apparently one of the largest LASSO problems ever solved at the time of publication in 2010. MPI was used for communication between nodes and the full solution took 6 minutes. Deng et al. [37] follow a similar approach, using two variations on ADMM called Jacobi ADMM and Jacobi-Proximal ADMM to develop a CS solver targeting distributed memory clusters using MPI communication. Problems of up to $N = 300,000$ and $n = 150,000$ are solved (requiring a total of 337GB of RAM using double-precision elements) using 80 workers spread equally over 10 machines. 43.5 minutes elapsed to solve a problem with $S/N = 0.05$ to an error of 10^{-3} and 48.5 minutes with a sparsity of $S/N = 0.15$ with the same error. This appears to be the largest solution of a CS problem reported in the literature.

Mota et al. [88] use an algorithm called D-ADMM, a decentralised BP solver, to enable distributed reconstruction in a DCS scenario. Two decompositions are described, splitting the

dictionary either based on rows or columns depending on the nature of the DCS problem. Experiments are carried out with dense unstructured dictionaries up to $N = 2560$ using row-based partitioning over up to 64 nodes and column-based partitioning over up to 10 nodes. The number of communication steps for solution of these relatively small problems is found to be lower than for similar algorithms but the elapsed time for solution is not presented. Bernabe et al. [89] use ADMM to adopt a domain-specific algorithm called HYCA to run on a GPU, where the use of cuBLAS and CUFFT allow speed-ups of up to 19 times over Central Processing Unit (CPU) for a structured problem. Fiandrotti et al. [20] also use ADMM to transform LASSO to utilise GPU compute, solving CS problems for astronomical image de-blurring. Signal sizes up to $N = 32,768$ are reconstructed, but the requirement of the use of circulant sampling matrices allows a significant reduction in memory and compute required over the use of generic unstructured matrices. Finally, Yang et al. [90] fuse an ADMM-based CS solver with deep learning resulting in ADMM-CSNet. Instead of being manually specified, the sparse transform is automatically learned from training data. Both speed-up and improved reconstruction performance over traditional CS methods are reported, using the inherently distributed nature of deep learning to exploit GPU acceleration. Although the algorithm is described as using a general sparse basis, it actually relies on a structured measurement matrix such as a partial-Fourier matrix or Toeplitz-structured matrix. All of the techniques based on ADMM effectively involve development of new algorithms, the properties of which will differ from standard, well understood, reconstruction algorithms. Furthermore, the exact reconstruction results will depend on the number of workers used.

2.6 Chapter summary

In this chapter, I have introduced and reviewed the literature relating to CS, HPC, parallelisation, and accelerated CS reconstruction. I have shown how CS fits in to the wider topics of information theory and sparsity, the variety of techniques which can be used to solve CS problems, and a number of applications of CS. Next, I showed how parallelism is an essential component of modern HPC and therefore why parallelisation is required to solve larger CS problems in a reasonable amount of time. I have reviewed common techniques applied to parallelise computing problems, looking in particular at distributed memory techniques and MPI which is used by the largest compute clusters.

In the final section of this chapter, I reviewed other work on the topic of accelerating, parallelising, and distributing CS reconstruction. I showed how many approaches to accelerated CS reconstruction rely on the use of structured dictionaries with fast transforms, and how work using unstructured dictionaries tends to focus on solving very small problems with low latency. Finally, I looked at novel algorithms developed using ADMM which have demonstrated ability to solve large unstructured problems in a distributed memory environment but deviate from the standard well-understood reconstruction algorithms and may produce results dependent on the

number of workers involved. No existing work allows reconstruction of very large CS problems using unstructured dictionaries in a distributed memory environment (such as an HPC cluster) while giving identical results to a standard algorithm such as OMP. This forms the motivation for my work in the following chapters.

Chapter 3

Parallelised greedy pursuits

In this chapter I describe my contributions to the field, specifically the development of the DistMP and DistOMP algorithms. I begin with a detailed description of MP, including both its strategy and various implementation details. This forms the foundation for the remainder of my work. I then introduce the DistMP algorithm, beginning with its motivation and overall structure. I then give a detailed description of DistMP including an algorithm listing. Next, I describe OMP. After introducing and motivating the algorithm, I give a high-level description of the steps involved. I cover in more detail how MGS decomposition updates may be used in an efficient implementation and then present a detailed algorithm for OMP using MGS updates. In the final section, I introduce DistOMP, a parallel successor to OMP. As with MP, I give a motivation and overview of the algorithm structure, including the different types of parallel worker used in the algorithm. I then give a detailed description of the tasks carried out by the two types of worker and present the algorithm listings for both workers.

3.1 Matching Pursuit

Matching Pursuit (MP) was introduced in 1993 by Mallat and Zhang [3]. It is the simplest greedy algorithm for CS reconstruction and also forms the basis for the other Greedy Algorithms used for this purpose. MP was introduced in order to allow the analysis of signals using redundant dictionaries (a matrix with more columns than rows), as opposed to linear isometries (a square matrix). Mallet and Zhang show how decomposing audio signals into a dictionary composed of Gabor time-frequency atoms can give a more useful result than decomposing into a wavelet dictionary. MP's ability to decompose a signal using a redundant dictionary is also essential for CS reconstruction, where the level of redundancy in the dictionary is a product of both redundancy in the sparse basis, desirable to increase the level of sparsity observed, and the level of sub-sampling carried out.

MP works by considering a *residual*, which is the portion of the starting signal not yet decomposed. The residual is initially the whole signal and is reduced at each iteration until it

has a magnitude of zero (or at least less than a small value). MP is the quintessential greedy pursuit: At each iteration it takes the simplest, most obvious, step in the direction of the goal. This is done by finding the atom of the dictionary which most correlates with the residual and removing its contribution to the residual. The sparse estimate is also updated, knowing how much of this atom is needed to reconstruct the signal. In the case of MP, our overall goal is find the most sparse solution $\mathbf{x}$ to the equation $\mathbf{b} = \mathbf{A}\mathbf{x}$. Internally, MP replaces this with the task of reducing the magnitude of the residual to zero (or near zero). For a well formed problem this leads to the same solution, however for some problems (for example where the number of samples taken is insufficient given the sparsity level) the residual may be eliminated while the sparse estimate is not actually a solution to the problem.

The algorithm for MP is shown in Algorithm 3.1. Cross-references to algorithm line numbers are made throughout this description to aid identification. Before execution, the dictionary and samples to approximate are loaded from disk (Lines 1–2). At initialisation the residual's starting value is the signal we are trying to decompose (Line 4) and the sparse estimate is set to zero (Line 3). The algorithm is terminated when the magnitude of the residual falls below a pre-defined threshold: Before each iteration begins the ℓ_2-norm of the residual is evaluated ($\|\mathbf{r}\|_2$) and compared to the defined threshold ϵ (Line 5). The threshold is chosen based on the numerical precision of the implementation and the required precision in its output (but is not dependant on the magnitude of error present in the samples taken). The result of the algorithm is the sparse estimate $\mathbf{x}$ which satisfies the equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ (Line 10). A limit on the number of iterations may be applied but is only necessary in case of numerical instability in the implementation: In theory, so long as the signal to be decomposed falls in the space bounded by the dictionary, the algorithm should always converge on a solution (however, not necessarily the correct solution).

The first step in each iteration is finding the correlation between each dictionary atom (column) and the residual. This is done by taking the inner product $\mathbf{c} \leftarrow \mathbf{A}^T\mathbf{r}$, after which $\mathbf{c}$ is a vector of correlations (Line 6). If the dictionary atoms are not already normalised to unit magnitude then each correlation must be divided by the magnitude of the corresponding atom so that atoms with a comparatively large magnitude do not appear to have a higher correlation than those with a smaller magnitude. For simplicity, for the remainder of my description of this algorithm I will assume the dictionary columns are normalised to unit magnitude. Only minor modifications are required to the algorithm if this is not the case. Following correlation, the selection step finds which of the correlations has the greatest absolute magnitude (Line 7). This selected atom which will be used for the rest of this iteration.

Having selected which dictionary atom to use for this iteration, the next step is to update our sparse estimate (Line 8). We can think of the sparse estimate as a recipe, describing how much of each dictionary atom to combine to produce the signal we are decomposing. The correlation of the chosen column, c_k, tells us how much of this column to add to the estimate. The update is incorporated into the sparse estimate as $x_k \leftarrow x_k + c_k$.

The final step in the iteration is to update the residual (Line 9). We have selected the

dictionary atom to be used and its correlation with the residual tells us how much of it is being incorporated in the sparse estimate. We can update the residual in two ways. One option is to subtract an appropriate amount of this atom from the residual: $\mathbf{r} \leftarrow \mathbf{r} - c_k \mathbf{A_k}$. Another option is to use our sparse estimate to reproduce our signal estimate, and subtract this from the actual signal to calculate the residual: $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}$. The former is generally faster as it only requires scaling and subtracting a known vector, whereas the latter requires a matrix-vector product by the (potentially very large) dictionary.

MP benefits from being both very simple to understand and implement, and efficient implementations are very fast to execute each iteration. The complexity is dominated by the correlation step at each iteration. MP tends to require more iterations than other algorithms to solve the same problem. In the case of a dictionary with no redundancy (all atoms are orthogonal and the dictionary matrix is square) MP takes only as many iterations as the signal has sparse components and so executes very quickly. However, we generally use MP with highly redundant dictionaries, where it can take many more iterations to converge. A particularly bad case occurs when two atoms are highly correlated: With some signals MP has a tendency to oscillate between choosing the two atoms on alternate iterations while making little progress towards approximating the signal.

MP has a particularly efficient implementation when the dictionary is represented by fast transform operations. The complexity of MP is dominated by the matrix-vector product used to correlate the dictionary and the residual. This matrix-vector product, $\mathbf{A}^T\mathbf{r}$ is one of the fast operators generally available. For example, with an explicit dictionary the correlation step requires on the order of Nn operations, where the FFT operator has a complexity on the order of $n\log(n)$, a significant reduction for large problems.

Algorithm 3.1 Matching Pursuit

Input: $\mathbf{b} \in \mathbb{R}^n$ ▷ Samples (loaded from disk)
Input: $\mathbf{A} \in \mathbb{R}^{n \times N}$ ▷ Dictionary (loaded from disk)
Initialise: $\mathbf{x} \leftarrow \mathbf{0}^N$ ▷ Initialise sparse estimate
Initialise: $\mathbf{r} \leftarrow \mathbf{b}$ ▷ Initialise residual
while $\|\mathbf{r}\|_2 \leq \epsilon$ **do** ▷ Stopping criterion
    $\mathbf{c} \leftarrow \mathbf{A}^T\mathbf{r}$ ▷ Correlation
    $k \leftarrow \text{argmax}_k|c_k|$ ▷ Selection
    $x_k \leftarrow x_k + c_k/\|\mathbf{A_k}\|_2$ ▷ Sparse estimate update
    $\mathbf{r} \leftarrow \mathbf{r} - c_k\mathbf{A_k}$ ▷ Residual update
Output: $\mathbf{x}$ ▷ Sparse estimate saved to disk

3.2 Distributed Matching Pursuit

Distributed Matching Pursuit (DistMP) was developed with two objectives in mind: To solve very large (larger than will fit in main memory) CS reconstruction problems without having to use disk storage as working memory, and to solve the problems faster than could be done using a single computer. For the first objective we need a cluster whose summed main memory is sufficient to hold the whole problem in memory. We then break the problem into small enough pieces such that each piece can fit in the main memory of a single node. To achieve the second objective we need to exploit the processing power of many computers in parallel, at the same time as minimising overheads. Both of these objectives can be achieved by using data parallelism; by considering the body of data worked on by the algorithm and finding an appropriate way to break it up. The challenge is finding an efficient way to break up the problem and develop the algorithm so as to minimise communication overheads.

The largest single piece of data held in memory is the dictionary, so this is the clear target for breaking into pieces distributed over the compute nodes. The other data we hold in memory (the current sparse estimate and residual) are significantly smaller than the dictionary, and so need not be broken up: They may even be duplicated over all the compute nodes if required. An important decision is whether we break up the dictionary in terms of rows or columns. Computation time is dominated by the correlation step, which involves carrying out the matrix-vector product $\mathbf{A}^T\mathbf{r}$. Splitting column-wise gives a more efficient implementation of the correlation step and so is the obvious choice.

We then have to decide whether to distribute the columns of the dictionary in a round-robin fashion or in contiguous blocks. As there is no significance to the ordering of columns in the dictionary, there is no difference in computation time whether distributing the columns to workers in contiguous blocks, in a round-robin fashion, or randomly. Therefore we distribute the columns in order, in contiguous blocks, as this leads to the simplest implementation. This distribution of the dictionary between compute workers, illustrated in Figure 3.1, forms the core of the DistMP algorithm, as well as the DistOMP algorithm introduced later.

Let W be the total number of workers we are tasking to solve the problem, and w be our

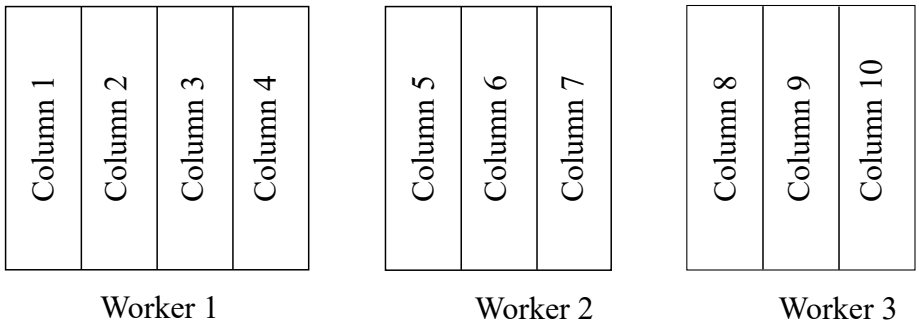


Figure 3.1: An example distribution of 10 dictionary columns to 3 compute workers in DistMP

worker index such that $1 \leq w \leq W$ [1]. The columns held on the wth worker are selected by the set $\mathcal{W}_w$, so that the worker's local dictionary matrix $\mathbf{L} \leftarrow \mathbf{A}_{\mathcal{W}_w}$. The sets $\mathcal{W}_w$ contain the contiguous integers

$$\mathcal{W}_w \leftarrow \begin{cases} 1 + (w-1)\left\lceil \frac{N}{W} \right\rceil \dots w \left\lceil \frac{N}{W} \right\rceil & w \leq N \bmod W \\ 1 + (w-1)\left\lfloor \frac{N}{W} \right\rfloor + N \bmod W \dots w \left\lfloor \frac{N}{W} \right\rfloor + N \bmod W & w > N \bmod W \end{cases}.$$

The sets $\mathcal{W}_w$ are all disjoint, so that the same column is never present on more than one worker

$$\mathcal{W}_i \cap \mathcal{W}_j = \emptyset \, \forall i \neq j \,,$$

and together they cover the whole matrix $\mathbf{A}$

$$\mathcal{W}_1 \cup \mathcal{W}_2 \cup \dots \cup \mathcal{W}_W = \{1 \dots N\} \,.$$

Each worker will access dictionary columns by their index in $\mathbf{L}$, but will need to be able to determine that column's index in $\mathbf{A}$. Referring to the column index in $\mathbf{L}$ as k and the column index in $\mathbf{A}$ as K, we can use the definition of $\mathcal{W}$ to derive the relationship given in Equation 3.1 for a particular worker w, total number of workers W, and total number of columns N.

$$K = \begin{cases} (w-1)\left\lceil \frac{N}{W} \right\rceil + k & w \leq N \bmod W \\ (w-1)\left\lfloor \frac{N}{W} \right\rfloor + N \bmod W + k & w > N \bmod W \end{cases} \tag{3.1}$$

DistMP is designed so that all workers execute the same program and work in lock-step. At start-up every worker reads its configuration along with the problem data (the signal to be decomposed, and that worker's portion of the dictionary) from disk. Because every worker maintains an up-to-date copy of the residual, at the beginning of each iteration every worker can independently evaluate the stopping criterion by comparing the magnitude of the residual to the stopping threshold. This means no extra communication or coordination is required to terminate workers when the stopping criterion is met. Every worker also maintains an up-to-date copy of the current sparse approximation. This means upon meeting the stopping criterion, only the root node needs to save the output of the algorithm and all other workers can terminate with no need to store their working data.

The algorithm for DistMP is shown in Algorithm 3.2. The inputs to the algorithm are $\mathbf{b}$, the signal to be approximated (Line 1), and $\mathbf{L}$, this worker's portion of the dictionary $\mathbf{A}$ (Line 2). The MPI environment provides us with W, the number of workers in the cluster (Line 3), and w, this worker's index (starting from 1, Line 4). Figure 3.2 shows the communications structure

[1] MPI actually numbers workers starting from 0, but for consistency I follow the convention of 1-based numbering for all algorithms in this thesis.

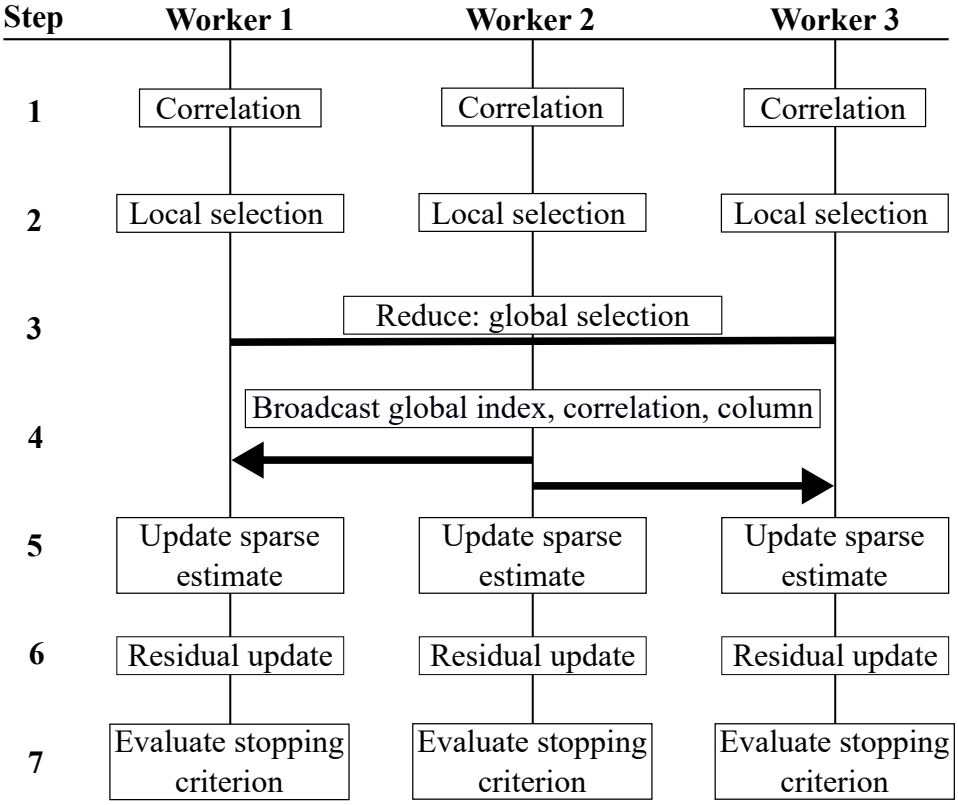


Figure 3.2: The communications between workers in a DistMP setup with 3 workers. In this iteration the second worker holds the column with the highest magnitude correlation to the residual.

of the algorithm.

The first step in each iteration is correlation (Line 8), where each worker carries out a matrix-vector product between its portion of the dictionary and the current residual in order to calculate how well each dictionary atom correlates with the residual. Since each worker holds effectively the same sized portion of the dictionary (a difference of one column would be negligible for large-scale problems) and has an up-to-date copy of the residual, this distributed inner product parallelises perfectly with no need for communication or overhead. This step is implemented identically to the correlation step in conventional MP, except the matrix represents only a portion of the dictionary instead of the entire dictionary.

The next task is to find which atom, across all the workers, has the highest absolute correlation with the residual. First, each worker evaluates amongst its own atoms which has the highest absolute correlation (Line 9). This is also implemented in the same way as conventional MP. Next, the workers convene to determine which worker holds the atom with the highest absolute correlation to the residual (Line 10). The worker which holds the column with the largest absolute correlation converts its local column index to a global column index using Equation 3.1 (Line 12), and stores the correlation and column itself in γ and $\mathbf{g}$ respectively (Lines 16–17). It then broadcasts K, γ and $\mathbf{g}$ to all the other workers (sent on Lines 18–20, received on Lines 22–24), which will need to know these to carry out the rest of the iteration.

Every worker individually keeps track of the current sparse estimate, and so performs the same update. The sparse estimate update (Line 25) is the same as with classic MP, but uses the index of the globally best dictionary atom (across all workers) and correlation. Residual update (Line 26) also proceeds in the same way as classic MP, multiplying the globally best correlation by the selected dictionary atom and subtracting this from the residual.

Each worker evaluates the stopping criterion (Line 7) independently by calculating the magnitude of the residual. Every worker has a record of the sparse estimate $\mathbf{x}$, but only one (for example the root node) need save it to disk (Line 27).

Algorithm 3.2 Distributed Matching Pursuit

1: **Input:** $\mathbf{b} \in \mathbb{R}^n$ ▷ Samples (loaded from disk)
2: **Input:** $\mathbf{L} = \mathbf{A}_{\mathcal{W}_w}$ ▷ Local dictionary (loaded from disk)
3: **Input:** $W \in \mathbb{Z}, W \geq 1$ ▷ Number of workers (from MPI)
4: **Input:** $w \in \mathbb{Z}, 1 \leq w \leq W$ ▷ This worker's index (from MPI)
5: **Initialise:** $\mathbf{r} \leftarrow \mathbf{b}$ ▷ Initialise residual
6: **Initialise:** $\mathbf{x} \leftarrow \mathbf{0}^N$ ▷ Initialise sparse estimate
7: **while** $\|\mathbf{r}\|_2 \geq \epsilon$ **do** ▷ Stopping criterion
8: $\quad\mathbf{c} \leftarrow \mathbf{L}^T\mathbf{r}$ ▷ Correlation
9: $\quad k \leftarrow \underset{k}{\operatorname{argmax}}|c_k|$ ▷ Local selection
10: $\quad$**Reduce** $v \leftarrow \underset{w}{\operatorname{argmax}}|c_k|$ ▷ Find which worker holds the highest correlation
11: $\quad$**if** $v = w$ **then** ▷ If we hold the selected column:
12: $\quad\quad$**if** $w \leq N \bmod W$ **then** ▷ Convert local index to global
13: $\quad\quad\quad K \leftarrow (w-1)\left\lceil \frac{N}{W} \right\rceil + k$
14: $\quad\quad$**else**
15: $\quad\quad\quad K \leftarrow (w-1)\left\lfloor \frac{N}{W} \right\rfloor + N \bmod W + k$
16: $\quad\quad\gamma \leftarrow c_k$ ▷ Store correlation
17: $\quad\quad\mathbf{g} \leftarrow \mathbf{L}_K$ ▷ Store the selected column
18: $\quad\quad$**Broadcast** K ▷ Broadcast global index
19: $\quad\quad$**Broadcast** γ ▷ Broadcast correlation
20: $\quad\quad$**Broadcast** $\mathbf{g}$ ▷ Broadcast selected column
21: $\quad$**else** ▷ Otherwise:
22: $\quad\quad$**Receive Broadcast** K ▷ Receive global index
23: $\quad\quad$**Receive Broadcast** γ ▷ Receive correlation
24: $\quad\quad$**Receive Broadcast** $\mathbf{g}$ ▷ Receive selected column
25: $\quad x_K \leftarrow x_K + \gamma/\|\mathbf{g}\|_2$ ▷ Update sparse estimate
26: $\quad\mathbf{r} \leftarrow \mathbf{r} - \gamma\mathbf{g}$ ▷ Residual update
27: **Output:** $\mathbf{x}$ ▷ Sparse estimate saved to disk

3.3 Orthogonal Matching Pursuit

MP's main weakness is non-orthogonal dictionaries. Upon encountering a residual composed of two highly correlated dictionary elements, Matching Pursuit will tend to oscillate between

choosing each dictionary element and converge slowly. Thus, despite each iteration being very fast, MP may be slower than more complicated algorithms. Orthogonal Matching Pursuit (OMP) was created to avoid precisely this problem. It ensures that each dictionary atom is only ever selected in one iteration, so the number of iterations is equal to the number of dictionary atoms required to approximate the signal. As well as requiring fewer iterations, OMP delivers better performance: it will tend to give a more sparse approximation of a signal than MP, falling closer to the results from Basis Pursuit and the optimum decomposition given by combinatorial ℓ_0 optimisation. OMP was described independently by Davis et al and Pati et al in 1994, based on the then-recently published MP algorithm.

In summary, OMP is a close variant on MP, which differs from MP by the introduction of an orthogonalisation step. Initialisation, correlation, and selection proceed as with MP. However, instead of updating the sparse approximation by simply adding the current correlation to the element corresponding to the current dictionary atom, we calculate a new sparse approximation from scratch. By doing this we can calculate an optimal sparse approximation, which best approximates the signal using the currently selected dictionary atoms. Having done this we also update the residual, which will now be orthogonal to the space spanned by the currently selected dictionary atoms.

Algorithm 3.3 shows a simplified rendition of OMP. Data is loaded from disk (Lines 1–2) in the same way as MP. Also as with MP, at start-up we initialise the residual to the samples (Line 4), and set the sparse estimate to zero (Line 7). OMP introduces two new state variables: the index set matrix (consisting of all the dictionary atoms selected so far as its columns, initialised at Line 6 as an empty matrix) and the working set matrix (a list of the indices of atoms selected so far, represented as a row vector, initialised to an empty row-vector at Line 5). OMP may be terminated in the same way as MP, by comparing the magnitude of the residual to a threshold based on our numerical precision – in this case we set the iteration limit $p \leftarrow n$ as this is the maximum number of iterations we may require. However, we may wish to limit the number of dictionary atoms used in composing the sparse approximation instead of trying to produce the most exact approximation. With OMP we can do this by decreasing the iteration limit: OMP will always use the same number of sparse atoms as it carries out iterations, and will always produce the best approximation possible, at every iteration, using the atoms selected so far. In this algorithm we use the former method (Line 8).

The differences from MP begin after correlation (Line 9) and selection (Line 10) operations. The first new step is to store the index of the selected column in the index set (Line 11) and store the column itself in the working set (Line 12). Rather than simply updating the sparse estimate as in MP, we now calculate the optimal sparse estimate (that which gives the smallest residual) possible using the dictionary atoms selected so far. Finding the optimal sparse estimate at each iteration can be written as a LLSQ problem (Line 13). Instead of giving our sparse estimate in the form of coefficients of $\mathbf{A}$, here we use coefficients of $\mathbf{\Phi}$ to simplify the algorithm. We call this new variable $\mathbf{a}$. Since $\mathbf{\Phi}$ is simply a subset of the columns in $\mathbf{A}$, $\mathbf{a}$ is just a re-ordering of

the elements of $\mathbf{x}$. Equivalently, $\mathbf{\Phi a} = \mathbf{Ax}$.

We now use $\mathbf{a}$ to calculate a residual update by multiplying it by $\mathbf{\Phi}$ to produce an estimate of the samples and subtracting this from the actual samples (Line 14). Since we want the output of the algorithm in terms of $\mathbf{x}$ rather than $\mathbf{a}$, we need to do a conversion step at the end of the algorithm (Line 17). This is where we use λ, since each element of λ tells us which column of $\mathbf{A}$ the corresponding element of $\mathbf{a}$ refers to.

Algorithm 3.3 Orthogonal Matching Pursuit: Concise form

1:	**Input:** $\mathbf{b} \in \mathbb{R}^n$	▷ Samples (loaded from disk)
2:	**Input:** $\mathbf{A} \in \mathbb{R}^{n \times N}$	▷ Dictionary (loaded from disk
3:	**Initialise:** $i \leftarrow 0$	▷ Initialise iteration counter
4:	**Initialise:** $\mathbf{r} \leftarrow \mathbf{b}$	▷ Initialise residual
5:	**Initialise:** $\mathbf{\Phi} \leftarrow \mathbf{O}^{n \times 0}$	▷ Initialise working set
6:	**Initialise:** $\lambda \leftarrow \mathbf{O}^{1 \times 0}$	▷ Initialise index set
7:	**Initialise:** $\mathbf{x} \leftarrow \mathbf{0}^N$	▷ Initialise sparse estimate
8:	**while** $\|\mathbf{r}\|_2 \leq \epsilon$ **do**	▷ Stopping criterion
9:	$\mathbf{c} \leftarrow \mathbf{A}^T\mathbf{r}$	▷ Correlation
10:	$k \leftarrow \text{argmax}_k c_k$	▷ Selection
11:	$\lambda \leftarrow [\lambda\, k]$	▷ Store index of selected atom
12:	$\mathbf{\Phi} \leftarrow [\mathbf{\Phi A}_k]$	▷ Add atom to working set
13:	$\mathbf{a} \leftarrow \underset{\mathbf{a}}{\text{argmin}} \|\mathbf{\Phi a} - \mathbf{b}\|_2$	▷ Orthogonalisation
14:	$\mathbf{r} \leftarrow \mathbf{b} - \mathbf{\Phi a}$	▷ Residual update
15:	$i \leftarrow i + 1$	▷ Increment iteration counter
16:	**for** $j \leftarrow 1$ **to** i **do**	
17:	$x_{\lambda_j} \leftarrow a_j$	▷ Calculate sparse estimate
18:	**Output:** $\mathbf{x}$	▷ Sparse estimate saved to disk

3.3.1 Implementing the LLSQ solver

The majority of compute time in OMP is taken by the LLSQ step,

$$\mathbf{a} = \underset{\mathbf{a}}{\text{argmin}} \|\mathbf{\Phi a} - \mathbf{b}\|_2 ,$$

where we calculate the optimal sparse decomposition using the dictionary atoms selected so far in order to minimise the remaining residual. One option to compute the solution to this problem is using Conjugate Gradients (CG) to iteratively approximate the Moore-Penrose pseudo-inverse of the working set, denoted $\mathbf{\Phi}^\dagger$. The solution is then easily calculated from $\mathbf{a} = \mathbf{\Phi}^\dagger\mathbf{b}$. Ordinarily, calculating the pseudo-inverse would be an inefficient way to solve this problem, however, we only need an approximate solution and so can terminate the CG process

after only a few iterations. This appears an efficient technique in isolation, however it ignores the fact that we are solving a very similar LLSQ problem in every iteration, differing only by the addition of a column to $\mathbf{\Phi}$ each time. The strategy I will use is that given by Tropp and Gilbert [10]: maintaining a QR decomposition between each iteration, and using an MGS update in each iteration to update the QR decomposition.

QR decomposition decomposes a matrix (for example $\mathbf{\Phi}$) into two matrices, $\mathbf{Q}$ and $\mathbf{R}$. The decomposition identifies the orthogonal components of the matrix and stores them in $\mathbf{Q}$, which has all orthonormal (orthogonal and normalised to unit magnitude) columns. $\mathbf{R}$ is an upper triangular matrix containing the coefficients such that $\mathbf{QR} = \mathbf{\Phi}$. MGS is a particular technique for calculating the QR decomposition of a matrix in a numerically stable manner [91]. The following algorithm is given by Björck [91] for applying MGS in a column-wise order:

The first column of $\mathbf{Q}$ is simply the first column of $\mathbf{\Phi}$ after normalisation, with its magnitude before normalisation stored in the first element of the first column of $\mathbf{R}$. The next column of $\mathbf{Q}$ is the second column of $\mathbf{\Phi}$ after subtracting any portion of it parallel with the first column of $\mathbf{\Phi}$. For each new column of $\mathbf{Q}$ we take the corresponding column of $\mathbf{\Phi}$ and iterate over the existing columns of $\mathbf{Q}$, subtracting any portion of the former parallel with the latter.

Each column $\mathbf{Q}_i$ is calculated from the corresponding column $\mathbf{\Phi}_i$ as

$$\mathbf{Q}_i = \frac{\mathbf{\Phi}_i - \sum_{j=1}^{i-1} (\mathbf{\Phi}_i \mathbf{Q_j}) \mathbf{Q}_j}{\left\| \mathbf{\Phi}_i - \sum_{j=1}^{i-1} (\mathbf{\Phi}_i \mathbf{Q_j}) \mathbf{Q}_j \right\|_2}.$$

At the same time, the corresponding column $\mathbf{R}_i$ is calculated as $\mathbf{Q}^T \mathbf{\Phi}_i$ with the addition of the element

$$\mathbf{R}_{i,i} = \left\| \mathbf{\Phi}_i - \sum_{j=1}^{i-1} (\mathbf{\Phi}_i \mathbf{Q_j}) \mathbf{Q}_j \right\|_2.$$

Since this algorithm considers each column of $\mathbf{\Phi}$ in order, appending a column each to $\mathbf{Q}$ and $\mathbf{R}$, it is very well suited to use for solving the LLSQ problem in our OMP algorithm. At each iteration, after adding the new column to $\mathbf{\Phi}$ we can update $\mathbf{Q}$ and $\mathbf{R}$ by carrying out a single iteration of MGS, avoiding any duplicate computation.

The algorithm for a single column MGS update is shown in Algorithm 3.4, adapted from the algorithm for MGS update after addition of a single column as given by Björck [91].

Algorithm 3.4 A single iteration of column-wise Modified Gram-Schmidt QR decomposition

1: **Input:** $\mathbf{g} \in \mathbb{R}^n$ ▷ Column added to $\mathbf{\Phi}$
2: **Input:** $\mathbf{Q} \in \mathbb{R}^{n \times i}$ ▷ Orthogonalisation to update
3: $\mathbf{r} \leftarrow \mathbf{Q}^T\mathbf{g}$ ▷ Find correlations of new column with orthogonal matrix
4: $\mathbf{h} \leftarrow \mathbf{g} - \mathbf{Q}\mathbf{r}$ ▷ Find portion of $\mathbf{g}$ orthogonal to $\mathbf{Q}$
5: $r_i \leftarrow \|\mathbf{h}\|_2$ ▷ Append normalisation factor to coefficients
6: $\mathbf{q} \leftarrow \|\mathbf{h}\|_2^{-1}\mathbf{h}$ ▷ Normalised orthogonal vector
7: **Output:** $\mathbf{q}$ ▷ Update vector appended to $\mathbf{Q}$
8: **Output:** $\mathbf{r}$ ▷ Update vector appended to $\mathbf{R}$

The column-wise MGS update algorithm takes as inputs $\mathbf{g} = \mathbf{A}_k$ (the new column added to $\mathbf{\Phi}$, Line 1), and $\mathbf{Q}$ (the existing orthogonalisation of $\mathbf{\Phi}$, Line 2). The algorithm begins by finding the correlation between the new column and each existing column of $\mathbf{Q}$ (Line 3). This is done by carrying out a matrix-vector product between the two, resulting in a vector of correlations, denoted $\mathbf{r}$. This vector will be appended to $\mathbf{R}$ as a new column. Next, we remove any portion of $\mathbf{g}$ which correlates with a column of $\mathbf{Q}$ (Line 4). The vector $\mathbf{r}$ already holds the correlation between $\mathbf{g}$ and each vector in $\mathbf{Q}$. Calculating the inner product $\mathbf{Q}\mathbf{r}$ multiplies each column of $\mathbf{Q}$ by its correlation with $\mathbf{g}$ and then sums the vectors, resulting in a vector which is the projection of $\mathbf{g}$ onto the basis $\mathbf{Q}$. By subtracting this from $\mathbf{g}$ we are left with the remainder of $\mathbf{g}$ which is perpendicular to the basis $\mathbf{Q}$ (and therefore orthogonal to every column already in $\mathbf{Q}$) which we call $\mathbf{h}$. The ℓ_2 magnitude of $\mathbf{h}$ is calculated and stored in the final element of $\mathbf{r}$ (Line 5). Finally, we normalise $\mathbf{h}$ to unit magnitude and call this new unit vector $\mathbf{q}$ (Line 6). The result of the MGS update algorithm is the vectors $\mathbf{q}$ (Line 7) and $\mathbf{r}$ (Line 8), which are appended as new columns to $\mathbf{Q}$ and $\mathbf{R}$ respectively. The new column we wish to append to $\mathbf{R}$ has one more row than $\mathbf{R}$, so we must first increase the height of $\mathbf{R}$ by one row (filling the new elements with zero) before adding the new column.

Having used MGS to compute the QR decomposition of $\mathbf{\Phi}$, we return to solving the LLSQ problem. Rewriting $\mathbf{\Phi}$ using the QR decomposition, the optimisation objective becomes

$$\|\mathbf{Q}\mathbf{R}\mathbf{a} - \mathbf{b}\|_2 .$$

Left-multiplying the objective by $\mathbf{Q}^T$ gives

$$\left\|\mathbf{Q}^T\mathbf{Q}\mathbf{R}\mathbf{a} - \mathbf{Q}^T\mathbf{b}\right\|_2 .$$

The term $\mathbf{Q}^T\mathbf{Q}$ has no effect on the position of the minimum of the objective function, so we can find the solution to the LLSQ problem by solving the equation

$$\mathbf{R}\mathbf{a} = \mathbf{Q}^T\mathbf{b}$$

for $\mathbf{a}$. We first calculate the right hand side of the equation by carrying out the matrix-

vector product $\mathbf{y} = \mathbf{Q}^T\mathbf{b}$. The properties of the QR decomposition tell us that $\mathbf{R}$ will be an upper-triangular matrix. This means we can easily solve $\mathbf{Ra} = \mathbf{y}$ for $\mathbf{a}$ using Back Substitution (BS).

I will illustrate the algorithm for BS by working through an example with $i = 3$. Writing out the vectors and matrices, the equation $\mathbf{Ra} = \mathbf{y}$ becomes

$$\begin{bmatrix} R_{1,1} & R_{1,2} & R_{1,3} \\ 0 & R_{2,2} & R_{2,3} \\ 0 & 0 & R_{3,3} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{3.2}$$

This describes a set of simultaneous equations, however the structure of $\mathbf{R}$ makes solution trivial. For this case the solution is given by:

$$a_3 = y_3 R_{3,3}^{-1},$$

$$a_2 = (y_2 - R_{2,3}a_3)R_{2,2}^{-1},$$

$$a_1 = (y_1 - R_{1,2}a_2 - R_{1,3}a_3)R_{1,1}^{-1}.$$

For larger problems we add more steps, each taking into account the result of previous steps. Algorithm 3.5 shows the BS algorithm we will use.

Algorithm 3.5 An implementation of Back Substitution for solving LLSQ problems

1: **Input:** $\mathbf{R} \in \mathbb{R}^{i \times i}$ ▷ Upper-triangular matrix
2: **Input:** $\mathbf{y} \in \mathbb{R}^i$ ▷ Input vector
3: **Initialise:** $\mathbf{a} \leftarrow \mathbf{0}^i$ ▷ Initialise result vector
4: **for** $i \leftarrow i$ **to** 1 **do**
5: $\quad a_i \leftarrow (y_i - \mathbf{R}_i\mathbf{a})\, R_{i,i}^{-1}$ ▷ Calculate next element of result vector
6: **Output:** $\mathbf{a}$

3.3.2 OMP with MGS updates and BS

Algorithm 3.6 shows the completed OMP algorithm, with the LLSQ step replaced by an MGS update to the QR decomposition, followed by using BS to calculate the new sparse estimate.

Comparing the algorithm to the simplified implementation presented in Algorithm 3.3, the same inputs are loaded from disk (Lines 1–2) and we still initialise the iteration counter, sparse estimate, residual vector, and index set (Lines 3–6). The first difference is that we no longer explicitly store a $\mathbf{\Phi}$ matrix: $\mathbf{\Phi}$ is implicitly stored in the form of $\mathbf{Q}$ and $\mathbf{R}$, which we hold as state, update in each iteration, and use to calculate the sparse estimate update. $\mathbf{Q}$ is initialised (Line 7) as an empty matrix with n rows and 0 columns: in each iteration we increase its size

by 1 column, filling the new column with $\mathbf{q}$. $\mathbf{R}$ is initialised (Line 8) to an empty matrix, with no rows or columns.

Each iteration of the algorithm is broken into sections for clarity. The first section, Correlation (Line 10), is the same as Lines 9–12 of Algorithm 3.3. Then, instead of updating the working set $\mathbf{\Phi}$ and solving the LLSQ problem, Section MGS Update (Line 15) contains the contents of Algorithm 3.4 and Section Back Substitution (Line 24) contains the contents of Algorithm 3.5.

The final section is Residual Update (Line 30). Since we no longer store $\mathbf{\Phi}$ explicitly, we move the conversion from $\mathbf{a}$ to $\mathbf{x}$ from the end of the algorithm into the iteration, now Lines 31–33. We also change the residual update step (Line 34) from using the product $\mathbf{\Phi a}$ to $\mathbf{Ax}$.

Several trade-off decisions were made when producing my implementation of OMP. The algorithm calls for the matrices $\mathbf{Q}$ and $\mathbf{R}$ and the vector λ to each grow at every iteration. $\mathbf{Q}$ grows by one column per iteration, $\mathbf{R}$ grows by a row and a column per iteration (padded with 0), and λ grows by one element each iteration. Actually changing the shape and size of these at runtime is quite inefficient, often requiring reallocation of memory and copying large quantities of data. To give a faster implementation, I use fixed size matrices for each of these, pre-allocated at the beginning of the algorithm. Each time the matrices are used in the algorithm, we use the subset of columns actually in use, creating a virtual matrix smaller than the actual matrix existing in memory.

To pre-allocate these matrices we need to decide what size to allocate, or, equivalently, an iteration limit for the algorithm. We know that OMP cannot take more iterations than n, the number of rows in the dictionary (because $\mathbf{\Phi}$ grows by one column in each iteration and the largest possible $\mathbf{\Phi}$ we could construct which has an unambiguous orthogonalisation is $n \times n$). In the absence of other information, we could set the iteration limit to n and know that this will never be exceeded. However, if we know the sparsity level (the number of non-zero elements) in the sparse signal, $\mathbf{x}$, we can estimate a lower bound on the number of iterations. OMP fills as many non-zero elements into $\mathbf{x}$ as it carries out iterations. Therefore, in the case of a perfect reconstruction, our number of iterations equals the sparsity level $S = \|\mathbf{x}\|_0$. If we can estimate the sparsity level of our captured signal, we can use this to estimate an iteration limit by including a safety factor, for example an iteration limit of $1.5S$. The number of rows, n, will always be greater than the sparsity level S, generally significantly so. Therefore, using the estimated sparsity level to set an iteration limit will usually give a much lower limit than simply using n. The trade-off in pre-allocating these matrices instead of letting them grow dynamically is that memory requirements are greater: For a given size of main memory we can not solve as big a problem as if we were sizing matrices dynamically. If the ability to grow the matrices is retained despite pre-allocating a size, then the safety factor can be made considerably smaller or removed entirely and the iteration limit exceeded if necessary. However, to simplify my implementation I maintain a hard iteration limit with no capability to grow these matrices.

Algorithm 3.6 Orthogonal Matching Pursuit: using MGS updates

Input: $\mathbf{b} \in \mathbb{R}^n$ ▷ Sample vector (loaded from disk)
Input: $\mathbf{A} \in \mathbb{R}^{n \times N}$ ▷ Dictionary (loaded from disk)
Initialise: $i \leftarrow 1$ ▷ Initialise iteration counter
Initialise: $\mathbf{x} \leftarrow \mathbf{0}^N$ ▷ Initialise sparse estimate
Initialise: $\mathbf{r} \leftarrow \mathbf{b}$ ▷ Initialise residual vector
Initialise: $\lambda \leftarrow \mathbf{O}^{1 \times 0}$ ▷ Initialise index set
Initialise: $\mathbf{Q} \leftarrow \mathbf{O}^{n \times 0}$ ▷ Initialise orthogonalisation matrix
Initialise: $\mathbf{R} \leftarrow \mathbf{O}^{0 \times 0}$ ▷ Initialise orthogonalisation coefficients
while $\|\mathbf{r}\|_2 \leq \epsilon$ **do** ▷ Stopping criterion
    Section Correlation:
        $\mathbf{c} \leftarrow \mathbf{A^T r}$ ▷ Correlation
        $k \leftarrow \text{argmax}_k c_k$ ▷ Selection
        $\lambda \leftarrow [\lambda\, k]$ ▷ Add selection to index set

    Section MGS update:
        $\mathbf{g} \leftarrow \mathbf{A}_k$ ▷ Column being added to $\mathbf{\Phi}$
        $\mathbf{r} \leftarrow \mathbf{Q}^T\mathbf{g}$ ▷ Find correlations of new column with orthogonal matrix
        $\mathbf{h} \leftarrow \mathbf{g} - \mathbf{Q}\mathbf{r}$ ▷ Find portion of $\mathbf{g}$ orthogonal to $\mathbf{Q}$
        $r_i \leftarrow \|\mathbf{h}\|_2$ ▷ Add normalisation factor to coefficients
        $\mathbf{q} \leftarrow \|\mathbf{h}\|_2^{-1}\mathbf{h}$ ▷ Normalise orthogonal vector
        $\mathbf{Q} \leftarrow [\mathbf{Q}\,\mathbf{q}]$ ▷ Update orthogonal matrix
        $\mathbf{R} \leftarrow [\mathbf{R}\,\mathbf{r}]$ ▷ Update orthogonalisation coefficients

    Section Back substitution:
        $\mathbf{y} \leftarrow \mathbf{Q}^T\mathbf{b}$ ▷ Calculate input vector for back substitution
        $\mathbf{a} \leftarrow \mathbf{0}^i$ ▷ Initialise result vector for back substitution
        for $j \leftarrow i$ **to** 1 **do**
            $a_j \leftarrow (y_j - \mathbf{R}_j\mathbf{a})\, R_{j,j}^{-1}$ ▷ Calculate next element of result vector

    Section Residual update:
        $\mathbf{x} \leftarrow \mathbf{0}^N$ ▷ Initialise sparse estimate
        for $j \leftarrow 1$ **to** i **do**
            $x_{\lambda_j} \leftarrow a_j$ ▷ Calculate each element of sparse estimate
        $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}$ ▷ Residual update

    $i \leftarrow i + 1$ ▷ Increment iteration counter
Output: $\mathbf{x}$ ▷ Sparse estimate saved to disk

3.4 Distributed Orthogonal Matching Pursuit

The objectives of DistOMP are to enable the user of a compute cluster to apply OMP to solve large problems, and solve them quickly. DistOMP uses the techniques introduced by the OMP algorithm detailed in the previous section, specifically, we use the MGS process to update a QR decomposition at each iteration and then use BS to solve the LLSQ problem. DistOMP is based on DistMP, with significant additions to handle the LLSQ solution and residual update of OMP.

The distributed algorithm is designed by first considering data storage. The three largest data structures in DistOMP are $\mathbf{A}$, $\mathbf{Q}$, and $\mathbf{R}$. The dictionary $\mathbf{A}$ will be distributed across workers in the same manner as in DistMP. In the next section I describe how the matrices $\mathbf{Q}$ and $\mathbf{R}$ are stored. This determines the structure of workers in DistOMP, which influences the distribution of computation between the workers. Finally, I will present the algorithms for DistOMP along with a line-by-line explanation.

3.4.1 Storage and organisation

This algorithm is split into two types of workers; the QA-worker and the R-worker, which store different data and carry out different computation. They are named for the data they store (as this also largely determines the computation they carry out): QA-workers together store $\mathbf{Q}$ and $\mathbf{A}$, and the R-worker stores $\mathbf{R}$. Figure 3.3 shows the portions of $\mathbf{A}$, $\mathbf{Q}$, and $\mathbf{R}$ which are stored on each worker in a simple problem example. In this section I will explain the reasoning behind the use of this structure.

The motivation of DistMP was that to be able to solve large problems efficiently we need to split $\mathbf{A}$ across multiple workers, both to gain access to more storage and to distribute computation across many processors. $\mathbf{Q}$ grows by one column at each iteration and so has a size of n rows by i columns at the ith iteration. So that computation involving $\mathbf{Q}$ remains well-distributed over workers even as it grows, we assign the columns of $\mathbf{Q}$ to workers in a round-robin fashion: With W workers, at the ith iteration we assign the new column of $\mathbf{Q}$, $\mathbf{Q}_i$, to worker $w = 1 + (i - 1) \bmod W$ (referred to as the *Q-holder* for this iteration). This keeps the columns as well-distributed as possible as $\mathbf{Q}$ grows. I shall show later how this distribution of $\mathbf{Q}$'s columns leads to simple and efficient parallelisation of the MGS and LLSQ processes.

In a similar manner to how we use the set $\mathcal{W}_w$ to denote the set of columns of $\mathbf{A}$ held on the worker w, we use the set $\mathcal{Q}_{w,i}$ to indicate which columns of $\mathbf{Q}$ are held on each QA-worker w at iteration i. Each $\mathcal{Q}_{w,i}$ contains

$$\mathcal{Q}_{w,i} \leftarrow \left\{ w,\ w + W,\ w + 2W,\ \ldots,\ \left\lfloor \frac{i - w}{W} \right\rfloor W + w \right\},$$

The final matrix to store is $\mathbf{R}$, the coefficients of the orthogonal QR decomposition. At each iteration i, $\mathbf{R}$ has a size of i rows by i columns. OMP, and therefore DistOMP, never takes more than n iterations to complete, so the size of $\mathbf{R}$ is bounded by $n \times n$. In practical CS problems, n

is generally much smaller than N, so $\mathbf{R}$ will generally be much smaller than $\mathbf{A}$. For this reason, in DistOMP $\mathbf{R}$ is not split up and is stored on a single worker. I will show later how this is also a sensible choice from the point-of-view of distributing computation.

In DistMP, communications take place across the world communicator, which includes all workers. Because DistOMP has two different types of worker, it now becomes desirable to introduce some further organisation. The QA-workers are all added to an MPI *group*, a type of communicator. QA-workers are numbered within this group, excluding the R-worker. This means that group communication primitives (such as **Broadcast** and **Reduce**) can be run across the group of QA-workers without including the R-worker. Communications between QA-workers and the R-worker take place over the world communicator, which still includes all workers. In the following algorithms I use W to mean the number of QA-workers and refer to the QA-workers by their group numbering, i.e., $1 \leq w \leq W$. There are actually $W + 1$ MPI workers in total including the R-worker, but for clarity the R-worker is referred to by name instead of by number.

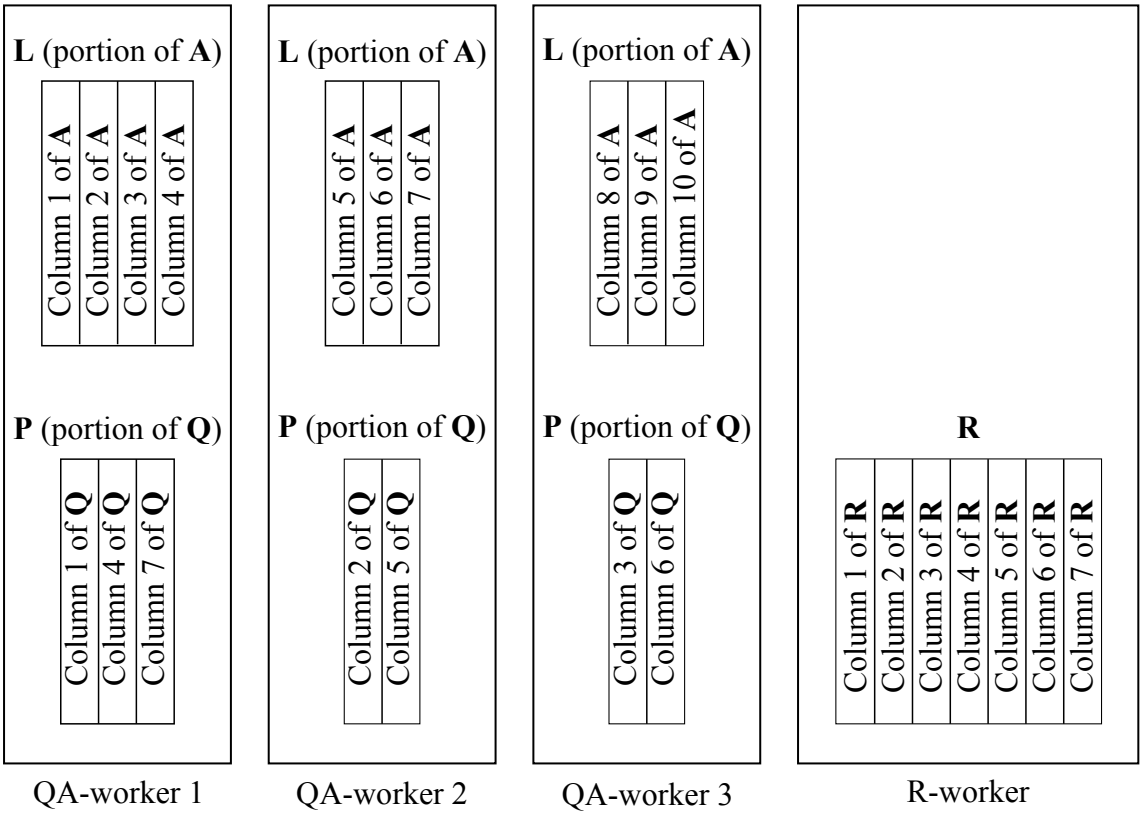


Figure 3.3: The large matrices stored on the different workers in a DistOMP setup with 3 QA-workers. The state shown is after the 7th iteration, so $\mathbf{Q}$ and $\mathbf{R}$ each have 7 columns. $N = 10$ and n is not shown on this diagram.

3.4.2 Distribution of computation

Each QA-worker stores $\mathbf{L}$ (a portion of the dictionary $\mathbf{A}$) and $\mathbf{P}$ (a portion of the orthogonal decomposition $\mathbf{Q}$). The computation carried out by the QA-workers is correlation, selection, MGS update, and residual update. Correlation and selection naturally distribute along with the columns of $\mathbf{A}$ in the same manner as in DistMP. The majority of the computation in the MGS update depends on the columns of $\mathbf{Q}$ and this computation naturally distributes with the columns of $\mathbf{Q}$ and so is distributed over the QA-workers. Finally, the residual update strategy used in DistOMP depends on the columns of $\mathbf{A}$ and so is distributed along with these columns, over the QA-workers. In order to allow for the use of a large dictionary (and therefore to allow $\mathbf{Q}$ to also grow large) there may be a large number of QA-workers. I will also show in the next chapter that, up to a point (which depends on problem size), increasing the number of QA-workers reduces the time taken for the algorithm to complete.

The primary compute task of the R-worker is to carry out back-substitution in each iteration. Unlike MGS, the algorithm for BS is not amenable to parallelisation: Calculating each step in BS depends on knowledge of every element of $\mathbf{a}$ found so far. Since each step relies on the result of the previous steps, BS is inherently sequential and the steps cannot easily be carried out in parallel, or indeed with any overlap. Therefore, the fastest way to carry out BS is for the entirety of $\mathbf{R}$ to reside on a single worker node which carries out BS by itself, with no parallelism.

In the next two sections I give the full algorithms for the QA-workers and R-worker, followed by Figure 3.4 which shows the communications between the workers in a single iteration and a summary of the computation carried out by each worker. I give a detailed explanation of the algorithms cross-referenced to each line in the algorithms (e.g. Line 1) and the numbered steps in Figure 3.4 (e.g. **Step 1**).

3.4.3 QA-worker algorithm

In this section I describe the QA-worker, which carries out a distributed MGS decomposition in order to solve a LLSQ problem, as well as other processes including the residual update calculation. The full algorithm for this worker is given in Algorithm 3.7.

The inputs to the QA-worker are the target signal $\mathbf{b}$, and this worker's portion of the dictionary, $\mathbf{L}$, both of which are loaded from disk before the algorithm begins (Lines 1–2). The MPI environment supplies variables indicating the number of QA-workers in the cluster (W), and w, the index of this worker (Lines 3–4).

Several variables are initialised for later use in the algorithm. $\mathbf{r}$ holds the current residual, which is initialised to $\mathbf{b}$ (Line 5). Indices of selected columns will be held in λ, which starts as an empty row-vector (Line 6). This worker's portion of $\mathbf{Q}$ (one output of the MGS QR decomposition) is held in the matrix $\mathbf{P}$, which is initially an empty matrix (specifically, n rows by no columns, Line 7)). Finally, i, the iteration counter, starts at 1 (Line 8).

The distributed correlation and selection algorithms of DistOMP are identical to those of

DistMP up to and including the broadcast of $\mathbf{g}$ (Lines 10–21, **Steps 1–2**), so I will not repeat my explanation of those lines. The first difference is that we do not store or broadcast the selected column's correlation (known as γ in DistMP), as this is not used in DistOMP. The next difference is that DistOMP requires us to remember the indices of the selected columns. Every QA-worker holds a copy of the row-vector λ and after receiving the broadcast of K (the result of the global argmax operation, Line 23) they append K to λ (Line 25).

The next step is to begin the distributed MGS decomposition update, taking into account the addition of $\mathbf{g}$ to the working set. The first step in the MGS decomposition update is to take $\mathbf{g}$ and make it perpendicular to the basis $\mathbf{Q}$. This is done by taking each column $\mathbf{Q}_j$ in $\mathbf{Q}$ and subtracting from $\mathbf{g}$ any portion correlating with $\mathbf{Q}_j$ by doing $\mathbf{h} \leftarrow \mathbf{g} - \mathbf{Q}_j^T\mathbf{g}\,\mathbf{Q_j}$.

Calculating the correlations $\mathbf{Q}_j^T\mathbf{g}$ can be done in parallel. This operation is especially suited to parallelisation because the columns of $\mathbf{Q}$ are spread over all of our workers, so every QA-worker w can calculate the correlation of $\mathbf{P} = \mathbf{Q}_{\mathcal{W}_w}$, its subset of $\mathbf{Q}$, with $\mathbf{g}$ at the same time (Line 28). Next, rather than individually subtracting each $\mathbf{P}_j^T\mathbf{g}\,\mathbf{P_j}$ from $\mathbf{g}$ one at a time, we can find the sum of all of these vectors in a distributed manner and perform the subtraction in one operation. The matrix-vector product $\mathbf{t} \leftarrow \mathbf{Ph}$ (Line 29, **Step 3**) computes the sum of contributions on a particular node. Since only one worker (On Line 30 we check if we are the *Q-holder* for this iteration) needs to know the end result, each worker w sends its $\mathbf{t}$ to the Q-holder (Line 40, **Step 4**), which receives $\mathbf{t}$ (Line 34) and performs the subtraction (Line 35).

The QA-holder for this iteration then carries out a few more steps to update its portion of $\mathbf{Q}$: It normalises $\mathbf{g}$ (Line 37) and sends the normalisation factor to the R-worker (**Step 5**, sent on Line 36, received at Algorithm 3.8 Line 5), then appends the remaining vector to $\mathbf{P}$ (Line 38). The final step in the MGS update, every QA-worker sends its correlations vector $\mathbf{h}$ to the R-worker (**Step 6**, sent on Line 42, received at Algorithm 3.8 Line 9), where these will be combined and put in the latest column of $\mathbf{R}$.

After the MGS update, we need to calculate the inner product $\mathbf{y} \leftarrow \mathbf{Q}^T\mathbf{b}$ in preparation for BS. Since every worker holds a copy of $\mathbf{b}$ as well as a subset of columns of $\mathbf{Q}$, we can do this as a distributed dot product. Each QA-worker w calculates the dot product $\mathbf{z} \leftarrow \mathbf{Pb}$ (Line 45, **Step 7**) and sends the result to the R-worker (**Step 8**, sent on Line 46, received at Algorithm 3.8 Line 16), which collects the results into $\mathbf{y}$ and then carries out back substitution (**Step 9**, I will detail the R-worker's algorithm later).

After the R-worker has carried out BS on $\mathbf{R}$ and $\mathbf{y}$ it sends the result, $\mathbf{a}$, back to all of the QA-workers (**Step 10**, Algorithm 3.8 Line 25). Each QA-worker receives $\mathbf{a}$ (Line 47) and converts $\mathbf{a}$ to $\mathbf{x}$, that is, it converts the vector of coefficients from being indexed in terms of $\mathbf{\Phi}$ to being indexed in terms of $\mathbf{A}$. This is done using λ, in the same way as with the single-threaded OMP algorithm: Each element of λ tells us to which element of $\mathbf{x}$ we should move the corresponding element of $\mathbf{a}$ (Lines 51–52).

The QA-workers carry out the residual update step by means of a distributed dot product. Overall the aim is to carry out $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{Ax}$. Every worker holds a copy of $\mathbf{b}$ and $\mathbf{x}$, but $\mathbf{A}$

Algorithm 3.7 Distributed Orthogonal Matching Pursuit: QA Worker

Input: $\mathbf{b} \in \mathbb{R}^n$ ▷ Samples (loaded from disk)
Input: $\mathbf{L} = \mathbf{A}_{\mathcal{W}_w}$ ▷ Local dictionary (loaded from disk)
Input: $W \in \mathbb{Z}, W \geq 1$ ▷ Number of QA-workers (from MPI)
Input: $w \in \mathbb{Z}, 1 \leq w \leq W$ ▷ This QA-worker's index (from MPI)
Initialise: $\mathbf{r} \leftarrow \mathbf{b}$ ▷ Initialise residual
Initialise: $\lambda \leftarrow \mathbf{O}^{1\times 0}$ ▷ Initialise index set
Initialise: $\mathbf{P} \leftarrow \mathbf{O}^{n\times 0}$ ▷ Initialise local component of orthogonalised matrix
Initialise: $i \leftarrow 1$ ▷ Initialise iteration counter
while $\|\mathbf{r}\|_2 \geq \epsilon$ **do** ▷ Stopping criterion
    Section Correlation and selection:
        $\mathbf{c} \leftarrow \mathbf{L}^T\mathbf{r}$ ▷ Correlation
        $k \leftarrow \text{argmax}_k |c_k|$ ▷ Local selection
        Reduce $v \leftarrow \underset{w}{\text{argmax}} |c_k|$ ▷ Find which worker holds the highest correlation
        if $v = w$ **then** ▷ If we hold the selected worker:
            if $w \leq N \bmod W$ **then** ▷ Convert local index to global
                $K \leftarrow (w-1)\left\lceil \frac{N}{W} \right\rceil + k$
            else
                $K \leftarrow (w-1)\left\lfloor \frac{N}{W} \right\rfloor + N \bmod W + k$
            $\mathbf{g} \leftarrow \mathbf{L}_K$ ▷ Store the selected column
            Broadcast K ▷ Broadcast global index
            Broadcast $\mathbf{g}$ ▷ Broadcast selected column
        else ▷ Otherwise:
            Receive Broadcast K ▷ Receive global index
            Receive Broadcast $\mathbf{g}$ ▷ Receive selected column
        $\lambda \leftarrow [\lambda\, K]$ ▷ Update index set

    Section Modified Gram-Schmidt decomposition:
        $\mathbf{h} \leftarrow \mathbf{P}^T\mathbf{g}$ ▷ Correlate new column with local orthogonal matrix
        $\mathbf{t} \leftarrow \mathbf{P}\mathbf{h}$ ▷ Sum contributions of local orthogonal matrix
        if $w = 1 + (i-1) \bmod W$ **then** ▷ If we are the Q-holder
            $\mathbf{g} \leftarrow \mathbf{g} - \mathbf{t}$ ▷ Subtract our $\mathbf{P}$'s contributions
            for $v \leftarrow 1$ **to** W **do**
                if $v \neq w$ **then** ▷ For every other worker:
                    Receive $\mathbf{t}$ **from** QA-worker v ▷ Receive its contributions to $\mathbf{G}$
                    $\mathbf{g} \leftarrow \mathbf{g} - \mathbf{t}$ ▷ And subtract them from $\mathbf{g}$
            Send $\|\mathbf{g}\|_2$ **to** R-worker ▷ Send the normalisation factor to the R-worker
            $\mathbf{g} \leftarrow \|\mathbf{g}\|_2^{-1}\mathbf{g}$ ▷ Normalise orthogonal vector
            $\mathbf{P} \leftarrow [\mathbf{P}\,\mathbf{g}]$ ▷ Update orthogonal matrix
        else ▷ If we aren't the Q-holder
            Send $\mathbf{t}$ **to** QA-worker $1 + (i-1) \bmod W$
            ▷ Send our contributions vector to the Q-holder
        Send $\mathbf{h}$ **to** R-worker ▷ Send our correlations vector to the R-worker

44: **Section** Preparation for back substitution:
45: $\mathbf{z} \leftarrow \mathbf{P}^T\mathbf{b}$ ▷ Calculate input vector for back substitution
46: **Send** $\mathbf{z}$ **to** R-worker ▷ Send input vector to R-worker
47: **Receive** $\mathbf{a}$ **from** R-worker ▷ Receive result from back-substitution
48:
49: **Section** Residual update:
50: **for** $j \leftarrow 1$ **to** i **do**
51: $\alpha \leftarrow \lambda_j$ ▷ Convert $\mathbf{\Phi}$ index to $\mathbf{A}$ index
52: $x_\alpha \leftarrow a_j$ ▷ Calculate sparse estimate
53: $\mathbf{d} \leftarrow \mathbf{L}\mathbf{x}_\mathcal{W}$ ▷ Multiply sparse estimate by local portion of dictionary
54: **Reduce** $\mathbf{b}_i \leftarrow \sum \mathbf{d}$ ▷ Sum result over all nodes
55: $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{b}_i$ ▷ Residual update
56:
57: **if** $w = 1$ **then** ▷ If we are the root Q-worker:
58: **Send** $\|\mathbf{r}\|_2$ **to** R-worker ▷ Send the residual's magnitude to the R-worker
59: $i \leftarrow i + 1$ ▷ Increment iteration counter
60: **Output:** $\mathbf{x}$ ▷ Sparse estimate saved to disk

is distributed over them. Each worker w calculates $\mathbf{d} \leftarrow \mathbf{L}\mathbf{x}_\mathcal{W}$ (Line 53, **Step 11**), where $\mathbf{x}_\mathcal{W}$ is the elements of $\mathbf{x}$ which correspond to the columns of $\mathbf{A}$ held by a particular worker. To calculate our current signal estimate, we can then sum $\mathbf{d}$ over all the QA-workers. Since all workers will need the updated residual, we perform a global reduce operation of $\sum \mathbf{d}$ with all workers receiving the result (Line 54, **Step 12**). Finally, each worker can subtract the result from $\mathbf{b}$ yielding the updated residual vector $\mathbf{r}$ (**Step 13**, Line 55).

To exit cleanly every worker must terminate, which is why the residual magnitude is evaluated on every worker. Since the R-worker does not know the residual it is unable to calculate its magnitude, so the root QA-worker sends the magnitude of the residual to the R-worker (**Step 14**, sent on Line 58, received at Algorithm 3.8 Line 5) so the R-worker can evaluate the stopping criterion. Then, every worker determines when to terminate the algorithm by comparing the magnitude of the residual, $\|\mathbf{r}\|_2$, to the target precision ϵ (**Step 15**, Line 9). An iteration limit may also be imposed in order to limit the number of non-zero elements in the sparse estimate output, or in order to ensure the algorithm still terminates in the case of significant numerical inaccuracy or ill-posed problem. If any of the stopping criteria are met, the sparse estimate is saved to disk as output by the root worker (Line 60). Otherwise, the final step in the algorithm is to increment the iteration counter (Line 59).

3.4.4 R-worker algorithm

The algorithm for the R-worker is shown in Algorithm 3.8. In summary, it receives from QA-workers the data needed to construct the $\mathbf{R}$ matrix, stores the matrix in memory, and carries out the back-substitution algorithm before sending the result back to the QA-workers.

All problem-specific data is received from the QA-workers: the R-worker does not load any

inputs from disk. It does, however, know the total number of workers in the problem (from the MPI environment, Line 1), because it needs to know how many QA-workers to communicate with. Before beginning, the iteration counter is initialised to 1 (Line 2) and $\mathbf{R}$ is initialised to an empty matrix with no rows or columns (Line 3).

During the MGS update step, the first contribution towards $\mathbf{R}$ sent from the QA-workers to the R-worker is the magnitude of the new column of $\mathbf{Q}$, prior to normalisation (**Step 5**, received on Line 5, sent at Algorithm 3.7 Line 36). The R-worker puts the magnitude into the bottom-right element, $R_{i,i}$ (Line 7). Next, the R-worker receives from each QA-worker the correlations between the new column being added and that QA-worker's columns of $\mathbf{Q}$ (**Step 6**, received on Line 9, sent at Algorithm 3.7 Line 42). As the columns of $\mathbf{Q}$ are striped across the QA-workers, the coefficients received from each QA-worker need to be re-ordered and inserted into the appropriate parts of the new column of $\mathbf{R}$. The set $\mathcal{Q}_{w,i}$, defined earlier, indicates the column indices of $\mathbf{Q}$ which are held on the QA-worker w at iteration i, thus, it tells us where to insert the received coefficients. We iterate over each element of the set and store the element of $\mathbf{h}$ in the appropriate element of $\mathbf{R}$ as indicated by $\mathcal{Q}_{w,i}$ (Line 12).

After MGS and updating $\mathbf{R}$, the next step is to prepare for back substitution. The input vector for back substitution is received in pieces from all the QA-workers, and the R-worker collates the portions into a single vector. This vector is first initialised to zeros (Line 14). Then, the vector portion is received from each QA-worker (**Step 8**, received on Line 16, sent at Algorithm 3.7 Line 46) and element-by-element stored in the appropriate location in $\mathbf{y}$ (Line 19).

With $\mathbf{R}$ updated and this iteration's $\mathbf{y}$ pieced together, the R-worker can carry out BS in the same way as the single-threaded OMP algorithm given previously (**Step 9**). The result of BS will be the vector of coefficients, $\mathbf{a}$. This is initialised to zeros (Line 21) and then each element is calculated in order (Line 23). Upon completion, the result is sent to every QA-worker (**Step 10**, Line 25, received at Algorithm 3.7 Line 47) so they can use it to calculate $\mathbf{x}$ and then carry out the residual update.

As with the QA-workers, in order to terminate the program cleanly, every worker must terminate individually. The stopping criterion is when the magnitude of the residual, $\|\mathbf{r}\|_2$, becomes less than the desired precision ϵ. After finishing the distributed residual update algorithm, the first QA worker ($w = 1$) sends the magnitude of the residual ($\|\mathbf{r}\|_2$) back to the R-worker (**Step 14**, received on Line 5, sent at Algorithm 3.8 Line 58). The R-worker compares this to the stopping criterion $\mathbf{epsilon}$ to determine whether to terminate the worker after this iteration finishes (Line 29, **Step 15**).

Algorithm 3.8 Distributed Orthogonal Matching Pursuit: R-worker

Input: $W \in \mathbb{Z}, W \geq 1$ ▷ Number of QA-workers (from MPI)
Initialise: $i \leftarrow 1$ ▷ Initialise iteration counter
Initialise: $\mathbf{R} \leftarrow \mathbf{O}^{0\times 0}$ ▷ Initialise orthogonalisation coefficients
repeat
    Receive $\|\mathbf{g}\|_2$ **from** QA-worker $1 + (i-1) \bmod W$
        ▷ Receive normalisation factor from Q-holder
    $R_{i,i} \leftarrow \|\mathbf{g}\|_2$ ▷ Store normalisation factor in $\mathbf{R}$
    for $v \leftarrow 1$ **to** W **do** ▷ For each QA-worker:
        Receive $\mathbf{h}$ **from** QA-worker v ▷ Receive its contributions vector
        $j \leftarrow 1$
        for β **in** $\mathcal{Q}_{v,i}$ **do**
            $R_{\beta,i} \leftarrow h_j$ ▷ Store each element in $\mathbf{R}$
            $j \leftarrow j + 1$
    $\mathbf{y} \leftarrow \mathbf{0}^n$ ▷ Initialise back substitution input vector
    for $v \leftarrow 1$ **to** W **do** ▷ For each QA-worker:
        Receive $\mathbf{z}$ **from** QA-worker v ▷ Receive partial input vector for back-substitution
        $j \leftarrow 1$
        for β **in** $\mathcal{Q}_{v,i}$ **do**
            $y_\beta \leftarrow z_j$ ▷ Integrate it into input vector for back-substitution
            $j \leftarrow j + 1$
    $\mathbf{a} \leftarrow \mathbf{0}^i$ ▷ Initialise result vector for back-substitution
    for $j \leftarrow i$ **to** 1 **do**
        $a_j \leftarrow y_j - \mathbf{R}_j\mathbf{a}$ ▷ Calculate next element of result vector
    for $v \leftarrow 1$ **to** W **do**
        Send $\mathbf{a}$ **to** QA-worker v ▷ Send result vector to every QA-worker
    Receive $\|\mathbf{r}\|_2$ **from** QA-worker 1
        ▷ Receive residual's magnitude from root QA-worker
    $i \leftarrow i + 1$ ▷ Increment iteration counter
until $\|\mathbf{r}\|_2 \leq \epsilon$ ▷ Stopping criterion

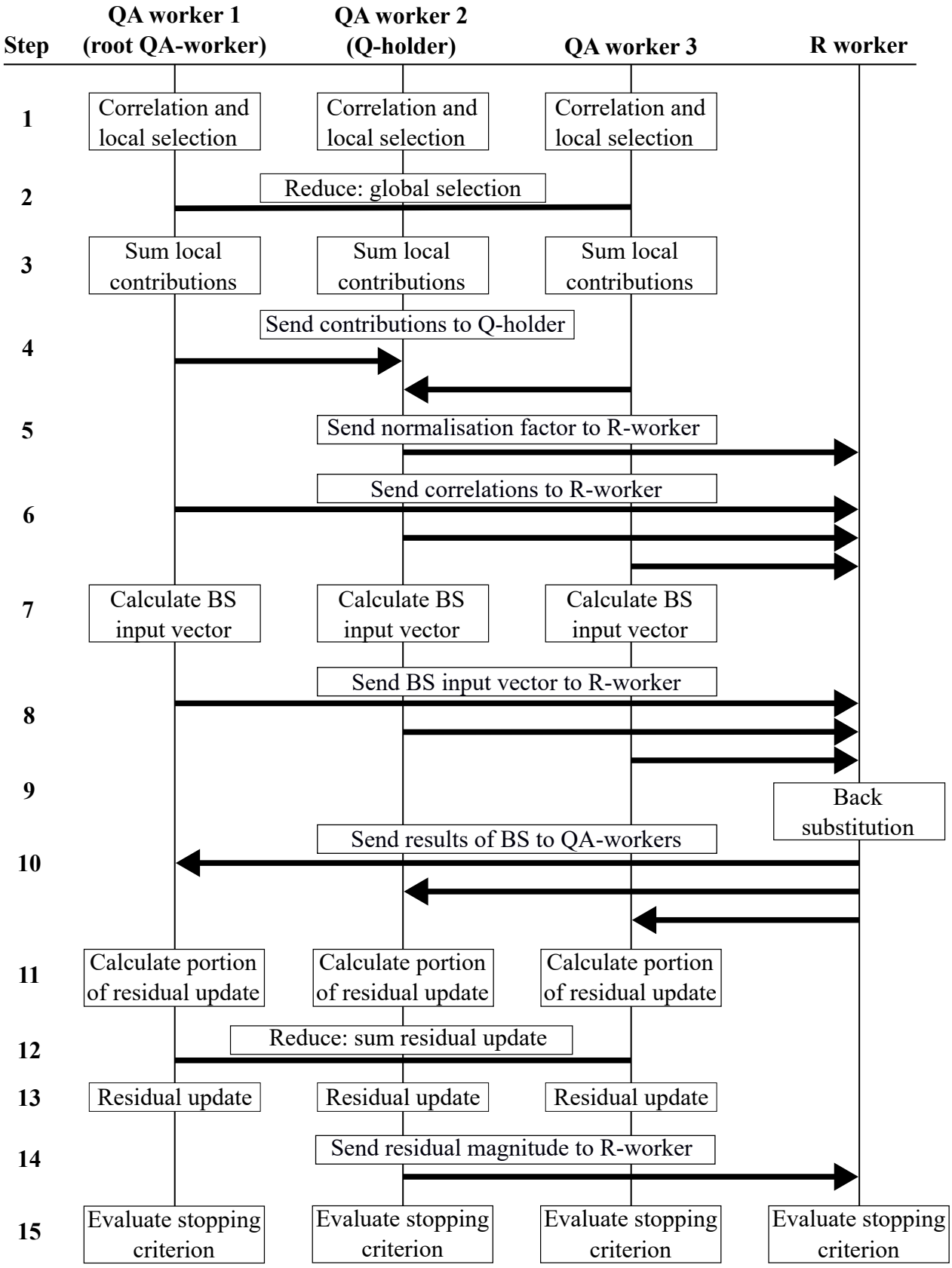


Figure 3.4: The communications between workers in a DistOMP setup with 3 QA-workers. In this iteration the second QA-worker is the Q-holder.

3.5 Chapter summary

In this chapter I introduced two novel algorithms I have developed, DistMP and DistOMP. In each case I first gave a detailed description of the pre-existing single threaded algorithm, MP and OMP respectively. With DistMP I showed how a data-parallel approach would allow solution of much larger problems than were previously possible, as well as allowing compute parallelism giving the possibility for speed-up as more workers are added to the problem. For DistOMP I first detailed a particular strategy for implementing OMP using MGS updates to a QR decomposition. I then used a data-parallel structure similar to DistMP but also including distribution of the QR decomposition and associated computation. Finally, I detailed the algorithms for the two types of workers, QA-workers and R-workers, and the communications involved between the workers.

Having explained the design and structure of DistMP and DistOMP and detailed the algorithms, the next step is to present an implementation of the algorithms which may be used to evaluate their performance. In the next chapter I will show how I developed implementations of all of these algorithms and used a large compute cluster to characterise their performance, comparing DistMP and DistOMP against MP and OMP respectively, and investigating how the time taken to solve problems depends on the number of workers assigned to each algorithm.

Chapter 4

Experimental results and analysis

In this chapter I present experimental results evaluating my parallel algorithms against a single-threaded implementation and a basic multithreaded implementation. I first describe the Darwin compute cluster which was used to run these experiments. I give details of the implementations, the language, libraries, and tools used. I then describe some decisions made when implementing the algorithms, in particular with respect to the parallel architecture. I explain how I will interpret the results and give an example of the raw data collected. Presenting the results, I first compare MP with DistMP and characterise the performance of each, looking at time taken to solve problems of various shapes and sizes using different numbers of threads with MP and different numbers of workers with DistMP. After discussing these results, I present the same characterisation and comparison between OMP and DistOMP. Finally, I demonstrate how DistOMP can be used to solve a problem with a 429GB equivalent sampling matrix by utilising a large number of workers.

4.1 The Darwin cluster

The experiments described in this chapter were executed on the Darwin cluster, a general purpose compute cluster run by the University of Cambridge's High Performance Computing Service (HPCS). At the time when these experiments were run (this cluster has since been decommissioned and replaced), Darwin consisted of 600 nodes, each with 16 Intel "Sandy Bridge" cores and 64GB of RAM, for a total of 9600 cores and 38.4TB of RAM. Within each node the cores were split over two processor sockets, so that each socket had 8 cores and 32GB of RAM. Each socket could access the RAM attached to the other socket but with increased latency and reduced bandwidth. This means each node was overall a NUMA architecture, with each socket and its directly attached RAM being a UMA domain. All nodes were connected to a full bisectional bandwidth interconnect network, with dual Infinniband links giving a bandwidth of 40Gb/s to each node. 433TB of storage was provided by several Lustre distributed filesystems connected to the Infinniband network.

4.2 Implementing the algorithms

In this section I give the details of my implementation of the algorithms described in the previous chapter and the environment in which experiments were run. I include details of the languages and libraries used as well as decisions taken when implementing the algorithms. By using the guidance in this section to implement the algorithms and running experiments in a comparable environment, readers should be able to reproduce the results presented.

Implementations of MP, OMP, DistMP, and DistOMP were developed using the Rust Language [92]. Rust was used because it should give an execution speed similar to C while providing a more modern and user-friendly environment and enhanced compile-time error checking, which accelerates development due to reduced debugging. OpenBLAS [93] is used via the Stainless Steel bindings [94] for basic linear algebra operations in all four algorithms. For the multithreaded baseline, the MP and OMP algorithms use OpenBLAS's built-in support for multithreading, which is based on OpenMP [95]. OpenBLAS multithreading uses a worker pool architecture: when the library is initialised worker threads are started for each processor core available and remain idle until jobs are submitted by a BLAS operation. Each BLAS operation uses a heuristic based on vector and matrix size to estimate whether parallelisation is worthwhile or whether overheads will dominate speed-up. If parallelisation is used then the matrices and vectors are partitioned using data decomposition and tasks are assigned to the worker pool. After all tasks finish a gather operation may be used to combine the output from each worker. The parallel algorithms, DistMP and DistOMP, use OpenMPI via the rsmpi [96] bindings for message passing communications between compute workers. rsmpi simply maps types and function calls between Rust and C and so should perform identically to the OpenMPI C API.

The heterogeneous structure of a high-performance compute cluster can be viewed in a simplified manner where each processor core is considered an UMA domain along with an area of memory dedicated to it. An MPI worker is run on each core. Message passing is used for communications between any two workers, whether they reside within the same processor socket or on a different node entirely. In theory, this should be less efficient than an algorithm architected for the NUMA structure of the cluster, however a flat organisation is much simpler to design and implement. This is the strategy used for the algorithms shown previously and the implementations discussed in this chapter.

In theory, OMP can be run without any iteration limit or guidance as to how many iterations it may expect to run for. The only stopping criterion is the remaining magnitude of the residual. The index set and working set (or, in the QR decomposition implementation, the index set, $\mathbf{Q}$, and $\mathbf{R}$) start off with zero size and grow in each iteration. This was implicitly the case in the algorithms given in the previous chapter. However, this sort of implementation poses some challenges: Having these variables grow necessitates dynamic memory allocation within the course of the algorithm. This introduces the possibility for slow-downs due to memory

reallocation, or even crashes if memory allocation fails. The algorithm can be much simpler and faster if these matrices are pre-allocated with a fixed size: The algorithm keeps track of the effective size of variables and ignores the unused portion. Instead of reallocating matrices to increase their size, the algorithm simply increases the variable describing the effective size of the matrix. The BLAS API is defined assuming all vectors and matrices are allocated within a larger buffer: For each operation, BLAS accepts parameters describing the start position of the vector or matrix, its shape, and the gap between elements or rows (referred to as *stride*). Since all vector and matrix operations in our implementations are carried out using BLAS, the use of pre-allocated oversized buffers and dynamic resizing of the contained matrices and vectors incurs no performance overhead or increase in complexity.

In order to pre-allocate buffers for these matrices we need to know the maximum size they could grow to, otherwise the algorithm may be forced to terminate prematurely if the matrix turns out to not have been allocated large enough for the problem being solved. In OMP, knowing the maximum number of iterations determines the maximum size that the index set, working set, $\mathbf{Q}$, and $\mathbf{R}$ can grow to. In addition to allowing pre-allocation of the matrices, imposing an iteration limit also ensures the algorithm will always terminate even in the event of numerical error or instability causing the algorithm to not converge. Conveniently, we can place an upper bound on the iteration limit because we know that OMP, with a dictionary of n rows by N columns, will find a solution in n iterations or less. If we have more information about the problem, in particular the expected sparsity level of the solution, we may be able to reduce this bound in order to reduce the surplus memory requirements of the algorithm and reduce the algorithm's execution time in the case of non-convergence. In our case, since the problems are synthetic, we know precisely the sparsity level of the solution and can set an iteration limit based on this. In realistic applications we are likely to be able to estimate the sparsity level in the solution and set an iteration limit based on this. In some scenarios we may be able to fully sample the signal once or occasionally, calculate its sparsity, and assume its sparsity level is unlikely to change significantly over time. Alternatively, if online reconstruction and feedback between reconstructor and sampler are possible, a closed-loop adaptive system as described by Chen and Wassell [97] could be used. Lopes [98][99] describes techniques where various proxies for sparsity can be estimated from a small number of linear samples and used to derive the number of measurements needed for guaranteed reconstruction.

Next, we turn our attention to the matrix of orthogonal coefficients, $\mathbf{R}$. Due to the properties of the QR decomposition, we know that $\mathbf{R}$ will always be upper triangular. For simplicity, my implementation stores the full square matrix. Since we know that just under half of the matrix will always be zero, the memory used for storing $\mathbf{R}$ could be reduced by just under half by using a more sophisticated storage scheme which took into account the structure of the matrix. However, such storage schemes are more complicated and may be slower. In the example problems solved in this chapter, the memory space taken by $\mathbf{R}$ is not a significant limitation, and for simplicity the R-worker was allocated a full node anyway, meaning there would be no benefit to reducing

its memory usage. The memory usage of $\mathbf{R}$ would be more significant in a problem requiring many iterations to solve, or when using a very large number of workers (so that $\mathbf{L}$ and $\mathbf{P}$ are smaller in comparison to $\mathbf{R}$).

4.3 Methodology

We wish to discover how the time taken for DistMP and DistOMP to solve problems depends on the number of workers. We also wish to determine how this dependence varies with problem size (specifically, the number of dictionary columns) and sparsity (the number of non-zero elements in the sparse vector). As a baseline, we will compare against simple multithreaded implementations of MP and OMP using different numbers of threads. In summary, the independent variables are algorithm (MP, OMP, DistMP, DistOMP), number of workers or threads, problem size, and problem sparsity. In all cases our dependant variable is wall-time elapsed in solving the problem.

Comparing all possible combinations of these independent variables would require an impractical number of experiments and produce a volume of results which would be difficult to interpret. Furthermore, we wish to trial DistMP and DistOMP with a large numbers of workers spread over multiple compute nodes, but the multithreaded MP and OMP algorithms are limited to the 16 threads available on a single node. To resolve this I selected a subset of independent variable combinations to give a reasonable number of experiments, explore the problem space and demonstrate the relationships between the variables.

I define three sizes of problem: large, medium, and small, corresponding to $N = 524288$, $N = 327680$, and $N = 131072$ respectively. I also define two sparsity levels: sparse and dense, corresponding to $S = 164$ and $S = 328$ respectively. I established in Section 2.2.3 that spark$(\mathbf{A}) > 2S$ is a necessary condition for guaranteed reconstruction, which requires that $n > 2S$. This condition is not sufficient for reconstruction, partly because perfect incoherence is not guaranteed with IID random matrices, and partly because reconstruction algorithms are non-ideal. Generally $n \geq CS\log(N)$ samples are required where C is a constant depending on the reconstruction algorithm and type of sample matrix [100]. Empirical results presented by Tropp and Gilbert [10] demonstrate reconstruction with high probability using OMP when $n \geq 1.5S\log(N)$ which was used to set the number of samples taken as $n = 20S$, or $n = 3277$ and $n = 6554$ respectively for the sparse and dense problems.

To establish a baseline, I trial the multithreaded implementations of MP and OMP with a large, sparse, problem and every number of threads between 1 and 15 inclusive. Although 16 threads are available, one is left idle for system processes in order to reduce the variability in results, hence the maximum of 15 threads.

To explore the effect that number of workers has on the solution time of DistMP and DistOMP I ran trials with 1, 15, 30, 60, 90, 120, 150, 180, 210 and 240 workers running on 1, 1, 2, 4, 6, 8, 10, 12, 14, and 16 nodes respectively. For each number of workers I ran trials with every combination of algorithm (DistMP or DistOMP), size (large, medium or small) and sparsity

Section 4.5.1: Multithreaded MP					
Threads	Algorithm	N	n	S	Repetitions
1–15	MP	524,288	3,277	164	10
Section 4.5.2: DistMP					
Workers	Algorithm	N	n	S	Repetitions
1–240	DistMP	131,072	3,277	164	10
1–240	DistMP	131,072	6,554	328	10
1–240	DistMP	327,680	3,277	164	10
1–240	DistMP	327,680	6,554	328	10
1–240	DistMP	524,288	3,277	164	10
1–240	DistMP	524,288	6,554	328	10
Section 4.5.3: Multithreaded OMP					
Threads	Algorithm	N	n	S	Repetitions
1–15	OMP	524,288	3,277	164	10
Section 4.5.4: DistOMP					
Workers	Algorithm	N	n	S	Repetitions
1–240	DistOMP	131,072	3,277	164	10
1–240	DistOMP	131,072	6,554	328	10
1–240	DistOMP	327,680	3,277	164	10
1–240	DistOMP	327,680	6,554	328	10
1–240	DistOMP	524,288	3,277	164	10
1–240	DistOMP	524,288	6,554	328	10
Section 4.5.5: Large demonstration problem					
Workers	Algorithm	N	n	S	Repetitions
240	DistOMP	2,400,000	24,000	2,400	10

Table 4.1: Summary of experiments

(sparse or dense).

Significant variability was observed in the result between runs of the same trial, as illustrated in the next section. To mitigate the variability and uncover the underlying trends, 10 trials were run for every scenario described previously.

For ease of comparison, all trials with the same size and sparsity, regardless of algorithm, solve exactly the same problem. For repeatability, the problems are generated using a Pseudo-Random Number Generator (PRNG) initialised with a fixed seed. After seeding the PRNG, the sparse signal is generated: A vector of N zeroes is created. S of its elements are selected pseudo-randomly and set to a value of 1. Next, the dictionary is generated, by filling an n-row by N-column matrix with pseudo-random Gaussian-distributed IID values. Finally, the samples vector is generated by multiplying the sparse vector by the dictionary. The dictionary, sparse vector, and samples vector are then saved to disk. This process is repeated for each distinct combination of N, n and S.

For each trial, the appropriate problem data is loaded from disk, the specified algorithm is used to solve the problem, and the result is compared to the sparse vector used to generate the problem. Two times are recorded: the total time, which includes the time taken to load the

problem from disk and compare the solution, and the solution time, which only includes the time taken running the solver algorithm. The details of the trial along with these two times is then saved to disk for later collation and processing. A summary of experiments is shown in Table 4.1

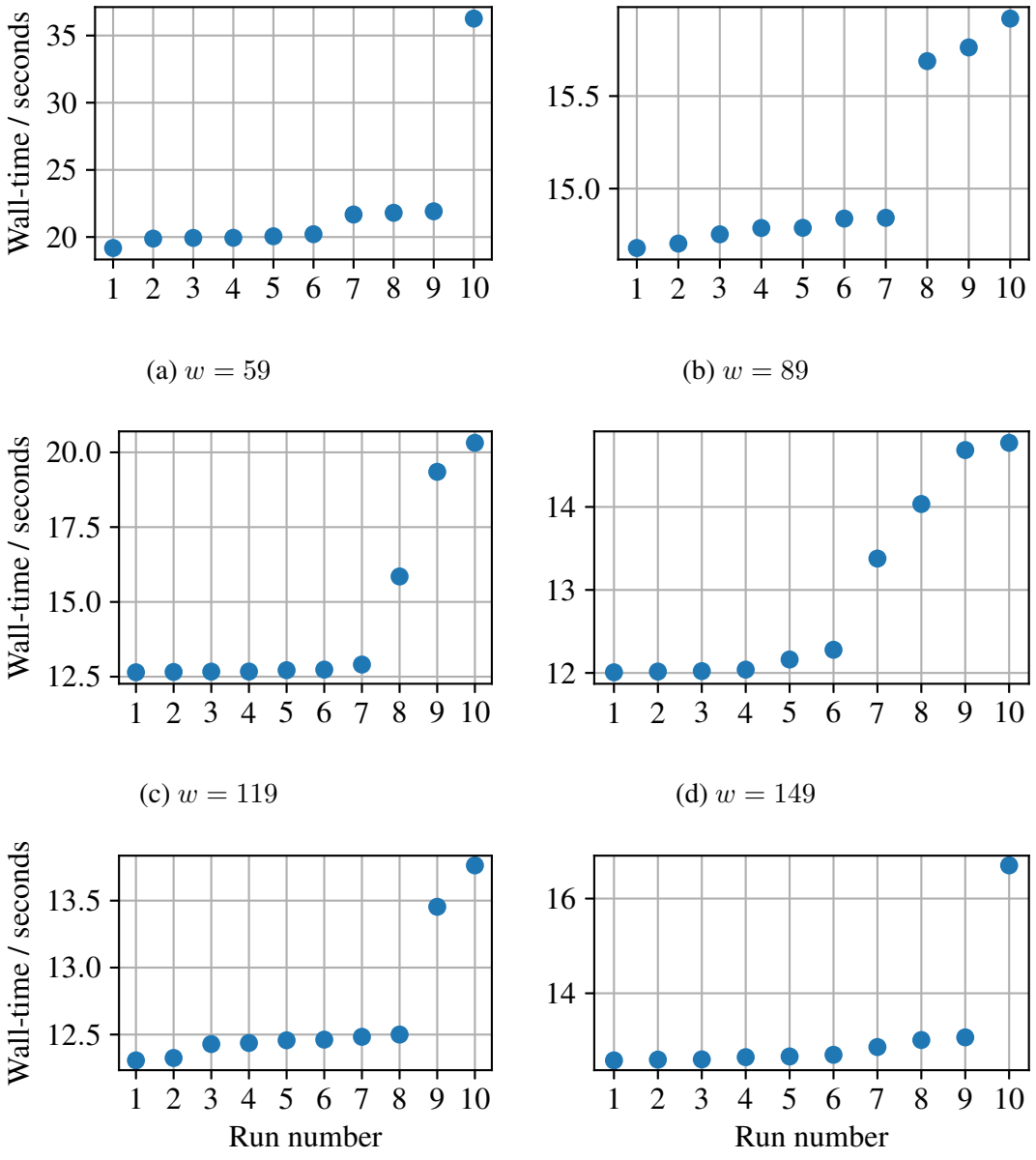
4.4 Interpretation of results

Since a computer (in the absence of a true random number generator) is a deterministic system, in an ideal world a program might always take the same amount of time to execute if performing the same operations on the same data. Indeed, for code running on a simple microcontroller with no operating system or interruptions this is the case; one can count the number of cycles required and divide by the clock frequency to find the execution time. However, this is not the case on the complicated multi-user operating systems generally used on high performance clusters (e.g., GNU/Linux in our case). Even though a cluster node may be nominally dedicated to running a particular program, system and house-keeping processes continue to run in the background. These awaken at apparently random times and cause a brief interruption, increasing the execution time of our program. A similar effect occurs from the fact that our experiments using MPI communicate over a network shared with many other users running other experiments. Load on the interconnect network will cause a slow-down in a similar way to the unexpected awakening of background processes.

All of these unexpected events are of no interest when attempting to compare the performance of algorithms. Ideally, all trials would be run on systems with no background or housekeeping processes and using an interconnect network devoid of other users or traffic. While this is not an option due to its impracticality, we can attempt to determine this ideal run-time by running multiple experiments. Consider the execution time in a particular trial as consisting of a constant (the ideal run-time without interruptions) plus a random variable representing time spent dealing with interruptions or waiting while the network is busy. The random variable takes values greater than or equal to zero. We observe that the minimum value of the random variable will approach zero as we increase the number of trials. Equivalently, if we take the minimum execution time over a number of trials this will approach the actual execution time of our program.

As an example of the distribution of execution time, Figure 4.1 shows the solution times for a range of trials using DistOMP to solve a large, sparse problem using various numbers of workers. In every case the results are clustered around a minimum execution time with a few outliers taking significantly longer. This data shows how using the minimum execution time over 10 runs minimises the effect of unexpected slowdowns and gives consistent results. In other words, if another 10 trials of each scenario were run, it appears unlikely that the minimum execution time of each scenario would be significantly affected.

Wall-time, in addition to being the actual result we record, is a useful metric because of its real-world implications: The execution wall-time tells us how long we have to wait for



(a) $w = 59$

(b) $w = 89$

(c) $w = 119$

(d) $w = 149$

(e) $w = 179$

(f) $w = 209$

Figure 4.1: A set of scatter plots demonstrating variability and distribution of execution time. A different number of workers was used for the trials in each plot. Within each plot, the same number of workers solved the same problem 10 times, yielding different execution times. In each case the runs are sorted by ascending solution time.

results after inputting data, or whether the solver could keep up with real-time data being fed to it. However, while more abstract, the metrics of speed-up and efficiency (introduced in Section 2.4.1) are useful for judging the performance of a parallel algorithm and comparing it to others. Therefore, I will present results for all three metrics: wall-time, to show the performance improvement in real-world terms; speed-up, to show more precisely how much performance has improved; and efficiency, to show how well the parallel algorithm utilises the available hardware and the impact of overheads and sequential sections.

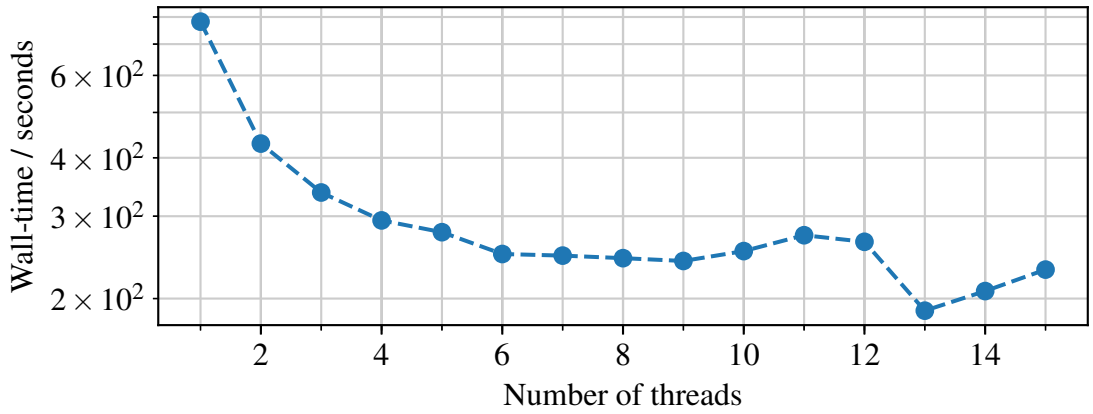
In general, most parallel algorithms (apart from embarrassingly parallel problems) can only utilise a certain number of parallel workers to gain additional speed-up. Beyond a point the wall-time stops decreasing and starts to increase again, and speed-up reduces from its peak at the optimal number of workers. This is because the time spent in parallelisable portions of the algorithm decreases with additional workers and so total elapsed time comes to be dominated by the serial portions which do not execute faster with the addition of workers. Furthermore, increasing numbers of compute workers tends to increase the overhead due to communication, which is why too many compute workers can actually harm performance compared to an optimal number. Knowing the maximum number of compute workers which can be harnessed without negatively affecting wall-time is a useful metric for characterising a parallel algorithm. It depends on the level of parallelisability of the problem along with the level of communication and coordination overheads.

4.5 Results

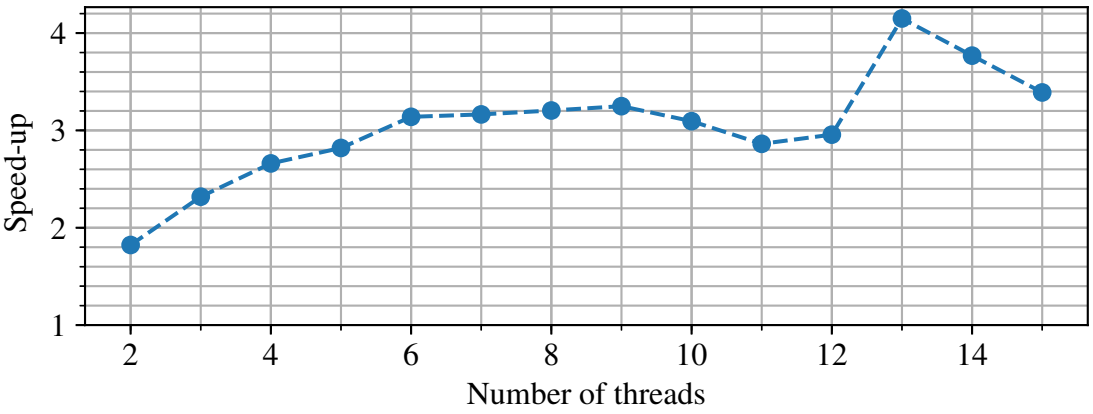
4.5.1 Multithreaded MP

To establish an initial baseline, my first experiments tested the simple MP algorithm using BLAS and OpenMP for multithreading. The implementation was tested using different numbers of threads on a single worker node. The nodes have 16 cores available, but a maximum of only 15 threads were used for testing in order to leave one core spare for background and housekeeping tasks. The results of these experiments are shown in Figure 4.2.

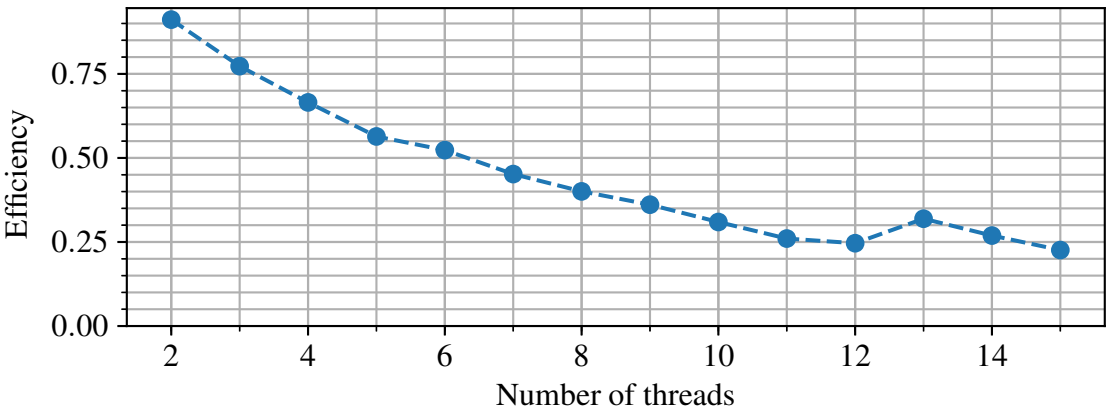
As previously motivated, each point on these charts represents the minimum wall-time over 10 experiments in order to minimise the effect of latency caused by background and housekeeping tasks. The plot of wall-time shows how introducing multithreading on a 16-core machine can reduce wall-time from 782 seconds to 188 seconds with very little effort on the part of the programmer. Extra speed-up is modest when introducing more than 6 threads, with the speed-up improving from 3.1 to a peak of 4.1 when moving from 6 threads to 13. The efficiency plot backs up the previous plots, showing how overheads come to dominate the compute resources as more threads are added. With 2 threads the efficiency is 91%, but at 15 threads the efficiency is only 23%, meaning almost 5 times more compute resource is being expended compared to using only a single thread. In summary, the addition of more threads is a trade-off between speed-up



(a) Wall-time against number of threads



(b) Speed-up against number of threads



(c) Efficiency against number of threads

Figure 4.2: Matching Pursuit with multi-threading ($N = 524288$, $n = 3277$, $S = 164$)

and efficiency, up to the point of maximum speed-up. Where efficiency is an important factor, for example in energy constrained portable computing, it may be desirable to use fewer threads at the expense of speed-up.

Up to 11 threads the wall-time and speed-up plots look as one might expect, with overheads and inefficiency causing speed-up to reach a peak and then begin to decline, with wall-time reaching a minimum and then slowly increasing after this point. The actual results show an unexpected sharp wall-time minimum at 13 threads, with 14 and 15 threads also unexpectedly improving upon the earlier minimum at 9 threads. The reason for this deviation is unknown, but could plausibly be caused by changes in the level of background activity taking place on the compute node used for experiments. Because of this unexpected trend in the results, it is difficult to say with certainty what the optimal number of threads is in this scenario. Since these results exist as a base-line comparison point and are not central to my experimental work, the cause was not further investigated.

4.5.2 DistMP

In order to characterise the performance of DistMP, I ran a number of experiments solving problems of different shapes and sizes using different numbers of workers, each time recording the elapsed wall-time in solving the problem. To investigate the effect of problem size and sparsity, I used three different problem sizes and two different sparsity levels. To compare the utilisation of multiple compute cores on a single node against MP, I carried out experiments with DistMP using a single worker and also using 15 workers on a single compute node. To explore the scaling effect of additional workers (and in order to find the optimum number of workers giving the greatest speed-up) I ran experiments using 15 workers-per-node on 2, 4, 6, 8, 10, 12, 14, and 16 nodes.

Figure 4.3 shows the wall-time elapsed across every combination of problem size, sparsity, and number of workers, allowing the effect of each independent variable to be compared. In the same manner, Figures 4.4 and 4.5 show the speed-up and efficiency respectively for the same variables. The speed-up and efficiency plots exclude the data-points with only one worker because speed-up and efficiency are calculated using this point as a baseline so its inclusion in a plot of the calculated value would be meaningless. We first consider the relationship between number of workers and wall-time. The clear trend from the plot of wall-time is that increasing the number of workers generally reduces wall-time. Looking closely, we see that this trend only holds up to some optimal number of workers for each scenario, beyond which the wall-time increases. This effect is more clear on the speed-up plots where every scenario has a clear peak in speed-up at the optimum number of workers, beyond which the speed-up reduces again. Next, we observe that in every case, increasing the problem size causes an increase in wall-time, as would be expected due to the extra time taken by every operation on larger vectors and matrices. Finally, by comparing Figures 4.3a and 4.3b we can see that the more dense problems have

a longer wall-time than sparse problems. This is as we would predict, because the increased number of rows in the dictionary means many operations in the algorithm involve larger vectors and so will take longer, and also because the decreased level of sparsity will generally cause the algorithm to require more iterations to converge.

We have considered how each of the independent variables (number of workers, problem size, and problem density) impact the wall-time elapsed in solving problems using DistMP. Next, we consider the interaction between the effects of the independent variables, which gives further insight into the performance characteristics of the algorithm. We first focus on the combined effect of number of workers and problem size. Recall how for each value of problem size and density, there is an optimal number of workers beyond which wall-time is no longer decreased with the addition of further workers. We might predict that with increased problem size the optimal number of workers increases: This is because many of the overheads are fixed or depend on the number of workers, but the proportion of the problem which is susceptible to parallelisation scales with the size of the problem. This means for larger problems overheads are less significant and so there is more scope for speed-up due to parallelisation. This effect is just about visible in Figure 4.3 but the troughs in wall-time are slight and so difficult to compare. Figure 4.4, however, makes this result very clear: For the sparse experiments, the small, medium and large scenarios have optimum numbers of workers of 90, 150, and 180, giving speed-up values of 24, 36, and 50 respectively. The effect is less pronounced for the more dense problems, but the same trend is still visible. This all supports the prediction that larger problems are more susceptible to parallelisation than smaller problems, allowing the exploitation of a greater number of compute workers to achieve a greater maximum speed-up.

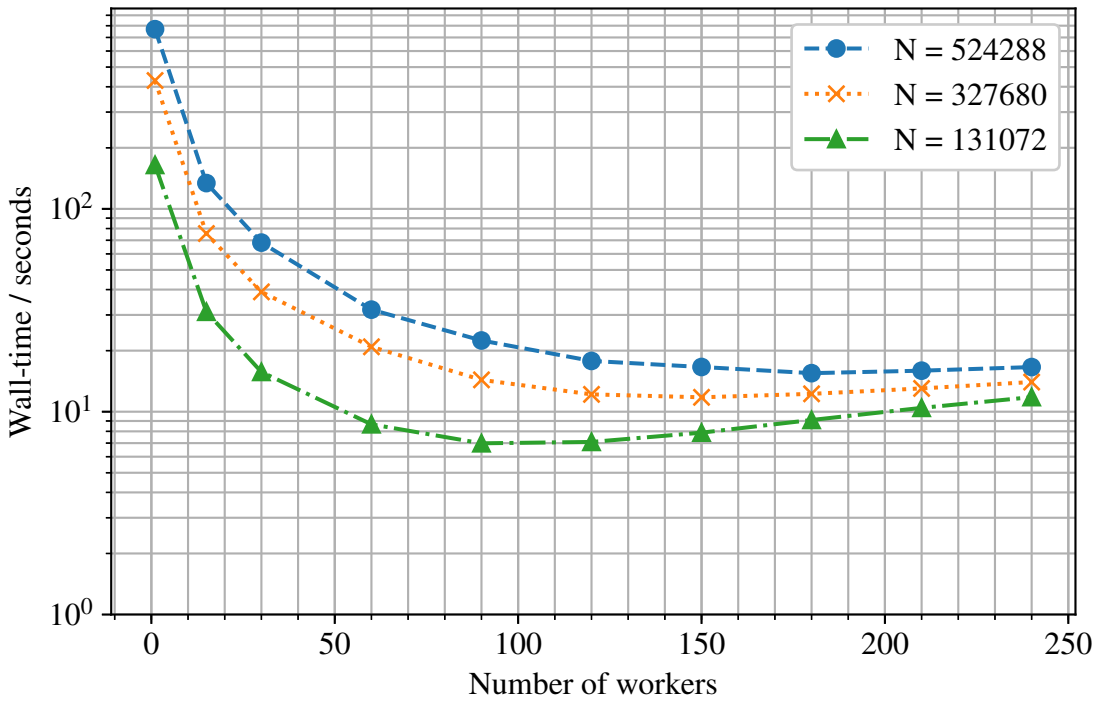
In the same manner, we can consider the combined effect of number of workers and problem density on solution wall-time and speed-up. The trend is not particularly clear in Figure 4.3, but once again Figure 4.4 shows the trend clearly: denser problems allow the useful exploitation of a larger number of compute workers than sparse problems, and while denser problems require a higher elapsed wall-time to solve, they benefit more from parallelisation and deliver a higher speed-up than sparse problems. For example, while the largest sparse problem had its greatest speed-up when using 180 workers, the dense problem had greatest speed-up when using 210 workers. The greatest speed-up observed on a sparse problem was a factor of 50 when using 180 workers on the large problem, while the equivalent dense problem delivered a speed-up of 76 using 210 workers. The same effect applies to the smaller problems tested.

Finally, we look at Figure 4.5, a plot of the efficiency calculated for each experiment. In a sense, efficiency is a direct measure of the overhead due to parallelisation: An efficiency of 1.0 would represent a perfect parallelisation with no overheads, and $(1-\text{efficiency})$ is the proportion of total compute effort expended on overheads. The plots show a different aspect to the same picture described so far. The first data-point is with 15 workers on a single node, by which point the efficiency has already dropped to around 30-40% in all cases. The efficiency then decreases further with the addition of more workers. Larger problems show higher efficiencies than smaller

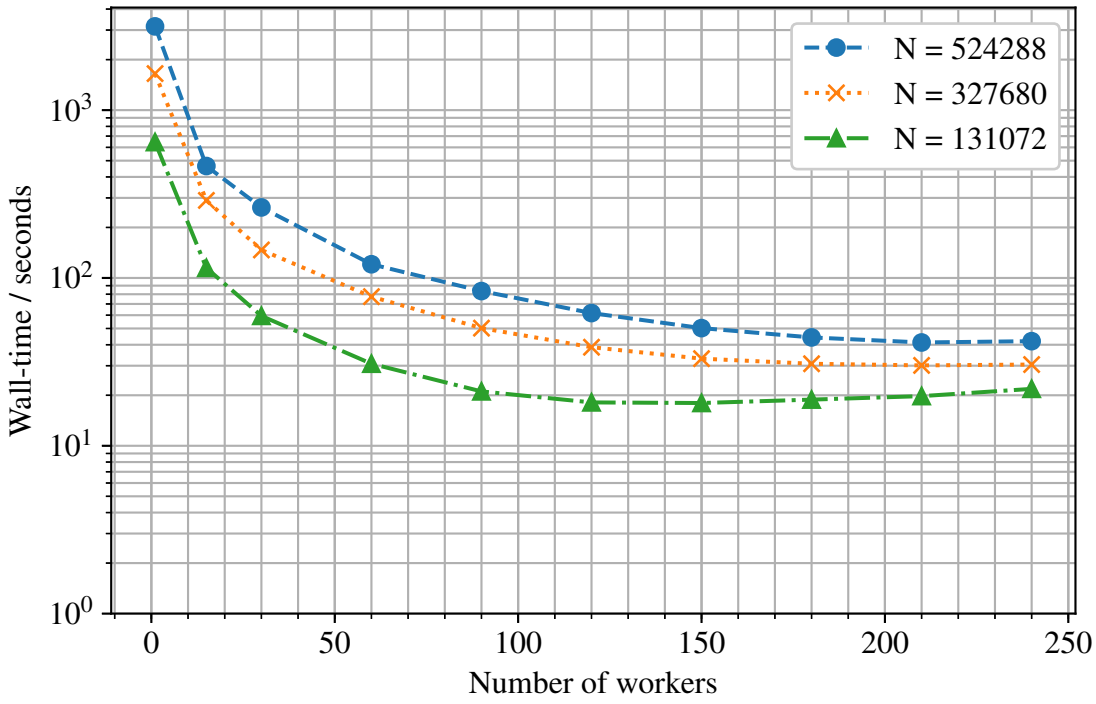
problems, and dense problems show higher efficiencies than the sparse problems, reinforcing the explanation that overheads are proportionately less significant when the problems involve greater quantities of data. Another clear property is that the efficiency of smaller problems drops off faster with increasing numbers of workers than that of larger problems: larger problems are more amenable to greater levels of parallelisation than smaller problems.

Having established the performance characteristics of DistMP from the results presented, we can now compare it against the multithreaded implementation of MP which we characterised previously. First, we compare the wall-time elapsed by both algorithms when using only a single worker or thread: MP takes 782 seconds for the large, sparse, problem (Figure 4.2a), while DistMP took 769 seconds (Figure 4.3a). This indicates that the algorithms are comparable when operating on the same problem and before any parallelism is introduced. The next point we can directly compare is when 15 workers or threads are applied to the same problem: this comparison represents how well the two algorithms can harness the compute power of a single compute node. Our initial expectation might be that the multithreaded algorithm should more efficiently utilise the resources of a single compute node, since threads have the ability to share data with very little communication overhead and the complexity of message passing is avoided. Indeed, with an algorithm carefully crafted to efficiently solve this problem in a multithreaded manner, this might be the case, but in my experiments DistMP was actually faster when utilising 15 workers than MP with 15 threads: DistMP took 134 seconds to solve the large sparse problem, while MP took 231 seconds (MP's fastest time was 188 seconds with 13 threads). Looking next at the plots of speed-up (Figures 4.2b and 4.4a) we see that the multithreaded MP implementation reached a peak speed-up of 4.1 utilising 13 threads. On the same problem, DistMP delivered a speed-up of 11 when utilising 15 workers, and did not reach its peak speed-up until utilising 180 workers. Considering efficiency, Figures 4.2c and 4.5 show how for both multithreaded MP and DistMP, adding large numbers of workers poses a trade-off between speed-up and efficiency. However, when solving large problems with DistMP, the speed-up can be as much as a factor of 50 to 80 with an efficiency of 20% to 40%, while the multithreaded MP implementation only delivered a speed-up of 4 with a similar loss of efficiency to DistMP.

This comparison is not meant to imply that a careful multithreaded implementation could not beat DistMP when both are utilising 15 threads on a single node, but shows how much better DistMP performs than an effortless multithreaded implementation. The reason for this is that even on a single compute node, DistMP treats each core totally separately and so ensures data locality. This means cores are relatively free to work on their own portion of the data without having to share the same areas of memory and incur the performance penalties of repeated cache invalidation.

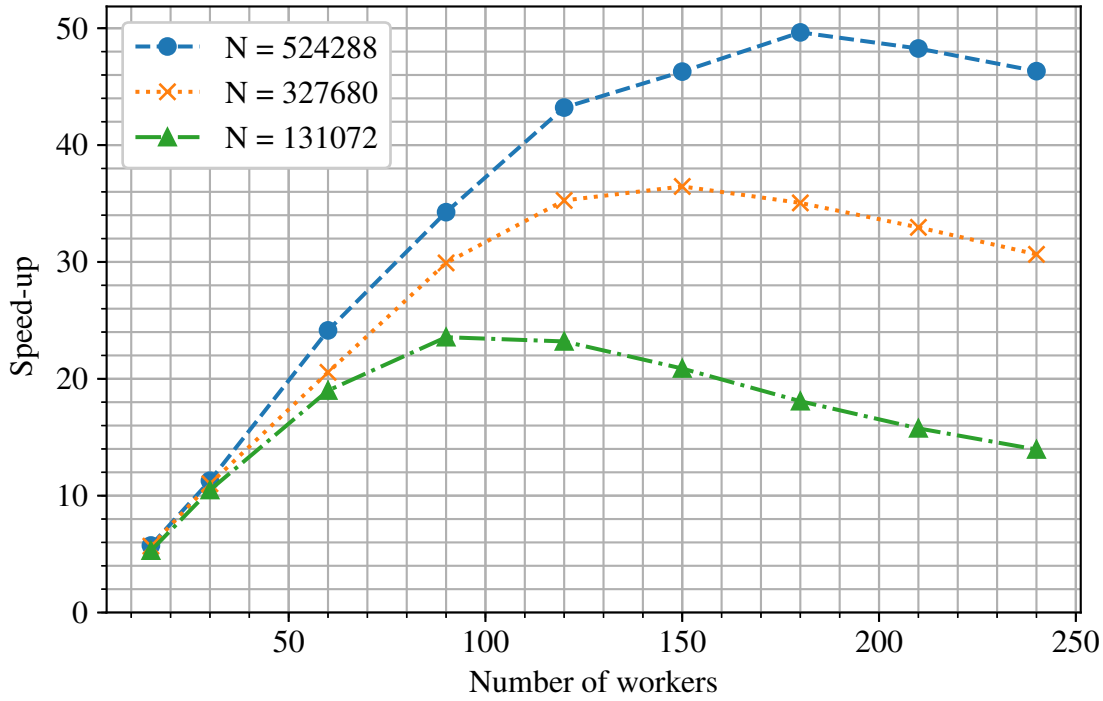


(a) Sparse problem ($n = 3277$, $S = 164$)

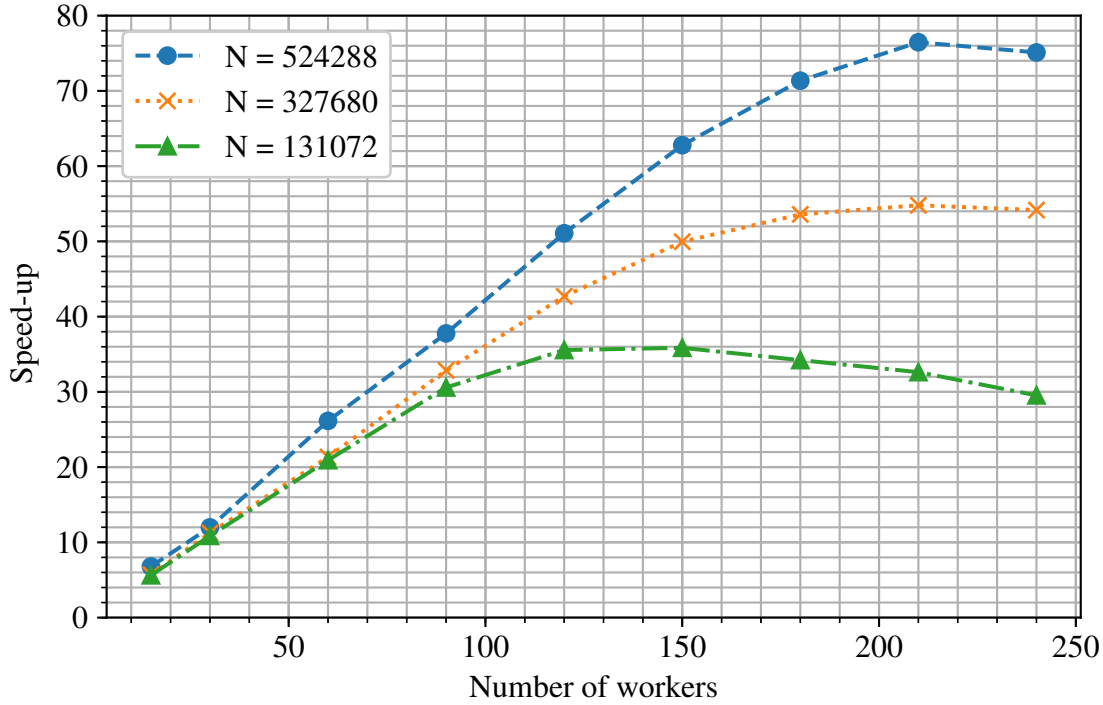


(b) Dense problem ($n = 6554$, $S = 328$)

Figure 4.3: Wall-time against number of workers for DistMP

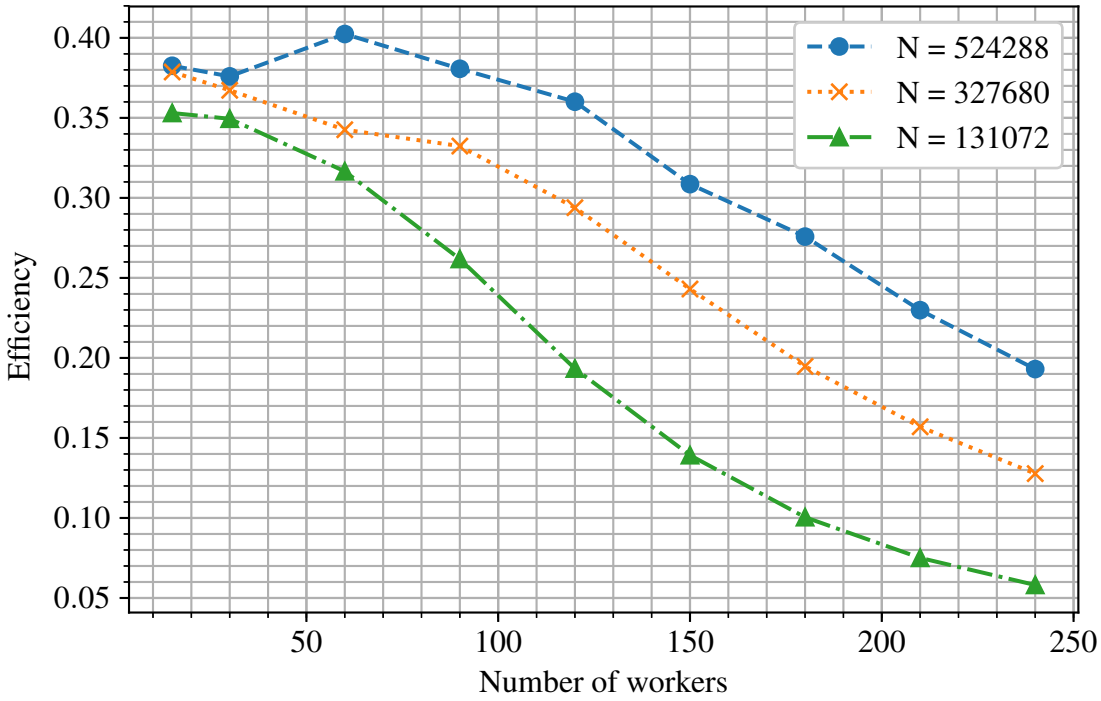


(a) Sparse problem ($n = 3277$, $S = 164$)

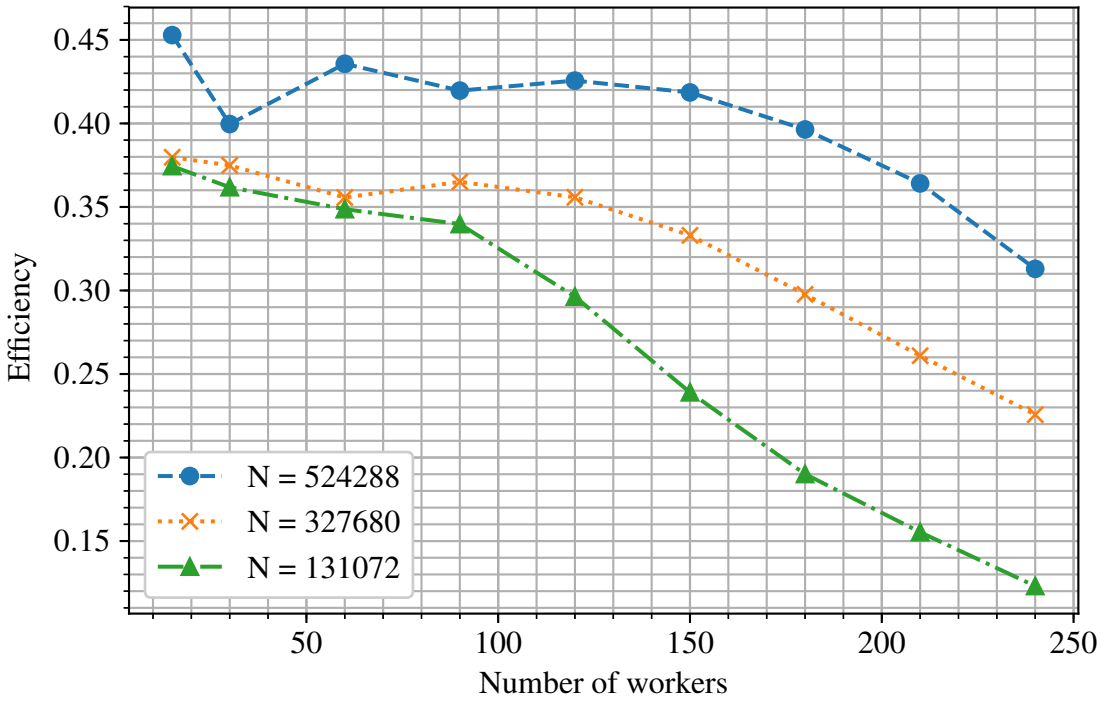


(b) Dense problem ($n = 6554$, $S = 328$)

Figure 4.4: Speed-up against number of workers for DistMP



(a) Sparse problem ($n = 3277$, $S = 164$)



(b) Dense problem ($n = 6554$, $S = 328$)

Figure 4.5: Efficiency against number of workers for DistMP

4.5.3 Multithreaded OMP

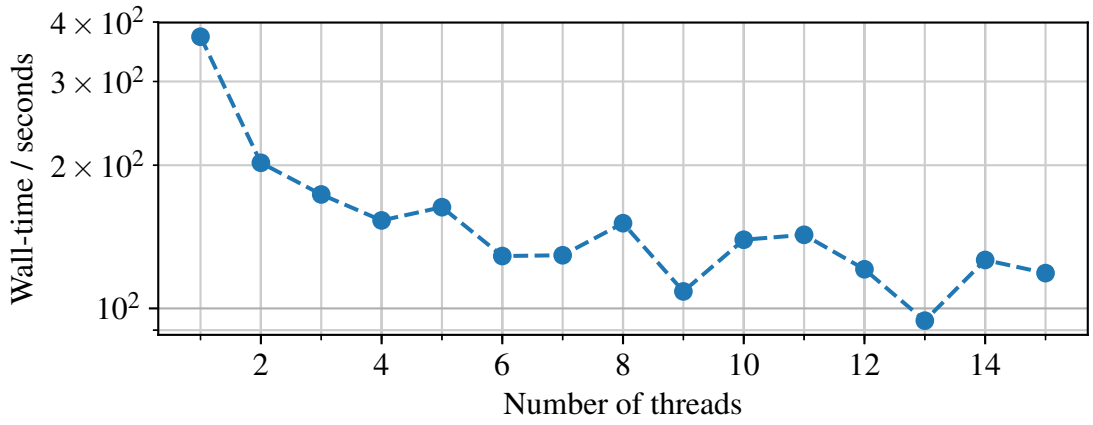
The next algorithm we turn our attention to is OMP. As with MP, a simple multithreaded OMP implementation was created to use as a baseline, utilising the OpenMP-based multithreading built-in to OpenBLAS. Once again, experiments were conducted using 1 to 15 threads on a single node. For each test scenario, the minimum recorded time was used out of 10 trials. The results are presented in Figure 4.6 including wall-time and the derived speed-up and efficiency metrics.

Compared to MP, the results show significant variation even after the filtering effect of taking the minimum time for each scenario. Despite this, the same overall trend is visible, where adding threads to the solver generally reduces wall-time, but with decreasing benefit as more threads are added. The majority of the benefit is realised with 6 threads providing a decrease in wall-time from 373 seconds to 129 seconds, a speed-up of a factor of 2.9 with no effort spent on parallelising the implementation. The maximum speed-up observed with this algorithm was 4.0 when utilising 13 threads, corresponding to a reduction of wall-time to 94 seconds. Figure 4.6c shows how the algorithm demonstrates reasonable efficiency with a few threads, dropping below 50% with the usage of 5 threads. The lowest efficiency observed was 21% when using 15 threads out of the 16 available.

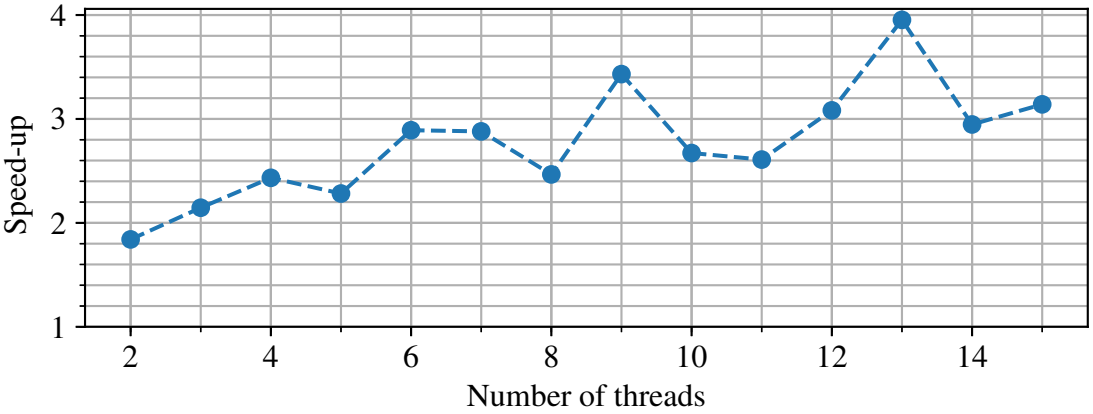
Despite the low efficiency when using 15 threads, it appears as though the underlying reduction in wall-time (increase in speed-up) had not reached its trough (peak) in these experiments, although it is difficult to say for sure given the somewhat erratic results. Further experiments to find this inflection point were not possible because only 16 threads were available on the worker nodes used for these experiments.

4.5.4 DistOMP

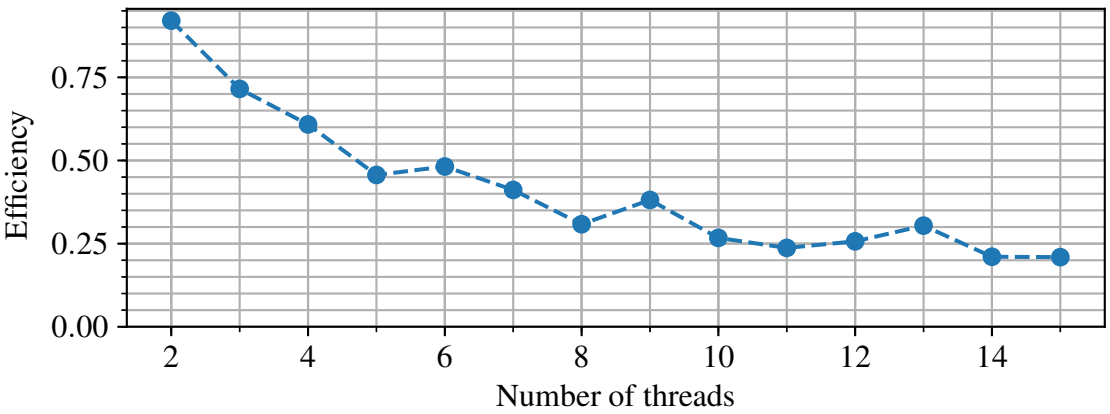
I ran experiments to determine the performance characteristics of DistOMP in a similar manner to the earlier DistMP experiments: problem size, problem density, and number of workers were all varied while recording the wall-time taken to solve problems. The same three problem sizes and two problem densities were used as previously. There was a slight difference in the number of workers applied to the problem due to DistOMP requiring a dedicated R-worker separate to the QA-workers. As with previous experiments, it was desirable to not use more than 15 of the 16 compute cores available on each node in order to minimise the effect of background processing. Using a single node, two scenarios were investigated: one QA-worker and one R-worker (two threads total), and 14 QA-workers with one R-worker (15 threads total). Then, using 2, 4, 6, 8, 10, 12, 14, and 16 nodes respectively, experiments were carried out using 29, 59, 89, 119, 149, 179, 209, and 239 QA-workers respectively. Thus, in each case the total number of threads utilised was the same as the equivalent DistMP experiment, but the number of QA-workers was one fewer than the number of threads. For the plots and derived calculations presented through the rest of this section, I refer to the number of QA-workers, as this is the variable I was directly



(a) Wall-time against number of threads



(b) Speed-up against number of threads



(c) Efficiency against number of threads

Figure 4.6: Orthogonal Matching Pursuit with multi-threading ($N = 524288$, $n = 3277$, $S = 164$)

altering. This means there is an additional overhead of one thread associated with DistOMP which is not reflected in these charts and calculations. Nevertheless, this does not invalidate the trends and conclusions derived from the data, not least because the extra thread is negligible when hundreds of threads are in use in total. Furthermore, for most problems where the $\mathbf{R}$ matrix is not particularly large, it would be possible for the R-worker to share resources with a QA-worker with minimal performance impact. For the sake of simplicity this was not investigated in these experiments, but would be worthwhile if applying DistOMP across a relatively small number of workers where a one thread overhead would cause significant detriment.

For each scenario (combination of problem size, problem density, and number of workers) 10 trials were carried out and the minimum wall-time in each case is presented in Figure 4.7. From this data, the speed-up and efficiency at each scenario were derived and are shown in Figures 4.8 and 4.9. I will first discuss the effect of each independent variable alone on the three metrics presented, after which I will consider the interactions between the three independent variables.

First, we consider the effect of the number of workers on the wall-time elapsed in solving a problem, as shown in Figure 4.7. For every combination of problem size and sparsity the trend is the same: wall-time reduces steeply at first, with the effect levelling off and eventually reaching a trough with a large number of workers. Eventually, after adding enough workers to the problem, the wall-time begins to increase slightly. This is exactly as we would predict: At first, with a small number of workers, a large proportion of the computation time is ripe for parallelisation. After significant gains are made by the introduction of a moderate number of workers, the time spent in parallelisable portions of the program has been greatly reduced, so further accelerating these portions of the problem provides minimal reduction in total wall-time. Eventually these gains are entirely counteracted by the additional overhead of a large number of workers, so the curve reverses and wall-time begins to increase.

Figure 4.8 clearly shows the same trend: For small numbers of workers, speed-up increases quickly with additional workers. As further workers are added, speed-up reaches a peak and finally decreases gradually past the peak. Figure 4.8a in particular shows a linear reduction in speed-up as further workers are added. The exception is the case of $N = 524,288$ and $n = 6,554$, where the peak speed-up is not reached with the maximum 240 workers, however it is likely that the same trend would be observed if more workers were available to be added to the problem. By looking at the peak speed-up for each problem we can find out the optimal number of workers in order to minimise wall-time. The optimal number of workers is higher for problems which are more dense and also for problems which are larger.

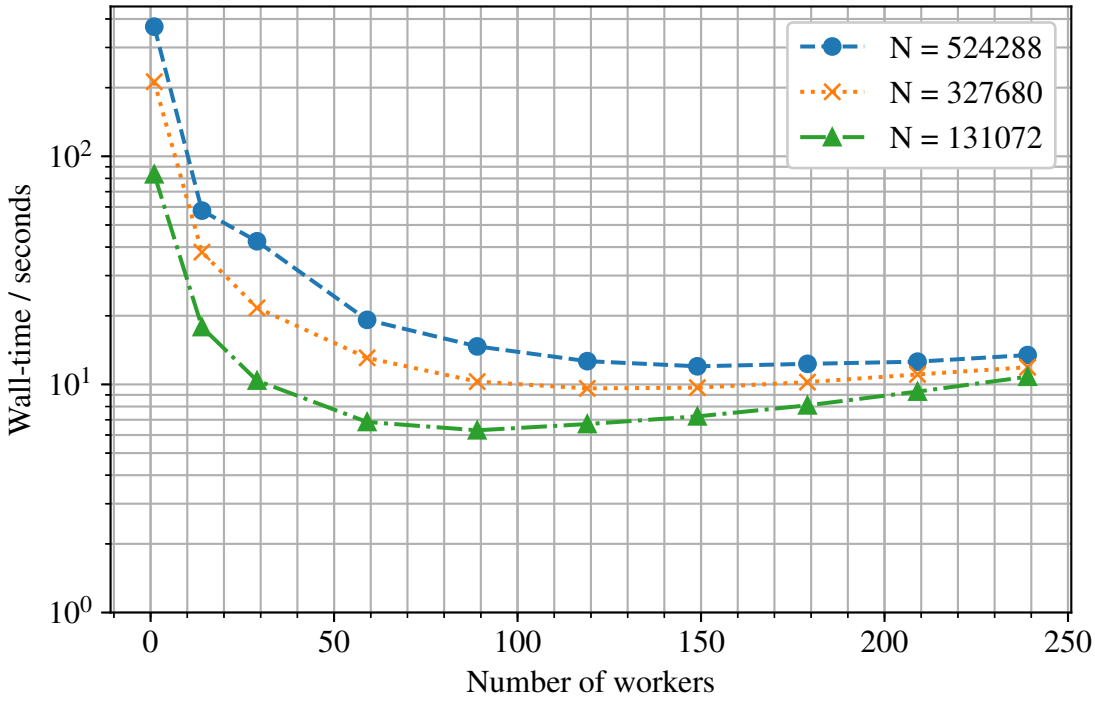
Finally, we look at Figure 4.9, the plot of calculated efficiency. The overall trend is that efficiency falls with increasing numbers of workers, quickly at first but with decreasing rate. In all the scenarios, the efficiency has dropped to around 40% with the move from 1 QA-worker to 14. After this, there is generally around a further 5% fall moving from 14 to 89 QA-workers, with more significant falls in efficiency thereafter.

I will now briefly consider the isolated effects of each of problem size and problem sparsity on the results. In all cases, increased problem size causes an increase in wall-time, due to the increased cost associated with each vector or matrix operation. However, increased problem size also came with significant increases in speed-up. For the same reasons as with DistMP, this will be because increased problem size increases the amount of time spent in parallelisable portions of the program and reduces the relative effect of many overheads. For the same reasons, increasing problem size causes an increase in efficiency in all cases. Changing problem density has similar effects to problem size, and for similar reasons: an increase in problem density causes increased wall-time in all cases, along with a significant improvement in speed-up and efficiency.

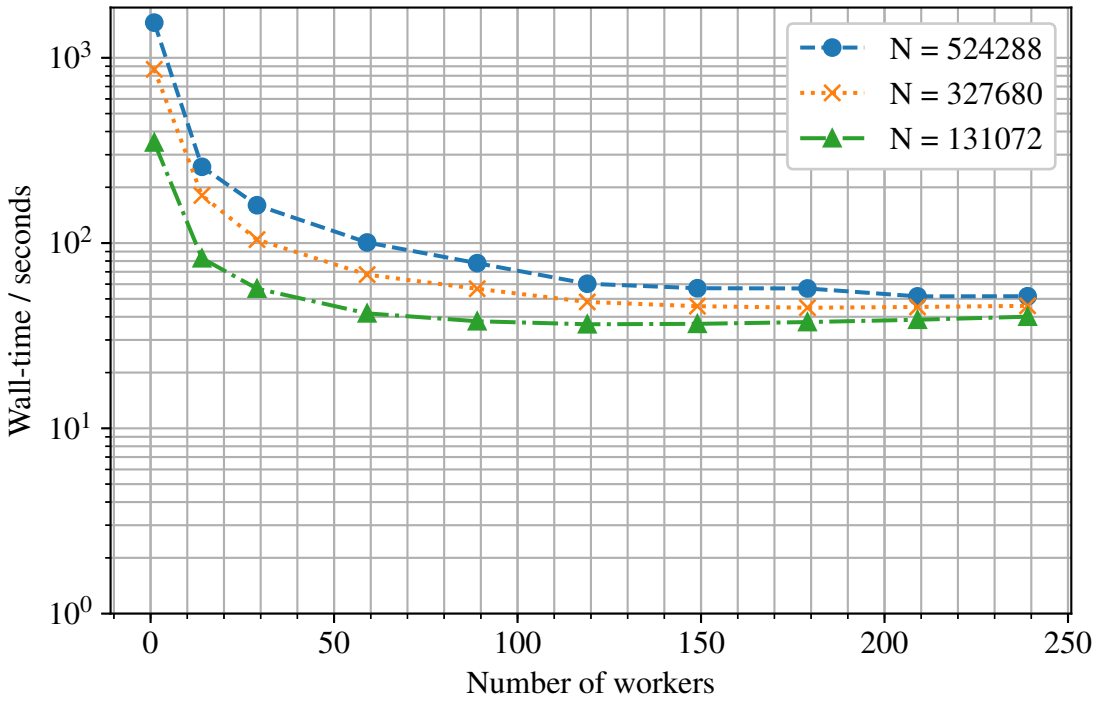
Next, I consider the interactions between the variables: How do problem size and problem density affect the shape of the curves in Figure 4.7? The effect is most clearly seen on the plots of speed-up shown in Figure 4.8, in particular by focusing on the point where speed-up peaks. The trend is clear: increasing either problem size or problem density shifts this peak to the right, allowing more workers to be added to the problem in exchange for greater speed-up. In other words, the larger and denser a problem is, the greater the optimal number of workers to task with solving it in order to achieve the greatest speed-up, and the larger this maximum speed-up will be. This is not to imply that increasing problem size or density will result in it being solved faster (Figure 4.7 shows this is not the case), but that larger or more dense problems have greater potential for benefit from parallelisation, and allow more workers to be effectively used. Of course, this consideration only applies in the case where more workers are available than the optimal number, which is by no means a given.

We can compare DistOMP to the simple multithreaded OMP implementation by directly looking at wall-time for equivalent scenarios. We will only consider the large, sparse problem ($N = 524288$, $n = 3277$) since this is the only one OMP was tested with. OMP using a single thread solved the problem in a minimum of 373 seconds (Figure 4.6a). DistOMP using a single QA-worker (and single R-worker on the same node) solved the problem in a minimum of 370 seconds (Figure 4.7a). These results are very similar, confirming that the algorithms work similarly before parallelisation is introduced. The lowest wall-time observed with OMP was 94 seconds, when 13 threads were used. DistOMP, when using 14 QA-workers plus one R-worker, solved the same problem in just 58 seconds. These figures correspond to a maximum speed-up of 4.0 for Multithreaded OMP (Figure 4.6b) or 6.4 for DistOMP (Figure 4.8a) when utilising up to 15 threads in total. This comparison shows how DistOMP is able to more effectively utilise the threads on a single worker node than a simple multithreaded implementation, in addition to continuing to reduce wall-time significantly with the addition of as many as hundreds of threads across a number of nodes.

We can also briefly compare the characteristics of MP and DistMP to those of OMP and DistOMP respectively. At first glance, we can see that OMP generally has a lower elapsed wall-time than MP (Figures 4.2a and 4.6a) for the same problem: this is because the orthogonalisation

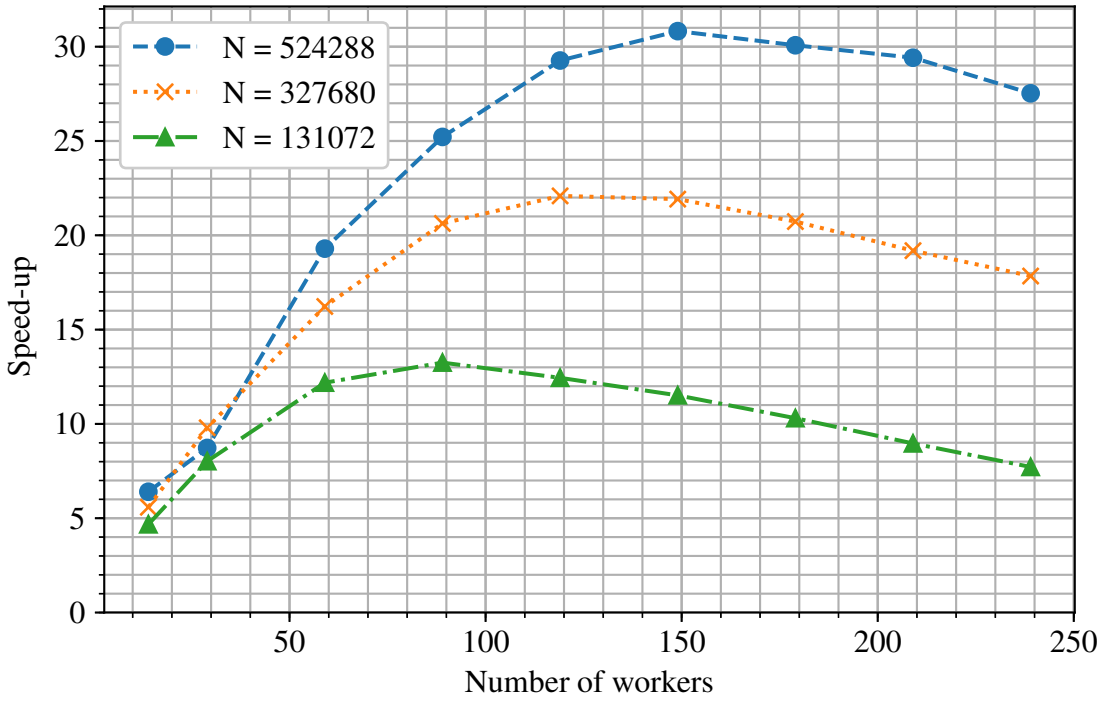


(a) Sparse problem ($n = 3277$, $S = 164$)

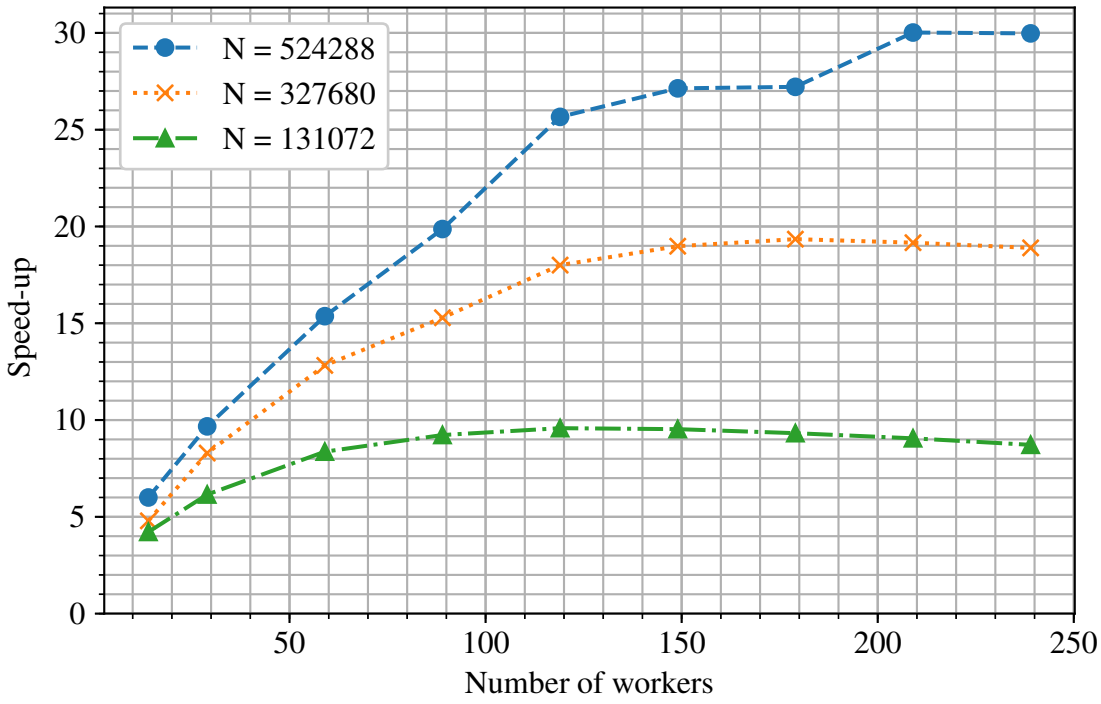


(b) Dense problem ($n = 6554$, $S = 328$)

Figure 4.7: Wall-time against number of workers for DistOMP

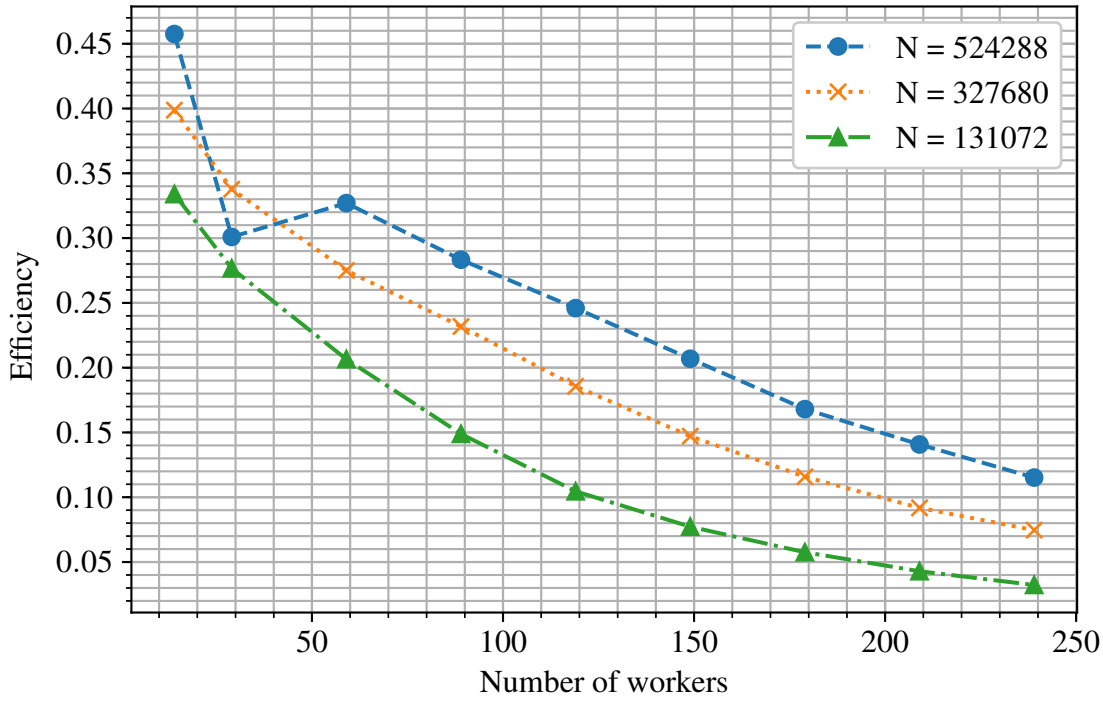


(a) Sparse problem ($n = 3277$, $S = 164$)

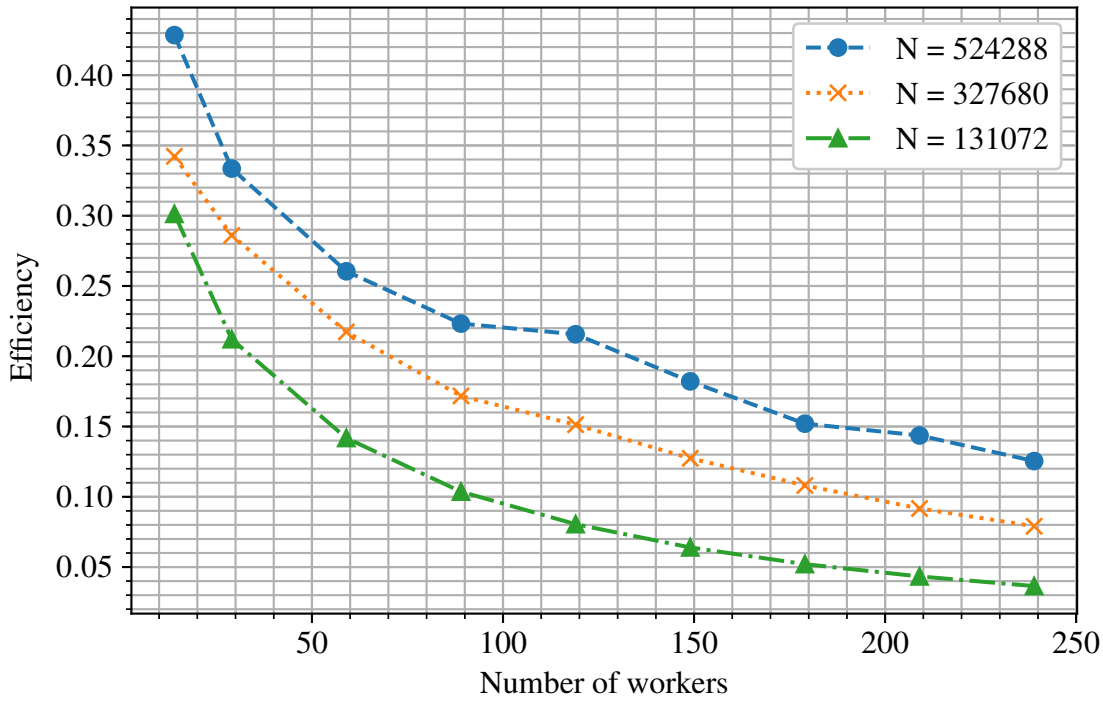


(b) Dense problem ($n = 6554$, $S = 328$)

Figure 4.8: Speed-up against number of workers for DistOMP



(a) Sparse problem ($n = 3277$, $S = 164$)



(b) Dense problem ($n = 6554$, $S = 328$)

Figure 4.9: Efficiency against number of workers for DistOMP

at each iteration means that fewer iterations are required. They also appear to offer similar speed-up from multithreading, with both reaching a peak speed-up of a factor of around 4 when utilising 13 threads (Figures 4.2b and 4.6b). Efficiency plots are also similar (Figures 4.2c and 4.6c). Differences between DistMP and DistOMP are, however, more pronounced. On the sparse problems, DistOMP ran with a slightly smaller elapsed wall-time than DistMP (Figures 4.3a and 4.7a). With a small number of workers, DistOMP was also faster for dense problems, but DistMP overtakes when it comes to solving dense problems with a large number of workers (Figures 4.3b and 4.7b). The reason for this is apparent from the plots of speed-up (Figures 4.4 and 4.8): DistMP shows much higher speed-ups than DistOMP for large numbers of workers. For dense problems, the peak speed-up of DistMP is over double that of DistOMP. This means that while OMP is generally faster than MP, with large numbers of workers DistMP may give a solution faster than DistOMP. DistMP probably shows greater speed-up than DistOMP because the latter requires more communication (leading to greater overheads) and has a significant serial portion (back-substitution carried out by the R-worker).

4.5.5 Large demonstration problem

In order to show the scale of problem which can be quickly solved using my techniques, a single large problem was chosen as a demonstration. The scenario was based on a hypothetical image compressively sensed using an unstructured dictionary obtained using dictionary learning. A reasonable size image for experimentation might have a resolution of 500×480, or about $240,000$ pixels. A dictionary which was over-complete by a factor of 10 would have $2,400,000$ atoms. At a 1% sparsity level, this image would have $2,400$ sparse coefficients. Using a 10% sampling ratio when sampling the image would give us $24,000$ samples. From this, we have our parameters for this scenario: $N = 2,400,000$, $n = 24,000$, $S = 2,400$. Using 8-byte doubles to store each element, the effective sampling matrix consumes 429GB of memory. A maximum of 256 threads across 16 compute nodes were available for this experiment, though only 240 threads were allocated to workers in order to leave spare capacity for background or housekeeping processes.

To simplify the experiment, synthetic data was used within the above parameters, as was done for the previous, smaller, experiments. The sparse signal was generated by producing a vector of zeros except for $2,400$ pseudo-randomly selected elements. The dictionary was then created pre-segmented for the 240 workers by generating each segment as a $24,000 \times 10,000$-element dictionary of IID normally distributed random numbers. Finally, the samples were calculated by multiplying each segment of the dictionary by the sparse vector and accumulating the summation of the result. The sparse vector, dictionary portions, and samples, were all saved to a scratch storage network share accessible by the worker nodes.

The problem was solved using DistOMP, using 240 QA-workers along with the single R-worker (Since the QA-worker and R-worker do not carry out computation at the same time, this

should equate to not more than 240 threads of computation). The elapsed time of solution was recorded for 10 trials, and the results are shown in Figure 4.10. For broader context, this chart shows the time spent directly computing the solution (Solution time) along with the total time, which includes initialisation, loading the problem data from disk, and verifying the solution. The time for each trial is shown so that the variability can be seen.

The minimum time taken for just the solution was 62 minutes, an impressive result for solving a problem of a scale previously considered intractable (Candès and Romberg [12] describe solution of a problem with $N = 1,000,000$ and $n = 25,000$ as impractical with an unstructured measurement matrix). Even the process of loading the problems and verifying the result only added an extra of 2 minutes on average. There was some variability in the results, but the slowest trial of the 10 took 72 minutes for the solution, or 74 minutes in total.

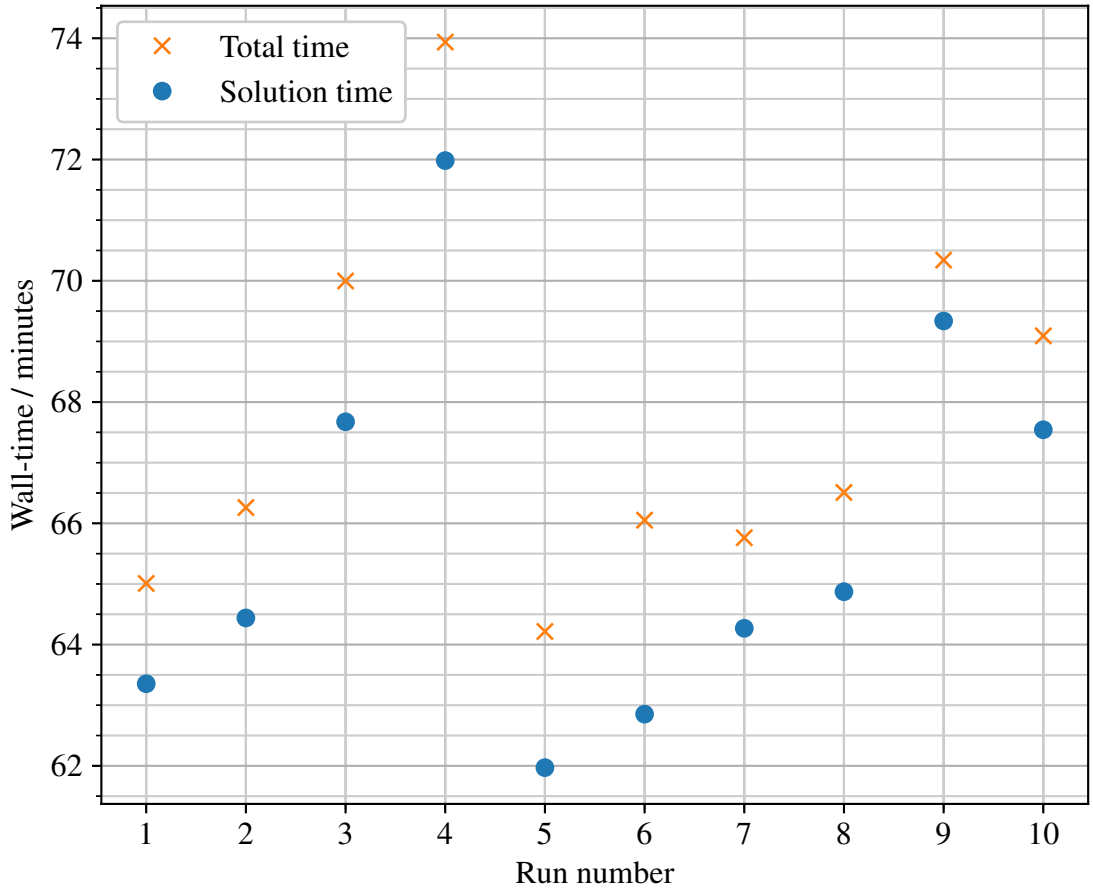


Figure 4.10: Time taken to solve a large problem with DistOMP ($N = 2,400,000$, $n = 24,000$, $S = 2,400$).

4.6 Comparison with Amdahl's Law

In this section, I will draw comparisons between the results presented and predictions made by Amdahl's Law [73] which I introduced in Section 2.4.2. Amdahl's Law is derived directly from the core concept of parallelisation, where part of a program experiences speed-up in proportion with the number of workers and the remainder runs serially and experiences no speed-up. Because of this, it can be a useful comparison point for a parallel algorithm to judge the performance of the algorithm and make predictions about the effectiveness of further parallelisation effort or the use of a greater number of workers.

4.6.1 DistMP

Figure 4.11 compares the results recorded for DistMP against predictions made using Amdahl's Law. Predictions are plotted for a range of different parallel proportions p in order to illustrate how the shape of the curve evolves. Values of p were chosen empirically to match the speed-up ratios observed in experimental results. Charts are plotted for speed-up and efficiency; no scaling or estimation of the specific costs involved is required because both of these metrics are normalised against the single-worker result. The predictions using Amdahl's law assume that the parallelisable proportion of the program experiences perfect speed-up with any number of workers. This assumption is justified by DistMP's theoretically ideal parallelisation of the correlation step which dominates the cost of each iteration. The limitations to this assumption will be discussed in more detail below.

First, we compare the speed-up results and predictions in Figure 4.11a. Speed-up predictions made using Amdahl's Law can be separated into several regimes: In the first, with small numbers of workers, speed-up is close to the number of workers and so increases linearly with increased numbers of workers. With larger numbers of workers the time spent in the serial portion of the program becomes more significant and we enter the second regime, where speed-up increases sub-linearly with addition of workers. As we add further workers we eventually enter the third regime where the speed-up curve approaches a horizontal line and addition of further workers has negligible effect.

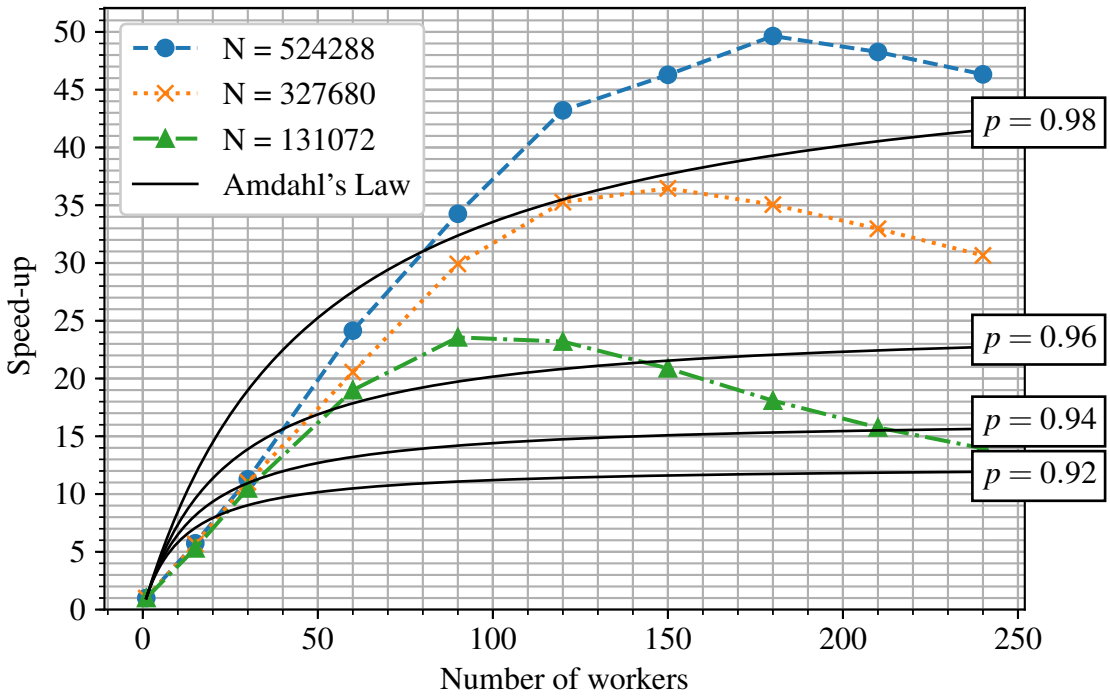
Both the DistMP results and Amdahl prediction show an initial linear increase in speed-up. This is the region of operation where the serial portion of the program takes a negligible amount of time. Amdahl's Law predicts a linear speed-up increase with a gradient of 1, however the linear speed-up increase of DistMP shows around half this gradient. This implies that each additional worker is only adding half of the benefit we would predict. This effect is apparent in Figure 4.11b: Where Amdahl's Law would predict high efficiency when the serial fraction of a program is negligible, DistMP shows efficiency below 40% for small numbers of workers. The initial linear shape of the speed-up curve suggests that inefficiency is not due to fixed overheads or the algorithm's serial fraction. One possible cause of this effect would be if the implementation was bottlenecked by memory bandwidth rather than processor capacity. To find

out whether this is likely, we can estimate the memory bandwidth required by the algorithm and the bandwidth available on each node.

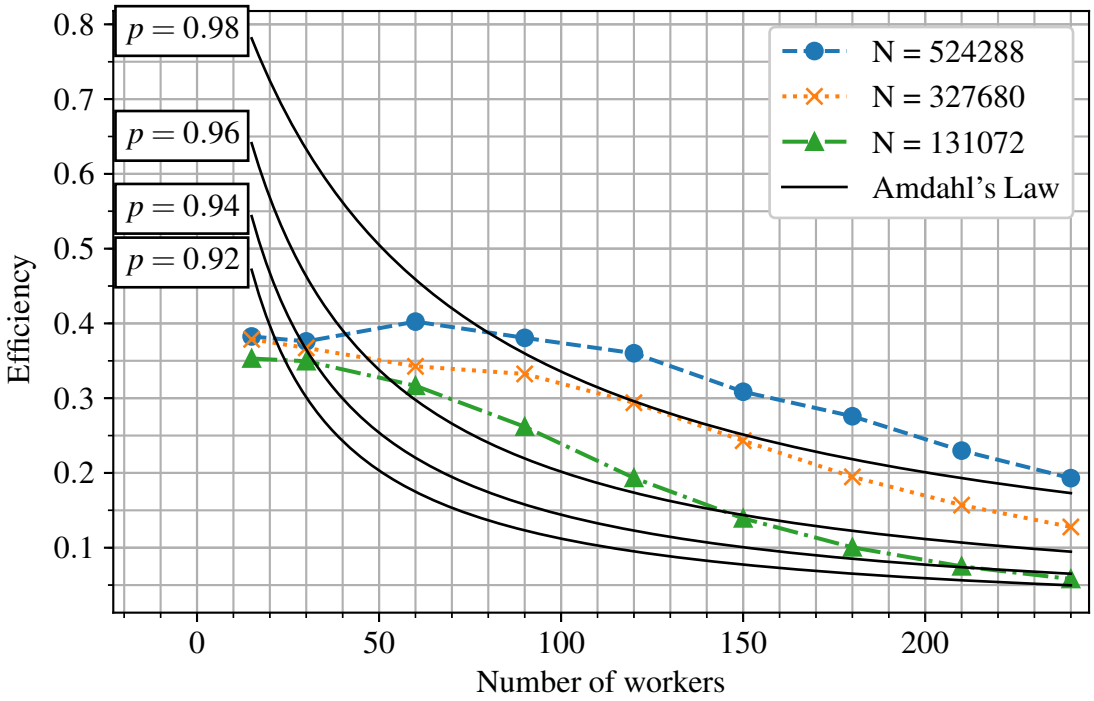
DistMP with $N = 524288$ and $n = 3277$ with 15 workers completed 637 iterations in 134 seconds, or 4.75 iterations per second. In each iteration, the memory access will be dominated by reading the whole of $\mathbf{A}$ with a size of $8Nn = 1.37 \times 10^{10}$ bytes (each dictionary element is stored as an 8-byte double-precision value). This results in a total memory bandwidth usage of 65.3 GB/s. The nodes used for these experiments had 8 channels of DDR3 memory running at 1600 MT/s; with 64 bits per transfer this results in a total theoretical bandwidth of 102.4 GB/s. It is plausible that memory throughput of 65.3 GB/s bottlenecked further performance increase since the theoretical bandwidth may only be achieved under ideal circumstances and the estimated throughput includes time spent outside of memory-intensive sections. If, on a single node, execution of workers on more than about half of the cores resulted in a memory bandwidth bottleneck then this would limit efficiency as only around half of the workers per node would be fully utilised. This hypothesis could be verified by only allocating workers to half of the processor cores on every node and testing whether efficiency significantly improves with small numbers of workers. Experiments could also be carried out with fewer than 15 workers on a single node to confirm whether a memory bottleneck appears and with what number of workers.

The second clear difference between the DistMP results and Amdahl's prediction is when large numbers of workers are used. The speed-up of DistMP actually decreases, which Amdahl's Law does not predict. For very large numbers of workers where the parallelisable fraction of a program is negligible, Amdahl's Law assumes the serial fraction will simply run on one worker while the other workers remain idle. Under this assumption adding further workers will never be detrimental. In contrast, DistMP distributes the problem over every available worker and incurs communication costs associated with this. For large numbers of workers, the rising communication costs begin to dominate the parallel fraction of the program resulting in decreasing speed-up and efficiency. In the speed-up plots shown in Figure 4.11a this manifests as a linearly reducing speed-up when large numbers of workers are used, suggesting the presence of a cost per worker proportional to the total number number of workers. The origin of this cost is likely to depend on the implementation details of the various MPI communication operations used.

In summary, DistMP with the number of workers tested does not seem to be primarily limited by the serial fraction of the program as predicted by Amdahl's Law. It is possible that the increase in speed-up per worker added could be as much as doubled by using a platform with more memory channels or greater memory speed. With large numbers of workers speed-up appears to be limited by rising communication costs, suggesting that a modified communication strategy or a change in underlying MPI library could be necessary to achieve higher speed-ups.



(a) Speed-up of DistMP versus Amdahl's Law



(b) Efficiency of DistMP versus Amdahl's Law

Figure 4.11: Comparison between DistMP and Amdahl's Law predictions. DistMP used $n = 3277$ and a range of problem sizes. Amdahl's Law predictions use a range of parallel proportions between $p = 0.92$ and $p = 0.98$.

4.6.2 DistOMP

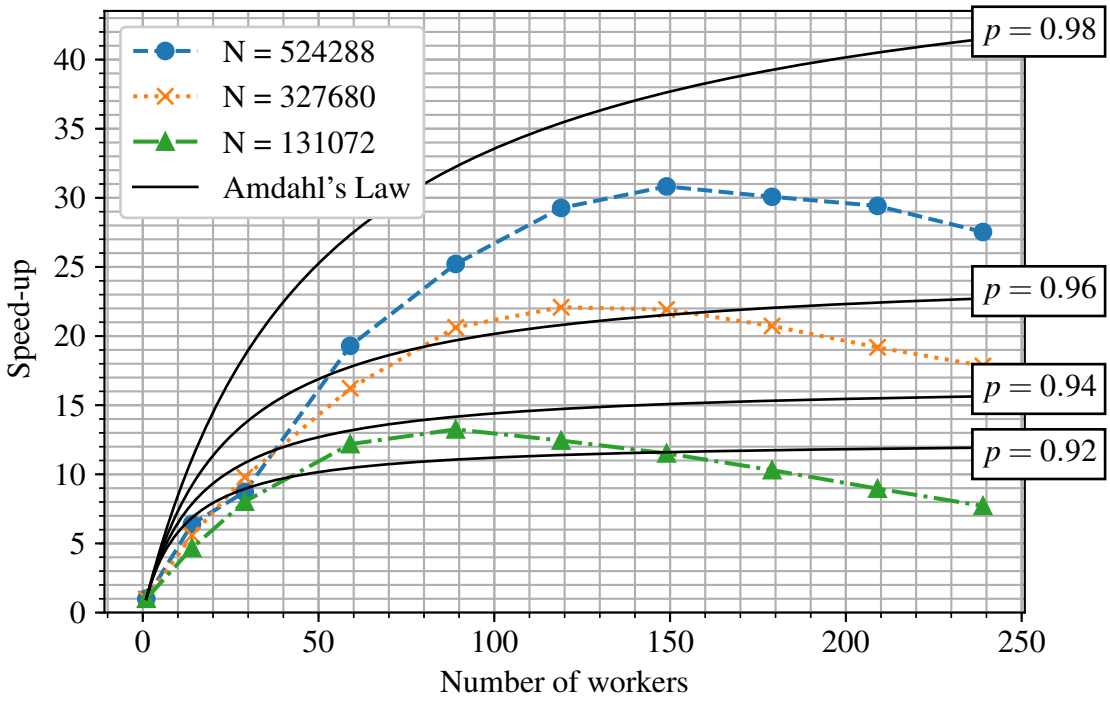
Figure 4.12 shows speed-up and efficiency results for DistOMP with $n = 3277$ and a range of problem sizes compared against predictions made using Amdahl's Law with a range of parallel proportions. The overall trends are similar to those seen for DistMP but with some significant differences.

DistOMP shows an initial efficiency of only around 0.4 with small numbers of workers, a similar value to that of DistMP. This implies that DistOMP may also be limited by available memory bandwidth rather than processor capacity, which would limit the speed-up increase with each added 15-worker node and explain the relatively poor efficiency with small numbers of workers. As with DistMP, this possibility could be investigated with experiments to identify the presence of a memory bottleneck on a single node. If such a bottleneck is limiting performance per node then allocating fewer workers per node could significantly increase calculated efficiency per worker, although this would not improve actual utilisation of the hardware available.

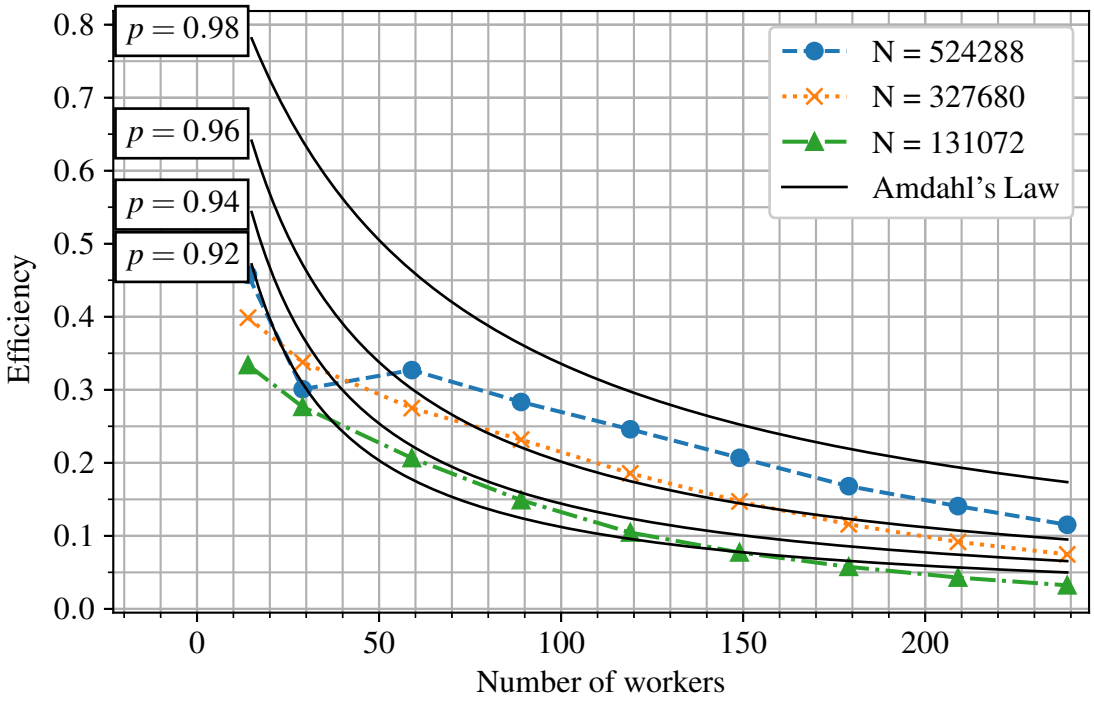
Beyond the optimum number of workers, the speed-up of DistOMP decreases approximately linearly as workers are added. This is similar to the effect seen for DistMP but DistOMP exhibits this effect with fewer workers so that a decrease in efficiency is observed when in excess of approximately 60 workers are used. This is to be expected since DistOMP involves many more communication steps than DistMP, each of which increases the additional cost for each worker added. Furthermore, for relatively dense problems DistOMP involves a significant serial fraction at the back substitution step. Beyond the initial linear speed-up, it appears that significant changes to the communications strategy of the algorithm would be required to increase the level of parallelism and the speed-up benefit from large numbers of workers. These results demonstrate how Amdahl's Law can be easily used to create predictions for the speed-up and efficiency of a parallel algorithm but that its simplistic nature limits the accuracy of these predictions, especially when adding workers significantly increases the total cost of the algorithm due to rising communication overheads.

4.7 Chapter summary

In this chapter I have described how my parallelisation ideas were taken from high-level algorithms to concrete implementations, giving details of the language, libraries and techniques used for the implementations. I also gave details of the Darwin cluster, the parallel compute cluster upon which experiments were run. I then described my methodologies for conducting experiments and for interpreting the results, including how I mitigate variability in observed elapsed time between trials. The bulk of this chapter consisted of my experimental results, characterising the performance of MP, DistMP, OMP, and DistOMP. I showed how DistMP and DistOMP can give significant speed-up over simple multithreaded implementations of MP and OMP when a large number of compute cores are available. I characterised how the performance



(a) Speed-up of DistOMP versus Amdahl's Law



(b) Efficiency of DistOMP versus Amdahl's Law

Figure 4.12: Comparison between DistOMP and Amdahl's Law predictions. DistOMP used $n = 3277$ and a range of problem sizes. Amdahl's Law predictions use a range of parallel proportions between $p = 0.92$ and $p = 0.98$.

of my algorithms varies as different problem parameters are altered. Finally, I showed how my techniques can scale to very large problems by demonstrating solution of a problem with a 429GB equivalent sampling matrix in just over an hour by utilising 240 cores across 16 compute nodes. In the next chapter I will draw conclusions based on these results and discuss their context and possible improvements and new directions for the techniques.

Chapter 5

Conclusions

In Chapter 1 I briefly introduced the fields of CS and high performance parallel computing. I explained the motivation for attempting to apply HPC parallelisation techniques to CS reconstruction and outlined the structure of the remainder of the thesis. In Chapter 2 I discussed CS, HPC, and parallelisation in more detail, covering the concepts which provide a basis for my original developments. In Chapter 3 I presented my novel contributions to the field, the DistMP and DistOMP algorithms. Starting from a detailed analysis of MP and OMP, I showed how partitioning by data parallelism can be used to enable the reconstruction of large CS problems. I presented detailed algorithms for DistMP and DistOMP in terms of the MPI communication primitives used. Then, in Chapter 4 I presented experiments comparing and characterising the performance of MP, OMP, DistMP, and DistOMP when reconstructing problems of various shapes and sizes using varying numbers of compute workers. I also demonstrated solution of a particularly large CS problem to demonstrate how far my techniques can scale.

5.1 Research questions

In Chapter 1 I posed five questions which I would attempt to answer with the research presented in this thesis. I put forward the answers as follows:

1. How can MP and OMP be parallelised using Message Passing Interface (MPI) on a compute cluster?

I presented a comparison of various common parallelisation strategies in Section 2.4.2. I then showed the specific strategies used for DistMP and DistOMP in Sections 3.2 and 3.4 respectively. In both cases the structure is based around data parallelism: the largest matrices are identified and split across the available compute workers in a manner which minimises communications between workers. A more detailed answer to this question is provided by Algorithms 3.2, 3.7, and 3.8.

2. Can the parallel algorithms be used to solve much larger problems than would ordinarily fit in main memory?

In Section 4.5.5 I presented results demonstrating reconstruction of a problem utilising a 429GB effective sampling matrix. This matrix alone is much larger than the main memory available on almost all workstations. Furthermore, this problem was solved in as little as 62 minutes.

3. Can the parallel algorithms solve problems faster than a single compute worker could alone?

The results presented in Section 4.5 demonstrate significant speed-up resulting from increased numbers of compute workers. A speed-up of as much as a factor of 76 was observed when using DistMP with 210 workers (spread across 14 cluster nodes).

4. How efficient are these parallel algorithms?

In Section 4.5 I also present plots of efficiency for the parallel algorithms, indicating the level of overheads present in various scenarios. Efficiency varies from around 5% to 45% for DistMP and DistOMP depending on the number of workers and problem dimensions.

5. How do the properties of the algorithms vary with number of workers and differing CS problems?

In general, the parallel algorithms demonstrate increasing speed-up with addition of compute workers up to some maximum speed-up, beyond which wall-time increases again. For both algorithms, speed-up and efficiency benefit from increased problem size and density. I answer this question in more detail in the discussion throughout Section 4.5.

5.2 Future work

5.2.1 Hybrid parallelisation

The Darwin cluster, described in Section 4.1, has heterogeneous memory access bandwidth and latency both at the socket level and at the node level. In theory, to exploit this environment with maximum efficiency, a hybrid approach should be taken to parallelisation, taking advantage of the increased communications capabilities between cores in a socket, and between the two sockets in a single node. This could be done by running one MPI worker process on each UMA domain (socket) and utilising multithreading to exploit the multiple cores available, for example using OpenMP. Another option would be to run one MPI worker per core and use shared memory communication rather than message passing to communicate between workers on the same socket, or on the same node. The latest version of MPI, MPI3, includes support for shared memory communication to allow a hybrid parallelisation which utilises both shared memory and message passing for communication. However, in addition to greatly increasing the complexity of the software implementation, the parallel algorithms would need to be redesigned from the ground up to support this heterogeneous parallelisation structure.

5.2.2 Implicit matrices

I described in Section 2.2.5 how large CS problems may be solved with modest memory consumption when large, structured matrices may be stored implicitly rather than explicitly. Furthermore, reconstruction algorithms may be accelerated greatly if the equivalent sampling matrix can be replaced with fast transform operators such as the FFT. In this thesis I focused on a general purpose implementation of the parallel algorithms rather than the use of implicit matrices. When implicit matrices are used parallelisation will not be required for memory reasons. However, the use of parallel computing with implicit matrices or fast transform operators may result in speed-up over using a single compute worker. I will now briefly describe how DistMP and DistOMP could be extended to allow the use of implicit matrices and fast transform operators.

In DistMP, two references are made to the local portion of the dictionary $\mathbf{L}$: The selection step ($\mathbf{c} \leftarrow \mathbf{L}^T\mathbf{r}$) and storing the selected column ($\mathbf{g} \leftarrow \mathbf{L}_K$). The former is already a multiplication by the transpose of $\mathbf{L}$ and the latter can be simply represented as a multiplication of $\mathbf{L}$. Since $\mathbf{L}$ is a contiguous subset of the columns of $\mathbf{A}$, if we have fast transform operators for application of $\mathbf{A}$ then these can be easily converted to fast transform operators for application of $\mathbf{L}$. Thus, DistMP may be converted to replace the explicit representation of $\mathbf{L}$ with an implicit matrix or fast transforms with minimal modification.

For DistOMP, the conversion to use implicit matrices or fast transform operators is not as convenient: $\mathbf{P}$ and $\mathbf{R}$ cannot be stored implicitly, so there are still significant memory requirements despite the replacement of $\mathbf{L}$. However, $\mathbf{P}$ and $\mathbf{R}$ are likely to be much smaller than $\mathbf{L}$, and replacing $\mathbf{L}$ is still likely to result in some speed-up. The correlation and selection step may be modified in the same manner described for DistMP in the previous paragraph. The MGS update and BS sections of the algorithm do not make reference to $\mathbf{L}$, so no modification is required. Finally, the residual update section is amenable to an implicit form of $\mathbf{L}$, as we only need carry out a single matrix-vector multiplication by $\mathbf{L}$.

5.2.3 Parallelisation of further greedy algorithms

In this thesis I have considered the parallelisation of MP and OMP, two fundamental greedy pursuits utilised for CS reconstruction. Many other greedy algorithms have been specifically developed for the purpose of CS reconstruction; three common examples are Regularized Orthogonal Matching Pursuit (ROMP) [54], Stagewise Orthogonal Matching Pursuit (StOMP) [52], and Compressive Sampling Matching Pursuit (CoSaMP) [55]. It is likely that these algorithms would be amenable to parallelisation in a broadly similar manner to that used to develop DistMP and DistOMP, but further research would be required to verify this and characterise the performance of the resulting parallel algorithms.

StOMP is similar to OMP except that it uses a hard-thresholding process for column selection: instead of picking the single correlation with the greatest magnitude, a threshold is applied and

all correlations exceeding the threshold are selected. This means multiple columns may be added to the working set in each iteration. Because the update to the working set in each iteration is more significant than OMP, it is no longer attractive to maintain and update the QR decomposition in each iteration. Instead, StOMP uses a CG solver to optimally project the residual onto the working set. Outside of working set storage and the CG solver, StOMP should parallelise in a similar manner to OMP. It is not immediately clear how the CG solver could be parallelised, but it is possible that the working set storage could be striped over compute workers in the same way that $\mathbf{Q}$ is in DistOMP and that the CG solver could then be carried out in parallel across the compute workers.

Iterative Thresholding (IT) is another greedy algorithm which can be used for CS reconstruction. It is very similar to StOMP except that the orthogonal decomposition is replaced by a simple relaxation of the residual onto the working set. It is therefore likely that IT could be parallelised in a similar manner to StOMP.

CoSaMP is also similar to StOMP but adds a pruning step at the end of each iteration where columns may be removed from the working set if they are not deemed useful. Parallelisation of CoSaMP should, therefore, be very similar to StOMP.

ROMP is similar to StOMP in that at each iteration some number of columns are selected by choosing those with the largest magnitude of correlation to the residual. A regularisation step is then applied where the selected columns are separated into groups with similar coordinates and in each group only the column with the greatest energy is selected. The regularised set is then merged with the working set. A LLSQ problem is then solved to orthogonalise the residual in the same way as in OMP. Since the working set is only extended and columns are never removed, it may be possible to use the same parallel MGS technique as OMP to parallelise ROMP. However if a large number of columns are added to the working set in each iteration and there are a relatively small number of iterations, it may be faster to simply use CG to approximate the pseudo-inverse to solve the LLSQ problem independently in each iteration.

5.2.4 Multiple R-workers

In Section 3.4.1 I stated that in OMP the matrix holding the coefficients from the QR decomposition, $\mathbf{R}$, has a maximum size of $n \times n$ where n is the number of rows in the effective sampling matrix. I also explained why $\mathbf{R}$ will usually be much smaller than $\mathbf{A}$ in practical CS problems, so storing $\mathbf{R}$ on a single compute worker would be unlikely to pose a limitation to the size of problem which could be solved. However, problems which require many iterations to solve will lead to $\mathbf{R}$ being large, and using a large number of workers will lead to $\mathbf{L}$ and $\mathbf{P}$ both being smaller. Combined, these could lead to a scenario where fitting $\mathbf{R}$ on a single worker is a limiting factor for the size of problem which can be solved. In this section I will outline modifications to DistOMP which could be made in order to avoid this limitation.

The key to removing this limitation is allowing $\mathbf{R}$ to be split over multiple compute workers.

DistOMP would then be structured as follows: We divide our logical workers into two distinct groups, QA-workers and R-workers, where $\mathbf{A}$ and $\mathbf{Q}$ are split over the QA-workers as before and $\mathbf{R}$ is split over the R-workers. The number of R-workers required is calculated based on the estimated size of $\mathbf{R}$ (from the predicted number of iterations) and the main memory available on each worker. The remaining available workers then become QA-workers.

To implement this strategy, we must first determine whether $\mathbf{R}$ is to be split up by columns or by rows and how we allocate these portions of $\mathbf{R}$ to the R-workers. In each iteration we extend $\mathbf{R}$ by appending a column and then carry out BS, which reads the contents of $\mathbf{R}$ row-by-row, starting from the bottom. Splitting up $\mathbf{R}$ by assigning contiguous sets of rows to each R-worker would allow an R-worker to carry out BS with no communication except to receive a partially complete $\mathbf{a}$ from a previous R-worker and to pass on a more complete $\mathbf{a}$ to the next R-worker. However, this complicates the updates to $\mathbf{R}$, as each new column would need to be distributed across the R-workers.

Splitting up $\mathbf{R}$ by columns greatly simplifies the updates, as we simply append columns on the lowest numbered R-worker which has yet to exhaust its available memory. However, each step of BS would need to involve all the R-workers which held columns. Carrying out the matrix-vector product between $\mathbf{a}$ and the row of $\mathbf{R}$ in parallel could possibly lead to some speed-up, however this is likely counteracted by the extra communication overhead involved.

I have developed a proof of concept implementation using the latter approach. Details are not included in this thesis as it has not been extensively tested and I have not measured the impact on run time of using multiple R-workers. If a convincing motivation for the use of multiple R-workers was found, this technique could be further developed in order to allow the solution of large CS problems which also require a large number of iterations. Furthermore, an implementation partitioning $\mathbf{R}$ row-wise could be developed to compare the performance against partitioning $\mathbf{R}$ column-wise.

5.2.5 Distributed reconstruction with compute accelerators

Darwin, the cluster used for the implementation of DistMP and DistOMP, equipped each compute worker with two CPUs but no specialised accelerator hardware. A recent trend in HPC clusters is for each compute node to be equipped with one or more GPU accelerators. In Section 2.5.1 I reviewed several pieces of research which used GPU compute to accelerate CS reconstruction. The unstructured problems reconstructed using GPUs were smaller than those which could be reconstructed by using distributed algorithms on a cluster, but the use of GPUs was reported to give significant speed-up. By combining the use of distributed algorithms with GPU acceleration of each worker it seems likely that reconstruction time could be greatly reduced for very large problems. Problem data could be transferred to GPUs at initialisation and stored in GPU memory between iterations so that during reconstruction only communications to or from other nodes need to be transferred via the host system. A toolkit such as cuBLAS

could be used to translate the vector operations in DistMP and DistOMP to execute on a compute accelerator without significant modification to the high-level structure of the algorithm.

5.3 Final remarks

In this thesis I introduced two new parallel algorithms I have developed which allow HPC cluster computing to be used for CS reconstruction. Experiments I carried out demonstrate solution of very large problems and enormous speed-up compared to reconstruction using a single processor. I believe my techniques could broaden the scope of CS research by allowing much larger problems to be solved than previously possible. I hope my work stimulates further collaboration between the research areas of CS and HPC: I believe further attention given to parallelisation and high-performance implementations of reconstruction algorithms will help to alleviate the significant computational burden associated with CS reconstruction, which could aid the adoption of CS in new applications.

Bibliography

[1] C. E. Shannon. 'A Mathematical Theory of Communication (Part I)'. In: *Bell System Technical Journal* (1948).

[2] R. Tibshirani. 'Regression Selection and Shrinkage via the Lasso'. In: *Journal of the Royal Statistical Society* 58.1 (1996), pp. 267–288.

[3] S. G. Mallat and Z. Zhang. 'Matching pursuits with time-frequency dictionaries'. In: *IEEE Transactions on Signal Processing* 41.12 (1993), pp. 3397–3415.

[4] D. L. Donoho and X. Huo. 'Uncertainty principles and ideal atomic decomposition'. In: *IEEE Transactions on Information Theory* 47.7 (2001), pp. 2845–2862.

[5] D. L. Donoho and M. Elad. 'Optimally sparse representation in general (nonorthogonal) dictionaries via l1 minimization'. In: *Proceedings of the National Academy of Sciences* 100.5 (2003), pp. 2197–2202.

[6] J. A. Tropp. 'Greed is good: Algorithmic results for sparse approximation'. In: *IEEE Transactions on Information Theory* 50.10 (2004), pp. 2231–2242.

[7] E. J. Candès, J. Romberg and T. Tao. 'Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information'. In: *IEEE Transactions on Information Theory* 52.2 (2006), pp. 489–509.

[8] S. S. Chen, D. L. Donoho and M. A. Saunders. 'Atomic Decomposition by Basis Pursuit'. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 33–61.

[9] E. J. Candès and T. Tao. 'Decoding by linear programming'. In: *IEEE Transactions on Information Theory* 51.12 (2005), pp. 4203–4215.

[10] J. A. Tropp and A. C. Gilbert. 'Signal Recovery from Random Measurements via Orthogonal Matching Pursuit'. In: *IEEE Transactions on Information Theory* 53.12 (2007), pp. 4655–4666.

[11] M. B. Wakin et al. 'An architecture for compressive imaging'. In: *Proceedings of the* 2006 *International Conference on Image Processing*. Atlanta, GA: IEEE, 2006, pp. 1273–1276.

[12] E. Candès and J. Romberg. 'Sparsity and incoherence in compressive sampling'. In: *Inverse Problems* 23.3 (2007), pp. 969–985.

[13] P. Binev et al. 'Compressed Sensing and Electron Microscopy'. In: *Modeling Nanoscale Imaging in Electron Microscopy*. Boston, MA: Springer, 2012, pp. 73–126.

[14] A. Stevens et al. 'The potential for bayesian compressive sensing to significantly reduce electron dose in high-resolution STEM images'. In: *Microscopy* 63.1 (2014), pp. 41–51.

[15] A. Stevens et al. 'Applying compressive sensing to TEM video: a substantial frame rate increase on any camera'. In: *Advanced Structural and Chemical Imaging* 1.10 (2015).

[16] D. J. Brady et al. 'Compressive Holography'. In: *Optics Express* 17.15 (2009), pp. 13040–13049.

[17] Y. Endo et al. 'GPU-accelerated compressive holography'. In: *Optics Express* 24.8 (2016), pp. 8437–8445.

[18] J. Hahn et al. 'Video-rate compressive holographic microscopic tomography'. In: *Optics Express* 19.8 (2011), pp. 7289–7298.

[19] J. Belle, R. Armstrong and J. Gain. 'Accelerated Deconvolution of Radio Interferometric Images using Orthogonal Matching Pursuit and Graphics Hardware'. In: *Journal of Astronomical Instrumentation* 6.4 (2017).

[20] A. Fiandrotti et al. 'GPU-accelerated algorithms for compressed signals recovery with application to astronomical imagery deblurring'. In: *International Journal of Remote Sensing* 39.7 (2018), pp. 2043–2065.

[21] M. Aharon, M. Elad and A. Bruckstein. 'k-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation'. In: *IEEE Transactions on Signal Processing* 54.11 (2006), pp. 4311–4322.

[22] L. Gan. 'Block compressed sensing of natural images'. In: 15*th International Conference on Digital Signal Processing*. 2007, pp. 403–406.

[23] J. E. Fowler, S. Mun and E. W. Tramel. 'Block-Based Compressed Sensing of Images and Video'. In: *Foundations and Trends in Signal Processing* 4.4 (2010), pp. 297–416.

[24] H. T. Kung and S. J. Tarsa. 'Partitioned compressive sensing with neighbor-weighted decoding'. In: *Proceedings of the IEEE Military Communications Conference*. 2011, pp. 149–156.

[25] NVIDIA. *cuBLAS*. 2019. URL: `https://developer.nvidia.com/cublas`.

[26] J. D. Blanchard and J. Tanner. 'GPU accelerated greedy algorithms for compressed sensing'. In: *Mathematical Programming Computation* 5.3 (2013), pp. 267–304.

[27] D. S. Smith et al. 'Real-Time Compressive Sensing MRI Reconstruction Using GPU Computing and Split Bregman Methods.' In: *International journal of biomedical imaging* (2012).

[28] A. Borghi et al. 'A Simple Compressive Sensing Algorithm for Parallel Many-Core Architectures'. In: *Journal of Signal Processing Systems* 71.1 (2013), pp. 1–20.

[29] M. Andrecut. 'Fast GPU implementation of sparse signal recovery from random projections'. In: *Engineering Letters* 17.3 (2009).

[30] Y. Fang et al. 'GPU implementation of orthogonal matching pursuit for compressive sensing'. In: *Proceedings of the International Conference on Parallel and Distributed Systems*. 2011, pp. 1044–1047.

[31] A. Kulkarni et al. 'Low Overhead CS-Based Heterogeneous Framework for Big Data Acceleration'. In: *ACM Transactions on Embedded Computing Systems* 17.1 (2018).

[32] A. Septimus and R. Steinberg. 'Compressive sampling hardware reconstruction'. In: *IEEE International Symposium on Circuits and Systems*: *Nano-Bio Circuit Fabrics and Systems*. 2010, pp. 3116–3119.

[33] H. Rabah et al. 'FPGA Implementation of Orthogonal Matching Pursuit for Compressive Sensing Reconstruction'. In: *IEEE Transactions on Very Large Scale Integration* (*VLSI*) *Systems* 23.10 (2015), pp. 2209–2220.

[34] Ö. Polat and S. K. Kayhan. 'High-speed FPGA implementation of orthogonal matching pursuit for compressive sensing signal reconstruction'. In: *Computers & Electrical Engineering* 71 (2018), pp. 173–190.

[35] A. Kulkarni and T. Mohsenin. 'Low Overhead Architectures for OMP Compressive Sensing Reconstruction Algorithm'. In: *IEEE Transactions on Circuits and Systems I*: *Regular Papers* 64.6 (2017), pp. 1468–1480.

[36] S. Boyd et al. 'Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers'. In: *Foundations and Trends in Machine Learning* 3.1 (2010).

[37] W. Deng et al. 'Parallel Multi-Block ADMM with O(1 / K) Convergence'. In: *Journal of Scientific Computing* 71.2 (2017), pp. 712–736.

[38] A. J. W. Mayer. 'The architecture of the Burroughs B5000: 20 years later and still ahead of the times?' In: *ACM SIGOPS Operating Systems Review* 10.4 (1982), pp. 3–10.

[39] P. Enslow. 'Multiprocessor Organization—a Survey'. In: *ACM Computing Surveys* 9.1 (1977), pp. 103–129.

[40] D. Geer. 'Chip makers turn to multicore processors'. In: *Computer* 38.5 (2005), pp. 11–13.

[41] M. J. Flynn. 'Very High-Speed Computing Systems'. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[42] M. J. Flynn. 'Some computer organizations and their effectiveness'. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960.

[43] E. Strohmaier et al. 'Recent trends in the marketplace of high performance computing'. In: *Parallel Computing* 31.3–4 (2005), pp. 261–273.

[44] The MPI Forum. 'MPI: A Message Passing Interface'. In: *Proceedings of the* 1993 *ACM/IEEE Conference on Supercomputing*. 1993, pp. 878–883.

[45] J. Dongarra. 'Trends in High-Performance Computing'. In: *Handbook of Nature-Inspired and Innovative Computing*. Boston, MA: Springer, 2006. Chap. 15, pp. 511–520.

[46] A. Grama et al. *Introduction to Parallel Computing*. United Kingdom: Pearson, 2003.

[47] P. S. Pacheco. *Introduction to Parallel Programming*. Elsevier, 2011.

[48] I. T. Foster. *Designing and Building Parallel Programs*. Boston, MA: Addison-Wesley Longman, 1995.

[49] E. J. Candès and M. B. Wakin. 'An Introduction To Compressive Sampling'. In: *IEEE Signal Processing Magazine* 25 (2008), pp. 21–30.

[50] R. Baraniuk et al. 'A Simple Proof of the Restricted Isometry Property for Random Matrices'. In: *Constructive Approximation* 28.3 (2008), pp. 253–263.

[51] T. Blumensath and M. E. Davies. 'Iterative thresholding for sparse approximations'. In: *Journal of Fourier Analysis and Applications* 14.5–6 (2008), pp. 629–654.

[52] D. L. Donoho et al. 'Sparse Solution of Underdetermined Systems of Linear Equations by Stagewise Orthogonal Matching Pursuit'. In: *IEEE Transactions on Information Theory* 58.2 (2012), pp. 1094–1121.

[53] D. Needell and R. Vershynin. 'Uniform uncertainty principle and signal recovery via regularized orthogonal matching pursuit'. In: *Foundations of Computational Mathematics* 9.3 (2009), pp. 317–334.

[54] D. Needell and R. Vershynin. 'Signal Recovery From Incomplete and Inaccurate Measurements Via Regularized Orthogonal Matching Pursuit'. In: *IEEE Journal of Selected Topics in Signal Processing* 4.2 (2010), pp. 310–316.

[55] D. Needell and J.A. Tropp. 'CoSaMP: Iterative signal recovery from incomplete and inaccurate samples'. In: *Applied and Computational Harmonic Analysis* 26.3 (2009), pp. 301–321.

[56] W. Dai and O. Milenkovic. 'Subspace Pursuit for Compressive Sensing Signal Reconstruction'. In: *IEEE Transactions on Information Theory* 55.5 (2009), pp. 2230–2249.

[57] E. J. Candès and T. Tao. 'Near-optimal signal recovery from random projections: Universal encoding strategies?' In: *IEEE Transactions on Information Theory* 52.12 (2006), pp. 5406–5425.

[58] E. J. Candès, J. K. Romberg and T. Tao. 'Stable signal recovery from incomplete and inaccurate measurements'. In: *Communications on Pure and Applied Mathematics* 59.8 (2006), pp. 1207–1223.

[59] A. M. Tillmann and M. E. Pfetsch. 'The computational complexity of the restricted isometry property, the nullspace property, and related concepts in compressed sensing'. In: *IEEE Transactions on Information Theory* 60.2 (2014), pp. 1248–1259.

[60] D. L. Donoho, M. Elad and V. N. Temlyakov. 'Stable Recovery of Sparse Overcomplete Representation in the Presence of Noise'. In: *IEEE Transactions on Information Theory* 52.1 (2006), pp. 6–18.

[61] D. Slepian and J. K. Wolf. 'Noiseless Coding of Correlated Information Sources'. In: *IEEE Transactions on Information Theory* 19.4 (1973), pp. 471–480.

[62] M. F. Duarte et al. 'Distributed compressed sensing of jointly sparse signals'. In: *Proceedings of the* 2005 *Asilomar Conference on Signals, Systems, and Computers*. Pacific Grove, CA: IEEE, 2005, pp. 1537–1541.

[63] D. Baron et al. 'An Information-Theoretic Approach to Distributed Compressed Sensing'. In: *Proceedings of the* 43*rd Allerton Conference on Communication Control and Computing*. 2005.

[64] M. S. Schlansker and B. R. Rau. *EPIC*: *An Architecture for Instruction-Level Parallel Processors*. Tech. rep. 2000.

[65] H. Sharangpani and K. Arora. 'Itanium processor microarchitecture'. In: *IEEE Micro* 20.5 (2000), pp. 24–43.

[66] W. J. Watson. 'The TI ASC: a highly modular and flexible super computer architecture'. In: *Proceedings of the Fall Joint Computer Conference, part I*. Anaheim, California: ACM Press, 1972, pp. 221–228.

[67] L. Stringer. 'Vectors: How the Old Became New Again in Supercomputing'. In: *HPC Wire* (2016).

[68] R. M. Hord. *The Illiac IV*. Springer, 1982.

[69] N. Hirose and M. Fukuda. 'Numerical Wind Tunnel (NWT) and CFD research at National Aerospace Laboratory'. In: *High Performance Computing on the Information Superhighway*. Seoul, South Korea: IEEE, 1997.

[70] R. Esser and R. Knecht. 'Intel Paragon XP/S - Architecture and Software Environment'. In: *Supercomputer* '93. Berlin, Heidelberg: Springer, 1993.

[71] J. Bashor. 'Researchers Achieve One Teraflop Performance With Supercomputer Simulation Of Magnetism'. In: *Berkeley Lab Research News* (1998).

[72] S. Ristov et al. 'Superlinear Speedup in HPC Systems: why and when?' In: *Proceedings of the Federated Conference on Computer Science and Information Systems*. 2016, pp. 889–898.

[73] G. M. Amdahl. 'Validity of the single processor approach to achieving large scale computing capabilities'. In: *Proceedings of the April* 18-20*,* 1967*, spring joint computer conference*. ACM, 1967.

[74] J. L. Gustafson. 'Reevaluating Amdahl's law'. In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[75] S. Balay et al. *PETSc Web page*. 2018. URL: `http://www.mcs.anl.gov/petsc`.

[76] S. P. Johnson, C. S. Ierotheou and M. Cross. 'Automatic parallel code generation for message passing on distributed memory systems'. In: *Parallel Computing* 22.2 (1996), pp. 227–258.

[77] A. Saa-Garriga. 'Automatic Source Code Adaption for Heterogeneous Platforms'. PhD thesis. Universitat Autonoma de Barcelona, 2016.

[78] Z. Yu et al. 'Fast compressive sensing reconstruction algorithm on FPGA using Orthogonal Matching Pursuit'. In: *IEEE International Symposium on Circuits and Systems* (*ISCAS*). 2016, pp. 249–252.

[79] J. L.V.M. Stanislaus and T. Mohsenin. 'High performance compressive sensing reconstruction hardware with QRD process'. In: *IEEE International Symposium on Circuits and Systems*. 2012, pp. 29–32.

[80] Y. Quan et al. 'FPGA Implementation of Real-Time Compressive Sensing with Partial Fourier Dictionary'. In: *International Journal of Antennas and Propagation* (2016).

[81] G. Huang and L. Wang. 'High-speed Signal Reconstruction for Compressive Sensing Applications'. In: *Journal of Signal Processing Systems* 81.3 (2015), pp. 333–344.

[82] Y. Chen and X. Zhang. 'High-speed architecture for image reconstruction based on compressive sensing'. In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2011.

[83] D. Sundman et al. 'Distributed predictive subspace pursuit'. In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 4633–4637.

[84] D. Sundman, S. Chatterjee and M. Skoglund. 'Distributed greedy pursuit algorithms'. In: *Signal Processing* 105 (2014), pp. 298–315.

[85] D. Sundman, S. Chatterjee and M. Skoglund. 'Design and analysis of a greedy pursuit for distributed compressed sensing'. In: *IEEE Transactions on Signal Processing* 64.11 (2016), pp. 2803–2818.

[86] T. Wimalajeewa and P. K. Varshney. 'Cooperative sparsity pattern recovery in distributed networks via distributed-OMP'. In: *IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. 2013, pp. 5288–5292.

[87] C. Ravazzi, S. M. Fosson and E. Magli. 'Randomized Algorithms for Distributed Nonlinear Optimization Under Sparsity Constraints'. In: *IEEE Transactions on Signal Processing* 64.6 (2016), pp. 1420–1434.

[88] J. F. C. Mota et al. 'Distributed basis pursuit'. In: *IEEE Transactions on Signal Processing* 60.4 (2012), pp. 1942–1956.

[89] S. Bernabe et al. 'GPU implementation of a hyperspectral coded aperture algorithm for compressive sensing'. In: *International Geoscience and Remote Sensing Symposium* (*IGARSS*). 2015.

[90] Y. Yang et al. 'ADMM-CSNet: A Deep Learning Approach for Image Compressive Sensing'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018).

[91] Åke Björck. *Numerical Methods for Least Squares Problems*. SIAM, 1996.

[92] N. D. Matsakis and F. S. Klock. 'The Rust Language'. In: *ACM SIGAda Ada Letters* (2014).

[93] Q. Wang et al. 'AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013).

[94] Andrew Straw et al. *BLAS* (*Rust binding*). 2018. URL: `https://crates.io/crates/blas` (visited on 21/07/2018).

[95] L. Dagum and R. Menon. 'OpenMP: an industry standard API for shared-memory programming'. In: *IEEE Computational Science and Engineering* (1998).

[96] B. Steinbusch and A. Gaspar. *rsmpi* (*Rust binding*). 2018. URL: `https://crates.io/crates/mpi` (visited on 21/07/2018).

[97] W. Chen and I. J. Wassell. 'Energy efficient signal acquisition via compressive sensing in wireless sensor networks'. In: 6*th International Symposium on Wireless and Pervasive Computing* (*ISWPC*). Vol. 2. Hong Kong: IEEE, 2011.

[98] M. E. Lopes. 'Estimating unknown sparsity in compressed sensing'. In: *Proceedings of the* 30*th International Conference on Machine Learning*. Atlanta, GA, 2013, pp. III–217–III–225.

[99] M. E. Lopes. 'Unknown Sparsity in Compressed Sensing: Denoising and Inference'. In: *IEEE Transactions on Information Theory* 62.9 (2016), pp. 5145–5166.

[100] D. L. Donoho. 'Compressed sensing'. In: *IEEE Transactions on Information Theory* 52.4 (2006), pp. 1289–1306.