PhD Thesis.

A Distributed Imaging Framework for the analysis and visualization of multi-dimensional bio-image datasets, in High Content Screening applications.

Submitted to:

Faculty of SEC

Kingston University

Surrey

UK

By

Colin Anthony MCLAY

April 2015

Abstract

This research presents the DFrame, a modular and extensible distributed framework that simplifies and thus encourages the use of parallel processing, and that is especially targeted at the analysis and visualization of multi-dimensional bio-image datasets in high content screening applications. These applications typically apply pipelines of complex and time consuming algorithms to multiple bio-image dataset streams and it is highly desirable to use parallel resources to exploit the inherent concurrency, in order to achieve results in much reduced time scales.

The DFrame allows pluggable extension and reuse of models implementing parallelizing patterns, and similarly provides for application extensibility. This facilitates the composition of novel parallelized 3D image processing applications. A client server architecture is adopted to support both batch and long running interactive sessions. The DFrame client provides functions to author applications as workflows, and mediates interaction with the server. The DFrame server runs as multiple cooperating distributed instances, that together orchestrate to execute tasks according to a workflow's implied order. An inversion of control paradigm is used to drive the loading and running of the models that themselves then coordinate to load and parallelize the running of each task specified in a workflow. The design opens up the opportunity to incorporate advanced management features, including parallel pattern selection based on application context, dynamic 'in application' resource allocation, and adaptable partitioning and composition strategies. Generic partitioning and composition mechanisms for supporting both task and data parallelism are provided, with specific implementation support applicable to the domain of 3D image processing.

Evaluations of the DFrame are conducted at the component level, where specific parallelizing models are applied to discrete 3D image filtering and segmentation operators and to a ray tracing implementation. A complete integrated case study is then presented that composes component entities into multiple image processing pipelines to more fully demonstrate the power and utility of the DFrame, not only in terms of performance, but also to highlight the extensibility and adaptability that permeates through the design, and its applicability to the domain of multi-dimensional image processing. Results are discussed that evidence the utility of the approach, and avenues of future works are considered.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisors. Firstly, I am indebted to my Director of Studies, Dr. Andreas Hoppe for his unwavering support and guidance throughout the project. His motivation, enthusiasm, belief and encouragement have propelled me through the project, enjoying the triumphs and enduring the disappointments that are an inevitable part of such an intense process. Our regular and numerous technical discussions have helped greatly in forming a cohesive, clear and balanced project spanning both the parallel processing and image processing domains. Secondly I would like to thank Dr. Souheil Kaddaj for his considerable advice and inspiration in the overall approach and direction, and as important for having the opportunity to leverage his expertise on the complex parallel processing aspects of the project and have benefited greatly from all the feedback and suggestions. Thirdly I would also like to thank Dr. Darrel Greenfield for the many discussions on various aspects of the project ranging from image IO to MPI. The sage advice from all of my supervisors has been critical to the success of this project, including the guidance on scope and depth and ensuring that I maintained focus on the important core components of the project. I have been fortunate to have had such positive mentors with wide ranging subject knowledge, insightful guidance and timely advise, recommendations, support and also reassurance whenever my resolved wavered, as it often did.

I also thank the technical staff at Kingston University for their support on an operational level. The project has made extensive use of the parallel processing facilities of the University, and being conducted on a part time basis, it has taken a long time to complete. Inevitably during that time, the cluster resources have evolved, being expanded, merged and reconfigured, and the assistance of the technical staff, in particular Colin Bethel has been very much appreciated.

Finally, I thank my family for their understanding and committed support and for the unilateral acceptance of the often extended periods of virtual absence where I was totally immersed in driving the project forward. They are my most treasured accomplices through life, and without their total encouragement and support it would have been inconceivable to push the endeavour to a conclusion, and I am lucky, grateful and very proud.

Chapter 1 Introduction

Advances in light microscopy and image processing led to the development of quantitative image analysis techniques in cell biology. The combination of confocal microscopy and reliable fast scanning of a sample at varying focal planes enabled the means to build up large 3D bio-image datasets (Gu 1996). A common motivating example in biomedical research is the capture of numerous 3D images from across a population, and to analyse them to detect anomalous conditions. Another related area of interest is the analysis of time lapsed 3D image observations on dynamic samples. For example, as part of a recent study to identify kinases that affect lung cancer cell migration (Lara, Mauri et al. 2011), multiple sequences of 3D image stacks were automatically captured for 18 hours (1 image/10 min), from a prepared 96 well plate cell motility assay.

It is impractical to manually analyse datasets of such size and number and this has prompted the development of automated imaging systems. These High Content Screening (HCS) systems are an efficient approach for assessing cell features (Loo, L. F. Altschuler, S. J. 2007). HCS uses automated microscopy to capture images of cells from a large number of cell culture dishes. Individual cells are resolved using segmentation techniques, so that features quantifying cell structure can be extracted for analysis, for thousands of cells. The study of dynamic processes of living cells in 3D, at high spacial and temporal resolutions is demonstrably feasible, and becoming more important and routine. The size of these 3D datasets is an order of magnitude larger than 2D images, and the applied image processing techniques are substantially more computationally expensive.

Commercial HCS systems generally bring other constraining factors into consideration. For instance, the detailed workings of proprietary algorithms may not be published, they often cannot be adapted, and extensibility can be restricted. The user may have to lobby for inclusion of unsupported features, and must accept the cadence of product development life cycles. In the non commercial sector, ImageJ (Perez, Pascau 2013) is a very popular open source laboratory work horse for analysing and visualising images, but is geared towards manual operation applied to individual images. CellProfiler (Carpenter, Jones et al. 2006) is also a notable open source image analysis software allowing the construction of

a pipeline of algorithms to apply to cell images, and it can be arranged to replicate a pipeline across a cluster of nodes such that batches of images can be processed concurrently. However, there is limited support for time-lapsed multi-dimensional image analysis, or parallelised algorithms.

This state of affairs motivates the core focus of this work, namely to facilitate applying parallel processing technologies to advanced dynamic 3D bio-image HCS applications to reduce the time to results, feedback and insight. To provide context and argument for a new approach, the following section expands on the motivations and challenges both in the domain of 3D image processing and in that of parallel processing. The aims and objectives of the research are then formally presented, followed by a section outlining the significant contributions and a brief description of the organization of the thesis.

1.1 Motivations and challenges

1.1.1 Multi-dimensional Image Processing

As outlined above, an HCS image processing application is usually arranged to apply pipelines of sophisticated and compute intensive operators to multiple streams of images. An archetypal pipeline design comprises multiple operators to prepare an image, extract features and analyse them, and then to visualize the results. Insights derived from this process then drives further adjustments and iterations. There is evidently scope for running separate streams in parallel, and there is often considerable opportunity to exploit parallelism within each operator. A brief discussion on types of image processing operators is presented below, elucidating those characteristics of an operator that are suitable and thus motivate parallel processing, and those that present challenges. It is also noted here that parallel processing is particularly attractive when handling large high resolution 3D images, not only because of the increased compute requirements, but also due to the ability to sufficiently partition and distribute such datasets while still maintaining enough computations for each part, compared to the introduced communication overhead (a trade off dependant on the characteristics of a particular distributed system). Another related advantage is that such data partitioning can also be of benefit in the handling of these extreme scale datasets by reducing the per node memory footprint (which also offers the potential to improve cache usage).

Image processing operators are categorised as 'local operators' when the calculation of an output image point is dependent only on a corresponding input image point or that input image point and its neighbours. Examples of these include smoothing, edge detection and thresholding operators, where a small kernel image is convolved locally with each input image point, to produce an output image. These local operators are particularly suited to parallel processing as an image can be partitioned into smaller sub-images and the appropriate kernel is then convolved independently with each partitioned sub-image. Output sub-images are then recomposed to produce the output image. A minor complication is introduced in that boundary information has to be supplied along with each sub-image increasing its size, commensurate with the size of the small kernel being applied. Smoothing operators are often used early in an imaging pipeline to prepare an image, while edge detection and thresholding operators form part of the image processing armoury to segment an image as a precursor to feature extraction (Sonka, Hlavac et al. 2008).

Another prominent image segmentation approach is the region growing technique that seeks to label image regions that are similar according to some stipulated criteria. Region based operators are more global in nature, and thus more challenging to parallelise. By definition, global operators require some form of communication amongst the processing entities, and the extent of this will impact the performance of parallel processing. In the case of region growing, distinct regions may span multiple sub-images of an image partitioned for parallel execution, and will require communication amongst the processes to carefully resolve and label the distinct regions. In general, greater consideration has to be made about how to effectively partition and distribute the processing of global operators, and the expected worth. Indeed, such considerations also motivate the search for alternative algorithms more suited to parallel execution.

Once an image has been segmented for some purpose, relevant features can be extracted based on the segmentation, and passed to analysis operators which can take many forms. For instance, a bio-cell based image segmentation may report features such as each cell's maximum length, breadth and volume, and across an image stream can report on a cell's motility. Analysis can then be performed to investigate cell size, shape and movement and there is often scope to run this in parallel, for instance by arranging for separate processing elements to extract

features from different cells and to then pass those features over to appropriate machine learning regression or classification algorithms (Bishop 2006) which are themselves parallelizable in many cases.

It is usually desirable to visualize the overall output of a pipeline, and possibly even the output of intermediate stages. Even in a fully automated system, anomalous or interesting output is usually marked or highlighted for subsequent manual inspection. An example may be the visualization of interesting artefacts revealed through the analysis. For 3D images, processing becomes commensurately more complex, one technique being to use direct volume ray tracing to zoom into and explore interesting areas of an image. This is a very powerful technique, and since individual rays can be processed separately, is well suited to parallel processing.

It is evident that there is plenty of scope for parallelizing a typical 3D image processing pipeline. Apart from separate streams, many of the common image processing operators themselves exhibit inherent parallelism, and there are also similar opportunities to be found during the more general analysis stage and for visualization, to employ parallel processing. That these compute intensive pipelines are likely to be run and rerun repeatedly argues strongly for the use of parallel processing, motivating its use to reduce time to results and to provide a more interactive experience.

1.1.2 Parallel Processing

In informal usage, the terms 'parallel processing' and 'distributed computing' are often used interchangeably, but there is a distinction. The term 'Parallel Processing' is usually used to describe a single application that creates sub-tasks of a particular task, that are to be executed at the same time usually on a set of homogenous computers connected using a dedicated high throughput network. The related term 'Distributed Computing' is used to describe applications that create tasks to be executed at different locations and at different times, using different resources, a prototypical example being the client server architectures implemented across heterogenous collections of computers (e.g. Beowulf clusters) connected over a more general and perhaps less reliable network. Parallel processing is also more about speed up and high performance computing (HPC), whereas distributed computing is about robustness using remote resources, with

core considerations being fault tolerance, availability and quality of service. However, both concepts do emphasise 'cooperation' (c.f. 'concurrent computing' where threads/processes compete for resources). This work concentrates primarily on parallel processing, but it is noted that parallel processing can also form part of a broader distributed computing system, aspects of distributed computing are relevant, and the distinction is somewhat blurred.

Parallel processing can be distinguished from it's serial counterpart in at least three core areas: the creation and support of parallel execution, an induced requirement for communication and synchronization amongst the cooperating processes, and the handling of partial failures must be considered. An early paper highlighting these differences (Bal, Steiner et al. 1989) is also illuminating in identifying nearly 100 distributed programming languages at that time! If a computation can be partitioned into independent subtasks, then 'task parallelism' can be exploited by arranging to execute the subtasks concurrently on multiple processors, to achieve a reduction in the total execution time of the computation, the essence of *high performance computing* (HPC). As well, datasets can often be partitioned to also leverage 'data parallelism'. Task and data partitioning for parallelism results in the need for communication to distribute the work, collect results and implicitly or explicitly manage dependencies between the subtasks.

There are compelling reasons for using parallel processing. A very simple concept and core attraction is to provide performance improvements such that the time to execute complex computations can be completed in much reduced timescales. A related motivator is the observation that many problems contain considerable scope for parallel execution, although it is important to note that a specific problem will still require careful analysis to identify the form and extent of the parallelism it contains. Nonetheless, as noted above, 3D imaging HCS applications present many opportunities to leveraging parallelism.

However, the conceptual simplicity masks the varied and formidable challenges that must be addressed to effectively harness parallelism. Testament to these challenges is that sequential processing has reigned supreme until very recently, in part due to the relative simplicity of the Von Neumann single processor model as compared to the complexity of parallel execution. Interestingly, Von Neumann did recognise the advantages of parallel processing (Neumann 2000) . However, technological challenges have beset continued improvements in the single CPU,

including the design and manufacture at ever decreasing scales, the 'power wall', and limits to Instruction Level Parallelism (ILP) (Wall 1991) . This has resulted in a profound shift to designing 'scaled in' parallelism, with the emergence of multi-core processor designs, and active research into many-core units (Gschwind 2006) . Indeed, it has been noted that the computing landscape is undergoing fundamental changes and the renewed interest in parallel processing is being forced upon a reluctant and concerned industry (Asanovic, Bodik et al. 2006) . Where in the past, the complexities of parallel processing could be sidestepped by relying on Moore's law and waiting for a faster single CPU, or kept somewhat manageable by building clusters from single CPU components, the true 'parallel age' has now arrived and should be embraced.

It is also worth emphasising the distinction between the identified parallelism in a problem, and the challenge of mapping that parallelism onto hardware for parallel execution, both endeavours being core to successfully parallelising a computation. However, they are not entirely independent of one another. For example, when the system architecture is pre-defined, this can constrain the approach to parallelism that can be taken. Even if a choice of hardware architecture can be made, it may not be possible to minimise the execution times of every problem using that same hardware, due to differing problem requirements. As well, a particular language may be appropriate for some problem but awkward to apply to others. The designer has to first identify parallelism and then map that onto a system that may not be optimal for the particular problem. A prime goal of parallel processing designers is to address this challenge transparently (Tanenbaum, Steen 2006), but full transparency is not always achievable in practice, having to be balanced with efficiency. Finding a suitable parallelising strategy for a given problem, managing the distribution of data and organising safe access and processing to take advantage of locality is challenging and substantially increases the burden of the programmer.

A major challenge is that using low level parallel processing technologies directly is an onerous and error prone enterprise that requires specialized skills. The diversity of distributed compute architectures is served by an even greater diversity of lower level languages and libraries, resulting in a somewhat bewildering choice of technologies. Identifying parallelism in a problem can sometimes be obvious (but not always), but applying this in practise is a difficult

endeavour. At a low level, reasoning is much harder, and this motivates the argument that a higher level framework is all but essential to enable the productive construction of novel and complex parallelised 3D imaging applications, which is the focus of this work.

1.2 *Aims and Objectives*

The aim of this research is to devise and develop a modular and extensible distributed framework which for brevity is called the DFrame. Targeting distributed and shared memory system architectures, the DFrame will be built on top of MPI technology. The DFrame will facilitate the independent development of models implementing parallel processing patterns, and apply these to the parallel processing of multi-dimensional bio-image datasets in the context of HCS applications. The DFrame will be extensible to allow for the subsequent incorporation and reuse of new models and application algorithms, and will thus enable the composition of novel parallelised image processing applications. As important, the DFrame design is intended to open up the opportunity to incorporate advanced management features, including automated parallel pattern selection based on application context, adaptable partitioning and composition strategies and dynamic 'in application' resource allocation. Furthermore, by shielding users from the complexities of low level parallelization, such a framework could bring the power of MPI to a wider audience, outside the High Performance Community.

Supporting this aim will be the development of 3D image processing infrastructure and operators that will be used to test the framework. A complete integrated case study of a pipelined HCS application will also be included. The analysis of the results will demonstrate the degree of success of the approach in terms of performance and its suitability in the targeted domain. Insights derived from this work are expected to guide future research.

Objectives for the distributed framework

1. The DFrame server core will be developed such that multiple running instances of the DFrame can be started and arranged to cooperate with one another.
2. Plugin mechanisms will be designed, to load models implementing parallel

patterns and the application algorithms that will use them.

3. An integrated workflow component will understand and orchestrate the running of workflow specifications (i.e. task graphs). Each node in the workflow will itself be potentially parallelised according to a selected model.

4. A higher level messaging protocol will be devised so that domain specific messages can be passed around the distributed system. This will be integrated into an MPI packing and unpacking mechanism, for transparent message passing (for DFrame and parallel control model use).

5. Partitioning, execution and composition interfaces will be specified that applications can implement, for reuse in the parallel processing models.

6. Design and integrate master-worker and mesh models to provide initial parallel patterns well suited to many operations in image processing.

7. Provide a DFrame graphical user interface (GUI) client to facilitate authoring of task graph specifications, and allow interaction with the DFrame server (using MPI), to run task graph specifications and receive results.

Objectives for multi-dimensional image processing

1. An image IO (input-output) library will be developed specifically for storing and retrieving 3D images to and from disk.

2. An imaging toolkit will provide 3D image operators. Initially this will include averaging filters, a 3D Sobel filter and a 3D image watershed segmentation operator. A separate direct volume rendering visualization (ray tracing) library will also be provided.

3. A module will be developed containing infrastructure to integrate imaging toolkit algorithms into appropriate parallel control models. Implementations of DFrame interfaces (driven by the DFrame models) will organise the partitioning, execution of 3D image application code and the composition of results. Implementations of the DFrame message protocol will allow for transparent distribution and gathering of 3D image partitions.

4. Run the DFrame with simple 'one parallelized task' workflows, to test the individual image operators and collect and analyse the results. Integrated

diagnostics will capture timing measurements for this purpose. This will provide feedback at the individual image operator level.

5. Run the DFrame in the context of a complex pipeline workflow on multiple sequences of 3D images, to fully test the DFrame at the component level and at the broader application level when running multiple pipelines each containing multiple parallelised 3D image operators.

6. Analyse the results in terms of performance and the success of the overall process of composing and running complex image processing workflows using the DFrame. Include recommendations to further develop the framework.

1.3 Contribution

This thesis presents the DFrame, a modular and extensible distributed framework that facilitates the use of parallel processing in large scale multi-dimensional image HCS applications, and thus enables speed up in these applications to obtain results in much reduced timescales. The user is spared much effort by reusing a framework that takes care of the details of managing a parallel problem efficiently. Specifically, the framework is necessary to bridge a gap not present in existing imaging systems that tend to focus on the image processing and not on the parallelisation. As such they are limited to simple batch processing of configured pipelines of algorithms and are often constrained in the parallel processing models they support.

Extensibility and adaptability permeate throughout the design of the DFrame. New models implementing mainstream and novel patterns of parallel processing can be plugged in to extend the framework, that can then be reused across multiple applications. Application modules then plug in to the parallelising models to leverage and reuse already available functionality. In this way, the DFrame provides the flexibility to compose new and novel parallelised multi-dimension image processing applications. The DFrame design provides a clear separation of concerns for those improving and extending the distributed framework and those using the framework for domain applications. Parallelisation experts can develop and plug in new parallel models into the framework, and the core framework itself can be separately developed, improved and enhanced. Application developers can

then focus on designing their specific domain algorithms and link them into modules that hook into the parallel models. Specifically, they will be shielded from much of the effort and complexity associated with developing programs for parallel execution and thus will be encouraged to leverage parallel processing resources via the proposed framework.

The DFrame uses an inversion of control paradigm to drive and manage the parallel runtime of the system, loading and running distributed model instances that themselves then coordinate to load and run application code, according to composed task graph specifications (workflows). The incorporated workflow component operates in concert with the core runtime, to manage the execution of tasks according to dependencies defined in a task dependency graph, processor groups being apportioned amongst the tasks according to resource availability and by application code requirements and characteristics (using advanced features of MPI). Appropriate model selection can be adapted based on the context in which a task runs. Generic partitioning and composition mechanisms for supporting both task and data parallelism are provided, with specific implementation support applicable to the domain of 3D image processing, including integrated automated partitioning strategies that adapt to the shape of the input 3D image data.

A simple GUI has been designed and implemented in QT4 (Blanchette, Summerfield 2008) that allows users to compose parallelised tasks into these dependency graphs using 'drag and drop' functionality, to express the required application workflows. The DFrame defines an API that exposes the available modules, and their functions, allowing the GUI client to retrieve and render information describing each application module and the functions it supports. The design also allows for the interactive update of node parameters with immediate parallelised rerun of the pipeline (or pipeline part), and inspection of the output. This should provide rapid feedback on the effect of specific parameter adjustments.

As part of the proof of concept, a 3D imaging toolkit library is provided with general 3D image IO, and image structure facilities. Other utilities include averaging filters and a gradient operator (edge detector), a watershed image segmentation algorithm and a direct volume ray tracing module is implemented. Infrastructure that bridges the imaging domain into the framework is also provided, including all the general 3D image partitioning apparatus that is required to partition, distribute,

gather and compose 3D images across multiple parallelised image processing algorithms (with 'ghost' cell support). For parallel processing control, pluggable master-worker and mesh models are provided. A cancer cell invasion case study is conducted that provides useful preliminary information on the success of applying the framework to a specific real world 3D image HCS application.

The DFrame provides a foundation for further research into parallel processing itself, and by alleviating some of the burden of parallel processing, it encourages the development of image processing algorithms that can be plugged in and reused by other researchers. A main motivator is to utilise parallel resources to provide faster research results in the multi-dimensional image processing domain, with specific emphasis on HCS. Together with the plugin mechanism for parallel patterns and application code, the DFrame can be composed into an application itself, and run in a client server fashion on long running batch or interactive sessions, and the project provides new information and insight on the success of this approach.

1.4 Structure of the Thesis

In this introduction, the target image processing application is described as the specific motivator for a novel distributed framework. Specific motivations and challenges are reviewed pertinent to the domain of 3D image processing and to that of parallel processing. The aims and objectives are formally stated, and finally the core contributions of the research are outlined.

Chapter 2 presents a review of the common computer architectures that support parallelism, and the languages and libraries developed to help program these systems. The architectures include shared memory and the more scaleable distributed memory systems, and the review elaborates on the categorisations in terms of creating and controlling parallelism and the appropriate process communication methods. The abstract modelling of parallelism, and the diverse software models and programming paradigms developed to support these system architectures is discussed. A selection of parallel languages and libraries is then reviewed that have evolved to support these models and paradigms. These assessments are of central interest to the project, in helping to determine the technology to adopt for a distributed framework suited to 3D image processing in an HCS context.

Chapter 3 moves to a higher level, reviewing patterns of parallel processing and their combination with components to form useful parallel programming frameworks. The definition of a pattern is introduced, along with a brief review of some development methodologies that leverage patterns and the concept of pattern languages. Consideration then turns to the concept and utility of frameworks, and a critical review of various representative frameworks designed to support patterns of parallel processing. The common goal of such frameworks being to simplify and assist the development of applications that can benefit from parallel processing. Although no one framework was deemed entirely suitable as a distributed framework for the targeted image processing applications, the review is important in assessing aspects of these frameworks that would be required or helpful to such a framework.

Chapter 4 presents the DFrame, a novel distributed framework that provides extensible support for different parallel processing models, and pipeline composition of tasks using these models. The case is argued, for a new approach, and the target application requirements are discussed. Concepts are then introduced that form the basis for the distributed framework. An overview of the DFrame architecture follows, and then a more detailed look at each component. The DFrame is of course only half the of equation, providing the generic parallel processing framework. Since the target is 3D image processing, the chapter goes on to describe a basic imaging toolkit designed to interface with the DFrame generic interfaces to provide core 3D image processing functionality. The chapter concludes with a brief description of a Graphical User Interface (GUI), used to construct and manage task graphs and to communicate with a DFrame root instance to request the running of a task graph specification as a workflow.

Chapter 5 centres on evaluating the DFrame at the component level, with the evaluations conducted in the context of 3D imaging of cells. The evaluations include parallelised components to de-noise 3D images, and to apply edge detection and region based cell segmentation techniques. A parallelised server side direct volume rendered visualization capability is also developed and evaluated. The objectives being twofold, firstly to demonstrate the design of the DFrame at the component level, and secondly to test the developed 3D imaging capability to filter, detect and segment cells in 3D cell biology images, together with the bridging infrastructure that links into the DFrame. These components

forming a useful initial collection of functionality in a basic toolkit for the construction of practical parallelised 3D imaging applications.

Chapter 6 presents an integrated case study evaluating a dynamic pipeline of tasks comprised of 3D image operators. This clearly demonstrates the DFrame capability in terms of flexibility and adaptability at both the component level and the task graph level. Objectives include the selection of the best performing models suitable for a task according to the context in which a task is running, and the automatic adapting of partitioning strategies to optimise each task computation across multiple DFrame instances. Another prime objective at the task graph level is to automatically adapt resource allocation according to the characteristics of tasks and also through metrics captured during the running of the task graph workflows. The case study application focuses on extracting sarcoma cell invasion signatures from time lapsed sequences of 3D cell biology images, and results are presented that verify the technique, alongside the framework performance results.

Chapter 7 concludes with an assessment of the project, and provides insight into further works.

Chapter 2 Parallel Programming Languages and Libraries

2.1 Introduction

The motivations for harnessing parallel processing are becoming ever more compelling, with advances in scientific and business technologies fuelling a data explosion running alongside the trend towards ubiquitous and more economic parallel processing hardware. Despite the challenges, the decreasing cost of hardware is making the use of parallel processing more appealing, and whilst the cost of development effort remains high, parallel processing is becoming more prevalent. A key aim of this project is to reduce the barrier to entry to harnessing parallel processing, by developing a framework that will assist in the effort, with target applications in the domain of image processing. In order to make a reasoned decision on the technologies to use, the parallel processing arena must be reviewed in considerable detail.

This chapter first considers the fundamental approaches in computer architecture that support parallel processing. It is worth clarifying that while the term 'concurrency' describes those sections of a program that may be run in parallel, the term 'parallel processing' describes the extent to which the identified concurrency can be actually realized on specific hardware. Common architectures include the shared memory Symmetric Multi-processors (SMP) and Non-Uniform Memory (NUMA) systems, and the more scaleable distributed memory systems that are a composite of these, and elaborates on the categorisations in terms of creating and controlling parallelism and the appropriate process communication methods. These include the shared memory Symmetric Multi-processors (SMP) and Non-Uniform Memory (NUMA) systems, and the more scaleable distributed memory systems that are a composite of these, and elaborates on the categorisations in terms of creating and controlling parallelism and the appropriate process communication methods. The focus then moves on to look at the abstract modelling of parallelism, reviewing the diverse software models and programming paradigms developed to support these common parallel processing system architectures. It is also noted that while the parallel hardware architecture directly influences the models that must express their capabilities, models themselves can

influence the evolution of hardware architecture, often due to application requirements. A selection of languages and libraries is then reviewed that have evolved to support these models and parallel programming paradigms in order to efficiently write programs for these systems. These span from the lower level parallel programming languages that form the bedrock of parallel programming, to some popular higher level abstractions. This chapter review is an important part of the project, as the assessment of target system architectures, and the available software languages and libraries that support modelling parallelism on these architectures is of central interest to the project, in helping to determine the technology to adopt for a distributed framework suited to 3D image processing and high content screening.

2.2 Common Systems Supporting Parallel Processing

Parallelism is supported in hardware by the provision of multiple CPU's and associated memory and IO subsystems. Varied interconnect arrangements, hierarchical memory and multi-core and many-core processors have added complexity to these increasingly diverse system architectures. Specialised components such as GPU's add to the fray. Indeed, parallelism spans multiple levels from instruction level parallelism to distributed machines. Instruction level parallelism (ILP) is at the compiler level, and has many facets such as instruction pipelining, superscalar execution, speculative execution driven by branch prediction, and out of order execution. An account of instruction level parallelism and its limits can be found in (Wall 1991). Limits also exist due to the 'memory wall' effect (McKee 2004), where a disparity between processor speed and slower memory access speed can in the worst case clamp performance to the speed of main memory access. The core focus of this chapter contrasts how parallelism is arranged on shared memory and distributed memory architectures, but it is noted that these architectures are still crucially impacted by instruction level parallelism and memory wall effects.

Flynn's taxonomy (Flynn 1972) classifies computer architecture according to the number of instruction streams and data streams. Although rather dated, it is still popular due to it's simplicity, identifying only four classes. The single instruction single data (SISD) class is associated with the typical single processor machine, while the single instruction multiple data (SIMD) class maps to data parallel

systems such as the traditional vector processing machine. The multiple instruction single data (MISD) class is associated with pipeline architectures and the more general multiple instruction multiple data (MIMD) class maps to the most flexible and general (task) parallel systems. In the following sections the most common architectures in use today are reviewed, and these all typically map to the MIMD quadrant of Flynn's taxonomy (somewhat reducing its utility).

2.2.1 Shared Memory Systems

These comprise multiprocessor systems in which all the processors have access to a common address space. Systems that share memory must ensure efficient access to the shared memory whilst maintaining consistency. Uniform Memory Access (UMA) architectures arrange processors and memory such that memory access speed is similar for all processors. Figure 2.1 shows a simple Symmetric Multi-Processor UMA design. The interconnect is usually either via a bus or crossbar switch.

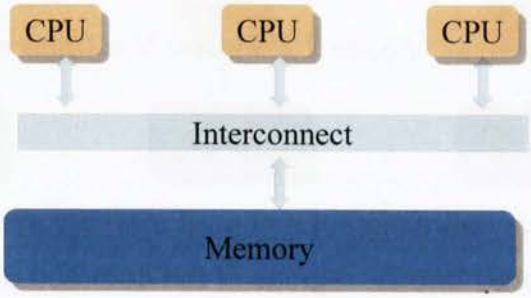


Figure 2.1: Typical Uniform Memory Access (UMA)

In modern systems, hierarchical memory subsystems have evolved to cater for the disparity in memory access speeds and CPU speed. System efficiency and scaling is further improved by using Non Uniform Memory Access (NUMA) architectures where processors have high speed access to local memory and lower speed access to remote memory. Algorithm designs can be more challenging as they have to consider data locality so that where possible, processors use local high speed memory to improve performance (Terboven, Schmidl et al. 2012). Complex cache coherence schemes are generally used to ensure that processors 'see'

consistent data (e.g. cache coherent NUMA or ccNUMA). Figure 2.2 shows a simplified typical NUMA arrangement which can be recognised as a logical scaling of the Symmetric Multiprocessor design. Although all CPU's can access both $memory_1$ and $memory_2$ as one address space, access speeds vary. For instance CPU_1 can only access $memory_2$ via a remote interconnect, which will typically be much slower than it's local interconnect to $memory_1$. Many other variants of the NUMA arrangement can be found in the literature (e.g. (Kumar, Grama et al.)).

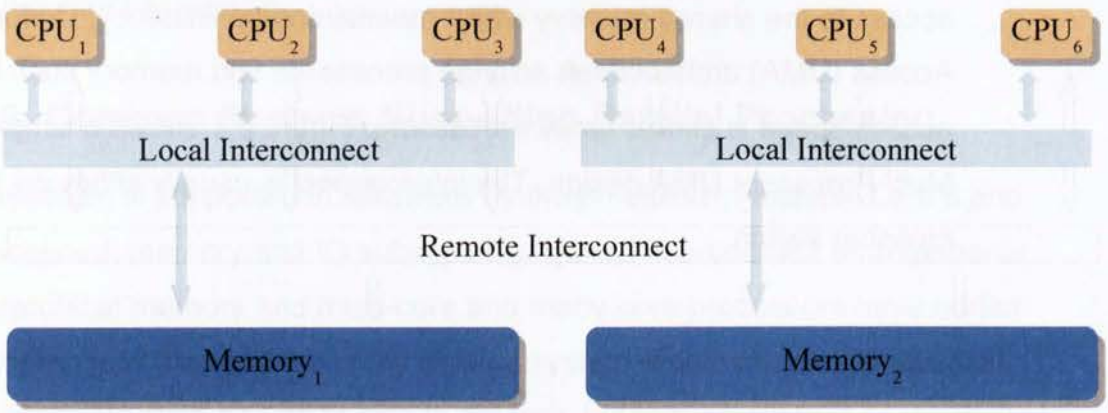


Figure 2.2: Typical Non Uniform Memory Access (NUMA)

A primary attraction of a shared memory system is that communication is implicit via shared 'critical' regions of the common address space, which avoids unnecessary data copying. However, synchronisation constructs must be used to manage contention, guarantee consistency and avoid introducing unnecessary blocks, race conditions and deadlocks. Synchronisation can complicate programming considerably, and requires great care and proficiency. Additional concerns such as managing data locality, avoidance of cache thrashing and false paging face the programmer (Jin, Li et al. 2001). Scaleability has historically been an issue with shared memory systems, as memory access bottlenecks can significantly degrade performance. Moreover, on 32bit systems, the maximum amount of memory that could be addressed was limited (this is not an issue with today's 64bit architectures). Shared memory systems typically rely on operating system support through 'process' and 'thread' constructs to express parallelism.

2.2.2 Distributed Memory Systems

In these systems, processors have their own memory address space as shown in Figure 2.3. There is no global shared memory and communication is via message passing between cooperating processes. Advantages of this architecture are that memory scales with the number of processor-memory units, synchronisation amongst processes is explicit and the cache coherence problem is eliminated. A programmer can avoid the complexity of managing concurrent accesses to local memory, and must instead arrange for the explicit communication between processes. The main challenges to scaleability in these systems are internode communication speed and network topology and reliability. Whereas in a shared memory system critical regions can inhibit processing, in distributed systems it is the explicit communication that can impede performance.

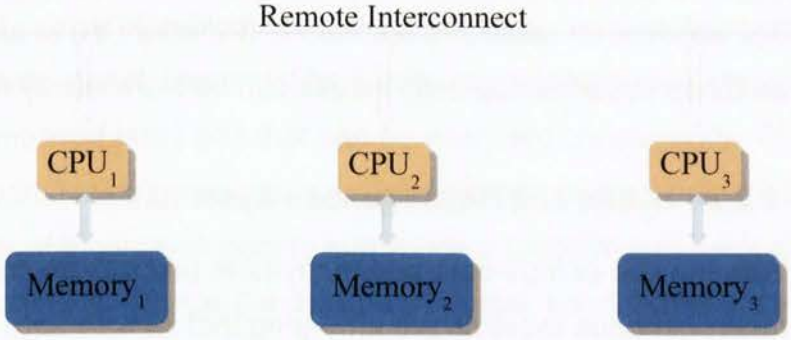


Figure 2.3: Typical Distributed Memory Access

Communication speed is generally much slower than computation, being impacted by protocol overhead costs, network latency and bandwidth limits, and can significantly affect the performance of a parallel application. Programmers rely on hardware support, and software optimization techniques to mitigate these communication overheads, such as using latency reduction (e.g. by simplifying protocols), optimising localisation to avoid latency, and prefetching or latency hiding where computation and communication are arranged to overlap. These optimisations can require judgements that depend on the nature of the problem. Of course, if processing is kept predominantly local, so that many independent processes execute on distinct data or execute distinct tasks, then interprocessor

communication can be much reduced. This is an extreme but not uncommon situation categorised as 'embarrassingly parallel processing'. Other common communication reduction scenarios include the distribution of tasks and data, followed by an independent processing phase, and a final results 'collection' phase (e.g. variants of the master/worker and map-reduce parallel models).

As processor counts increase, it becomes increasingly infeasible to fully connect them, and so more practical network topologies are sought that match an application's requirements. Rather than mandating a topology, application designs must often account for the expected or available target topologies (e.g. mesh, torus, hypercube, butterfly etc.). In moving away from a fully connected system, as well as latency and bandwidth, the designer must consider the message path between nodes (properties such as network diameter) (Kumar, Grama et al.). A topology that is suitable for one application may not be optimal for another, and it is interesting to note that the IBM Blue Gene architecture has two separate interconnects, a 3D torus for inter-process point-to-point communication, and a tree network for collective operations (Peterka , Yu et al. 2008). In modern switched systems, topology issues can be significantly reduced.

2.2.3 Hybrid and Heterogenous Systems

With the rise of muti-core and many-core processors and systems, hybrid and heterogeneous systems are emerging that include both shared memory and distributed memory technologies. Hybrid systems refer to the combining of shared and distributed memory models across multicore and many core systems, while heterogenous systems extend this to include general purpose GPU's (Al-Gharaibeh, Jeffery et al. 2012) . In a hybrid arrangement, the programmer must have expertise in both the message passing and shared memory models, and the technologies that implement them. In heterogenous systems, the parallel programmer must also acquire expertise in GPU programming. This is significant and in addition to domain expertise, and has prompted efforts to devise higher level abstractions that present a common model across these different lower level models.

2.3 Parallel Programming Models

Models (parallel programming or otherwise) are extensively used in software at

multiple levels to abstract, simplify, organise, generalise and idealise concepts. They provide guidance, clarity and also constraint. Generally, a good model should provide a high degree of abstraction, express important concepts, be simple and stable and sufficiently removed from the concrete architecture to be more robust and long lived when the architecture changes. Parallel programming models are numerous and diverse, spanning many different architectures and levels of abstraction (Leopold 2001) (note that in (Leopold 2001), abstract models that specifically focus on API's and algorithm design are referred to as paradigms). A very good recent survey of contemporary parallel programming models can also be found in (Diaz, Munoz-Caro et al. 2012).

In this section, attention is restricted to those core parallel programming *control models* and *memory models* that are fundamental to the clear expression of parallelism, and that are widely used as a basis for categorising parallel programming languages and libraries. These control and memory models tend to align with particular system architectures (see previous section), and so there is often a natural pairing of control and memory models. In order to arrange for parallel execution, parallelism must be identified and expressed. A computation must be decomposed into parts that can be executed concurrently. The predominant partitioning strategies are task and data decomposition. These are related aspects of a common goal to sub divide a problem into parts that can be executed concurrently. Hence the data parallel, task parallel and the related Single Program Multiple Data control models are specifically reviewed in the following sub-sections, and then the associated distributed and shared memory models, emphasising the impact on communication.It is not always possible to absolutely categorise languages according to one control model and one memory model, as some languages and libraries support multiple models. Hence the core models are outlined, and a review of languages and libraries implementing these models is separated out to a subsequent section.

2.3.1 Data Parallelism

Data parallelism considers how data can be partitioned, with each data partition then being operated on concurrently. Traditionally, pure data parallelism (SIMD in Flynn's taxonomy) implicitly mapped data to specialised vector processors. However, pure data parallel languages are too inflexible for general use, as each

recipient of data is constrained to execute the same instructions on the data in lock step. Another issue is that it is difficult to map the logical data parallelism with the available physical parallelism of current more general purpose architectures (this was not an issue on historical data parallel specific architectures). Nowadays, there is more interest in the looser definition of data parallelism, as the explicit partitioning and distributing of data for processing. Typically this is on non vector processors although vector capabilities are also available on some of these processors, such as Intel's SSE (Siewert). Some higher level languages do provide implicit support for general data parallelism via a *global view* of distributed data structures (e.g. Chapel - see section 2.5.6), and indeed higher level languages can be categorised according to whether they support a *local* or *global* view of distributed data structures.

2.3.2 Task Parallelism

Task parallelism considers whether the application can be broken down into independent subtasks that can be executed concurrently. Often a task dependency graph is required that defines which tasks can be run concurrently and which tasks depend on the results of other tasks. Once a program is broken down to express parallelism, other considerations must be dealt with such as task scheduling (onto processors), and efficient distribution of data into processor memory caches. This then requires coordination of communication and synchronisation to ensure efficiency and avoid race conditions and deadlock (i.e. an aspect of correctness not considered in serial computing). Although optimally mapping a program (task) graph onto a processor graph is known to be NP hard (Garey, Johnson 1997), sub optimal solutions are usually adequate. The speed up attained is dependent on optimising the load balance such that processors are assigned similar amounts of work to maximise concurrency, and by minimising extra overhead that is particular to parallel processing.

In the general case the runtime model can arrange for a static or dynamic number of processes to be available. In the static case the number of processes (and often threads) is fixed at program start up and tasks are mapped to processes that are already running. In the dynamic case, the number of processes (and threads) can be modified at runtime according to program requirements, for example using fork-join mechanisms. In some variants the number of processes may be fixed but the

number of threads per process is variable.

2.3.3 The SPMD Model

The Single Program Multiple Data (SPMD) computation model originally proposed in 1984 (Darema 2011) has a single program running on all participating processes. Although these processes collectively cooperate to execute the program, at any instance in time each process can run a different section of the program and operate on different data (c.f. Flynn's MIMD quadrant, of which this is a variant). The flexibility, and reuse of one program has proved to be a very practical parallel programming paradigm. It is particularly suited to distributed system architectures and is the core mechanism for creating parallel execution in many popular languages and libraries supporting these architecture (e.g. PVM(Geist , Beguelin et al. 1994)and MPI implementations). The SPMD model can also easily be implemented on shared memory systems as well, when a 'share nothing' approach is favoured. The SPMD model is essentially a static process model with one instance of the program running on each process, although the number of threads per program can be either static or dynamic. Hence, in comparison with the general task parallelism model, which is more usually associated with dynamic thread creation on shared memory systems, the SPMD model offers more user control of the physical parallelism.

2.3.4 Communication

The parallel execution of decomposed tasks and data generally implies a requirement for communication, such that information can be transferred between the separate processes and threads according to the particular parallel processing requirements. Alongside control models, languages are also categorised according to the memory model they support, as this fundamentally impacts the mechanisms that must be provided to arrange for communication. In shared memory systems, communication takes the form of updates to critical regions of a shared address space by multiple compute entities, whereas in message passing systems data is explicitly passed between its compute entities. These dominant communication strategies have different characteristics that require particular attention to ensure effective parallelisation, and impact the design of languages and libraries supporting them.

Shared memory systems that update critical regions of memory must arrange for synchronised serial access such that the memory is consistent to all processes and it can be quite challenging to maximize concurrency while avoiding race conditions and deadlocks. In shared memory systems, memory access is much slower than cpu usage so even here, memory accesses must be considered carefully. Indeed in NUMA arrangements, data locality is still very important.

Distributed memory systems must communicate using message passing, and this presents other challenges. Synchronization is explicitly specified in the communication statements, which can be helpful in making clear when control process interaction occurs (when information is being transferred), but still requires complex reasoning. The communication fabric is now very important as is the network topology. That data has to be copied from one address space to another imposes copy and transport overhead, as well as increased memory usage, and this can severely impact the success of a parallelised computation. For this reason, distributed systems operate more efficiently at a coarser grain, arranging for communication to be conducted with a smaller number of messages containing more data, rather than a larger number of messages container less data, and so suits algorithms that align more with a coarse grained approach.

2.4 *Theoretical Performance Considerations*

Although it seems intuitive that using multiple processors will lead to increased performance, Amdahl (Amdahl 1967) pointed out that this would be difficult in practise, due to irregularities in the computation of practical problems, and that 'housekeeping' would also add an unavoidable and significant serial fraction to parallel computations. This serial fraction places an upper bound on the speed up that can be achieved by increasing the number of processors used on a fixed size problem (ignoring super linear speed up and cache aggregation). Amdahl described the problem, and it's subsequent mathematical formulation is known as Amdahl's law. This sobering view was reframed in terms of problem size by Gustafson (Gustafson 1988) to demonstrate that linear speed up was attainable, and Yuan shi (Shi 1996) subsequently pointed out the equivalence of these approaches. Because Gustafson's law (also known as the Gustafson-Barsis law) reframes the speedup metric in terms of the accuracy of a computation that can be achieved in a fixed time (i.e. let the time remain constant and increase the number

of processors and problem size), it is sometimes referred to as 'scaled speedup'. Amdahl's law and the Gustafson-Barsis law are ideals that do not take into account the efficiency loss incurred in communication. The Karp-Flatt metric addresses this issue in what is called the 'experimentally determined serial fraction' (Karp, Flatt 1990). Another very useful metric is the 'isoefficiency metric' which can be used to compute the scalability of a parallel system (Grama, Gupta et al. 1993). This provides a function of the increase in speed up as the number of processors is increased. Efficiency can be maintained by increasing the problem size, but this may lead to memory problems if increasing the problem size more than linearly increases the memory usage. An excellent comparison of these various metrics can be found in (Quinn 2003). It is now generally understood that for many practical problems, the serial fraction can be very small, allowing near linear speed up. In practise, system architecture will also affect the performance calculations. For example, bandwidth and latency effects on communication speeds on distributed systems.

2.5 Languages and libraries supporting parallel programming

Significant challenges confront the programmer wanting to harness parallel resources. It is clear that assistance is needed, and a large number of programming languages have been developed over the years to help express parallelism, although the choice can add to the programmer's dilemma. Here a representative subset of languages that provide explicit support for parallel programming are reviewed. As outlined in the previous section, parallel programming languages can be categorised according to how they support the expression of parallelism such as the identification of parallel regions and how processes are created and managed (control models) and how communication and synchronisation are expressed and handled (memory models). This is particularly influenced by their intended target system architecture (see the *Architecture* section above). As well as using these models to broadly categorise parallel programming languages (at all levels of abstraction), programming style is also considered. Furthermore, many languages provide support for parallel processing through libraries, or a combination of language constructs and core support libraries. So alongside specific language constructs, languages that have core libraries supporting parallel programming are also considered, and specifications and libraries that are not core to a particular language. The review

extends to higher level programming abstractions where a shared memory programming paradigm is imposed on a physically distributed memory system using distributed memory systems (DMS) and Partitioned Global Address Space (PGAS), and where the message passing paradigm is imposed on shared memory systems (Actors). It is also noted in these sections that the lower level programming abstractions for shared memory and distributed memory systems commonly use the imperative programming style, as a core method of defining sequential behaviour. This style of programming (c.f. C/C++, Java, Pascal, Ada etc.) is composed of a sequence of commands that are executed one after the other, where order is crucial and memory management is explicit. Programming languages are commonly categorised according to the programming models or paradigms that they expect programmers to adhere to, the most common being imperative, declarative, functional, object oriented and process oriented paradigms, although the distinction is often blurred with many languages supporting multiple paradigms. Declarative and functional programming languages represent higher level abstractions, and will be briefly discussed in the context of parallel programming in a subsequent subsection.

Regardless of which language is used, the final processing will be a transformation into microcode suitable for a target machine architecture. This assumes a corresponding compiler is available that can convert a program to efficient machine instructions. A language must either have it's own compiler, or a program must be translated to a form that some appropriate and available compiler can parse and process. In any case, a compiler for a parallel programming language will to a lesser or greater extent incorporate the runtime infrastructure that implements the expressed parallelism, so it is critical that a compiler is available that can generate an efficient parallelised implementation. Although the reviews concentrate on languages that provide constructs for the explicit expression of parallelism, it should be noted that another area of *automatic* parallel programming research is in auto-parallelising compiler technology itself, where attempts to identify and generate parallel code is performed during compilation. However, this has had limited success, in part due to incomplete information available to the compiler with regard to ascertaining parallel regions (i.e. a lack of knowledge of the semantics of a problem), and has mainly been constrained to loop checking. To apply this technique, programs must still be serially described in the first instance.

2.5.1 Shared Memory Programming

Many languages and libraries rely on operating system support for shared memory parallel processing through 'process' and 'thread' constructs. A process has considerable set up overhead, as it must initialise and manage all it's resources and ongoing execution state. To set up multiple processes, a process can fork new processes, and again explicitly set up the environment of each forked process, including shared memory. Threads are lightweight virtual processes that run within the context of their creating process, and can utilise resources of the process, much reducing resource and lifecycle management overhead within a particular application. Typically, programs developed for the shared memory model implement parallelism using threads, placing the burden of managing non-determinism and access to shared memory critical regions on the programmer (Lee 2006). To illustrate the use of threaded programming, this section reviews two popular and established programming languages, C++ and java, and in particular, the libraries they use to express concurrency in shared memory systems.

C++ (Stroustrup 2000) is a superset of the C programming language (Kernighan 1988) , providing many additional facilities to aid the programmer, core amongst these being the concept of a 'class' to support object oriented programming, and 'templates' to support generic programming. However, C++ does not have direct language support for expressing concurrency, this functionality being incorporated via a library when required. *Pthreads* (Josey 2011) is the de facto standard low level thread library in UNIX environments (POSIX threading library IEEE 1003.1c-1995), providing operating system support for thread creation and management, together with facilities to control access to critical regions of shared memory . Considerable expertise is required to write correct and maintainable programs using pthreads. It should be noted that although the operating system will attempt to automatically assign threads to processors in an efficient manner, there is no direct relationship between the number of threads and number of processors, a point of concern to parallel processing performance (many operating systems do however provide system calls that can be used to determine the number of available processors and set processor affinity). Compilation of programs written in higher level languages and libraries can integrate pthreads under the covers, making this consideration still relevant. The 'Windows threading API' available on

the Windows operating system is broadly similar to pthreads, although there are significant differences. For example the API largely uses the one 'HANDLE' type , which the system resolves as required at runtime, trading programming simplicity at some cost to runtime performance.

Threads based programming is the assembly language of shared memory parallel programming, allowing the programmer to reason about sequential processes making blocking system calls {191 Tanenbaum,Andrew S. 2006;}. In this sense, they aid parallel programming, but conversely their unstructured nature is not so well suited to high performance computing development. Working at a higher abstraction, OpenMP is the defacto standard for shared memory explicit parallel programming in C/C++ and Fortran programs. OpenMP is comprised of a collection of compiler directives (pragmas), runtime library routines and environment variables (OpenMP 2011) . Whereas pthreads is low level and largely unstructured, OpenMP provides block structured constructs to organise parallel sections of code and particularly supports the expression of task and work-sharing centric fork join parallelism. At it's core OpenMP itself implements a task based master-worker model, spawning child threads to provide concurrent execution of tasks. An OpenMP preprocessing phase converts the compiler directives to thread parallel code. This generated code is preprocessor and target compiler dependent, but should be more consistent and performant. A programmer is thus relieved of this effort and the manual code is correspondingly reduced, improving maintenance, with openMP transparently managing thread creation, lifecycle, and synchronisation. Although tasks can be implemented and composed, OpenMP is primarily appropriate for loop based data parallelism rather than for the design of whole systems, being used to introduce parallelism incrementally to sections of existing code where thread based parallelism is appropriate. Also, code incorporating openMP pragmas and directives can still be run in a uniprocessor environment. A good introduction to OpenMP (open multi-processing) can be found in (Quinn 2003).

Intel Threading Building Blocks (TBB) (Reinders 2007) is a C++ template library for multi-core parallel programming. At a similar abstraction level to OpenMP, it provides a task based parallel abstraction above platform specific thread implementations. TBB provides iteration patterns (as templates - e.g. the tbb:parallel_for template function) for task based parallelism, transparently

managing the underlying threading details and complexity for the programmer. Whereas OpenMP is more a C based library, TBB is designed to be more aligned with the Object Oriented and template based approaches common to C++. Although at a higher abstraction than threads, the programmer still requires a significant understanding of shared memory parallel programming concepts such as synchronisation and barriers, to fully utilise the versatility of these frameworks, and in the case of TBB, skill in C++ templates. For an overview of these and similar frameworks with examples, see (Chen, Bairagi 2010) .

The java programming language (Gosling, Joy et al.) has become very popular, java programs being compiled to java byte-code that runs on a java virtual machine (JVM). As such, java programs can run on any platform for which a JVM has been written. In this sense, java programs adhere to a 'run anywhere' philosophy (i.e. anywhere a JVM is available). Although designed with a syntax similar to C++, java removes some of the more controversial aspects such as multiple inheritance, and includes automatic heap memory management through a garbage collection mechanism, thus removing a time consuming and error prone aspect of programming. Java has included thread support in its core java.lang package since JDK 1.0. A programmer arranges parallel execution by creating and starting Thread objects (Lea 2000b) . Synchronization mechanisms are provided to arrange safe communication across shared memory. The code that the thread executes can be implemented within the run method of the thread or any object implementing a *runnable* interface can be passed to the thread as the code to run, prior to starting the thread. As with pthreads, the arrangement for concurrency will not necessarily translate to parallel execution, as this depends on the resources available and the underlying process scheduling. As well, being low level constructs, java threads have the same pitfalls as pthreads in terms of lack of structural support. However, a java.util.concurrency package is available (since JDK 1.5) that provides more high level threading support through facilities such as an ExecutorService, Executors, Futures and Callables that handles much of the details of managing threads (e.g. creating and managing thread pools). Indeed a whole armoury is available to the concurrency programmer that also includes blocking queues, synchronous queues, countdown latches, semaphores etc. to assist in concurrent programming. Furthermore, in JDK 1.7 a fork-join framework has been added (Lea 2000a) , that is more tuned to parallel processing rather than concurrent programming, the core implementation incorporating a work stealing

algorithm (borrowing ideas from the Cilk programming language). This framework is intended to assist parallel programming by imposing a more structured approach to creating and controlling parallelism, and tuned to using multiple processors (still using worker thread pools under the covers). A main usage being to express the 'fork-join' pattern commonly used in recursive divide and conquer algorithms.

2.5.2 Message Passing

The message passing paradigm is a natural fit for distributed memory architectures, and languages targeting this architecture employ message passing as the base abstraction. CSP (Hoare 1978), (Hoare 1985) was an early and influential attempt to apply rigour and theory to processes, process composition and communication (a 2004 version of the book "Communicating Sequential Processes" is also available on line at http://www.usingcsp.com/cspbook.pdf). In CSP, sequential processes execute on their own variables and only communicate by sending and receiving messages across I/O 'channels' ('distributed assignment' semantics) with non determinism being handled using guarded statements. The original CSP is now known as Theoretical CSP, and has influenced the design of many imperative distributed programming languages, Occam (May 1983) being it's most well known direct language descendant. Occam was originally developed by INMOS for transputer systems, the design of which meshes well with message passing communication. The CSP and Occam approach is also referred to as 'process oriented programming'. Occam-pi (Welch 2012) is also an important area of research, extending Occam to include features from the pi calculus (Milner 1982), (Milner 1995), (Milner 1999). The 'Go' programming language provides a contemporary example using the channel paradigm to send and receive messages (Doxsey 2012).

In the past, parallel hardware vendors typically implemented highly optimised message passing libraries for their systems, which were fast but not portable. A highly successful portable library called the Parallel Virtual Machine (PVM) was conceived in 1989; gained huge popularity throughout the 90's and is still popular today as a message passing system to hook up disparate heterogeneous computer clusters, provide dynamic reconfiguration and some fault tolerance (Geistet al., 1994). However, the proliferation of different message passing

implementations continued to raise portability concerns, and the need for some standard was generally realised. This led to the development the Message Passing Interface specification.

The Message Passing Interface (MPI) (Gropp, Lusk et al. 1999a) is the assembly language for writing portable and scaleable applications for distributed memory systems using the message passing model, explicitly specifying the communication protocols for point to point and collective communications. MPI introduces the concept of a communicator which provides a context to multiple communications and is also suited for library development and fault tolerance (Geist et al., 1994). It also provides a feature for an application to specify topological information. Version 2 (MPI-2) adds dynamic processes, remote memory access, and parallel I/O capabilities (Gropp, Lusk and Thakur, 1999), but not process migration among nodes. MPI is a library based approach targeting distributed memory systems, so data copying and buffering now become prominent as impediments to performance. The optimum grain size of an application is influenced by the cost of communication. Close coupled systems can support finer grain size, and more loosely coupled systems work better at coarser grain sizes. For this reason, MPI is more suited to coarse grained algorithms, because fine grained data copying and transfer generally impacts performance. The specification includes bindings for C and Fortran. Bindings for C++ were added in version 2, but later deprecated and removed in version 3 as these added maintenance overhead to the specification with little advantage over the C bindings. As C++ can use a pthreads library to express threaded programming, it can use a library implementing the MPI specification to arrange for distributed programming using the message passing model (using that library's C bindings).

Although targeted at distributed architectures, the message passing model can be used on shared memory machines, to leverage the model's advantages and to provide consistency across hybrid systems. This can be simply achieved by programmatically partitioning shared address space into disjoint parts and programming using MPI. That processes are only visible to the outside world via the messages they send and receive aids software modularity in composing larger programs.

Java has already been mentioned as a programming language suitable for concurrent programming on shared memory systems, but also has considerable

library support for distributed programming. In particular, java remote method invocation (RMI) offers advanced library support for distributed programming, that hides much of the low level details behind remote object interfaces. Through this mechanism, code and data can be passed to remote instances either by reference or by value, providing a very powerful distributed programming abstraction. However, java objects are marshalled and unmarshalled automatically using a rather heavyweight protocol, so RMI is more normally associated with distributed computing, although there have been efforts to adapt and improve for efficient parallel processing (Maassen 2001).

2.5.3 GPU Programming

The trend towards parallelism through multi-core and many-core processors has also led to an interest in harnessing the processing power and vectorising capabilities of GPU's for general purpose processing (GPGPU: General Purpose GPU's). Indeed, complex heterogenous systems mix shared memory, message passing and GPU processing (Liang, Li et al. 2012) . Vendor specific parallel programming kernel type languages such as CUDA (Farber 2011) , and more recently the open standard heterogeneous openCL platform are garnering more interest (Gaster, Howes et al. 2013) , to incorporate GPU resources and harness resources across heterogeneous systems.

2.5.4 Distributed Shared Memory Languages

The increasing prevalence of hybrid architectures having both distributed and shared memory led to the development of distributed shared memory (DSM) models. Instead of using multiple technologies such as MPI, pthreads and openMP to develop parallel programs, these models aim to provide one consistent logical shared memory model across distributed systems. The underlying movement of data is managed transparently by the system, easing the programmers task. In relieving the programmer's burden, these systems must solve fundamental issues that include replication of data across nodes and maintaining data consistency and coherence (Li, Hudak 1989a). These early systems had no concept of data locality, impacting program efficiency. Porting shared memory programs to a DSM model to leverage the scalability of distributed systems should be straightforward. There are numerous DSM implementations,

examples include IVY (Li, Hudak 1989b) and Mirage (Fleisch, Popek 1989). A number of other implementations, along with the viability and issues associated with DSM are surveyed in (Nitzberg, Lo 1991).

2.5.5 Partitioned Global Address Space Languages

The PGAS languages similarly define a global address space over distributed memory architectures, again aiming to provide one paradigm across shared memory and distributed memory architectures. As such, PGAS can be considered a memory model, often associated with an SPMD control model. These higher level languages also allow shared memory semantics to be used across distributed systems and extend the distributed shared memory model by logically partitioning the global shared memory to exploit processor locality. As popular examples, Co-Array Fortran (CAF) and Unified Parallel C (UPC) extent the C programming language to implement the PGAS programming model. CAF and UPC both follow the SPMD model with all processes starting and finishing together (c.f. openMP declarations within one process spawning other processes dynamically), and have constructs to allow synchronization amongst processes (processes are referred to as threads in UPC and images in CAF). Where CAF and UPC differ is in their approach to managing distributed memory. CAF uses a special dimension called a co-array to reference memory across images, and so variables declared with a co-array must be replicated in shape and size across all images. In this way, images can reference global memory using logical co-array indexes. UPC declares distributed arrays using a shared keyword and blocking factor qualifiers, and accessed using shared pointers. In this approach the distributed memory extents does not have to be the same across all the UPC threads. Somewhat incongruous with the SPMD model, UPC also supports a upc_forall loop (c.f. openMP). Both CAF and UPC usually rely on remote direct memory access and global address space support (e.g. the GASNet communication layer (Bonachea 2002)). An interesting comparison of UPC with MPI collectives can be found in (Nishtala, Zheng et al. 2011), and a discussion describing a hybrid MPI/UPC model in (Dinan, Balaji et al. 2010).

Titanium (Yelick, Hilfinger et al. 2007) is another PGAS language worthy of note, being an explicitly parallel dialect of java that supports the SPMD model. Titanium extends java for parallel scientific computing, including support for multi-

dimensional arrays, immutable classes, cross language calls and templates, and introduces the concept of regions as a memory management alternative to garbage collection, which is difficult to implement on distributed systems.Titanium is a local view language with distributed arrays being built from local sub-arrays and global pointers distributed through an exchange operation. Local and global pointers implement locality control in Titaniums PGAS memory model, using the local pointer declaration qualifier.

2.5.6 HPCS Languages

In 2002, DARPA initiated it's High Productivity Computing Systems (HPCS) program to rejuvenate and encourage the HPC industry to improve efficiency, productivity and performance (Dongarra, Graybill et al. 2008). The X10, Chapel and Fortress languages (Weiland 2007), developed as part of this initiative received substantial support from DARPA, with the more prolonged support for X10 and Chapel. Although Fortress is an interesting HPCS language developed by Sun micro-systems, that uses a more mathematically inclined symbolic syntax, the review below focuses on X10 and Chapel, as these emphasis the salient points of the HPCS initiative, to partition but provide a global view, with the facility to incorporate locality information.

X10 (Charles, Grothoff et al. 2005) is a PGAS language developed by IBM, targeted at non uniform SMP clusters. A core design goal being to increase productivity for scaleable parallel programming with both distributed and shared memory paradigms being spanned by the one higher level language. The syntax of the language is designed to be accessible to java programmers with functional programming features (single assignment, mutable shared memory but no mutable global memory) . Concurrency in X10 is expressed using the 'async' keyword to spawn processes, with the 'finish' keyword controlling concurrency extent, where condition and reduction operations can also be specified. Phased computations can also be introduced using the clock functionality which acts as a generalised barrier. X10 recognises the significant difference in shared memory and distributed memory communication performance, and provides explicit programming support to express processing locations using the concept of 'places'. Under the covers, place shifting using '*at(place) <statement or expression>*' is implemented by sending a message to another process, the

processing is then performed at that location. Meanwhile the sender suspends awaiting the response. So X10 encourages the programmer to consider where the data is, and rather than accessing remote data directly, the programmer can move processing to the data using the places abstraction. At one place, using shared memory, the '*atomic*' keyword is used to delineate statements that must be performed atomically, with respect to other atomic statements going on at that place (lock free synchronisation). This design maps well to hardware 'transactional memory', which was the X10 designers' intent. However, if transactional memory is not available, then atomics are implemented using one global lock, the serial acquisition of which can cause scaleability issues. To circumvent this, a library is available that provides low level locks and ancillary support (a failure of the language in this respect). X10 programs can be compiled to C++ (native) or to java (managed runtime), although the java target is currently restricted to one place. When an X10 program is run, the number of places is stipulated. Scheduling of activities is implemented using a work stealing algorithm, borrowing ideas introduced into the cilk programming language (Blumofe, Joerg et al. 1995)(Joerg 1996), the benefit being that each spawned process has it's own queue.

The Chapel language developed by Cray is an imperative, block structured language with a multithreaded programming model supporting task and data parallelism, concurrency and nested parallelism. It is a new language, although the base syntax draws from C, Java Fortran and Ada, together with 'modules' from Pascal and Modula. The design is guided by the objective to align with the experience of the HPC community, to ease transfer of skills. It specifically incorporates data parallel features from ZPL (Chamberlain, Choi et al. 2000) which itself is a sublanguage of the Orca family of languages (Bal, Kaashoek et al. 1992), and is also influenced by High Performance Fortran (HPF) and the Cray MTA™/Cray XMT™ extensions to C and Fortran (Cray Inc.), and targets development on both high performance computers and commodity clusters. A core design goal is to improve productivity, whilst matching the performance and portability of lower level models such as MPI and OpenMP (Chamberlain, Choi et al.), (Chamberlain, Callahan et al. 2007) (the name Chapel is derived from 'Cascade High Productivity Language). Chapel supports tuning for locality using the locale type which maps to a unit of uniform memory access such as an SMP node, giving the programmer control over placement of tasks and data (c.f. X10's places), and also provides global multidimensional array distributed data

structures. These global view abstractions encourage thinking about the whole problem rather than how to fragment it, and can significantly simplify programming (i.e. Chapel is a 'global view' language rather than a fragmented/local view). Associated with these constructs, 'domains' are a feature core to implementing data parallelism, defining index sets for array definition, manipulation, reallocation and distribution. Domains can index distributions across a set of locales. Task Parallelism is supported through *forall*, *begin* and *cobegin* parallel statement constructs, synchronisation being expressed with *single* and *synch* variables. Chapel supports a novel multi-resolution philosophy, so that programmers can span a programming spectrum from the very abstract high level distribution statements, down to machine level statements using the 'on' clause for task placement.

2.5.7 The Actor Model

The Actor model (Hewitt, Bishop et al. 1973) is an inherently concurrent model. Actors are 'share nothing' compute entities that communicate with each other using the message passing paradigm. The high level of abstraction is attractive to programmers, as it removes the necessity to deal with low level constructs (e.g. threads, synchronization and I/O). Upon receipt of a message an Actor can undertake some local computation, alter its state, create other Actors and communicate with other Actors synchronously or asynchronously using a mailbox metaphor. Actors are commonly implemented as, or mapped to separate processes or threads. This simple specification provides a very powerful model of scaleable concurrency, and aids reasoning about the inherent complexity of indeterminate parallel systems. Although introduced more than 40 years ago, the Actor model is arguably more relevant today due to the increasing ubiquity of multicore and many-core systems.

The Erlang functional programming language (with some declarative features) has direct support for concurrency, distribution and fault tolerance, with the Actor model being built into the language itself. In Erlang, Actors are 'spawned' as processes. Evolving within the telecoms industry, Erlang is designed to support large scale concurrency. Although Erlang's emphasis is on distributed processing and fault tolerance rather than parallel processing speed, it has influenced a number of other languages including some whose objectives are more focus on

performance.

Scala (Odersky, Spoon et al. 2010) is a contemporary object oriented language that also supports the functional style of programming and provides library support for Actors, the implementation being influenced by Erlang's functional programming style. Scala programs are compiled to byte-code that can then be run on a JVM (Java Virtual Machine). In Scala, primitives and statics are absent, the prime construct being an object. Scala actors provide a high level abstraction for concurrency based on message passing, as an alternative to more complex low level thread programming that requires mastery of synchronisation mechanisms to manage communication and data integrity amongst competing or cooperating threads. Any thread signalling and coordination being hidden within the Scala concurrency package. Scala is interesting in that it provides support for both thread based and event based Actors. A thread based Actor maps directly to a thread of control, with state being maintained on the thread's stack and communication is via send and receive messages. However, when an application must scale to thousands or millions of Actors, it is more appropriate to implement Actor's based on a 'react' event based style with closures for continuation. Despite increased complexity due to the 'inversion of control' in an event based system, the Actor implementation is much more lightweight, implemented atop a thread pool, with state now stored in continuation closures rather than stack frames.There has been considerable debate on the relative merits of thread based and event based models (Lauer, Needham 1979), and Scala attempts to unify these competing styles through Actors (Haller, Odersky 2007).

Akka (Kuhn, Antonsson et al.) is an excellent Actors library for highly concurrent systems in Java and Scala that also uses the Erlang style Actor model, providing a platform for writing concurrent, event driven, scaleable, fault tolerant applications. Asynchronous non blocking message passing scales across multicores and multiple nodes. Like Scala, Akka compiles to byte-code that runs on the JVM, and can thus integrate with Java and Scala.

2.5.8 Functional and Declarative Programming

The functional programming paradigm composes programs as a hierarchy of function expressions, providing an elegant alternative to the imperative style of programming (Backus 1978). Theses programs are largely stateless, they work

with immutable data and can be very concise, Lisp, Scheme and Haskel being amongst the more well known examples. Erlang and its descendant Scala also support the functional programming style. Although the functional programming paradigm appears to align well with parallel programming, it has not enjoyed the expected success in this arena. Although stateless function expressions map well to parallelisation, the parallelism is implicitly fine grained and load balancing is hard to predict, since it is not generally know how long any particular expression would take to execute. Indeed, at certain scales, the cost of setting up for parallel execution, and subsequent communication may outweigh the gains from concurrent execution itself. This inability to infer a suitable grain size of parallelism has hampered successful use of functional programming languages for parallel processing. As well, the high level abstraction means that programmers do not have direct access to control the parallelism (c.f. the more flexible imperative/procedural languages).

The declarative programming model puts the focus on what to do, rather than how to do it. The most ubiquitous example being SQL, which states what to do in order to retrieve data from a database in concise, easy to read and understand declarative statements. A database implementation transforms these statements into an execution plan (which can include concurrent processing). Again, this style of programming is at a high level of abstraction, and so the user must usually rely on the underlying implementations to find and impose any parallelism (which obviates any direct low level bespoke programming to improve efficiency), trading ease of use and increased productivity against less control.

2.6 Reliability and Fault Tolerance

Not much has been said about reliability and fault tolerance, but in the multi-processor world this becomes more relevant. With increased processor counts, the probability that any one processor will fail increases. A failure of one processor participating in a computation across a group of processors could mean the loss of results across all the participating processors. Often this is accepted as an inconvenient but infrequent event and if encountered processing is just restarted. However, having multiple processors does also offer an opportunity for the programmer to weave redundancy into algorithms to account for local faults or simply to duplicate processing of the same algorithm, an extra burden for the

programmer but in certain circumstances it is worthwhile. One simple approach is to provide checkpointing, where intermediate results are continuously stored so that if a failure occurs the processing can resume from a check point rather than from the beginning. One thorny issue here is that the programmer must decide how much data and processing replication is required (e.g. in order to reduce the risk to that of a single processor). In other cases, it is left to the container to manage faults, but even here an algorithm would usually need to interact to resume execution, meaning application software must still be 'fault tolerant aware'.

Systems can also be categorised according to their stance on fault tolerance and reliability: one differentiator between parallel processing (high performance) and distributed systems may be the usage: parallel processing emphasis on speedup, whereas distributed systems emphasis reliability (fault tolerance) and resource sharing.

2.7 Summary

Today, the dominant scaleable high performance computing systems typically comprise clusters of multi-core and many-core nodes, where internode communication uses message passing and intra-node communication can use message passing or shared memory. Vector processing capabilities and the use of general purpose graphics processing units add further variety. The resultant complex mix of scaled in and scaled out parallelising capabilities complicates how to efficiently utilise these technically advanced systems. The task of this chapter has been to provide a critical review of the more typical systems, and to highlight various parallel control and memory models used to categorise them and help to understand and manage the complexity. The chapter then goes on to consider a representative selection of programming languages and libraries that have been developed to support parallelism within the context of these systems and models. The goal of such efforts being to elucidate the merits and limitations of the various programming approaches, so that an informed decision can be made as to what approach may be best suited to the task of applying parallel processing in the image processing domain, specifically the desire to leverage these systems to parallelise multiple complex 3D image processing algorithms.

It is apparent that message passing cannot be avoided, being required to accommodate internode communication. When appropriate, using shared memory

is attractive, avoiding main memory copy overhead (although all systems must still consider cache memory behaviour). However, scaleability has proved problematic, and the burden of managing access to critical regions of memory using explicit synchronization and lock management can be more onerous than the implicit synchronization afforded by message passing. It is of course a contentious debate as to whether synchronizing via message passing or across memory critical regions is the more difficult. Higher level languages were reviewed that attempt to provide a common programming model across shared and distributed systems, the latest evolution of such languages (PGAS/HPCS) incorporating mechanisms that allow the programmer to cater for locality of data, an admission that this information is often required in order to write efficient parallel programs. Recently, interest has also been rekindled in languages that enforce a message passing only style (Actors), illuminating the trend back towards the 'messages only' style. In any case, it is evident that even these higher level languages are still quite complex, and require considerable expertise in parallel processing, including knowledge of the underlying systems, to be used effectively. Although a large improvement on the more lower level programming languages, the assumed goal to provide tools that simplify parallel programming is thus blunted to some degree.

Deliberating on this review, in the context of an image processing requirement, it is evident that the low and more high level languages that support both message passing and shared memory systems would be suitable candidates for parallelising image processing applications. However, none would be sufficient in themselves to provide complete transparency to the image domain (application) programmer, and thus some framework would be required to bridge the gap. Such a framework could be built using any of these languages, but the very need for such a framework to separate the parallel processing from application domain processing means that the more powerful and low level parallel processing approach can be retained. The assumption being that experts in parallel processing would develop and maintain this aspect.

To summarise, the extensive review of current parallel systems and programming language and library support was motivated by a desire to determine the implementation strategy of a parallel processing framework, and to that aim assess the various approaches to arrange for parallel execution and gain an in depth appreciation of the issues and various solutions. Ultimately, a more low level

approach is chosen, specifically a combination of C++ and MPI. This being driven by the above considerations but also other aspects including extensibility and future proofing (C++ is a popular broad market language), performance and the intended distributed target architecture. As well, MPI is the de-facto standard message passing specification, such that both C++ and MPI will appeal to a large class of users, and avoid learning new languages whose future may be uncertain. In the next chapter, the review moves up a level to consider parallel programming frameworks expressing parallel programming patterns, setting the scene for the introduction of the proposed distributed framework, build using C++ and MPI.

Chapter 3 Parallel Programming Frameworks

3.1 Introduction

Languages that support parallel processing form the generic underpinning of development for programs harnessing parallelism. Their utility is in providing abstractions that the programmer can use to express and manage parallelism in a program. As generic tools, these languages cannot provide help at higher levels that are domain specific, and this is the responsibility of the programmer, using an appropriate language to craft the program requirements. This is an arduous exercise and it is good software practise to consider extracting common patterns of use where possible, and implement such functionality in reusable components and frameworks.

This chapter reviews patterns of parallel processing and their combination with components to form useful parallel programming frameworks. The definition of a pattern is introduced as the description of a recurring problem together with some prototypical solution, together with a brief review of some development methodologies that leverage patterns and the idea of pattern languages. Consideration then turns to the concept and utility of frameworks, and a review of various frameworks designed to support patterns of parallel processing. Template based and auto generative frameworks are discussed, and frameworks that are constrained to model specific patterns. The review then goes on to look at data flow and streaming frameworks, and leads into composition and workflow frameworks. Finally, some representative application specific frameworks are reviewed in the interesting evolutionary algorithm domain and the project's target image processing domain. The common goal of such frameworks being to simplify and assist the development of applications that can benefit from parallel processing. Although no one framework was deemed entirely suitable as a distributed framework for the target image processing applications, the review is important in assessing aspects of these frameworks that would be helpful. In subsequent chapters, a variant lightweight framework is proposed, whose architecture incorporates many of the positive aspects of the reviewed frameworks, and allows for simple extension of the framework itself. The impetus for such a framework initially emerged from consideration of image processing

algorithms and applications, although the framework can be otherwise generalised.

3.2 Patterns of Parallel Programming

In making sense of the world, people use their pattern recognition capabilities to categorise and simplify the vast stream of information continually presented to them. By the same token, identifying recurring patterns is useful in software engineering. A pattern encompasses the description of a recurring problem in some domain, a context in which it is relevant, and a known prototypical solution that has wide acceptance amongst the domain experts (this format is particularly standard to the software industry). Introduced by Christopher Alexander in the context of building architecture (Alexander, Ishikawa et al. 1977), the concept has subsequently been adopted as the 'lingua franca' for communicating design reuse in object oriented software development, popularised by the seminal book 'Design Patterns' (Gamma, Helm et al. 1995). Patterns emerge from experience within a particular domain, encode best practise and facilitate communication and reuse. It is common for many patterns to be identified, supporting various recurring problems within a domain. Many of these patterns are related, and indeed an application solution is often composed of many interconnected patterns. The composition of patterns may itself form a reusable pattern. Consequently, patterns can be usefully further organised into taxonomies and pattern languages, with languages evolving over time to incorporate new patterns as they are identified. Subsets of a pattern language are often also relevant across a number of domains. In (Züllighoven 2004), a simple taxonomy of patterns is described that identifies *application* domain patterns as high level "conceptual patterns", *design* patterns that form the "micro architectures for software construction", and *programming* patterns that map to source code constructs and idioms.

The project's particular focus is on patterns pertinent to parallel processing. Parallel processing is considerably more complex than its serial counterpart, and identifying patterns here has the potential to significantly improve development productivity. Analysing the potential parallelism in an application is of course crucial. Identifying concurrency and its characteristics, and mapping to appropriate algorithmic structures and implementations is core, the premise being that patterns can significantly assist in this effort. In 'Patterns for Parallel Programming'

(Mattson, Sanders et al. 2004) , a pattern language for parallel programming is proposed. The language is organised into four design spaces that delineate specific temporal stages in the development of a parallel program. The *finding concurrency* design space includes patterns that aid in the initial identification of exploitable concurrency. This is a significant first analysis step, with domain knowledge being required to discern and map to appropriate patterns such as task and data decomposition. Next, an *algorithmic structure* design space includes patterns that are suitable in the primary refinement and expression of identified concurrency, such as task parallelism, divide and conquer, geometric decomposition etc. A subsequent *supporting structures* design space contains patterns that allow the organisation and mapping of identified concurrency into programmable constructs. Here is found familiar patterns such as SPMD, Master-worker, Loop parallelism, Fork-join, Shared data and Shared queue. Lastly, an *implementation mechanisms* design space presents various choices that can be used in implementing a parallel program, such as threads, processes, synchronisation and message passing. The categorisation of 'design spaces' within this pattern language present an overarching temporal pattern that outlines a methodology in developing a parallelised application.

The Parallel Computing Laboratory at Berkeley is at the forefront of research into application development techniques that facilitate the use of parallel processing (Su, Catanzaro et al. 2009), and a core part of this effort is in defining its own pattern language (Catanzaro, Keutzer 2010), (Keutzer, Mattson). Their approach is to study specific applications, explore algorithmic changes and implementation choices and 'discover' and 'lift' reusable patterns. Their pattern language places more emphasis on the hierarchical levels of patterns (and also division between application and programming frameworks - see the frameworks section). *Structural* and *computational* patterns are at the highest level, respectively describing patterns that concern the organisation of an application (coarse grain) and the computations within that organisation (fine grain). This domain centric top level, broadly architects the application structure and does not in itself mandate parallelism. At the next level are *algorithm* strategies whose aim is to exploit concurrency (e.g. where a higher level computation pattern would guide the choice of algorithm strategy). Next are *implementation* strategies describing patterns realised in source code. A useful distinction is made at this level between *program structure* patterns and *data structure* patterns. At the lowest level are *parallel (or*

concurrent) execution patterns such as thread pools and message passing strategies, and this level is also usefully partitioned into *program counter* patterns and *coordination* patterns. A premise of this pattern language being that a full understanding of a problem or application will allow the extraction of core structure, enable identification of patterns of parallelism, which will then facilitate mapping to suitable parallelisation strategies (hence the group's focus on study of applications. c.f. the finding concurrency design space above).

The pattern languages reviewed above differ in their emphasis on temporal and structural characteristics. At a higher level, Berkeley's patterns are influenced by architectural style and computation patterns. At the lower levels, Berkeley's classification also incorporates ideas from the earlier proposals in 'Patterns for Parallel Programming'. Despite their significant differences in content and emphasis, some aspects of the *Structural* and *computational* patterns map into the *finding concurrency* and *algorithmic structure* design spaces. The *algorithm* and *implementation* strategies straddle the *algorithmic structure* and the *supporting structures* design spaces, and the *parallel execution* patterns map closely to the *implementation mechanisms* design space. A complementary review of the above can also be found in the thesis (KEKEC 2010) , which goes on to use the design spaces methodology to design and implement a parallel processing example (dynamic list management) on a multi-core processor based real time embedded system.

3.3 Frameworks Overview

Software Frameworks are closely related to patterns. Frameworks are an object oriented reuse technique that combines the concept of components and patterns (Johnson 1997). In (Züllighoven 2004) they are described as "prefabricated software structures that usually implement design patterns...". Frameworks are the composition of component building blocks and commonly implement concepts described using patterns. As such, frameworks are more concrete than patterns, providing code that realises the core infrastructure of an application, usually as abstract classes, and defining the intended interaction of these classes, or control flow of a program that uses the framework. Users then extend the framework classes to implement specialised applications. A framework can be implemented within a library that a user program calls, and in this case the user is responsible

for ensuring the interactions that the framework defines. However, in order to enforce the control flow, *inversion of control* is commonly employed, where the framework drives the program and users of the framework plug in specialised functionality pertinent to their requirements. In this arrangement, the framework calls or drives the 'plugged in' components. Inversion of control is a common characteristic rather than a defining feature of a framework, often used to further distinguish them from libraries (where a library contains code that a user calls).

Of particular interest in this chapter are frameworks that support parallel programming. In an influential early work, Cole introduced 'algorithmic skeletons' ((Cole 1991)). Similar to frameworks, skeletons implement patterns to provide a structural outline to manage specific parallel computations. A user adds implementation to fill out the skeleton to produce a specialised solution. Cole goes on to consider four example skeletons: “fixed degree divide and conquer”, “iterative combination”, “clustering” and “task queues”. Cole argues that the additional constraints imposed by a skeleton, as compared to general programming languages, guide the user to a suitable solution, and deters inappropriate implementations. He uses the notion of an abstract machine based on algorithmic skeletons, as a higher level abstraction to aid development. Similar to the earlier discussion on patterns, Cole makes the point that skeletons can extract and record best practise from programmers' experiences (this point extends to frameworks in general). Cole acknowledges that the introduction of a level of abstraction can impact efficiency, whilst pointing out the advantage that the abstraction will have in separating the application programmer from the detailed skeleton implementations. Also, see (González-Vélez, Leyton 2010) for a recent survey of algorithmic skeleton frameworks.

In the following sections, a number of representative parallel programming frameworks are reviewed.

3.4 Parallel Programming Frameworks

Frameworks can be categorised by their level of abstraction, where generally, higher level frameworks trade productivity for efficiency. Frameworks at a high level of abstraction can completely shield the programmer from the complexities of parallel programming, but by the same token can remove access to potential low level machine or architecture specific tuning efficiencies. Conversely, lower level

frameworks can provide such flexibility, but require much more specialised parallel and architecture specific programming skills. In this chapter, interest is primarily focused on higher level abstractions that aid application programmers in harnessing parallel processing resources more transparently, but it is acknowledged that these abstractions are commonly built on lower level more explicit languages and libraries that support parallel processing models and frameworks that were covered in chapter 2. The separating out of levels of abstraction is a common and very useful practise, allowing specific expertise to be brought to bear at each level. For example, application programmers can use the higher level abstractions of parallel models and frameworks and thus be more productive developing domain code, whilst parallel programming experts concentrate on implementing the orthogonal low level parallel programming part.

At a very high level, parallel processing can be embedded within a domain specific application framework. Here the parallelism is implicit and completely transparent to the programmer, who has only to implement any required domain specific specialisations. However, this removes any opportunity to reuse the parallelising part of the code, which may be largely orthogonal to the application framework. Still at a high level of abstraction, instead of embedding parallel processing within a framework specialised for a particular application, a generic parallel framework can be employed. In this arrangement, a separate framework provides the required parallel support structure required by the application, but does not tightly integrate with a particular application. These more generic frameworks can be useful across a broad range of applications, and this increased reuse can result in a more tested and robust framework. They can still be embedded in applications but through distinct interfaces.

3.4.1 Template Based and Auto Generated Frameworks

Rather than providing an already implemented generic and plugin extensible framework, another approach is to have a required specific framework automatically generated from appropriate parallel pattern specifications. A pattern template repository or library then provides a collection of templates for the known parallel patterns. Extensibility is supported by being able to add new patterns to the library. An example of this approach is the Parallel Design Patterns process, implemented as the Correct Object-Oriented Pattern-based Parallel Programming

System (CO_2P_3S) (MacDonald, Anvik et al. 2002) , itself influenced by earlier work on Design Patterns and Distributed processes (DPnDP) (Siu, Simone et al. 1996) and the "Frameworks" template-based approach (Siu, Simone et al. 1996, Singh, Schaeffer et al. 1991) , and ultimately being distilled into a tool independent Parallel Design Pattern (PDP) process for pattern based parallel programming (MacDonald 2002) . Applications are auto-generated from parametrized design pattern templates, allowing the customisation of parallel patterns to a specific problem (the application developer providing the specific application code). These systems are independent of programming language and parallel architecture. As well, this is very flexible in terms of supporting application specific interfaces, and in generating otherwise useful support code. A prime advantage is the guarantee of parallel structure correctness of the generated framework (but not user code!). In order to support openness, the system allows for low level modification of the generated code, if found necessary to improve performance. It is acknowledged that this would then remove any guarantee of correctness. Furthermore, these manual modifications results in bespoke code that is less portable and maintainable (a common trade off, to be reconciled when tuning for performance). The authors recognise that any low level programming would require expert parallel programmers, and make the distinction between this work and application programmers work (implying that low level programmers may be needed during application development). Another consideration is that the template parameters used to generate the code are applied at compile time rather than at run time, which may constrain flexibility in tuning performance, although it should be possible to provision for runtime parameters as well. The current implementation of CO_2P_3S is in java using threads (i.e. for shared memory systems).

Alongside work in defining a pattern language, the PALLAS group at the Parallel Computing Laboratory at U.C. Berkeley (Su, Catanzaro et al. 2009) , is engaged in ongoing research into parallel libraries and parallel frameworks (Asanovic, Bodik et al. 2008) (Catanzaro, Keutzer 2010), noting that different frameworks can expose different design patterns that are then parameterised with plugin components. A prime objective being to separate out the application development productivity by use of a framework, from an efficiency layer concerned with the design and implementation of that framework. One promising line of research into building parallel frameworks is 'Selective Embedded, Just In Time specialization' (SEJIT) (Catanzaro, Kamil et al. 2010). In this approach, templates form

generalised frameworks, and specialized code and parameters are then set in at runtime (i.e. specialization of the template - c.f. C++ templates). The approach separates out the low level parallel programming from domain development, such that "efficiency" experts can develop and enhance parallel programming support for multicore and gpu environments. A novel aspect being that parallel processing is *selectively* introduced, when it is expected to improve performance.

3.4.2 Frameworks Modelling Specific Patterns

Master Worker

The master worker pattern of parallel processing is ubiquitous. It is simple and flexible, and particularly suited to 'embarrassingly parallel' problems that can be divided into tasks that can be independently executed. Asynchronous use allows for increased tolerance, and task size can be adjusted to ensure that tasks requiring varied processing times are executed optimally where a demand driven approach supports automatic load balancing. The pattern also lends itself to hybrid implementations, where message passing paradigms (e.g. MPI) can be used at the coarse level between nodes, and shared memory techniques can be used at a finer level (e.g. threads, openMP).

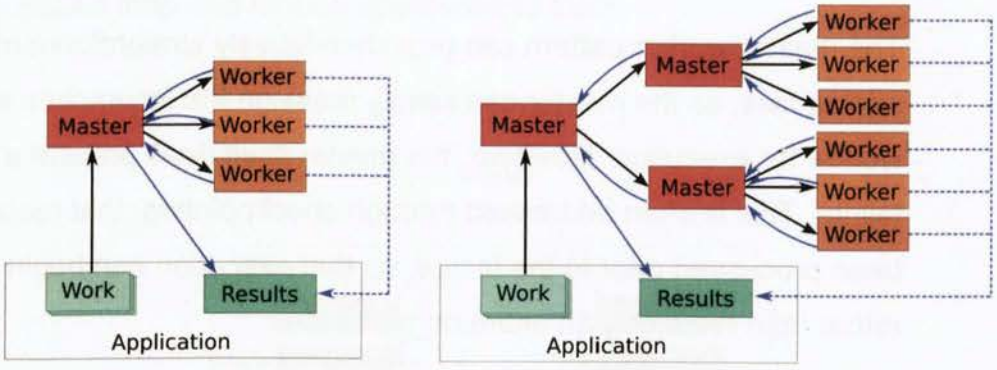


Figure 3.1: Common master-worker variants

Figure 3.1(a) shows the simplest and most common master-worker arrangement. One master is in control of retrieving work as discrete tasks that can be run independently in parallel, and of distributing them to a set of available workers for execution. The black arrowed lines in the figure depict the flow of work to the

workers. The partial results from each worker are then returned to the master for delivery back to the application as depicted by the solid blue arrowed lines. The means by which the work is partitioned and recomposed is application dependent. The most serious draw back to the master-worker pattern is that the master can become a bottleneck and impede speed up. This can be acute when the master must coordinate the partitioning, as well as delivery of work, and the gathering and composing of results. One way to alleviate this is to arrange for the results from each worker to be delivered via another route to the application (if possible) such as via the file system, as depicted by the dashed blue arrowed lines. It could also be organised that each worker retrieves its work from the file system, and the master processing could then be more lightweight (not shown).

Another common way to alleviate the load on a single master is to introduce a hierarchy of masters, such that each child master is responsible for a sub-set of the work, as shown in Figure 3.1(b). In this case, the root master and child masters can work at a courser grain, which can improve performance for applications that suit this variant. A hierarchical arrangement presents an increased choice as to how each master and child master coordinates the distribution of work and collection of results. The diagram cannot show all the possibilities, but as an example does similarly depict an alternative route to deliver the results to the application, via the dashed blue arrowed lines.

The master-worker pattern can provide relatively straightforward fault tolerance if a worker fails, as the master can simply reassign the incomplete work to another worker for execution. However, the master itself does present a single point of failure. This is often addressed through checkpointing, that records what work has been processed prior to the failure, so that execution can begin from that point, rather than restarting an entire computation.

MW (Goux, Kulkarni et al. 2000, Goux, Linderoth et al. 2000) is a C++ framework that allows applications to parallelise computations using the master-worker paradigm. It is targeted for scientific use in a meta-computing environment such as Condor (Thain, Tannenbaum et al. 2005). MW provides a 'top level' application programming interface (API) to applications in the form of Master (MWDriver), Worker (MWWorker) and Task (MWTask) abstract classes that applications implement (specialise). A prime objective being to provide a convenient and simple abstraction to the application programmer, to harness parallel compute

resources. MW also provides a 'bottom level' Infrastructure Programming Interface (IPI, another similar contemporary term is 'Service Provider Interface' or SPI) that allows for portability across grid computing toolkits by defining the communication and resource management as abstract classes. IPI implementations have used PVM and the Condor high throughput system, with the IPI design allowing other implementations including MPI. This adheres to good software practise, where a model provides an abstraction at one level, and provides interfaces to abstraction levels above and below it. The core abstracts the communication and resource management to abstract classes. Of note is that MW provides support for fault tolerance. In the Master worker paradigm, if a worker fails the task can be restarted. However, the master can be a weakness and MW provides a checkpointing interface to allow applications to record checkpoints as processing progresses.

MapReduce

The MapReduce programming model (Dean, Ghemawat 2008) abstracts parallel distribution and aggregation to provide a simple framework for parallel processing. Figure 3.2 depicts a simple schematic of the initially proposed master worker arrangement, with the master creating mapper and reducer workers and assigning user supplied map and reduce operations to them.

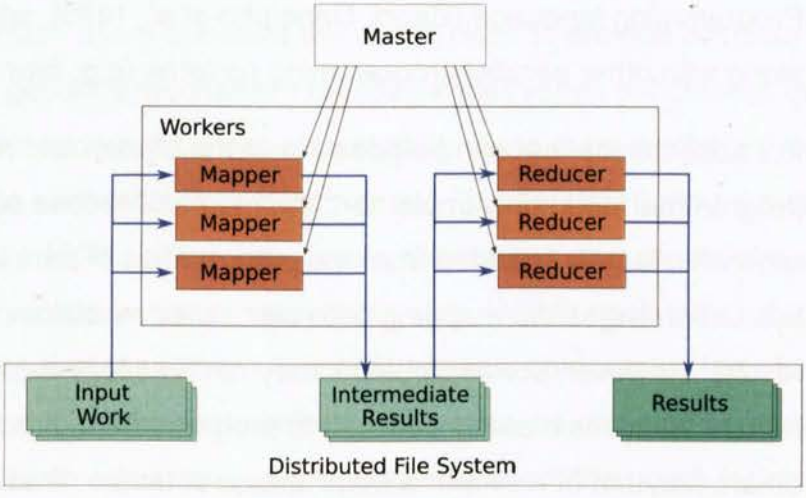


Figure 3.2: Map-reduce simple schematic

Central to the operation of MapReduce is the concept of a key-value pair. Each mapper takes from the input domain a logical value identified by a key, and applies a user supplied map implementation to transform the value to some intermediate key-value pair in the target domain. This intermediate output is stored to disk. Each reducer is assigned an intermediate output key, and applies a user supplied reducer implementation to the intermediate output values for that key to compose the final results.

Many variants of the MapReduce model can be formulated, being influenced by the target architecture and type of problem. For instance, other control structures might be used for instantiating and managing the mapper and reducer processes, and intermediate output can be delivered directly to reducers and only stored to disk when there is insufficient memory, as is often the case for 'Big Data' problems. There are many ancillary elements used to augment MapReduce systems, including optional combiners that can pre-aggregate a mapper's intermediate output, and partitioners to partition the intermediate output, ensuring that each reducer receives input associated with the key it is assigned.

The Hadoop framework (Cutting 03/19/2012) is a popular open source java implementation of the map reduce programming model together with a supporting distributed 'Hadoop file system' (HDFS). IBM uses an augmented variant of Hadoop in its Big Insights product (Ebbers, De Souza et al. 2013)). This style of programming is well suited to the handling of unstructured data and similar approaches can be found in earlier works such as the P3L Pisa Parallel Programming language (Bacci, Danelutto et al. 1995), which supports map-reduce along with other parallel programming patterns (e.g. farm).

For applications that can be posed in terms of map and reduce functions, the programmer need only implement the map and reduce abstractions, and is otherwise largely shielded from the complexities of parallel programming. The system arranges the mapping onto distributed resources, and the collection of results. However, some problems may not be easily expressed as map and reduce functions making it difficult to program them directly in MapReduce, while others may not fit well with a MapReduce structure, leading to performance issues (e.g. bottlenecks). The sweet spot for MapReduce appears to be for huge 'Big Data' problems that are not that hard, and only require a forward sweep through the data.

For analysis problems that require an iterative approach, the MapReduce batch oriented model can be particularly cumbersome and inefficient, and this has motivated other approaches to fill this space. One of note is Spark, an in memory distributed computing engine targeting Big Data analytics (Zaharia, Chowdhury et al. 2010) and in particular machine learning algorithms, many of which are highly iterative in nature. Spark's main data structure is the 'read only' Resilient Distributed Data (RDD) which is reminiscent of distributed shared memory techniques (but specific to one data structure). Spark applications typically create pertinent RDD representations and define transformations on them. Actions are then defined to run these transformations, with the Spark system organising RDD distribution and processing. Of additional note is Spark's novel fault tolerance approach using the concept of lineage to rebuild data lost through failed processes rather than traditional checkpointing. Although the RDD abstraction seems potentially restrictive, the in-memory iterative approach is attractive and Spark is becoming increasingly popular.

3.4.3 Graph based frameworks

Although designed for 'Big Data', the MapReduce model is not a natural fit for large graph problems. This has prompted efforts to devise frameworks specifically for this type of problem, prominent amongst these is Pregel, described as a "*framework for processing large graphs*" (Malewicz, Austern et al. 2010). Pregel uses a generic programming style (it is implemented in C++), the core element being an abstract vertex class. Much as MapReduce requires the user to provide a compute function, users subclass the vertex class to provide bespoke functionality. Although the generics mechanism mandates that vertex subclasses use the same types, this restriction can be circumvented by using more flexibility types. Pregel is inspired by the venerable Bulk Synchronous Parallel (BSP) model proposed by Valiant (Valiant 1990), where processing progresses as a series of super steps, with synchronization between each super step. Within each super step, each vertex receives messages from its input vertexes, computes a user function, sends output messages to its output vertexes, and vertexes can also be created, removed and even adjust their output edges to adapt the graph topology. The input messages are those that were send on the previous super step, and a vertex's output messages will be delivered in the next super step, a message passing paradigm being used. That processing is synchronized after each super

step allows for much simpler reasoning, and deterministic behaviour. A large graph must be partitioned in order to distribute vertexes amongst the available processors and in the default implementation, vertexes are simply assigned to partitions according to a hash on the vertex's id. This is admittedly not ideal and for many problems the user may have to provide a custom mapping to optimally distribute vertexes amongst the processors. Distributed processing is orchestrated via a master-worker model, with fault tolerance arranged through checkpointing.

Earlier graph based approaches abound, prominent amongst these being the Parallel Boost Graph Library (parallelBGL). ParallelBGL is also a template based distributed graph library, that can use MPI as its communication substrate (Gregor, Lumsdaine 2005). It is a distributed counterpart to the renowned Boost Graph Library. In parallelBGL, a graph is represented as a distributed adjacency list, with graph partitions being stored locally as vertex-output edge adjacency lists. As important to this library is the concept of process groups, these representing an extension of the BSP model where super steps are executed followed by internode communications, but where communication could be arranged within a step for specific requirements. In this way, synchronization points are introduced into graph algorithms to ensure deterministic behaviour (which of course requires considerable skill). Another feature of the library is support for ghost cells, through distributed property maps that essentially allow replication of remote data locally. This is a useful feature for some algorithms, but care needs to be exercised in its use, as data replication overhead can impact scalability. ParallelBGL is a very efficient and quality choice for handling reasonably sized graphs, whereas Pregel is designed to scale to very large graphs. Most users will not be developing graph algorithms for parallelBGL, but instead will want to use the already developed and battle hardened graph algorithms available.

The two examples chosen here are representative of multiple graph based approaches, evidencing that each must still grapple with the fundamental problems of optimising partitioning and distribution. Of course, a key observation is that these libraries are only of use to problems that can be represented as graphs, and as such would only form part of a larger application that required more than just a graph processing ability. In these applications, multiple libraries would have to be employed, and to leverage distributed processing, they would have to be distributed libraries, likely introducing integration challenges. Graph parallel

approaches are stateful, in that the data structure values are updated in situ as processing proceeds, in contrast to the data flow approach, presented in the following section.

3.4.4 Data flow and Streaming Frameworks

The data-flow programming paradigm explicitly describes parallelism by representing a program as a graph of connected nodes, where nodes process input data, produce output data, and each node can execute asynchronously when all its inputs are available. The pure data-flow model initially gained prominence as an instruction level data driven alternative to the Von Neumann sequential control-flow model, its multi-locus control being proposed as a means to ameliorate memory latency and synchronization overhead inherent in the control-flow model (Arvind, Iannucci 1983). However, it became apparent that this fine grained model had shortcomings, such as the increased overhead incurred in arranging instruction execution and in detecting enabled instructions, and as important, in handling more complex data structures (Lee, Hurson 1993). This has motivated the development of hybrid (or macro) courser grained approaches to data-flow where multiple instructions at a node proceed according to the sequence control-flow model, and inter-node control follows the data-flow model. In a multi-processor architecture, this hybrid data flow approach is usually implemented at the thread level. The data-flow model has much in common with functional programming, wrestling with similar issues such as grain size and data structures, and using similar solutions such as statelessness and immutability.

A number of frameworks supporting the data-flow model have been implemented, and here interest particularly extends to those 'hybrid' examples that also offer application level patterns of parallel programming. In (Aldinucci, Anardu et al. 2012), the '*macro* data-flow' pattern based programming framework is proposed as a structured parallel programming approach targeting multicore architectures. The authors argue that fine grained data-flow has historically had mixed performance results, and that a coarser grained (i.e. a macro task) approach is more appropriate. Of note are the common core issues of task size and communication. The paper also mentions an optimisation to 'group' subgraphs that are too fine grained into a single node. The data-flow graphs are compiled from structured programming environments, and expressed within these environments as

compositions of provided parallel design patterns such as *pipe*, *farm* and *map-reduce*.Efficient macro data-flow interpreters then automate the running of the macro data-flow graphs. An API allows for the introduction of new patterns, thus supporting extensibility.

A related research effort, FastFlow (Aldinucci, Danelutto et al. 2012) is a framework supporting development of multicore parallel programming, particularly suited to stream based applications. It uses a layered approach to separate low level parallel programming (i.e. threads and synchronisation) from intermediate and high level pattern based development. Although accessible, the low level programming layer is more the domain of the framework developers, and includes thread and synchronisation management, and asynchronous lock free queue constructs used for composition and communication in data-flow or streaming based applications. At the intermediate level, developers can compose graph representations of programs using skeletons of predefined parallel programming patterns, connecting via queues. Similar to Intel's TBB's pipeline streaming support, this level requires some knowledge of parallel programming. At a higher level, problem solving environments can be built that support abstractions for specific domain usage (e.g. a parallel Monte Carlo simulation abstraction) - largely removing an application developer's exposure to parallel programming. The main aim of the framework is again to reduce programmer effort by providing skeletons of common parallel programming patterns . The skeletons are parametrizable, with parameters being supplied at compile time. A flexible feature of the framework is that parallel patterns can be arbitrarily nested, within other parallel patterns as appropriate. The data-flow and streaming approach places emphasise on composition, as a core concept. This is in contrast to imperative paradigms that use composition to organisation and connect software components, through more varied interfaces (see next section). The framework could be extended across a distributed architecture using the streams paradigm, but its application in this context would of course be constrained to problems that fit well with a streams based approach.

In closing this section it should be pointed out that implementing a thread based hybrid or macro data-flow model targeting shared memory multiprocessors does inherit the limitations of thread level parallelism, principally in terms of scalability. Indeed, such concerns have also renewed interest in the design of data-flow

processors (Hurson, Kavi 2008).

3.4.5 Composition and Workflow

Composition is an overloaded concept. It is an effective core software engineering methodology for building up a complex application from constituent components that may also include one or more frameworks, with frameworks often being build similarly from multiple components. However, in this section the interest is on frameworks supporting the concept of a workflow, and in this context composition is used to describe the construction of large graphs of compute entities that can potentially be run in parallel (i.e. as a composed workflow), exploiting parallelism at the component level. Each component may also itself be parallelised in a nested fashion. Related to the composition of component or task graphs is the data flow through such a graph, and together these specify a runtime workflow. The most common and more amenable graphs are directed acyclic graphs that mandate the direction of dependencies and have no cycles.

The Pegasus framework (Deelman, Singh et al. 2005) separates the logical or abstract directed acyclic graph representation of an application workflow's components and data dependencies, from that workflow's mapping onto distributed resources for execution (Pegasus stands for "Planning for Ex- ecution in Grids"). The mapping of workflow tasks to resources is generally an NP-complete problem, and so heuristics are often employed to help map to 'near optimal' acceptable solutions, with the goal being to minimize the execution time. This separation simplifies application development, improves flexibility, tools can be used for the workflow composition, and the mapping to concrete resources can be automated, which is the core Pegasus functionality, handing off work to a job scheduling and resource management sub-system such as Condor-G (Frey, Tannenbaum et al. 2002) or PBS (Henderson 1995). As is common with workflows that have an executive at the component/task level, Pegasus has some built-in fault tolerance to re-execute tasks. The Pegasus framework emphasises the merit in separating the workflow definition (or application description and commonly expressed as a DAG), from it's eventual mapping to resources for execution, which may not be known or change.

Cascade (Tagliasacchi, Best et al.) is a Parallel Processing framework facilitating parallelisation in complex C++ systems. Users implement *CascadeTask* 's and

compose them into a task dependency graph. At run time a *CascadeJobManager* instantiates threads to execute the tasks according to the defined dependencies. The framework is designed to exploit task parallelism at the thread level (implemented in C++ and using pthreads), and includes explicit expression and management of data-flow, interactive and real-time dependencies to optimise performance. The authors point out that Cascade can be integrated with, or contain other parallel constructs such as OpenMP and map/reduce, presumably with a commensurate increase in programmer effort and complexity. As with many frameworks specifically targeting shared memory architectures, integration into distributed architectures may be non trivial.

3.5 Domain Specific Frameworks

These frameworks provide core infrastructure and support libraries that are specific to a domain. Parallelisation of such frameworks can be incorporated at the outset, although it is also common to subsequently introduce this functionality once the utility of the application has been established, and performance becomes a concern. In this section, a selection of application frameworks that have incorporated parallel and distributed patterns are reviewed.

3.5.1 Parallel Frameworks for Evolutionary Algorithms, Simulations and AI.

Optimisation problems are inherently compute intensive, and usually have a high degree of implicit parallelism. As an example, evolutionary computations seek to find an optimum solution for a given problem using an analogy to Darwinian evolution, using selection, crossover and mutation operators. The evaluation of solutions found can be very compute intensive, and that each candidate solution can be evaluated independently lends itself to parallelism. Distributed BEAGLE (Gagne, Parizeau et al. 2003) is an extension to the BEAGLE framework for evolutionary computations, supporting a master-slave model to distributed the work amongst available processors, and recognising that the grain size of tasks has a significant impact on the speedup obtained. Distributed BEAGLE also embeds an Island model that divides the global population into separate sub-populations that includes migration between these populations, a fine grained model suitable for SIMD architectures and hierarchical hybrids of these models. Communication is achieved via an xml data protocol communicating over TCP/IP

sockets

Similarly, the ParadisEO (parallel and distributed evolving objects) (Cahon, Melab et al. 2004) extends the 'Evolving Objects' framework and incorporates reusable parallel and distributed meta-heuristics for both evolutionary computation, and supports local search optimisation problems using a layered architecture of *solvers, runners and helpers*. For evolutionary computations, ParadisEO distinguishes three parallel and distributed models, the Island cooperative model to partition the population, the parallel evaluation of the population, and the distributed evaluation of an individual. These form a hierarchy, where sub populations are distributed for execution, a master (farmer) will then apply selection, crossover and mutation on each sub-population to evolve new solutions, and the quality of each solution is then evaluated, potentially again in parallel. Local search optimisations is similarly parallelised. The emphasis in ParadisEO is on a multi-layer and modular architecture, expressing the advantages in terms of design and code reuse, of using a framework. The framework supports both shared memory communication models using posix threads, and distributed memory models through PVM or MPI, and a higher level 'channel' abstraction. Goals include maximising design and code reuse, flexibility and adaptability, utility, transparent and easy access, performance and robustness.

Note that in the above examples, the parallel support structures such as master-slave are embedded into these frameworks, although it is acknowledged that they could be extracted and generalised for reuse as the parallelisation is usually orthogonal to the function of the application and should be separated out. Tightly integrated frameworks can be tuned to the application domain, but remove the advantage of code reuse.

3.5.2 Image Processing Frameworks

Image processing applications typically use compute intensive algorithms. Furthermore, many of these algorithms exhibit inherent parallelism. To this end, a number of libraries and frameworks have been proposed and implemented, to realise performance improvements. To shield the end user from the details of parallel programming, libraries commonly implement parallel versions of image processing operators. In this way, the user is unaware of the underlying parallel processing. An interesting approach (Seinstra, Koelma et al. 2002) introduces the

notion of *parallelizable patterns* in order to increase code reuse, and improve maintainability. In this data parallel image processing library, a single uniform api is provided although the underlying implementation can be sequential or parallelised. The *parallelizable patterns* encompasses the maximum amount of work that can be processed either sequentially or in parallel, and this is implemented once, and used in both the sequential and parallel implementations, so removing maintenance issues due to code duplication.

In (Nicolescu, Jonker 2002) it is recognised that data and task parallelism when used separately are relatively limited, and that exploiting both can yield better performance. This is particularly applicable to image processing, and a processing environment is proposed in which data parallelism is supported using algorithmic skeletons, and task parallelism through composition. Here the skeletons are abstractions encapsulating particular patterns of parallelism, where image operators are implemented as higher order functions, shielding the programmer from the data parallelism details. The programmer can compose these functions as tasks that are the nodes of a directed acyclic macro-dataflow graph (called an *image application task graph* in this context, examples indicating this as implicit in the program structure, rather than as explicit task objects, being implemented in C). In this way, both data and task parallelism are exploited. A cost model is also proposed, to optimise task scheduling, accounting for the expected communication costs of data distribution within tasks and redistribution across tasks, and computation costs of the tasks.

3.6 Domain Specific Languages

This chapter has so far focused on frameworks, highlighting key characteristics including reuse and user guidance. This section broadens the discussion to include an alternative that can be employed at the domain level, namely a domain specific language (DSL). In contrast to frameworks written in a general purpose programming language such as C++, Java or Scala, a DSL defines its own higher level bespoke syntax and semantics for a particular domain. The main motivation for the development of a DSL is that a domain expert can then develop the domain specific aspects of an application in a higher level language that is more naturally expressive of that domain. Many characteristics of frameworks have their counterpart in DSLs. For instance reuse is manifest in a DSL as the concepts it

defines are reusable across multiple applications and a DSL confines and guides a user perhaps even more so than frameworks since the user is restricted to the defined language (and user code does not normally extend a DSL). Some of the limitations of frameworks are also evident in a DSL, with the so called 'framework gap' having its equivalent in a DSL that does not completely specify its domain. Creating and using a DSL does introduce its own challenges. A DSL is usually embedded into a host language and thus extra transformation steps are required to convert the DSL syntax to a target language representation for compilation, a converter being required for each target language. As with frameworks, multiple DSLs may be needed to support various domains in one application, and these may not mesh well within the target language. A detailed comparison of frameworks and DSLs is elucidated through an e-commerce case study in (Johansen 2009).

Returning to the theme of this chapter, a core interest is how to integrate various DSLs into a parallel processing environment. The Pervasive Parallelism Laboratory at Stanford University is working on the 'Delite Compiler Framework and Runtime', that includes infrastructure to facilitate creating parallelised DSLs, together with compilation and parallel runtime support (Brown, Sujeeth et al. 2011). It is incumbent on the DSL developer to identify parallelism within the domain, and map that to parallel patterns made available within the Delite framework, an admittedly challenging exercise. This also implies that the DSL developer can only use the parallel patterns available at the time the DSL is created, which may be restrictive. However, another core aim of the research is about multiply transforming and compiling a single program so that it can be run on many different parallel architectures (e.g. multi-core, gpu). Besides being restricted to the parallel patterns that the framework encodes, this implies a further restriction in that an application can only run on hardware for which Delite has implemented a complier. Further insight as to the advantages and disadvantages of the compiler versus library approaches will undoubtedly unfold as this important research progresses.

3.7 Service Oriented Architecture

The frameworks reviewed so far in this chapter have placed the greater emphasis on parallel processing performance. As outlined in the introduction, distributed

computing brings modularity and reuse to the fore, the focus shifting to architectural considerations that aid flexible composition of modular applications, with performance being also measured in terms of reliability and scaleability, rather than just processing speed. The most popular architectural style that supports the provision of scaleable distributed computing, especially in the realm of business applications, is the Service Oriented Architecture (SOA) and it will be illuminating to briefly review its merits and limitations. It should be noted that due to its popularity, and the manner of its rise to eminence, SOA is defined in more than one standard and one can gain a better footing by first consulting the Object Management Group paper that provides guidance on the standards landscape in this respect (Heather Kreger IBM, Jeff Estefan NASA/Jet Propulsion Laboratory 2009). This chapter section confines itself to the core concepts of SOA pertinent to distributed computing, and goes on to review how services can be productively realized using the Spring framework (Johnson, Hoeller et al. 2013).

3.7.1 Distributed Services

The essence of SOA is about organizing the distribution of application capabilities through loosely coupled services. Each service encompasses some self contained functionality and exposes it to other services via an agreed communication protocol. A service may enlist multiple other services to provide its function. This architectural style promotes reuse and composition of applications by connecting services. It is a common practise for applications to be organised into multiple horizontal layers (also referred to as 'tiers'). A three layer arrangement might contain a presentation layer, a business layer and a data tier layer and the SOA approach extends this to allow for vertical layering. For example, a business layer may be composed of multiple connected services that collaborate to provision the business logic of the application. The architecture allows for the independent development, testing and maintenance of services, with flexible update and addition of these into an application (c.f. a monolithic application where the whole application may have to be redeployed after an update). Another major attraction of this approach is that the number of service instances deployed can be adjusted to align with the load requirements, and services can be swapped out for maintenance or upgrade.

There are some challenges and pitfalls to the SOA approach. Although the

modularity can aid system understanding from an architectural perspective, it can also become a burden if care is not taken. The granularity of the splitting up of functionality into separate services must be carefully considered, as this impacts the extent of service dependencies. Related to this, multiple concurrent service interactions can quickly become complicated, and applying consistent security across multiple services can be difficult. The attraction of swapping out services for upgrade may not actually be a simple matter, for instance if a service API changes. Admittedly this is not specific to SOA, but does become important with myriad interfaces to manage. Reliability and performance may suffer when all service calls are to remote interfaces, using heavy communication protocols. An ameliorating observation is that collaborating services are often run on the same data centre cluster where the connecting fabric is local and high speed. Fault tolerance becomes more important because myriad distributed services increase the chance of service failure, and this may require complicated application specific recovery mechanisms (conversely, the architecture allows for the provision of redundancy). Given the pros and cons, it is readily apparent that SOA isn't a silver bullet, and doesn't obviate the need for good application and system design. However, it is demonstrably successful in providing an organising architecture for the flexible and adaptable composition of distributed applications, and it fits very well with the emerging trend towards cloud infrastructure technologies. A SOA is also applicable to 'Big Data' problems that are amenable to course grained decomposition, where the computation of each partitioned task significantly outweighs the service communication overheads.

3.7.2 The Spring Framework

There are numerous libraries and frameworks that have evolved to assist the development of SOA systems. A popular choice is the Spring framework, which provides a lightweight application container together with a number of supporting modules. Of chief interest here is the inversion of control (IOC) module that facilitates the flexible assemble of applications, which of course includes SOA services. IOC is a seasoned software technique to decouple the execution of an abstraction from its implementation. Control is handed over to the framework, which then facilitates the dynamic binding of specific implementations at runtime (this is also commonly encountered in event driven systems). Although IOC facilitates modularity, and loose coupling of components, the prime advantage is in

enabling extensibility, where new implementations of behaviour can be plugged into a service. This is an contrast to traditional programs, where application logic is statically expressed as an integral and fixed part of the program. The Spring framework uses dependency injection to implement IOC (Fowler 2004), with annotations in code, and configuration files describing how the framework should 'wire up' an application. Another common dynamic binding technique to implement IOC is through the use of a service locator. IOC is a very powerful technique, that will be of central importance to the proposed design of a distributed framework in the next chapter.

A second powerful module worthy of mention before concluding this section is the Spring MVC framework, used to implement the 'model view controller' architectural pattern in an application's presentation layer (Fowler 2006). This greatly simplifies the exposing of a service's remote interfaces, typically as REST endpoints (Fielding 2000), again by the incorporation of annotations in code, and is a key enabler to productively creating services for SOA systems. The communications overhead is necessarily significant in SOA services, to provide robust and secure interactions, and this does of course bear heavily on the type and extent of parallel processing that can be arranged within this architecture, and hence the usage is more focused on distributed computing.

3.8 Summary

The future route to more performant computing will be via parallel processing. There has been limited success in automatically parallelising programs. Low level parallel programming requires considerable specialised skill. It is anticipated that the majority of programmers will be application domain developers rather than skilled parallel programmers (system designers) (Samuel H. Fuller, Lynette I. Millett et al. 2011). This implies that higher level abstractions must be sought, that can provide an adequate level of productivity for application programmers and leave the lower efficiency layer to parallel programming experts. Parallel programming frameworks fit the bill, providing core reusable infrastructure code together with abstract interfaces that users can hook into to leverage and extend the framework functionality. It is widely accepted that higher level abstractions trade ease of use and productivity for efficiency and performance. So a parallel processing framework must be judged not only on the productivity it affords, but

also on the performance it provides. Lower level parallel processing frameworks are often 'open' for this reason, such that they can be tweaked to improve performance for specific applications.

A broad range of frameworks, represented at varying levels of abstraction that support many parallel programming patterns have been reviewed. All these frameworks aim to assist the user, by supplying implemented core parallel processing infrastructure, with those at a higher level also providing domain specific libraries. Useful categorisations are based on the abstraction of a framework in terms of the level of exposure to the details of the underlying parallel processing implementation, and on the generic or application specific design intent of the framework. It is recognised that in the general case, parallel processing is orthogonal to the domain specific application, and it is good programming practise to separate out this aspect to improve reuse, development, testing and maintenance.

Some common attractive features of a parallel processing framework can be discerned from this review. In generic parallel processing frameworks, the user supplies specialised code that is then inserted into appropriate parallel patterns, parallelisation is then automatic based on the available resources. Common patterns should be available as a core part of the framework, but because it is difficult to predict the requirements of future applications, a framework should be extensible, such that novel parallel patterns can be easily plugged in. As well, it is desirable for a framework's design to be broad enough to accommodate distributed and shared memory systems, and hybrids of these and also allow the inclusion of GPU resources when applicable. Many of the frameworks reviewed target shared or distributed memory systems, but not both. Task and data parallelism should be supported, with facilities to compose task graphs and execute tasks according to the implied dependencies, and partition data for each task as appropriate to improve performance. Openness is a double edged sword as it is often necessary to allow for potential low level performance optimisations on specific systems, so a framework should be open, but effort should be made to remove or reduce the need for optimisations, or to automate optimisations whenever practical. To this end, parallel processing frameworks should employ diagnostics and feedback of runtime statistics to continually monitor and improve the parallel runtime of the framework and the application code. This can also feed

back into further development of both the framework itself, and application code that uses it. A lot is said about optimised manual code being much more performant than generic code (i.e. provided by frameworks). However, the assumption is that expert programmers are available, with sufficient time to commit. These assumptions do not always apply, and it may be that the more generic but heavily used and tested code evolves to be more performant than the average manually written code.

Instead of embedding a framework into an application or sections of an application in an ad hoc fashion, for many applications it is arguably favourable to hand over execution control to a parallel framework. Certainly this inversion is attractive in image processing applications where pipelines and workflows are defined, which are in any case usually delegated to workflow runtime infrastructure for execution. When a framework runs 'plugged in' parallel patterns and application code, the ensemble becomes the composed application. Indeed, the application can be run as a separate server. These ideas feed into the requirements for a novel parallel processing framework introduced in chapter 4.

Finally, some of the shortcomings of frameworks should be admitted. There is always a learning curve to using a framework, even if comprehensive documentation is available (as it should be). The design of a framework may be incomplete or not be intuitive to all users. There can be framework gaps, with functionality falling short of domain requirements, framework overlap and issues of framework composition and cohesion can surface, or incomplete design intention and limited or no access to source code can impede understanding and usage (Mattsson, Bosch 1997). Extensibility can address many issues, but if core functionality is missing, the framework must be modified. By its nature, a framework guides but also constrains. This is intended to assist, but may be too restrictive. The framework design must be carefully crafted to get the right balance here. Also, much is stated about the separation of concerns, but a framework is bound to its users via the interfaces it exposes. These interfaces must also be carefully designed, as once a framework is in general use it is more difficult to update these interfaces and maintain backward compatibility.

Chapter 4 The DFrame

4.1 Introduction

Previous chapters have outlined common parallel programming models, languages and patterns, and a number of implementing frameworks have been reviewed. It is evident that many choices confront the designer, who must prioritise and compromise according to the specific aims of a particular framework. While some frameworks focus primarily on real time performance, others regard simplicity, flexibility and reuse as core. Still other approaches hold scalability as paramount with specific target hardware in mind, and many more frameworks target particular domains. The various approaches reflect not only the increased choice space that anyone wanting to harness parallel resources must contend with, but also the priority given to the myriad characteristics particular to parallel programming. Indeed, two frameworks may strive to support the same set of characteristics such as performance, scaleability, openness and reuse, but the framework designs will certainly differ if the priority given to each characteristic is different. Even if the core parallel processing goals are similar, differing solutions are often proposed based on the designers experience, target users and subjective aspects such as what constitutes 'ease of use'. Simplicity, generality and flexibility are all in tension and compromise is unavoidable.

4.2 The Argument for a New Approach

None of the reviewed frameworks fully provide the required flexibility, extensibility and adaptability at every level, and all the particular requirements sought, and so research began on how to support the core aims in a separate flexible parallel processing framework, driven by a specific focus to aid the speed up of 3D image processing, with an emphasis on supporting high content screening of cell biology imaging. It was immediately apparent that the effort expended on the parallel processing aspect was going to be demanding, and that this effort would be specialised and largely orthogonal to the image processing itself, and as such should be separated out and generalised into a framework, allowing for independent development and reuse. Further consideration led to the determination that parallel programming patterns would provide the necessary

structures that could separate and organise the parallel programming not only for flexibility and reuse, but also for extensibility, maintenance and testing, and allow tools to be devised to aid further development and runtime diagnostics. As outlined in chapter 1, the requirements include the ability to have a long running interactive experience, as well as provide batch processing for background high content screening of multiple 3D images. This suggested the need for a client server architecture, with clients composing and forwarding task specification graphs to a relatively long lived parallel processing server, that would automatically manage the parallel processing of each task according to dependencies expressed in the graphs. Alongside performance, the framework would prioritise flexibility, extensibility, adaptability and reuse at the parallel processing level and at the application level, this being important as it is not known in advance, all the parallel patterns to support or the imaging applications that will be composed and run. The long running batch processing or interactive sessions provide an environment in which ongoing performance feedback can be captured and used to automatically adapt the framework in terms of compatible parallel pattern selection and tuning to optimise task execution, and also the splitting to create process groups and their assignment to tasks so that the tasks can be further partitioned and parallelised.

This chapter introduces a new and novel distributed framework, referred to as the 'DFrame', designed to provide the required levels of flexibility, extensibility, adaptability and reuse. Chapter 5 then presents preliminary evaluations of per task (data) parallelism where the DFrame is applied to 3D imaging filters, segmentation and visualization operators and Chapter 6 goes on to present a completely integrated case study of a 3D image processing pipeline application that brings into relief both task and data parallelism and more fully demonstrates the power and adaptability of the DFrame in a real world application.The framework is primarily designed for use in an 'in house' high content screening image processing application, and so is initially targeted at Kingston University researchers. Of course, even in this context it is still of prime importance to aim for simplicity in use, and so a simple GUI is provided for presenting the available functions, composing task graphs which constitute specific application workflows, connecting to a server instance and running the composed workflows.

Figure 4.1 shows a schematic of the distributed imaging architecture, in the context of the targeted image processing (high content screening) applications,

and interactions with the distributed framework. Parallelisation is to be supported at every stage of the anticipated core image processing pipeline. The logical order of the project is apparent, where 3D filters and segmentation algorithms are first being parallelised, which are necessary to extract features that can be piped into the analysis stage. Also the visualization is progressed to provide feedback on inputs and outputs at all stages of an image processing pipeline.

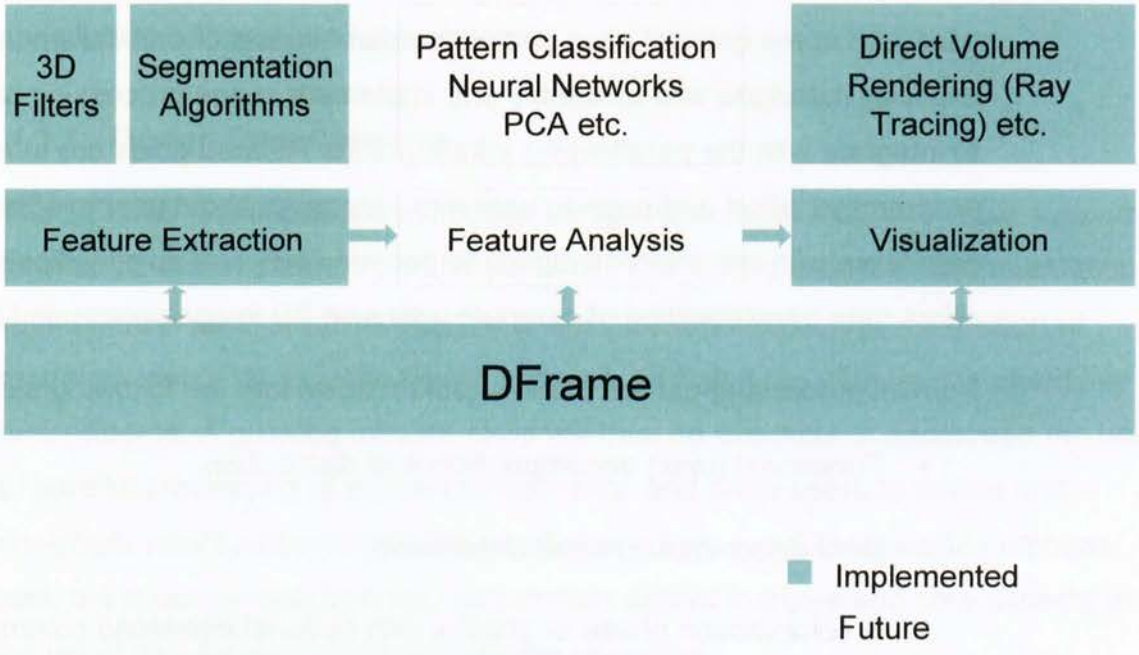


Figure 4.1: Overview of the Distributed Imaging System Architecture

Although the schematic makes clear that image processing is key at least in the feature extraction and visualization stages of the processing, it is anticipated that the feature analysis stage will require more general and quite different parallelisation strategies, and this encourages the design of a more general distributed framework. So although the distributed framework design must map to and work well with image processing, a principle of the design is to be generic enough to cater for other usages within a pipeline, such as in an analysis stage.

4.3 Conceptual Overview

Good software practise encourages the separation of concerns through modular design and the composition of software systems from component building blocks.

As well, frameworks provide additional organising structure, guidance and support for programmers, reducing development effort and increasing productivity. Applying these principles, a key part of this research is to provide a framework that allows for the separation of core parallelisation infrastructure from domain image processing functionality, a prime motivator being to encourage independent development and reuse of the parallelisation support framework and the image processing domain, as far as possible. The research endeavour will also help illuminate the extent to which such parallelism can be made transparent to the user, and in the general case bring into relief aspects of parallelism that must be explicitly managed and to identify and implement image processing support code to integrate with the parallelising infrastructure. Related goals are to reduce programmer effort and provide help with reasoning about how to effectively parallelise with minimum disruption to domain code and so specifically to encourage parallelisation of compute intensive 3D image processing functionality.

Parallel processing can be broadly broken down into the following steps:

- Functional (task) decomposition and distribution.
- Data decomposition and distribution.
- A computation phase or phases with optional interlaced communication dependent on the problem being solved.
- Recomposition and delivery of resultant data.

A particular problem may not require either functional or data decomposition, but some method of assigning work and data to independent processes will be required by definition, to parallelise processing. Likewise, processing is only useful in that a result is produced, assembled and made available or delivered to some target destination. This naturally leads to a requirement to separate out the decomposition, distribution, execution, collection and composition strategies, concepts fundamental to arranging for parallel execution.

If distributed tasks are independent, then the computation phase is quite straightforward and parallelisation is usually very effective. More complexity is involved when there exist dependencies amongst the distributed tasks such that an order is imposed on task execution. In this case, the expected parallelism can

be severely impacted by these dependencies, and effort must be made to ensure that dependencies are minimised and managed to optimize parallelisation. For instance, an adjustment of the task granularity or reworking of the domain algorithms can help. It is always important to compare parallel execution with serial execution, but even more so in these cases, to ensure parallelisation is effective and worthwhile, and a framework should provide rudimentary facilities to monitor performance in this regard and adapt processing to optimise. The following sections describe how the proposed DFrame incorporates these concepts to support parallel execution.

4.3.1 Design Core Concepts

The DFrame encompasses a modular design, with separated extensible support for models that implement parallel patterns. This facility enables experimentation with different parallelisation designs and assists and encourages evolution of model variants that express known patterns, and also the introduction of entirely new patterns. A growing infrastructure will thus be available to encourage the use of parallel processing of domain functionality, and allow users to devise and contribute new functionality to the parallelising framework for specific purposes. As well, the research should bring out common salient features and core aspects and issues of parallelising image processing operators.

Inversion of control (IOC) was introduced in the previous chapter (see section 3.7.2) as a technique that decouples an abstract representation from its implementation. This encourages modularity and loose coupling, and most importantly facilitates the dynamic binding of specific implementations at run time, which is key to extensibility. In practise, IOC is typically arranged by handing over control to a framework, that can then manage the dynamic runtime binding of implementations and drive execution. IOC forms a central part of the DFrame design, where multiple DFrame instances load and execute models implementing parallel patterns, which in turn load and execute application code. Inversion of control allows for the extensibility that is such an essential part of the DFrame design to enable novel parallel models and application code to be 'plugged in'. This mechanism is described in more detail in the following sections.

The DFrame design draws together the concepts of decomposition (partitioning), distribution, execution, collection and recomposition in models and tasks. Models

lie at the heart of the DFrame design, implementing diverse parallel patterns of execution and working in concert with a task's specification to achieve parallel execution of a task. A task in this context is a DFrame defined structure that contains application provided implementations of partitioner, executor and composer interfaces. The DFrame defines generic unstructured and structured variants of the partitioning and composition interfaces that applications can implement to provide bespoke flexible decomposition and composition strategies. Operationally, a model will use a task's partitioner to obtain and distribute parts for execution, and similarly after gathering results will use that task's composer to effect recomposition. Logic implemented within each model defines the details of how the distribution and collection is effected, and will map to the parallel pattern the model implements. A model drives execution of the distributed application code through the task provided executor interface. Implicit in this description is that models are distributed entities, being managed by distributed DFrame instances.

Model implementations can expose which decomposition and composition interfaces they support, and the DFrame can use this information to identify which models are suitable for a given task and so allow automatic adaptability within the DFrame, in its choice of model to use for a given task. The aim being to allow a more flexible mapping of compatible models to tasks such that the best model may be chosen at runtime based on the task itself, and the context in which it is running. The DFrame will also be able to use other information about the algorithmic characteristics of a task in order to further determine suitable models and adapt to an optimum model, such as whether each partitioned sub-task requires the same time to process, which is often related to data size.

The interplay between a task and an associated model arrange for a more fine grained parallelism, where each task defines a parallel unit of execution usually exhibiting data decomposition. At a broader level, a core objective of the DFrame design is to incorporate support for the assembly of multiple such tasks into task graphs. These graphs then define any task dependencies that impose an order on task execution. Indeed, by these means, the DFrame is used to construct complete bespoke applications through the workflows that these graphs define. The design also allows the DFrame to assess the resource requirements for a task not only on its own, but also in the context of its placement in a task graph, to determine the optimum execution plan in terms of what models to use for each

task, and what level of resources (processes) would optimise performance. This coarse grained parallelism adds considerable flexibility and power to the framework.

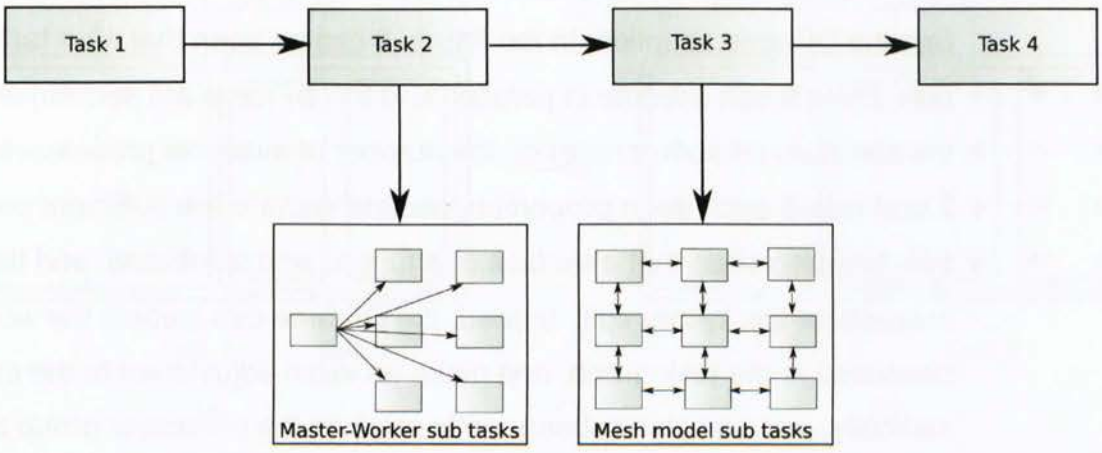


Figure 4.2: Task graph showing task dependencies and decompositions into sub tasks

In Figure 4.2 a simple task graph representing a pipeline application is shown. Each task has an associated model implementing a parallel pattern of execution. In this figure, Task 2 is using a Master-Worker model to arrange for unstructured parallel execution, whilst Task 3 is using a structured mesh model (although not shown, Task 1 and 4 will also be associated with models). At a more advanced level, the DFrame can detect (or be explicitly directed) that the output data from task 2 is used as input to task 3, and in this this case can choose to keep the data distributed if the models are compatible. As well, the model associated with task 2 may be adapted to align with the model used in task 3, with the participating DFrame instances caching such models and data to reduce set up and distribution costs and thus improve performance. This behaviour is described in more detail in the DFrame architecture section.

Figure 4.3 shows a task graph with multiple branches. Each task is again associated with a model (not shown). This figure depicts another feature of the DFrame design. When a task graph splits into sub-branches, the DFrame can arrange for the splitting of the available processes into appropriate processor groups, such that each sub-branch is assigned a number of processes. This

translates to the number of DFrame instances that will participate in parallel execution of the sub-branch, and hence the number of processes available to each task and its associated model. The DFrame can arrange this in a recursive manner, and in the figure this is shown at 'split 1' and 'split 2'. It is the responsibility of the DFrame to manage this, with tasks and models working with the resources that the DFrame supplies. In the figure, it can be seen that after task 1 executes, task 2 and 5 can execute in parallel, and the DFrame will perform a split (and transfer data as appropriate) on the number of available processors such that task 2 and task 5 each get a proportion, assuming there are sufficient processors to do this. It is also seen that after task 5, a further split is initiated, and the available processors are further split. Indeed, the DFrame can inspect the workflow described in the task graph, and make an initial adjustment to the splitting statically, and can also subsequently readjust the processor group assignments, adapting to runtime performance feedback. Merging will be automatically handled.

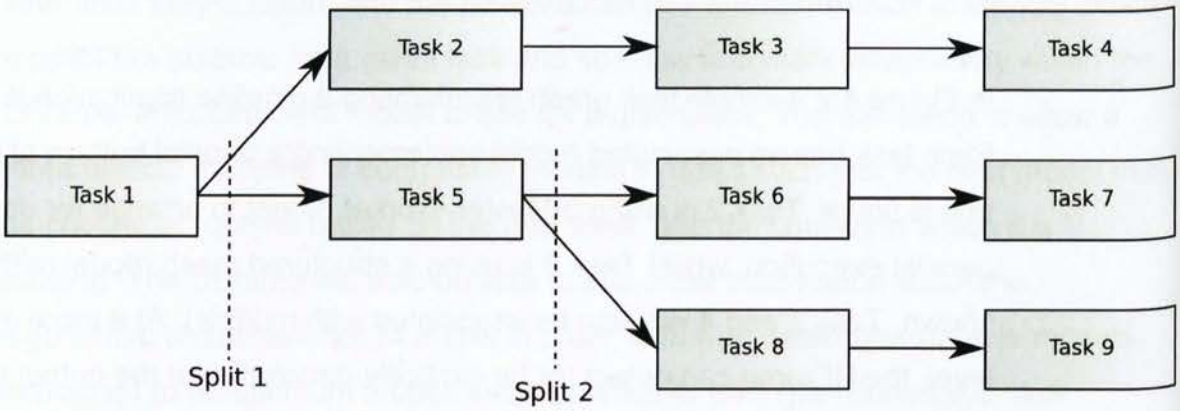


Figure 4.3: Task graph showing DFrame splitting of simple sub-branches

Figure 4.4 shows a processor grouping view of the task graph depicted in Figure 4.3 , when running the DFrame over 32 processors. Initially the DFrame may decide to run the tasks according to the layout shown in Figure 4.4 (a). On multiple runs, the DFrame will collect metrics that determine that the branch containing tasks 2, 3, and 4 is consistently completing earlier than the task 5 branch, and on subsequent runs can elect to reassign more of the available processors to the task 5 branch, and in this example, to the processor group running tasks 6 and 7 as shown in Figure 4.4 (b). This dynamic adjustment is possible due to the designed separation of resource usage from task execution, and scheduling at the

processor group level rather than the processor level. Every processor group has a root process for that group, that is responsible for running the sub-task graph within that group.

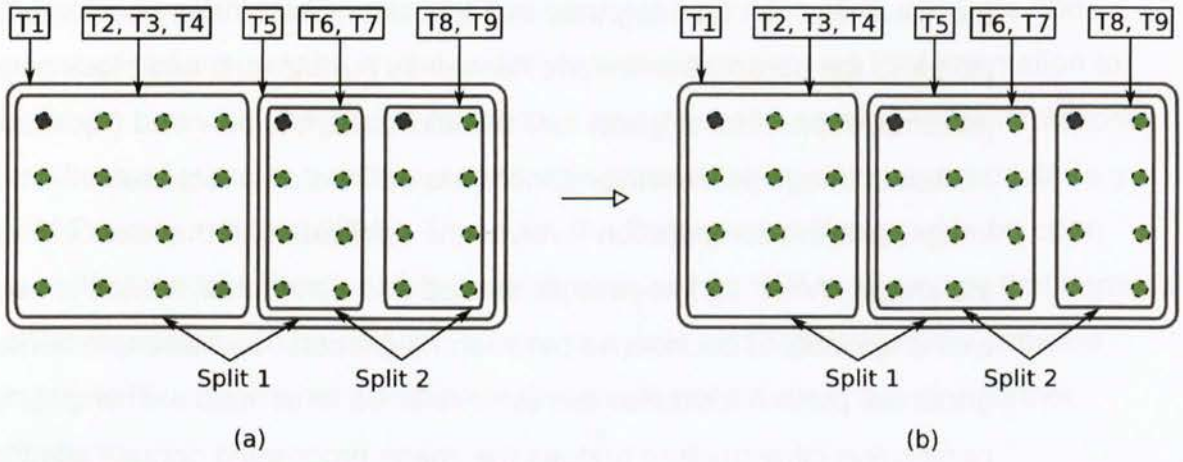


Figure 4.4: Adaptive processor groups when running a simple task graph

To summarise briefly, the DFrame runs models which themselves arrange for the parallel execution of tasks. The DFrame manages the number of processors that will participate in a specific model run and thus the extent of parallelism at the task level. Built into the DFrame is the ability to adapt the resources supplied to tasks (via models) within task graphs, and also to determine and adapt model selection based on individual task characteristics, and the context in which a task is run. Within a model run, the task partitioning strategy can be arranged to be dynamically adapted at runtime such that the distribution strategy is determined that optimises processing efficiency in terms of speed up and memory usage.

4.3.2 Image Processing

Although inclining towards supporting generic parallelism, the DFrame design is influenced by an immediate requirement to cater for common image processing operators. Given this intention, the DFrame of course maps well to usage in image processing applications and this section provides a brief overview of the already included support infrastructure that links the more generic DFrame to specific 3D image processing capabilities.

In 3D image processing, data decomposition implies the partitioning of a 3D image and distribution of the parts to independent processes for the computation phase.

It is evident that such decomposition and distribution is useful to a large number of parallel image processing operators and such functionality should be available for reuse to any programmer parallelising 3D image processing applications. So ancillary support for the decomposition and recomposition of 3D images is provided, that can be integrated into models implementing parallel patterns that plug into the parallel framework. As well as flexible strip and block decomposition patterns, support for a 'ghost cell' or 'halo' pattern is provided (Kjolstad, Snir 2010) . Often in image processing, stencils are defined on a local sub volume of an image, and the computation involves the application of this stencil to each cell of the image. When an image is partitioned and distributed, it often is necessary to exchange data at boundaries between neighbours, and an implementation of the ghost cell pattern facilitates this (also referred to as 'halo exchange'). Such partitioning infrastructure bridges the image processing domain into the parallel processing framework. Note that decompositions such as strip and block, and associated ghost cell patterns are generic concepts, and the research efforts include implementations specific to 3D image domain decomposition and recomposition.

The DFrame support for models that encompass parallel patterns of execution and associated task and data decomposition is designed to map well to the requirements of a 3D image parallel processing system. Image operators map to the coarse grain task parallelism, and each image operator can be applied to partitions of a 3D image, executing on independent processes in data parallel fashion. In sum, (at least) two levels of dependencies are distinguished: at a lower level there is the decomposition of a task into sub-tasks that work together on a single parallel computation and often employ data decomposition (e.g. at the image operator level), and at a higher level the composition of these tasks into a task graph that defines dependencies such as the order of application of image operators (more functional centric). The sub-tasks of a single task progress processing in concert using the same parallel model, while each task in a task graph specifies the particular model and hence parallelising patterns it uses.

4.3.3 Technological Choices

As reviewed in chapter 2 the technological choices available to implement the framework are many. Target systems for this research include distributed HPC

clusters of multiprocessor nodes connected via fast inter-node communication networks, so the framework provides core support for the distributed memory model (message passing). Although it is common that processors on one node are able to access shared memory, the framework's initial implementation focuses on a homogenous message passing approach for both inter-node and intra-node communications, relying on the underlying message passing implementation to optimise resource usage. For instance, many message passing implementations can be configured to pass messages via shared memory when appropriate (e.g. IBM Power8 systems (IBM)). That said, the framework is designed to be open such that models directly using posix threads or openMP, and models that harness GPU resources can be plugged in to form hybrid and heterogeneous software systems. Such systems would pave the way to potentially interesting further research.

It should be noted that there is a trade off between honing and optimising code for a particular architecture, and making programs efficient on multiple hardware architectures. This is pertinent as the DFrame provides a more high level facility via a framework, to reduce effort in developing, improving, extending and maintaining an imaging system that utilises parallel processing. A large body of literature emphasises the importance of designing programs to align with the target system's memory hierarchies to arrange optimised flow through the memory caches, and to leverage compiler optimisations etc. (e.g. (National Center for Atmospheric Research)). This is of course in tension with making programs efficient on multiple hardware architectures. The implemented models are targeted at the Kingston University clusters, but other models can be plugged in that are more suitable for different architectures, so there is some flexibility and future proofing. As well, the domain decomposition implementations adhere to interfaces, such that the interfaces can be more long lived while the implementations could be updated to align with different architectures.

A prime goal is to maximise performance, so the framework is implemented in C++, a low level powerful programming language allowing full control to develop very efficient implementations (with commensurate effort). Also, the framework is designed to be extensible, so using an existing well known and supported language should lower the learning curve and encourage contribution to and extension of the framework. Users can create models and applications that can

plug into the framework in a familiar language, and be able to leverage existing domain code and third party libraries (e.g. Boost (Schling 2011)). The target architecture is an HPC cluster (Figure 4.5), and this aligns very well with a SPMD message passing model of parallel computing and so MPI is used to support the message passing model of parallel execution.

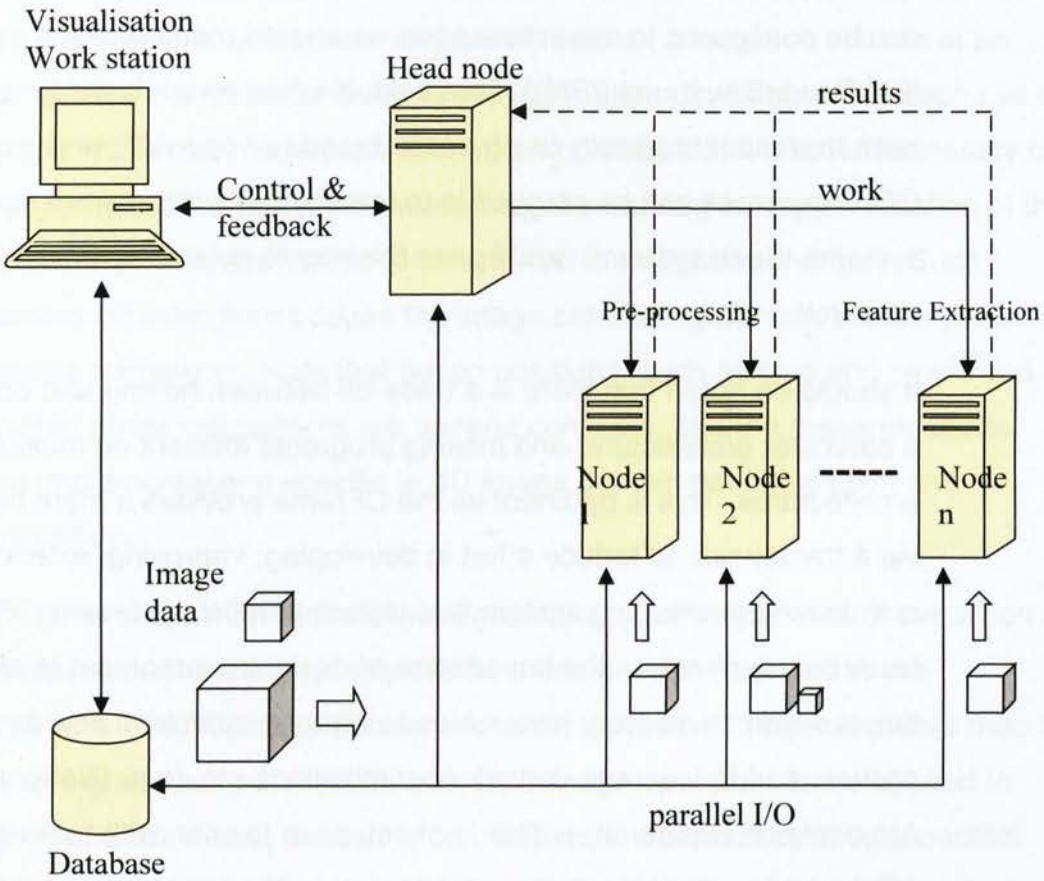


Figure 4.5: Schematic of the Distributed Framework Target Architecture

Complementing C++, MPI is the assembly language of parallel programming upon which very performant implementations can be programmed. MPI is the established de facto standard parallel programming model for distributed memory systems within the HPC community, and similar reasoning applies in that many model developers would be familiar with MPI and able to contribute without the extra burden of learning to use a new technology with an uncertain future. A key feature for libraries writers is MPI's concept of a communicator. Using separate communicators ensures that communication can be scoped to an application or

library and also can provide separation within a library. This concept is leveraged extensively in the DFrame design to provide separation of the DFrame from application code, and more directly within the DFrame to create and isolate process groups that are participating in sub-sections of a parallelised computation.

Finally, C++ and MPI are seasoned open standards driven by a wider cross vendor community.

4.4 The DFrame Architecture

The DFrame's modular design provides an extensible architecture that enables the flexible plugin of models that implement parallel patterns, and the plugin of application code modules. (see Figure 4.6). The DFrame Core is comprised of the runtime management infrastructure, a client proxy and configuration, timing and diagnostics capabilities. The DFrame runtime manages the workflow and plugin components, and drives models implementing parallel patterns, according to client requirements, and integrating timing and diagnostics. A dispatcher component provides high level access to a communication subsystem, providing the means for DFrame instances to interact. The models implementing parallel patterns are also encouraged to use the dispatcher, but the design is open in this respect.

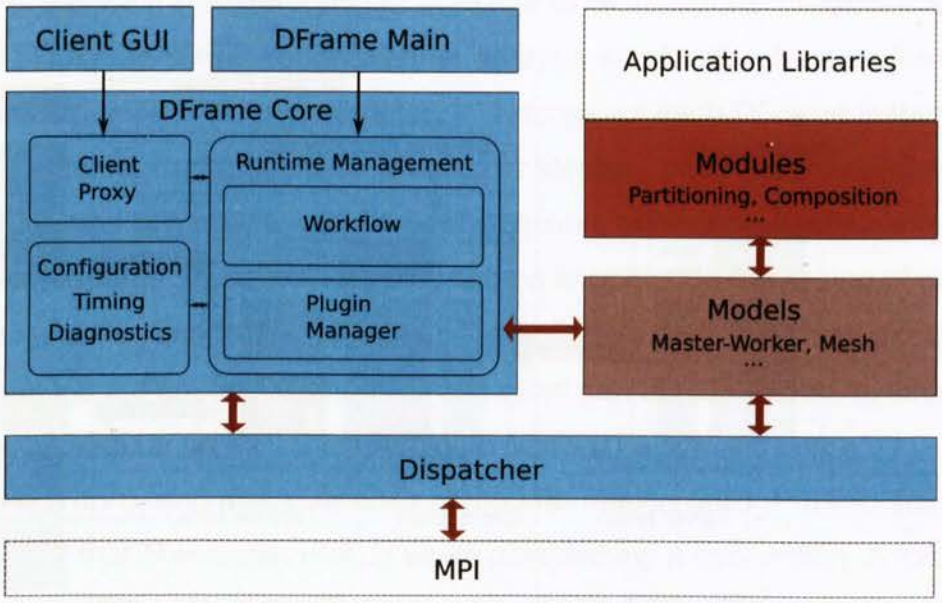


Figure 4.6: DFrame Component Architecture

A client Graphical User Interface (GUI) is also provided to allow a user to author specifications, remotely interface with a DFrame server to run specification graphs, and to view results (data visualizations) which can then reveal insights that guide further exploration.

Before exploring the component design in more detail in the following sections, it will be helpful to first give a brief overview of the DFrame runtime design, to provide context. In Figure 4.7, a simple schematic is shown, that exemplifies the typical arrangement of the DFrame at runtime. The DFrame executable is passed to a runtime system as the MPI program to execute on each processing instance, along with information requesting the number of processes (for example, the DFrame executable is passed directly to mpiexec, or via a cluster manager such as MOAB). One instance is configured to take up the role of 'root process', and it is with this instance that clients interact. All other DFrame instances await further instructions from the root process, that will define how they are to set up and participate in any parallel processing task. The design aligns with the SPMD model, targeted at scalable distributed compute resources and communicating using the message passing paradigm, where the same program is running on a number of separate processes. All the DFrame instances operating in concert to accomplish parallel execution.

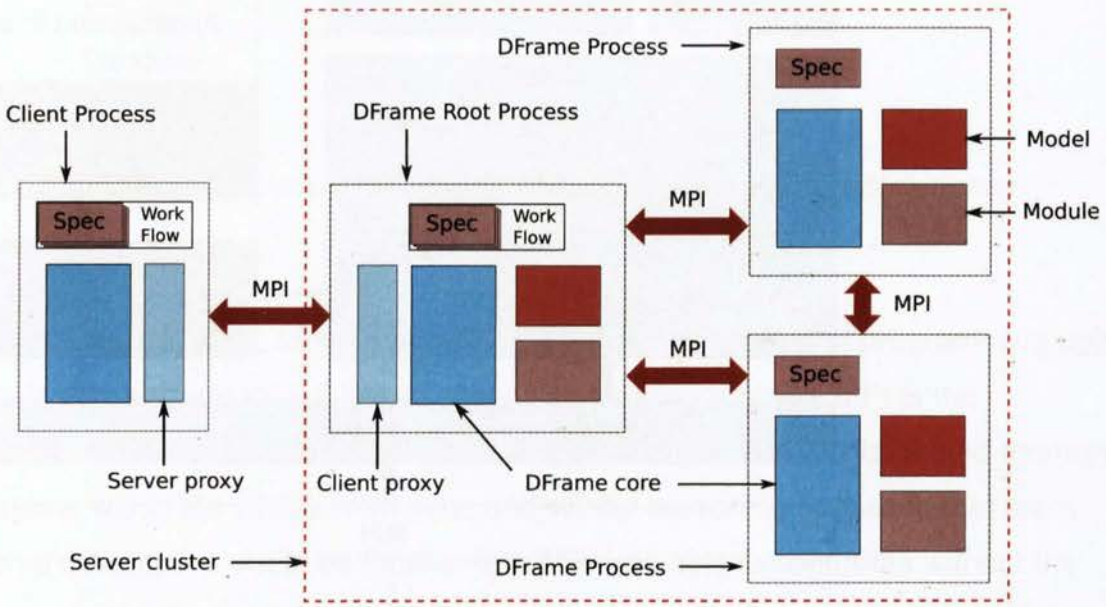


Figure 4.7: Simple Schematic of the DFrame runtime interactions

At runtime, DFrame instances use an inversion of control paradigm to drive models implementing parallel patterns which themselves drive modules that link to application code, which in many cases can be unaware of the context in which it is running. This is in contrast to typical library usage to support parallel programming where the parallelism is woven into the application code via library calls, which can severely constrain further development and maintenance. The rationale for the extensible design is so that models implementing as yet unsupported parallel patterns can be plugged in, together with application code. The inversion of control ensures that the parallel models drive application code, giving guidance and constraint to ensure correct operation, and productive development of novel parallelised applications. Also, as the DFrame is the executable passed to the runtime system, developers do not have to set up a bespoke runtime arrangement for every parallel application. An application developer supplies the application code module, and at runtime provisions specifications that dictate the models and modules to load, together with model and application parameters that specify the runtime set up and execute details. The DFrame runtime will then manage the processing, adapting to optimise execution.

4.4.1 Runtime Configuration

The runtime behaviour of all DFrame instances can be configured. This is accomplished via a configuration file available to all instances. Diagnostics can be enabled to capture timing information for analysis and feedback on performance (with scope to adjust model parameters). Timings for each DFrame instance are gathered onto the root process for viewing or storage, and also provide feedback that the DFrame can use to adapt model selection, parameter assignment and processor resource allocation. Logging can be enabled for debugging of each DFrame instance, and for the plugged in components. DFrame client access is also set up via the configuration. A remote client can be configured, in which case the DFrame sets up an MPI Client proxy to listen on a specific port (MPI_Open_port), and also publishes name information (MPI_Publish_name) to a name server that clients can lookup when establishing a connection. A local client can also be configured, and in this case the required runtime graph of task specifications are stored in a file, and the local client is configured to access this information from the file. The remote client allows support for an interactive experience, whilst the local client is useful for batch (background) processing.

At runtime, each DFrame instance participates in initialising the MPI environment, and reads configuration information to ascertain the runtime set up, including the type of client to support. Only one DFrame instance, referred to as the 'root' or 'server' instance, will set up to support a client and this defaults to process 0. If the client is a remote client, this DFrame will initiate an MPI Client proxy and await connections and run requests. If a local client is configured, runtime specifications will be read from a file.

4.4.2 The Task Specification

In Figure 4.7 , each DFrame instance, including the client is depicted as having a task specification or graph of task specifications (a workflow). This is central to the design of the DFrame. A task specification contains information that brings together the DFrame, a model implementing a parallel pattern, and the application code that the model will interact with. A typical specification will define a model library and a model within that library, and possibly a model group for adaptive model selection. The specification will also provide parameters that will define the application code module, and specific functions within that module, that the model will link to and drive. So the task specification contains information pertinent to the DFrame itself, on which models should be loaded, and information for models to resolve corresponding application code. As well, the specification will contain parameters for the application code itself, such as the input and output data or data location, and other parameters specific to application code control.

4.4.3 The Task Graph Specification and Workflow component

A task graph specification is a graph that defines dependencies between task specifications, and thus allows the composition of complex parallel processing applications. The term 'task graph specification' is used to describe an authored xml specification, and it is realised at runtime as an instance of a DFrame workflow. A task graph specification can be authored in a client GUI or a simple text editor. An abridged sample is shown in Figure 4.8.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<project>
  <nodes>
    <node module="imageTkModule">
      <id>1377719795</id>
      <function name="average3D">
        <task>
          <property name="modelLibrary">mwmodels</property>
          <property name="model">master_worker</property>
          <property name="module">imageTkModule</property>
          <!— other properties omitted —>
        </task>
      </function>
      <layout>
        <x>-217.5</x>
        <y>-10</y>
      </layout>
      </node>
      <!— other nodes omitted —>
  </nodes>
  <edges>
    <edge>
      <source nodeId="1377719795"/>
      <destination nodeId="1377719820"/>
    </edge>
    <!— other edges omitted —>
  </edges>
</project>
```

Figure 4.8: Outline of an typical (abridged) task graph xml file

When a DFrame server instance receives or obtains a graph of task specifications, it is passed to a workflow component on the root process which creates a workflow representation, and orchestrates the runtime order of each task specification, according to the task dependencies implied by the graph. As described, each task specification (i.e. each node in the task graph) defines the parallel processing model to use, the module code to load, and associated parameters.

4.4.4 The DFrame Server Run Loop

A key design of the DFrame is its use of the 'inversion of control' paradigm, where DFrame instances form the distributed executing program that drives application code through models implementing parallel patterns. It does this by initiating a run

loop on each instantiated DFrame instance. Each DFrame instance creates a root DFrame context, in which is stored a communication context, a message packer and dispatcher, and information such as an instances rank in that context. These DFrame contexts can also reference a workflow. In fact each DFrame instance manages a stack of such contexts and this is described in more detail in the DFrameSplitter section below. The usual execution flow is for a root DFrame instance (one whose rank is zero within its context) to receive a task graph specification as a workflow, either from a client, or another DFrame instance. The DFrame will then initiate execution of the workflow within that context. A core insight here is that the workflow is then executed within the scope of a communicator that defines the process group within which the workflow will be processed. Assuming there is more than one process in the context, the execution of each task starts with the broadcast of the task specification to each DFrame instance participating in the parallel processing in that context. In this way, all DFrame instances now have global state within the context, that specifies the parallel model and module code to set up and execute, together with any pertinent parameters. Each DFrame instance inspects the received task specification and extracts the model library and model parameters that indicates the name of the model library and model to use (or model group to select from), and with this information loads a model instance, passing the task specification as an initialiser parameter (in the implementation, the model plugin manager is called to retrieve an appropriate model factory that can create the specified model). In this way, all DFrame instances load the specified model from the appropriate model library, and call the models 'run' method, passing the task specification. Each model instance drives application code and manages parallel processing interactions according to the parallelising pattern defined by the model. The DFrame monitors the time taken to run each parallelised task, and can both report this information and potentially use it to adjust further processing.

In this way, the DFrame core provides a lightweight but powerful mechanism for setting up extensible parallel programs using the SPMD programming model, and orchestrating the running of multiple such programs according to the task specification graph.

4.4.5 The Plugin Manager

A central feature of the DFrame architecture is the ability to plugin in new models for different parallel patterns or variants of the same pattern that use interfaces defined by the DFrame. As well, application modules that use these models are plugged in using the same approach, implementing interfaces defined by the models that they plug into. Dynamic library loading is utilised to effect this extensibility (Open Group (Reading 2013). Separate to the configuration file, model plugins are specified in a modelPlugins.txt file that maps a model name to the implementing library. Similarly, modules are added to the system by updating a modulePlugins.txt file that maps module names to the implementing library (modules will likely be more numerous than models, as many will reuse the same model). Using a plugin manager, DFrame instances load these libraries, and maintain a cache mapping of names to libraries, to provide access to the required libraries via name lookup. This is core to the functioning of the DFrame, with task specifications containing name information on which model and modules to load and run, which is more flexible than directly looking for library files.

4.4.6 The DFrame Dispatcher (communication)

The DFrame dispatcher is designed to provide a higher level of abstraction to the communication layer, shielding the programmer from lower level message passing libraries. The design facilitates the passing of message objects between processes, rather than using low level facilities directly. This has a twofold advantage in that it affords a more high level simpler communication abstraction to the DFrame core, the models and modules, and secondly that it separates the low level communication from the DFrame, allowing scope to swap out the low level message passing subsystem (adhering to good design practice). Messages and the associated message packing design are integral to the DFrame architecture. A simple schematic of the message packing and unpacking protocol is shown in Figure 4.9. The dispatcher delegates message packing to a packer object that serialises a message object to a packed array for sending, and deserialises a received packed array to a message object. The packing mechanism relies on the provision of an array of MessageInfo objects that describe the structure of the message contents (the type, size and data of a message's fields). To facilitate object reconstruction, the first two entries in the packed array specify the module

name and object class id, such that the packer can resolve to and call the correct object implementation when deserialising an array of MessageInfo objects.

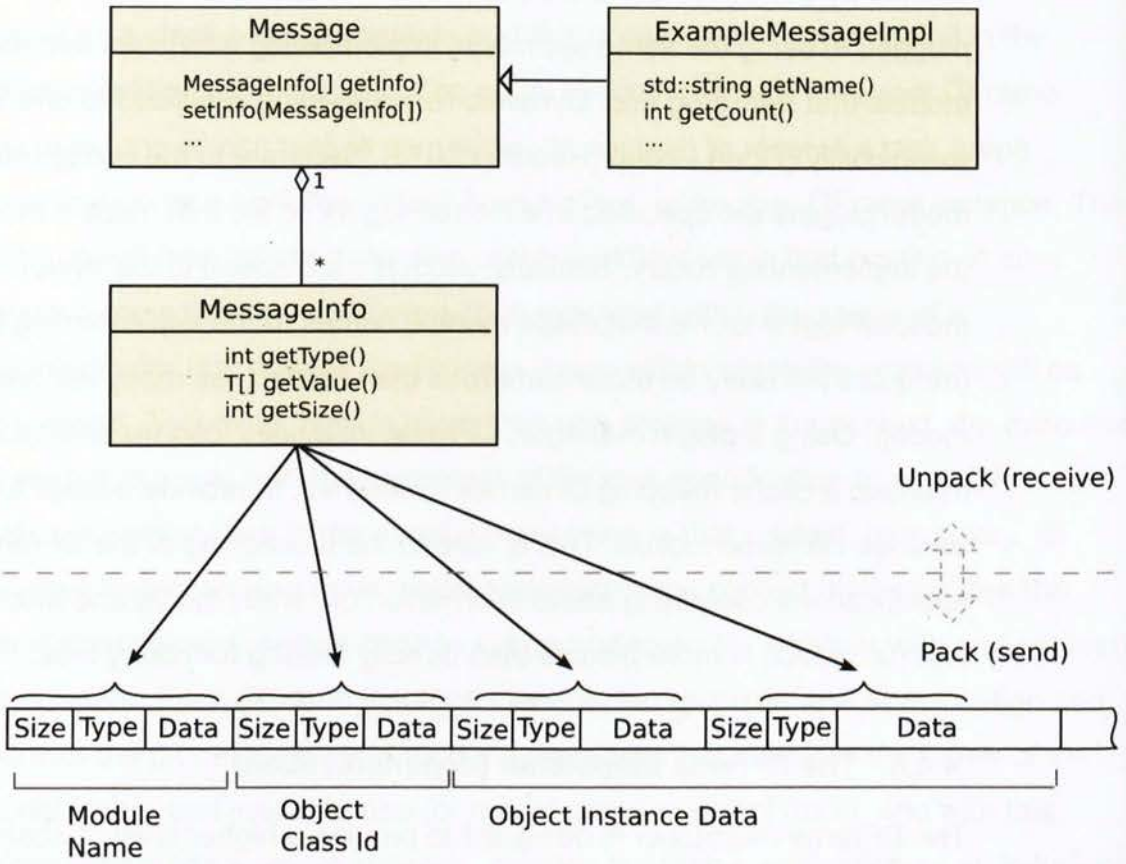


Figure 4.9: Message packing and unpacking protocol schematic

The dispatcher provides higher level methods for non blocking and blocking point to point communications, and methods for blocking collective communications, that accept and return message objects. Using the packer mechanism, the dispatcher converts the message objects from and to message buffers used by the underlying communication mechanism, which in the current implementation is MPI. Indeed, the methods exposed by the dispatcher map closely to the underlying MPI specification, wrapping to provide the higher level abstraction. In the case of the non-blocking point to point operations, higher level abstractions are also provided to allow users to test and wait for calls to complete. When appropriate, this leverages a powerful feature to allow the caller to do useful work in parallel with ongoing communication.

4.4.7 Tasks, Partitioners and Composers

Before going on to look at models in more detail, it is first helpful to describe the DFrame's generic 'Task' interface. First, some care has to be taken to distinguish the concept of a Task in the DFrame from that of a task specification. While a task specification defines the set up and interplay between the DFrame, models and application code, the Task interface describes a generic way for models to interact with application code. In order to arrange for parallel execution, an application needs some means of expressing how it should partition, execute and compose its parallel computations. This can be done in a bespoke manner, and aligned with custom models, but a more generic interface allows the possibility to use any model that supports the interface. This allows both for reuse and also opens up the possibility to adapt the selection of appropriate models for example to improve performance. Such an approach also guides the application developer.

The Task interface and its core ancillary classes is shown in Figure 4.10. Essentially, these classes define the partitioner, executor and composer components of a parallel execution at the model-task level. A model that works with the Task interface can access a known Partitioner interface, and thus be able to request partitions from application code without knowing the details of the implementation. It is up to the the application module code that links to the model to provide an appropriate implementation. Similarly, the application module code implements the Composer and Executor interfaces. Thus models using these interfaces can apply parallel patterns to application code generically. It is the models themselves, in collusion with the DFrame that will arrange for the distribution of the retrieved partitions, the delivery to application code for execution and the subsequent gathering and redelivery to application code for composition.

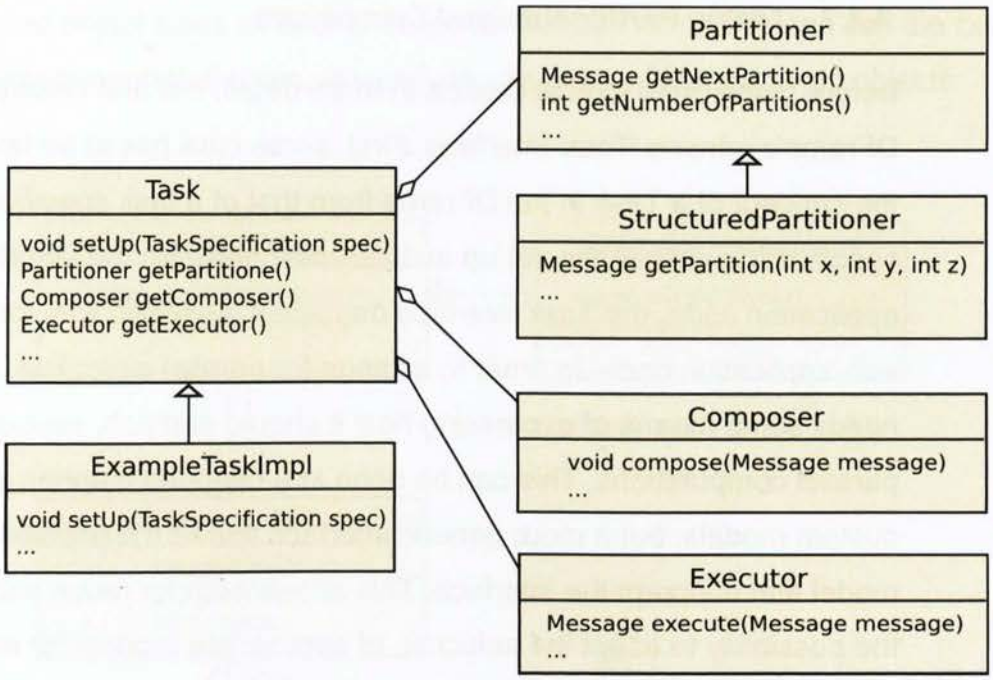


Figure 4.10: Class diagram of the DFrame Task interface and ancillary classes

4.4.8 Models

Model libraries are developed separately to implement parallel patterns of execution, and are subsequently plugged into the DFrame. A model library can include more than one model, and so must expose all its available models and allow model instantiation through DFrame defined interfaces. A DFrame defined mechanism following the factory pattern is used to instantiate model instances (Gamma, Helm et al. 1995). Indeed, models are interposed between the DFrame core and the application modules, and as such must support two interfaces. Firstly, a model must implement a 'model' interface to allow integration into the DFrame runtime, such that the model's lifecycle can be managed by the DFrame. Secondly, a model defines its own interfaces that application code using the model must implement. As already described, the DFrame also defines a generic Task interface that models can instead choose to expose to applications, and if this mechanism is used, the model can become much more flexible and reusable (although encouraged, this is not mandated so as to keep the design 'open'). The DFrame-Model interface allows provision of the dispatcher for the models use, and optionally the setting of a client. DFrame instances then manages the runtime behaviour, distributing tasks, resolving and loading the particular model specified for the current task, and each participating DFrame instance then runs the same

model, effectively handing over control to each model instance. The model is then in charge until it returns control to the DFrame. Models already provided and described in the following sections include a special task splitter model that integrates closely with the DFrame, a master-worker model, a variant scatter-gather master-worker model and a mesh model. Note that the core design of parallelising models is in the expectation that there will be more than one process available to a model. However, this may not always be the case, and in a splitting scenario, a model may only have one process made available to it, and in this condition it would arrange to run its computations locally.

4.4.9 The DFTaskSplitter Model

As already described in the concepts section above, the DFrame manages the running of task specifications via models, and at a higher level the DFrame also manages the assignment of process groups to the running of each task specification according to the task graph specification, and accounting for the number of processors available. This process group management is handled by a DFTaskSplitter. In order to maximise flexibility, and to align with the DFrame design, the DFTaskSplitter is actually implemented as a model that plugs into the DFrame, and provided by a core DFrame model factory. A DFTaskSplitter can be automatically invoked when a DFrame instance managing a workflow encounters an explicit split in the task graph and thus determines that branches of the graph can be run concurrently. Being implemented as a model, the DFTaskSplitter can also be explicitly defined in a task specification as part of the task graph. This is very useful in cases where the task graph itself does not explicitly represent multiple branches, but where the introduction of an explicit splitter can generate multiple branches dynamically. Figure 4.11 (a) shows a snippet of a task specification graph that employs implicit splitting. In this case the DFrame will automatically insert a splitter. From a task graph point of view, this behaviour is considered as static, since branches are all predefined. In Figure 4.11 (b) a splitter is explicitly referenced in a task specification, and is run by DFrame instances as a normal task that invokes a splitting model, and instigates the model's associated behaviour to generate task branches dynamically.

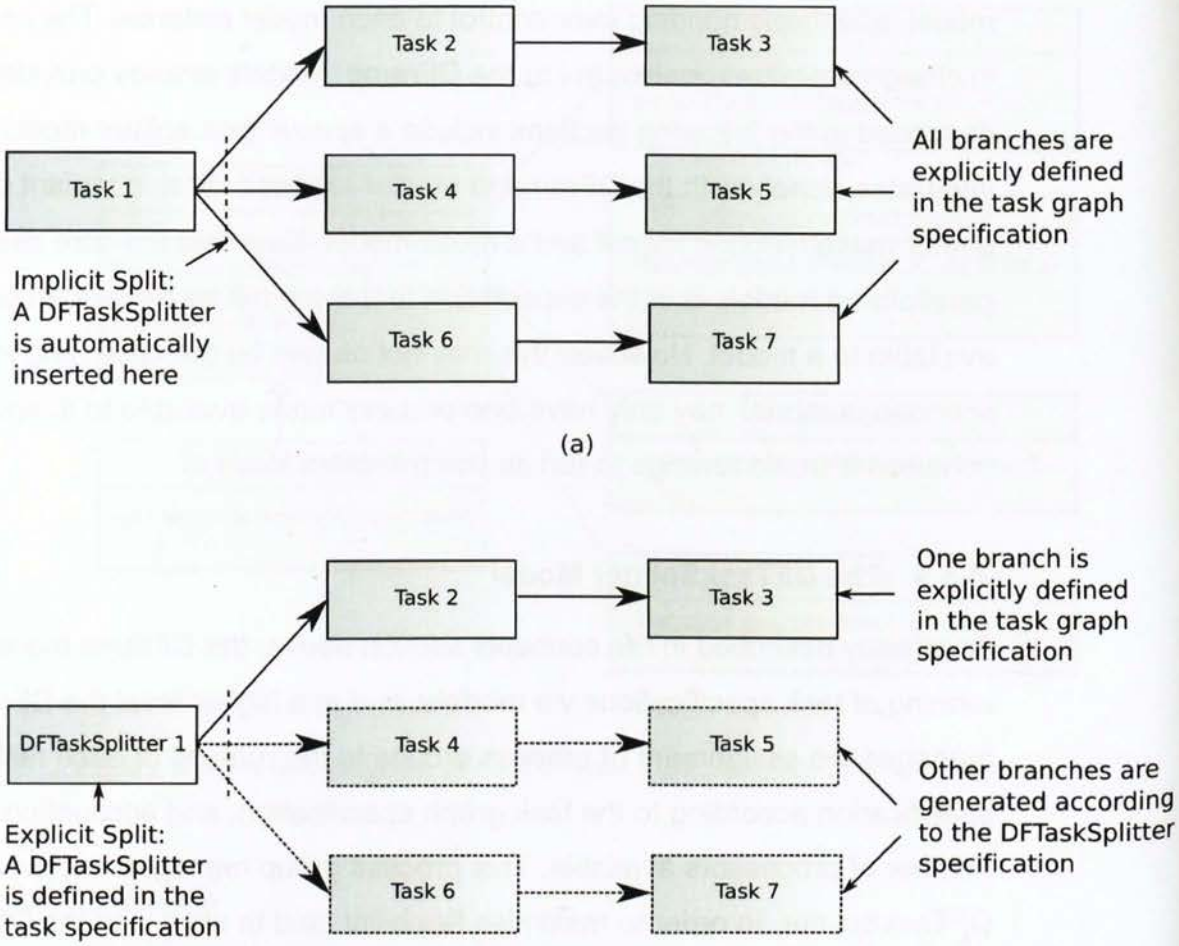


Figure 4.11: Task graphs showing implicit and explicit splitting

When the DFrame determines that a splitter should be invoked, either implicitly or through an explicitly defined task specification, the splitter task is handled in a similar manner to normal task specifications. Namely, the splitter task is broadcast to all DFrame instances participating in the current context, and each instance then identifies the model to load from the task specification. In this case, the DFTaskSplitter is loaded on each instance (hence is designed as a model, to dovetail in with the DFrame core design). Internally, the current implementation of a DFTaskSplitter uses the MPI_Comm_split collective operation to effect the splitting of a process group into sub process groups. This is a general and very powerful feature of MPI. Ordinarily, the splitter will split the process groups into equal sub groups according to the number of task specification branches at that point in the task graph, or into sub groups according to the dictates of a dynamic

task splitter. However, information can also be provided in the distributed task specifications that can modify this behaviour. As well, for an explicitly defined dynamic splitter, application partitioner code can provide information on the number of task branches to create, and possibly the size, in terms of expected computation effort required in each branch. Application code can also provide other parameters to propagate into the generated branches (all this leveraging the normal DFrame design of model-module interaction).

In operation, a splitter splits the current process group into a number of sub-process groups. The root of each sub-process now needs a workflow itself, to run, and this is arranged from the root process of the parent group that is driving the split. At the point of the split, this root can infer the root process of each sub group root and can thus send it a sub-workflow. The sub-workflow is extracted from the main workflow being managed by the parent group. The functionality to extract a sub-workflow at a particular location in the parent workflow is implemented in the workflow component. The splitter then updates the task starting the newly extracted sub-workflow with any defined parameters, and sends to the root process of each newly formed process group. The managing process also sets in a sub-workflow to itself. At the same time, each DFrame instance creates a new DFrame context and sets as its current context. This new context is added to the DFrame instance's context stack and is where the newly created communicator is cached. Each new process group is now running as an isolated process group, with its root process managing the running of a sub-workflow and distributing tasks to other processes in the group according to the normal DFrame core design. As the splitting design distributes control, it can also help reduce the likelihood of the root process or group root processes becoming a bottleneck.

The splitting of process groups into sub-process groups can be repeated to form a hierarchy of sub-process groups within process groups, with each DFrame containing a stack of DFrame contexts that represent the current state of the splitting. As sub-workflows complete, contexts are popped off the context stack, and a DFrame instance then reverts to the next context up the stack which will represent the parent process group, and this will continue until a DFrame is again operating within the root context (the global process group).

As well as the DFrame context that manages communications across process groups, the DFrame also has a facility to cache model contexts. Model contexts

are used by models to cache and propagate data from task to task, and this feature would also be leveraged to support more complex task graphs.

The current design of the DFTaskSplitter allows for the splitting of a regular task graph where the dependencies of the task are such that sub-branches can be independently run. In a more general case where a task graph has very irregular and complex interdependencies, the splitting of process groups would be even more involved. A modified splitter can be implemented and plugged in to cater for such cases and it is envisaged that this would likely make use of MPI's inter-communicator features to support more convoluted task interactions. This is not pursued further, as the current design more than caters for the intended target usage, but it is acknowledged as an interesting avenue of further research.

4.4.10 The Master Worker Model

The master-worker model is the ubiquitous model of embarrassingly parallel computing. It is a versatile model, especially suited to managing tasks with no inter-task communication. Each task computes a sub-problem, and returns the result. In this arrangement, the master becomes the centralised task scheduler. A prime attraction is that load balancing becomes automatic, with each worker either explicitly requesting another task once it has completed its current task, or is automatically given another task once it returns the results of its current task. The master is responsible for managing the distribution of tasks and signalling to workers when all tasks have been completed. In some circumstances, the manager can become a bottleneck as it has to resolve the tasks to execute and communicate with each worker to distribute tasks and gather results. When this is an issue a hierarchically distributed master-worker variant can be used such that a supervisor manages multiple master processes, which themselves manage a subset of worker (Aida, Natsume et al. 2003). Other alternatives include employing a distributed task scheduling algorithm where each process manages its own task queue, with processes taking work from, or offloading work to neighbouring processes depending on workload (in this arrangement, a mechanism is required to detect when a computation is complete).

As a first model for the DFrame, a simple master-worker model is implemented. In Figure 4.12, a simplified sequence diagram shows the DFrame interactions to set up and run the master-worker. This model uses the DFrame's generic Task api,

and in the simplified schematic, interactions on the Task are shown, with the associated partitioner, executor and composer interfaces excluded for clarity. Using the Task api increases the reusability, flexibility and adaptability of this common parallel programming model. The DFrame receives a task graph specification from a client, and from this creates a workflow representation and hands it to the workflow component. The DFrame then requests the next enabled task specification from the workflow, and broadcasts it to all participating DFrame instances. Each DFrame inspects the task specification to establish the appropriate model library and resolves and loads the specified master-worker model from that library, and runs the model. Each model instance then queries the DFrame to similarly resolve the specified module library to use. By default, the root process model runs as master, loading the master application code from the module factory (the process to run as master can be configured). All other models run as workers, loading the worker application code similarly. The master then requests work from the application code (via a task's partitioner), and distributes to worker processes. Workers complete each received parcel of work and send results back to the master which itself passes on to application code (via a task's composer). While there is more work, the master continues to send work to workers, and the workers continue returning results. Once a computation is complete, the master sends a 'no more work' signal to the workers. The application code is primarily responsible for managing the results, but the model can signal the client that the current task has completed and also send data, depending on the specification. Figure 4.13 shows the corresponding simplified sequence diagram for workers engaged in the master-worker model. The processing is similar, but without client and workflow interactions.

Like many approaches to parallelising applications, the master-worker model brings to the fore the required explicit partitioning of tasks and data. Although the onus remains with the application module code to partition the problem into tasks that can be distributed for parallel processing, the master worker model adheres to the defined Task interfaces, so the application provides this partitioning through defined generic interfaces. In the targeted imaging applications, this means the partitioning of the 3D images and handing of the partitions to the master which in turn sends to available workers, and also providing the task executor (worker implementation) to apply to each partition. After the execution phase, the results are handed back to the application module's implementation of a generic

composer interface, which then assembles, commonly into a resultant transformed image. It is recognised that implementing partitioning and composition patterns is onerous in itself, and by using the generic partitioning and composition interfaces, an application can provide implementations that can be reused. This is a useful feature of the DFrame design, in that various partitioning and composition strategies can be implemented once, and reused transparently thereafter. In a subsequent section, a rudimentary imaging toolkit is described that provides some of this partitioning infrastructure for reuse in a 3D imaging system.

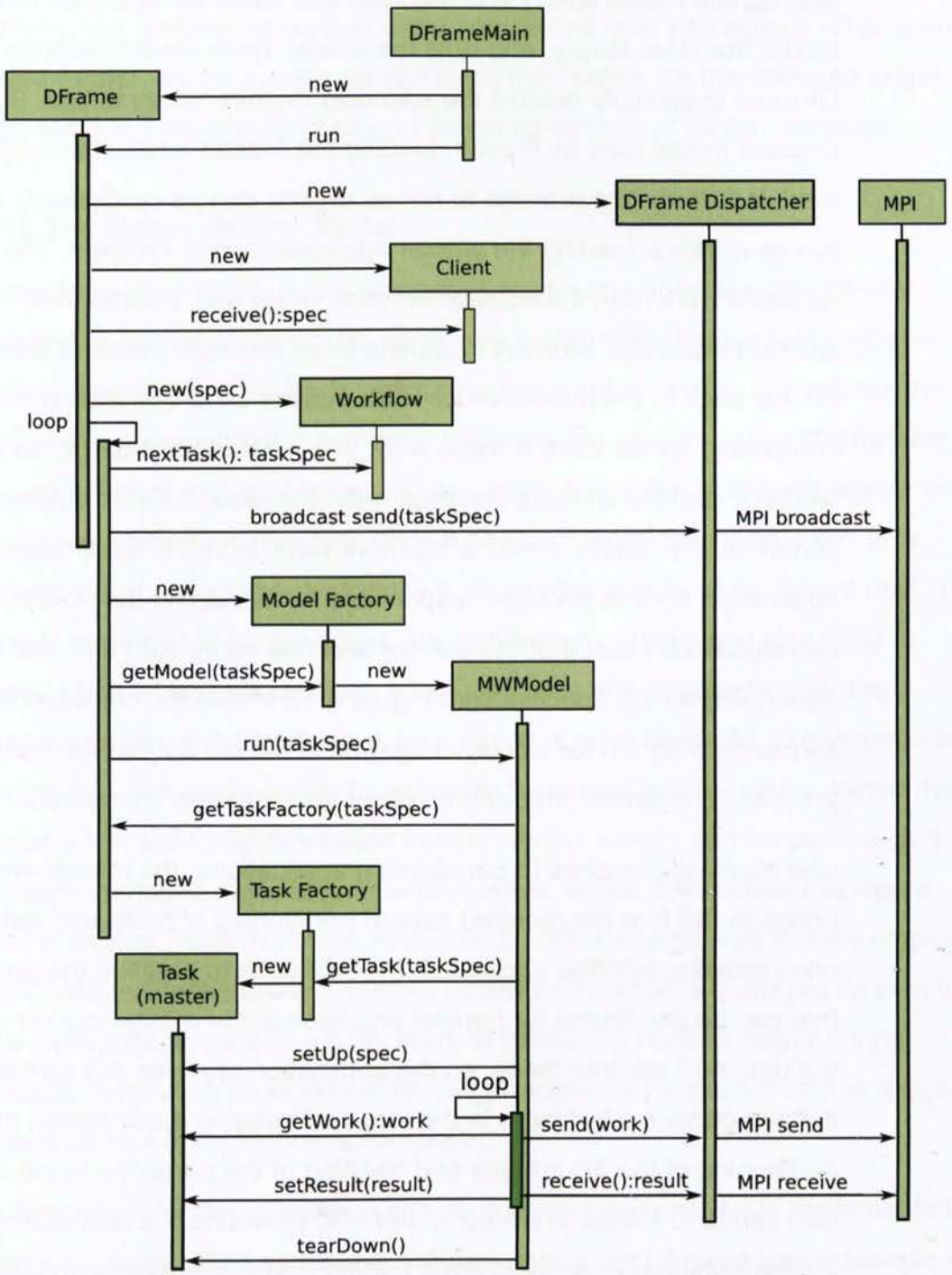


Figure 4.12: Sequence diagram master-worker model: main interactions of a master

Only the essentials of the DFrame operation pertinent to the lifecycle of a model are shown in the schematics. Mechanisms such as the recording and collection of timing diagnostics are not shown, as this would obscure the core interactions relevant to running the models. Similarly, detail such as the message packing and unpacking is omitted for clarity.

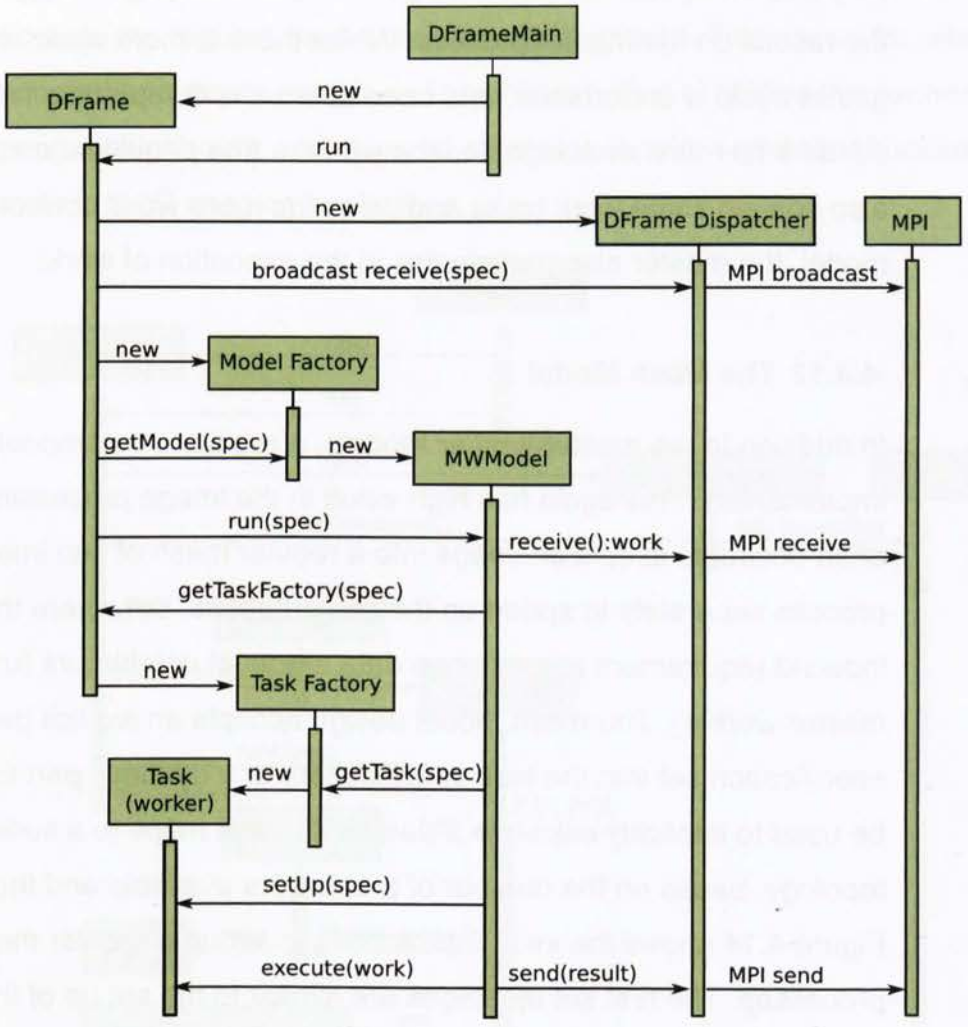


Figure 4.13: Sequence diagram master-worker model: main interactions of a worker

4.4.11 The Scatter Gather Master Worker Model

A variant master-worker model is also implemented, that uses collective communications. This model assumes that all the tasks are of similar size, taking the same processing time to complete. In this case, instead of distributing tasks

individually to each worker in a round robin fashion, the master can request from application code a number of tasks equal to the number of workers and the master. Then using a scatter collective operation, all the tasks are distributed to the workers in one call. The aim being to leverage the optimised collective distribution implementations of the underlying communication mechanism (e.g. MPI_Scatter). In the same way, once workers have processed the received tasks, they all participate in a collective gather operation (e.g. MPI_gather), to collect all the results on the master process. Whilst there is more work, another scatter gather cycle is undertaken, and once all work is complete, a final scatter-gather sends a no more work signal to the workers (the penultimate scatter gather may also contain some work tasks and some 'no more work' control tasks). In this model, the master also participates in the execution of work.

4.4.12 The Mesh Model

In addition to the master-worker models, a regular mesh model is also implemented. This again has high value in the image processing domain, as it is often desirable to split an image into a regular mesh of sub image partitions and process separately to speed up the computations, but where there is then an induced requirement to exchange data amongst neighbours (unlike the simple master-worker). The mesh model design accepts an explicit partitioning specification set into the task specification, or a dynamic partitioning strategy can be used to implicitly calculate a partitioning that maps to a suitable processor topology, based on the number of processors available and the image shape. Figure 4.14 shows the initial interactions to set up a regular mesh model amongst processes. The first set up stages are similar to the set up of the master-worker model, with the DFrame receiving a task graph specification, from which a workflow representation is created and handed to the workflow. The DFrame runtime then retrieves the next enabled task and broadcasts to set up the task run. In this case a mesh model is then resolved, loaded and run on each DFrame instance, with each model itself then requesting the appropriate module code to run (application code) via the DFrame, again according to the task specification.

The root process of the mesh model first instigates the loading of the data (e.g. a 3D image) and determines the partitioning information, either explicitly from the specification or from an associated dynamic partitioning strategy. When chosen,

the dynamic partitioning of the data will depend both on the shape of the data itself, and on the number of available processors. The result is an appropriate processor topology specification, which is broadcast to all DFrame process instances that are to participate in the collective operation to set up the mesh. If the number of processors exceeds that required by the defined topology, then only a subset will be used (if there are not enough processors, then an error is reported). Under the covers, the implemented mesh model uses MPI's topology API's to create the regular mesh communication pattern. This allows for relatively straightforward coordinate indexing of neighbours, that simplifies the exchange of data amongst neighbours, which would otherwise require significant bookkeeping for example in a 3D imaging application.

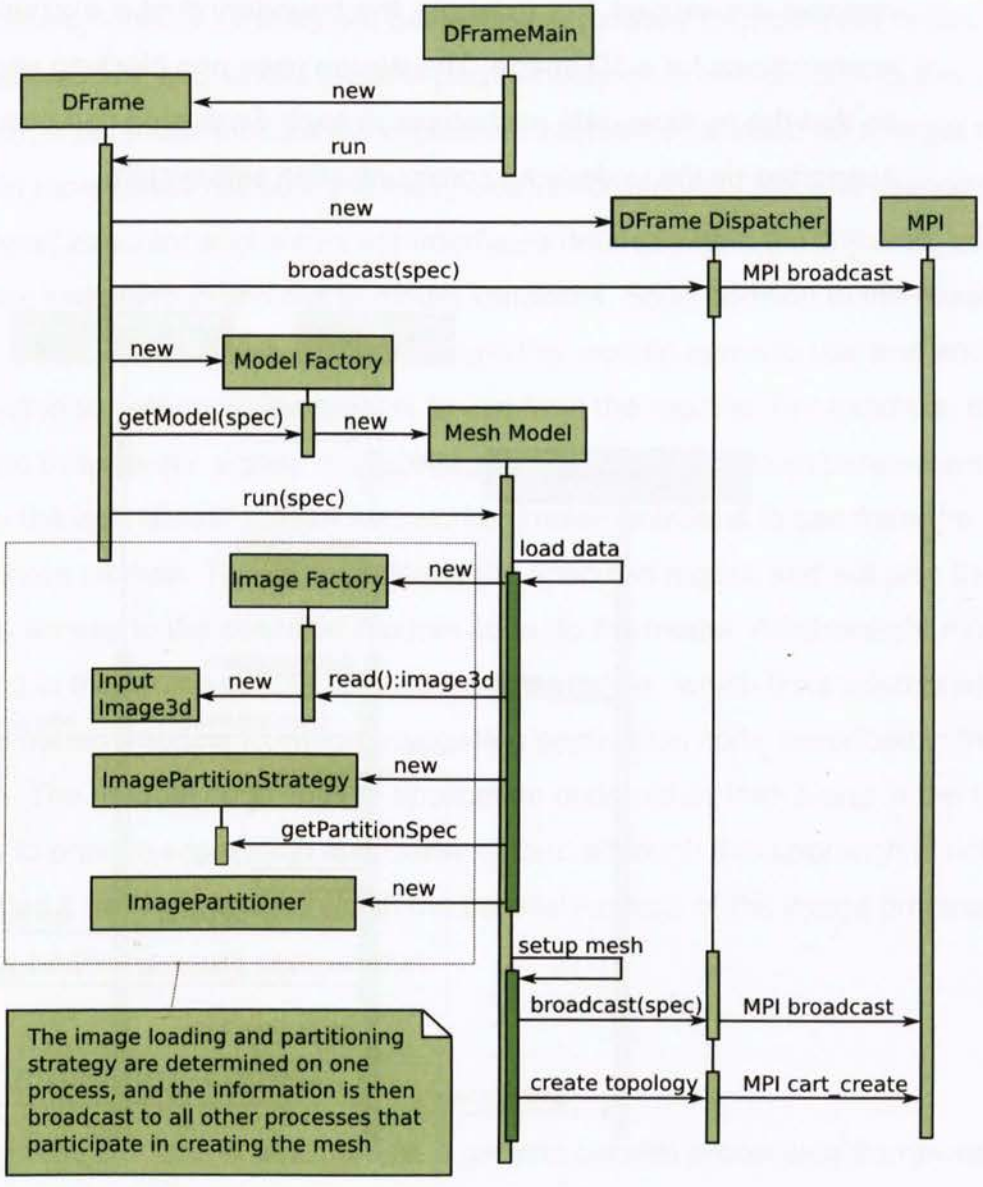


Figure 4.14: A sequence diagram showing the initial setup of a mesh model

After the initial setup of the mesh topology, the data can be partitioned and distributed as it is now known to which processor each partition should be sent. This is now quite simply arranged using the coordinates generated when setting up the mesh topology, which affords a much more natural way to reference partitions to processors. Once, all the partitions (e.g. 3D sub-images) have been distributed to the requisite DFrame instances, the main execution loop of the mesh model is entered. This proceeds to execute each local task (e.g. an image operator), on the received partition. After each execution, and assuming there are further executions, an exchange data step is initiated between neighbours so that results in one partition can be propagated to neighbouring partitions. The exchange of data proceeds along each dimension, such that boundary data (ghost regions) is swapped. For example, the boundary data is exchanged in the x, y, and z dimensions for a 3D image. The design uses non blocking sends and receives so that the multiple data exchanges in each dimension can occur concurrently if supported by the underlying communication sub-system.

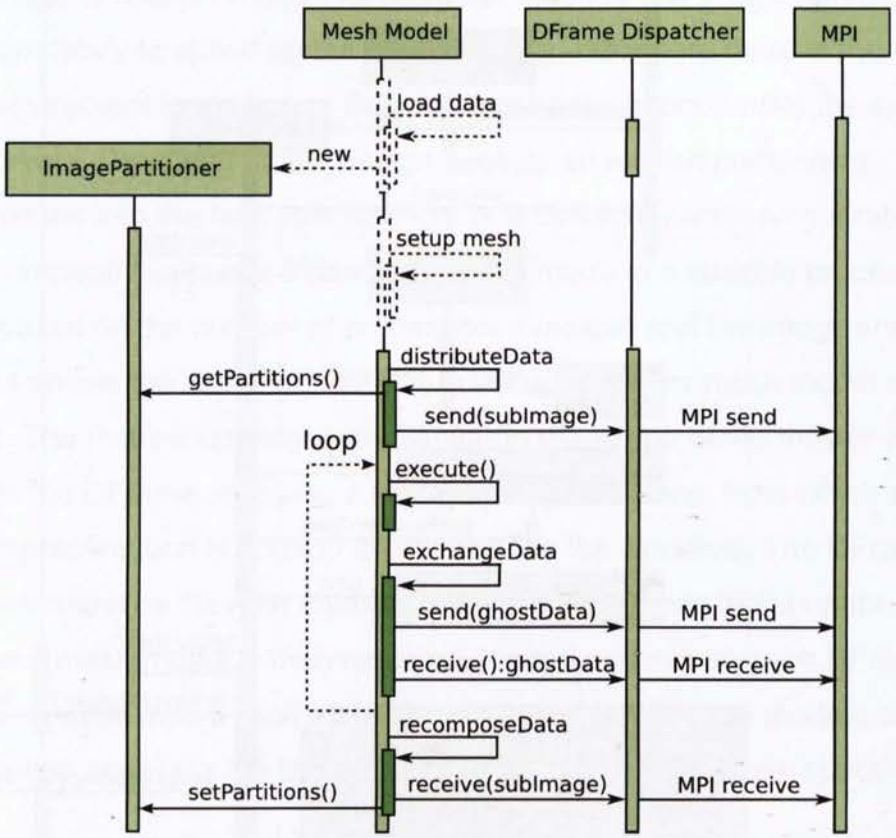


Figure 4.15: A sequence diagram showing the processing stages of a mesh model

One facility being woven into the DFrame, is the ability to distribute further tasks (e.g. image operators) to already distributed data. This is pertinent to the mesh model, as, in addition to multiple exchanges of neighbour data during the parallel processing of one operator, an exchange of data between separate operators can be arranged that avoids the overhead of gathering and then redistributing data between successive operators in a pipeline for instance, that require a similar communication pattern. Once all processing and exchanges have completed, the resulting partitions are normally gathered onto the root mesh model process, for delivery to application code, or an explicitly stipulated target output.

4.4.13 Modules

Like models, module libraries are developed separately to implement or link to application code, and similarly use a plugin mechanism to register with the DFrame. A particular module will implement interfaces exposed on a target model which in most cases will be the already described generic Task and associated partitioner, executor and composer interfaces defined within the DFrame, so as to allow the swapping in and out of model variations. So in addition to the model name, a task specification will also contain the module name to use and ancillary information to determine the objects to use from the module. For example, to interface to a master worker model, the specification will contain parameters to identify the appropriate master and worker implementations to use from the appropriate module. The DFrame loads the specified model, and will also then provide access to the specified module code, to the model. A lightweight module is supplied in the prototype, the 'imaging toolkit module', which links appropriate parallel pattern models to image processing application code described in the next section. The module code links to application code rather than being in the same library, to provide separation and clarity of use, although this approach is not mandated it can help to distinguish the parallel aspects of the image processing such as partitioning and composition.

4.5 *An Imaging Toolkit*

Although the DFrame is designed as a generic parallel processing framework, the impetus for its inception is to support the parallelisation of image processing applications, and in particular to speed up the compute intensive high content

screening of 3D images. To this end, application infrastructure code supporting 3D image storage and retrieval, 3D image data manipulation, partitioning and recomposition is provided in a basic 'imaging toolkit' library. The 3D image IO design is an adaptation and port to C++ of ImageJ's Tiff encoder and decoder software. The adaptation being to load from and save to the toolkit's high level 3D image objects. The 3D image object design allows for direct access to image voxel data using a more natural 3 dimension indexing (i.e. identifying an image voxel using x,y,z coordinates). This is accomplished in the implementation by an integral 3D pointer structure.

A prime use case is to be able to geometrically partition large 3D images into sub-images that can be sent to separate processing elements that then apply a compute stage to each sub-image. Subsequently, these sub-images often have to be gathered and recomposed into a resultant image. To address this, a prototype 3D image partitioning component is incorporated into the imaging toolkit, that includes a 3D image partitioner and composer along with an extensible Partitioning Strategy design. The Partitioning Strategy design separates out the strategy from the partitioner and composer, so that different partitioning strategies can be devised and injected in the future. The partitioner and composer can be explicitly set with a partitioning specification, or a partitioning strategy instance can be injected to take care of this aspect more automatically. Once a partitioner has an associated 3D image and a partitioning specification, sub-images can be retrieved from the partitioner for distribution. The partitioning specification can also define sub-image 'ghost cell' requirements (boundary cells), and if specified the partitioner will return expanded sub-images that include the ghost cell data (geometric decomposition and ghost cells are commonly used together). The 3D composer inverts the partitioning, recomposing sub-images into a complete 3D image again. Partitioners and composers normally come in pairs, such that a composer is then recomposing sub-images from a corresponding partitioner, using the same partitioning specification.

Although the partitioning in the imaging toolkit is specific to images, the concept is generalised, being abstracted out to a more generic partitioning api, so that models can directly use the partitioning functionality without knowing about imaging per se. The mesh model uses the partitioning information to set up an appropriate topology and distribute sub-images transparently to application code.

Furthermore, the mesh model arranges for the exchange of sub-image boundary data on completion of a computation step, again transparent to application code. This separates the domain application from the parallel model and allows the model to move data around on behalf of and transparent to application code, but the model must of course have information on how to partition applications data, and this is via a generic api, with an application supplying an appropriate implementation. The partitioning functionality can be reused across a broad range of image processing algorithms. That useful communication patterns are captured, and the implementations to effect and support these patterns specific to image processing are provided means that they can then be tested and improved separately to the domain code that ultimately uses them.

As well as the imaging infrastructure above, that can be used across many algorithms that parallelise 3D image processing, the imaging toolkit also provides some specific image processing capability that uses this general functionality. In particular, 3D averaging filters and gradient local image operators, a rudimentary watershed segmentation implementation, and ray casting are provided. These are described in more detail in the case studies in chapter 5, where they are used to test the DFrame performance, and to inform on improvements to the DFrame design itself, the associated models and indeed the imaging toolkit infrastructure design. Many other useful 3D image processing algorithms amenable to parallelisation can be found in the literature (e.g. (Nikolaidis, Pitas 2001)).

4.6 DFrame Graphical User Interface

A GUI client is implemented in QT4 (Dalheimer 1999) to allow users to visually construct a task graph that defines a task specification (see Figure 4.16). The main components of the GUI are the Task Graph View, and an associated Tasks View and Task Properties editor.The GUI links to the DFrame library and thus has access to information about the available model and module plugin libraries. The GUI requests information on the functions that each module supports, and makes this available in the Tasks View, for composition such that users can drag and drop tasks onto the graphical surface of the Task Graph View, and add links to define the dependencies. Selecting a Task deposited on the Task Graph View will display its parameters in the Properties Editor, which essentially describes the specification for a single task in the task graph. The parameters that govern the

running of that task can then be further edited.

In the Task Graph View, directed edges can be added that link tasks together to define the dependencies in a directed acyclic graph as required for the particular application. The ultimate purpose of the GUI is to build up an xml file that stores the task specification of each node of the graph that is to be run in parallel, and also stores the links between the nodes of the graph, that define the dependencies between tasks. These xml files can then be stored to disk, and reloaded and reused. Moreover, these graphs can be sent to the DFrame server, where they are loaded as workflows to be run.

When authoring a graph of task specifications describing a processing pipeline, there is no need to connect to a DFrame server (so a server does not have to be running), but a user must of course connect to a running DFrame server in order to run the task graph. A separate facility is provided such that the user is in control of when to connect to a server (also depicted in Figure 4.16). Once connected, graphs can be run repeatedly, and when finished, the user can then choose to disconnect from the server. Closing the client will also disconnect it from the server.

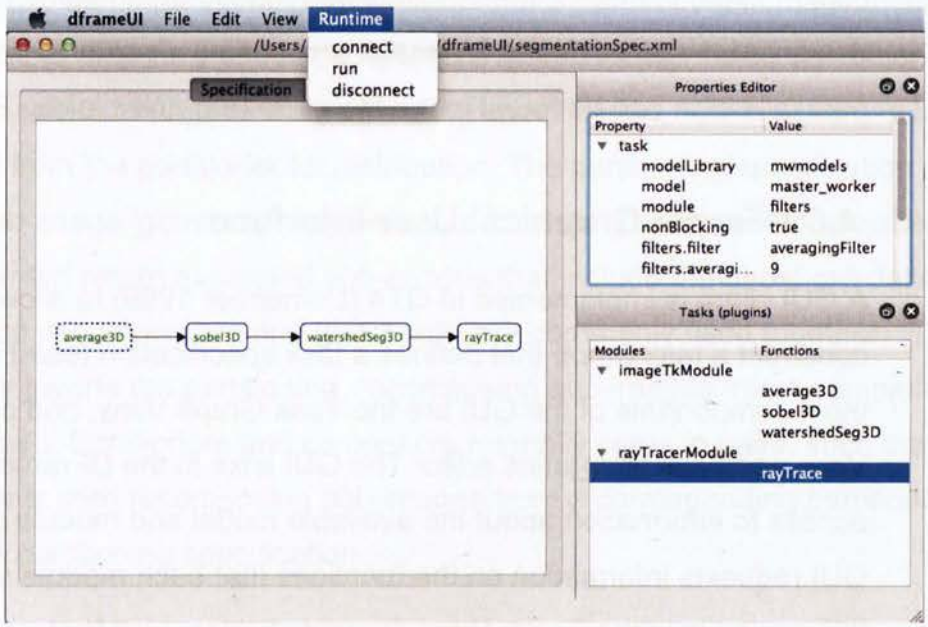


Figure 4.16: The DFrame Graphical User Interface

At the GUI level, image processing users are shielded from the complexity of parallelism and can concentrate on composing and tuning tasks as required for a particular application. Additional functionality is also provided such that a user can view the output image after running a task specification when appropriate.

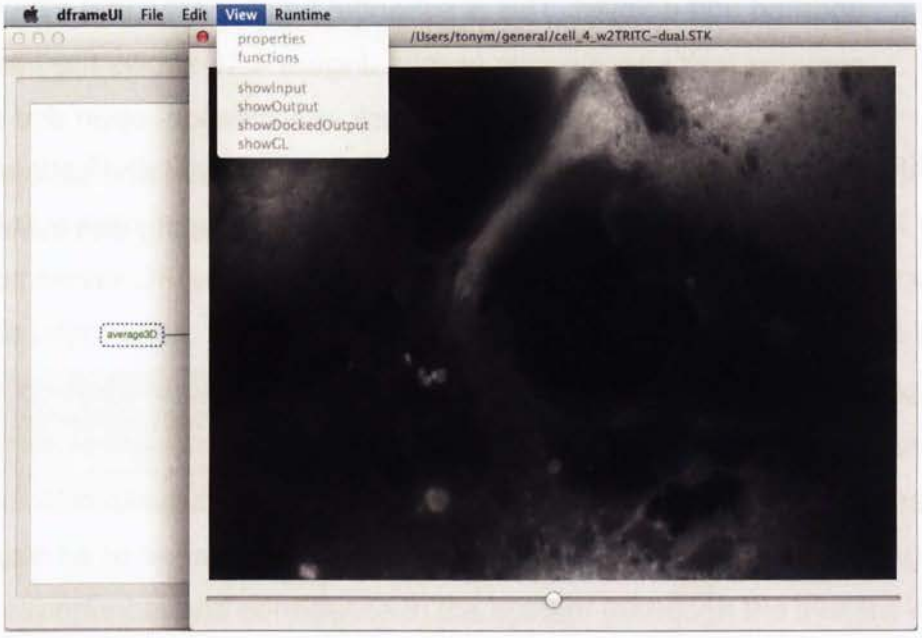


Figure 4.17: The GUI 3D Image Viewer

Figure 4.17 shows a snapshot of the 3D image viewer. Also shown is the view menu, that allows control of all the views. Using this menu, the Task View (view->functions) and Task Properties editor (view->properties) can be shown or hidden along with the modal input and output viewers. The view menu is shown as greyed out in the snapshot, as a modal 3D image viewer has been selected and rendered (view->showInput). This rudimentary viewer allows for cycling through each slice of a 3D image to examine image transformations across tasks. The current prototype only supports a 3D image viewer, but the plan is to augment the design such that viewers can be supplied that match input and output formats, which may be text or graphs as well as images. The view menu also has a showGL sub-menu. This adds functionality to the GUI to be able to navigate around an input image, in wire frame mode, with the resultant orientation parameters can then be added to appropriate tasks. This functionality formed the inception of the GUI design, originally implemented in QT3 and described in more detail in appendix A.

Aspects of this functionality are being ported to the current QT4 implementation, to be leveraged in tasks such as the ray tracing functionality, to provide a graphical way to navigate around 3D images and then ray trace from the chosen orientation.

As well as being able to construct task graphs, the GUI supports the full CRUD functionality, such that specification graphs can be created, opened (read), updated and deleted. This is essential functionality, that allows the persistence of complex task specification graphs. Figure 4.18 shows the File Menu, that can be used to create a new (unsaved) task specification, open and existing specification, save, save as, and close options. The open, save and save as sub-menus open platform specific file navigation dialogs for selecting and saving task graphs.

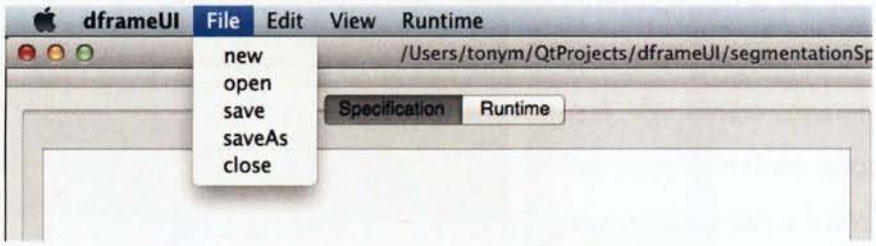


Figure 4.18: The GUI File Menu

Figure 4.19 shows the simple Edit menu, that integrates with the Task Graph View selection capability. Tasks are dragged and dropped onto the graph view, and then any two tasks can be selected, and the addEdge menu will then connect the two tasks with a dependency edge. The direction of this edge follows the order of selection, with the first selected task being the source, and the second selected task being the destination (dependent task). The deleteSelected is used to delete edges and tasks. If a task is deleted that has edges connected to it, the edges are also deleted. The changes become permanent when the File->save menu (there is no undo/redo functionality built into the current GUI prototype).

Figure 4.19: The GUI Edit Menu

As well as the Task Graph View's prime purpose to aid pipeline composition, another tab is also incorporated that is intended to support the viewing of a connected server DFrame's runtime behaviour. This will display timing and diagnostic information, on the running of a task specification, such as the timings of each task, the models used, processor distribution and task partitioning. This is still being designed and developed, and currently server DFrame instances dump this information to a file. The impetus for delivering such information to the GUI would again be to enhance productivity, and ease of use, to improve image analysis experience, and confidence in the system (although the system would in many respects be automatically optimising performance, feedback is still important to judge its success).

4.7 *Summary*

This chapter has presented the motivation and purpose of the DFrame, introduced its core concepts, given an overview of its fundamental design and gone on to consider its component building blocks in more detail. In evolving the design, significant effort has been made to separate out and express the various components of a parallel computation, such that by using the framework the effort in developing real world parallelised applications is much reduced. Nevertheless, the complexity of the DFrame design and implementation is still substantial, and in this respect is itself evidence that using such a framework is worthwhile. In particular, the effort in developing a generic parallel processing framework that can be reused, and that can itself be further developed in isolation to the applications that use it is an attractive alternative to 'going it alone' on each application. The cost of developing the framework being amortised over the many applications it will support, where each task graph can be considered an application. The

separation allows the DFrame to be independently improved for instance to incorporate more advanced performance monitoring and the plugin nature of the design encourages further extensions and adaptations.

The framework provides for the creation of a task graph that describes the required coarse grained functionality. At a more fine grained level, each node in the task graph is a task specification that encompasses the information to orchestrate the parallel running of an algorithm. This includes the model or models that are appropriate to running the algorithm, together with any algorithm specific parameters and contextual task graph information. Upon loading a task graph, the DFrame constructs a corresponding static workflow, and will automatically generate further dynamic sub-workflows if splitter tasks are implicitly or explicitly defined within the task graph, via a novel task graph splitting arrangement. This mechanism demonstrates the flexibility of the DFrame design in that it leverages the plugin model architecture and can thus be embellished, and other schemes retrospectively plugged in. The implemented splitter provides a core novelty in distributing groups of processes, and dynamically adjusting the number of processes in each group (currently by task size). Leveraging MPI's advanced features to isolate processor groups to collectively work on different tasks simplifies the communication amongst processes participating in the parallel processing of each task. A workflow sub-component is provided that manages the building and manipulation of workflows, and workflow messaging infrastructure is also provided to allow sub-workflows to be transferred to nodes across the distributed system.

A prime focus of the DFrame design is its emphasis on separation, extensibility, adaptability and reuse. This allows functionality to be added to the system in an incremental manner as requirements unfold, and the separation allows for maintainability as component parts can be worked on and improved in isolation. The functionality can be augmented with the ability to swap out models to trial other approaches, or to adapt and dynamically optimise model selection and model parameters based on contextual knowledge of the task graph (adapting to a task execution plan). Model decisions can be based on a tasks input size and/or other characteristics, such as whether a task's partitioning is static or dynamic, structured or unstructured, and whether each part has a regular or irregular load. Application code can interface to standard model interfaces, and this is

encouraged by the framework, but not mandated, as bespoke models can also themselves be plugged into the framework, exposing specific interfaces if needed. The normal expected usage is to endeavour to make application code unaware of any parallelisation (when this is possible and practical), and to provide adapter infrastructure that bridges into the DFrame interfaces. This is the route taken with the basic 3D imaging infrastructure already provided in the prototype.

The 3D imaging infrastructure includes 3D image IO to and from disk storage, and image averaging filters and a Sobel edge detection operator. A watershed segmentation operator is implemented, and also a more straightforward experimental thresholding and histogram segmentation functionality is implemented. As well, a parallelised ray tracing module is implemented that incorporates server side direct volume ray tracing of 3D images, that outputs 2D views. A camera design is built in, to enable orientation of the ray tracing. Of more fundamental importance, 3D image partitioning functionality is included, with ancillary partitioning strategies, partitioners and complementary recomposition implementations, that also support ghost cells. The evolving development of this functionality has guided the DFrame to a more generic design.

That the DFrame encourages the organisation of a parallel application into component tasks, and separates out the parallel aspects also allows for the capture of diagnostics information. To this end, a timing and timing context component is provisioned by the DFrame, and used by the DFrame itself to time each task run, and passed to models for use in timing model functionality. The timings are gathered onto the root process and saved to a separate file for inspection and analysis. Currently, the analysis is post run, but the goal is to leverage this information to dynamically update runtime parameters to optimise performance (possibly used to adjust properties of the task graph).

Chapter 5 DFrame Component Evaluations

5.1 Introduction

The motivation for the architectural design of the DFrame, and its core concepts have been outlined. As well, the prototype implementation has beed described. Before embarking on an integrated case study that draws out the full power of the DFrame, it is first necessary to do some limbering up evaluations to provide a degree of confidence in the approach. The intent of this chapter is to prove that the various components of the DFrame are working as expected, and to produce preliminary results at the task level, that can be examined and assessed to ascertain the viability of the DFrame design in meeting its core aims to improve performance, to provide flexibility and reuse in the parallel processing layer and the application layer, and as important to provide a runtime system that is adaptable, such that it is choosing suitable models to run and partitioning strategies based on initial parameters and also on dynamic system behaviour (performance feedback). As each task in an application is itself parallelised, there is considerable complexity at the individual task level, and so it also adds clarity to evaluate the DFrame performance at the (parallelised) task level first.

This chapter presents the results of applying the distributed framework to four discrete practical image processing algorithms (tasks): distributed averaging filters, a 3D Sobel local image operator, 3D image segmentation, and visualization using direct volume ray tracing. In the first evaluation, preliminary tests are conducted to confirm the correct operation of the DFrame using a master worker model to test simple 2D and 3D averaging operators. In the second evaluation, a master worker model is used to run a Sobel operator across a 3D image. This highlights the overhead of partitioning, distributing, gathering and recomposing data. When appropriate, this can be amortized by distributing the data, computing multiple operators and then gathering the results. The third evaluation looks at the mesh model approach, using a segmentation algorithm as the compute load. In the forth evaluation, whole images are loaded by each node and the master-worker model used to distribute camera information in a ray tracing test. The tests use custom benchmarking on non standard 3D test images from cell data research, to align with the intended prime use of the framework.

5.2 *Cluster Hardware Configuration*

A simple schematic of the HPC cluster architecture at Kingston University is shown in Figure 5.1. Users can connect to the HPC cluster via the cluster's gateway node, either from the Kingston University Network, or from the internet using ssh (the 'secure shell' protocol).

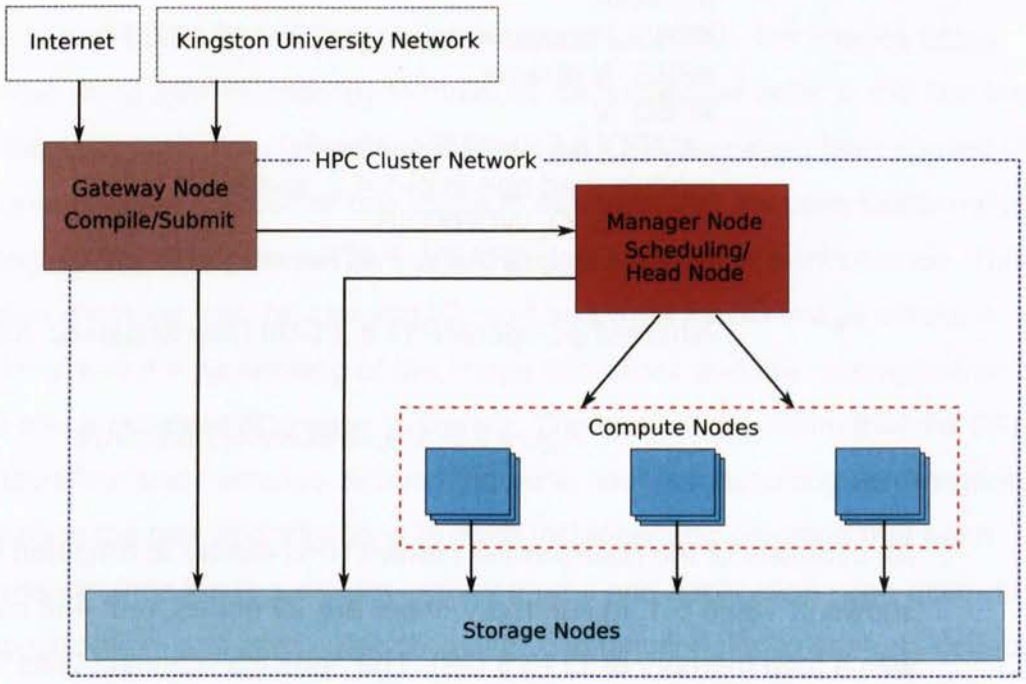


Figure 5.1: Schematic of the HPC Cluster architecture at Kingston University

Typically, programs are uploaded to the gateway node and compiled there. A module system allows users to also arrange for the loading of any required ancillary libraries. A TORQUE distributed resource manager based on the original PBS software allows users to submit jobs and monitors the available compute resources (the number of CPU's and memory). The resource manager integrates with a MOAB workload manager based on the open source MAUI cluster scheduler, that determines where and when jobs can run (Adaptive Computing 2015). From the gateway node, compiled programs are submitted to MOAB/PBS for scheduling and execution. This is normally accomplished by submitting a PBS script, that contains information on resource requirements such as the required number of processors and the expected time duration alongside information on the

program to run and any required dependencies. An example PBS script is shown in Figure 5.2 below. This requests 4 nodes with 1 processor per node (i.e. requests 4 processors), stipulates an expected runtime of 10 minutes, names the job, and loads the required openmpi module. The last line requests mpiexec to be run, passing the program as an argument.

```
#!/bin/sh
#PBS -l nodes=4:ppn=1,walltime=0:10:00
#PBS -N dframe
#PBS -V
. $MODULESHOME/init/bash
module load openmpi-1.4.2_shared
cd $PBS_O_WORKDIR
export LD_LIBRARY_PATH= <required paths>

/shares/hpc/openMPI/1.4.2/BUILD/bin/mpiexec ./dframe
```

Figure 5.2: Simple Example PBS Script

An overview of the nodes in the current HPC cluster at Kingston University is shown in Table 5.1. In summary, there are 29 nodes, with 488 cores in total (and with a total memory of 2113.5 GB). The node interconnect uses Gigabit Ethernet. The case studies for this project used up to 64 processors.

Num. Nodes	Node Type	Model	sockets/ machine	cpu's/ socket	Memory MB (/machine)
16	ProLiant BL465c G5	Quad-Core AMD Opteron(tm) Processor 2384	2	4	32768
2	ProLiant BL465c G6	Six-Core AMD Opteron(tm) Processor 2435	2	6	65536
2	ProLiant BL465c G7	AMD Opteron(tm) Processor 6172	2	12	65536
9	ProLiant BL465c Gen8	AMD Opteron(tm) Processor 6380	2	16	147456

Table 5.1: Kingston University cluster node core and memory details

5.3 Averaging Image Operators applied to 3D Bio-Cell Images

This first evaluation presents preliminary results of running two simple parallelised averaging operators on the DFrame. The tests provide custom benchmarking on a non standard 3D test image (*696x520x21 pixels; 16-bit; 14MB*) from cell data research, to align with the intended prime use of the framework. The 3D bio-cell images used in tests in this chapter and the next are made available courtesy of previous research efforts that Kingston University has conducted in partnership with Cancer Research UK (Hagglund, Hoppe et al. 2009), the images being captured using light microscopy techniques on fluoresced cells. In the first test, one image is partitioned into image slices, and a 2D averaging filter applied to each image slice. The aim of this test is to establish that the core functionality is working as expected, as well as starting to gain insight into performance. This includes checking that the imaging IO read and write for 3D image stacks is functional and the partitioning of the image into slices and the recomposition of the slices into a resultant 3D image is correct. The tests also confirm that the DFrame core workflow and message dispatching behaviour is functioning as designed, distributing the task specifications to each instance, and ensuring that each DFrame instance loads a master-worker model and application code according to the specification, and completes correctly. The functionality to capture, collect and record timings is also exercised and checked. In the second test, the aims are similar, but with a further objective to process more images, as the processor count increases, to simulate increasing the workload, commensurate with the increase in the number of available processors.

5.3.1 2D Averaging Filter Applied to a 3D Bio-Cell Image

This evaluation applies a 2D averaging filter to slices of one 3D image, with distribution of the image slices. The DFrame is configured in 'batch' mode, to read a specification workflow from a file. The workflow contains only one task in the task specification used for this test. The designated root DFrame reads the specification graph, and distributes the one task to each participating DFrame instance. Each instance reads the task specification, and loads a master-worker model. By default, the model running on the root DFrame instance takes on the master role and other instance models assume a worker role. The master loads the 3D test image from disk, and interfaces with the application module to retrieve

and distribute one partitioned image slice to each worker process via the DFrame dispatcher. Each worker then processes the received image slice and loads and runs application code according to the task specification. In the master-worker model used for this test, if only one process is used, the master does all the work. Otherwise, when more than one process is defined, the master only distributes the work and collects the results, and does not process an image slice itself. In this test a 9x9 kernel filter is applied to each image slice, resulting in an updated image slice that the worker then returns to the master. The master interfaces with application module composers to recompose the slices into a resultant 3D image and stores to disk. The test image contained 21 image slices, and so the optimal partition configurations in Table 5.2 were tested.

Number of Processor Cores	Master	Number of Workers	Tasks per Worker
4	1	3	7
8	1	7	3
22	1	21	1

Table 5.2: Partitioning information for the Averaging filter tests

5.3.2 3D Averaging Filter Applied to Multiple 3D Bio-Cell Images

This evaluation applies a 3D averaging filter to many 3D images, with distribution of image paths using the master-worker model, to test the degree of speedup attainable in a high content throughput scenario. Again, the DFrame is configured in batch mode, with the specification workflow containing only one task specification, set to load a master-worker model. However, the specified application code is different, with the master module code implemented to distribute file paths of a collection of 3D images, one to each worker process. As the number of workers is increased, the number of images processed is also increased so that each worker executes a read-process-write workflow on one 3D image (c.f. tests that process the same image on an increasing number of processors). Each worker processes all voxels of a distinct 3D image using a

3x3x3 averaging filter, with all images being the same size *(test image: 696x520x21 pixels; 16-bit; 14MB)*. Ideally, as processors are added, the total time to execute should remain constant, as each additional processor will process it's own distinct image. However, in practise some impact on performance is expected, due to IO contention as the load on the storage system rises (system dependent), and also increased interprocessor communication as the processor count is increased. Since each worker processes only one image in this test, the total time recorded to execute the program will be 'worst case', with the total time reflecting the 'slowest' worker's read-process-write workflow. It is also worst case in the sense that each worker will attempt to read it's image at about the same time, and because processing is the same for all workers, the writes will also be at around the same time, with some increased variability. Further dframe timing metrics were inserted to capture some of this variability of processing across the processors, recording startup time and each worker's time to read, process and write it's image.

Note that when testing one process, one task is arranged to be run locally by the master, which thus acts as it's own worker. To calculate speed up, the serial time it would take to process n images, is divided by the tested parallel time to process n images (on n+1 processors).

5.3.3 Master Worker Model Performance Results

a) Applying a 2D averaging filter to slices of one 3D bio-cell image, with distribution of the image slices

One process, the master, reads a 3D input image and writes the output 3D image (as would be the case on a single machine). Additional overhead is incurred in distributing the specification, and the subsequent distribution of the image slices to workers and collection of the resultant image slices. However, the resulting performance improvements are very encouraging. In Figure 5.3 the execution time decreases from around 30 seconds for one processor core, down to 2 seconds for 22 processor cores. Figure 5.4 shows the corresponding speed up is above 15, for 22 processor cores (including the master).

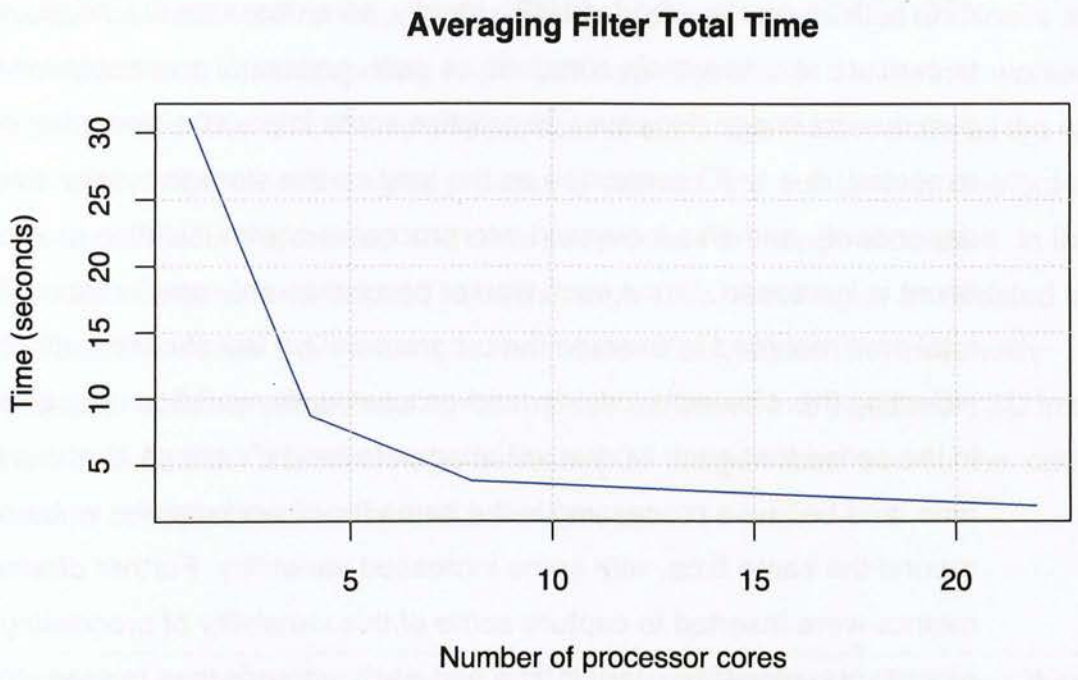


Figure 5.3: Time with each worker processing one image slice of a 3D image

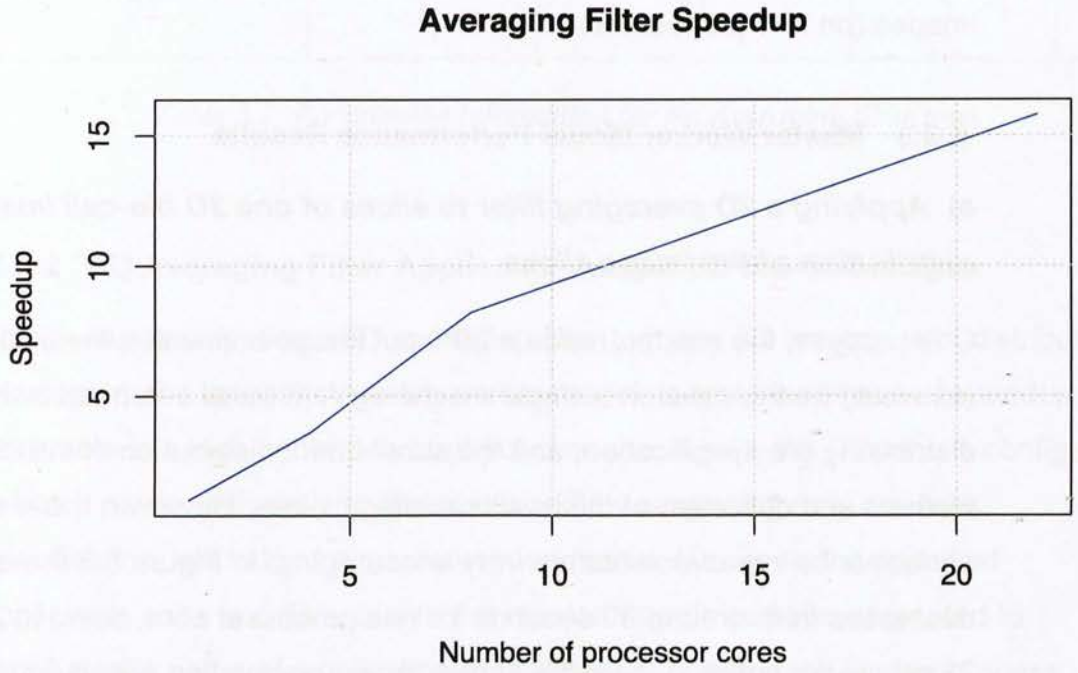


Figure 5.4: Speedup with each worker processing one image slice of a 3D image

Subsequently a 'scatter-gather' master worker model variant was applied to this test case, in which all processors engage in MPI scatter and gather collective operations to distribute image slices and return results. In this model variant the master also does work processing an image slice. Broadly similar results were observed, using one less processor in each test as the master also acted as a worker (so results are not included). The 'scatter-gather' variant of the model works well if the computational effort on each processor is identical, as each process receives work and returns results as a collective operation across all processes, and thus take advantage of efficient collective scatter-gather implementations.

b) Tests on applying a 3D averaging filter to many 3D bio-cell images, with distribution of image paths:

The performance graphs in Figure 5.5 and Figure 5.6 show the timing and speedup mapped to the n+1 processor cores used (n = the number of images).

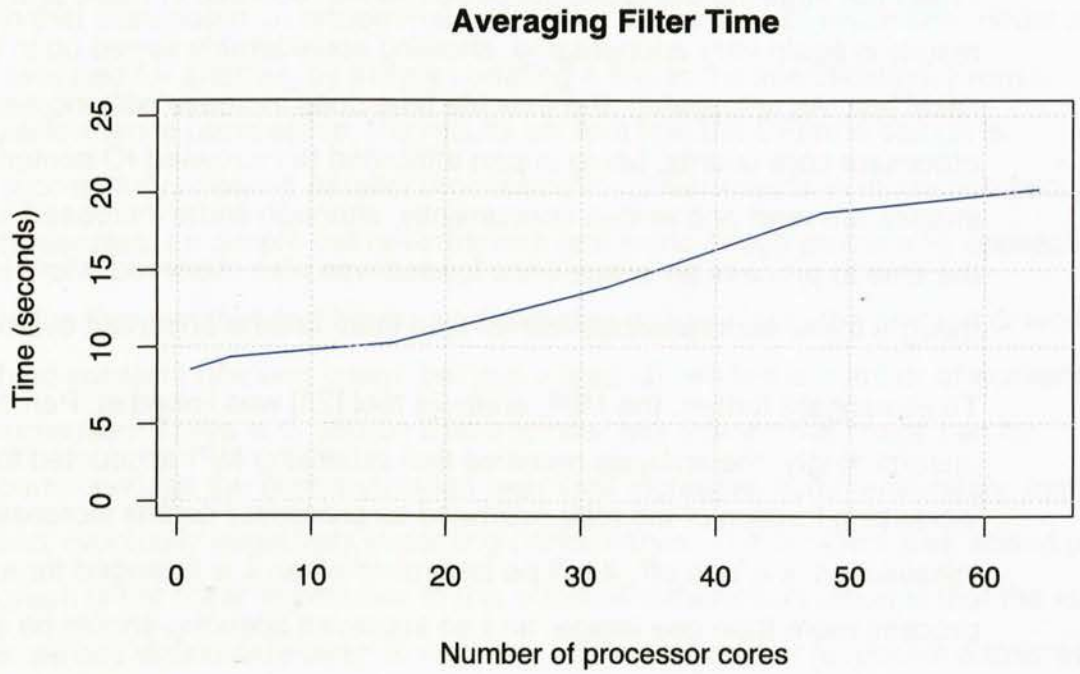


Figure 5.5: Time with each worker processing one 3D image.

(i.e. load increasing as processor core count increases)

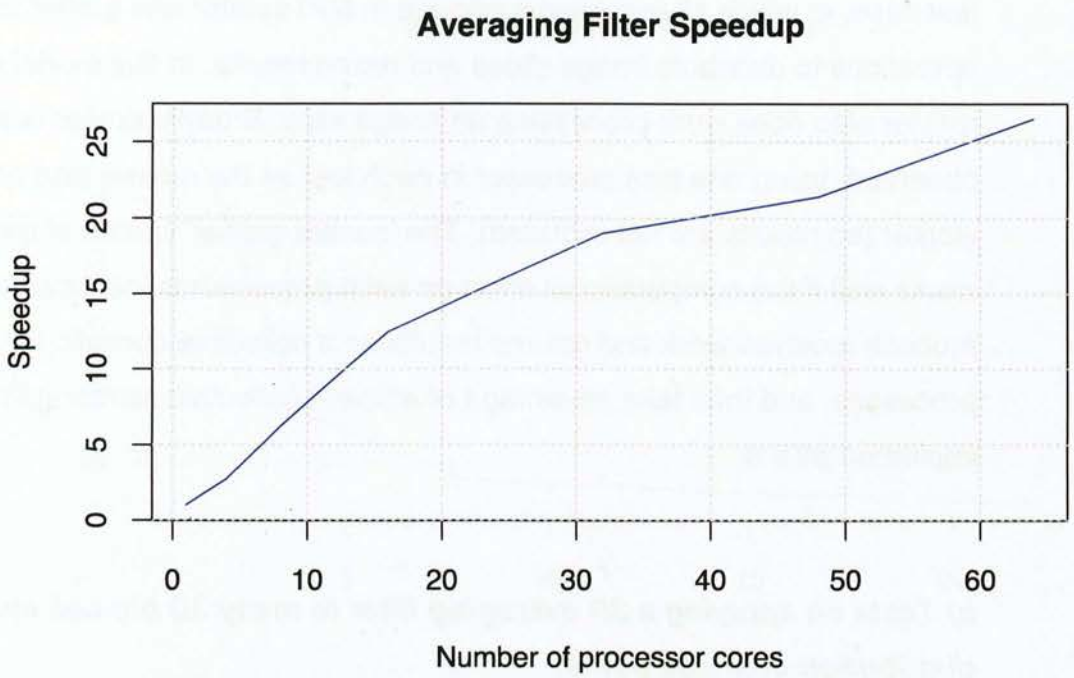


Figure 5.6: Speedup with each worker processing one 3D image

Whilst not attaining the ideal, the performance recorded in these preliminary results is again very encouraging, showing considerable speed up in this 'worst case' test. As anticipated, the execute time does increase with increased processor core counts, being in part attributed to increased IO contention as more images are read and written concurrently, although some increased variability in the time to process an image once loaded was also observed. More interestingly though, these factors accounted for less than half the observed overhead.

To investigate further, the MPE analysis tool [23] was linked in. Perhaps unsurprisingly, the analysis revealed that initialising MPI accounted for an increasing fraction of the total overhead as processor counts increased. Since this initialisation is a 'one off', it will be amortised when it is arranged for each worker to process more than one image, and an improved speedup should be expected.

As interesting, the analysis also revealed that significant time was spent at barriers inserted into the program. Some barriers were inserted to allow reasoning of program flow and are otherwise redundant and will be removed to improve speedup. However, it is still an area of continued investigation as to why processes can remain at a barrier for a significant period after all processes have

arrived at the barrier (using openmpi 1.4.2). Else wise, the internode master-worker communication is comparatively very small and did not significantly impact performance.

5.3.4 Discussions

This preliminary evaluation had two aims. Firstly, to establish that the DFrame core behaviour was working as designed. This encompasses the receipt of a task specification from a client, the operation of the workflow component, the resolution of models and application module code, and the cooperative execution of the models loaded on each DFrame instance, to progress and complete a parallel computation. The second aim was to provide early feedback that performance gains were attainable using the DFrame to apply image operators to a partitioned image and to multiple whole images. These being legitimate goals for the project, and a precursor to the subsequent more involved evaluations in this chapter.

Overall, the results are very positive on both accounts, confirming the core functionality in the DFrame design. Also, as mentioned in the results, in the first 'image slices' test, a 'scatter-gather' master worker model variant was also plugged in that illuminated an orthogonal success in the ease with which one model can be swapped for another, by simply updating a line in the specification. From a performance perspective, the results confirm that the DFrame design is successfully applying parallel processing to deliver results in much reduced timescales, on simple but nevertheless real world image processing applications.

In the first test that partitions and distributes image slices, the amount of work is held constant (the one image being processed), while the number of workers is increased. There is of course a limit to how fine grained the image can be partitioned, as the communication overhead increases commensurately, inhibiting and eventually negatively impacting performance. The 'image slices' speed up graph is not linear in part due to this effect. Another observation is that the master is serially distributing tasks to workers, so the last worker to receive a task will be some time after the first worker, effectively extending the apparent time to execute a task. Also, each worker may take a slightly different time to execute a task, and the worker that takes the longest time to execute may be reflected in the results (if this is not by chance obscured by the order of distribution).

The second test increases the workload as more processors become available. It

was anticipated that the speed up would be more linear in this case, as each processor continues to do the same amount of work. Although the results are encouraging, this expectation was not fully realised, with the time increasing as more processors were added with an equal amount of extra work. One factor was the increased startup cost of MPI and this is not an undue worry as the DFrame is designed to start up once and then to continue substantial batch or interactive processing in one session. Another observation was that the DFrame processing was stalling longer than expected in MPI barrier calls. This is an ongoing investigation, but it is admitted that some of the barriers are inserted for tests and early reasoning of the program flow, and will be removed. In this test, each worker is reading its own 3D image from disk, and there is the possibility that as the number of workers reading from disk increases, contention will increase impacting performance.

The tests provided useful feedback on the operation of the DFrame design, the master-worker models and the integration of application code (in strong scaling and weak scaling scenarios). The mechanism for packing, sending, unpacking and receiving work and results did not itself manifest in a huge extra cost, an aspect of initial concern. Even so, it is expected that now a rudimentary implementation is in place, effort can also concentrate on tuning areas that are identified as performance bottlenecks. However, further investigations in this regard is left to the more detailed evaluations that now follow.

5.4 Sobel 3D Image Operator

5.4.1 Background

Many image operators are local, such that computations on each pixel or voxel only require data from its neighbours to compute an updated value for that location. Such operators are ideal candidates for parallelisation. In this evaluation, a master-worker model configuration is again used to apply a Sobel operator to a larger 3D image. The Sobel operator is an edge detector that determines the approximate gradient of an image. An image is convolved with kernel filters that approximate the local derivatives of image intensities in each dimension, to produce an output image that accentuates regions of high spacial frequency (i.e. edges). A good description on edge detectors can be found in (Klette 2014)

alongside a general discussion on local image operators, and points out that the Sobel filter was first published in (Sobel 1970). Commonly used for 2D image edge detection, the Sobel operator can equally well be applied to 3D images. The Sobel operator was chosen for this evaluation, as the output is useful for subsequent processing such as segmentation which is considered in the third evaluation. It should be recognised, that the contents and size of the kernel is adjustable so that many local operators can use the same approach (and plug into the same algorithms).

5.4.2 Sobel Operator Parameters

The kernel filters of the 3D Sobel operator used in this evaluation are tabulated in Figure 5.7 below, these being extrapolated from (Wikipedia 2015) .

'x' dimension

1	0	-1
2	0	-2
1	0	-1

2	0	-2
4	0	-4
2	0	-2

1	0	-1
2	0	-2
1	0	-1

'y' dimension

-1	-2	-1
0	0	0
1	2	1

-2	-4	-2
0	0	0
2	4	2

-1	-2	-1
0	0	0
1	2	1

'z' dimension

1	2	1
2	4	2
1	2	1

0	0	0
0	0	0
0	0	0

-1	-2	-1
-2	-4	-2
-1	-2	-1

Figure 5.7: Kernel filters for a 3D Sobel operator

The computation convolves the kernel filters with each voxel of the 3D input image, to produce a 3D gradient output image. The operator can be viewed as a smaller 3D image whose centre is positioned at each voxel of the input image and the masked voxels of the original image are then convolved with the kernel voxels at the same point, to produce the output at the centre location.

5.4.3 Image Partitioning Strategy

To arrange for parallel execution, the input 3D image is partitioned into sub-images that are distributed to each participating process. As the convolution of the Sobel operator at a voxel requires access to that voxels neighbours, each partitioned sub-image must include boundary or 'ghost cell' information for boundary computations of the sub-image. The 'shape' of the sub-image will affect the amount of extra boundary information that needs to be communicated alongside each core sub-image, and this overhead should be taken into account. Although a cube has an optimally minimum surface area, a partitioning strategy may require some latitude, taking other information into account, such as the number of available processes, the shape of the 3D image and the task size.

In this evaluation, a coarse grained partitioning strategy is chosen such that the input 3D image is partitioned into a number of sub-images equal to the number of available processors. The assumption is that the partitioning will be minimised, and the communication optimised with the fewest number of tasks being communicated (one to each processor). Another side effect of this choice is that the analysis is also simplified, and thus clearer to review. The master process does not participate in the processing of sub-images, so the partitioning must align with the number of workers (i.e. one less than the number of processors). That this test chooses a partitioning that exactly aligns with the number of workers has ramifications for the number of processors to use. For instance, if 12 processors were made available, then 11 would be available to participate as workers, and this (prime) number of workers will only allow partitioning across one dimension, into 11 strip images, which is likely to result in less optimally shaped sub-images when ghost cells are required. Although the partitioning strategy is implemented to look for a better partitioning even if that means using fewer processors (e.g. 10 processors), the test is conducted with numbers of processors that can be

partitioned according to the number of workers. The choice of the number of processors to use is complicated further when the number required exceeds that available on a single node (32 on the cluster used), as the required number of processors is specified as the number of nodes and the number of processors per node (two parameters rather than one). Thus, only certain configurations are available when the number of processors required is greater than the maximum number of processor per node. In summary, the processor count and configuration was adjusted in order to map more reasonable sub-image shapes to each processor and Table 5.3 tabulates the number of processors used and the corresponding image block sizes for the test image (688x512x144, 16 bit, 97MB).

Number of nodes	Processor cores per node	Total processor cores	Number of workers	Sub-image Block size	Number of Blocks
1	1	1	1	688:512:144	1
1	4	4	3	230:512:144	3
1	9	9	8	172:256:144	8
1	13	13	12	230:128:144	12
1	16	16	15	138:171:144	15
1	25	25	24	172:171:72	24
1	31	31	30	138:171:72	30
3	17	51	50	138:103:72	50
3	19	57	56	99:128:72	56

Table 5.3: Partitioning information for the Sobel operator tests.

3D image size: 688x512x144, 16 bit, 97MB

The partitioning strategy is defined by an entry in the task specification, or is automatically determined if not explicitly specified. By passing a 3D image along

with the number of available processors as parameters to a partitioning strategy implementation, the sub-image block sizes are calculated. Specifying ghost cell information to a partitioner along with this block size information will configure the partitioner to return appropriately sized sub-images for distribution to the workers. The partitioning strategy implementation can be swapped out to provide different behaviour, and this will be useful in further investigations such as the effect of more fine grained tasks (smaller sub-images). When only one processor is used, of course there is no partitioning, and the master runs the computation locally.

5.4.4 3D Cell Image Results

The following figures show the same slice through the test 3D cell image before and after the Sobel operator is applied.

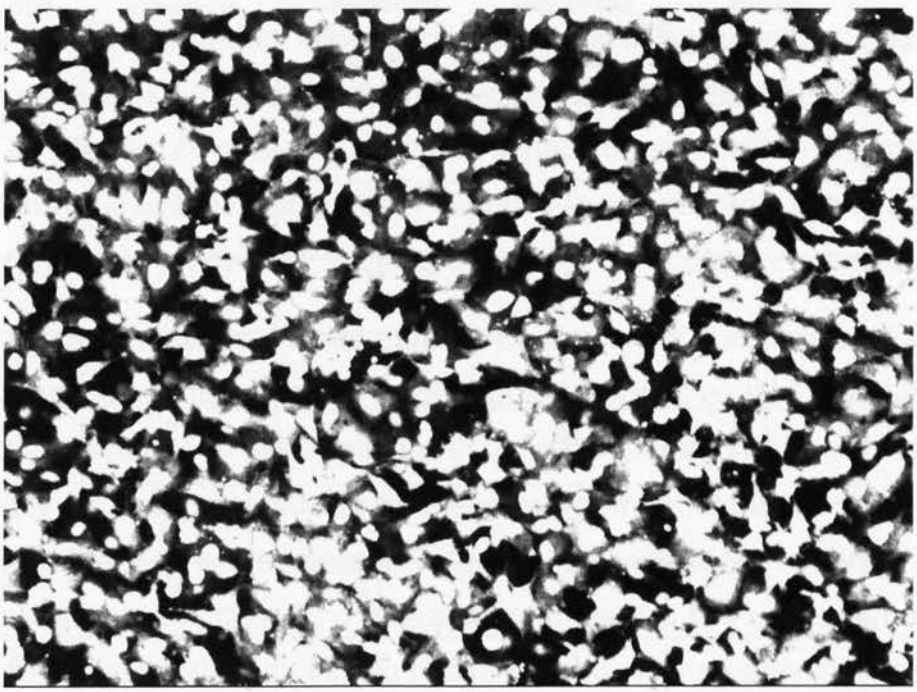


Figure 5.8: Input image x-y slice of labelled sarcoma cells.

3D image size: 688x512x144, 16 bit, 97MB

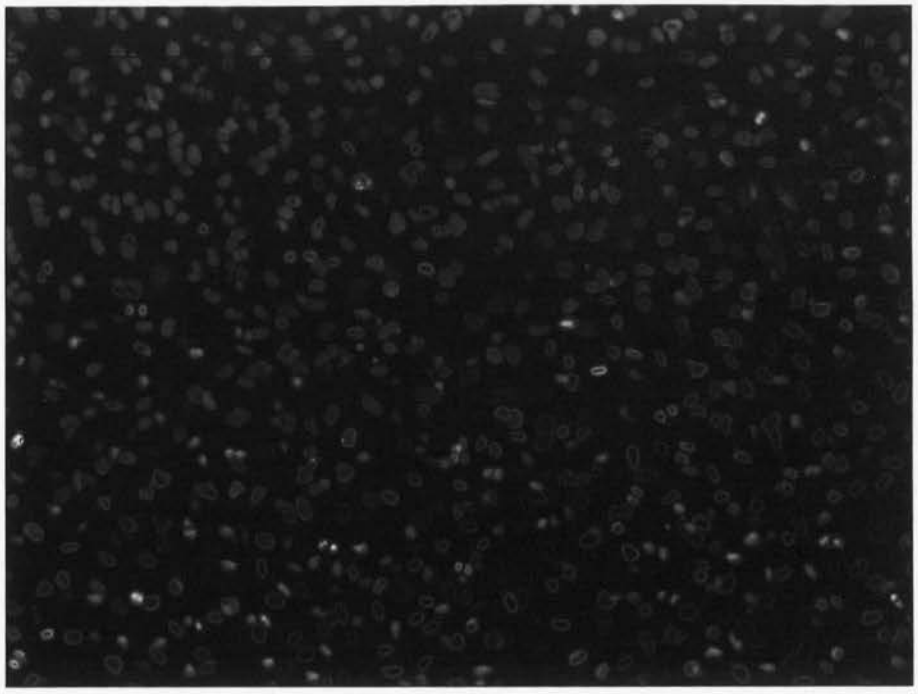


Figure 5.9: Sobel operator output image x-y slice detecting nuclei of labelled sarcoma cells

Figure 5.8 shows an x-y slice of the input image of labelled sarcoma cells and in Figure 5.9 an x-y slice of the output image shows the successful application of the Sobel gradient operator as indicated by the now clearly discernible cell nuclei edges.

The graphs in Figure 5.10, Figure 5.11 and Figure 5.12 plot the execution time, speed up and efficiency respectively, when running the Sobel operator on the test image using a varying number of processor cores. As the graphs show, although useful speedup is observed, the parallel execution falls some way short of the ideal. Indeed, above 16 processor cores, the speedup begins to plateau and at higher processor core counts starts to drops slightly, with a corresponding decline in the efficiency. Note that even in the more ideal case, the efficiency will not attain 100%, as the master is not participating in the processing of the image, so for 4 processor cores only 3 are participating and the efficiency is approaching 75% rather than 100%. This effect should of course diminish as the number of processor cores used increases.

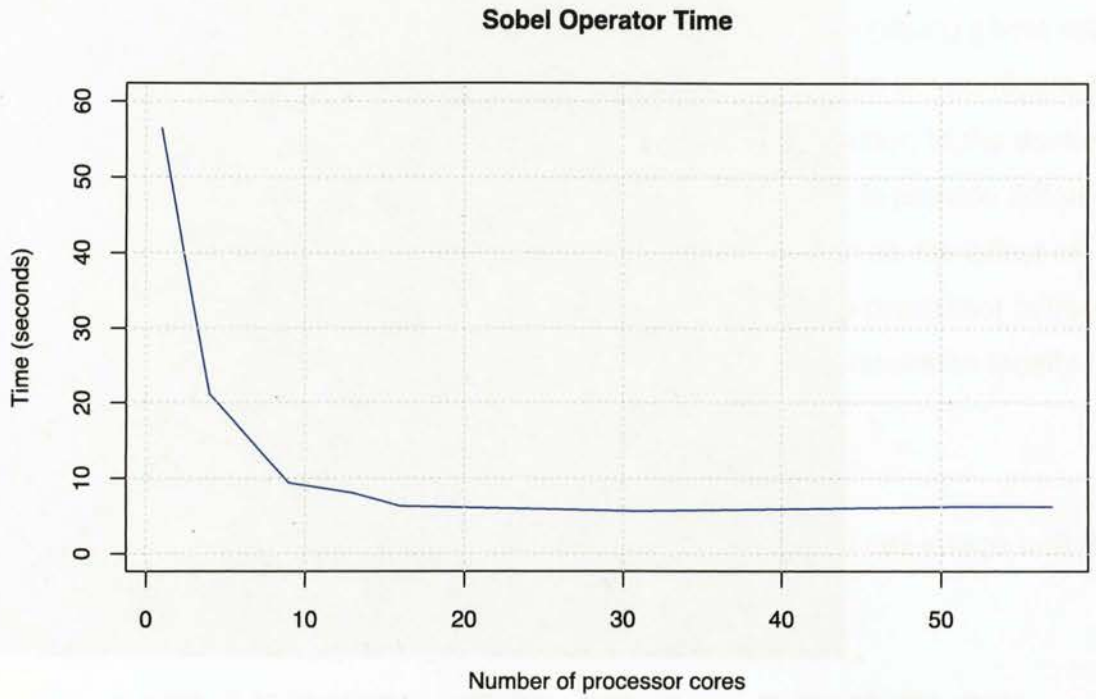


Figure 5.10: Sobel operator processing time

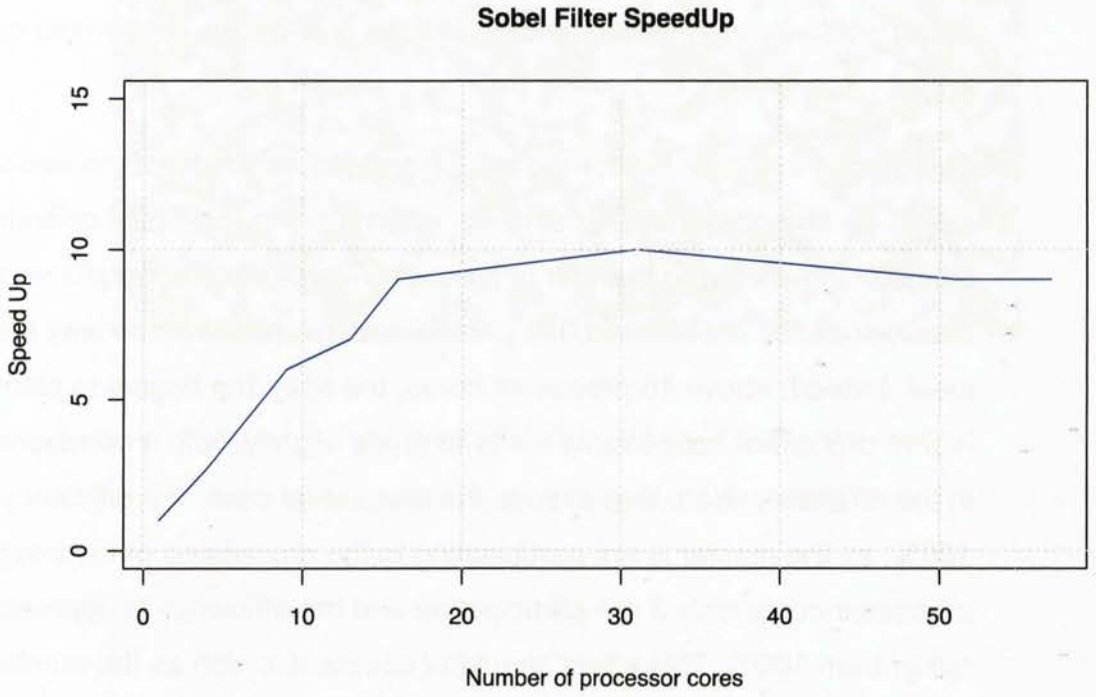


Figure 5.11: Sobel operator speedup when applied to a 3D image

3D image size: 688x512x144, 16 bit, 97MB

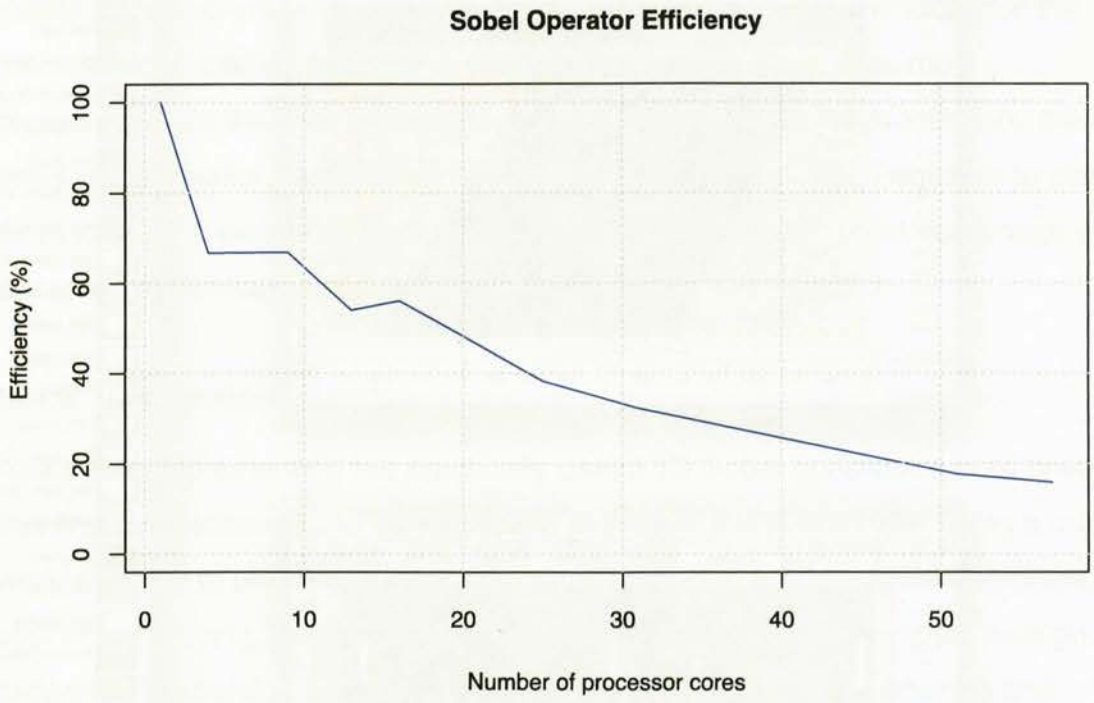


Figure 5.12: Sobel operator efficiency

The tail off in performance warranted further investigation, and so the DFrame was recompiled with the MPE profiling tool linked in to collect more information, and to visualize the distributed processing behaviour in more detail using Jumpshot (Zaki, Lusk et al. 1999). In Figure 5.13, it became immediately apparent that the distribution and collection of the sub-images is having a significant impact. In particular, serial 'laddering' of the distribution and collection is observed (top image).

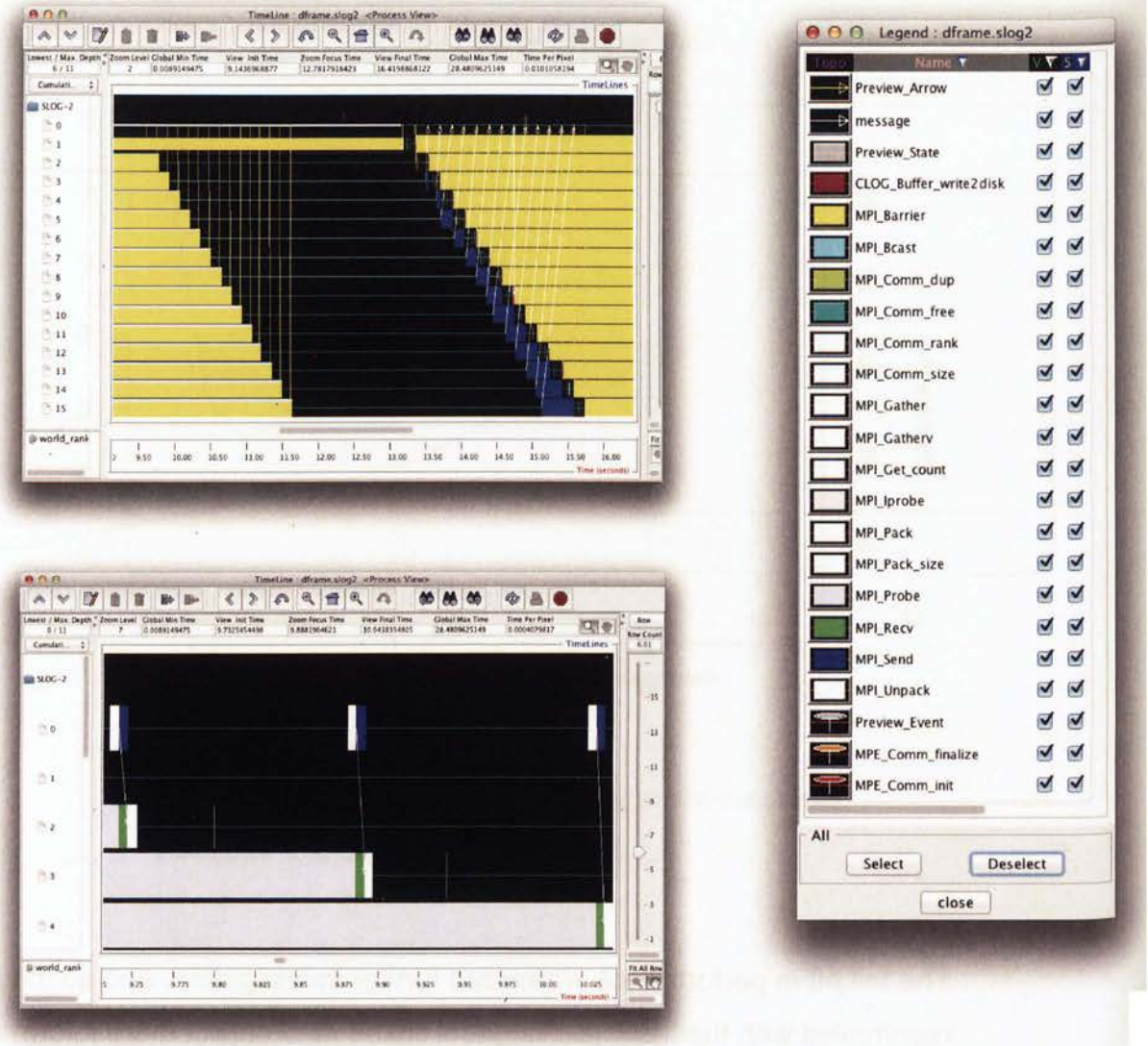


Figure 5.13: Sobel operator timings using 16 processor cores (MPE/Jumpshot)

This staircase effect is expected when the master process serially distributes data to workers, and in cases where it becomes a significant overhead non-blocking communication is normally introduced to try to mask it, such that communication is undertaken while the master processes more work (the DFrame now supports non blocking sends, so this is being incorporated into the model). However, the lower 'zoomed in' image in Figure 5.13 adds further insight to the performance degradation. Indeed it appears that the serial sending of sub-images is not the main culprit. Using 16 processors, the time to pack, send, receive and unpack a partitioned sub-image is only around 15ms whereas there is an interval of over 150ms between each send from the master to the worker! This is the time that the master spends creating each of the 16 sub-images prior to packing and sending

them (i.e. the time in between posting sends). As the processor count increases, further inspection via the MPE tool confirms that although it takes less time to create each sub-image there are correspondingly more of them such that the recorded total partitioning time is similar and the effect becomes more pronounced, as the time to execute the work reduces. For 31 processors, the sub-images are smaller, creating each accounts for 80-90ms which equates to around as much as 2.8 seconds! Recomposition of each returned output sub-image is observed as similarly expensive.

5.4.5 Discussion

Apart from implementing the practically useful 3D Sobel operator, a core driver for this first evaluation was to illuminate the overhead associated with using a master worker model to parallelise local operators applied to a 3D image. In particular, the cost of partitioning, sending, receiving and reconstructing sub-images with ghost cells prior to applying the local operator, and the subsequent gathering and recomposing of the result. It was expected that the packing, sending and receiving of the sub-images would be the dominant impact but these timings are actually quite encouraging. The much more significant impact appears to be the cost of using higher level structures to access and partition the 3D image into sub-images and the initialization of the sub-images with the appropriate data prior to sending, and the reconstruction of the image from the gathered sub-images upon completion of the distributed computation. Although there is evidence that non-blocking sends could be usefully employed in the model layer, the results reveal that priority should be given to investigating the application domain code that partitions and recomposes the 3D image, as the observed costs far exceed those anticipated. It is suspected that this is due to the creation and use of higher level pointer structures designed to allow access to and setting of image voxels via dimension index (e.g. x, y, z), rather than the extra boundary data added to the sub-images. This suggests that more innovative 3D image partitioning is required. A next step will be to arrange for a virtual sub-image design that provides a window into the whole image rather than copying out a section to a separate sub-image. Sub-image descriptors would then define a striding in the whole image that can be used to pack the sub-image data directly into an MPI buffer, thus removing the intermediate copy and population of a separate sub-image prior to such packing. A similar arrangement could also leverage MPI's derived data types

feature. This more innovative virtual sub-image design is expected to be much more performant. The need for a separate sub-image would still be required on the worker side, but the partitioning and recomposing of the 3D image would avoid the extra burden of creating and populating separate sub-images for this purpose, and this has been identified as a bottleneck on the master process.

Another point to mention is that the set up for this evaluation has the 3D image partitioned into a number of sub-images that matches the number of worker processes, so as to reduce the number of tasks to one per processor. This means that an optimum sub-image shape cannot usually be obtained, and although this is likely to be of less impact, there is more research to be done to establish if more fine grained but optimally shaped sub-images would be better (or no worse given the required increased communication). The partitioning strategy used attempts to optimise the block size as much as possible, and in so doing can choose to use less than the available number of processors. However, for this evaluation only processor counts that result in all processes being used are considered, which is somewhat artificial but makes the tests across different processor counts more consistent and comparable. Even so, with this imposed constraint the partitioning strategy cannot achieve an optimum block size and must settle for a good compromise. This is relevant here because each sub-image includes ghost cells. Even though there is no data exchange in this evaluation, boundary information from neighbouring sub-images is still required.

Although the results are immediately useful in identifying specific performance black spots where further optimisations can be sort, they also encourage consideration of other approaches. Once all practical optimizations have been introduced, there will still be some minimum overhead associated with partitioning, distributing, gathering and recomposing the 3D image. The computation associated with the 3D Sobel operator is non-trivial, so even with the one operator, increased performance should be achievable once further application code optimisations are introduced (i.e. parallelisation is still worthwhile). One obvious observation is that the impact of this overhead will depend on the size of the computation. Specifically, if the computation time is significantly greater than the parallelisation overhead, the speedup will be commensurately better. One way to achieve this in the DFrame is to arrange for multiple operations to be performed on the deployed sub-images, prior to the gathering and recomposing stage, to

amortize the overhead across many image processing tasks. This is a practical solution for image processing applications which often apply a pipeline of multiple operators to an image. However, the master worker is not as well suited to this arrangement. Although a master worker model could introduce data location information that helped in this respect, this would add complexity to an otherwise simple and well understood model (the DFrame does have a scatter-gather variant of the master-worker that could be used to investigate this further). If sub-images are to remain deployed across many image operators, then there must be a way for them to update their boundary data after each operator. This requires the exchange of data amongst neighbours and is more suited to a mesh model that arranges the sub-images in a topological manner such that a sub-image is (virtually at least) adjacent to its neighbour sub-images. The next evaluation on image segmentation employs a mesh model to distribute sub-images across the available processors, together with infrastructure code that arranges for the exchange of data amongst neighbours. The performance of the exchange of data being a core interest.

When appropriate, another way to amortize the overhead of distributing and gathering data is to arrange for each node to load the whole image, and then pass only operation parameters (instructions) to each node rather than image data. In this case a master worker model is still very effective. This is the core technique used in the final evaluation, as it is well suited to a ray tracing application which requires repeated camera updates only, and produces a much smaller 2D image, which can be effectively gathered without too much impact on performance.

Lastly, instead of distributing sub-images, the MPI-2 parallel IO functionality could be used such that each worker reads its sub-image from the whole image on disk (Gropp, Lusk et al. 1999b). Although disk access is much slower, it would be worth investigating the relative merit of this approach. As a 3D image is stored on disk as a single array of bytes, some work would be required to devise a strategy to read non-contiguous sections of the image to assemble each sub-image, and this would be non-trivial and of course add to the access time. It would then be possible to compare the direct sub-image reads from disk against one process reading the whole image, and partitioning and distributing to all worker processes. In the ideal case, the expectation is that direct sub-image reads would become relatively more attractive as processor counts increased, but in practice this would be heavily

influenced and constrained by the storage architecture.

5.5 3D Image Segmentation

5.5.1 Background

After preprocessing an image, a common next (automation) stage is to extract relevant features that are then used in subsequent analysis and classification. For example, in cell biology core features would include the size and shape of a cell, and in order to calculate such features, the image must be segmented to identify and isolate each cell. The aim of segmentation is to partition an image into disjoint sets of pixels or voxels such that each set has members whose properties adhere to some similarity or homogeneity function (such as intensity, colour, proximity etc.) and that differ from neighbouring sets. Segmentation approaches can be broadly categorised into edge based methods that detect contours or discontinuities, and region based methods that detect similarities including thresholding, and various region growing, clustering, splitting and merging operations (Sonka, Hlavac et al. 2008). These segmentation methods can also be usefully categorized as local or global (Morris 2004) according to whether processing is amongst neighbours or involves the entire image, an aspect of particular interest to parallel processing. The accuracy of the segmentation stage will impact all subsequent analysis.

5.5.2 3D Image Watershed Segmentation

This evaluation focuses on the watershed segmentation, a popular region growing variant expected to provide good results for the class of problem under examination, namely the segmentation of cells in a 3D image. Watershed segmentation has its origins in the fields of mathematical morphology and topography (Serra 1982), where a 2D grey scale image is conceptualized as a topological relief. Rainfall or immersion techniques are applied to resolve distinct catchment basins (Beucher, Meyer 1993). In the immersion approach, the idea is to apply a flooding process to an image starting at image intensity local minima, and then to fill up the separate catchment basins defined by these minima. Where catchment basins meet, watersheds are built that delineate each catchment basin. In the rainfall approach it is the path that a drop of water would take when incident

at a particular location, to reach a minimum that determines which catchment basin that location belongs to. Although the topological analogy may not be as appropriate for the 3D images, the techniques are just as valid.

The parallelisation of the watershed transform is non trivial, due to the global nature of the process. Indeed, fast serial implementations using a global ordered queue have proved difficult to parallelise, as the partitioning of the queue across processes induces dependencies that limit parallel execution. Parallelisation is also particularly problematic for images that contain plateaus of constant intensity within an image. Without plateaus, topographical distance can be considered, but when plateaus are present in an image, processing must include the determination of geodesic influence zones (Bieniek, Burkhardt et al. 1997). A common approach is to calculate the geodesic influence zones using a breadth first search, but when plateaus straddle partitions, ensuring that wave fronts generated by a breadth first search are synchronous inhibits any parallelism (but is necessary to cater for process indeterminism). In (N. Moga, Cramariuc et al. 1998), a good review of various watershed parallelisation endeavours and issues is presented, along with proposed rainfall and immersion algorithms that exploit local properties that increase data locality to improve parallel execution. The key aspect to successful parallelisation of these algorithms is to preprocess the image to remove the plateaus (except the minima), this being accomplished by computing a 'lower complete' image.

This evaluation concentrates on the immersion technique to establish image watersheds. Vincent and Soille introducing a notably fast implementation of the immersion variant (Vincent, Soille 1991), and the test algorithms incorporate this approach together with preprocessing and post processing to accommodate parallelisation. A gradient image is used as an input to the segmentation stage, generated by applying a Sobel operator to a 3D input image (see previous evaluation). Although the watershed transform is very popular, it does have one significant disadvantage in that it can over-segment images (particularly noisy images). Various approaches have been proposed to eliminate this, one successful method being to use markers that identify the catchment basins instead of automatic identification based on local image minima, but this shifts the problem to identifying the markers (Moga, Gabbouj 1998). Other approaches include generating mosaic images or hierarchical representations from an over-segmented

image and again applying a watershed, and introducing more advanced techniques such as the 'p' algorithm (Beucher, Marcotegui 2009). Here the focus is in parallelisation, so this evaluation does not fully investigate all these approaches, save for some experimentation with 'filling in' segments smaller than some parameterised size, to examine whether this would adjust the 'resolution' of the segments (segment size). The anticipation is that other strategies could eventually be plugged into the algorithms to aid segmentation.

5.5.3 Image Partitioning Strategy and the Mesh Model

A parallel regular Mesh Model is used to control the parallelism, such that a 3D image is partitioned into sub-images in a block mesh topology suitable for the number of available processors, and the tests repeated using varying numbers of processors. Each sub-image includes a 1 pixel wide surrounding ghost data (also referred to as 'halo' data). One process loads the 3D image, partitioning and distributing 3D sub-images to each other process according to the topology, and keeping one sub-image for itself. The immersion watershed segment algorithm is then applied locally to each sub-image. Prior to recomposition, an exchange of ghost cells is performed, this being just for test purposes in this evaluation, but with the goal being to test the functionality and performance of redistributing boundary information for subsequent processing prior to recomposition. A recomposition phase then gathers the data and recomposes into a resultant 3D image on one process for inspection, storage or further processing.

The test image is the same as that used in the previous evaluation (688x512x144, 16 bit, 97MB) and the partitioning strategy is the same in that one process is loading and serially distributing the sub-images to each process sequentially. Indeed, when using a mesh model, the partitioning strategy must partition the image onto a regular mesh topology so that the number of sub-images aligns with the number of processors available. Although the processor topology can be set explicitly, the default is to let the partitioning strategy automatically determine the optimum sub-image block-size for the available number of processors, and to use that to also extract the corresponding required topology. The default is used in this evaluation. The mesh model then sets up this topology across the processors (under the covers the MPI topology features are being used), and distributes the sub-images according to their position in the topology.

Once all the processes are initialised with the topology, and have received their respective sub-images, the segmentation computation is performed on each sub-image. After this, an exchange data step is undertaken such that all the sub-images exchange boundary data with the appropriate neighbours. For this evaluation, this ends the processing and the watershed information is gathered and reconstructed for inspection. The core interest being to get the initial segmentation algorithm to run, and in particular to test the mesh model's exchange data design and implementation is functioning and performant. Work is ongoing to embellish the segmentation algorithm itself to reduce the over-segmentation, and propagate the watershed information across sub-images, but is not pursued further here.

Table 5.4 presents the partitioning strategy for the mesh model, showing the image partitioning for each tested processor core count. Note that for up to 16 processor cores, the partitioning strategy chooses to only partition in two dimensions, with the third dimension not split up (144 pixels). This is deemed to be the best compromise to get near to an optimum sub-image shape. Note that the mesh model was not run with only one processor core, and the time it would take to execute with one processor core was inferred from the average execute only figures obtained when running on 4 processor cores.

Number of nodes	Processor cores per node	Total processor cores	Number of workers	Sub-image Block size	Number of Blocks
1	4	4	4	344:256:144	4
1	8	8	8	172:256:144	8
1	12	12	12	230:128:144	12
1	16	16	16	172:128:144	16
1	24	24	24	172:171:72	24
1	32	32	32	172:128:72	32
2	24	48	48	115:128:72	48
2	32	64	64	86:128:72	64

Table 5.4: Partitioning information for the Segmentation tests.

3D image size: 688x512x144, 16 bit, 97MB

5.5.4 Mesh Model Performance Results

Figure 5.14 and Figure 5.15 show a slice through the gradient input 3D image and a corresponding slice through the output 3D image after applying an immersion watershed segmentation. The output image is actually rendering the watershed lines as this is more illuminating, although the segment labelling is the more useful or pertinent information for analysis (but not so easy to depict). An immediate observation is the over-segmentation of the output. As discussed in the introduction to this evaluation, this is no surprise for watershed segmentations of noisy, high resolution images, and pre-processing and further post processing techniques are usually applied to improve the segmentation prior to feature extraction and analysis. The over-segmented output is sufficient for the purposes of this rudimentary evaluation, that is initially focusing on the mesh model parallelisation and data exchange aspects.

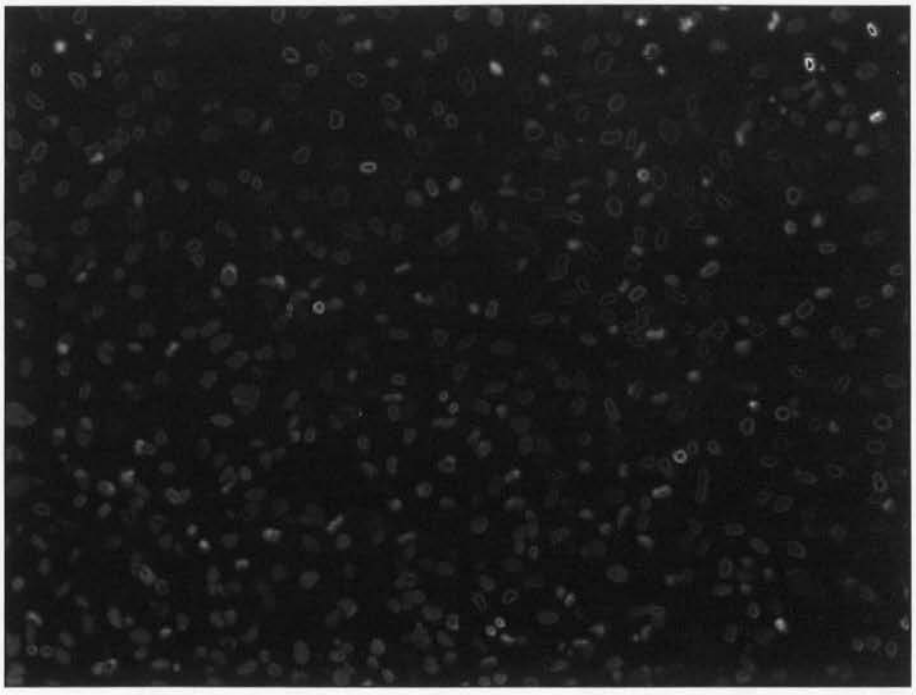


Figure 5.14: Segmentation input image slice (Sobel operator output image x-y slice detecting nuclei of labelled sarcoma cells)

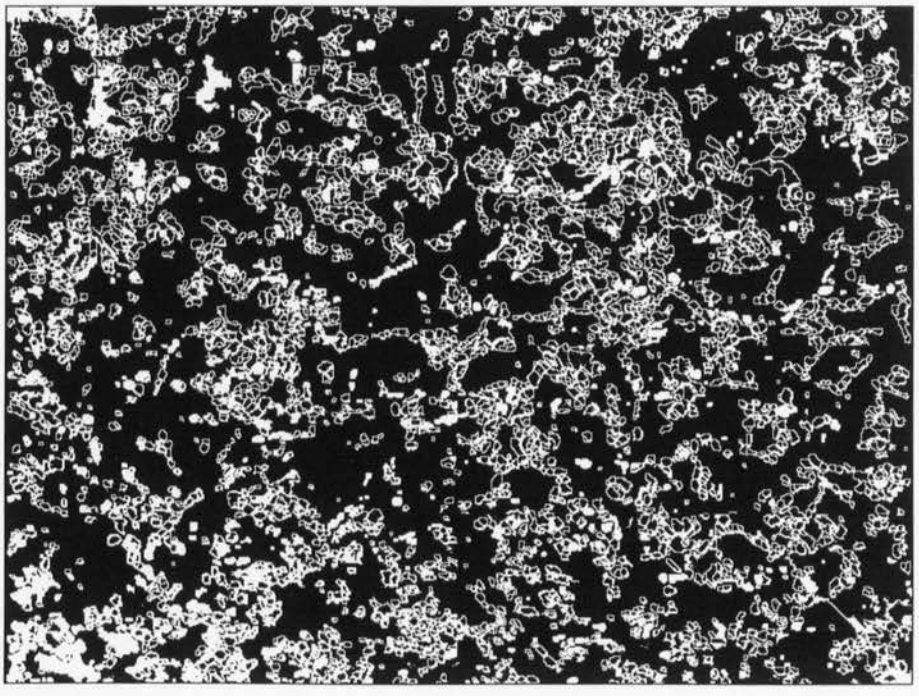


Figure 5.15: Segmentation output image slice of labelled sarcoma cells

The performance plots shown in Figure 5.16, Figure 5.28 and Figure 5.18 present the execution time, speedup and efficiency results. The plots show the total combined time to partition, distribute, execute, exchange data, gather and recompose an image. The impact of generating and distributing the sub-images and gathering and recomposing the results is apparent. Although the speedup is encouraging, the burden of parallelising is significant.

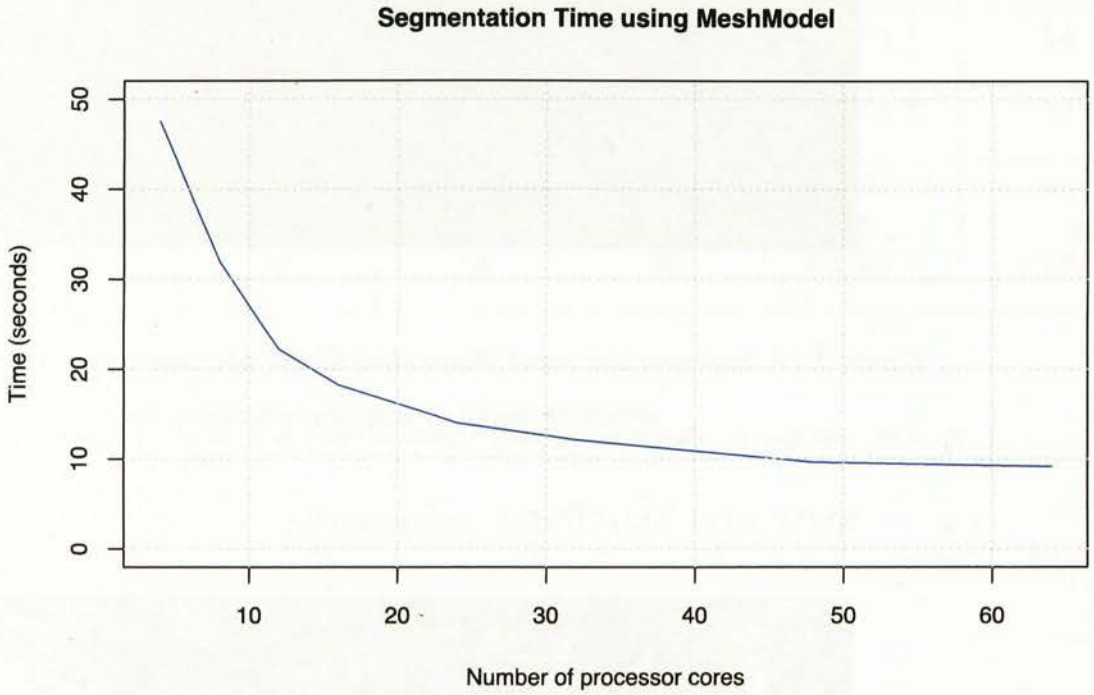


Figure 5.16: Processing time of a 3D watershed segmentation operator

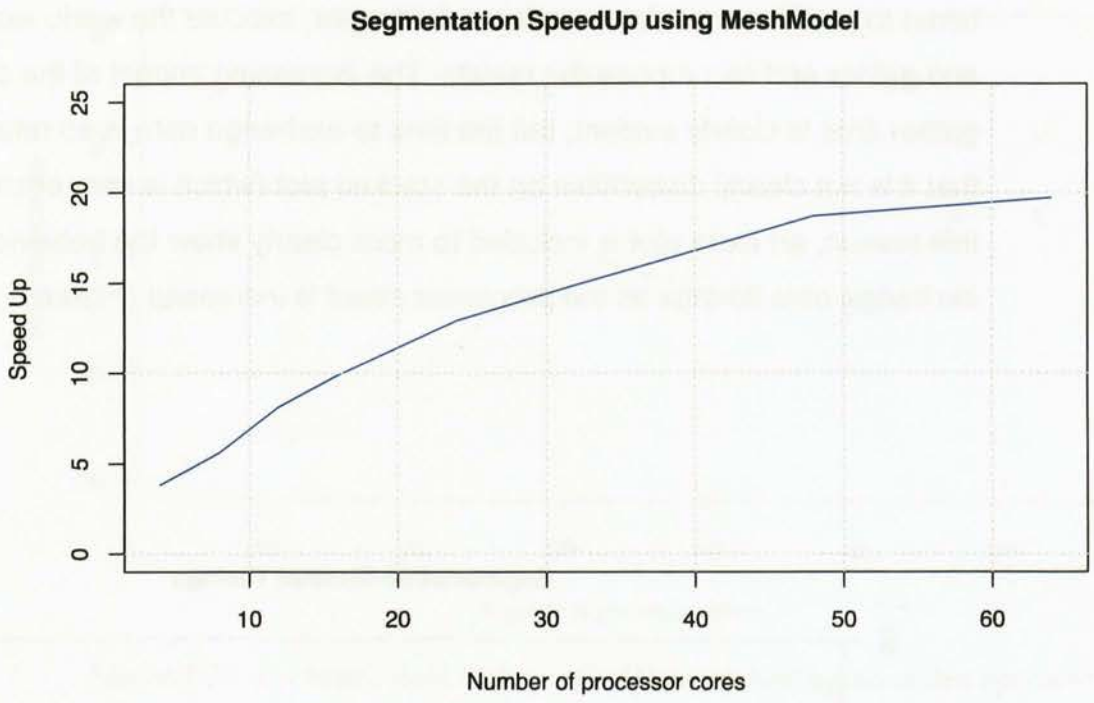


Figure 5.17: Speedup of a 3D watershed segmentation operator

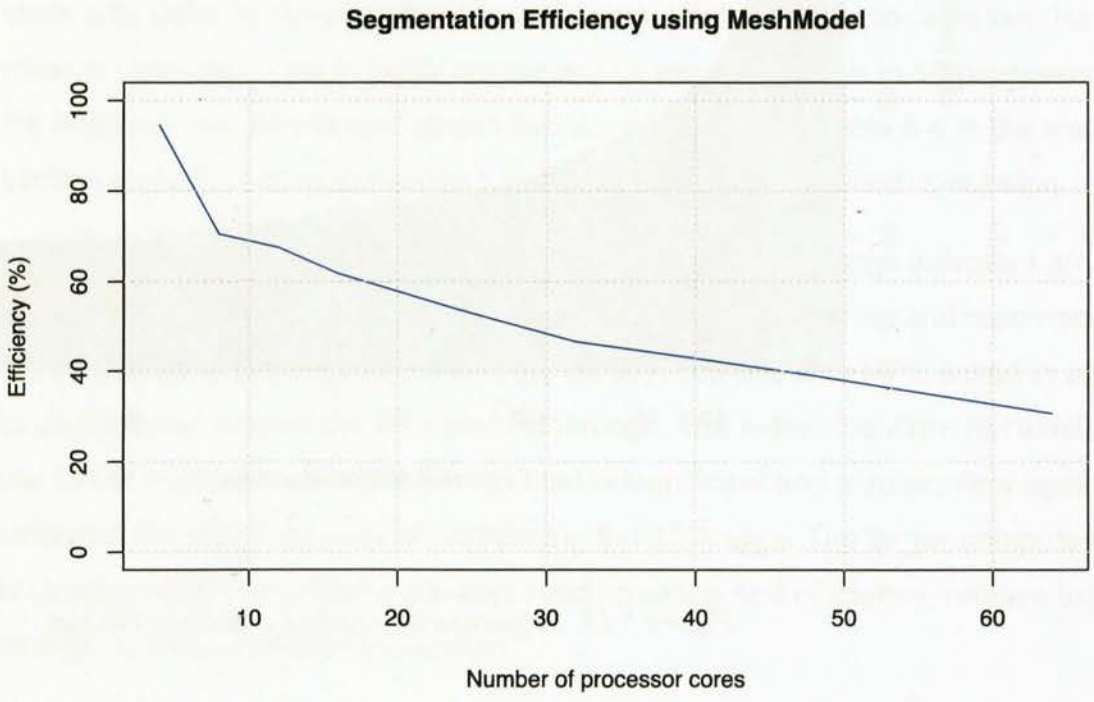


Figure 5.18: Efficiency of a 3D watershed segmentation operator

As the exchange data aspect of this evaluation is of central interest, an extra 'stacked' plot (Figure 5.19) is provided to shine more light on the proportion of time taken to partition and distribute the sub-images, execute the work, exchange data and gather and recompose the results. The increasing impact of the distribute and gather time is clearly evident, but the time to exchange data is so relatively small that it is not clearly discernible on the stacked plot (which is very encouraging). For this reason, an extra plot is included to more clearly show the behaviour of the exchange data timings as the processor count is increased (Figure 5.20).

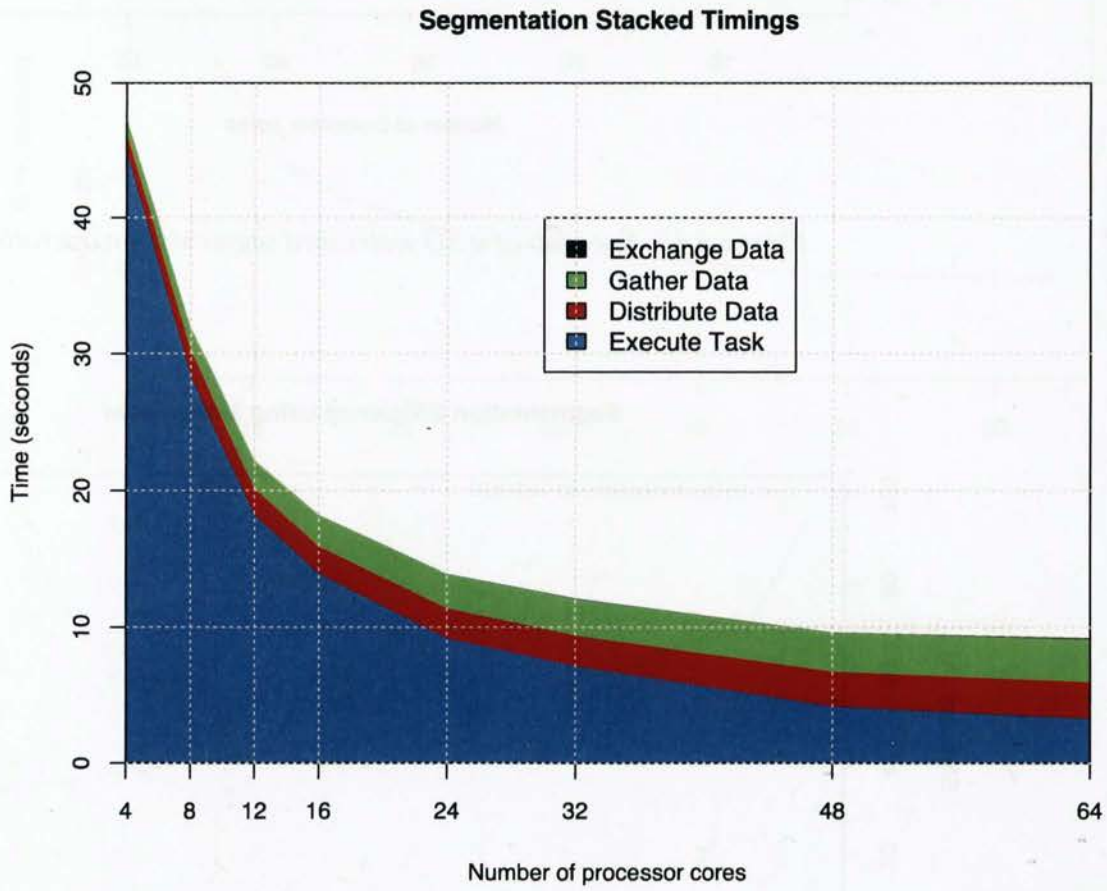


Figure 5.19: Segmentation stacked processor timings

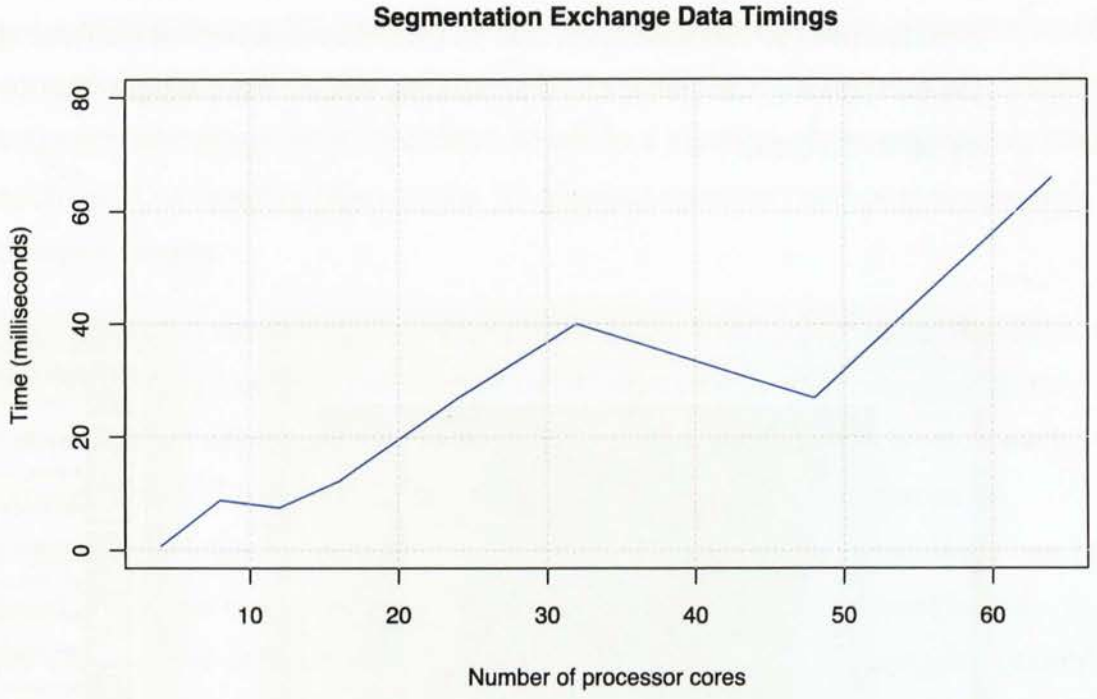


Figure 5.20: Exchange data timings of a 3D watershed segmentation operator

The exchange data plot is interesting, as it appears counterintuitive. The more logical expectation would be that the exchange data total time would reduce, as the amount of boundary data to exchange reduced due to the smaller sub-image block size used as the processor count increased. Instead, a variable but rising trend is observed. This is partly accounted for because for up to 16 processors, the image is only partitioned across two dimensions (see Table 5.4 in the method section above), removing the need to exchange data in the third dimension.

To investigate further, and in particularly as the DFrame timings indicated an increasing burden due to the partitioning, distributing, gathering and recomposing the sub-images, it was deemed worthwhile to recompile with MPE linked in again, to capture and inspect the MPI profiled timings. The results confirm that similar to the Sobel evaluation, the distribution cost is significant and a zoom view again indicates the dominant cost of partitioning the 3D image. The larger compute time is clearly evident, explaining the improved speedup and efficiency, relative to the timings in Sobel operator evaluation.

The exchange data figures are so small as to not be discernible in the MPE/Jumpshot full view (top left of Figure 5.21). However, zooming in to investigate (not shown), it was found that there is significant variability in

processors joining the MPI_Cart_shift collective operation. Although a barrier was inserted prior to the collective, and all processors leave the barrier together, some joined the MPI_Cart_shift call immediately while others lagged (for 16 processors, by up to 1.76ms).

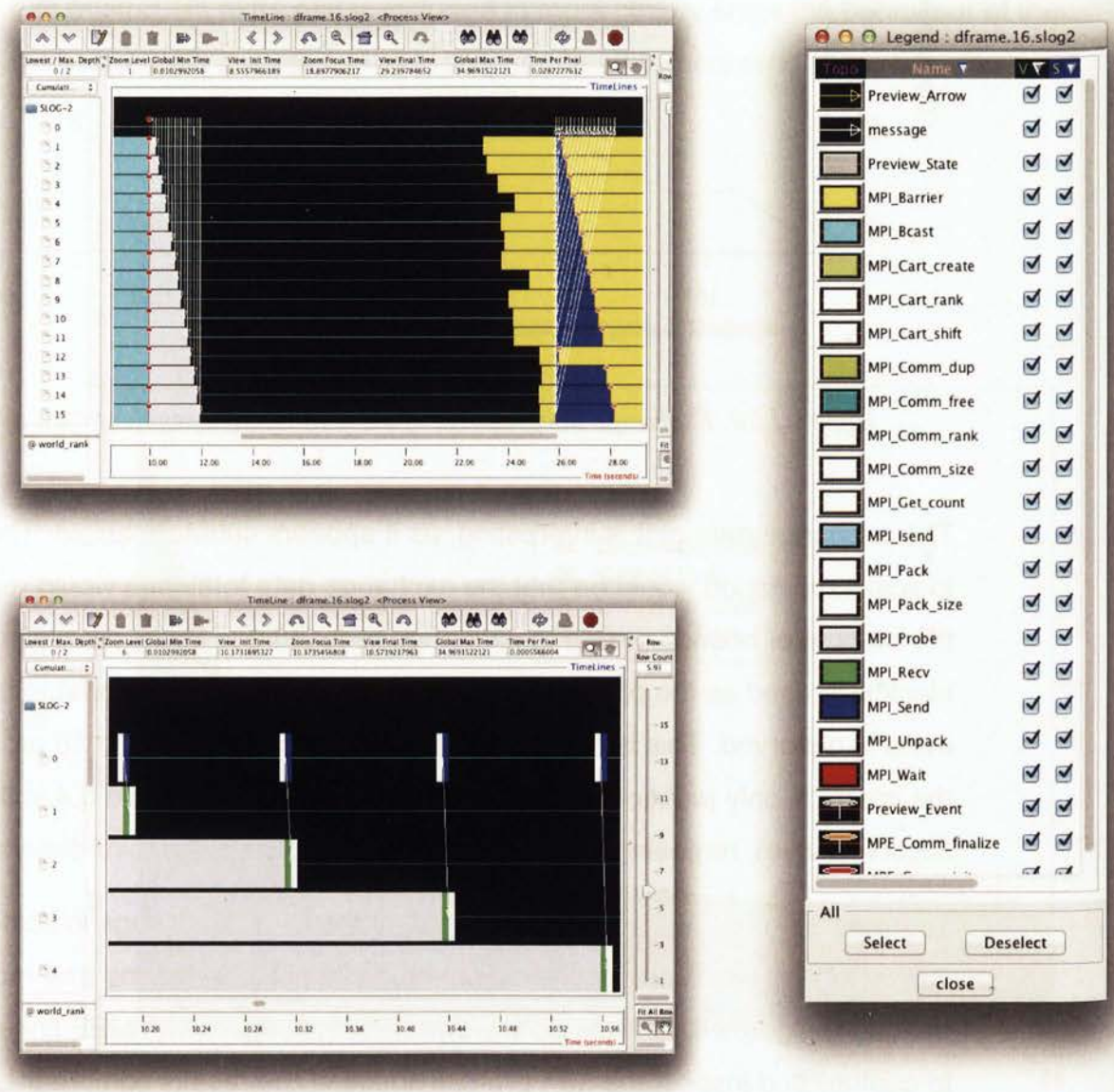


Figure 5.21: Segmentation timings using 16 processor cores (MPE/Jumpshot)

5.5.5 Discussion

This evaluation is centred on using a mesh model and 'halo' exchange to parallelise a watershed segmentation algorithm. As the previous evaluation has highlighted, using a mesh model is an attractive strategy to amortize the cost of

parallelisation when multiple operators can be applied to the partitioned data, prior to gathering the results. This is a good fit for image processing in general and for common segmentation approaches, which can entail multiple operations within the segmentation stage itself, and often sit within a pipeline of pre-processing stages such as 3D averaging filters and a 3D gradient operator, and post-processing analysis stages.

The presented results appear more positive than those in the first evaluation, and this is primarily because the segmentation algorithm is even more compute intensive than applying a Sobel operator, so that the execution time on each processor is still a significant proportion of the total time even across 64 processors. However, as the same partitioning and distribution strategy has been used, the same observations apply to this evaluation, as witnessed in the MPE/Jumpshot detailed view (bottom left of Figure 5.21). In particular, priority should be given to optimising the high level image structures, while retaining the intent to provide abstraction, clarity and productivity at the application layer. One further point is that when parallelising the segmentation algorithm, in addition to exchanging the data, further processing must be done to merge boundary information and relabel so that a global labelling is achieved, and this may involve multiple iterations (not included in this evaluation, but acknowledged as would further impact performance). This extra work is not usually required in a sequential algorithm. One notable issue evident during the evaluation was that due to the over-segmentation, the number of labels required was large, and potentially more than could be represented in a 'short' numeric type in a sequential implementation, although this could be alleviated with pre-processing to smooth out the image (a short data type is preferred, to reduce memory footprint and communication data size). Additionally, the representation of a watershed image that comprises intensity, distance and label information on each voxel inflates the memory requirements. These latter points also being good arguments for parallelising the segmentation of 3D images to reduce memory usage.

The success of the mesh model to process multiple imaging operators hinges on the performance of the 'halo' exchange: the local exchange of ghost or halo data amongst neighbours between each operator. Here, the results are impressive, showing that the halo exchange adds little to the overall execution time. In the exchange data design, non-blocking sends are introduced as each process must

communicate with as many as 6 neighbours, and so the cost of creating the boundary sub-images to exchange with each neighbour can be overlapped with communication to other neighbours, to optimise performance. Some variability in the MPI_Cart_shift collective operation is evident. This could be due to some anomaly in the mpi implementation, or it could of course be an architecture issue, or some usage issue pertinent to the evaluation (all of which would have to be considered in further investigations). OpenMPI 1.4.2 is used in the case studies, and version 1.8 is now available. One of the huge benefits of open source software is that the extensive use and development cycles incrementally improve the software, and this area may have received attention. So an upgrade should be considered as part of further tests. Notwithstanding this, the results prove that the mesh model and halo exchange combination are a performant parallelising strategy when appropriate. The ghost cell pattern can as well be used in investigating the most efficient implementation of the initial distribution of ghost partitions, where sub-images are collectively distributed and ghost cell information is subsequently exchanged prior to the start of a computation phase, rather than sub-images that include ghost cells being initially distributed.

The partitioning of the 3D image leads to the requirement to exchange boundary information prior to further operators. Also, where an algorithm has a global element, further iterations may be required to align all the sub-images, to complete the current operator (also requiring neighbour data exchange). This highlights the well known notion that parallelising algorithms that operate on local regions is more successful than parallelising global operators, and efforts to devise these should take precedence (such as the way neural networks work with local operators). Alternatively, strategies such as increasing the depth of the ghost cell region to exceed the expected size of cells could be considered. Other more innovative partitioning strategies could be sort, that partition the image in more novel ways, as sub-image data that overlapped more fully, or partitioned into sub-images that span the whole image but at some lower resolution and shifted phase. The partitioning strategy influences the distribution strategy, and a modification to the current distribution strategy is planned, to enlist the help of multiple processors to distribute the data, such that all processes in the x-dimension receive sub-images, and these then distribute to processes in the y-dimension, and then all processes in the x-y-dimension distribute to the z-dimension. This will then be compared with the one processor partitioning and distributing all the sub-images to

all processors serially. An optimum strategy may also include the use of mpi's parallel IO functionality, as mentioned in the previous evaluation.

As well as the many alternatives and combinations that can be used to parallelise computation, different segmentation approaches can be used, some of which may be a better fit with parallel processing. For instance, finding the image minima in one step and then segmenting using the watershed or other region growing techniques based on only these minima. The segmentation may form intermediate representations such as a lower complete image and then a mosaic image that is then piped to further segmentation algorithms. Indeed, splitting up the process can give more flexibility and allow experimentation at various stages. Although the output from the segmentation evaluation depicts a watershed output image, in reality the output would be cell features such as volumetric size and shape, that are then piped into an analysis stage. The output of the analysis stage would then identify unusual cells for further scrutiny. This then would provide information that can be used in a visualization stage, to zero in on the interesting artefacts uncovered (see next evaluation).

One final point is that in a high content screening application, many images will be processed and in this scenario it would be sufficient to process one image per processor and completely avoid the overhead associated with partitioning and reconstructing images (but note that the memory implications would have to be assessed).

5.6 Visualization Ray Tracing

5.6.1 Background

Visualisation is the process of translating a dataset into a form suitable for viewing, and this evaluation focuses specifically on the visualization of 3D image datasets. Once cells have been segmented, and prominent characteristics extracted, analysis may identify unusual cells for further automated analysis or manual operator inspection and review. As well, the ability to interactively visualize 3D images while adjusting the parameters of any component of an imaging pipeline provides direct feedback of the effects of such changes, helping to tune a pipeline for a particular purpose and check that the system is producing the intended output. Indeed, visualization can be applied to any appropriate pipeline stage to

check the output, aid tuning and provide reassurance that each stage is operating as expected. Although the mechanics of 3D visualisation is somewhat more involved than for 2D images, the extra degree of freedom adds scope for a much more immersive experience. Using a camera metaphor, a user can 'fly' through the 3D image space, adjusting orientation and zoom, to inspect the whole image, or any artefact within the image at close range, from any angle. This introductory section examines polygonal rendering and volume rendering of 3D images, with the evaluation then using the DFrame to apply 'direct volume' cluster rendering to visualize 3D cell images.

Mainstream graphics processing units (GPU's) primarily support fast, hardware accelerated parallel rendering of multi-polygon representations of 3D objects within a 3D image, with OpenGL (OpenGL Architecture Review Board, Shreiner 2004) being the de-facto powerful standard programming API for rendering such primitives, including model-view transformations, projection transformations, lighting calculations, clipping and hidden surface removal. In artificial 3D images, objects can be directly specified as polygons. However, for raw 3D images, a conversion must be made to extract polygonal surfaces from the image, suitable for the GPU's graphics processing pipeline, one well known example being the 'marching cubes' technique that extracts surfaces of equal intensity as polygonal representations (Lorensen, Cline 1987). The number of polygons generated to represent an iso-surface can be very large. Alternatively, where GPU's are not available (e.g. thin client) or the data is too vast, 'volume rendering' can be arranged on a server cluster. In this approach, polygons are still the primitives, but the processing of an image is arranged across a server cluster, with nodes in the cluster then processing a subset of the primitives or image space. A compositing step assembles the resultant 2D image for display (Molnar, Cox et al. 2008). Volume rendering of scenes of polygonal objects is also a seasoned technique used for shading realism (Appel 1968).

Direct volume rendering (ray casting) is an image order volume rendering technique to directly render 3D images without the need to convert to a polygonal representation (Levoy 1990). Rays are propagated through the image and the resultant intensities to be rendered are calculated directly, according to a ray function. Although computationally intense and requiring recalculation whenever the viewing position is changed, the technique is appealing for its ability to produce

realistic shading, and is arguably more suitable for biomedical datasets as polygon triangulation may remove some important detail. An additional attraction is the flexibility of the ray function used, which can be adapted to provide effects that are difficult to achieve with a polygonal approach (Hege, Hollerer et al. 1993). Ray tracing does not align as well with a GPU's vector processing pipeline, but is suited to server side (cluster) rendering and as such is an ideal candidate for parallel processing using the distributed framework. For a server rendering example, see "Parallel rendering on the IBM Blue Gene/P" (Peterka , Yu et al. 2008). To test the DFrame, this evaluation uses a simple threshold based ray function. It is anticipated that the ray trace library will also eventually provide a variation where instead of detecting iso-surfaces, intensity samples will be taken at regular intervals as the ray penetrates the volume, and these values accumulated according to some ray function to compose the final 2D image (Razdan , Patel et al. 2001).

5.6.2 Ray Tracing Module Tests

One common requirement is to inspect specific areas (e.g. cells) in an input image, possibly with some preprocessing, guided by information in the output of a processing pipeline. As such, the method here uses raw images to test 3D direct volume rendering using the DFrame. A smaller image is first used as a preliminary functional test, to establish that the software is working and producing expected output, and that the camera implementation is operating correctly to navigate through the image. In short, the DFrame is initialised across a number of processors, and the root process distributes a specification that instructs each instance that a master-worker model is to be set up. The specification also informs each instance to load the Ray Tracing Module, to plug into the master-worker model, and contains initialisation parameters for the camera. Each worker then loads the entire 3D image. The DFrame continues on to run the master-worker model, the master then partitioning and sending an equal number of rays to each worker, according to the camera's configured parameters, which dictates the number of rows and columns of pixels in the output image. It is the ray trace module software that determines the partitioning, and for these tests a column strip partitioning of the output image was used to generate ray information for each worker.

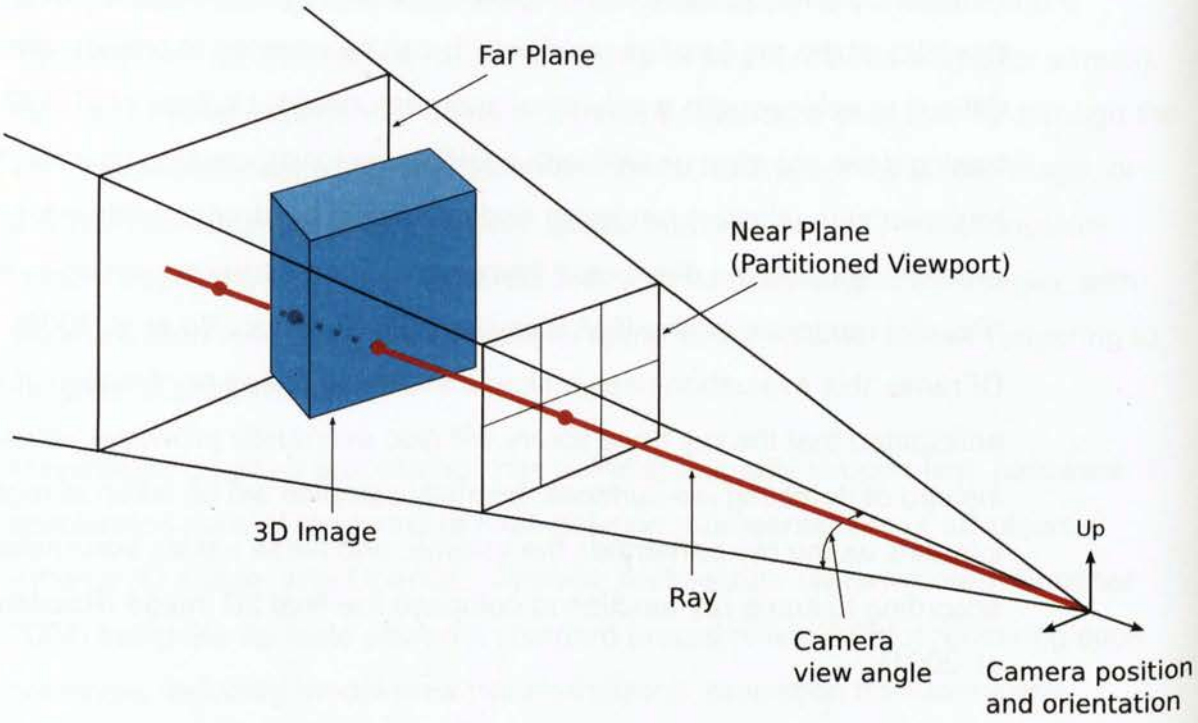


Figure 5.22: Camera orientation and ray trace schematic

Figure 5.22 shows a schematic of the camera and ray trace mechanism. The camera follows a common design allowing the user to specify its location and orientation, together with a camera view angle, near plane and far plane distances (Hill 2001), (Astle, Hawkins 2004). The viewport is set to the near plane, and it is this 2D viewport that is partitioned into sub-images, as shown in the figure. Workers are then sent an instruction message that specifies the start row and column, and the width and height of the sub-image (i.e. output partition) that they should work on. Each worker then uses the partition and camera information to drive the image application ray tracing implementation for all pixels in the viewport 2D sub-image and returns the result to the master. The master composes all the sub-images to form the complete output 2D image represented by the viewport.

The ray tracing method is to first form a bounding box around the 3D image, and transform that to a generic cube for efficient ray intersection tests. Then, for each ray to be traced (and according to the orientation of the camera), an initial check is made to see if the ray intersects the bounding box. If the ray misses the image's bounding box the processing of that ray is complete. Otherwise, the ray is stepped into the image volume using a variant of "A fast Voxel Algorithm for Ray Tracing"

(Amanatides, Woo 1987), until a set threshold is detected or the ray passes out of the volume again.

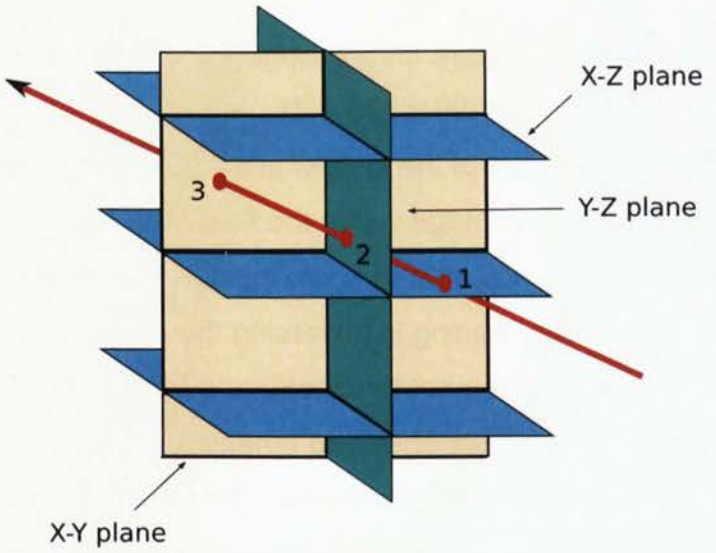


Figure 5.23: Ray Trace detail through 3D image planes

In Figure 5.23 the ray trace stepping is shown on a small sub-volume of the 3D image. A ray is passed through the volume, and at point 1 intersects the X-Z plane. In detail, all three planes are checked to evaluate which one the ray hits next, as it propagates through the image (on each step). These planes are the image slices in each dimension, and a ray will pass through one image slice after another for planes it is not running parallel to. The intensity at the point is then interpolated from the 4 known intensities that surround the point on that plane. Assuming the intensity is below the threshold, the stepping continues, and at point 2 the ray intersects the Y-Z plane. The intensity is similarly interpolated at this point from the known surrounding points on this plane. Again assuming the intensity is below the threshold, the stepping continues until point 3 on the X-Y plane is intersected and the intensity similarly interpolated from surrounding points on this plane. Now assume that the intensity at point 3 is above the threshold, then a further interpolation between the intensities at point 2 and point 3 is calculated to determine the actual point at which the intensity matches the threshold (on the line between point 2 and point 3). Once this point is found, a further calculation is made to establish the normal at this point, and the intensity value to be rendered is

computed using the dot product of the normal and lighting vector. The lighting vector is fixed to be behind the camera for this evaluation.

5.6.3 Parallelised Ray Tracing Results

Figure 5.24 and Figure 5.25 present the initial results of ray tracing through a smaller 3D image (696 x 520 x 21). In Figure 5.24, the specification configures the camera at the front of the image, at such a position and camera angle as to contain the whole image within the field of view. In Figure 5.25, the camera parameters are adjusted to zoom into an area of interest in the image. It can be seen that the ray tracing is traversing the image and calculating the normals at incident points at the set threshold, and setting the intensity such that the 3D nature of the image is observed (shading). The camera orientation provides the information on the camera location, the direction it is pointing, and a vector determining the camera position on the view axis, commonly referred to as the 'up' vector.

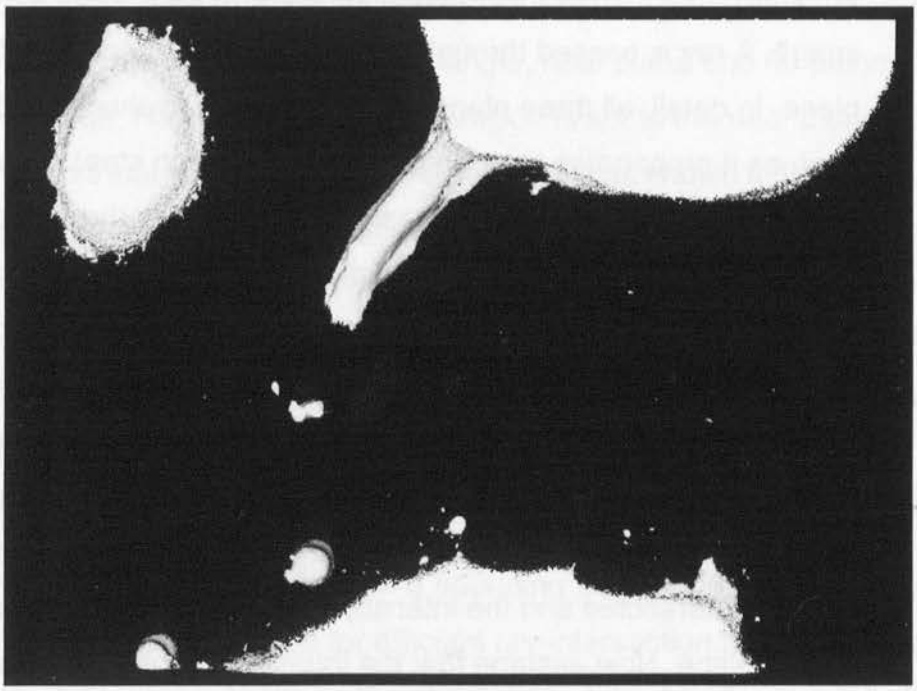


Figure 5.24: Ray trace full view of sarcoma cells 14MB 3D Image

image = 696x520x21, 16 bit, threshold = 850, camViewAngle = 50.0
camNear = 500.0, camFar = 1000.0, camWidth = 696, camHeight = 520
camOrientation = 348,260,600,348,260,0,0,1,0

Figure 5.25: Ray trace zoom view of a single sarcoma cell 14MB 3D Image

image = 696x520x21, 16 bit, threshold = 850, camViewAngle = 40.0
camNear = 500.0, camFar = 1000.0, camWidth = 696, camHeight = 520
camOrientation = 80,100,320,80,100,0,0,1,0

Having established the correct working of the ray tracing implementation, a larger image was used to actually test the performance of the DFrame. A similar procedure was undertaken to set up the DFrame, distribute specifications and work, and collect the results. Alongside this, the DFrame configuration was modified to switch on the gathering of timings from each DFrame instance.

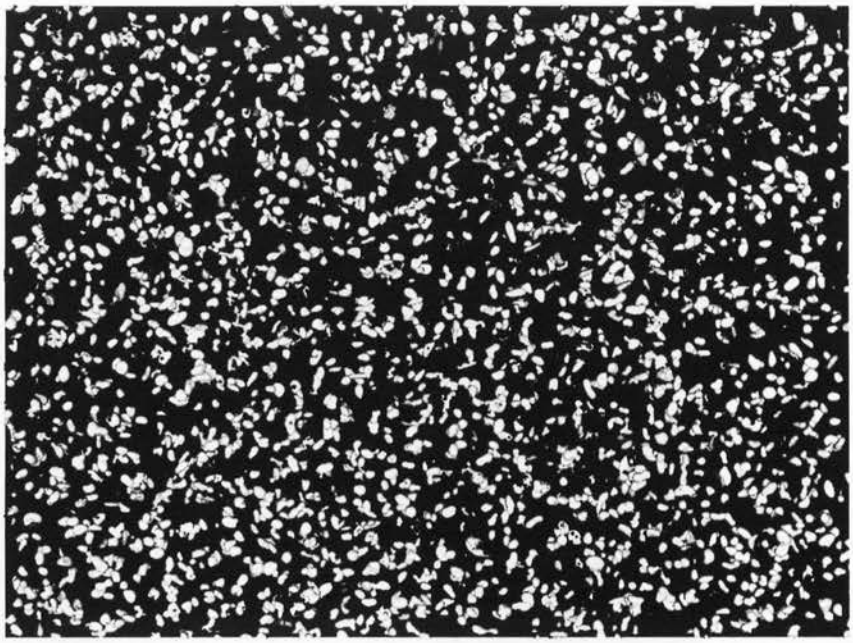


Figure 5.26: Ray trace full view of multiple sarcoma cells. 97MB 3D Image

image = 688x512x144, 16 bit, threshold = 850, camViewAngle = 50.0
camNear = 500.0, camFar = 1000.0, camWidth = 688, camHeight = 512
camOrientation = 344,256,600,344,256,0,0,1,0

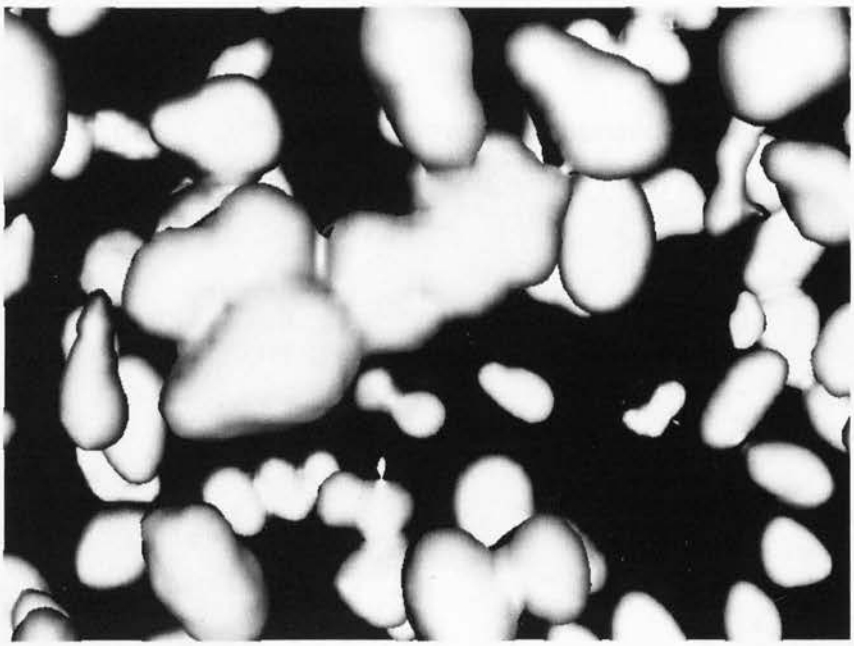


Figure 5.27: Ray trace zoom view of multiple sarcoma cells. 97MB 3D Image

image = 688x512x144, 16 bit, threshold = 500, camViewAngle = 25.0
camNear = 500.0, camFar = 1000.0, camWidth = 688, camHeight = 512
camOrientation = 80,110,200,80,110,0,0,1,0

The performance results presented below are for the ray tracing specification that generated the output shown in Figure 5.26. Equally sized column oriented strips of rays were sent to each participating worker, the rays traced and results returned to the master, and saved as a 2D pgm image. The presented timings exclude the loading of the 3D image onto each node and the saving of the output (after composition), the rationale being that the loading and saving would also be borne by a sequential run. More importantly, the intention is to have the images loaded once, and then be able to navigate around the 3D image by sending updated camera information to each worker for an immersive long running real time interactive experience with data already loaded across the cluster (until the user decides to end the session) and the presented timings give a good indication of the performance in this regard. The timings do include the recomposition of the 2D output image.

In Figure 5.28, Figure 5.29 and Figure 5.30, the execution time, speedup and efficiency graphs are plotted. The results are very encouraging, and not unexpected, as the workers operate independently on a subset of rays for the output image. The time to perform a trace of a particular ray will depend on a number of factors: whether the ray intersects with the image's bounding box, and if so the depth that has to be traversed through the image until the threshold is encountered, or the ray completely traverses through the 3D image. This will vary for each ray, giving a diverse work load. This appears to be reflected in the results, such that a very good linear speedup is seen, but that falls short of the theoretical optimum speedup, with the efficiency dipping to 60% for 64 processors. The time to completely render an image will be influenced by the worker that takes the longest to process the rays assigned to it.

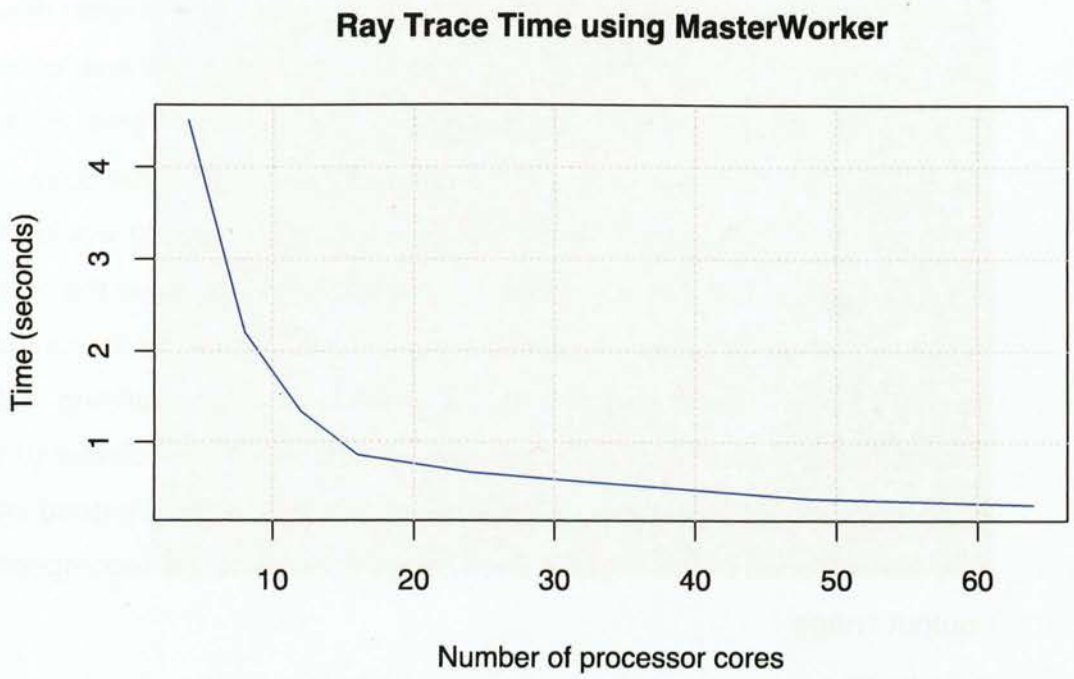


Figure 5.28: Ray trace execution time

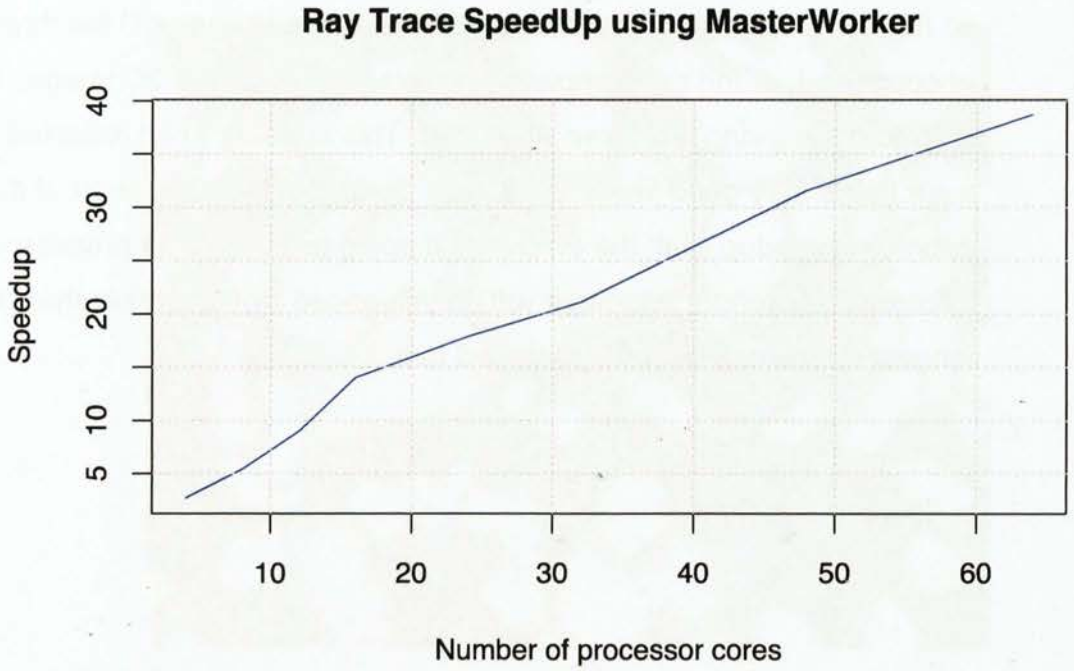


Figure 5.29: Ray trace speedup

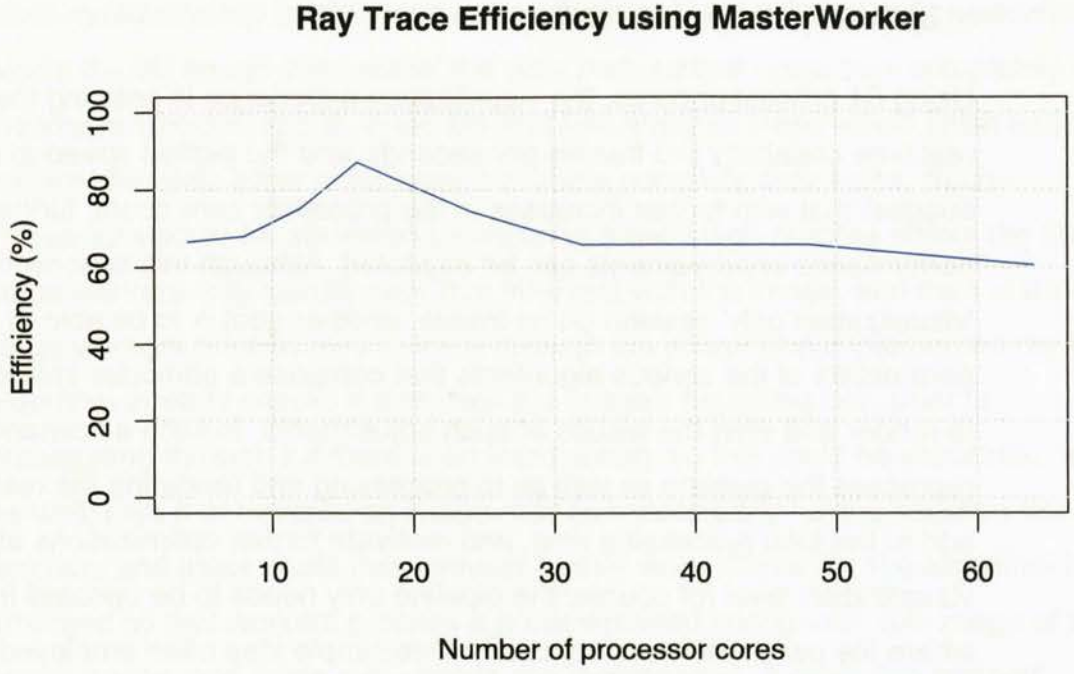


Figure 5.30: Ray trace efficiency

Is the speedup sufficient to provide real time interaction? Figure 5.28 plots the decreasing execution time to ray trace the 3D image as the processor core counts are increased. Using 64 processor cores the time to trace the full image is 0.3 seconds, or over 3 frames per second. This is an acceptable starting point into real time interaction, that is considered further in the following discussion.

5.6.4 Discussion

The goal of this evaluation was to develop a 3D image direct volume rendered visualization capability, and to integrate it into the DFrame to leverage cluster parallel resources to speed up the compute intensive algorithm to provide a real time interactive experience. Another core aspect was to test the extent to which the domain specific algorithm could be kept separate from the parallelisation framework. The separation has proved successful, as each ray trace can be represented as an independent task and the framework only has to determine how to distribute the ray processing to workers, which is accomplished outside the application code and transparent to it. The results also conclude that the performance improvements are commensurate with expectations. The irregular work load, in that each ray trace may take a different time to complete is

illuminated in the results, suggesting that other optimisations could be attempted to optimise performance further.

Using 64 processor cores, the visualization experience is entering the realm of a real time capability (>3 frames per second), and the plotted speedup results suggest that with further increases in the processor core count, further performance improvements can be expected. Although this is acceptable for a 'visualization only' session on an image, another goal is to be able to adjust parameters of the various algorithms that compose a particular image pipeline and to render and view the results of such adjustments. In such a scenario, the time to reprocess the pipeline as well as to processing and rendering the results would add to the total processing time, and motivate further optimizations at the visualization level (of course, the pipeline only needs to be updated from the node where the parameter was adjusted). One simple idea often employed is to introduce a 'level of detail' parameter, which in this context would mean that only a subset of the rays are traces, with each ray contributing to a small block of pixels in the resultant less detailed 2D image (block size controlling the level of detail). The facility to do this has been added to the algorithm, to be propagated to the UI for user control and it is envisaged to semi-automate this such that more detail can 'flow in' as processing time allows. Another facility is to only show a wire frame of the image's bounding box, and to allow the user to position the camera according to the wire frame, prior to switching on the compute intensive ray tracing. This has been tested on an early QT 3 implementation, and is being ported to the current QT 4 UI. Essentially, openGL is used to render a wire frame bounding box of the image within the view port, and as such determine the camera parameters, which are then used in the parallelised ray tracing step.

What other improvements can be made? A prime intention, now the core apparatus is in place, is to inject other ray tracing strategies into the ray tracing algorithm, and it would even be interesting to investigate the generation of secondary rays for processing across the cluster. The anticipation is that even more realistic effects will be attained as other novel ray tracing strategies are introduced. There is also more to investigate in regard to the impact of updates to the partitioning strategy and the related load balancing to further improve performance. This ray tracing evaluation uses a simple column strip partitioning strategy. It is assumed that a row strip strategy would be similar and for the type of

images tested (and the tested camera orientation), and a more general block or block cyclic strategy might prove more effective. More relevant will be situations where the 3D image does not fill the view port, so that some rays completely miss the image's bounding box. A worker that has many of these would finish early and remain idle while other more loaded workers complete their tasks. Such imbalances could be alleviated by applying a two stage process where the first stage workers only identify rays that intersect with the image, and then in a second stage workers retrieve rays to trace through the image (the implemented ray trace algorithm already checks if a ray hits the image's bounding box, prior to propagating through it if there is an intersection, so this could be separated out). As well, rays that traverse an image may terminate early having satisfied some ray function, and these could then request further work. Currently, the algorithm is arranged so that workers process a predetermined contiguous sub-image of the output image, and these sub-images are recomposed. A more fine grained approach can be taken by simply reducing the size of the sub-images, and this would be expected to alleviate some of the imbalance (i.e. adjust the block size). A more general approach would allow each worker to grab arbitrary rays, but in this case the gathering strategy would also have to be adapted. In any case performance tests would need to be conducted to establish whether the trade off between an improved load balancing through more fine grained tasks and the extra communication involved would be justified by improved processing times. That the results would vary from image to image suggests that further research would also be helpful to establish characteristics of an image that hinted at what might be an optimum strategy (and that might even change as the camera parameters are changed).

The master-worker model is well suited to the irregular load expected in ray tracing a 3D image. The DFrame also supports a mesh model, aimed at more regular processing operations across an image, and primarily used where exchange of data amongst neighbours would be required. If a processing pipeline has already partitioned an image as 3D sub-images in a mesh topology, it would be interesting to test if ray tracing could be arranged across such a structure. Each sub-image could receive camera information, compute rays that hit it, and a compositing step could then be used to determine the final result through a collective gather compositing operation. Although this doesn't seem like it would be as performant as the master-worker approach, it may nevertheless provide further insight and be

a useful addition to the 3D image processing tool bag. A likely show stopper being that under close zoom, one process could be doing much of the work. But this then brings into question whether there should be a max zoom - as a view that is mostly interpolated could be misleading. On the other hand, workers would not have to load an entire image, and in some circumstances this would be attractive, such as when visualization is targeted at a session where the image under scrutiny is being updated often through image processing pipeline parameter modifications. Communication is a common bound to parallel processing, and its minimization is core to performance. Choices include moving processing to nodes containing data subsets, moving data to processing nodes, or some hybrid scheme dependent on the application (Nebel 1998). It would be worth further investigation to determine whether ray tracing would benefit from a hybrid approach.

5.7 *Summary*

The aims in this chapter were manifold. A prime intention being to put the DFrame through its paces, at the individual task level. In doing this, the flexible plugin model and module architecture could be established as functional and performant. Functional in that the DFrame was receiving and broadcasting the task specification for an individual task and loading and running models suitable for the particular task, that then ran the specified application code, partitioning executing and composing data as appropriate. Performant in that speed up was observed, and related to this, testing that the capture and gathering of the distributed metrics was functional and reporting the overall timings and timings of key steps in the running of each task. The DFrame was configured to gather the timings in a csv format for analysis. Another aim was to evaluate the integration of 3D image processing algorithms into the DFrame. In this respect, the application of the DFrame to a specific domain provided insight into the integration requirements, in terms of which models would be most appropriate, and also the ramifications of the partitioning strategies. These insights would then feed back into further refinement of the design at the DFrame and model integration points and at the model to application interfaces. As anticipated, the partitioning has proved to be of central importance to the successful execution of a task in terms of performance. Alongside this, the various models that encapsulate the partitioning and associated runtime logic also influence the outcome. A master worker model was

initially evaluated, and subsequently a mesh model was introduced as the need to exchange neighbour data became apparent. This was at first expected to be only useful when the exchange of data was necessary during the progression of a specific algorithm (e.g. the watershed segmentation example), but it became apparent from these evaluations that the gathering and redistribution of data on each task would be a performance impediment that could be also avoided using this mechanism. The success of the exchange data evaluation suggested that partitioning strategies should be assessed not only at the individual task level, but also in the broader context of a pipeline that the task forms a part of. In this way, the optimum partitioning strategy (and model) may be evaluated using broader criteria than the individual task. This insight is further examined in the next chapter.

At the individual task level, the results are very encouraging, showing respectable speedup in all the tests. The ray tracing evaluation was particularly successful. Of course, the DFrame is motivated by a purpose, and that is the parallelisation of image processing applications. In this respect, a number of useful algorithms have been implemented for these evaluations, that are added to an expanding toolkit that the application programmer can utilise to construct bespoke applications. The averaging filter and Sobel operator utilities are straightforward implementations, with the watershed segmentation being more complex, and requiring further experimentation in its own right. The design and implementation of the ray tracing is also a formidable exercise including a server side camera model, that application programmers can reuse. The design is such that ray tracing functions can be adapted, offering interesting research scope in this area at parallelised speeds. Visualization of 3D biomedical images through direct volume ray tracing is a core goal of the project, being an important and novel feature for in depth real time assessment of intermediate snapshots as well as pipeline final outputs, particularly in HCS applications.

The discussions sections provide detail on each specific evaluation, assessing the performance and also providing useful insight into the further development of the DFrame architecture. Points that are now being designed and implemented. Overall, the evaluations support the DFrame approach, that models expressing specific parallel patterns and running over an SPMD model can provide significant speed up in the image processing domain (and this is likely to generalise).

Chapter 6 A DFrame Application to Analyse Multiple Sequenced 3D Bio-Cell Images to Detect Sarcoma Cell Invasion Signatures

6.1 Introduction

Chapter 5 centred on evaluating the DFrame at the component (model) level, with the evaluations conducted in the context of 3D cell biology imaging. The evaluations included parallelised components to de-noise 3D images, and to apply edge detection and region based cell segmentation techniques. A parallelised visualization capability was also evaluated. The objectives were twofold, firstly to demonstrate the design of the DFrame at the component level, and secondly to provide a 3D imaging capability to filter, detect and segment cells in 3D cell biology images, together with bridging infrastructure to link into the DFrame. These components forming a useful initial collection of functionality in a basic toolkit for the construction of practical parallelised 3D imaging applications. The evaluated components can be brought together to form the basis of an application that segments 3D cell images, with the expectation that other components would be developed and plugged in to provide further analysis of the segmented cells, and so enable the identification of morphological characteristics of the cells.

This chapter now goes on to present a fully integrated case study that clearly demonstrates the DFrame capability in terms of flexibility and adaptability at both the component level and the task graph level. To this end, an integrated pipeline of tasks comprised of 3D image operators is evaluated. Rather than composing the previously evaluated components to provide an application that will segment and visualise 3D cell images (which hopefully is self evident), this case study focuses on a slightly different imaging pipeline that looks not at the segmentation and analysis of the features of individual cells in one 3D image, but on the temporal study of cell migration over multiple 3D images. The application simultaneously analyses multiple temporally sequenced 3D bio-cell images in dynamically created and concurrently executing distributed pipelines. This is an important further assessment, as it focuses on the operation of the DFrame in executing multiple tasks whose individual characteristics impact the overall execution strategy. In this sense, the system is being tested not only at the task level, and the adaptability at

that level, but also is adapting to the wider requirements of the constructed pipeline. Indeed, the most efficient execution strategy of a single task taken in isolation may not be the most efficient when taken in the context of other tasks in the pipeline. The requirements of one task may necessitate a certain partitioning strategy, or require a model more suited to a particular strategy, and it may be more performant for other tasks to align with this strategy, rather than choose their standalone optimum strategy, as that may then require recomposition and gathering steps that could be avoided. So one objective of the DFrame is to select the best performing models suitable for a task, but to also adapt that decision according to the context in which a task is running, to for instance include consideration of the partitioning strategy of the models to optimise the computation across multiple distributed tasks in a pipeline. Another prime objective at the task graph level is to automatically adapt resource allocation according to the characteristics of tasks and also through metrics captured during the running of workflows associated with the task graphs.

6.2 Motivation and research background to 3D image capture

In the UK at least a third of the population develops cancer (UK). Formation of secondary tumours through a process known as metastases, is the most dangerous complication and is thought to involve cell growth, detachment, migration, adhesion and invasion. The process of metastasis begins when tumour cells detach from the primary tumour and enter the blood stream. Those cells must then survive in the circulation system and attach to blood vessel walls (vascular endothelium) and eventually migrate through the vessel wall (transmigration) in order to invade new tissue (Liotta 1992). The ability of cells to move (motility) is considered a requirement for the migration of cancer cells to reach the blood circulation system as part of the metastatic cascade (Meyer, Hart 1998). This has lead to efforts to segment cell images to extract and identify principal features which can be linked to having an impact on cancer cell motility (Loo, L. F. Altschuler, S. J. 2007). The morphology and behaviour of cancer cells can be linked to their invasion potential, and the bio-imaging group at Kingston University has recently developed an invasion assay procedure to quantify cell invasion potential under shear stress (Hagglund, Hoppe et al. 2009). This motivates further research to establish other morphometric parameters and behaviour that could be linked to the ability of cancer cells to invade, and related research into cell motility

itself that would help identify new spatio-temporal parameters, which have not be considered in the past. Such endeavours requires the screening of many multi-dimensional images, with a multitude of varied and complex algorithms being applied to each image. It is also of benefit to increase the experimental throughput to improve the statistical significance of analysis results. The use of automated analysis is highly desirable in these circumstances, together with the application of parallel processing in order to avoid bottlenecks in the analysis and be able to conduct these efforts in reasonable timescales. This being a prime motivator of this project.

The 3D image sequences used in this case study were acquired as part of experiments to investigate the effect of shear flow on the adhesion and transmigration of invasive inbred rat sarcoma cells through a confluent monolayer of rat brain endothelial cells RBE4 (Hagglund, Hoppe et al. 2009), the images used in this case study being made available courtesy of the research collaboration between Kingston University and Cancer Research UK. In this case study, further analysis is undertaken to detect the invasion signature of the sarcoma cells, and so investigate the temporal changes to a cell's distribution signal in the invaded dimension, the z-distribution in this case. Figure 6.1 depicts a simple schematic of the salient details of the experimental setup of the above cited research, to provide context for the 3D image sequences used in this case study. Of note is the custom cell chamber design, comprising a MatTek culture dish with a novel flow chamber insert and the image capture arrangement. The experiment captured flow and no-flow (control) images, and this case study presents the results of the invasion signature analysis using the no-flow sequences, the aim being both to determine the efficacy of this image analysis technique, and as prominent, to test the DFrame in a more complex real world application. Of note for this research is that the staining of the sarcoma cells is predominantly punctate with some diffusion, which argues for the use of an edge detecting mechanism within the image processing pipeline, to more effectively detect a cell's invasive signature. It is the tracking of the labelled sarcoma cells that this case study is analysing.

A Nikon TE2000 inverted microscope (63× 1.4 NA) fitted with a scientific cooled CCD camera (Cascade-II, Photometrics) and a motorized stage (MS2000, ASI) was used to record the image sequences, acquiring both red and green

fluorescence images. The system automatically acquired a Z-stack of 101 slices at 0.2 μm intervals using a piezoelectric drive in the motorized stage, images being captured at 15 minute intervals.

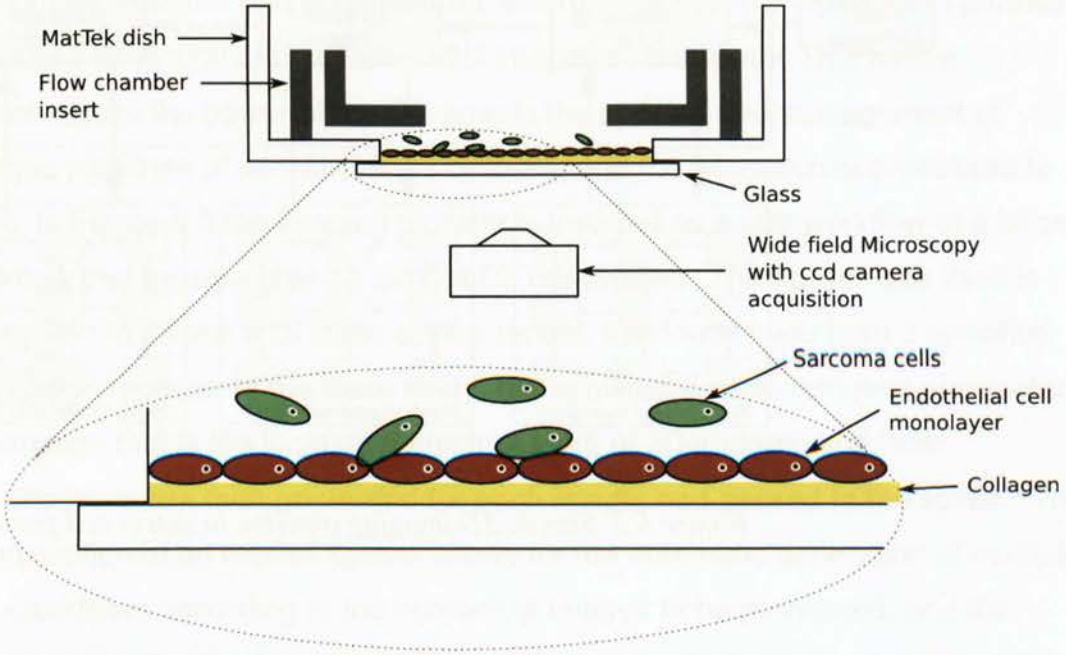


Figure 6.1: Schematic of invasion assay apparatus

6.3 DFrame Pipeline Design

As described, the case study centres on the quantification of invasion signatures of sarcoma cells across a temporal sequence of 3D cell images. An important objective in its own right, this endeavour is also appropriate to demonstrate the DFrame capability when parallelised operators are composed into an imaging pipeline, and when multiple pipelines are operating in parallel to process a collection of 3D images simultaneously. The simple imaging pipeline used in the case study is shown in Figure 6.2 where a 3D image is passed through a number of successive operators. A 3D averaging filter is applied, a common first step to reduce image noise. Then a 3D Sobel filter is applied to produce a 3D gradient image. Finally, a stage involving segmentation using simple thresholding, and the application of a histogram operator is applied to the pre-processed 3D gradient image. The histogram is calculated across one dimension (the z dimension), to allow rudimentary detection of a cell's position in that dimension.

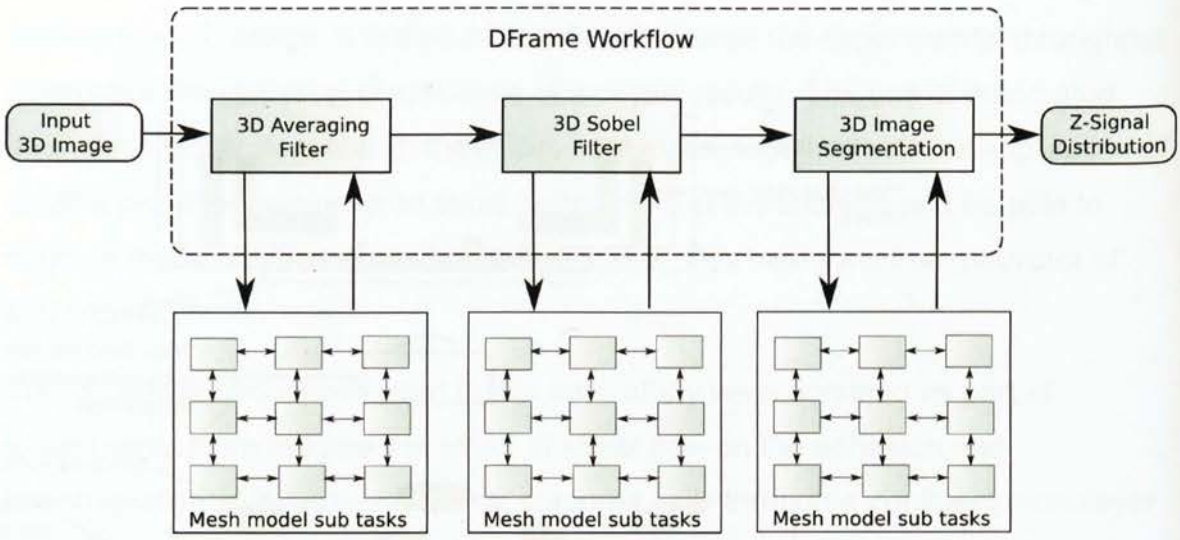


Figure 6.2: Simple 3D imaging pipeline to detect cell position

As per the DFrame design, each image operator task can be arranged to run according to an appropriate parallel processing pattern, implemented in a suitable model. The extent of parallel processing at the task level is broadly controlled by the DFrame, through the number of processes it assigns to a task. A model will then utilise the processes available to it, according to the pattern it implements.

In a pipeline application, individual tasks are no longer executing in isolation (as was the case in the component evaluations in chapter 5) but are now connected such that the output of one task can be the input of a subsequent task. Now the context in which a task is executing becomes important, and this can affect the most suitable model a task should employ. In particular, if the output of the distributed partitions of one task can be directly used in the next task, then it will likely be more efficient to keep the partitions distributed, and avoid a gather and redistribution stage between tasks. This can easily be arranged if a mesh model is now chosen instead of a master worker model. For 3D images, a regular mesh model is suitable for many operators, and can be reused across operators that are processing the same 3D image. If the mesh model is retained in memory, the topology and partitioning need only be calculated and set up once. This means that the overhead in setting up a model across participating processes at the task

level can be amortized across multiple tasks, with a global gather and redistribution stage in between tasks also being replaced with a much more efficient local neighbour exchange.

The efficiency gains that can be achieved within one pipeline is one part of this case study. Another part is concerned with running multiple pipelines in parallel to process a collection of time lapsed 3D images concurrently. This further demonstrates the power of the DFrame in the splitting and management of multiple pipelines of sub-workflows of tasks, and the allocation of processes to each. In Figure 6.3 the imaging pipeline is inserted as a sub-workflow of a larger workflow that incorporates an explicit DFTaskSplitter. The splitter task itself is configured to invoke a DFrame splitter model, that loads code from a specified application module. In this case study, the application code retrieves parametric information that is the location paths to a store of 3D images, and task specifications are then generated for each image, and passed to the splitter. The incorporation of an explicit splitter allows for the automatic generation of multiple sub-workflows according to the number of images to be processed, and the number of available processes, since a DFrame splitter works with an associated DFrame context, such that it is aware of how many processor cores have been made available to it (aligning with the DFrame model design). Also, the splitter works with plugged in application code, that furnishes it with the information on the number of tasks. Application code can also supply characteristics of the tasks, such as the size or expected computation complexity, that can be used by a splitter to determine how to split resources. Size is the basic supported static characteristic that will be used if supplied, in the implemented prototype splitter.

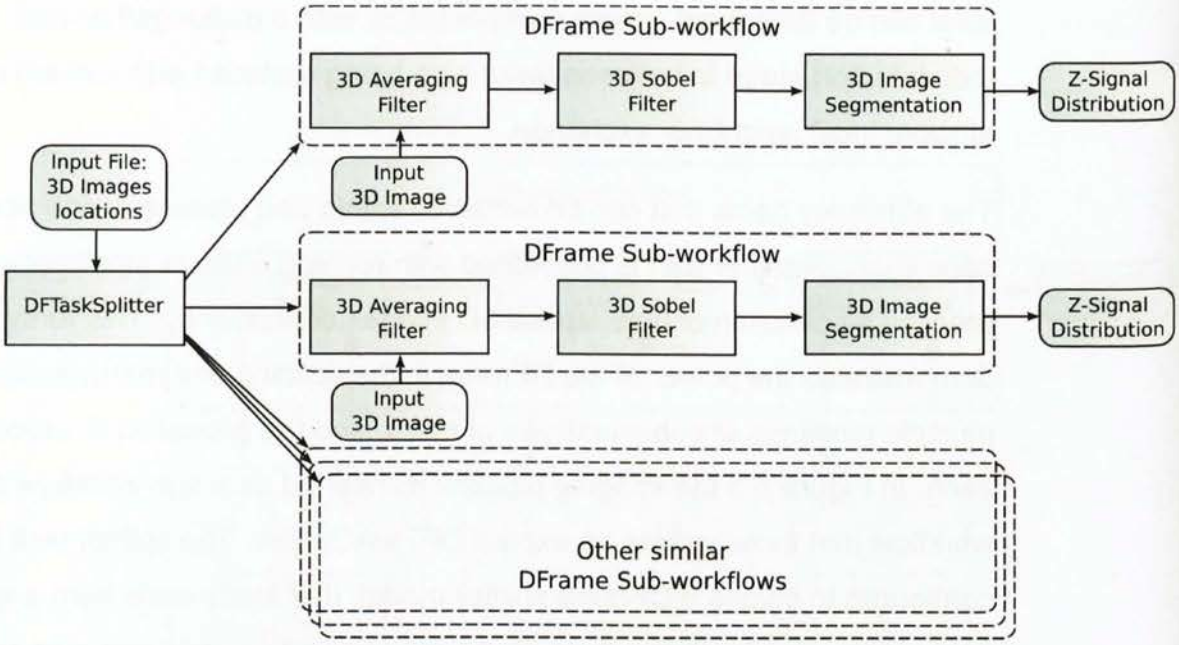


Figure 6.3: Multiple 3D imaging pipelines operating in parallel

In the scenario of multiple running pipelines, each pipeline specification can be distinct and be applied to the same image or to different images, or the pipelines can be similar, and be applied to different images. In this case study, the focus is on applying multiple similar pipelines to a collection of distinct images, so that each input image will have a unique location and name combination. The management of multiple generated pipelines reveals a multitude of further issues that the DFrame design must address. One important aspect is the tracking and management of information that flows through each pipeline. Where a dynamic splitter is used, that is generating multiple pipelines, some means must be made available to propagate task and image meta-information for subsequent tasks to use. For instance, if image outputs are required, these will have to have unique names, and these may be either explicitly specified, or can be generated from input information, which requires parameter passing through a pipeline. In general, labelling of intermediate and final results will be necessary to facilitate eventual aggregation and storing, and this must be carried through a pipeline, such that the outputs to multiple pipelines running in parallel, but operating on related data can retain an ordering to allow correct composition of the results of each pipeline output. This is achieved through a model context, which is propagated through each pipeline of tasks, and indeed can carry information used in merge processes

in the case of more complex task graphs. It is recognised that some form of explicit merge task will be required in more complex cases, but in the current case the objective is to absorb aggregation into contextual composers.

The case study also demonstrates the impetus for the more generic DFrame design in terms of the partitioner and composer interfaces, by incorporating a histogram composition analysis stage that uses a custom composer to gather histogram information across 3D image slices. That composition results will not be limited to 3D Images implies the DFrame must be able to accept more generic composers that can apply custom aggregations to data and this is infused into the design. The algorithms necessary for a specific aggregation then being embedded in the composers that a task supplies to the model, maximising the flexibility and adaptability of the system. The case study uses custom composers to capture and aggregate 3D image data that will form multiple histograms of cell locations within images, and this data is then brought together for inspection, to determine the motility of cells within the time ordered set of images.

6.4 Cell Segmentation and the Histogram Design

The 3D image averaging filters and sobel operator have already been discussed and evaluated in Chapter 5. The segmentation strategy and invasion distribution signal method used in this case study are further described here prior to presenting results. The core objective is to detect invasion signatures amongst multiple cells across a sequence of time lapsed 3D images, with the assumption (prior knowledge) that cell movement is predominantly along one dimension, being a 3D image's z-dimension in this case study. A rudimentary technique for segmenting and detecting such movement is to use a 2D image mask, that broadly identifies the cell locations in the x-y plane. Each cell is identified within the mask by an area marked out using a distinct grey scale value for that cell. This 2D mask is then applied to each x-y image slice in the 3D image to produce output counts for each z-index, for each cell. Figure 6.4 shows a schematic of the basic arrangement. In this schematic, the 2D mask identifies the locations of 3 cells. As the mask is applied to each x-y image slice, the counts for each cell are collected. This results in 3 histograms being generated, one for each cell, that provides information of the distribution of each cell across the z-dimension within the one image. From this information, the location of each cell within the z-dimension can

be determined.

A core prerequisite is that a suitable threshold must be applied, in order to separate each cell in the image from the image background. There are various manual, semi-automatic and automatic techniques for determining a suitable threshold (Sonka, Hlavac et al. 2008), and for this study an adaptive percentage of the image intensity range was identified by applying multiple passes with manually set thresholds across sample images from the image sequences, and used to then automatically calculate percentage thresholds for all 3D images in each image sequence, using each image's intensity range. ImageJ's plot profile feature was also used to assess the distribution of image sections across sample cells to gain further confidence in the chosen threshold values.

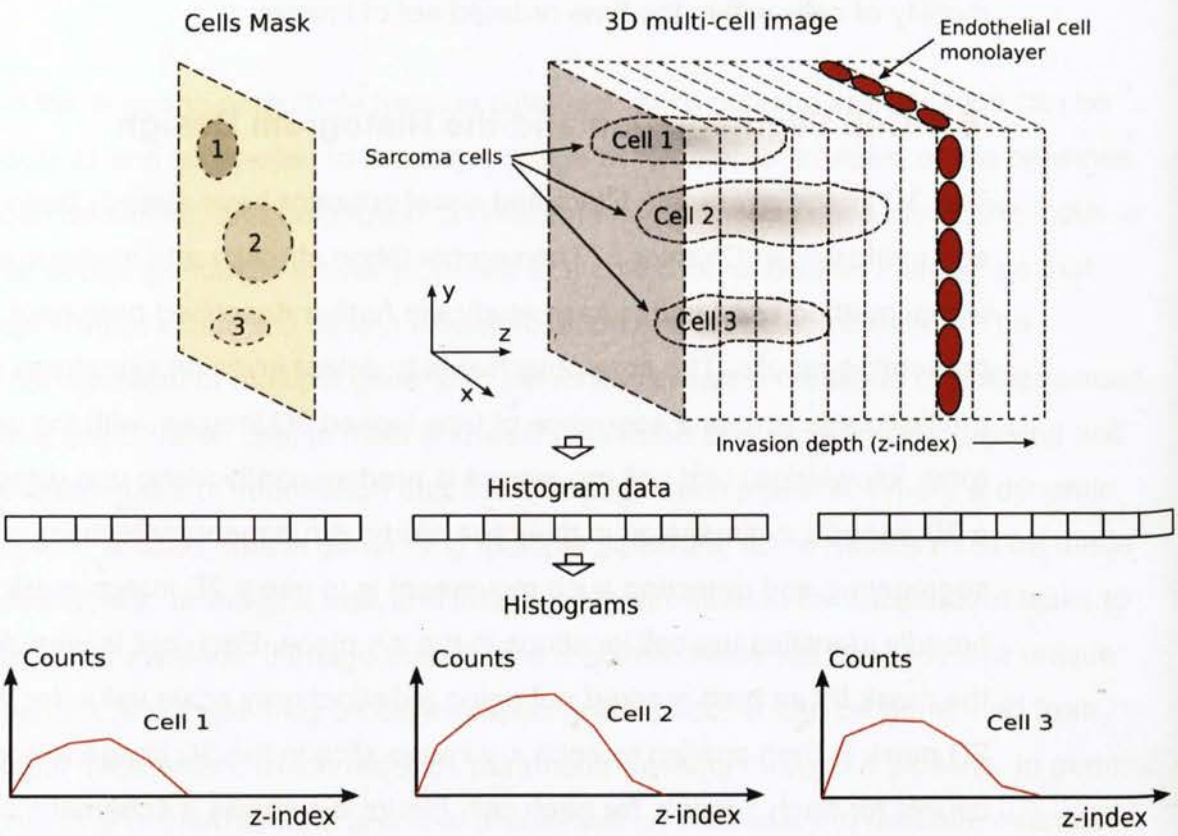


Figure 6.4: Mask applied to a 3D multi-cell image to generate z-dimension histograms

Although the thresholding could be incorporated into a separate thresholding image operator, for this case study, one operator is used to both calculate a threshold, and to scan the 3D image x-y slices with the mask and to use the threshold to determine the extent of each cell's presence in each x-y slice, and collect these counts. As further research expands in this area, the flexibility of a separate thresholding operator would likely prove more worthwhile, especially the development of a module to automatically select a threshold.

The described z-distribution segmentation-histogram approach is a simple thresholding technique even for a 3D image, when one mask is applied to whole image slices across the image. However, the technique becomes somewhat more involved when a 3D image is partitioned. Figure 6.5 shows a schematic where the 3D image is partitioned across the x-y plane, with the mask being likewise partitioned to align.

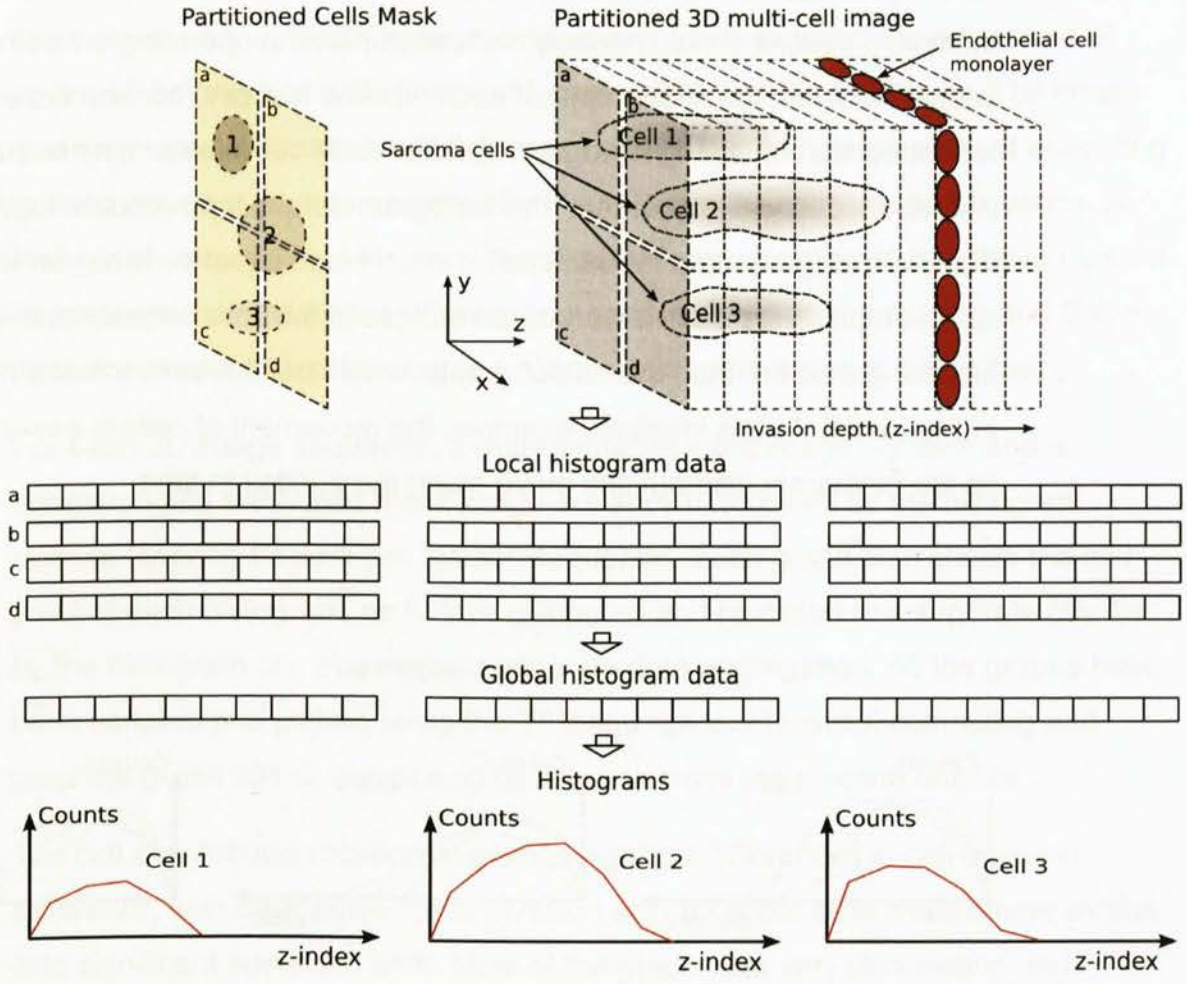


Figure 6.5: Mask applied to a partitioned 3D multi-cell image to generate z-dimension histograms

When the image is partitioned and distributed across multiple processes, the histogram data must be collected locally, and then gathered and recomposed into the corresponding global histogram. Indeed, if the 3D image is partitioned across the x-y plane, then the operator design has to take this into account, and must also partition the mask, such that the appropriate part of the mask is applied to each partitioned sub-image. For partitioned 3D images, there is more book-keeping to be done, to ensure that local histogram data is aggregated into the global histogram data in the correct position. This ancillary work is built into the histogram operator (in DFrame terminology this is the 'module' code), which supplies a bespoke composer that knows how to aggregate the local data, and also a histogram data message adhering to the DFrame message design is implemented to facilitates the transport of local histogram data to the process composing the global histogram, aligning with the mesh model and the DFrame infrastructure design.

As the previous section outlined, the DFrame workflow is constructed to process multiple pipelines simultaneously, with each pipeline operating on a single 3D image. As such, the final output of each pipeline is a csv (comma separated value) formatted file containing histogram data for each cell in each time sequence. To visually inspect cell movement, the histogram data is then brought together such that the histogram data for each cell in each time sequence is rendered on the same graph. Thus if there were three images in the time sequence, and three cells are being studied, then the output graphs would be similar to the examples shown in Figure 6.6. In this idealised example, the movement of cells is easily discernible, as the histogram distributions move along the z-index in time.

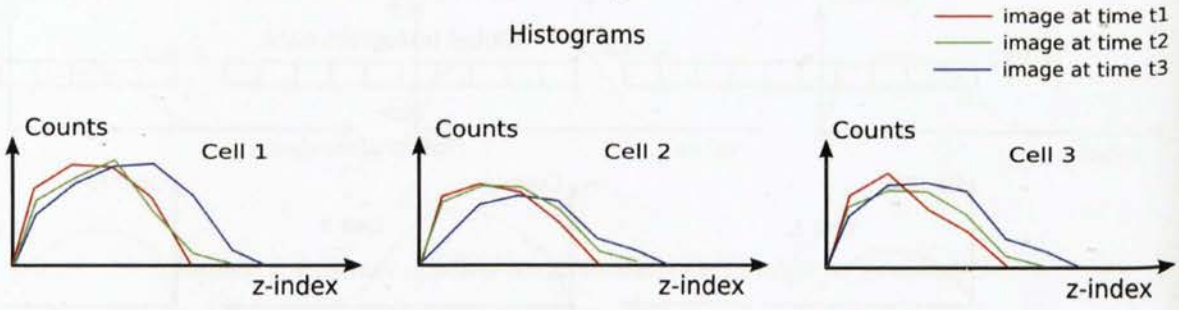


Figure 6.6: Visualization of cell movement through the z-dimension for three cells

Of course, in real experiments the outputs may not be so obliging. Some cells may not move or move somewhat sideways, and their shape can change. Also, signal attenuation across an image sequence adds distortion to the result that may necessitate further processing to compensate. However, this rudimentary histogram technique can be regarded as a useful part of the cell imaging toolkit. The threshold calculation takes into account signal attenuation to some extent, but only on a global level. Instead of adjusting or compensating the histogram outputs further, the histogram operator used in this case study also calculates the mid point of each histogram, and this can be rendered on the resulting graphs to give an indication of the weighted central location of each cell. So the output from each pipeline consists of two csv files, one containing the detailed histogram data, and one containing the mid point z-index and count for each cell in each sequence.

6.5 Cell Invasion Signature Detailed Results

In this section, results are presented for the invasion signature of cells moving in the z dimension, in four 3D image sequences. Each sequence is comprised of seven 16 bit grey scale 3D images of size: x=696, y=520, z=101. An 8 bit image mask is manually created for each image sequence, with areas marked according to the average x-y areas occupied by the cells across each image sequence. A task graph is composed for each image sequence according to the above pipeline description, that includes a DFrame task splitter, 3D image averaging and Sobel operators and the thresholding and histogram segmentation analysis task.

For each 3D image sequence, a representative input image x-y slice and a corresponding annotated mask image are shown, followed by the histogram outputs for each tracked cell in each sequence. Each graph also shows the mid-point of each histogram, as further computed and recorded to a separate csv file by the histogram composer after histogram data aggregation. All the graphs have been scripted and plotted using the 'R' language for statistical computing and graphics (Knell 2013), composing data from across the pipeline outputs.

The cell z-distribution histogram graphs highlight differences in cell invasion behaviour, with clear evidence of invasion shift for some cells while others exhibit little significant sustained shift. Most of the graphs are very illuminating and demonstrate the utility of the approach. For instance, in sequence 1, cells 73, 135, 164, 182, 205 and 234 provide reasonable information on cell invasion signatures,

with the histograms all having acceptable signals. Interestingly, the results for cell 43 shows an unexpected reduced signal for the first three images in the sequence and on further investigation this is determined as due to poor registration of the image mask for this cell in these images, due to lateral movement of the cell. Cell 108 results also show some unusual variability, this time at the end of the sequence, with the reducing signal strength to some extent impacting the reliability of the thresholding. In sequence 2, the histograms for cells 76, and 136 show good invasion shift behaviour. The result for cell 187 prompts further scrutiny, and on closer inspection, it is determined from the image slice and mask images that the registration of cell 187 to the mask could be improved. As well, the signal for time step 1 (t1) for this cell is very low, and inspecting the relevant image, it appears that this cell was likely still floating at the start of the sequence, before settling down and so was not at this time fully within the z-acquisition capture range of the microscope. In sequence 3, histograms for cells 44, 64, 79, 80, 105 and 151 all provide reasonable results across all the images except the last image 7, where the attenuation of the signal is quite pronounced. The results for cells 177 and 211 appear less reliable, and again it is observed that the registration of the masks for these cells are sub-optimal. Also, cells 136 and 246 exhibit low signal at the start as well as the end of the sequence, in the case of cell 136 this is explained by poor mask registration across the initial images in the sequence, and for cell 246 it appears that the cell doesn't land within the acquisition range until later in the sequence. In sequence 4, the results again show some variable, with the histograms for cells 136, 163, 188 and 230 showing reasonable invasion signatures. Cell 58 shows marked signal attenuation affecting results at the end of the sequence, and on closer inspection, the mask for cell 101 is seen to actually straddles two cells, contributing to the skewed histogram results in this case.

In summary, these tests both support the utility of the method and also highlight its limitations (as anticipated). The histograms can reveal information on the registration of the cell mask across an image sequence, with a low signal across all images suggesting that the registration be reviewed, or the signal to threshold be reviewed or indeed the signal itself. However, there are limits to the interpretation as when one cell moves out from a mask and another moves in, the results may still seem adequate. So there are caveats to the utility, and the technique should be used with its limitations born in mind. In particular, mask registration errors and lateral movement of the cells can adversely affect the

results, and this is apparent on detailed image inspection across some of the image cell sequences. The temporal signal attenuation across each image sequence is also an added variable that must be taken into account and indeed confirmed that the signal for each cell is adequate across a sequence. This makes the task of determining an optimum threshold more involved as it has to be dynamically established accounting for the signal degradation (as mentioned, a simple signal percentage is employed in these tests). Nevertheless, the technique is simple and fast, and while rudimentary, it can provide quick and useful direct insight into cell behaviour, and also identify those cells that might usefully be further investigated.

6.5.1 Sequence 1 Invasion Signatures

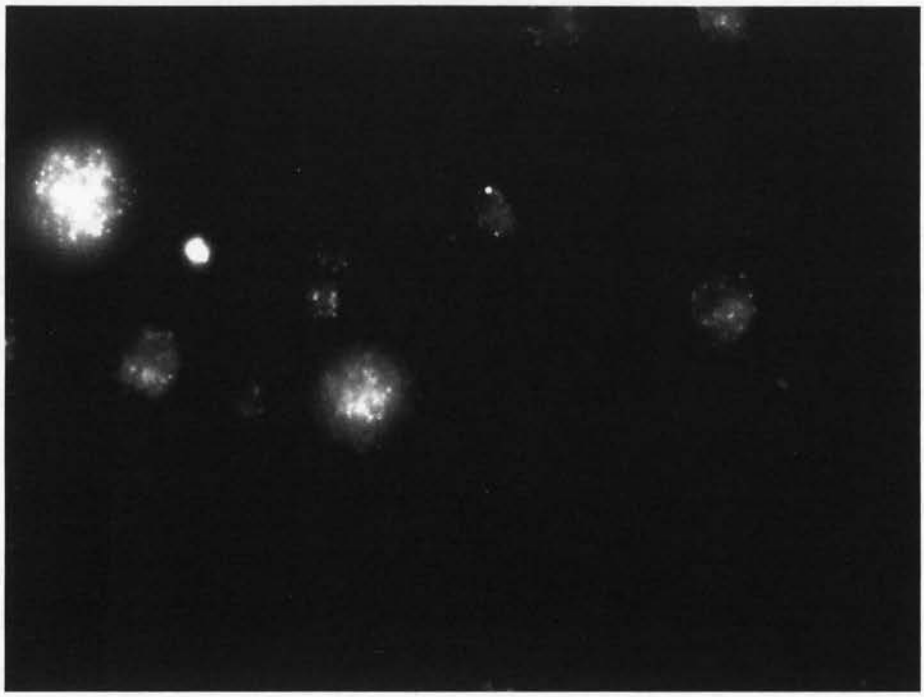


Figure 6.7: 3D Image x-y slice from the first image of sequence 1

image = 696x520x101, 16 bit, 70MB, image slice = 10/101

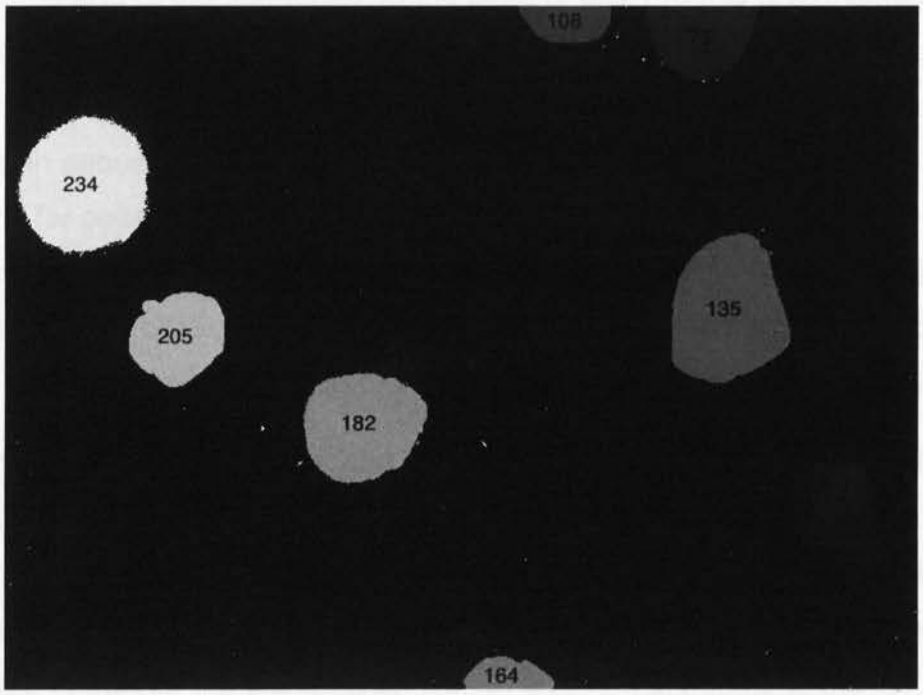
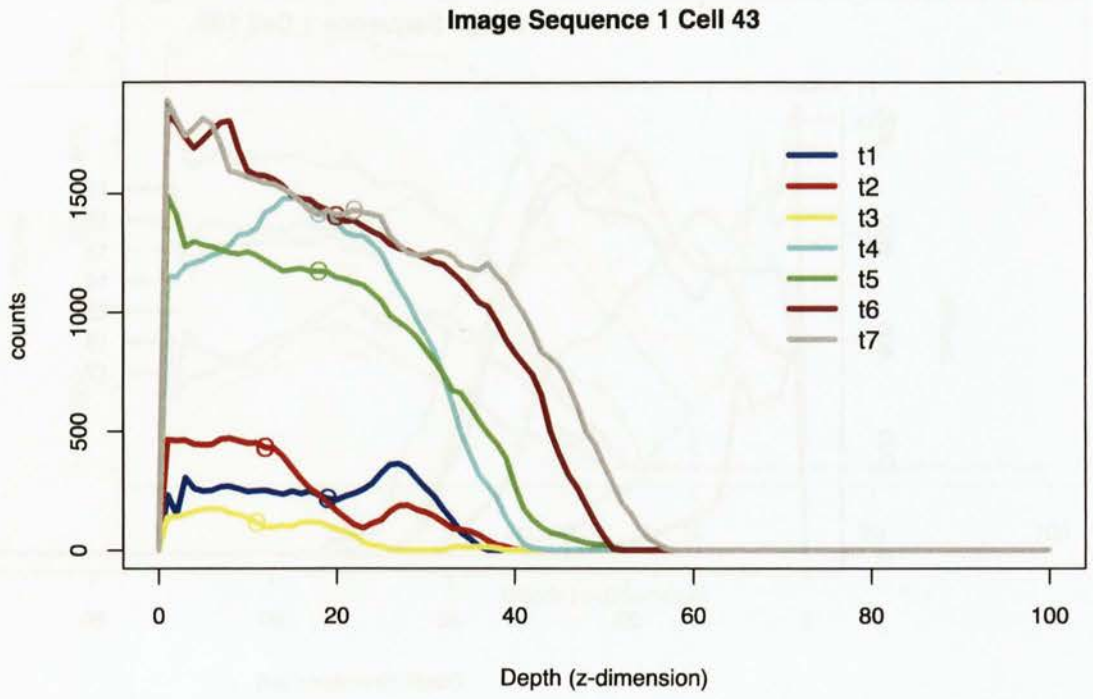


Figure 6.8: 2D x-y mask for sequence 1

(the annotated numbers are grey scales that also identify each cell being tracked)



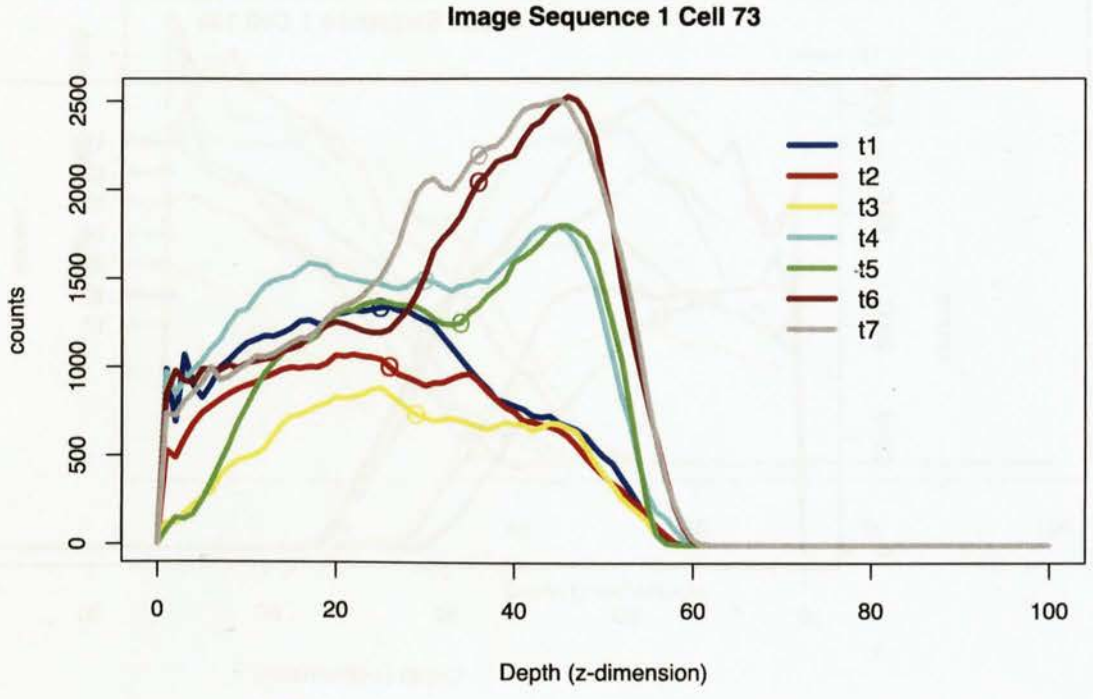
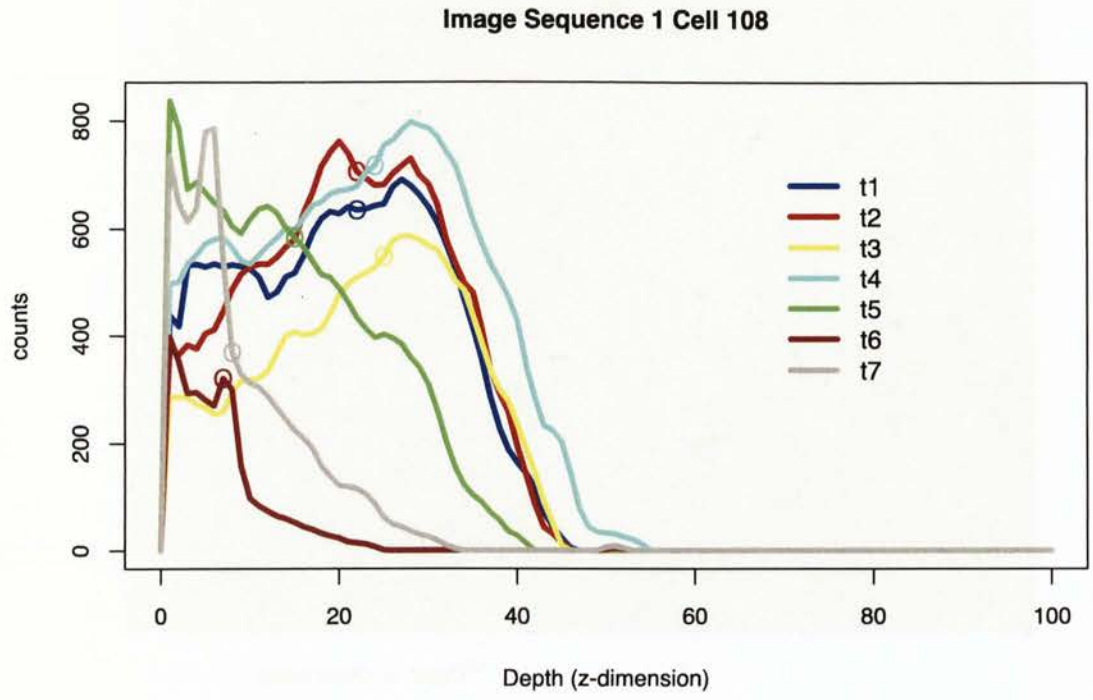


Figure 6.9: Sequence 1: Cell invasion plots for Cell 43 and Cell 73



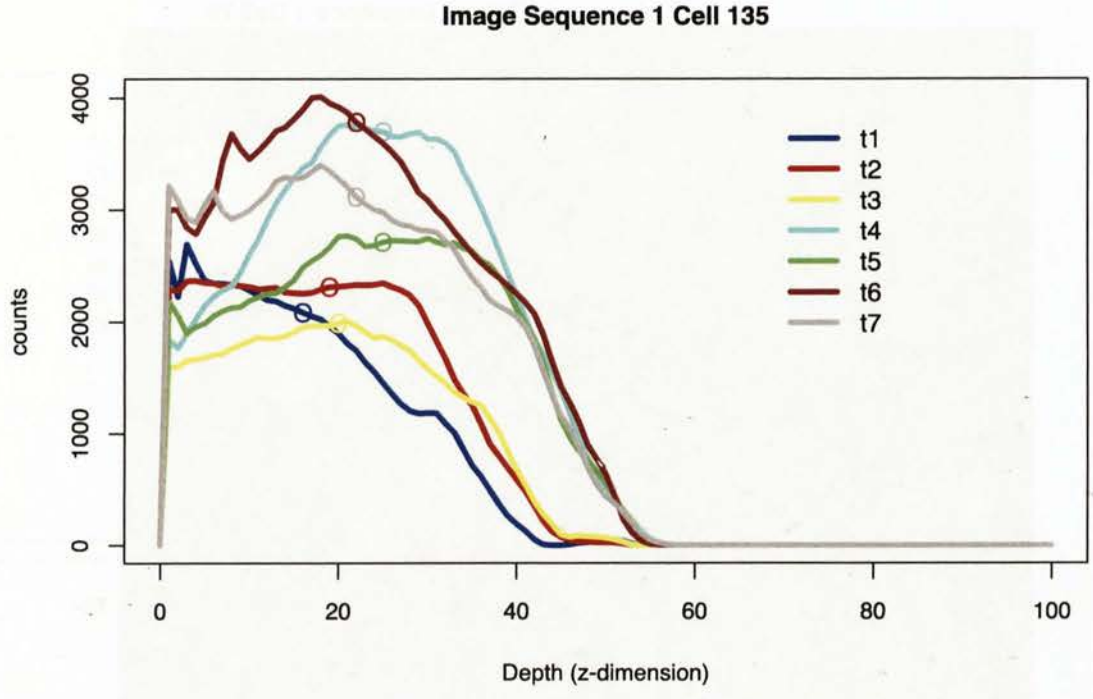
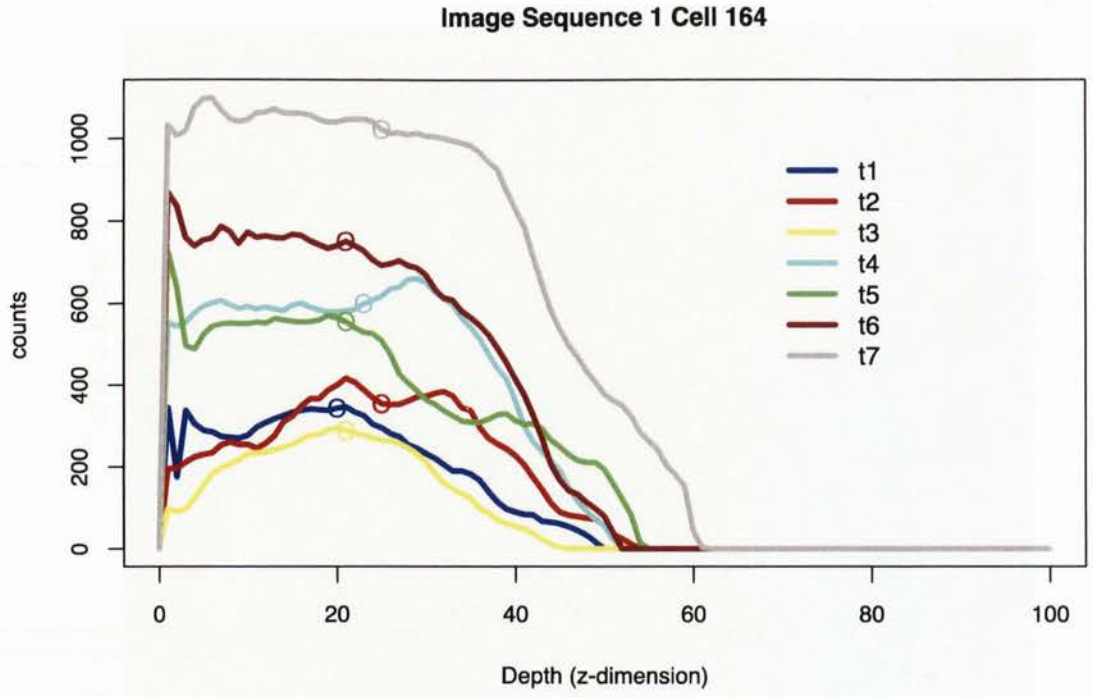


Figure 6.10: Sequence 1: Cell invasion plots for Cell 108 and Cell 135



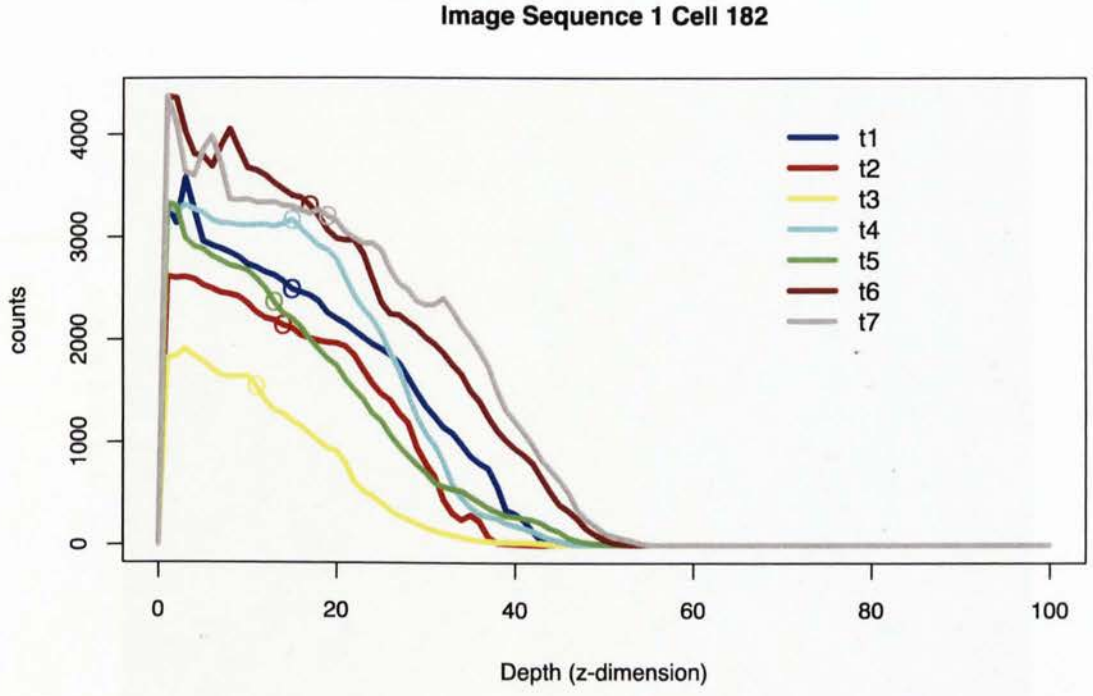
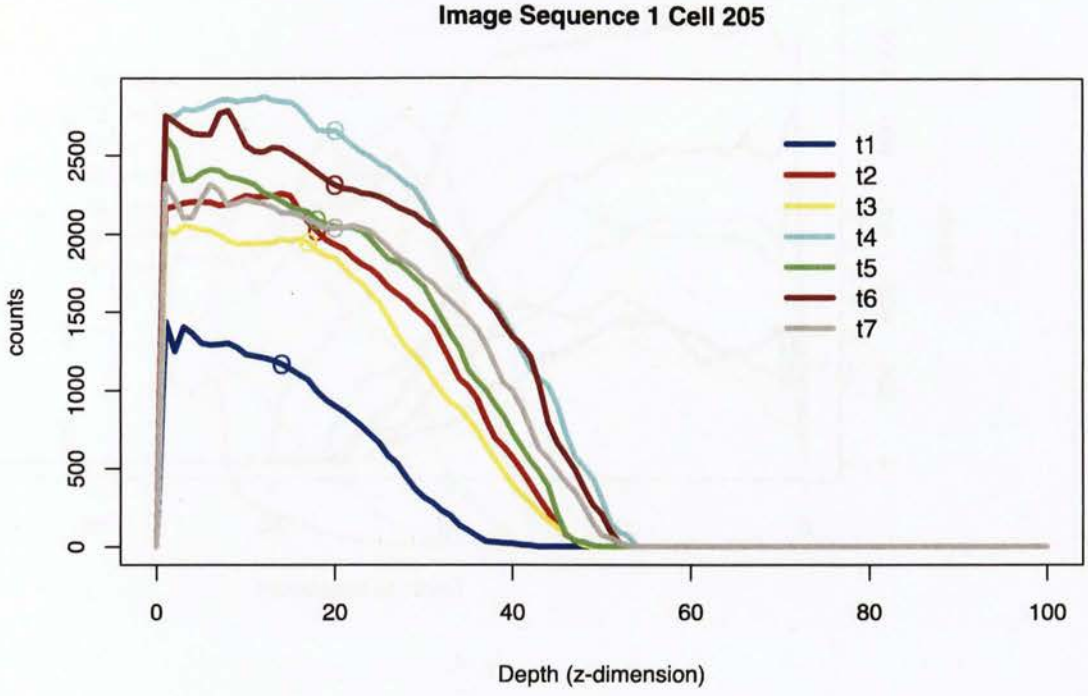


Figure 6.11: Sequence 1: Cell invasion plots for Cell 164 and Cell 182



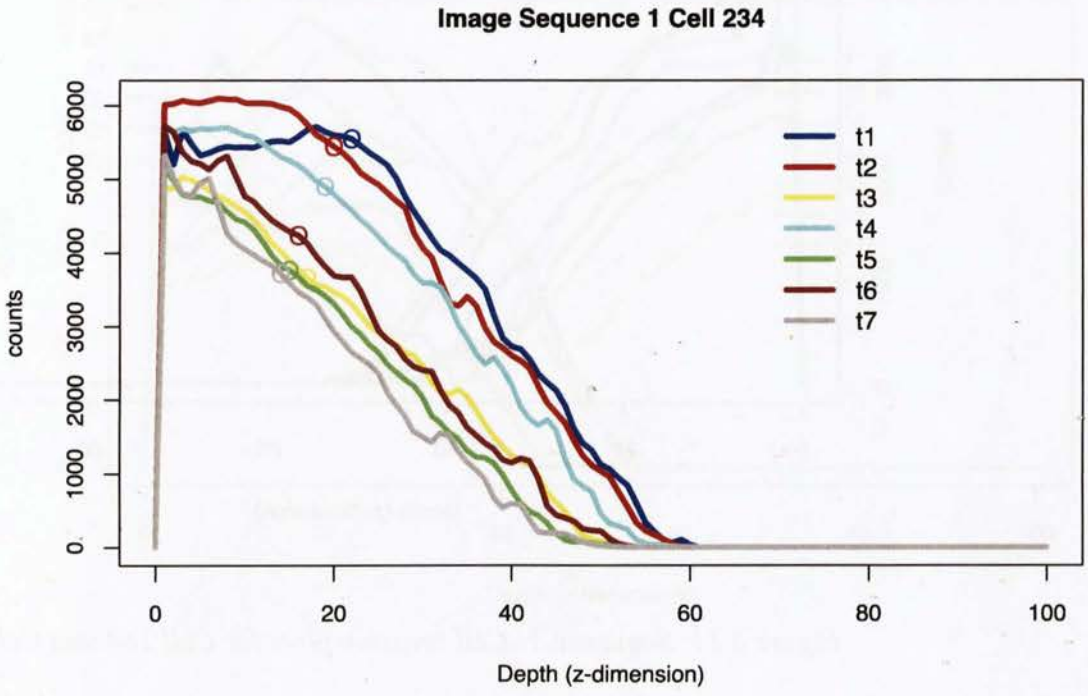


Figure 6.12: Sequence 1: Cell invasion plots for Cell 205 and Cell 234

6.5.2 Sequence 2 Invasion Signatures

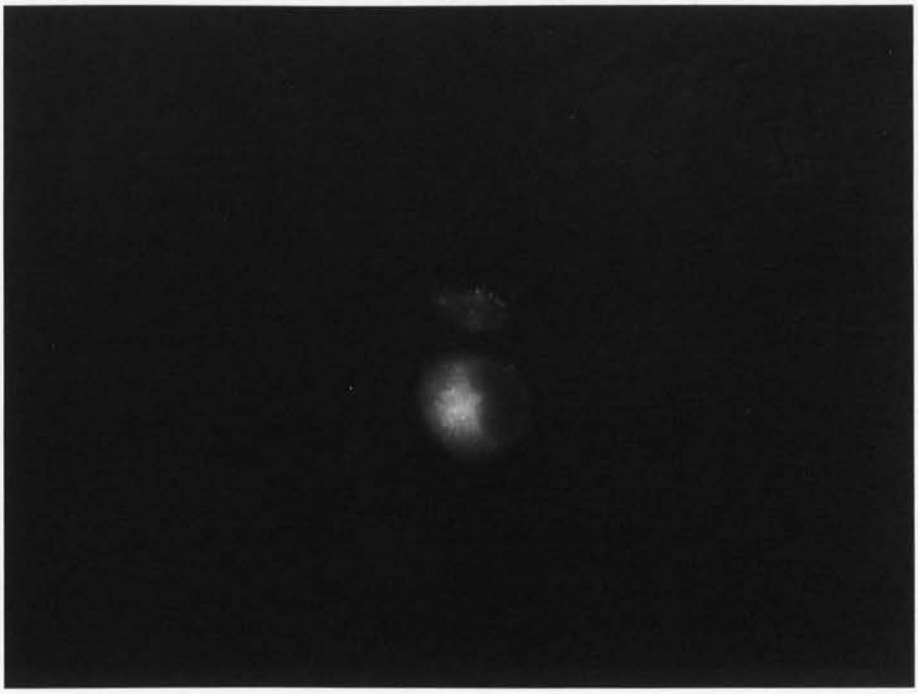


Figure 6.13: 3D Image x-y slice from the first image of sequence 2

image = 696x520x101, 16 bit, 70MB, image slice = 10/101

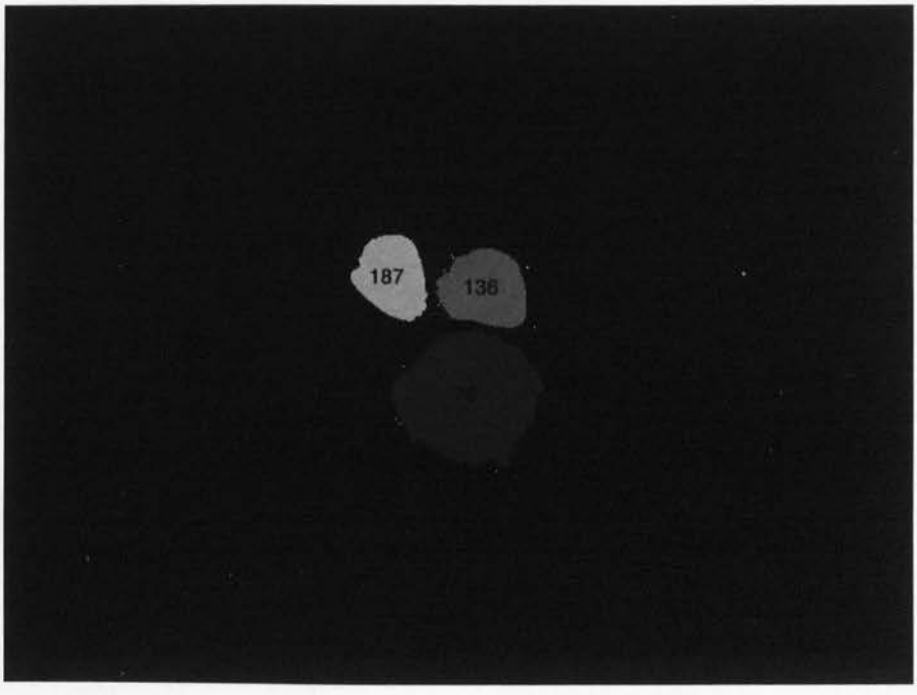
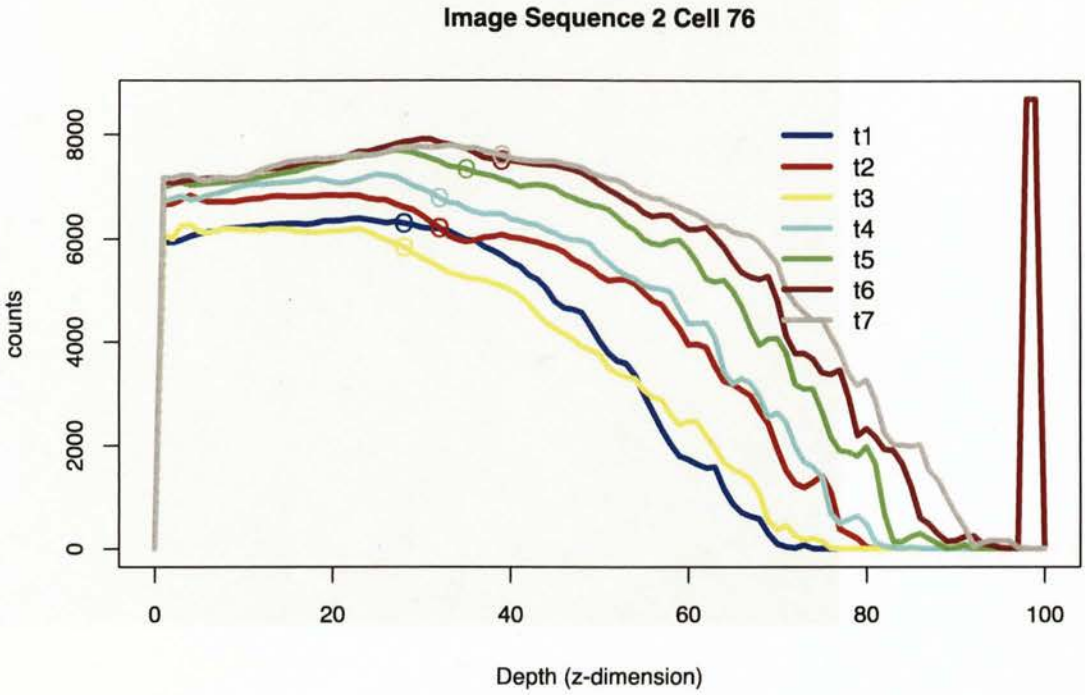


Figure 6.14: 2D x-y mask for sequence 2

(the annotated numbers are grey scales that also identify each cell being tracked)



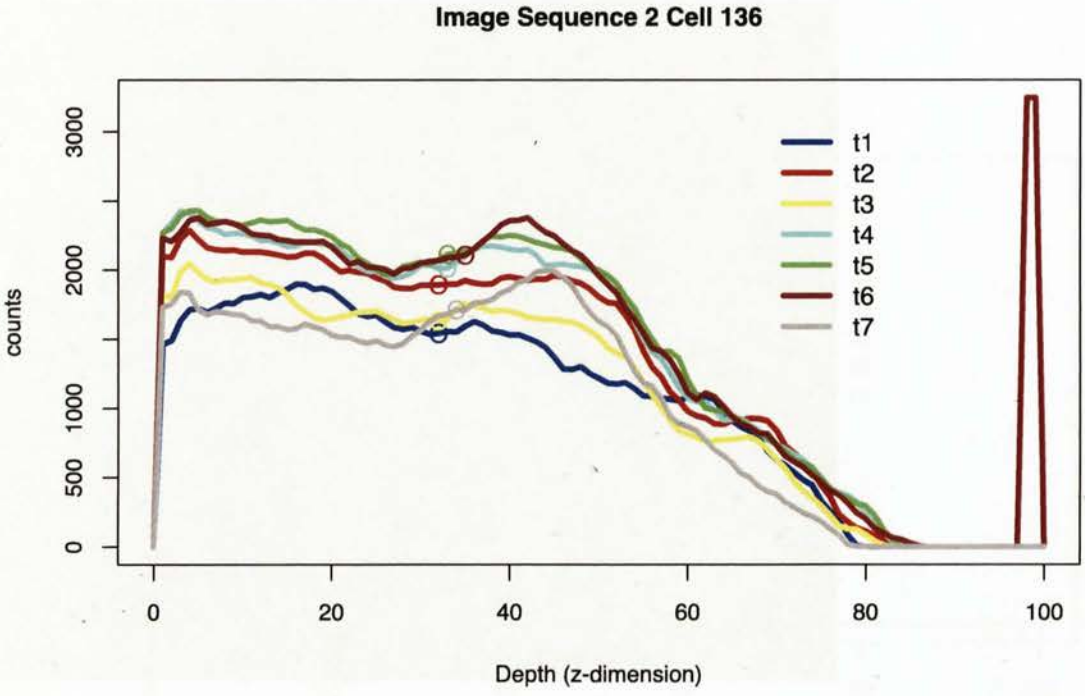


Figure 6.15: Sequence 2: Cell invasion plots for Cell 76 and Cell 136

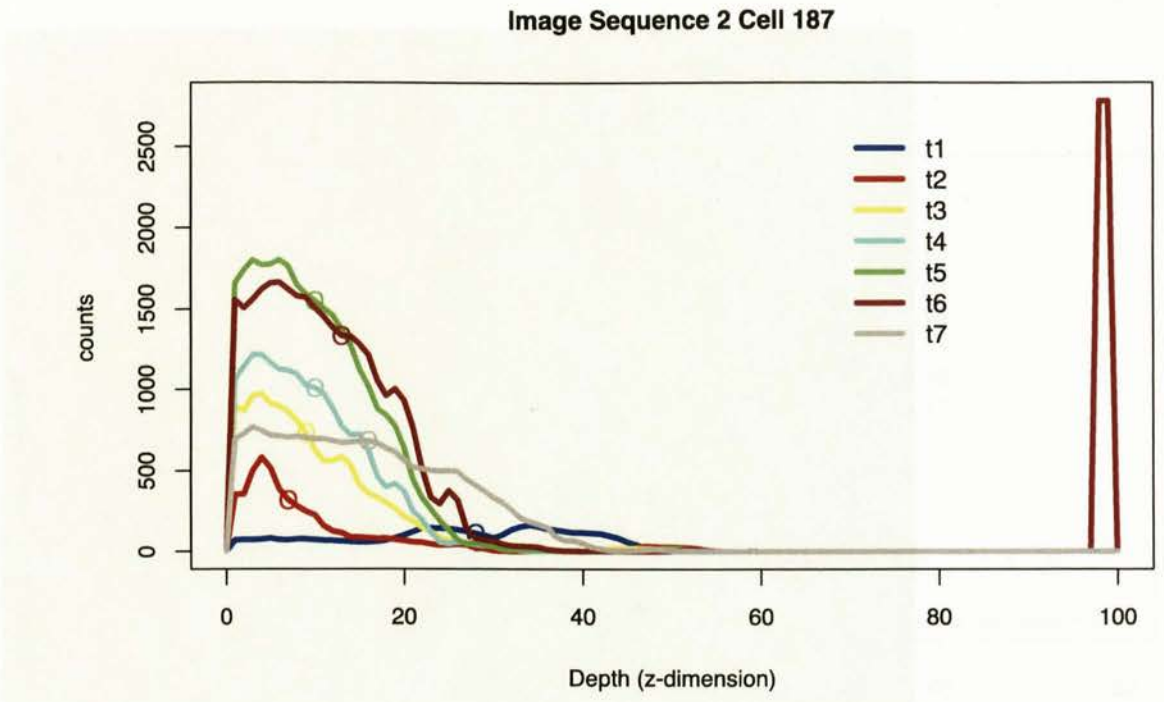


Figure 6.16: Sequence 2: Cell invasion plots for Cell 187

6.5.3 Sequence 3 Invasion Signatures

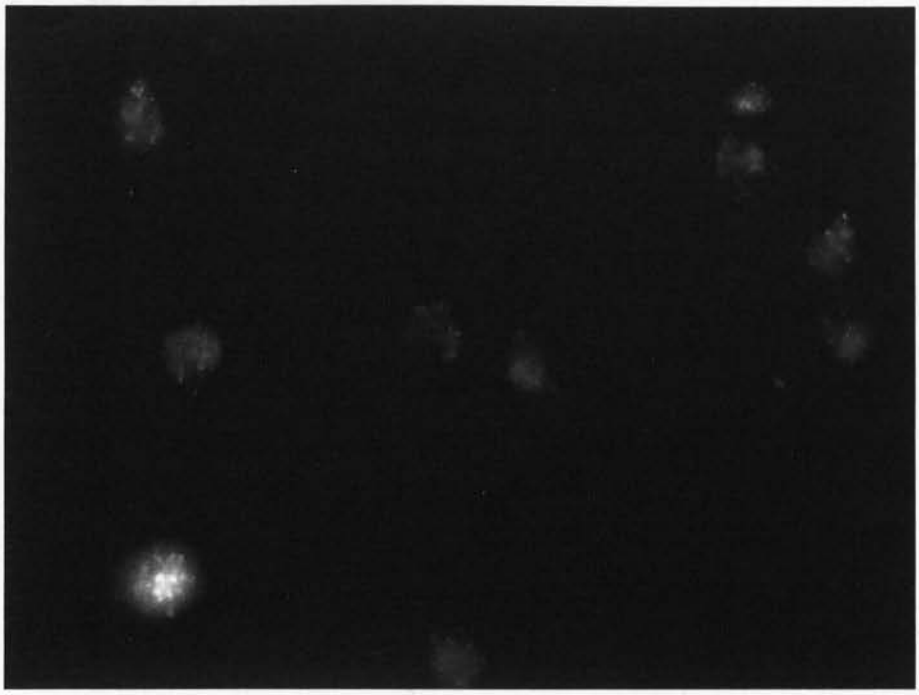


Figure 6.17: 3D Image x-y slice from the first image of sequence 3

image = 696x520x101, 16 bit, 70MB, image slice = 10/101

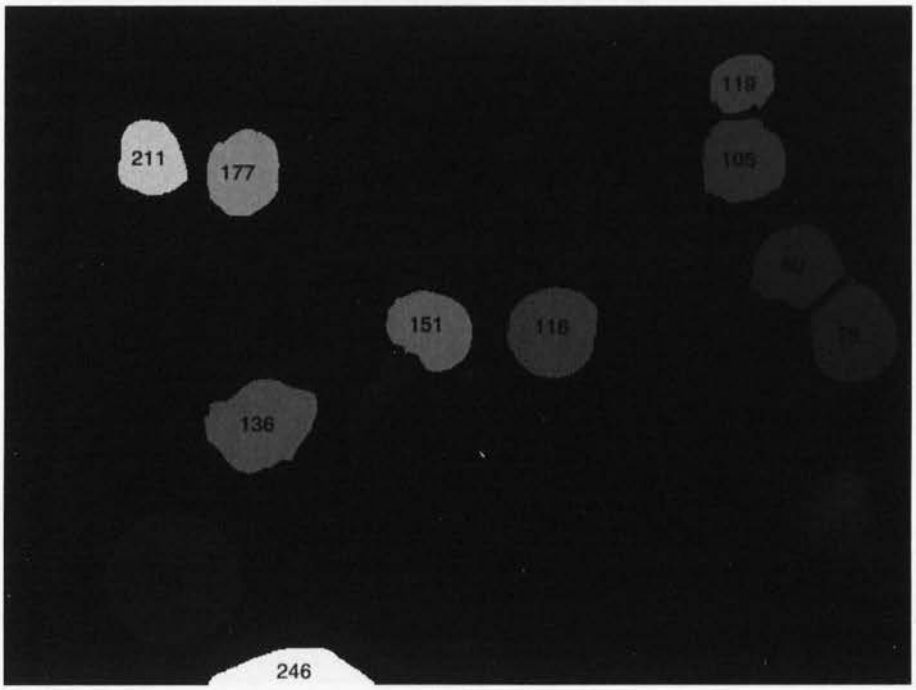
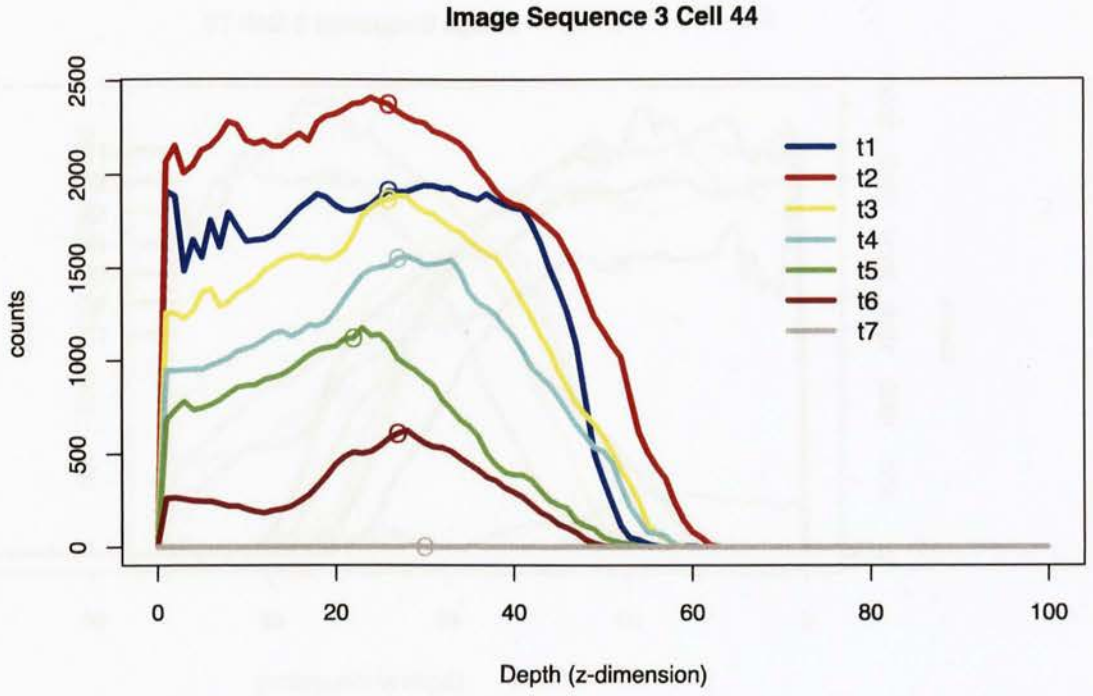


Figure 6.18: 2D x-y mask for sequence 3

(the annotated numbers are grey scales that also identify each cell being tracked)



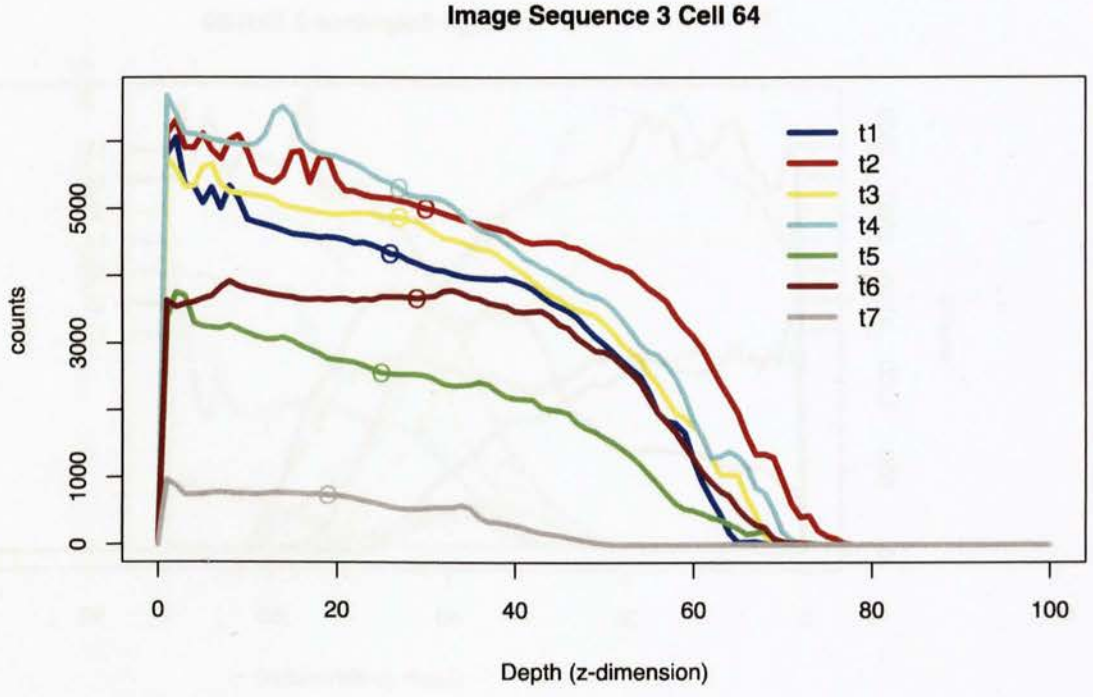
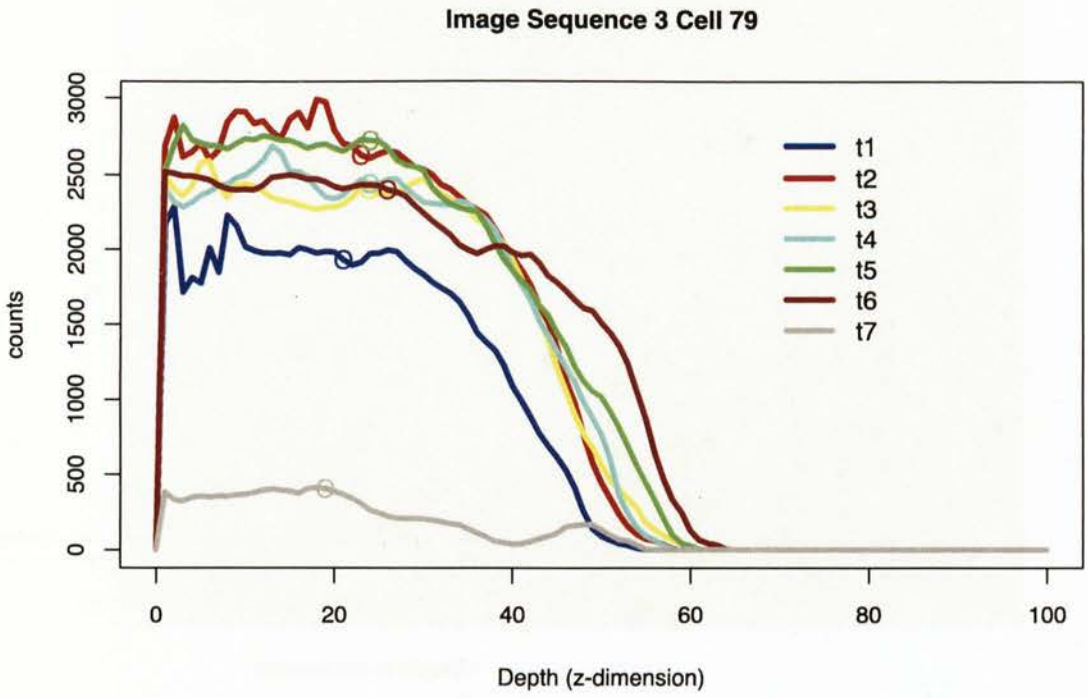


Figure 6.19: Sequence 3: Cell invasion plots for Cell 44 and Cell 64



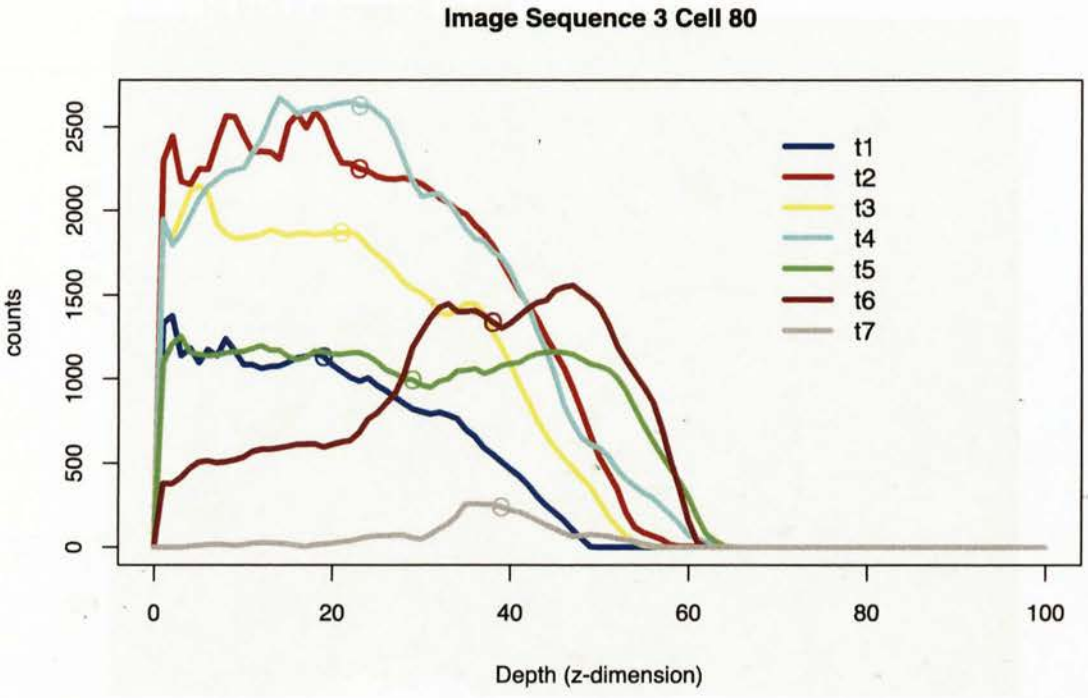
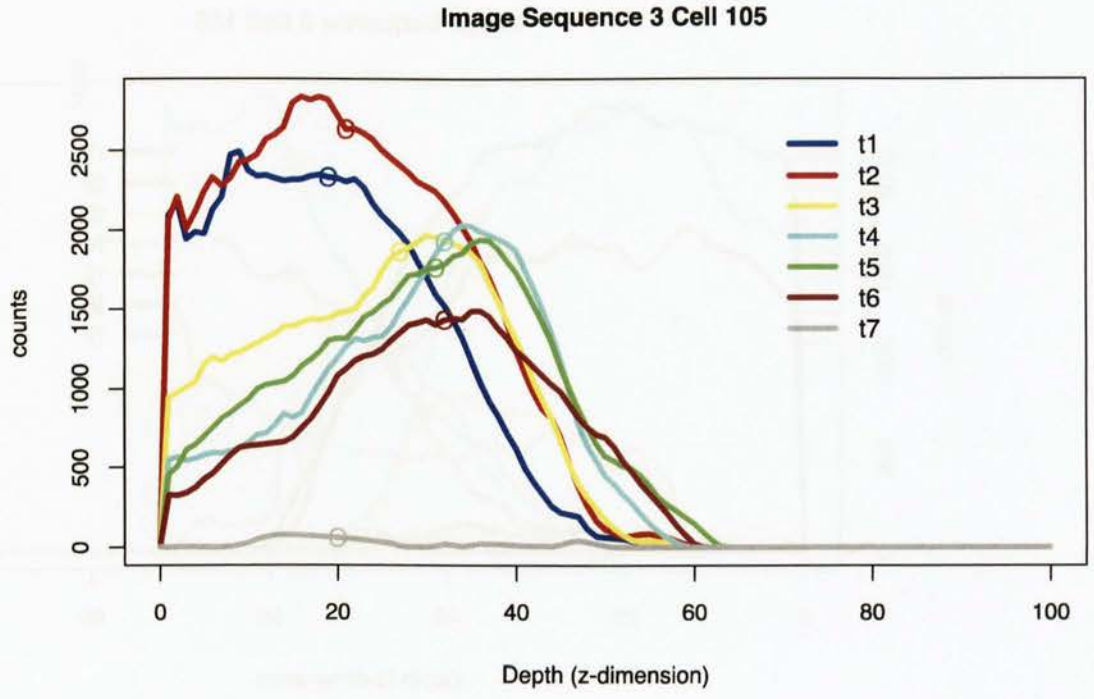


Figure 6.20: Sequence 3: Cell invasion plots for Cell 79 and Cell 80



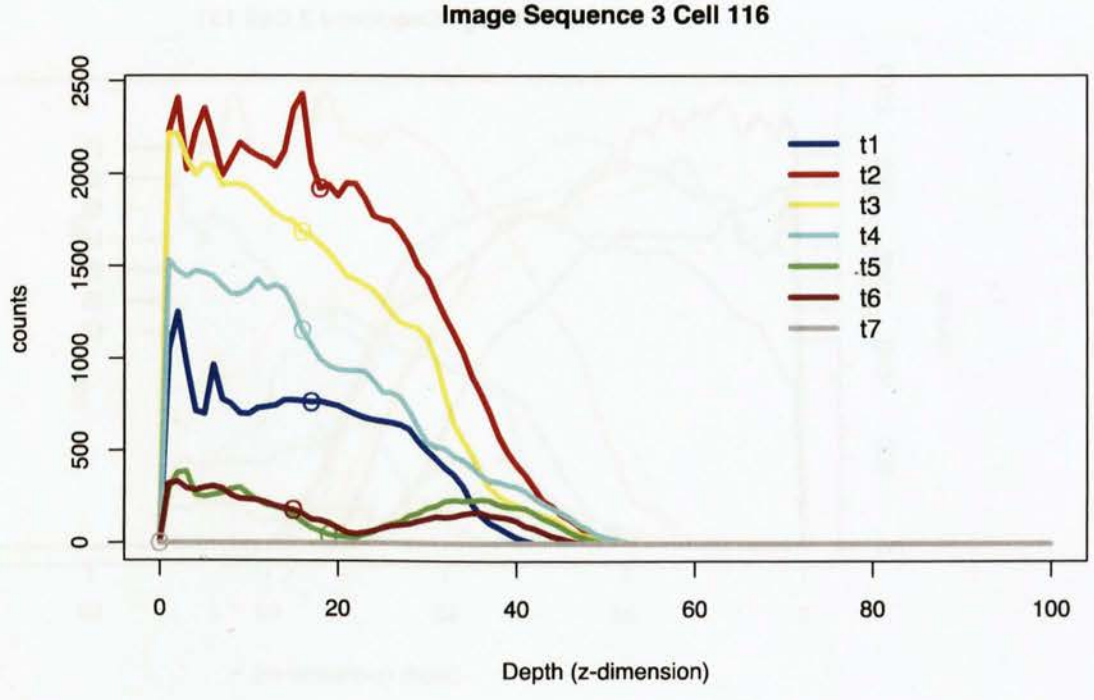
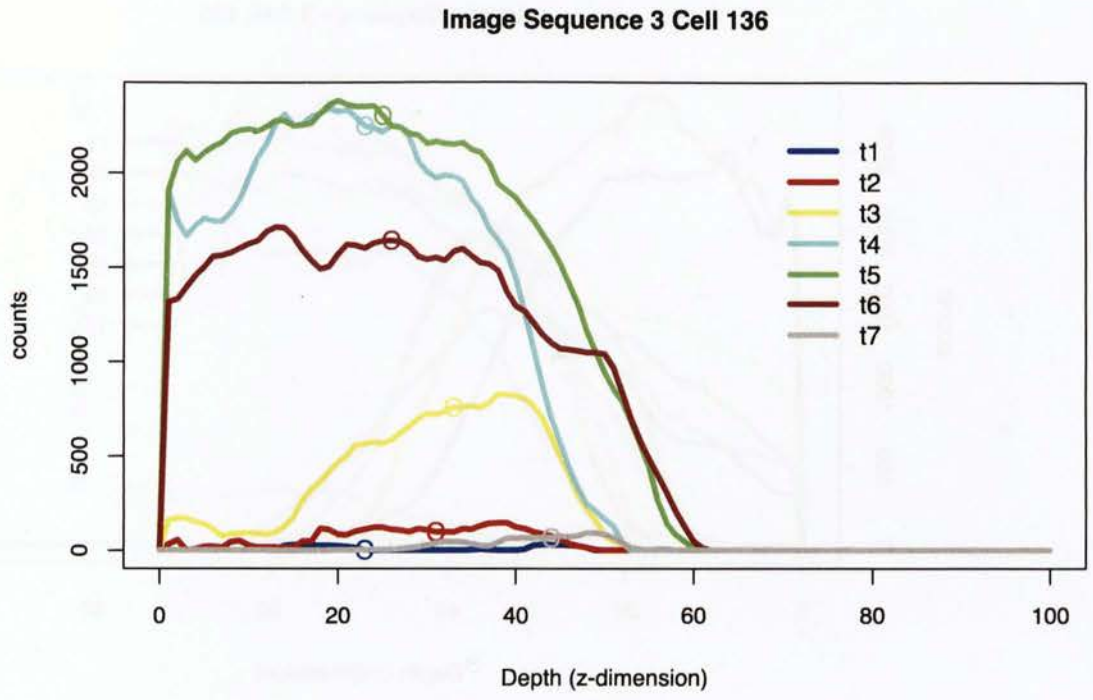


Figure 6.21: Sequence 3: Cell invasion plots for Cell 105 and Cell 116



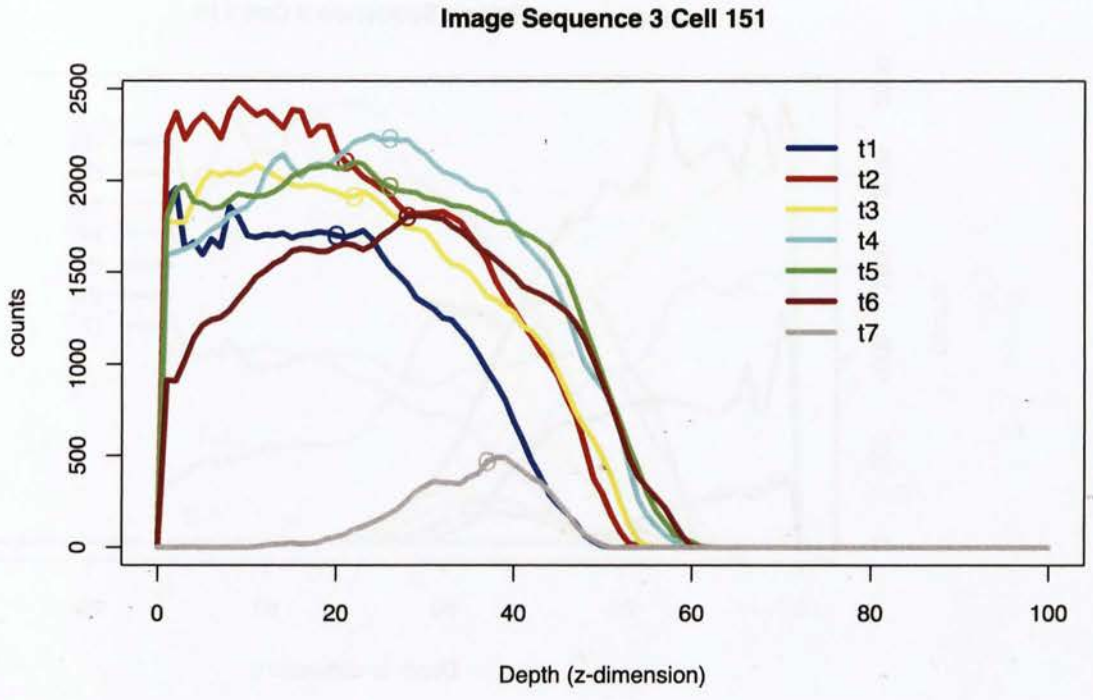
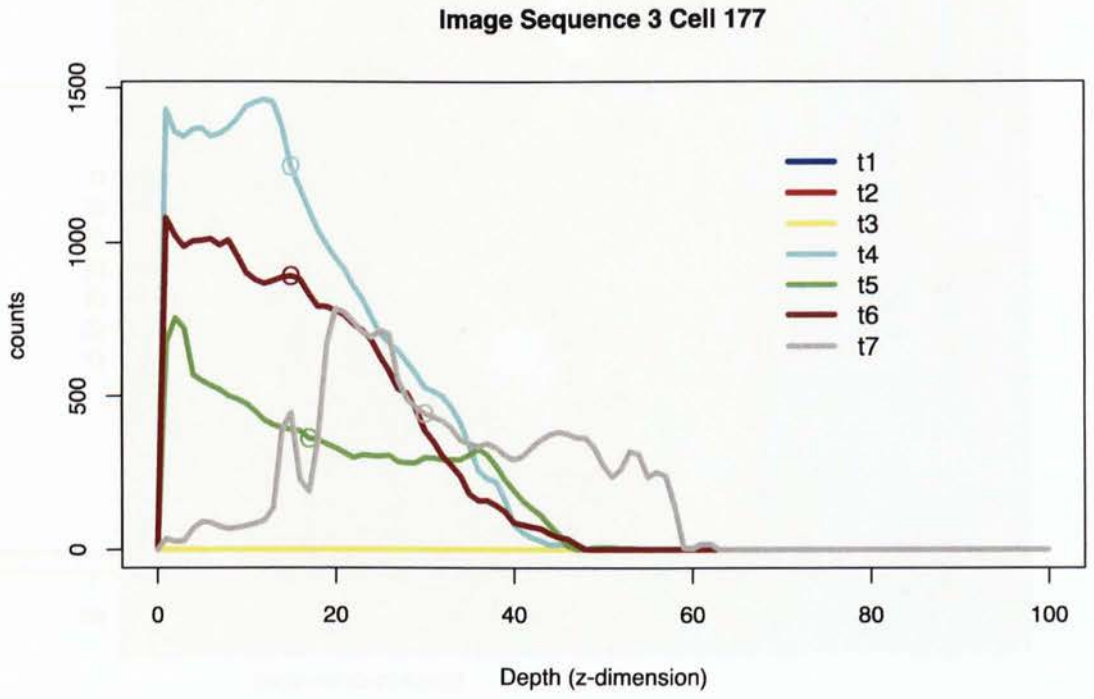


Figure 6.22: Sequence 3: Cell invasion plots for Cell 136 and Cell 151



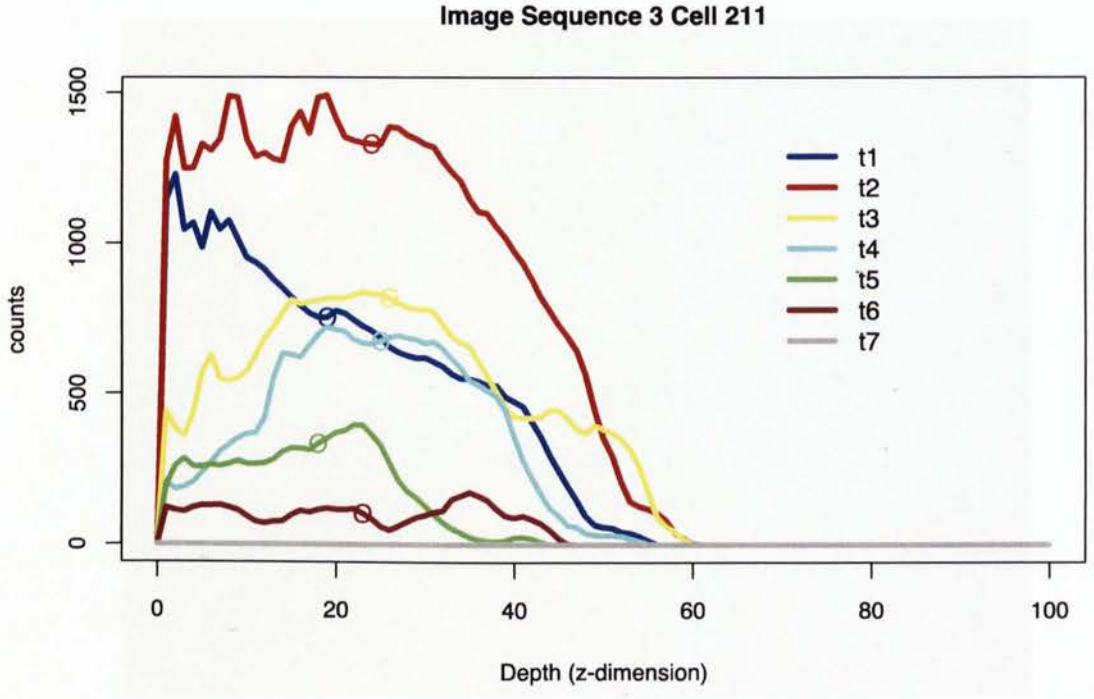


Figure 6.23: Sequence 3: Cell invasion plots for Cell 177 and Cell 211

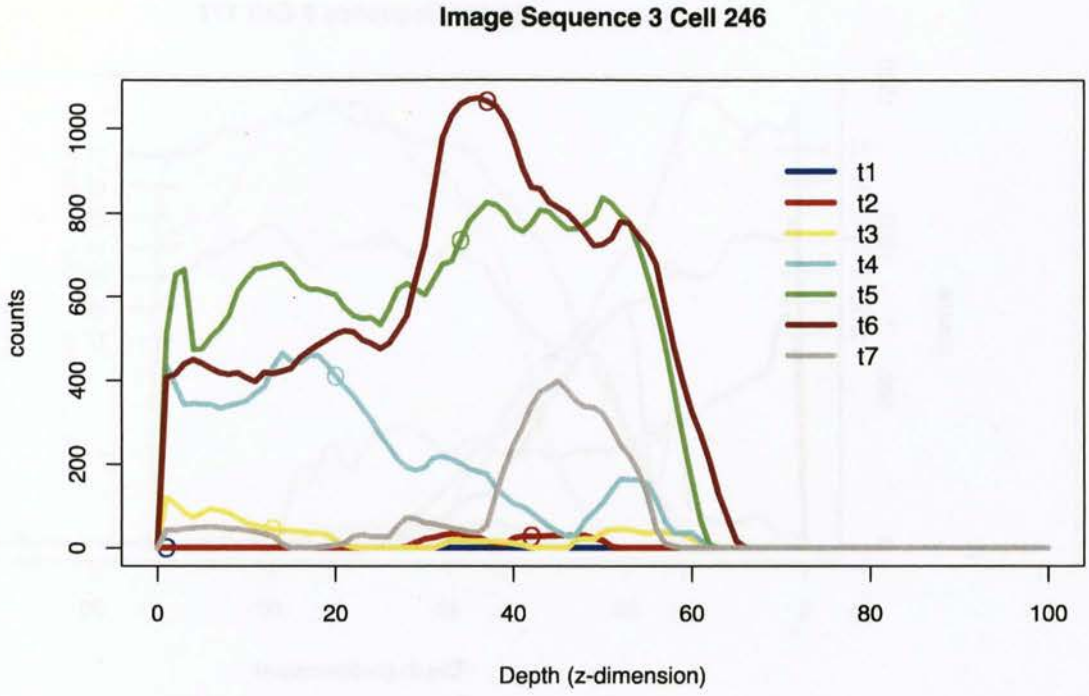


Figure 6.24: Sequence 3: Cell invasion plots for Cell 246

6.5.4 Sequence 4 Invasion Signatures

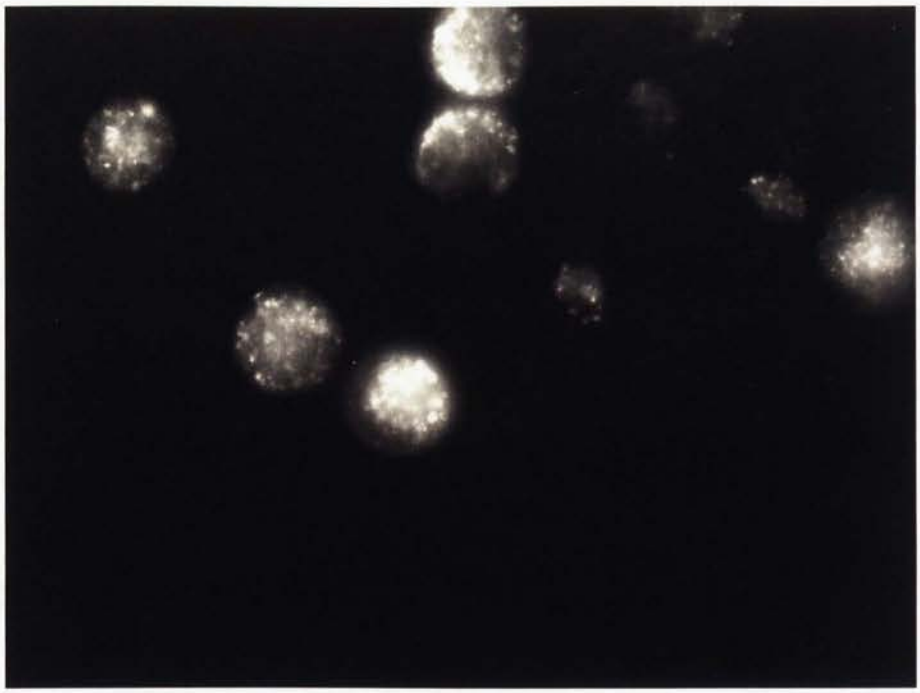


Figure 6.25: 3D Image x-y slice from the first image of sequence 4

image = 696x520x101, 16 bit, 70MB, image slice = 10/101

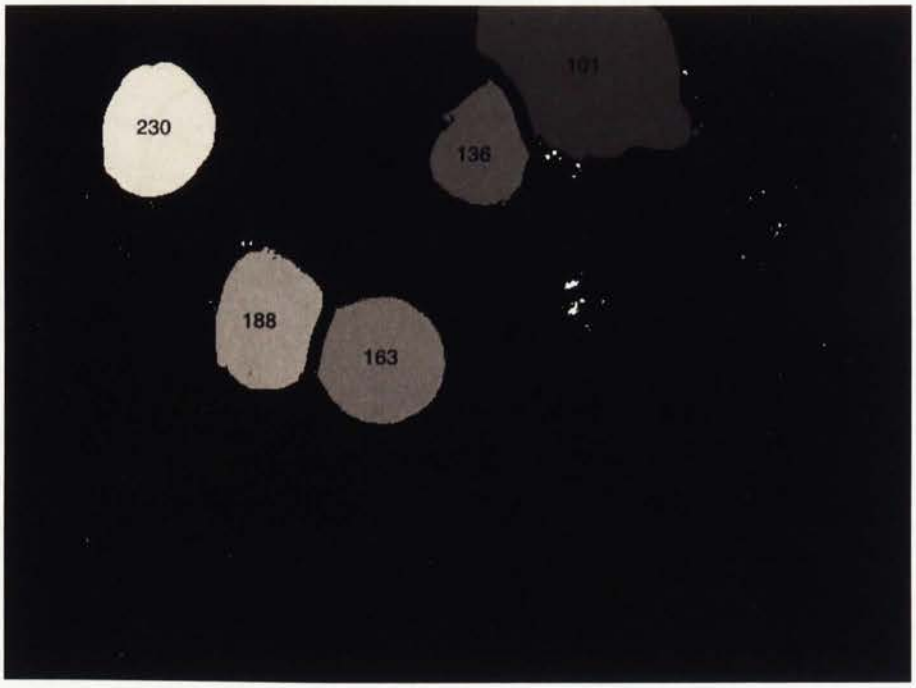
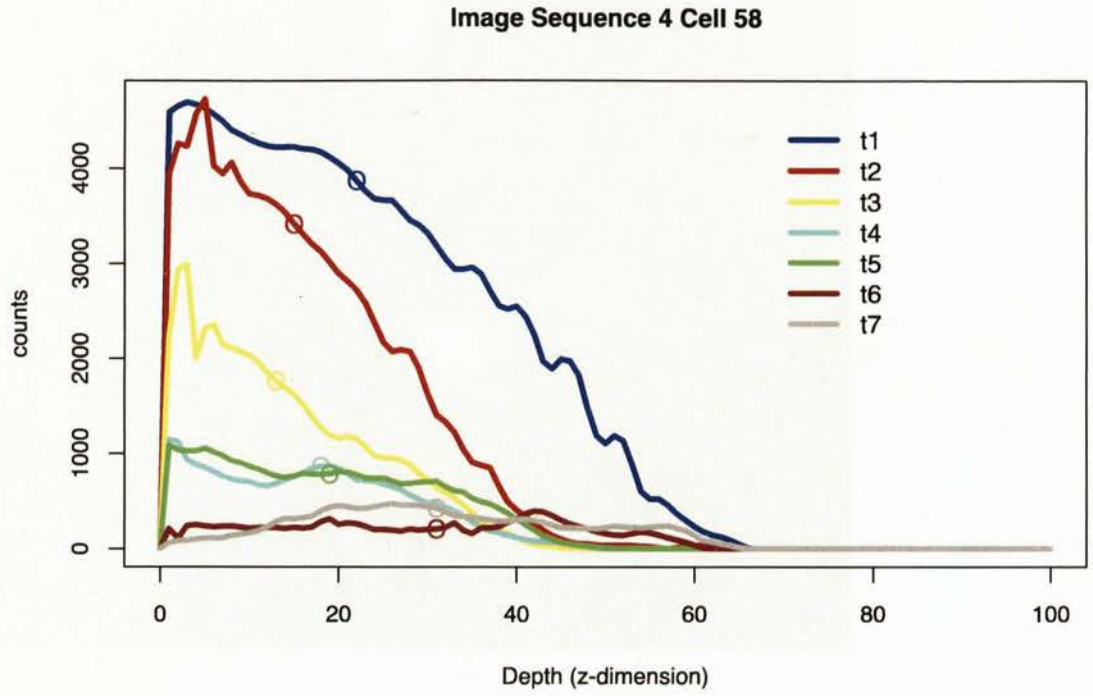


Figure 6.26: 2D x-y mask for sequence 4

(the annotated numbers are grey scales that also identify each cell being tracked)



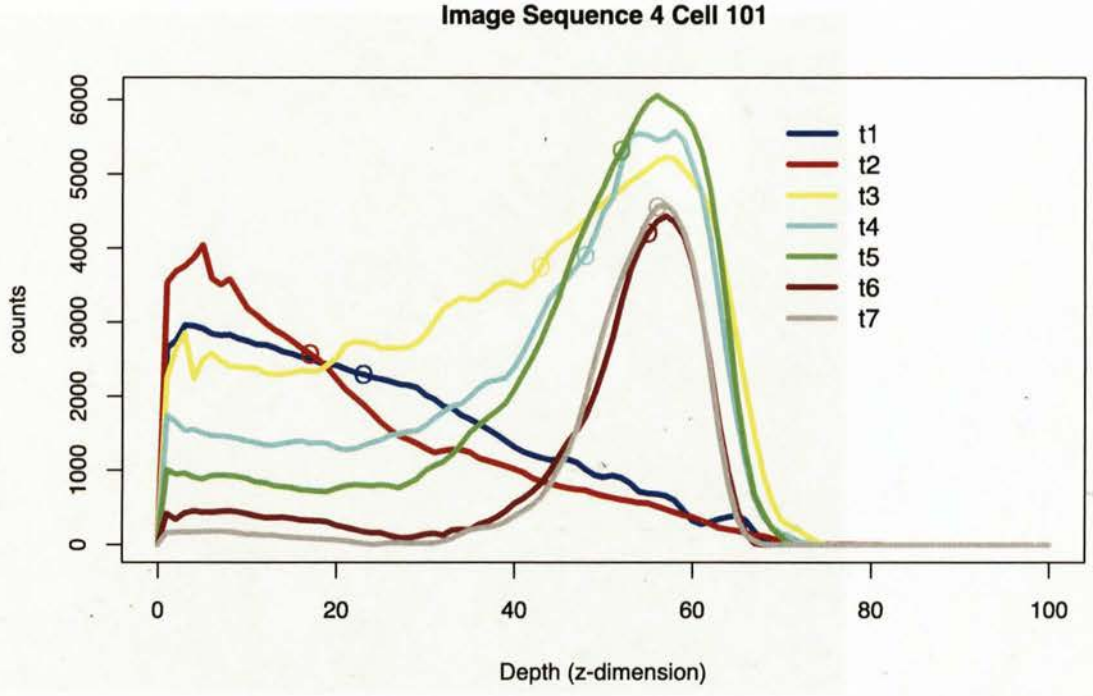
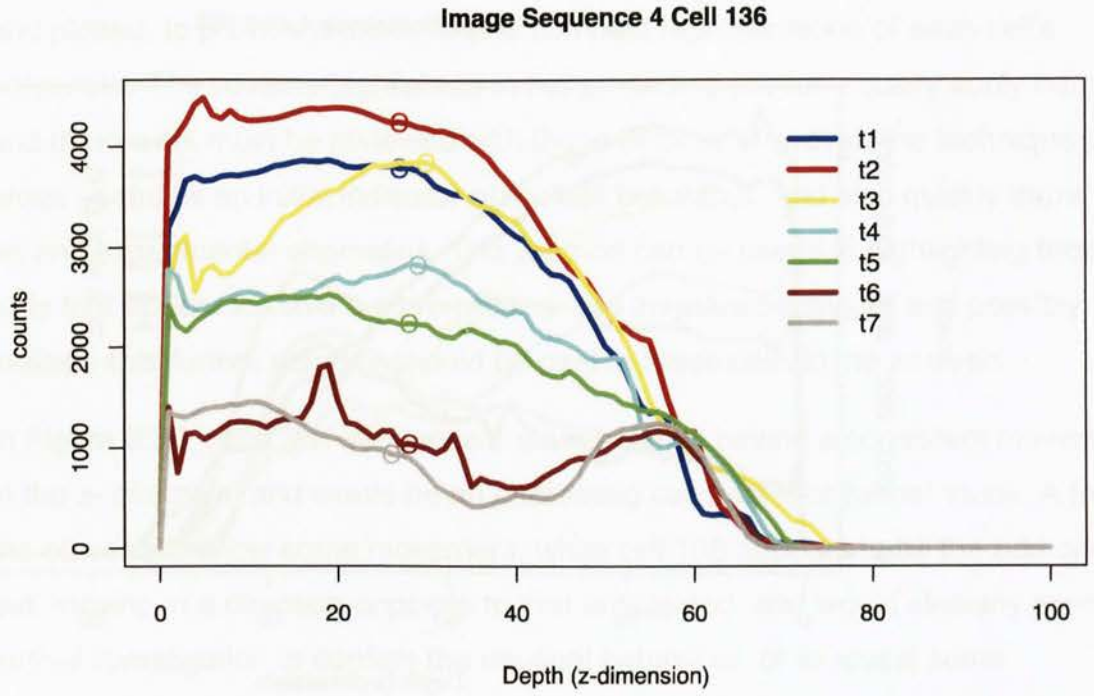


Figure 6.27: Sequence 4: Cell invasion plots for Cell 58 and Cell 101



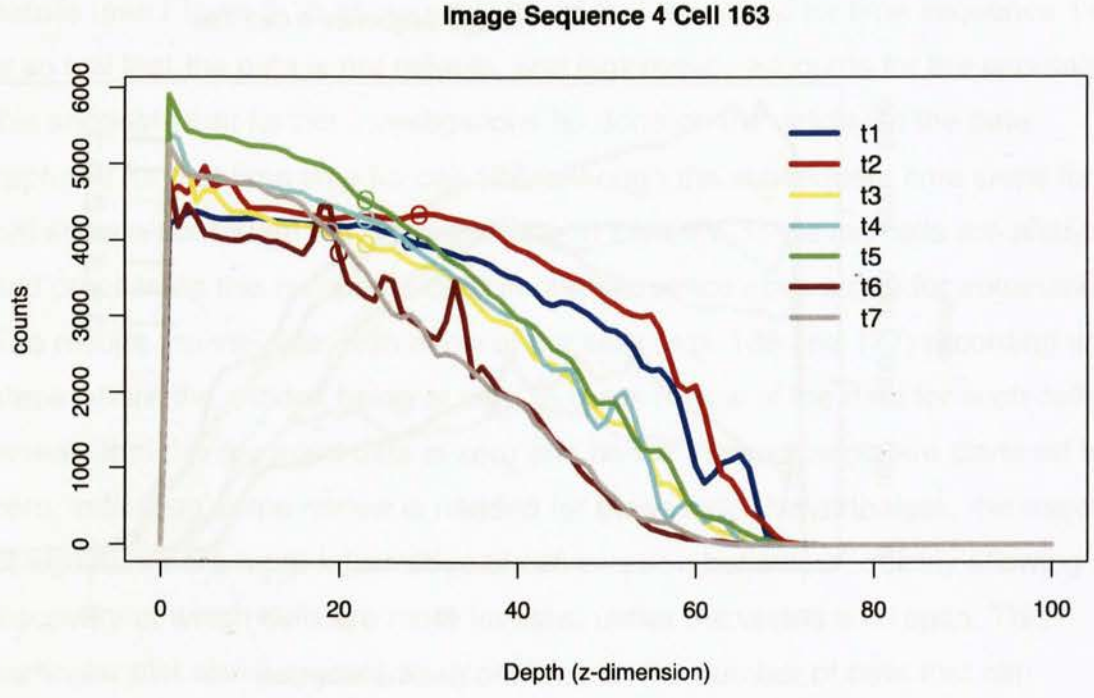
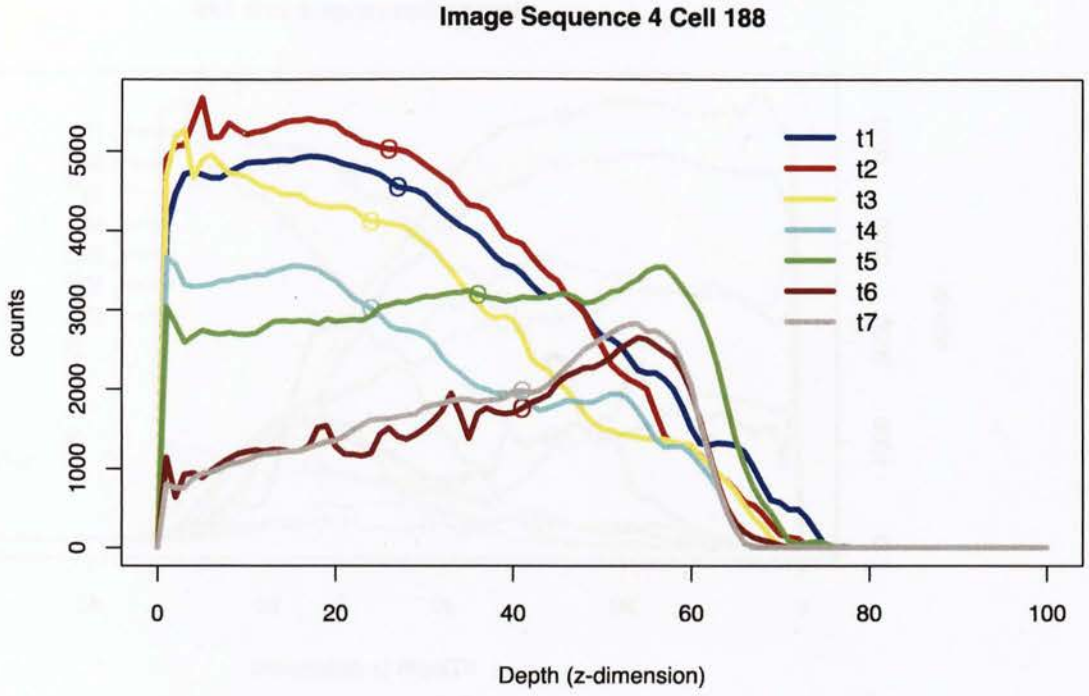


Figure 6.28: Sequence 4: Cell invasion plots for Cell 136 and Cell 163



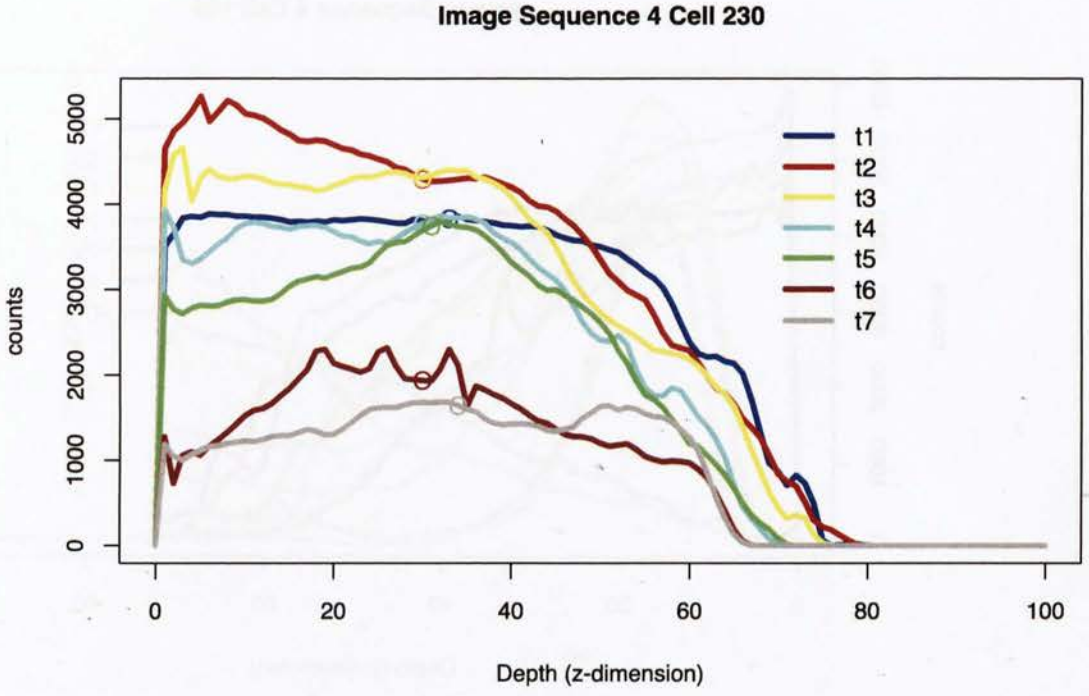


Figure 6.29: Sequence 4: Cell invasion plots for Cell 188 and Cell 230

6.6 Cell Invasion Signature Mid-Point Summary Plots

In this section, the midpoints of each cell invasion for each sequence are collected and plotted, to provide a primitive and compact representation of each cell's behaviour. The caveats highlighted in the preceding section equally apply here, and the results must be reviewed with those in mind. However, the technique can prove useful as an initial indicator of the cell behaviour, and also quickly throw light on any experimental anomalies. This method can be useful in highlighting those cells that appear to have a more pronounced invasive behaviour and possibly indicate that further scrutiny should be paid to these cells in the analysis.

In Figure 6.30, cell 73 of sequence 1 stands out as having a consistent movement in the z- direction, and would be an interesting candidate for further study. A few of the other cells show some movement, while cell 108 appears to be the odd one out, moving in a direction opposite to that anticipated, and would similarly prompt further investigation to confirm the unusual behaviour, or to reveal some shortcoming in the captured data. In Figure 6.31 cell 76 shows movement, while cell 136, movement is less noticeable. The plot for cell 187 flags up some odd behaviour, with an initial movement in an unexpected direction. On inspecting the details (see Figure 6.16 above), it is found that the signal for time sequence 1 (t1) is so low that the data is not reliable, and legitimately accounts for the anomaly, this suggests that further investigations be done on the validity of the data captured for that time step for cell 187, although the subsequent time steps for this cell show a consistent movement profile. In Figure 6.33 eleven cells are analysed, and processing this number of cells in one sequence does argue for automation. The results are variable, with a few of the cells (e.g. 136 and 177) recording time steps where the z-index being at zero. A quick review of the data for such cells reveals that the recorded data is zero and hence the mid-points are clamped to zero, indicating some review is needed for these cells. Nevertheless, the majority of signatures are more informative of cell invasion behaviour, quickly allowing the discovery of which cells are more invasive under the tested time span. This particular plot also suggests an upper limit on the number of cells that can reasonably be placed on the same graph, and in this case a next step might be to split the information across multiple plots to reduce the information per plot to a manageable size. Figure 6.33 present very clear and informative data on cell invasion signatures for image sequence 4.

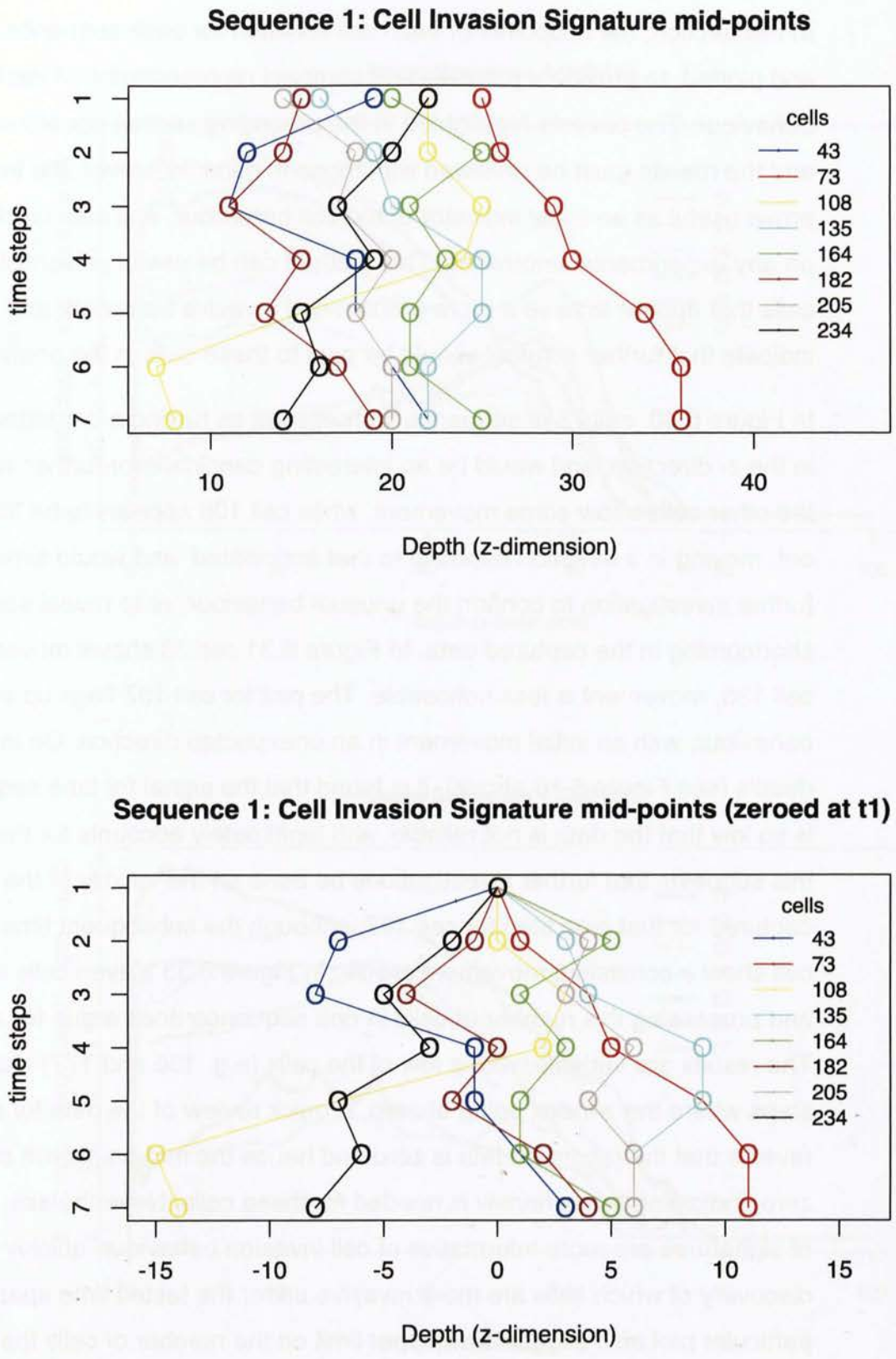
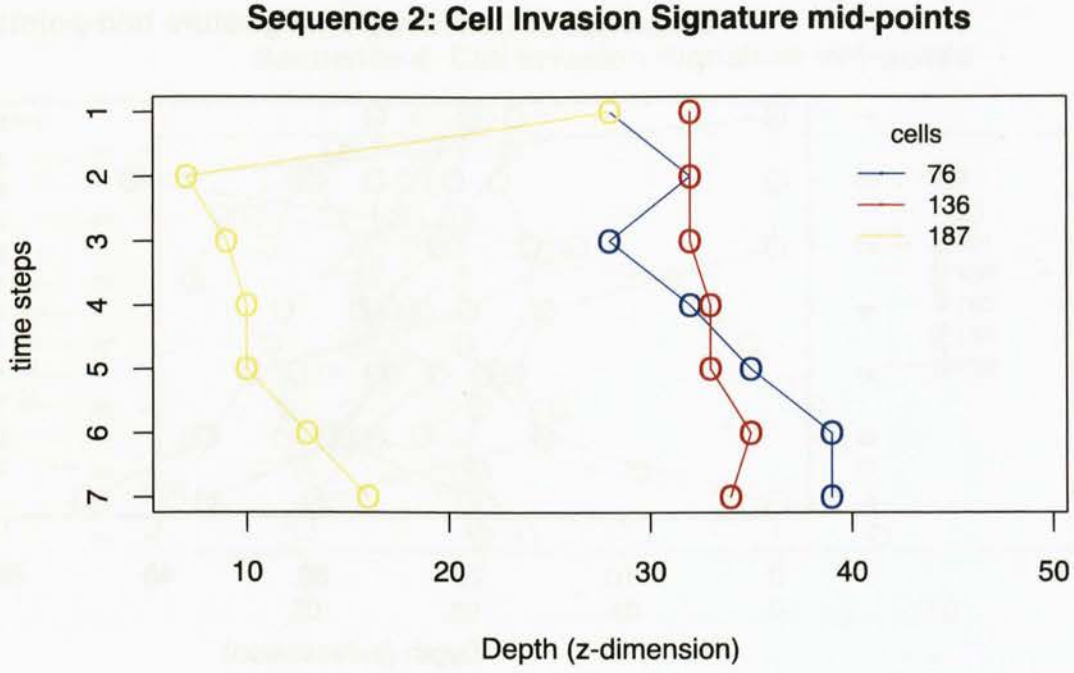


Figure 6.30: Sequence 1: Cell invasion signature mid-point plots

Top graph: absolute depth in z-dimension, Bottom graph: depth relative to t1=0



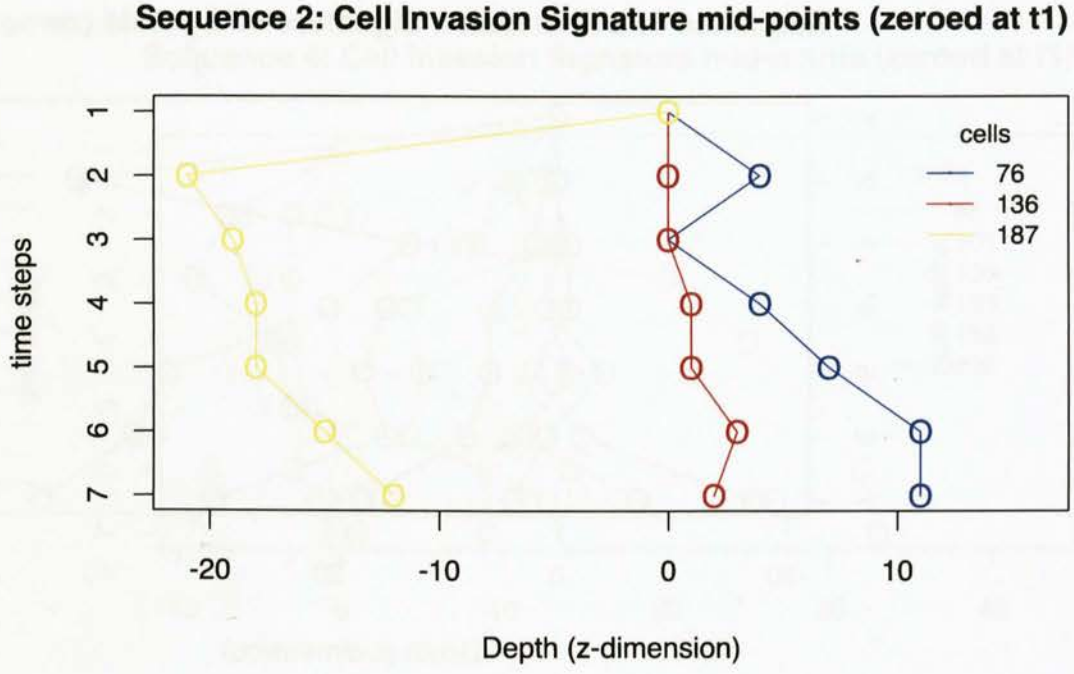
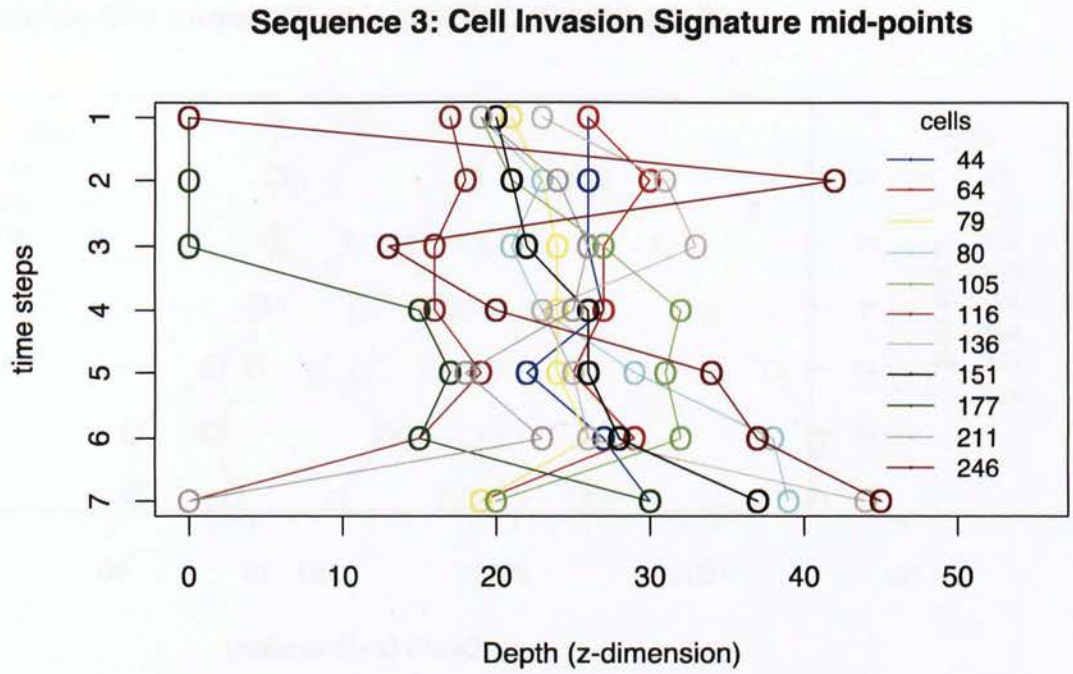


Figure 6.31: Sequence 2: Cell invasion signature mid-point plots

Top graph: absolute depth in z-dimension, Bottom graph: depth relative to t1=0



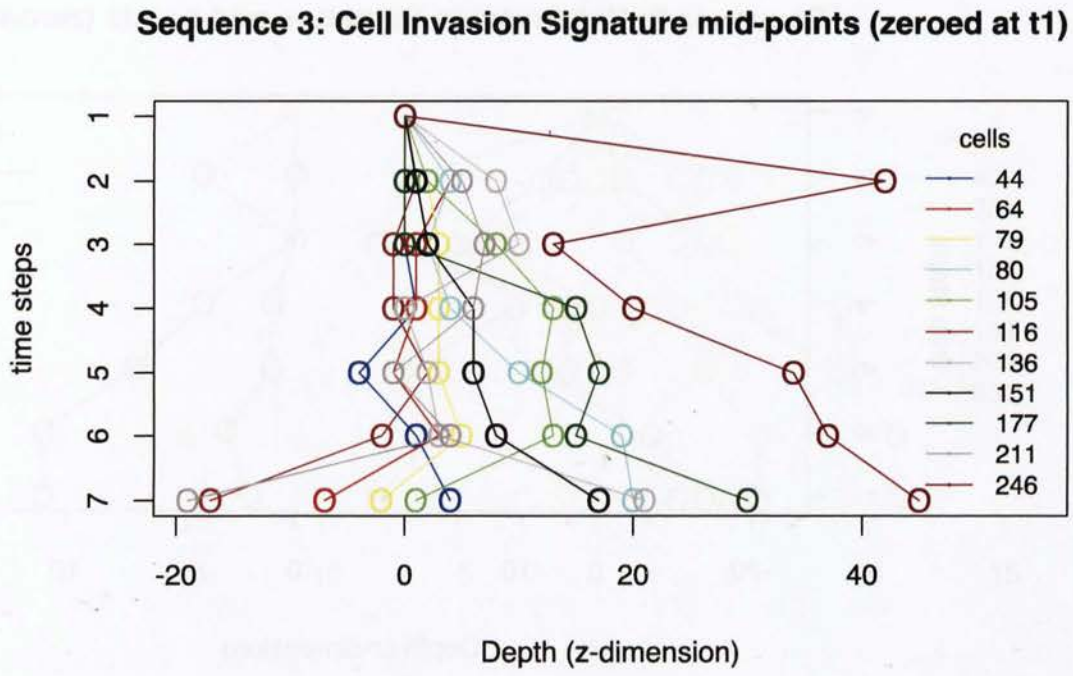
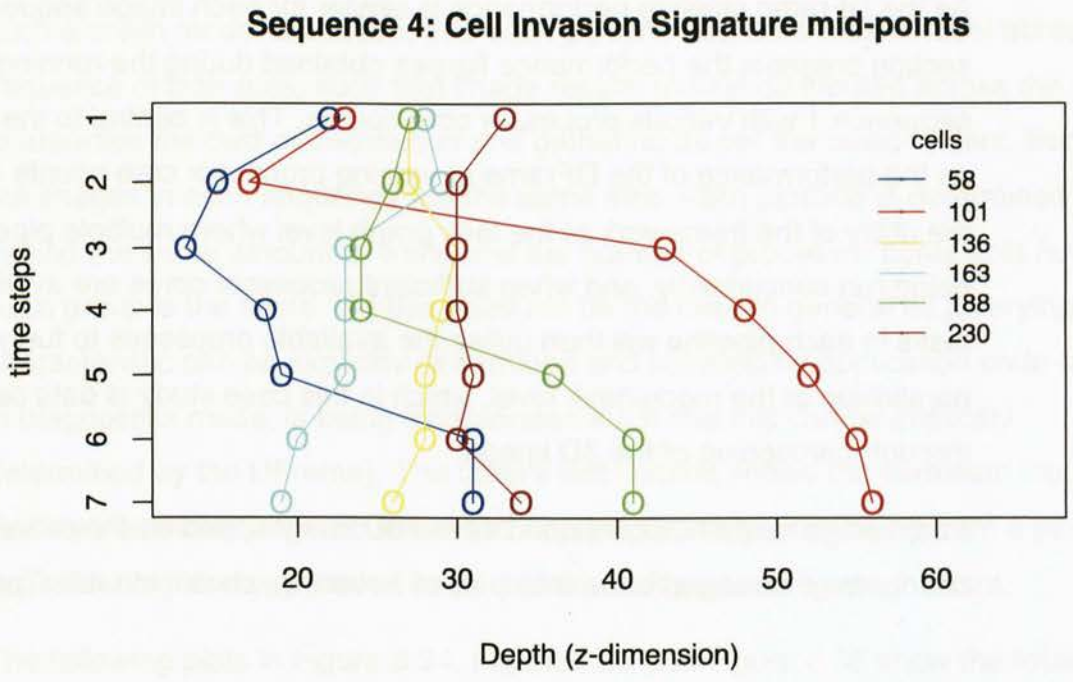


Figure 6.32: Sequence 3: Cell invasion signature mid-point plots

Top graph: absolute depth in z-dimension, Bottom graph: depth relative to t1=0



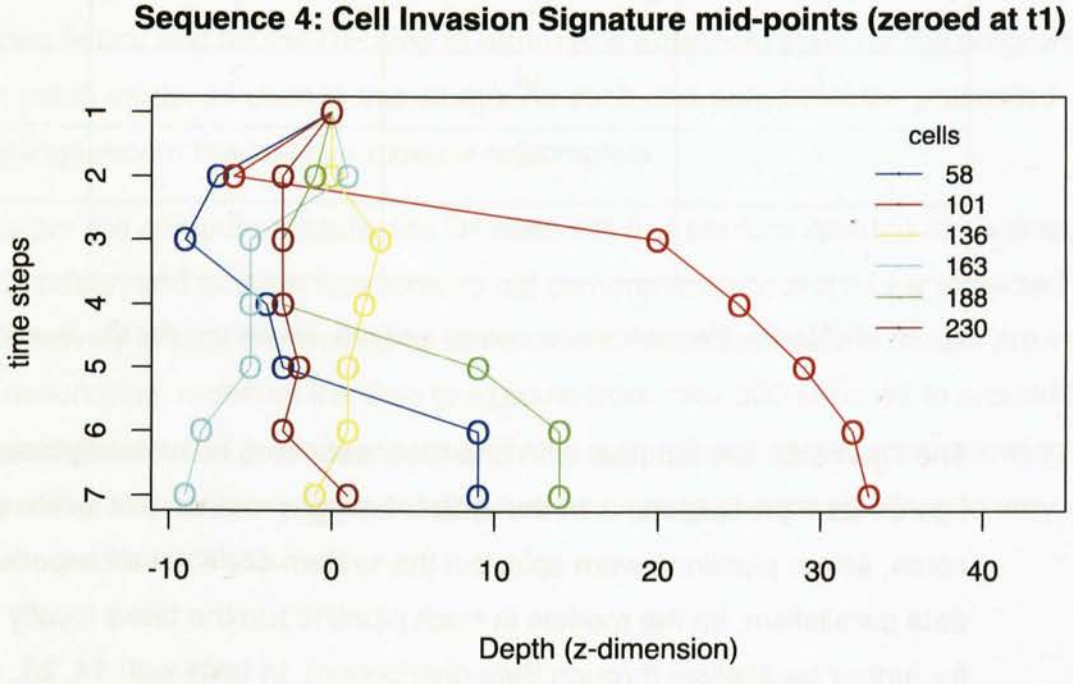


Figure 6.33: Sequence 4: Cell invasion signature mid-point plots

Top graph: absolute depth in z-dimension, Bottom graph: depth relative to t1=0

6.7 DFrame Pipeline Performance

As the DFrame pipeline performance is similar for each image sequence, this section presents the performance figures obtained during the running of image sequence 1 with various processor core counts. This is central to the case study, as the performance of the DFrame at varying processor core counts demonstrates the utility of the framework at the task graph level where multiple pipelines are being run concurrently, and when sufficient processor cores are available, that tasks in each pipeline will then utilise the available processes to further harness parallelism at the model/task level, which in this case study is data parallelism through partitioning of the 3D image.

Each image sequence contained seven 3D images, and so the processor core counts were arranged to be multiples of seven, as shown in table Table 6.1.

Number of Images	Number of Processor cores	Processor cores per Split Group	3D Topology created
7	1	n/a	1:1:1
7	7	1	1:1:1
7	14	2	2:1:1
7	28	4	2:2:1
7	56	8	4:2:1

Table 6.1: Processor core counts and process groups for the case study test

The figures for the top row, with one processor core were extrapolated from the figures for 7 processors, with the splitter timing removed. For seven processor cores, seven pipelines were split, but the system does not arrange for any further data parallelism, so the models in each pipeline run the tasks locally (i.e. no scope for further parallelism through data distribution). In tests with 14, 28, and 56 processors, each split pipeline has available multiple processor cores and so each split process group can arrange for partitioning of its image and distribution to leverage (data) parallelism at the model level. The 'Processor cores per split

group' column tabulates the number of processor cores available to each group, and these counts are suitable for data parallelism using a regular mesh model. As such a mesh model is chosen, and each pipeline caches a mesh model across the sequence of task runs, such that image results remain distributed across the runs to amortise the cost of distribution and gathering as per the design intent. Because the images in each sequence are the same size, each pipeline is determined as having the same amount of work, and the number of processor cores split out for each group is the same, but this need not be the case in general as a varying size characteristic can be explicitly determined and supplied by application code (and in diagnostics mode, is being incorporated such that this can be implicitly determined by the DFrame). The table's last column shows the cartesian topology that was automatically calculated and applied, the behaviour being from a plugged in partitioning strategy loaded as part of the task partitioning mechanism.

The following plots in Figure 6.34, Figure 6.35 and Figure 6.36 show the total execution time, the speedup and efficiency of the pipeline runs with different processor core counts. Although the DFrame timing diagnostics captures the results for each process in the run, the graphs only show the total timings for processor core 0. This is sufficient for presenting the total run times as the total time captured is the time for all split processor groups (sub-pipelines) to run to completion, and for the DFrame to return to a quiescent state (or the program exits in batch mode, as used in this study). As such, it is noted that the presented timings record the last sub-pipeline to complete.

As per the component tests, the DFrame still has barriers inserted at each task boundary, and at other locations, to aid comprehension, and this is expected to negatively impact the timings to some extent. Nevertheless, the results are very encouraging, reducing the time to execute from over 500 seconds to around 21 seconds using 56 processor cores. This is a significant contribution, allowing the running, inspection, adjustment and further running of such pipelines in very much reduced timescales.

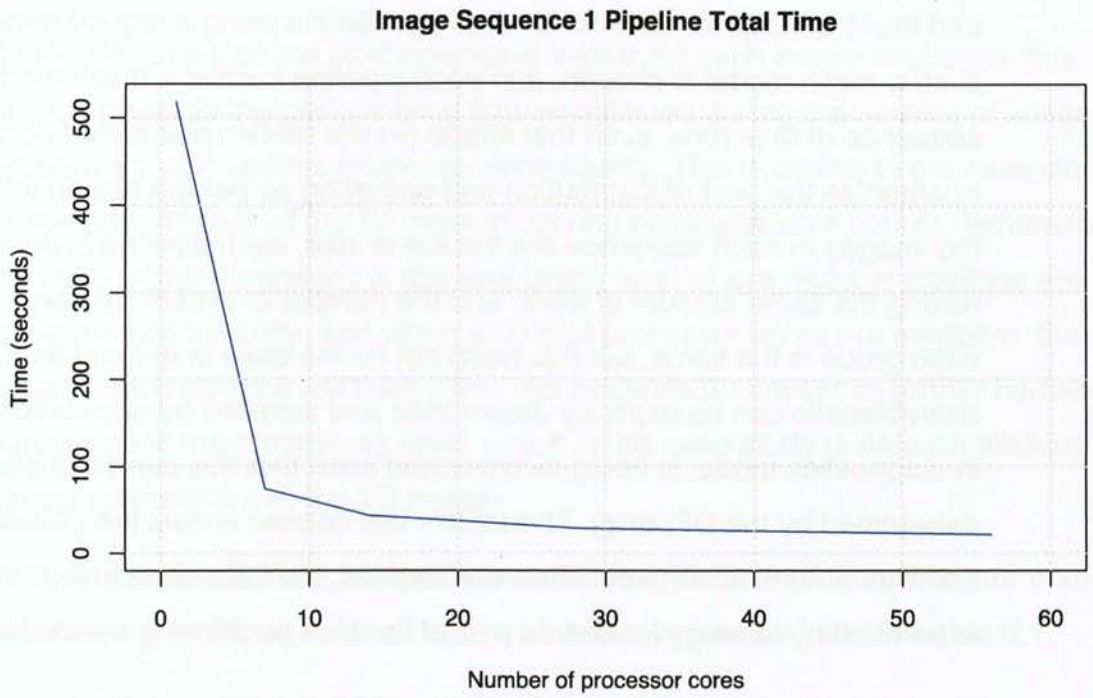


Figure 6.34: Image Sequence 1: DFrame pipeline total time

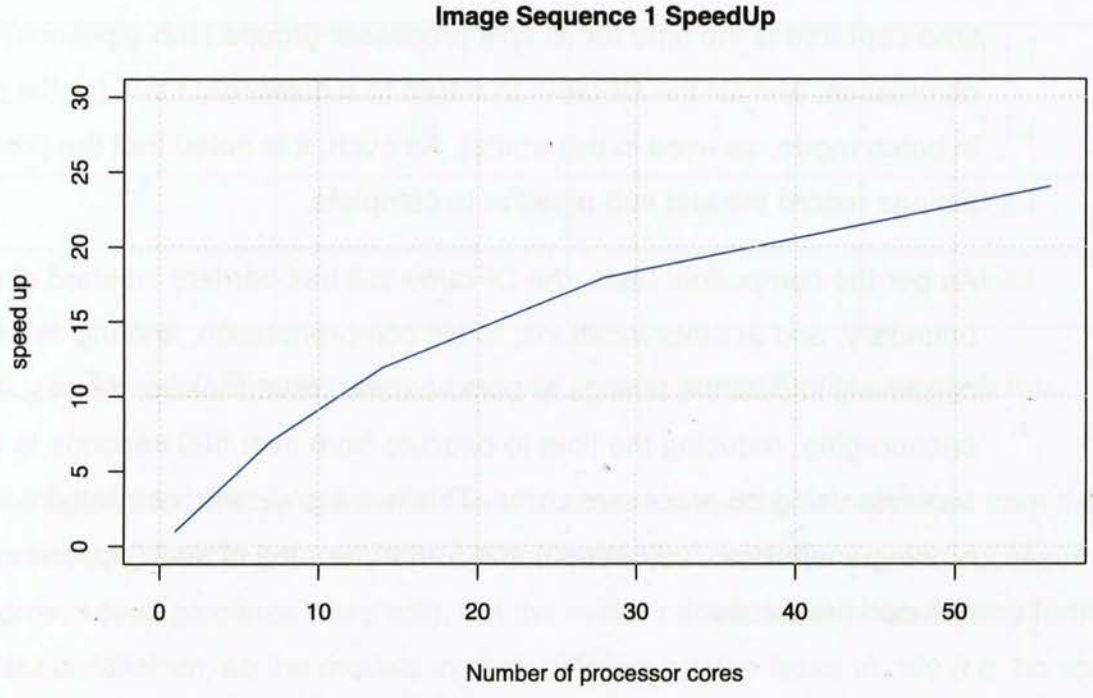


Figure 6.35: Image Sequence 1: DFrame pipeline speedup

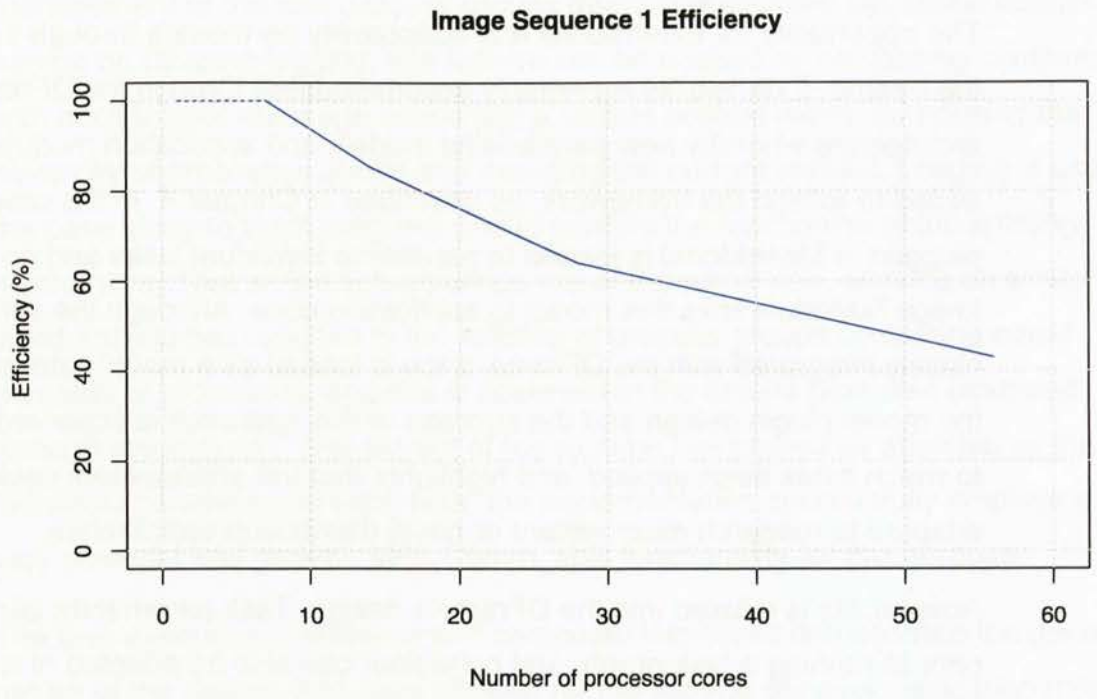


Figure 6.36: Image Sequence 1: DFrame pipeline efficiency

6.8 Discussion

This case study more fully demonstrates the DFrame design by running a task graph composed of a number of parallelised 3D imaging tasks. The task graph also explicitly specifies the dynamic generation of sub graphs (pipelines) through a DFrame task splitter, and so exercises the hierarchical control of parallel resources through the creation of multiple process groups. The DFrame performance results evidence the utility of the framework in delivering significantly reduced timescales to run such applications. Although performance is a prime goal, other aspects of the study are important, including the ease of use and reuse, adaptability and extensibility of the DFrame. The flexibility of the DFrame in terms of ease of use and reuse is most obvious to the domain user working at the task and task graph level, where the GUI provides assistance in constructing the task graph that forms a specific application workflow. Here the user is presented with the available functionality, and can build bespoke task graphs and configure parallelised task parameters according to the application requirements. This is

how the case study task graph is authored. The full power of the DFrame then being leveraged in the running of this task graph.

The opportunity for extensibility and adaptability permeates through the design of the dframe. Extensibility is primarily accommodated through the DFrame's plugin architecture whereby new parallelising models and application modules can be added to extend the framework, as described in Chapter 4. In the case study a plugged in MeshModel is loaded to parallelise individual tasks and an ImageTkModule links this model to application code. Although the splitter is closely integrated with the DFrame, it too is loaded as a model extension, using the model plugin design and the success of this approach is observed in the ease to which it has been applied, and highlights that the arrangement could be easily adapted to research other variant or novel distribution approaches.

Adaptability is infused into the DFrame's design. Task parameters can be adjusted prior to running a task graph, and behaviour can also be adapted at runtime. The ability to select and change the model that a task uses and the runtime apportioning of groups of processors to branches of task graphs according to task characteristics are two prime examples. As well, the partitioning and recomposition of task data is also adaptable, through the plugin of variant partitioning and recomposition strategies. The case study encompasses all these features, particularly adapting through automatic process splitting and automatic image partitioning.

Model selection is simple when tasks are run separately, and various suitable models can be swapped in to determine the performance of each. However, when tasks are arranged in a pipeline, selecting models while accounting for the pipeline context can improve the overall performance. In the case study, this is the case and the model selection is a mesh model. This is because the output of the averaging task is designated as the input of the Sobel task, and by leveraging a mesh model, the model setup can be reused, and the results of the averaging step can remain distributed for the Sobel step. The model performs a neighbour exchange of data rather than gathering and redistributing the data. A built in mechanism allows the DFrame to cache the model such that the same model is used across the tasks. This arrangement also extends to the analysis and histogram task, where the output from the Sobel operator remains distributed, as its input.

The splitter functionality managing the apportioning of resources to sub-pipelines has proved to be very effective, accomplishing its objectives including the management of the split process groups through associated DfContext objects. By turning on DFrame logging, this feature can be tracked as processing continues, with each logged message containing a unique context name, comprising the hierarchy of processes above and including the current context. Logging is used in the case study to track progress and to confirm the functioning of the splitting mechanism. That all the sub-pipelines are of the same size, working on similar sized images has resulted in the splitting of process groups containing equal numbers of processes, and this is observed in the results (and also observed through the logging). This aspect of the runtime has proved as effective as the individual parallelism of each task, the implementation successfully employing the very powerful and generic MPI_Comm_split functionality for this purpose.

The task execution, partitioner and composer interfaces demonstrate the generic nature of the design, and have allowed for the loading of novel application module code that implements the interfaces. A 3D image partitioner was loaded that supported the automatic calculation of partitioning strategies, according to the shape of the 3D image (i.e. the size of each of its dimensions), and the number of processes available. From this a topology was calculated that the mesh model then used to set up a corresponding topology across the processes and used to distribute the partitioned sub-images accordingly. This has proved very effective, and also brings out the design in that the partitioning strategies can easily be changed to trial novel approaches and augment automation and adaptation in new ways. For completeness, a manual topology can also be mandated by adding an appropriate property to a task's specification, which should then be aligned across a pipeline in order to avoid gathering and redistribution of data, when it is appropriate to leverage that advantage. The final composer aggregated histogram data and persisted results to the file system for subsequent examination and rendering as compound cell invasion plots for each cell. In this case, the transformed image was not required to be recomposed (except for inspection during trial runs, but not in the presented results), supporting the argument for the more generic design.

Another goal of the case study, is that in conducting a full end to end test on a complex and dynamic task graph, it would be possible to discern where the design

could be further developed. One simple improvement will be to arrange for the loading of data in a separate task. This would allow the data to be loaded once, and then used in multiple distributed runs of a task graph or subset thereof, for diagnostics purposes, including assessing timing effects of certain parameter changes, or when assessing the use of different models or partitioners. The DFrame design already allows this to be simply expressed as another task. This will improve the runtime flexibility of the system, aiding the development of iterative routines that multiply exercise subsequent tasks to provide feedback to automate further the tuning capability. This adjustment would also simplify the timing of the retrieval of resources, and obviate the need for multiple retrievals on multiple runs.

In the case study each pipeline sub-workflow saves its own output files directly. This arrangement is in preference to the amalgamation of all the results onto one root process or to a parent in the context hierarchy, as there will be some variation in the end times of each pipeline, and this could be a positive effective on disk writes to some storage systems. Indeed, if the simultaneous running pipelines were to write to the same file, then race conditions would have to be considered such that data from each process did not corrupt data from other processes. The MPI parallel IO feature could be useful in this respect, but is possibly excessive and restrictive in the current case. However, it is within the design of the DFrame to be able to arrange a merged parent task that can amalgamate and write out all the data concurrently, and this will be a useful strategy, if the data is to undergo further automatic analysis. For example the 3D image cell histograms may not be the output, but some metric derived from them. In this case, instead of using parallel IO, the dframe would control the phased collection of data onto one process, and that process would then write out the results, or deliver to the client. This is indeed how the timing metrics are collected and recorded.

Another useful feature brought out by the case study is that an improvement could be made to more flexibly dictate when data should remain distributed and when it should be collected, with particular emphasis on the ability to do both. For example, the histogram data could be gathered, and the image data kept distributed or vice versa. This could also be useful in order to view intermediate outputs, and this would require both to gather data and leave it distributed, and this is being incorporated into the design. Related to this is more flexibility in a model's control of data placement. The mesh model supports a neighbour

exchange of data and a mechanism to support the global exchange of data should also be available to models, suggesting that this aspect be further separated out as an independent feature. This was illuminated in the analysis task, where it may be useful to globally broadcast calculated thresholds, while keeping image data local. Although in this case a separate thresholding task would make sense, the capability for specifications to define or for module implementations to instigate global broadcasts of parameters in models (even the mesh model), would be of use, as some parallelised algorithms will undoubtedly require this. This can of course already be done in a bespoke manner in any model, but the idea is to investigate supporting generic and reusable distribution patterns other than neighbour exchange.

Finally, the actual analysis of data in the case study has provided useful insight into the effectiveness and limitations of the approach. Looking at the fit of the mask to images in a sequence, it appears that in some cases there is significant cell movement in the x-y plane (e.g. see image sequence 3), and here it may be worthwhile and even necessary, to create masks for each image or automate some method of reregistering a mask in the lateral plane, or define some criteria for determining other cell characteristics such as a cell's centre. The case study does highlight that the selection of the mask area is sensitive to which image and image slice it is extracted from, and whether it is broad enough to cater for a cell's anticipated lateral movement in the x-y plane. Also, the exercise highlights that the technique is more practical when cells in an image sequence are distinct and do not overlap. The upshot of this is that more analysis is required, and further experimentation and this is the strength of the DFrame in that a framework now exists, that can be used to more quickly expand functionality, and that the prime objective of speeding up the analysis makes this much more practical.

6.9 Summary

Chapter 5 evaluated the DFrame at the component level, and this chapter has then gone on to demonstrate the power of the DFrame in concurrently processing multiple streams of 3D images in a time lapsed cell invasion case study that pumps each image through a pipeline of preprocessing and analysis operators. The build time power is observed through the plugin of model and module functionality, and the composition of a complex task graph. The runtime functional

operation of the DFrame is then observed, with features such as models being selected and then loaded in a distributed fashion, workflows being created from the task graphs, processor resources being split according to the workflow requirements and available resources. The generic partitioning and composition design is seen in action, and the automatic calculation of partitioning requirements and associated topologies is demonstrated.

Significant contributions of the DFrame have been highlighted through its use in a more complex application (see section 6.8). These include demonstration of the utility of the extensible design, allowing plugin and reuse of components within the parallel processing and the application domains. Adaptability at runtime is evidenced in multiple functions such as model selection, partitioning strategies and in dynamic processor group allocations. The generic execution, partitioner and composer interface design have also allowed the DFrame to be used more broadly as in the histogram analysis. Myriad other features have been brought to bare such as GUI drag and drop specification authoring and distributed runtime control, configuration, and diagnostics capabilities. The power of the DFrame is observed not only in the performance gains that it can give, but also in the provision of a framework that is particularly flexible and easily extended, simplifying and so encouraging the harnessing of parallel resources.

The successes and ongoing improvements into automated adaptability will continue to provide further power, with the DFrame's extensible design aiding this. Planned next is a consolidation phase, with more experiments with models and investigations into the generic exposure of global as well as local exchange at the model level, and keeping some data distributed while collecting or redistributing other data. Delivering DFrame performance information and application results to the GUI is also part of this plan. The UI design is being progressed to allow viewers to be plugged in for supported types. Currently, a 3D image viewer is inserted for use with the 3D image task inputs and output, but the goal is to make more generic, to support other formats such as the csv (comma separated value) dataset format.

Chapter 7 Conclusions and Future Research

7.1 Introduction

The project started with a requirement to apply multiple complex feature extraction, analysis and visualisation algorithms to large numbers of high resolution 3D bio-images in High Content Screening applications. It was immediately apparent that parallel processing would be essential, to obtain results in reasonable timescales, and allow alterations and reruns and visualisations in batch and interactive sessions. It was proposed that a new and novel high level extensible and adaptable framework would be useful, implementing much of the core parallel processing infrastructure and that aligned well with the parallelisation requirements of typical 3D image processing algorithms.

This initiated research to distil concepts and design an appropriate distributed framework (referred to as the 'DFrame') running on top of MPI, and to implement core imaging functionality pertinent to parallelising 3D image processing operators, using the framework. An assessment of typical image processing usage led to the view that both task and data parallelism would be useful, with task parallelism at the broader level to specify task pipelines and graphs of operators, and data parallelism at the finer level to parallelise individual operators (tasks), that would include decomposition and recomposition of the 3D images. Extensible support for this was to be accounted for in the design of the distributed framework.

Accomplishments of the project include a detailed conceptual design of the framework and a fully functional prototype, along with bridging infrastructure to core 3D image processing capabilities that includes de-noise filters, edge detection and region based cell segmentation operators and a parallelised server side direct volume rendered visualization capability. Results of the evaluations of these components are presented. Furthermore, a fully integrated case study has been undertaken that uses the DFrame in a real world setting, to investigate cell invasion signatures in multiple streams of time lapsed 3D cell biology images. The case study demonstrates the success of the DFrame in its largely transparent application to the target domain and shows how interesting and novel imaging techniques can leverage the framework to achieve faster time to results and insight on complex issues of much interest to research in this field.

7.2 *Principle Findings*

The DFrame design has elevated extensibility and adaptability to prime objectives, alongside performance, clarity and reuse through a modular design.The parallel infrastructure and pluggable models implementing parallel patterns of execution are separated out from application code, so that proficient parallel programmers can concentrate on developing parallelising patterns, whilst domain experts are largely shielded from this aspect and can concentrate on developing domain applications. This separation of concerns has been demonstrated to be of great benefit when improving the system, where effort to improve the DFrame and models can be shared across all associated application code, and the modular design encourages its use across many image processing applications requiring parallel processing capabilities. The experience of implementing the complex application partitioning, and arranging master worker and mesh model parallel execution does argue strongly for a framework. The hours of scrutiny of complex timing interactions to determine performance bottlenecks is a difficult and time consuming task, that can be helped with framework feedback. Although the models on first inspection appear to be conceptually straightforward, there is much reasoning to be done, so a framework is arguably essential.

A further level of complexity is in arranging for the execution of a graph of tasks, and the apportioning of processes to tasks and task pipelines running in parallel, and the management of such workflows. The workflow component has proved its worth in this regard, as a core part of the framework managing the runtime execution order of tasks according to a defined task graph. The modular design was shown to admit the relatively straightforward introduction of a novel powerful hierarchical splitting mechanism that operates in concert with the DFrame and the workflow component to dynamically manage resource allocation. This would simply not be productive (or even feasible) to do in each custom application. The complementary GUI successfully facilitated both the task graph construction from available components and the runtime management to set and interactively update task parameters and then to immediately rerun the application to provide interactive feedback on the effects of any intermediate task parameter adjustments. One of the difficulties in reusing code is simply in finding what is available, and this is addressed in the DFrame by exposing functions made available by the plugged in components in the GUI.

A generalise partitioning design has been incorporated that abstracts application neutral partitioning and composition API's that the DFrame models can use, and that applications implement, so that applications remain largely transparent to models and vice versa. This provides further insight into the separation of concerns, fuelling interesting research in this area.

Of principle interest in undertaking this work is the use of parallel processing to reduce the running time of pipelines of complex and varied 3D image operators, and the performance of the framework in this regard is of course crucial. A framework that meets goals of extensibility, adaptability and reuse, but falls short in this area would be of little value. Component evaluations and a case study have successfully demonstrated the framework's utility in enabling the harnessing of parallel processing resources to run such pipelines in much reduced timescales, and this is expanded upon in the following section.

7.3 *Critical Evaluations and Limitations*

The initial evaluations demonstrate the DFrame model and module design at the component level, and prove the specific utility in the image processing domain, in the reuse of parallelising models and the generic interfacing to 3D image specific partitioning and composition. The success of parallelising simple imaging operators, and the more elaborate segmentation and visualization implementations are testament to the frameworks parallelised model design at the component level. The direct volume ray tracing performance results are particularly encouraging, holding the prospect of real time visualization, using the GUI to provide interactive updates to the camera orientation and zoom and present immediate feedback. This method is very versatile, and further research could include other partitioning strategies, the incorporation of variant ray tracing algorithms and could even be adapted as an alternative segmentation approach.

The case study bears out the utility of the framework where components are brought together to compose a complex image processing application that focuses on a real world biomedical area of research, namely the investigation of invasion signatures of cancer cells across a number of sequences of time lapsed 3D images. This proof of concept required the set up of a pipeline of image operators, applying compute intensive processing to each image, and recording the results. A further complication was the generic nature of the partitioning and composition

stages, which not only involved the partitioning of 3D images, but also the composition of resultant analysis data. This was successfully accomplished due to the generic design of the partitioning and composition interfaces. The full power of the framework is demonstrated, including the adaptable splitting of process groups across multiple pipelines of image streams, appropriate model selection that where possible can take account of image partitioning to avoid unnecessary recompositions and repartitioning, and the integration of automated partitioning strategies. The flexibility and extensibility of the pluggable model and module design becomes more evident in this study.

Performance analysis can help illuminate the characteristics and bounds of system storage IO, interprocess communication, memory hierarchies and processors, and their interrelation and effect on the performance of the problem being solved, and must be considered in concert with the 'shape' of the problem being solved. Instrumentation within the DFrame is made available across models and modules, to aid performance analysis. At present this is still rudimentary, but does collect sufficient information to establish core metrics such as execution time and speed up, and being part of the framework, continued improvements will benefit all applications. It is planned that this information will be returned to a client for immediate feedback, but more importantly can be used to automatically adjust operation parameters based on reported timings. The higher level abstractions can impose a trade off with performance, but the presented preliminary results are very positive.

The assessment of results guides and drives improvements to algorithms to maximise performance (and can even highlight which hardware components can be improved). Analysis of the tests has revealed where parallel pattern model and image processing module implementations can be improved, for example in synchronisation and intercommunications, removing barriers and using topology routines, and that it might be worth further research into using parallel IO (supported by MPI-2) and perhaps dedicating a subset of processors to storage IO and then to use internode communication to transfer data about the network. This must be balanced against increased program complexity.

Overall, the component evaluations and case study results are all quite encouraging, and the feedback has already targeted areas for further improvement, both in the imaging domain code, and the distributed framework.

Using the framework has also brought into relief the suitability of particular patterns for different algorithms and applications, further demonstrating that a framework not only allows reuse, but provides guidance and flexibility on available parallel patterns of computation. The project illuminates areas for further research in the parallel processing arena and in parallelising image processing applications, that without this initial extensive endeavour might seem too daunting and so deter interest. More often in research, the goal is very specific, and the construction of bespoke parallel processing may be secondary and divert much effort from the core interest. The DFrame can alleviate such concerns and encourage and aid rapid experimentation by facilitating parallel execution.

7.4 Future research

Although intended as a proof of concept prototype, the presented DFrame is a functional piece of software able to demonstrate flexible, extensible and adaptable parallelised processing. However, the enterprise is an extensive one, ranging across both the parallel processing discipline and the parallelisation of 3D image processing applications and there is scope to improve and extend in both domains. Below are some areas that are already targeted for more input.

The performance timings are generated by the DFrame but they are still somewhat rudimentary and incomplete, and could be improved, while attempting to maintain a balance between the DFrame diagnostics and the leveraging of external tools that are already available (e.g. MPE/Jumpshot). Some improvement in the already supplied models using the DFrame timing infrastructure is also planned. The DFrame outputs timing data to a file in cvs format and can be configured to also output summary runtime data, and this should be augmented to be piped to the UI for immediate inspection. It is still rather time consuming to run a test, and have a post operation to plot the data, and the intention is to have this feedback immediately rendered as a graph in a UI 'runtime' view (a facility is already included in the GUI to view output 3D images).

The evaluations suggest that the performance might be improved by adjusting the 3D image partitioning design to reduce the copying of data to sub-images, by experimenting with a windowing design instead, such that data is packed directly from the whole image to communication buffers, to reduce copy (and memory) overhead. In the current implementation, MPI's packing functionality is used

exclusively to pack and transport messages, and it would also be an interesting avenue of further investigation to incorporate MPI's datatype feature and determine whether such usage could improve productivity and performance.

It is intended to apply this framework to a much larger scale cancer cell invasion study, targeting many gene modifications and this will entail the development and parallelisation of many more plugins implementing 3D imaging techniques that are beyond the scope of this project. Here, a form of 'smart processing' could be introduced that automatically identified regions of interest, marking and queuing them for further processing. It would also be of great utility to apply the framework to other interesting and topical compute intensive applications, such as research on how differential proliferation rates orient cell growth (Mao, Tournier et al. 2013).

Reuse is a corner stone of productive quality software development (Sametinger 1997). It plays a fundamental role in object oriented methodologies and is very prominent in the design of the DFrame, encouraging a modular approach where applications are built from reusable components. As well as avoiding the duplication of effort, reusable components become more 'battle hardened' through use in many different applications, and the investment to develop and rigorously test a reusable component can be amortized across all applications that use it. At a high level, the DFrame reuse is 'compositional', in that available models and application code can be reused to compose novel applications and there is room for further study in this fertile area. For instance, another more specific and higher level of reuse is the 'generative' approach, where components (or applications) are generated from high level descriptors, and although less generic, it might be useful to consider the merits of such an approach perhaps sitting atop the DFrame.

The high performance computing community is largely populated by experienced practitioners in the field of parallel processing, with particular skills in performant low level technologies such as MPI and openMP. One of the motivating aims of the DFrame is to shield users from the complexities of low level parallelization, and so bring the power of MPI to a wider audience, outside the high performance community. Indeed, the adopted client server architecture opens up the opportunity to integrate the DFrame as a service in a contemporary system employing a service oriented architecture (see section 3.7). In the same vein, cloud computing is becoming an increasingly popular way to harness scalable remote resources (e.g. map reduce for Big Data analytics), and it would be an

interesting further pursuit to establish the expected suitability of the DFrame in these settings.

7.5 *Summary*

This research has shown that parallel resources available over distributed memory architectures can be successfully applied to 3D imaging HCS applications through the use of an extensible and adaptable framework build atop MPI, and leveraging its point to point and collective operations as well as its more advanced topology and process management features. Task parallelism is employed in the processing of a pipeline (graph) of 3D image operators, and task and data parallelism can be further applied to parallelise each image operator. Core benefits of the framework include its modular design, separating the parallel processing infrastructure from domain code and allowing for extensibility, adaptability and reuse. As well, the provision of a system that can provide interactive feedback to application parameter adjustments should prove very helpful in many research domains associated with 3D imaging. The consideration of parallelising image processing operators (individually and grouped as a pipeline of operators) should provide a fertile area for interesting future research. As this project demonstrates, some algorithm designs map better to parallel resources, and with the computer landscape trending to be ubiquitously parallel, the search for such algorithms and approaches that can take advantage of this is becoming a more important area of research.

Appendix

A.1. Scientific Visualization

Preliminary project effort concentrated on visualization techniques applied to 3D imaging datasets, including the decision on what technology to adopt for a prototype graphical user interface (GUI). Although the initial focus is on the server side parallelisation of direct volume rendering visualization techniques, the GUI should also be able to arrange the pipelined rendering of generated polygons, such as the surfaces of discovered image artefacts or to render the bounding box of an image or image artefacts as a wire frame. To this end, the GUI technology should allow support for openGL (OpenGL Architecture Review Board, Shreiner 2004). However, openGL itself does not specify how to interact with a windowing system. Some experimentation was conducted on using the X window system, on using the GLUT openGL utility toolkit (openGL 2008), and on using QT3 from Trolltech (Dalheimer 1999). GLUT is more useful in allowing easy and quick access to learning openGL, and the X window system is extensively used in the unix world. In the end, QT was chosen as the most attractive option to quickly develop a prototype GUI. QT implements a QGLWidget that provides the support framework to easily leverage openGL functionality in applications. It is merely necessary to extend this class and add functionality as required. QT also offers a designer tool that helps create GUI's visually, speeding up the design process. As well, QT uses a signals and slots mechanism that makes it particularly straightforward to hook up the various GUI components. The latest version of QT is version 4, but the version initially used in the project and illustrated in this appendix was 3.3, as it was already installed on the default Suse Linux distribution used at Kingston University (Suse Linux 10.1). As is noted in Chapter 4, the UI is now reimplemented in QT 4. Excellent introductions to computer graphics using openGL can be found in (Hill 2001),(Angel 2001a) and (Angel 2001b).

A.1.1 Preliminary pipeline prototyping

Prior to adapting to a server side distributed design, and to gain insight into the distributed pipeline architecture, a simple pipeline framework was first developed. The objectives being to provide some order and management to the application layer in the initial GUI prototype design, to help manage the complexity as

research and development progressed, and more importantly to extract salient factors to be included or accounted for in the envisaged distributed design. Artefacts of this generic pipeline architecture would eventually port across to the development of the distributed framework. The base object is a 'system component' which can be assigned inputs and outputs and can be extended to meet particular component implementations needs. The inputs and outputs of system components can be connected to form a pipeline through which a data context is passed (each system component contributing some specific processing to the data). In the context of visualization, subclasses can be source components that provide image datasets, filter and analysis components, and rendering components. The generic pipeline architecture has no link to the visualization code proper, as this is provided by specific sub classes. An interesting consideration that emerged was whether to have a push or pull type pipeline. The distributed framework is intended to provide both features, in that events that propagate back from downstream components can initiate a refresh of the data upstream (pull), and if data upstream is changed in some way (say a parameter change) this will also cause a refresh of the pipeline (push). The VTK also uses a pipeline architecture that enables flexible construction of visualization applications (Schroeder, Martin et al. 2004a, Schroeder, Martin et al. 2004b) and much insight was gained from studying this approach. Another source of inspiration in this regard was the now open sourced IBM software package openDX (IBM 2007), designed for the advanced visualization of scientific and engineering data.

Alongside this work, much of the preliminary and experimental work to implement 3D volume structures was undertaken. The concomitant required supporting IO functionality was also implemented, which entailed porting the core ImageJ 'tiff' encoder and decoder functionality to C++ and adapting the design to directly support the required 3D volume data structures.

A.1.2 Iso-surface visualization prototype

One particularly useful visualization technique is the extraction of iso-surfaces from a 3D image dataset. Rather than using a variant of the 'marching cubes' algorithm, a more direct thresholding approach was researched, undertaken as a preliminary investigatory exercise associated with the required ray tracing facility and to aid the GUI prototyping. The initial prototype GUI was implemented using

C++ and QT3, and incorporating openGL is shown in Figure A.1. The rendering was set up with openGL lighting disabled and z buffering enabled. A synthetic 3D image of a sphere was generated to provide an initial source image. The algorithm proceeds by traversing the image dataset to detect and identify all those points on or above some specified threshold. All identified points are then clamped to the threshold. Normals are then calculated (via the calculated gradients at a point) for all points on the surface, points within a surface being ignored as they are subsequently removed. A smoothing algorithm is then applied to these surface normals only, to smooth with adjacent normals. The amount of smoothing can be adjusted via the GUI. Finally, the extracted surface data and normals are compacted and propagated to code that renders the points.

The calculation of the normals is necessary in order to be able to calculate the 'shading' when rendering a particular point. The light source was assumed to be fixed at the point of the camera, and the shade for each point calculated by the dot product of the direction of light incident at a point, and the surface normal at that point. In order to cope with this requirement, a camera object was developed that tracked the camera's position and orientation (and subsequently adapted for the ray tracing functionality). This also enabled easy implementation of pitch, yaw and roll of the camera for dynamically viewing the results. Design ideas and skeleton code for implementing the camera were extracted and developed from (Hill 2001) which also provides a comprehensive introduction to computer graphics. Note that by arranging to move the camera rather than the object(s), object normals only need to be calculated once. A capability was added so that the camera angle of vision could be adjusted. At very small angles, the resultant visualization rendered discrete points as expected. To provide a smooth rendering, cubes were then constructed and rendered for each point, each cubes dimension being equal to the spacing of the points in the dataset.

Another sphere source was also implemented in which the normals were calculated rather than determined via gradients and smoothing, to compare against the above implementation. With sufficient smoothing cycles (~20) the above implementation approaches the calculated image. Figure A.1 and Figure A.2 show the resultant rendering of the test sphere dataset with different amounts of smoothing applied to the normals.. Although this design has proved successful in the case of a simple sphere source, further experimentation will be needed, to

determine how well this technique translates to arbitrary iso-surfaces.

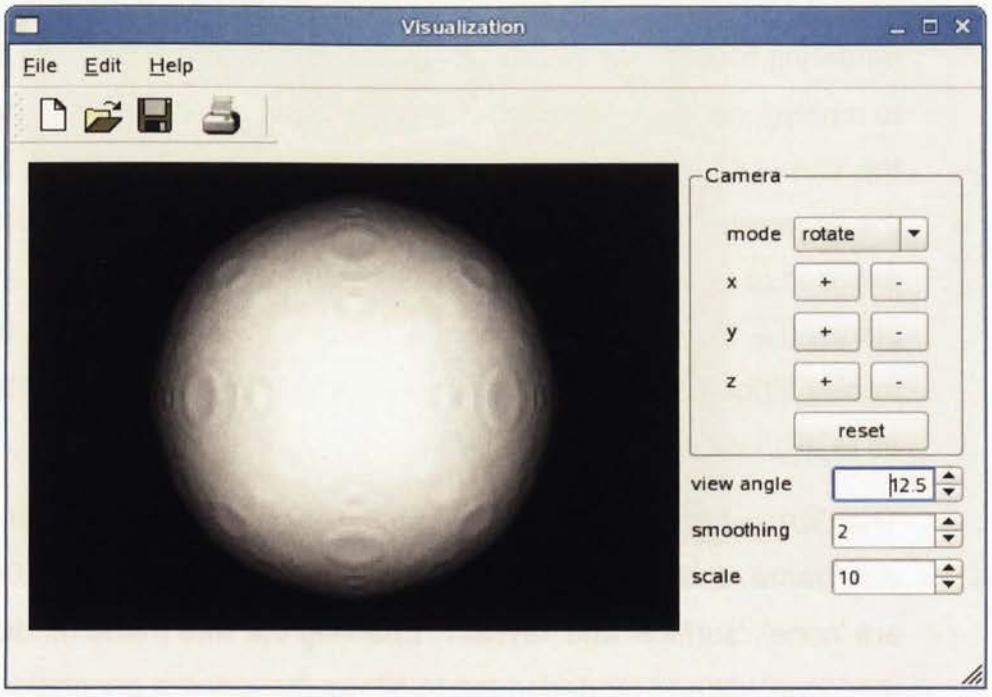


Figure A.1: Rendering of a sphere with 2 smoothing cycles applied to the normals

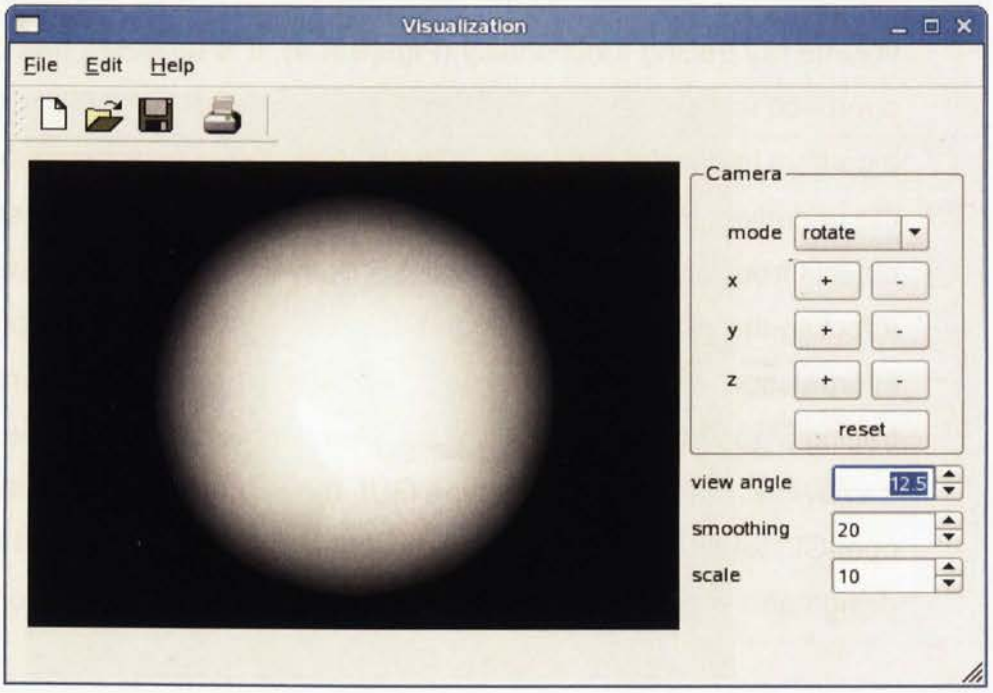


Figure A.2: Rendering of a sphere with 20 smoothing cycles applied to the normals

A.1.2 Ray tracing visualization first prototype

To provide a first use case for the distributed framework, a direct volume ray tracing visualization capability was designed and implemented. Direct volume rendering traditionally works on the basis of ray tracing without the need to convert to a polygonal representation. The ray tracing algorithms incorporate the design of the 'camera' navigation mechanism so that the ray direction could be oriented in the appropriate direction relative to a 3D volume, and other various parameters exposed to allow control of the size of the view port, the view angle, near and far planes (i.e. to compute the openGL model view and projection matrices). The ray tracing algorithm was also designed to allow the distribution of blocks of rays, so as to align with the proposed DFrame design.

The GUI was updated to include the selection of volume render modes. As well, wire frame rendering of the image boundary can be selected. The volume modes are 'none', 'surface' and 'raycast'. Enabling the wire frame rendering, and selecting the 'none' volume rendering mode allows the camera orientation to be set without incurring the processing costs of the ray tracing or iso-surface processing. The 'surface' mode switches in the iso-surface rendering described in the previous section (Figure A.3), while the 'raycast' mode switches in the introduced direct volume ray tracing functionality (Figure A.4). It is apparent that the ray tracing approach is more successful than the current implementation of the iso-surface algorithm in determining and rendering the outline of general objects. However, the underlying iso-surface detection approach should still be useful in delineating objects from the background, and it is likely that the crude normal calculations are affecting the rendered output, and future effort will look at a more exact interpolation (that is already integrated into the ray tracing approach). The ray tracing algorithm has subsequently been parallelised to use the distributed framework. In this early prototype GUI, the camera implementation is tied into openGL, and this functionality has subsequently been adapted to an independent design and implementation suitable for the distributed framework.

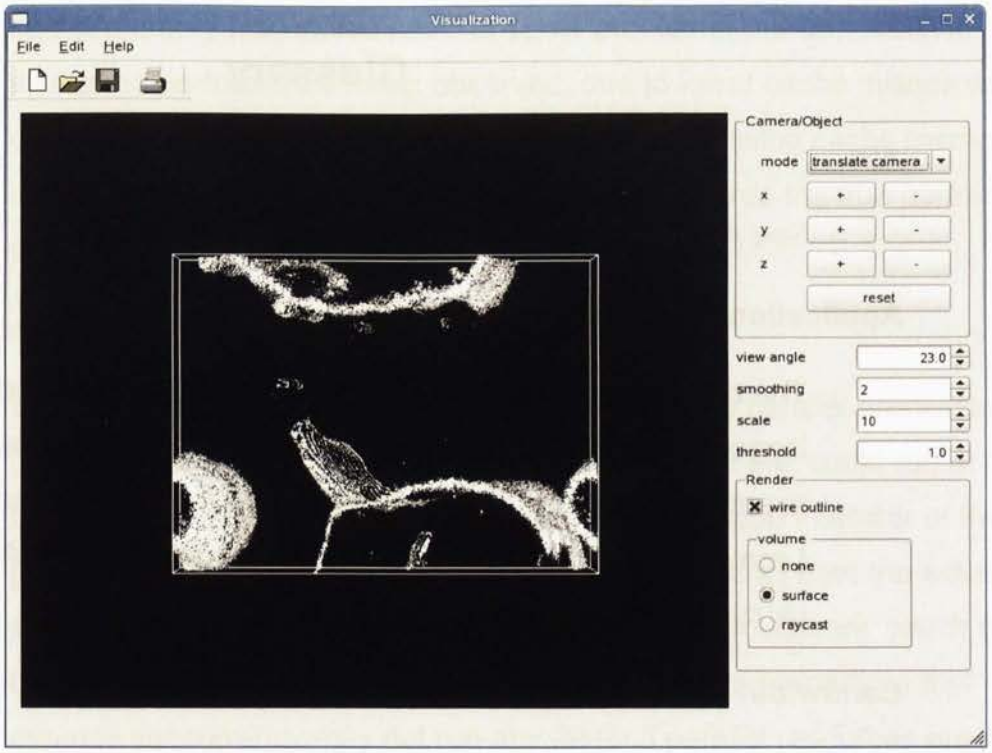


Figure A.3: Prototype GUI visualizing iso-surface of 3D cell biology

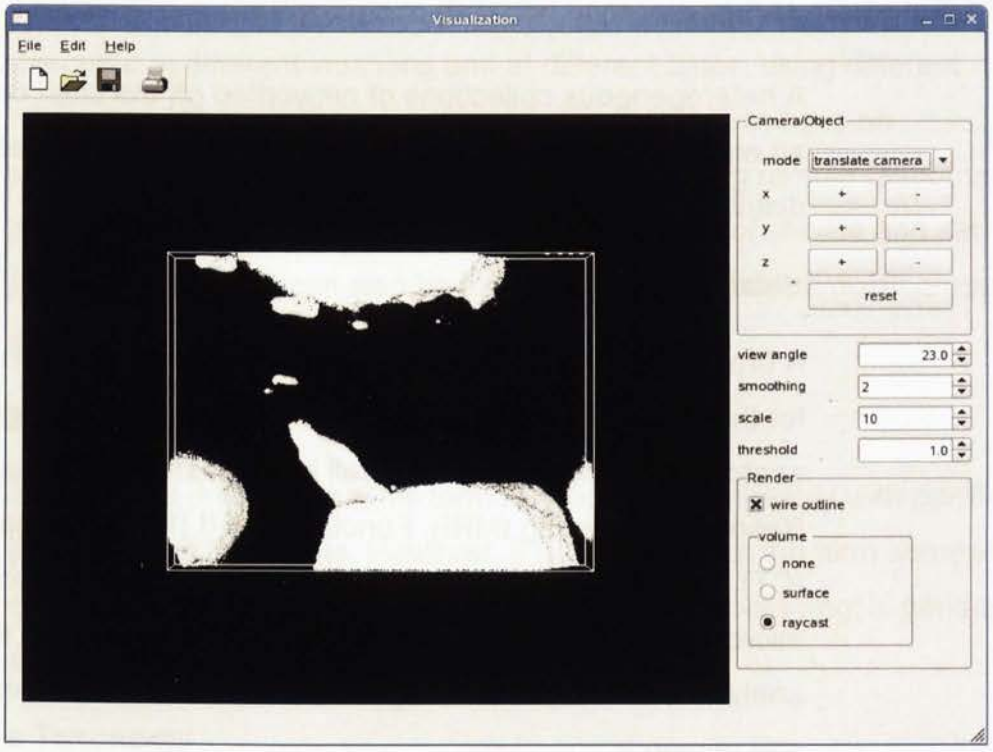


Figure A.4: Prototype GUI visualizing ray tracing of 3D cell biology

Glossary

API

See Application Programming Interface

Application Programming Interface

The collection of functions, (or methods in object oriented programming languages) that expose a software entities external behaviour. Entity examples include software programs and services and software libraries. Less often, software entities also expose state, and the API definition is usually loosened to include this.

Bandwidth

In the context of communications between compute entities, bandwidth refers to the capacity of the interconnecting channels, usually expressed in terms of the number of bits of information that can be transferred per second.

Beowulf Cluster

A heterogeneous collections of networked off the shelf computers, used as an economic alternative to a bespoke supercomputer design to provide for distributed computing.

Biomedical imaging

A broad term describing the development and use of various technologies for the acquisition and representation of 2D and 3D images of the internal anatomy of living organisms. Well known examples include Magnetic Resonance Imaging (MRI), Functional MRI (fMRI), Computed Tomography (CT), Positron Emission Tomography (PET, light microscopy and electron microscopy. The term is usually implicitly understood to extend to the analysis of the captured images for purposes such as health assessment and research.

Cache Aggregation

As the number of available processors is increased, the total processor

cache memory also increases. For some problems, this can result in increased performance being observed, due to lower cache misses when data fits into the increased cache memory, as accessing cache memory is usually much faster than main memory. This obscures the true performance of the memory system.

Concurrency

When assessing whether a program is amenable to parallel processing, a first step is to identify those sections of the program that could run at the same time. The identified concurrency provides a good indicator of the programs suitability in this respect. This is quite distinct from the actual mapping of the identified sections to independent processors, which of course depends on their availability. Indeed, threaded programs that express concurrency may not run any faster if parallel resources are not available. In this case an operating system usually employs a time slicing mechanism to ensure fair sharing of the limited resources.

Distributed Computing

This term is generally used to describe applications that create tasks to be executed at different locations and at different times, using different resources. Whereas the emphasis for parallel processing is on performance, here it is more about robustness using remote resources, with priority given to fault tolerance, availability, quality of service and security. Parallel processing can also be incorporated into a distributed computing system.

Distributed Processing

See parallel processing. This term is used interchangeably with parallel processing in this thesis. However, it is admitted that the term somewhat overlaps with 'distributed computing', so 'parallel processing' is generally the preferred term.

Flynn's Taxonomy

A simple classification of computer architectures (Flynn 1972), according to the number of instruction streams and data streams supported. It defines the following four classes:

Single Instruction Single Data (SIMD). A typical single processor machine.

Single Instruction Multiple Data (SIMD). Vector processors are exemplars of this class.

Multiple Instruction Single Data (MISD). These primarily include pipeline architectures applying multiple instructions to the same data.

Multiple Instruction Multiple Data (MIMD). Refer to the most flexible task parallel systems.

Generic programming

Programming where types can be used as parameters. C++ allows types to be used as parameters to specialize template classes and functions.

High Content Screening

The large scale screening of multiple cell images from specific biological organisms under study. Important objectives being to investigate cell characteristics (e.g. morphology, physiology and motility) and behaviour. For example, aggregate characteristics and behaviour can be determined across a cell population, and time lapsed behaviour of individual cells can be studied.

ILP

See Instruction Level Parallelism

Instruction Level Parallelism

ILP encompasses hardware and software techniques to exploit opportunities for parallelism found in programs at the instruction level. Prominent examples of ILP include the rearrangement of instruction execution order to expose parallelism, concurrent speculative execution (branch prediction). Also see 'pipelining' and 'superscalar execution'.

Latency

The time it takes to initiate communication between compute entities. Whereas bandwidth defines the volume of information that can be transferred per second, latency is a measure of the initial delay before a recipient starts to receive data once it has been transmitted.

Message Passing Interface

A communication protocol specification applicable to parallel processing across systems using a distributed memory architecture, rather than those using common shared memory. Compute entities in these system interact by passing messages to one another rather than through the update of shared memory, and MPI specifies an open and portable interface for this purpose.

MISD

See Flynn's Taxonomy.

MIMD

See Flynn's Taxonomy.

MPI

See Message Passing Interface

Parallel Processing

When multiple processing elements are available, and sections of a program can be run concurrently, then those sections can be mapped to the available processors and actually physically run at the same time. The fundamental advantage being the running of such a program in much reduced timescales. Parallel processing is generally about bringing resources to bear on a particular task at a particular time (c.f. Distributed Computing)

Pipelining

A limited form of parallelism induced by the staging of operations in programs suited to this arrangement. In a fully utilized multi-stage pipeline, each stage is performing the same operation, but on different data. For example, a ten stage pipeline can process ten data streams concurrently. Successful pipelining requires that the stages complete in similar time periods, or pipeline 'bottlenecks' can impact performance. Pipelining is also an important technique at the instruction level (see ILP).

Service Oriented Architecture

A software architecture pattern in which individual components expose their functionality as services to other components via some agreed communication protocol. A service may use multiple other services to provide its function. This architecture is specifically suited to distributed computing environments.

Single Program Multiple Data

A parallel processing model where a single program runs on all participating processes. Although these processes collectively cooperate to execute the program, at any instance in time each process can run a different section of the program, at any instance in time each process can run a different section of the program and operate on different data.

SISD

See Flynn's Taxonomy.

SIMD

See Flynn's Taxonomy.

SPMD

See Single Program Multiple Data

Strong scaling

A measure of the scalability of a program, in circumstances where the data size remains fixed with increasing processor count. This is a common situation for problems that run for a long time (cpu bound), and increasing the number of processors will spread the cpu load and reduce the total running time. Efficiency usually drops off with increased processor counts as communication overhead typically increases as the processor count increases.

Super linear speed up

This describes a performance speed up due to memory aggregation or cache aggregation. As more resources are made available to a problem, memory is often increased commensurately (e.g. when adding another machine to a cluster). If the increased resources removes the need for data to be 'spilt' to disk for some problem, the associated IO costs can be

eliminated, resulting in an observed performance boost. The same effect is observed if a problem can fit in increased cache memory (see cache aggregation).

Superscalar execution

A ILP technique applicable to processors having multiple internal execution units. When a program's data dependencies allow (i.e. parallelism exists), multiple instructions can be arranged to execute during the same clock cycle.

Synchronization

An umbrella term spanning various techniques for controlling access to resources (e.g. thread locking mechanisms) or to control the progress of processes relative to each other (e.g. through a 'barrier' construct). In message passing systems, synchronization is implicitly set up between the message sender and message receiver(s).

Weak scaling

A measure of the scalability of a program, in circumstances where the processor count is increased proportionally with increasing problem size. This situation can be encountered in memory bound problems, when it is necessary to split a larger problem across a larger number of processors, so as to fit into memory caches.

References

ADAPTIVE COMPUTING, 2015-last update, Documentation Index [Homepage of Adaptive Computing], [Online]. Available: http://docs.adaptivecomputing.com2015].

AIDA, K., NATSUME, W. and FUTAKATA, Y., 2003. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm, *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, May 2003, pp. 156-163.

ALDINUCCI, M., ANARDU, L., DANELUTTO, M., TORQUATI, M. and KILPATRICK, P., 2012. Parallel patterns + Macro Data Flow for multi-core programming, *Proc. of Intl. Euromicro PDP 2012: Parallel Distributed and network-based Processing*, feb 2012, IEEE, pp. 27-36.

ALDINUCCI, M., DANELUTTO, M., KILPATRICK, P. and TORQUATI, M., 2012. FastFlow: high-level and efficient streaming on multi-core. In: S. PLLANA and F. XHAFA, eds, *Programming Multi-core and Many-core Computing Systems.* Wiley, .

ALEXANDER, C., ISHIKAWA, S. and SILVERSTEIN, M., 1977. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series).* Later printing edn. Oxford University Press.

AL-GHARAIBEH, J., JEFFERY, C. and OIKONOMOU, K.N., 2012. An hybrid model for very high level threads, *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores* 2012, ACM, pp. 55-63.

AMANATIDES, J. and WOO, A., 1987. A fast voxel traversal algorithm for ray tracing, *Eurographics '87*, aug 1987, pp. 3-10,.

AMDAHL, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities, *Proceedings of the April 18-20, 1967, spring joint computer conference* 1967, ACM, pp. 483-485.

ANGEL, E., 2001a. *Interactive Computer Graphics: A Top-Down Approach With OPENGL primer package-2nd Edition.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

ANGEL, E., 2001b. *Open GL Primer.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

APPEL, A., 1968. Some techniques for shading machine renderings of solids. *AFIPS 1968 Spring Joint Computer Conference,* , pp. 37-45.

ARVIND and IANNUCCI, R.A., 1983. A Critique of Multiprocessing Von Neumann Style, *Proceedings of the 10th Annual International Symposium on Computer Architecture* 1983, ACM, pp. 426-436.

ASANOVIC, K., BODIK, R., CATANZARO, B.C., GEBIS, J.J., HUSBANDS, P., KEUTZER, K., PATTERSON, D.A., PLISHKER, W.L., SHALF, J., WILLIAMS, S.W. and YELICK, K.A., 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* EECS Department, University of California, Berkeley.

ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J.D., LEE, E.A., MORGAN, N., NECULA, G., PATTERSON, D.A., SEN, K., WAWRZYNEK, J., WESSEL, D. and YELICK, K.A., 2008. *The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View.* EECS Department, University of California, Berkeley.

ASTLE, D. and HAWKINS, K., 2004. *Beginning OpenGL Game Programming.* Premier Press.

BACCI, B., DANELUTTO, M., ORLANDO, S., PELAGATTI, S. and VANNESCHI, M., 1995. P3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience,* **7**(3), pp. 225-255.

BACKUS, J., 1978. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun.ACM,* **21**(8), pp. 613-641.

BAL, H.E., KAASHOEK, M.F. and TANENBAUM, A.S., 1992. Orca: a language for parallel programming of distributed systems. *Software Engineering, IEEE Transactions on,* **18**(3), pp. 190-205.

BAL, H.E., STEINER, J.G. and TANENBAUM, A.S., 1989. Programming languages for distributed computing systems. *ACM Comput.Surv.,* **21**(3), pp. 261-322.

BEUCHER, S. and MEYER, F., 1993. The Morphological Approach to Segmentation: The Watershed Transformation. In: E.R. DOUGHERTY, ed, *Mathematical morphology in image processing.* New York: Marcel Dekker, pp. 433-481.

BEUCHER, S. and MARCOTEGUI, B., 2009. *P algorithm, a dramatic enhancement of the waterfall transformation.* MINES ParisTech: Centre de Morphologie Mathématique.

BIENIEK, A., BURKHARDT, H., FREIBURG, A., MARSCHNER, H., SCHREIBER, G., I, T.I. and NOLLE, M., 1997. A Parallel Watershed Algorithm, *In Proc. 10th Scandinavian Conference on Image Analysis (SCIA'97* 1997, pp. 237-244.

BISHOP, C.M., 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

BLANCHETTE, J. and SUMMERFIELD, M., 2008. *C++ Gui Programming with Qt 4, Second Edition.* Second edn. Upper Saddle River, NJ, USA: Prentice Hall Press.

BLUMOFE, R.D., JOERG, C.F., KUSZMAUL, B.C., LEISERSON, C.E., RANDALL, K.H. and ZHOU, Y., 1995. Cilk: An Efficient Multithreaded Runtime System, *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, jul 1995, pp. 207-216.

BONACHEA, D., 2002. *GASNet Specification, V1.1.* Berkeley, CA, USA: University of California at Berkeley.

BROWN, K.J., SUJEETH, A.K., LEE, H.J., ROMPF, T., CHAFI, H., ODERSKY, M. and OLUKOTUN, K., 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages, *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* 2011, IEEE Computer Society, pp. 89-100.

CAHON, S., MELAB, N. and TALBI, E.-., 2004. ParadisEO: A Framework for the

Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics,* **10**(3), pp. 357-380.

CARPENTER, A.E., JONES, T.R., LAMPRECHT, M.R., CLARKE, C., KANG, I.H., FRIMAN, O., GUERTIN, D.A., CHANG, J.H., LINDQUIST, R.A., MOFFAT, J., GOLLAND, P. and SABATINI, D.M., 2006. CellProfiler: image analysis software for identifying and quantifying cell phenotypes. *Genome biology,* 7(10), pp. R100.

CATANZARO, B. and KEUTZER, K., 2010. Parallel computing with patterns and frameworks. *XRDS,* **17**(1),.

CATANZARO, B., KAMIL, S.A., LEE, Y., ASANOVIĆ, K., DEMMEL, J., KEUTZER, K., SHALF, J., YELICK, K.A. and FOX, A., 2010. *SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization.* EECS Department, University of California, Berkeley.

CHAMBERLAIN, B., CHOI, S., HILDEBRANDT, T., LITVINOV, V. and TITUS, G., , The Chapel Parallel Programming Language. Chapel Overview. Available: http://chapel.cray.com [Aug/16, 2012].

CHAMBERLAIN, B.L., CALLAHAN, D. and ZIMA, H.P., 2007. Parallel Programmability and the Chapel Language. *Int.J.High Perform.Comput.Appl.,* **21**(3), pp. 291-312.

CHAMBERLAIN, B.L., CHOI, S., LEWIS, E.C., LIN, C., SNYDER, L. and WEATHERSBY, W.D., 2000. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Trans.Softw.Eng.,* **26**(3), pp. 197-211.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C. and SARKAR, V., 2005. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.,* **40**(10), pp. 519-538.

CHEN, L.T. and BAIRAGI, D., 2010. *Developing Parallel Programs -- A Discussion of Popular Models.* World Headquarters 500 Oracle Parkway Redwood Shores, CA 94065 U.S.A.: Oracle Corporation.

COLE, M., 1991. *Algorithmic skeletons: structured management of parallel computation.* Cambridge, MA, USA: MIT Press.

CRAY INC., , Chapel Language Specification [Homepage of Cray Inc. Seatle, WA], [Online]. Available: http://chapel.cray.com [Aug/16, 2012].

CUTTING, D., 03/19/2012-last update, Welcome to Apache™ Hadoop™!. Available: http://hadoop.apache.org/ [06/14, .

DALHEIMER, M.K., 1999. *Programming with Qt.* Cambridge: O'Reilly.

DAREMA, F., 2011. SPMD Computational Model. In: D. PADUA, ed, *Encyclopedia of Parallel Computing.* Springer US, pp. 1933-1943.

DEAN, J. and GHEMAWAT, S., 2008. MapReduce: simplified data processing on large clusters. *Commun.ACM,* **51**(1), pp. 107-113.

DEELMAN, E., SINGH, G., SU, M., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA,

G., VAHI, K., BERRIMAN, G.B., GOOD, J., LAITY, A., JACOB, J.C. and KATZ, D.S., 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci.Program.,* **13**(3), pp. 219-237.

DIAZ, J., MUNOZ-CARO, C. and NINO, A., 2012. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems,* **23**, pp. 1369-1386.

DINAN, J., BALAJI, P., LUSK, E., SADAYAPPAN, P. and THAKUR, R., 2010. Hybrid parallel programming with MPI and unified parallel C, *Proceedings of the 7th ACM international conference on Computing frontiers* 2010, ACM, pp. 177-186.

DONGARRA, J., GRAYBILL, R., HARROD, W., LUCAS, R., LUSK, E., LUSZCZEK, P., MCMAHON, J., SNAVELY, A., VETTER, J., YELICK, K., ALAM, S., CAMPBELL, R., CARRINGTON, L., CHEN, T., KHALILI, O., MEREDITH, J. and TIKIR, M., 2008. DARPA's \HPCS\ Program: History, Models, Tools, Languages. In: MARVIN V. ZELKOWITZ, ed, *Advances in COMPUTERS High Performance Computing.* Elsevier, pp. 1.

DOXSEY, C., 2012. *An introduction to programming in Go.* CreateSpace Independent Publishing Platform.

EBBERS, M., DE SOUZA, R.G., LIMA, M.C., MCCULLAGH, P. and NOBLES, M., 2013. *Implementing IBM InfoSphere BigInsights on IBM System x.* 2nd edn. Vervante.

FARBER, R., 2011. *CUDA Applocation design and development.* 1 edn. Morgan Kaufmann.

FIELDING, R.T., 2000. *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine.

FLEISCH, B. and POPEK, G., 1989. Mirage: A Coherent Distributed Shared Memory Design. *SIGOPS Oper.Syst.Rev.,* **23**(5), pp. 211-223.

FLYNN, M., J, 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers,* **C**(21), pp. 948-960.

FOWLER, M., 2006-last update, GUI Architectures - Model View Controller [Homepage of Thoughtworks, inc], [Online]. Available: http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController2015].

FOWLER, M., 2004-last update, Inversion of Control Containers and the Dependency Injection pattern [Homepage of Thoughtworks, inc], [Online]. Available: http://www.martinfowler.com/articles/injection.html2015].

FREY, J., TANNENBAUM, T., LIVNY, M., FOSTER, I. and TUECKE, S., 2002. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing,* **5**(3), pp. 237-246.

GAGNE, C., PARIZEAU, M. and DUBREUIL, M., 2003. Distributed Beagle: An Environment for Parallel and Distributed Evolutionary Computations, *Proceedings of the 17 th Annual International Symposium on High Performance Computing Systems and Applications (HPCS* 2003.

GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J., 1995. *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

GAREY, M.R. and JOHNSON, D.S., 1997. *Computers and intractability : A guide to the theory of NP-completeness.* San Fransisco: W. H. Freeman and Company.

GASTER, B., HOWES, L., KAELI, D.R., MISTRY, P. and SCHAA, D., 2013. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition.* 2 edn. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

GEIST , A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R. and SUNDERAM, V., 1994. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* Cambridge, MA, EUA: MIT Press.

GONZÁLEZ-VÉLEZ, H. and LEYTON, M., 2010. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience,* **40**(12), pp. 1135-1160.

GOSLING, J., JOY, B., STEELE, G., BRACHA, G. and BUCKLEY, A., *The Java language specification.* Java SE 7 edn.

GOUX, J., KULKARNI, S., YODER, M. and LINDEROTH, J., 2000. An Enabling Framework for Master-Worker Applications on the Computational Grid, *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing* 2000, IEEE Computer Society, pp. 43.

GOUX, J., LINDEROTH, J., YODER, M., LINDEROTH, J.G.\.J. and YODER, D.M., 2000. Metacomputing and the Master-Worker Paradigm, *Preprint MCS/ANL-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne* 2000.

GRAMA, A.Y., GUPTA, A. and KUMAR, V., 1993. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel Distrib.Technol.,* **1**(3), pp. 12-21.

GREGOR, D. and LUMSDAINE, A., 2005. The parallel bgl: A generic library for distributed graph computations, *In Parallel Object-Oriented Scientific Computing (POOSC* 2005.

GROPP, W., LUSK, E. and SKJELLUM, A., 1999a. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface.* Cambridge, MA, USA: MIT Press.

GROPP, W., LUSK, E. and THAKUR, R., 1999b. *Using MPI-2: Advanced Features of the Message-Passing Interface.* Cambridge, MA, USA: MIT Press.

GSCHWIND, M., 2006. Chip multiprocessing and the cell broadband engine, *CF '06: Proceedings of the 3rd conference on Computing frontiers* 2006, ACM, pp. 1-8.

GU, M., 1996. *Principles of three-dimensional imaging in confocal microscopes.* Singapore ; River Edge, NJ: World Scientific.

GUSTAFSON, J.L., 1988. Reevaluating Amdahl's law. *Communications of the ACM,*

31(5), pp. 532-533.

HAGGLUND, S., HOPPE, A., AUBYN, D., CAVANNA, T., JORDAN, P. and ZICHA, D., 2009. Novel shear flow assay provides evidence for non-linear modulation of cancer invasion. *Frontiers in bioscience (Landmark edition),* **14**, pp. 3085-3093.

HALLER, P. and ODERSKY, M., 2007. Actors that unify threads and events, *Proceedings of the 9th international conference on Coordination models and languages* 2007, Springer-Verlag, pp. 171-190.

HEATHER KREGER IBM and JEFF ESTEFAN NASA/JET PROPULSION LABORATORY, 2009-last update, Navigating the SOA Open Standards Landscape Around Architecture [Homepage of Object Management Group], [Online]. Available: https://www2.opengroup.org/ogsys/catalog/w096.

HEGE, H.C., HOLLERER, T. and STALLING, D., 1993. *Volume Rendering Mathematical Models and Algorithmic Aspects.* TR 93-7. Berlin, Germany: Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB).

HENDERSON, R., 1995. Job scheduling under the Portable Batch System. **949**, pp. 279-294.

HEWITT, C., BISHOP, P. and STEIGER, R., 1973. A universal modular ACTOR formalism for artificial intelligence, *Proceedings of the 3rd international joint conference on Artificial intelligence* 1973, Morgan Kaufmann Publishers Inc, pp. 235-245.

HILL, F.S.J., ed, 2001. *Computer Graphics using open GL.* 2nd edn. Upper Saddle River, NJ 07458: Prentice Hall.

HOARE, C.A.R., 1985. *Communicating sequential processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

HOARE, C.A.R., 1978. Communicating sequential processes. *Communications of the ACM,* **21**(8), pp. 666-677.

HURSON, A.R. and KAVI, K.M., 2008. Dataflow Computers: Their History and Future. *Wiley Encyclopedia of Computer Science and Engineering.*

IBM, R., October 14, 2007, 2007-last update, OpenDx [Homepage of OpenDx.org & IBM Research], [Online]. Available: http://www.opendx.org/index2.php [May, 2008].

IBM, R., 2014, , Parallel Environment Runtime Edition Version 2 Release 1. MPI Programming Guide.

JIN, G., LI, Z. and CHEN, F., 2001. A theoretical foundation for program transformations to reduce cache thrashing due to true data sharing. *Theoretical Computer Science,* **255**(1–2), pp. 449.

JOERG, C.F., 1996. *The Cilk System for Parallel Multithreaded Computing*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

JOHANSEN, M.F., 2009. *Domain Specific Languages versus Frameworks*, UNIVERSITY OF OSLO Department of Informatics.

JOHNSON, R., HOELLER, J., DONALD, K., SAMPALEANU, C., HARROP, R., RISBERG, T., ARENDSEN, A., DAVISON, D., KOPYLENKO, D., POLLACK, M., TEMPLIER, T., VERVAET, E., TUNG, P., HALE, B., COLYER, A., LEWIS, J., LEAU, C., FISHER, M., BRANNEN, S., LADDAD, R., POUTSMA, A., BEAMS, C., ABEDRABBO, T., CLEMENT, A., SYER, D., GIERKE, O., STOYANCHEV, R. and WEBB, P., 2013-last update, Spring Framework Reference Documentation [Homepage of Spring Framework], [Online]. Available: http://docs.spring.io/autorepo/docs/spring/3.2.x/spring-framework-reference/html/index.html2015].

JOHNSON, R.E., 1997. Frameworks = (components + patterns). *Commun.ACM,* **40**(10), pp. 39-42.

JOSEY, A., 5th October 2011, 2011-last update, POSIX 1003.1 FAQ [Homepage of Austin Group], [Online]. Available: http://www.opengroup.org/austin/papers/posix_faq.html [Aug/22, 2012].

KARP, A.H. and FLATT, H.P., 1990. Measuring Parallel Processor Performance. *Commun.ACM,* **33**(5), pp. 539-543.

KEKEC, B., 2010. *EFFECTS OF PARALLEL PROGRAMMING DESIGN PATTERNS ON THE PERFORMANCE OF MULTI-CORE PROCESSOR BASED REAL TIME EMBEDDED SYSTEMS*, THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF MIDDLE EAST TECHNICAL UNIVERSITY.

KERNIGHAN, B.W., 1988. *The C Programming Language.* 2nd edn. Prentice Hall Professional Technical Reference.

KEUTZER, K. and MATTSON, T., , A Pattern Language for Parallel Programming ver2.0. Available: http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.

KJOLSTAD, F.B. and SNIR, M., 2010. Ghost Cell Pattern, *Proceedings of the 2010 Workshop on Parallel Programming Patterns* 2010, ACM, pp. 4:1-4:9.

KLETTE, R., 2014. *Concise Computer Vision An Introduction into Theory and Algorithms.* London: Springer London.

KNELL, R.J., 2013. *Introductory R: A Beginner's Guide to Data Visualisation and Analysis using R.*

KUHN, R., ANTONSSON, B., MALAWSKI, K., NORDWALL, P. and VARGA, E., , AKKA. Available: http://akka.io [Aug 30, 2014].

KUMAR, V., GRAMA, A., GUPTA, A. and KARPIS, G., *Introduction to Parallel Computing.* 2 edn. Harlow: Pearson Addison Wesley.

LARA, R., MAURI, F.A., TAYLOR, H., DERUA, R., SHIA, A., GRAY, C., NICOLS, A., SHINER, R.J., SCHOFIELD, E., BATES, P.A., WAELKENS, E., DALLMAN, M., LAMB, J., ZICHA, D., DOWNWARD, J., SECKL, M.J. and PARDO, O.E., 2011. An siRNA screen identifies RSK1 as a key modulator of lung cancer metastasis. *Oncogene,* **30**(32), pp. 3513-3521.

LAUER, H.C. and NEEDHAM, R.M., 1979. On the duality of operating system structures. *SIGOPS Oper.Syst.Rev.,* **13**(2), pp. 3-19.

LEA, D., 2000a. A Java Fork/Join Framework, *Proceedings of the ACM 2000 Conference on Java Grande* 2000a, ACM, pp. 36-43.

LEA, D., 2000b. *Concurrent programming in Java : design principles and patterns.* 2nd edn. Reading, Mass. ; Harlow: Addison-Wesley.

LEE, B. and HURSON, A.R., 1993. Issues in dataflow computing. *ADV.IN COMPUT,* **37**, pp. 285-333.

LEE, E.A., 2006. The Problem with Threads. *Computer,* **39**(5), pp. 33-42.

LEOPOLD, C., 2001. *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches.* New York, NY, USA: John Wiley \& Sons, Inc.

LEVOY, M., 1990. Efficient ray tracing of volume data. *ACM Trans.Graph.,* **9**(3), pp. 245-261.

LI, K. and HUDAK, P., 1989a. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans.Comput.Syst.,* **7**(4), pp. 321-359.

LI, K. and HUDAK, P., 1989b. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans.Comput.Syst.,* **7**(4), pp. 321-359.

LIANG, T.-., LI, H.-. and CHIU, J.-., 2012. Enabling mixed OpenMP/MPI Programming on hybrid CPU/GPU computing architecture. , pp. 2369-2377.

LIOTTA, L.A., 1992. Cancer cell invasion and metastasis. *Sci Am,* **266**, pp. 54-59, 62-63.

LOO, L.H.W. and L. F. ALTSCHULER, S. J., 2007. Image-based multivariate profiling of drug responses from single cells. *NATURE METHODS,* **VOL 4**(NUMBER 5), pp. 445-453.

LORENSEN, W.E. and CLINE, H.E., 1987. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput.Graph.,* **21**(4), pp. 163-169.

MAASSEN, J., 2001. Efficient Java RMI for Parallel Programming. *ACM Trans.Program.Lang.Syst.,* **23**(6), pp. 747-775.

MACDONALD, S., 2002. *From Patterns to Frameworks to Parallel Programs*, University of Alberta.

MACDONALD, S., ANVIK, J., BROMLING, S., SCHAEFFER, J., SZAFRON, D. and TAN, K., 2002. From patterns to frameworks to parallel programs. *Parallel Comput.,* **28**(12), pp. 1663-1683.

MALEWICZ, G., AUSTERN, M.H., BIK, A.J.C., DEHNERT, J.C., HORN, I., LEISER, N. and CZAJKOWSKI, G., 2010. Pregel: A System for Large-scale Graph Processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* 2010, ACM, pp. 135-146.

MAO, Y., TOURNIER, A.L., HOPPE, A., KESTER, L., THOMPSON, B.J. and TAPON, N., 2013. Differential proliferation rates generate patterns of mechanical tension that orient tissue growth. *The EMBO journal,* **32**(21), pp. 2790-2803.

MATTSON, T., SANDERS, B. and MASSINGILL, B., 2004. *Patterns for parallel programming.* First edn. Addison-Wesley Professional.

MATTSSON, M. and BOSCH, J., 1997. Framework composition: problems, causes and solutions, *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. Proceedings* 1997, pp. 203-214.

MAY, D., 1983. OCCAM. *SIGPLAN Not.,* **18**(4), pp. 69-79.

MCKEE, S.A., 2004. Reflections on the memory wall, *CF '04: Proceedings of the 1st conference on Computing frontiers* 2004, ACM, pp. 162.

MEYER, T. and HART, I.R., 1998. Mechanisms of tumour metastasis. *Eur J Cancer,* **34**, pp. 214-221.

MILNER, R., 1982. *A Calculus of Communicating Systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

MILNER, R., 1999. *Communicating and Mobile Systems: The \&Pgr;-calculus.* New York, NY, USA: Cambridge University Press.

MILNER, R., 1995. *Communication and concurrency.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd.

MOGA, A.N. and GABBOUJ, M., 1998. Parallel Marker-based Image Segmentation with Watershed Transformation. *J.Parallel Distrib.Comput.,* **51**(1), pp. 27-45.

MOLNAR, S., COX, M., ELLSWORTH, D. and FUCHS, H., 2008. A sorting classification of parallel rendering, *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses* 2008, ACM, pp. 1-11.

MORRIS, T., 2004. *Computer Vision and Image Processing.* Palgrave Macmillan Limited.

N. MOGA, A., CRAMARIUC, B. and GABBOUJ, M., 1998. Parallel watershed transformation algorithms for image segmentation. *Parallel Computing,* **24**(14), pp. 1981-2001.

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH, 2015, , Parallel Computing Concepts.

NEBEL, J.C., 1998. A New Parallel Algorithm Provided by a Computation Time Model. Eurographics Workshop on Parallel Graphics and Visualisation, *Eurographics Workshop on Parallel Graphics and Visualisation,* 24-25 September 1998 1998.

NEUMANN, J.V., 2000. *The Computer and the Brain.* 2nd edn. New Haven, CT, USA: Yale University Press.

NICOLESCU, C. and JONKER, P., 2002. A data and task parallel image processing environment. *Parallel Comput.,* **28**(7-8), pp. 945-965.

NIKOLAIDIS, N. and PITAS, I., 2001. *3-D Image Processing Algorithms.* New York, NY, USA: John Wiley & Sons, Inc.

NISHTALA, R., ZHENG, Y., HARGROVE, P.H. and YELICK, K.A., 2011. Tuning

collective communication for Partitioned Global Address Space programming models. *Parallel Computing,* **37**(9), pp. 576-591.

NITZBERG, B. and LO, V., 1991. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer,* **24**(8), pp. 52-60.

ODERSKY, M., SPOON, L. and VENNERS, B., 2010. *Programming in Scala.* 2 edn. Artima.

OPEN GROUP (READING, E., 2013. *Base Specifications, Issue 7.* The Open Group.

OPENGL, 2008, 2008-last update, GLUT - The OpenGL Utility Toolkit [Homepage of opengl.org], [Online]. Available: http://www.opengl.org/resources/libraries/glut/ [01/15, 2008].

OPENGL ARCHITECTURE REVIEW BOARD and SHREINER, D., 2004. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4.* 4 edn. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

OPENMP, A.R.B., 2011. *OpenMP Application Program Interface.* Version 3.1.

PEREZ, J.M.M. and PASCAU, J., 2013. *Image Processing with ImageJ.* 1 edn. Birmingham, UK: Packt Publishing.

PETERKA , T., YU, H., ROSS, R. and MA, K., 2008. Parallel Volume Rendering on the IBM Blue Gene/P. , pp. 73-80.

QUINN, M.J., 2003. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Education Group.

RAZDAN , A., PATEL, K., FARIN , G., E. and CAPCO, D.G., 2001. Volume visualization of multicolor laser confocal microscope data. *Computers and Graphics,* **25**(3), pp. 371-382.

REINDERS, J., 2007. *Intel threading building blocks.* First edn. Sebastopol, CA, USA: O'Reilly \& Associates, Inc.

SAMETINGER, J., 1997. *Software Engineering with Reusable Components.* New York, NY, USA: Springer-Verlag New York, Inc.

SAMUEL H. FULLER, LYNETTE I. MILLETT, E., COMMITTEE ON SUSTAINING GROWTH IN COMPUTING PERFORMANCE and NATIONAL RESEARCH COUNCIL, 2011. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press.

SCHLING, B., 2011. *The Boost C++ Libraries.* XML Press.

SCHROEDER, W., MARTIN, K., AVILA, L., BARRE, S., BLUE, R., GEVECI, B., HENDERSON, A., HOFFMAN, W., KING, B. and LAW, C., 2004a. *The VTK User's Guide, Version 4.4.* {Kitware Inc.}.

SCHROEDER, W., MARTIN, K. and LORENSEN, B., 2004b. *The Visualization Toolkit, Third Edition. An Object Oriented Approach to 3D Graphics*
. Third Edition edn. USA: {Kitware Inc.}.

SEINSTRA, F.J., KOELMA, D. and GEUSEBROEK, J.M., 2002. A software architecture for user transparent parallel image processing. *Parallel Comput.*, **28**(7-8), pp. 967-993.

SERRA, J.P., 1982. *Image analysis and mathematical morphology.* Academic Press.

SHI, Y., 1996. *Reevaluating Amdahl's Law and Gustafson's Law.* Temple University.

SIEWERT, S., 21/12/2009, , Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms.

SINGH, A., SCHAEFFER, J. and GREEN, M., 1991. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Trans.Parallel Distrib.Syst.*, **2**(1), pp. 52-67.

SIU, S., SIMONE, M.D., GOSWAMI, D. and SINGH, A., 1996. Design Patterns for Parallel Programming, HAMID R. ARABNIA, ed. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1996, August 9-11, 1996, Sunnyvale, California, USA* 1996, CSREA Press, pp. 230-240.

SOBEL, I.E., 1970. *Camera Models and Machine Perception*, Stanford University.

SONKA, M., HLAVAC, V. and BOYLE, R., 2008. *Image Processing, Analysis, and Machine Vision.* Third Edition edn. United States of America: Thomson-Engineering.

STROUSTRUP, B., 2000. *The C++ Programming Language.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

SU, B., CATANZARO, B., LAI, C.(., GONINA, E., CHONG, J., KEUTZER, K., MURPHY, M., MOSKEWICZ, M., ANDERSON, M. and SUNDARAM, N., 2009-last update, The Parallel Computing Laboratory, Berkeley, University of California [Homepage of The regents of the University of California], [Online]. Available: http://parlab.eecs.berkeley.edu/research/pallas [20/04, .

TAGLIASACCHI, A., BEST, M.J., DICKIE, R., FEDOROVA, A., COUTURE-BEIL, A. and BROWNSWORD, A., *Cascade: A Parallel Programming Framework for Video Game Engines.*

TANENBAUM, A.S. and STEEN, M.V., 2006. *Distributed Systems: Principles and Paradigms.* 2nd edn. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

TERBOVEN, C., SCHMIDL, D., CRAMER, T. and MEY, D., 2012. Task-Parallel Programming on NUMA Architectures. **7484**, pp. 638-649.

THAIN, D., TANNENBAUM, T. and LIVNY, M., 2005. Distributed Computing in Practice: The Condor Experience: Research Articles. *Concurr.Comput.: Pract.Exper.*, **17**(2-4), pp. 323-356.

UK, C.R., 23/04/2013,26/07/2012, , Lifetime Risk of Cancer [Homepage of Cancer Research UK], [Online]. Available: http://www.cancerresearchuk.org/cancer-info/cancerstats/incidence/risk/statistics-on-the-risk-of-developing-cancer#Lifetime.

VALIANT, L.G., 1990. A bridging model for parallel computation. *Commun.ACM*, **33**(8), pp. 103-111.

VINCENT, L. and SOILLE, P., 1991. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* **13**(6), pp. 583-598.

WALL, D.W., 1991. Limits of instruction-level parallelism, *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* 1991, ACM, pp. 176-188.

WEILAND, M., 2007,. *Chapel , Fortress and X10 : novel languages for HPC.* UoE HPCx Ltd.

WELCH, P.H., 2012, 2012-last update, OccamPiReference [Homepage of Computing Laboratory, University of Kent at Canterbury, CT2 7NF], [Online]. Available: https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference [Aug/30, 2014].

WIKIPEDIA, 2015, 2015-last update, Sobel Operator [Homepage of Wikipedia], [Online]. Available: https://en.wikipedia.org/wiki/Sobel_operator [May, 2014].

YELICK, K., HILFINGER, P., GRAHAM, S., BONACHEA, D., SU, J., KAMIL, A., DATTA, K., COLELLA, P. and WEN, T., 2007. Parallel Languages and Compilers: Perspective From the Titanium Experience. *Int.J.High Perform.Comput.Appl.,* **21**(3), pp. 266-290.

ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M.J., SHENKER, S. and STOICA, I., 2010. Spark: Cluster Computing with Working Sets, *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* 2010, USENIX Association, pp. 10-10.

ZAKI, O., LUSK, E., GROPP, W. and SWIDER, D., 1999. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications,* **13**(2), pp. 277-288.

ZÜLLIGHOVEN, H., 2004. *Object-Oriented Construction Handbook : Developing Application-Oriented Software with the Tools & Materials Approach.* Morgan Kaufmann.

BRITISH LIBRARY

IMAGING SERVICES NORTH
Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

ORIGINAL COPY TIGHTLY BOUND