INTERACTIVE DIGITAL PHOTOGRAPHY

AT SCALE

by

Brian Mark Summa

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2013

Copyright © Brian Mark Summa 2013

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Brian Mark Summa**

has been approved by the following supervisory committee members:

Valerio Pascucci	, Chair	**2/25/2013** Date Approved
Paolo Cignoni	, Member	**12/13/2012** Date Approved
Charles Hansen	, Member	**12/14/2012** Date Approved
Christopher Johnson	, Member	**12/14/2012** Date Approved
Paul Rosen	, Member	**12/14/2012** Date Approved

and by **Alan Davis**, Chair of the Department of **School of Computing**

and by Donna M. White, Interim Dean of The Graduate School.

ABSTRACT

Interactive editing and manipulation of digital media is a fundamental component in digital content creation. One media in particular, digital imagery, has seen a recent increase in popularity of its large or even massive image formats. Unfortunately, current systems and techniques are rarely concerned with scalability or usability with these large images. Moreover, processing massive (or even large) imagery is assumed to be an off-line, automatic process, although many problems associated with these datasets require human intervention for high quality results. This dissertation details how to design interactive image techniques that scale. In particular, massive imagery is typically constructed as a seamless mosaic of many smaller images. The focus of this work is the creation of new technologies to enable user interaction in the formation of these large mosaics. While an interactive system for all stages of the mosaic creation pipeline is a long-term research goal, this dissertation concentrates on the last phase of the mosaic creation pipeline – the composition of registered images into a seamless composite. The work detailed in this dissertation provides the technologies to fully realize interactive editing in mosaic composition on image collections ranging from the very small to massive in scale.

To Delila

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation was made possible through the support of others who I'd like the thank:

First, I would like to thank my family whose endless support made this work possible. Thank you Mom, Dad, Chris, Jason and Amira for your wholehearted support of my decision to quit my job and move across the country to go back to school (in Utah of all places). Most importantly, I would like to thank Delila, my partner in all of this, for taking this adventure with me. You are the source of my inspiration in all things.

I would like to thank my advisor and mentor, Valerio Pascucci, for his continued guidance and encouragement. We both took a very big chance in our first semester as student and professor at Utah, which, I believe, reaped a very huge reward. Since the start of my work, he has given me the support and confidence I needed to succeed and for this I am grateful. I hope this dissertation is not the end, but the beginning of a long collaboration.

I would also like to thank the other members of my committee: Chris, Chuck, Paul and Paolo for their feedback on this work. I would like to especially thank Krzysztof Sikorski who was a member of my committee for most of my time at Utah. Despite his illness, he was a constant source of guidance and encouragement, for which I am grateful.

Finally, I would like to thank the many collaborators I have had while here at Utah, and I hope you forgive me for not writing the numerous list. This work was only possible through these collaborations. I cannot express the perpetual astonishment on what we were/are able to accomplish together.

CHAPTER 1

MOTIVATION AND CONTRIBUTIONS

Interactive editing and manipulation of digital media is a fundamental component in digital content creation. One media in particular, digital imagery, has seen a recent increase in popularity of its large or even massive image formats. Unfortunately, current systems and techniques are rarely concerned with scalability or usability with these large images. For example, the support for large imagery in the most prevalent interactive image editing application, Adobe Photoshop^{TM}, lacks true viability for today's massive images. The application's large image format has a 90 gigapixel maximum image size, limited editing functionality beyond 900 megapixel, and a tedious processing time during an interactive session. Moreover, the creation and processing of large imagery is assumed to be an offline, automatic process though many of the problems associated with these datasets require human intervention for repair. The work outlined in this dissertation will show that this expensive, offline assumption need not be true and that real-time interaction provides new and powerful environments for the creation and editing of massive images. Specifically, this work will detail how to design interactive image processing algorithms that scale.

There has always been an inherent human desire to document or replicate large vistas of our natural world or to document historical events in detail. Panoramic paintings reached the height of their popularity in the early 19th century due to improvements in perspective drawing techniques. A few decades later, the advent of modern photography was closely followed by the earliest work in the creation of panoramic images, see Figure 1.1. In the years since, the popularity of panoramas has not waned, see Figure 1.2. These large, sweeping images capture the feeling of being an observer, whether it is of a beautiful natural view, a historic event such as a Presidential inauguration,[1] or a reminder of the destruction of war.[2] Consequently, there exists a significant interest in creating and using large mosaics

[1]Barak Obama Presidential Inauguration: http://gigapan.org/gigapans/15374/

[2]Hiroshima Panorama Project: http://www.iwu.edu/ rwilson/hiroshima/

Figure 1.1: A 360 degree panorama taken of the city of Toronto, Ontario, Canada credited to Armstrong, Beere and Hime in 1856.

Figure 1.2: Massive imagery is typically constructed as a mosaic of many smaller images. (a) A panorama of Salt Lake City comprised of 624 individual images. The combined image is over 3.2 gigapixels in size. (b) The panorama after being composited into a single seamless image.

for personal, scientific, and/or commercial applications. Examples include medical imaging, where electron microscopy data is composited into ultra-high resolution images [159] or the study of phenology and genomics.[3] Massive imagery is also common in geographic information systems (GIS) in the form of aerial or satellite data and used for anything from urban planning to global climate research.

While many-megapixel cameras do exist,[4] they are overly expensive and unwieldy to use. Therefore, massive imagery is typically constructed as a mosaic of many smaller images.

[3]GigaVision Project: http://www.gigavision.org/

[4]Seitz 6x17 Digital: http://www.roundshot.ch/xml_1/internet/de/application/d438/d925/f934.cfm

At one time, images such as the one in Figure 1.1 were painstakingly constructed by hand. Recent innovations in algorithms and available hardware have drastically simplified the creation of small-scale panoramas. This process can be computed simply offline and can now be embedded in commodity cameras (e.g., the Sony Cyber-shot 3D Sweep Panorama) or mobile devices such as Apple's iPhone. The panoramas for these algorithms are assumed to be small and therefore, are not designed to scale. For example, the iPhone's panorama feature, released in September 2012, has a strict 28 megapixel upper limit on the panorama size. This is small by today's standards. An online search for the word "gigapixel" powerfully demonstrates the increasing desire to create ever larger panoramas. To date, the largest panorama contains roughly 272 gigapixel, yet if the current trend continues, this record is bound to be broken within a few months. This trend is aided by the introduction of new, high-resolution image sensors. For example, with current state-of-the-art 36.3 active megapixel CMOS sensors, it would take as little as 70 images to produce a gigapixel panorama.

Creating panoramas at large scales has become exponentially more difficult than the simple, small cases for which panorama techniques were originally designed. For example, the 3.2 gigapixel panorama shown in Figure 1.2 took several hours to capture and an order of magnitude more time to process on conventional hardware. Furthermore, this timeline assumes a perfect capture and one-time processing. In practice, the process of setting up an automated camera is complicated and error prone and often problems, such as unanticipated occlusion or global misalignment occur. Unfortunately, many of these issues are only subtly expressed in the individual images and become apparent only after the creation of the final panorama. Additionally, today's processing pipelines are less than ideal and typically involve a large number of interdependent and unintuitive parameter choices. A mistake or unlucky choice in the setup can easily cause unacceptable artifacts in the image requiring a repeat of the process. Consequently, it may take several weeks and significant computational resources to produce one large-scale panorama. This makes it difficult for all but a select few to create such images and makes this imagery impractical for many interesting applications. For example, acquiring imagery from unusual locations such as national parks, or covering transient events like an aurora, becomes a significant logistical and monetary challenge. Furthermore, in security applications waiting hours or days for viable results defeats the primary purpose of acquiring the images. Finally, in scientific applications, while typically less time constrained, the personal and computational

resources necessary to create a large-scale image of the night sky, for example, are beyond the reach of all but the largest projects. Therefore, despite significant interest, creating these massive images remains an esoteric hobby or a closely guarded research project.

Work must be done to close the gap between the desire to create large-scale panoramas (and their potential applications) and the ability to capture, process, and utilize such imagery. An ideal panorama system should allow a user to browse individual images as they are acquired, to setup and preview the processing pipeline and results through accurate, real-time approximations, and include a flexible and scalable offline component to produce a final image. The system should be divided into two components: First, a real-time framework to supervise and steer the acquisition process and to guide the postprocessing; and second, a computational back-end based on a distributed or cloud computing framework. The real-time system should be able to run on devices as small as an iPad or netbook computer and be designed to be used in the field to detect any problems as early as possible. The back-end fills the gap between commonly available but slow commodity hardware and specialized distributed computing resources. By implementing a flexible framework able to run on a wide variety of heterogeneous systems, the back-end scales gracefully between a single multicore machine, or a small cluster, to more powerful hardware. My dissertation research has been the creation of technologies to aid in the creation of such a system.

1.1 The Panorama Creation Pipeline

Creating large-scale panoramas can be divided into three stages: *acquisition*, *image registration*, and *composition*. See Figures 1.3 and 1.4. Each stage individually has been the focus of a large amount of research but little effort has been spent on real-time performance or their interdependence. Traditionally all three stages are treated as separate postprocesses, making performance or pipelining a secondary consideration. However, as discussed above, this approach is rapidly becoming unsustainable as long acquisition and processing times lead to errors as well as an increased number of hardware and software failures. To this end, my research goals are and have been to develop algorithms for each stage that produce high quality approximations in real-time and provide a scalable infrastructure to create full solutions exploiting all available hardware. Such new technologies would enable a wide variety of applications currently infeasible. For example, the ability to quickly and cheaply produce high resolution images of art galleries, historical events, or national parks would

Figure 1.3: The three stages of panorama (mosaic) creation. First, the individual images must be acquired. Second, they are registered into a common coordinate system. Third, they are composited (blended) into a single seamless image. My dissertation research has been to provide technologies to enable interactivity for the final composition stage while a high-performance back-end provides a final image.

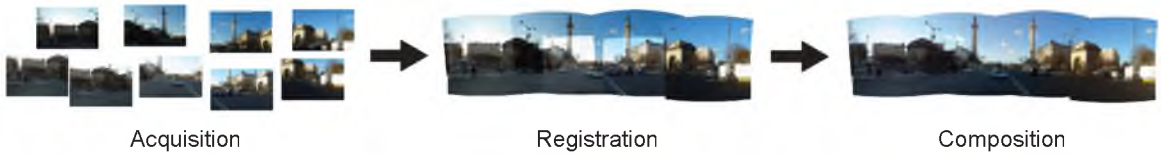


Figure 1.4: An example of the three stages of a panorama's creation. First, the individual images are acquired, typically, from a handheld camera. Second, for registration, the common coordinate system for the images is computed. Third, they are composited (blended) into a single seamless image.

greatly benefit schools, universities and the public in general. Enabling the military to combine footage from multiple security cameras, satellites, or flying drones into a seamless overview would allow operators to more accurately spot changes in a secure area or direct ground operations. While a full system is a long-term research goal, my dissertation work has focused on the last phase of the mosaic creation pipeline, specifically the *composition* stage.

After registration, image mosaics are combined in order to give the illusion of a seamless, massive image. Images acquired with inexpensive robots and consumer cameras pose an interesting challenge for image processing techniques. Often, panorama robots can take seconds between each photograph, causing gigapixel-sized images to be taken over the course of hours. Due to this delay, images can vary significantly in lighting conditions and/or exposure, and when registered can form an unappealing patchwork. Dynamic objects between images may also move during acquisition, ruining the illusion of a single, seamless image. Images acquired by air or satellite also suffer from an extreme version of this problem, where the time of acquisition can vary from hours to days for a single composite. Therefore,

minimizing the transition between images is the fundamental step in the composition stage, see Figure 1.5. The simplest transition approach is an alpha-blend of the overlap areas. Szeliski [155] provides an excellent introduction to this and other blending techniques. Such an approach does not work well in the presence of dynamic elements which move between captures, artifacts from poor registration, or varying exposures across images, see Figure 1.6. Often, it is preferable to compute a "hard" boundary, or seam, between the images as a final step, or as the preprocess for a technique such as gradient domain blending [133, 103]. Techniques exists to compute these seams based purely on distance [174, 132], but like blending, these will perform poorly when the scene contains moving elements.

A more sophisticated approach is to compute the boundaries between images through an energy function minimization to produce a nice transition between the mosaic images. These boundaries often provide the illusion of a seamless composited image. If exposure or lighting conditions vary among the images, a final color correction is necessary to produce a smooth image. Techniques such as gradient domain blending [133, 103], mean value coordinates [54], or bilateral upsampling [93] have been shown to provide adequately smooth images. Gradient domain blending remains the most popular, but also the most computationally expensive technique for color correction due to the quality of its final results.

The techniques associated with panorama boundaries and blending are typically computationally expensive and are considered an offline, postprocess for large (and even small) panoramas. As the focus of my dissertation work, I provide novel algorithms and techniques to bring these operations into an interactive setting for massive imagery.

Figure 1.5: A diagram to illustrate the main options available during the composition stage of panorama creation. The most simple process is to merge the image directly from the registration. The simplest approach is an alpha-blend of the overlap areas to achieve a smooth transition between images.

Figure 1.6: A simple blending approach is usually not sufficient in mosaics with moving elements. In these cases, the elements produce "ghosts" (circled here in red) in the final blend.

1.2 Boundaries

In the past, panorama image collections were captured in one sweeping motion (i.e., with image overlaps in only one dimension as in Figure 1.7). Today's images are often collections of multiple rows and columns or in more unstructured configurations. Consequently, more sophisticated panorama processing techniques continue to be developed to account for their more complex configurations.

After the initial registration, the panorama's individual images are blended to give the illusion of a single seamless image. As a usual first step, a boundary between images must be computed as input for a color correction technique such as gradient domain blending [133,

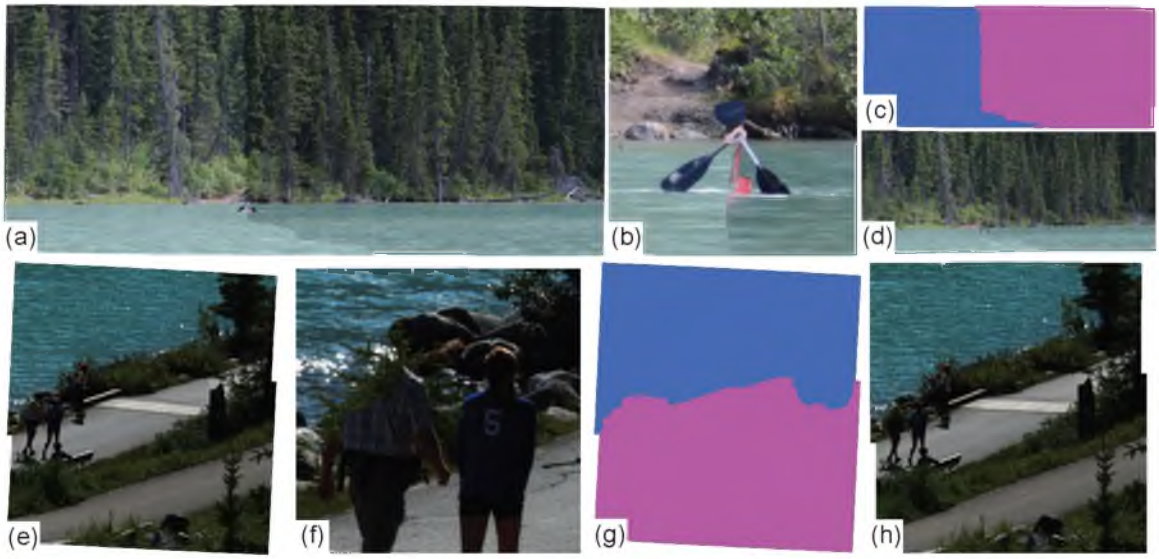


Figure 1.7: (a, e) Two examples (Canoe: 6842 x 2853, 2 images and Lake Path: 4459 x 4816, 2 images) of undesirable, yet exactly optimal seams (unique pairwise overlaps) for the pixel difference energy. (b, f) A zoom of visual artifacts caused by this optimal seam. (c, g) The pixel labeling. (d, h) The result produced by Adobe Photoshop^{TM}. Images courtesy of City Escapes Nature Photography.

103]. These boundaries are often called seams. Using a global optimization technique, these seams can be optimized to minimize visual artifacts due to transition between images. This is typically a pixel-based energy function such as color or color-gradient variations across the boundary.

Currently, the most used technique for global seam computation in a panorama is the Graph Cuts algorithm [26, 24, 92]. This is a popular and robust computer vision technique and has been adapted [101, 4] to compute the boundary between a collection of images. While this technique has been used with good success for a variety of panoramic or similar graphics applications [101, 4, 5, 3, 2, 94, 89, 44, 90], it can be problematic due to its high computational cost and memory requirements. Moreover, Graph Cuts applied to digital panoramas is a typical serial operation. Since computing the globally optimal boundaries between images is known to be NP-hard when the panorama is composed of more than a collection of unique pairwise overlaps [26], Graph Cuts aims to efficiently approximate the optimal solution and can therefore fall into local minima of the solution space.

There has been a large body of work to reduce some of the costs associated with Graph Cuts [25, 24, 142, 5, 107, 61, 124, 167, 138, 69, 106], but each of these works primarily focuses on Graph Cuts' typical image segmentation or de-noising applications. The success of many of these algorithms has yet to be demonstrated for digital panoramas. Those which have been used in a panorama context can suffer from limitations. For example, the popular Hierarchical Graph Cuts technique [107, 5] has been shown to operate well on hierarchies up to two to three levels in digital panoramas [5] and can be observed practice in panoramas such as the one shown in Figure 1.8. Given the recent trend of the increasing resolution of panoramas (many megapixels to gigapixels), one can see that this limited hierarchy would not be sufficient to compute the seams of images of these sizes. As a second example, Graph Cuts often needs an integer based energy function to guarantee convergence. This can prove problematic for high dynamic range (HDR) panoramas.

To overcome these types of limitations, a new approach was designed based on the following observations:

- A minimal energy seam does not necessarily give visually pleasing results. Figure 1.7 provides two examples of panoramas with an exact pairwise optimal energy boundary based on pixel difference across the seam. This should be sensitive to dynamic, moving objects which appear in the overlap. As you can see, neither seam would be considered ideal by a user since they cut through moving objects in the scene. Additionally, to

Figure 1.8: Hierarchical Graph Cuts has only been shown to work well on hierarchies of two to three levels. For this four picture panorama example, we can see that Hierarchical Graph Cuts produces a solution that passes though a dynamic scene element when using four levels of the hierarchy. A typical input value of a ten pixel dilation was used for this example. While a larger dilation parameter could be used, this would require a larger memory and computational cost which negates the benefits of the technique.

further argue the importance of this observation, the figure also shows very similar seams computed by Adobe Photoshop^{TM}, a widely used image editing application.

- There can be more than one valid seam solution. Even if the initial seam solution is visually acceptable to the user, there may be a large number of additional, valid solutions. Some of these alternative seams may be preferable and this determination is completely subjective. For example, a user may have wished that the high energy in a seam occurred in an area where it is less likely to be noticed such as the grassy area or the water in the images in Figure 1.7. Given moving elements in a scene, such elements may occur entirely within the area of an overlap. Therefore, there can be acceptable seams where the element is included and ones where it is not. Figure 1.9 provides examples.

- An interactive technique is necessary and attainable. Given the subjective nature of the image boundaries and the possibility of techniques falling into bad local minima, a user must be interjected into the seam boundary problem. Currently, finding panorama boundaries with Graph Cuts is an offline process with only one solution presented to the user. The only existing alternative is the manual editing, pixel by pixel, of the individual image boundaries. This is a time-consuming and tedious

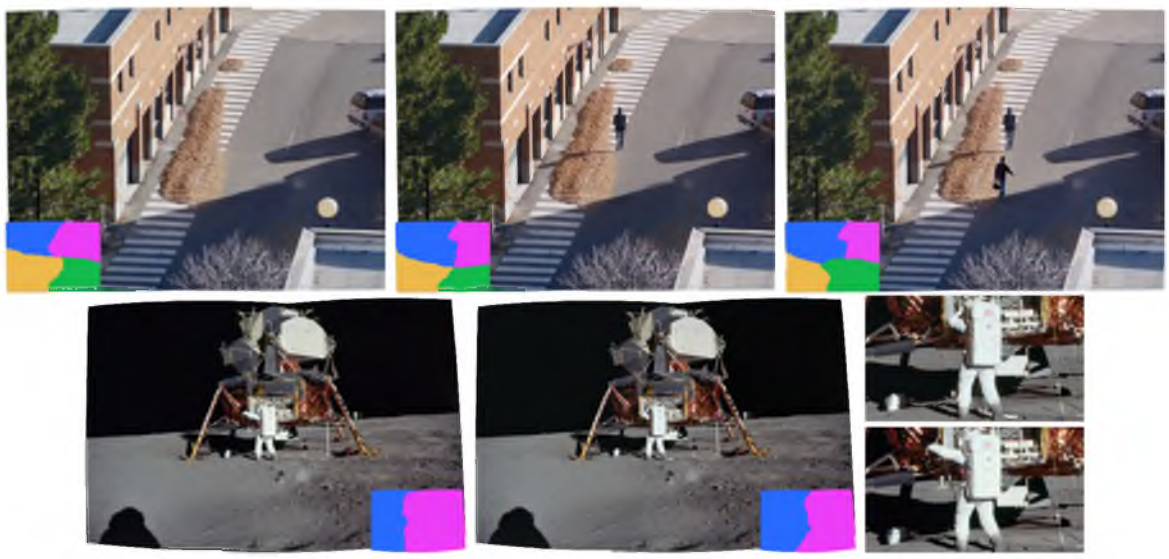


Figure 1.9: Even when seams are visually acceptable, moving elements in the scene may cause multiple visually valid seam configurations. On the top, this figure shows a four image panorama (Crosswalk: 4705 x 3543, four images) with three valid configurations. On the bottom, this figure shows a two image panorama (Apollo-Aldrin: 3432 x 2297, two images) with two valid configurations. Images courtesy of NASA.

process where the user relies on perception alone to determine if the manual seam is acceptable. Therefore, a guided interactive technique for image boundaries is necessary for panorama processing. This technique should allow users to include or remove dynamic elements, move an image seam out of possible local minima into a lower error state, move the seam into a higher error state (but one with more acceptable visual coherency) or hide errors in locations where they feel it is less noticeable. During these edits, the user should be provided the optimal seams given these new constraints.

- A solution based on pairwise boundaries can achieve good results for panoramas giving a fast, highly parallel, and light system. Computing pairwise-only optimal boundaries is both fast and exact (i.e., is guaranteed to find the global minimum). It is then of no surprise that these boundaries have been used often in past work for pan- or tilt-only panoramas [145, 46, 157, 165]. Although it has been thought not to generalize beyond this case, there has been no technique to use pairwise boundaries in panoramas with more complex structure, save for efforts to combine them via a distance metric [69] or sequentially [53]. This dissertation work not only provides a global solution based on pairwise boundaries, but also shows that this solution often

produces lower energy seams than Graph Cuts for panoramas. Given a technique to combine pairwise boundaries into a coherent seam network, each disjointed seam can be computed separately and trivially in parallel. Moreover, the solution produced for each is typically independent and therefore, memory and resources for each can be allocated and released as needed. In addition, the solution domain is only the overlap between pairs of images in contrast to some previous applications of Graph Cuts for panoramas [101, 4], which often consider the entire composite image as the solution domain. All of these properties give the potential for a very fast and light system even when operating on the full resolution imagery. Moreover, such a system should have the ability to be extended to an out-of-core or distributed setting.

This dissertation describes a new image boundary technique called *Panorama Weaving*. First, *Panorama Weaving* provides an automatic technique to create approximate optimal boundaries that is fast, has low memory requirements, and is easy to parallelize. Second, it provides the first interactive technique to enable the exploration of the seam solution space. This gives the end-user a powerful editing system for panorama seams. In particular, the contributions of this work on a technical level are:

- A novel technique to merge independently computed pairwise boundaries into a global, consistent seam network that does not cascade to a global calculation.
- A panorama seam creation technique based purely on pairwise boundary solutions. This technique is fast and highly parallel and shows significant speed-ups compared to previous work, even when run sequentially. More importantly, it achieves all of this even with full resolution imagery.
- Out-of-core and distributed seam creation algorithms which extend the creation technique to mosaics massive in size. These algorithms provide speed-ups compared to the state-of-the-art.
- The first system that allows interactive editing of seams in panoramas. This system guarantees minimal user input thanks to an efficient exploration of the solution space.
- An intuitive mesh specialization of a region adjacency graph that encodes seam and image relations. This adjacency mesh provides a way to guarantee the global consistency of the seam network of the interactions and also enables a robust editing of the network's topology.

1.3 Color Correction

Creating a single seamless image from a mosaic has been the subject of a large body of work for which gradient-domain (Poisson) techniques currently provide the best solution. Only one method exists to operate on the gradient-domain of massive images: the streaming multigrid [89] technique. However, processing the three gigapixel image of Figure 1.2 using this technique still takes well over an hour, which does not support an interactive trial-and-error artistic process. An additional disadvantage of traditional out-of-core methods is their tendency to achieve a low memory footprint at the cost of significantly proliferating the disk storage requirements. For example, the multigrid method [89] requires auxiliary storage an order of magnitude greater than the input size, almost half of which is due to gradient computation. In contrast, our approach completely avoids such data proliferation, thereby allowing the processing of data which already pushes the boundaries of available storage. The multigrid method [89] is also limited by main memory usage since it is proportional to the number of iterations of the solver. This can cause the method to not achieve acceptable results for images that may require a large number of iterations, as shown in Section 4. This work introduces a new method with memory usage independent of the number of iterations of the Poisson solver and, therefore, would scale gracefully in these cases.

An option to reduce times is to design a similar scheme to run in a distributed environment. Consequently, there has been recent work to extend the multigrid solver [90] to a parallel implementation, reducing the time to compute a gigapixel solution to mere minutes. However, this approach is primarily a proof-of-concept since it does not supply the classic tests of scalability (weak or strong) nor is it tested significantly. Like many out-of-core methods, proliferation of disk storage requirements is a major drawback. For example, testing was only possible with a full 16-node cluster for some of the streaming multigrid test data due to excessive storage demands. Finally, the technique assumes that a small number of predetermined iterations is sufficient to achieve a solution, which may not always be the case. This implementation was optimized for a single distributed system and therefore, is unlikely to port well to other environments. History has shown that levels of abstraction that remove complexity from a code base can be instrumental in the advancement of technologies. Abstraction that allows simple and portable code accelerates innovation and reduces time to develop new ideas. The cloud should be explored as such abstraction, allowing a developer to ignore much of the more tedious and complex elements in implementing a distributed graphics algorithm. A general scheme cannot beat the performance of highly specialized and

optimized code. Often for organizations with resources, there may be cases where speed and efficiency are more important than the cost to create and maintain a typical implementation. Although with increased availability of cloud commodities, there is now the opportunity to offer more members of our community the ability to develop new algorithms for a distributed environment.

In particular, in the area of color correction this dissertation work introduces a simple and light-weight framework that provides the user with the illusion of a full Poisson system solve at interactive frame rates for image editing. This framework also allows for the computation of a full solution on a single machine with a simple approach, rivaling the run time of the current best out-of-core technique [89], while producing equal or higher quality results on images that require a large number of iterations. The system is flexible enough to handle different hierarchical image formats such as tiling for higher quality images or HZ-order for greater input/output (I/O) speed. In particular, by exploiting a new implicit kd-tree hierarchy for HZ-order, the framework needs only to access and solve visible pixels. This allows an artist to interactively apply gradient-based techniques to images gigapixels in size. This new framework is straightforward and requires neither complicated spatial indexing nor advanced caching schemes. Additionally, this work introduces a framework for parallel gradient-domain processing inspired by the out-of-core technique with a novel reformulation to provide an efficient parallel distributed algorithm. This new framework has both a straightforward implementation and shows both strong and weak parallel scalability. When implemented in standard MPI (Message Passing Interface), the same code base ports well to multiple distributed systems. Furthermore, this distributed algorithm can be wrapped in a level of abstraction to be run on the cloud, allowing for a simple implementation, as well as allowing it to be distributed to the community at large.

Specifically, the contributions of this work are:

- A coarse-to-fine progressive Poisson solver running at interactive frame rates, extended to a wide variety of gradient domain tasks, with the ability to scale to gigapixel images. This cascadic solver entirely avoids the coarsening stage of the V-cycle yet produces high quality results.

- A method to locally refine solutions having time and space requirements that are linearly dependent on the screen resolution rather than the resolution of the input image.

- A full out-of-core solver that maintains strict control over system resources, rivals the run-times for the best known method and consistently achieves quality results where previous methods may not converge well in practice.
- A light-weight streaming framework that provides adaptive multiresolution access to out-of-core images in a cache coherent manner, without using intricate indexing data structures or precaching schemes.
- A distributed algorithm based on the out-of-core scheme, which has a straightforward implementation and shows both strong and weak parallel scalability.
- The first distributed Poisson solver for imaging implemented in the cloud.

CHAPTER 2

RELATED WORK

This chapter will outline the previous work for the two major portions of the composition step of the panorama creation pipeline: image boundaries, Section 2.1, and color correction, Section 2.2. In particular, these sections will address the related work for the state-of-the-art for image boundaries, minimal image seams, and color correction, gradient domain editing.

2.1 Image Boundaries: Seams

This section details the related previous work for computing image boundaries for an image mosaic. In particular, this section focuses on the current state-of-the-art that is the computation of minimal mosaic seams.

2.1.1 Pairwise Boundaries

Some of the seminal works in digital panoramas assume that an image collection is acquired in a single sweep (either pan, tilt or a combination of the two) of the scene. In such panoramas, only pairwise overlaps of images need be considered [119, 120, 157, 145, 46, 165]. The pairwise boundaries which have globally minimal energy can be computed quickly and exactly using a min-cut or min-path algorithm. There is an intuitive and proven duality between min-cut and single-source/single-destination min-path [72]. These pairwise techniques were thought to not be general enough to handle the many configurations possible in modern panoramas. Recent work [69] has dealt with the combination of these seams for more complex panoramas, although the seam combinations are still based on an image distance metric. Other recent work [53], combined these separate seams for the purposes of texture synthesis combining seams sequentially. For their work, this was sufficient to provide good results for textures. The combination and intersection of these seams in a digital panorama can be more complex and therefore, a more expressive combination is necessary. In addition, interaction was not considered as a necessary functionality in these works. This dissertation presents a novel technique to combine these disjoint seams into a

global panorama seam network and allow for manual user interaction.

2.1.2 Graph Cuts

The Graph Cuts technique [26, 24], computes a k-labeling of a graph, typically an image, to minimize an energy function on the domain. An algorithm that guarantees to find the global minimum is considered to be NP-hard [26] and therefore Graph Cuts was designed to efficiently compute a good approximation. Graph Cuts has been shown to give good results for a variety of energy functions [92]. Thus, it is of no surprise given this versatility that it has been shown to adapt to the image mosaic and panorama boundary problem [101, 4]. However, Graph Cuts is both a computationally expensive and memory intensive technique. Given these requirements, there has been work on accelerating the Graph Cuts process by, for instance, adapting the technique to run on the GPU [167], in parallel [106], or in parallel-distributed [48] environments. Building a hierarchy for the Graph Cuts computation [107, 5] has shown to be popular due to its reduction of memory and computation costs. For panoramas, this strategy has only been shown to provide good results for a hierarchy of two to three levels [5]. There has also been work on bringing Graph Cuts into an interactive setting [25, 142, 104, 61, 124] although these works have focused only on user guided image segmentation. This dissertation provides the first technique to allow interactive editing of panorama boundaries.

2.1.3 Alternative Boundary Techniques

While Graph Cuts still maintains its popularity as a solution to the minimal boundary problem, there has been other ongoing work on alternative techniques. For example, there has been work on techniques based on luminance voting [82]. There has also been recent work using geodesics to interactively compute pairwise minimal boundaries [45].

2.1.4 Out-of-Core and Distributed Computation

While there has been previous work to bring the Graph Cuts technique to massive grids, the previous work has only dealt with extending the algorithm to a distributed and out-of-core environment [48], No current technique decouples the two. The work of this dissertation has the flexibility to operate in-core, out-of-core, or distributed depending on the application or available resources. Moreover, the inherent parallelism of the new technique is likely to outperform the previous work. Finally, there has been no work to allow interaction with these seams at massive scales. Hierarchical Graph Cuts has been used on large images [2],

although given the documented limitation on the viable number of levels in the hierarchy [5] this will not scale massively. Applying standard Graph Cuts as a sweeping window over an image neighborhood [94] has been used to produce boundaries for gigapixel imagery. Such a process has yet to be formulated in parallel and is potentially very computationally expensive.

2.2 Color Correction: Gradient Domain Editing

This section details the related previous work in the most popular and sophisticated color correction procedure for image mosaics called gradient domain image editing, or by its alternative name, Poisson image editing.

2.2.1 Poisson Image Processing

A variety of gradient-based methods provide a popular, but computationally expensive, set of techniques for advanced image manipulation. Given a guiding gradient field constructed from one or multiple source images, these techniques attempt to find a closest-fit image using some predetermined distance metric. This basic concept has been adapted for standard image editing [133], as well as more advanced matting operations [152] , and high level drag-and-drop functionality [79]. Furthermore, gradient-based techniques can tone map high dynamic range images to display favorably on standard monitors [55] or hide the seams in panoramas [133, 103, 4, 89]. Other applications include detecting lighting [76] or shapes from images [173], removing shadows [57] or reflections [6], and gradient domain painting [114]. Recently, an alternative approach using mean value coordinates has smoothly interpolated the boundary offset between source images, thereby mimicking Dirichlet boundary conditions [54]. This promising new line of research has yet to show support of Poisson techniques such as tone-mapping, the ability to work well out-of-core, or consistently acceptable results for methods that typically require Neumann boundary conditions.

2.2.2 Poisson Solvers

The solution to a two-dimensional (2D) Poisson problem lies at the core of gradient based image processing. Poisson equations have wide utility in many engineering and science applications. Computing their solution efficiently has been the focus of a large body of work and even a cursory review is beyond the scope of this dissertation. For small images, methods exist to find the direct analytical solution using Fast Fourier transforms [75, 7, 8,

113]. Simchony [146] provides a survey of these methods for computer vision applications. Often the problem is simplified by discretization into a large linear system whose dimension is typically the number of pixels in an image. If this system is small enough to fit into memory, methods exist to find the direct solution and we refer the reader to Dorr [51] who provides an extensive review on direct methods. Typically, iterative Krylov subspace methods, such as conjugate gradient, are used due to their fast convergence. For much larger systems, memory consumption is the limiting factor and iterative solvers, such as Successive Over-Relaxation (SOR) [13] become more attractive.

Depending on the application, different levels of accuracy may be required. Sometimes, a coarse approximation is sufficient to achieve the desired result. Bilateral upsampling methods [93] operating on a coarse solution produced good results for applications such as tonemapping. Such methods have not yet been shown to handle applications such as image stitching where the interpolated values are typically not smooth at the seams between images.

When pure upsampling is insufficient, the system must be solved fully. Multigrid methods are often employed to aid the convergence of an iterative solver. Such methods have proven particularly effective by dealing with the large scale trends at coarse resolutions. These techniques include preconditioners [66, 155] and multigrid solvers [27, 28]. There exist different variants of multigrid algorithms using either adaptive [20, 88, 21, 2, 139] or nonadaptive meshes [87, 89]. As a first step in a complete multigrid system, the mesh is coarsened. The Poisson equation can then be solved in a coarse-to-fine manner. One full iteration, from fine to coarse and back, is typically called a V-cycle. Most recently, a V-cycle was implemented in a streaming fashion for large panoramas [89]. However, other systems only implement parts of the V-cycle. Kopf et al. [94] implement only the second half in a pure upsampling procedure, while Bolitho et al. [21] implement a purely coarse-to-fine solver also called cascadic [22]. Lischinski et al. [105] applied this pure coarse-to-fine approach for interactive tonal adjustment. The technique outlined in this diseration (for the first time) shows that a cascadic approach has applications well beyond the adjustment of tonal values and can be used for a wide variety of gradient based image processing techniques. This work also extends such techniques to allow the interactive editing and processing of gigapixel images. The solver propagates sufficient information from coarse-to-fine, allowing us to achieve local solutions at interactive rates that are virtually indistinguishable from the full-resolution solution.

2.2.3 Out-of-Core Computation

Toledo [160] presents a survey of general out-of-core algorithms for linear systems. The majority of algorithms surveyed assume that at least the solution vectors can be kept in main memory, which is not the case for large images. For out-of-core processing of large images, the streaming multigrid method of Kazhdan and Hoppe [89] has so far provided the only solution. However, processing a three gigapixel image using this technique still takes well over an hour which does not support an interactive trial-and-error artistic process. Many algorithms such as tone mapping require careful parameter tuning to achieve good results. Thus, waiting several hours to examine the effects of a single parameter change is not feasible in this context.

An additional disadvantage of traditional out-of-core methods is their tendency to achieve a low memory footprint at the cost of significantly proliferating the disk storage requirements. For example, the multigrid method [89] requires auxiliary storage an order of magnitude greater than the input size, almost half of which is due to gradient computation. In contrast, in this dissertation, I with my collaborators, introduce an approach that completely avoids such data proliferation, thereby allowing the processing of data, which already pushes the boundaries of available storage.

The multigrid method [89] is also limited by main memory usage since it is proportional to the number of iterations of the solver. This can cause the method to not achieve acceptable results for images that may require a large number of iterations, as shown in Section 4. This work provides a new method with memory usage independent of the number of iterations and, therefore, scales gracefully in these cases.

2.2.4 Distributed Computation

Recently, the streaming multigrid method has been extended to a distributed environment [90] and has reduced the time to process gigapixel images from hours to minutes. The distributed multigrid requires 16 bytes/pixel of disk space in temporary storage for the solver as well as 24 bytes/pixel to store the solution and gradient constraints. For the terapixel example of Kazhdan et al. [90], the method had a minimum requirement of 16 nodes in order to accommodate the needed disk space for fast local caching. In contrast, the approach outlined in this work needs no temporary storage and is implemented in standard MPI. Streaming multigrid also assumes a precomputed image gradient, which would add substantial overhead in initialization to transfer the color float or double data. Our new approach is initialized using original image data plus an extra byte for image

boundary information which equates to 1/3 less data transfer in initialization than the previous method. Data transfers between this solver's phases, while floating point, only deal with the boundaries between compute nodes which is substantially smaller than the full image and therefore are rarely required to be cached to disk. The multigrid method [89, 90] may also be limited by main memory, since the number of iterations of the solver is directly proportional to the memory footprint. For large images, this limits the solver to only a few Gauss-Seidel iterations and therefore may not necessarily converge for challenging cases. Our method's memory usage is independent of the number of iterations and can therefore solve images that have slow convergence.

Often large images are stored as tiles at the highest resolution; therefore, methods that exploit this structure would be advantageous. Stookey et al. [148] use a tile-based approach to compute an over-determined Laplacian PDE (partial differential equation). By using tiles that overlap in all dimensions, the method solves the PDE on each tile separately and then blends the solution via a weighted average. Unfortunately this method cannot account for large scale trends beyond a single overlap and therefore can only be used on problems which have no large (global) trends. Figure 2.1 illustrates why this would be a problem for panorama processing. The coarse up-sampling of our approach fixes this issue.

2.2.5 Cloud Computing - MapReduce and Hadoop

MapReduce [47] was developed by Google as a simple framework to process massive data on large distributed systems. It is an abstraction that owes its inspiration to functional programming languages such as Lisp. At its core, the framework relies on two simple operations:

- Map: Given input, create a key/value pair.
- Reduce: Process all values of a given key.

All the complexity of a typical distributed implementation due to data distribution, load balancing, fault-recovery and communication are under this abstraction layer and therefore can be ignored by a developer. This framework, when combined with a distributed file system, can be a simple yet powerful tool for data processing.

Hadoop is an open source implementation of MapReduce maintained by the Apache Software Foundation and can be optionally coupled with its own distributed file system (HDFS). Pavlo et al. [131] found that Hadoop was easy to deploy and use, offered adequate

Figure 2.1: Although the result is a seamless, smooth image, without coarse upsampling the final image will fail to account for large trends that span beyond a single overlap and can lead to unwanted, unappealing shifts in color.

scalability, has very effective fault-tolerance, and, most importantly, was easy to adapt for complex analytic tasks. Hadoop is also widely available as a commodity resource. For example, Amazon Web Services, a service suite that has become nearly synonymous with cloud computing in the media, provides Hadoop as a native capability [10]. Companies have begun to use Hadoop as a simple alternative for data processing on large clusters [71]. For instance, The New York Times has used Hadoop for large scale image to PDF conversion [68]. Google, IBM, and NSF have also partnered to provide a Hadoop cluster for research [41].

2.2.6 Out-of-Core Data Access

Given an image, it is well known that the standard row-major order exhibits good locality in only one dimension and is therefore ill-suited for an unconstrained out-of-core storage scheme [168]. Previous out-of-core Poisson methods [89] have been noted to be severely limited by this constraint. Instead, indexing based on various space-filling curves [143] has been proposed in different applications [126, 70, 17, 102] to exploit their inherent geometric locality. Of particular interest is the Z-order (also called Lebesgue-order) [17, 128] since it allows an especially simple conversion to and from standard row-major indices. While Z-order exhibits good locality in all dimensions, it does so only at a fixed resolution and does not support hierarchical access. Instead, this work will utilize the hierarchical variant, called HZ-order, proposed by Pascucci and Frank [128].

CHAPTER 3

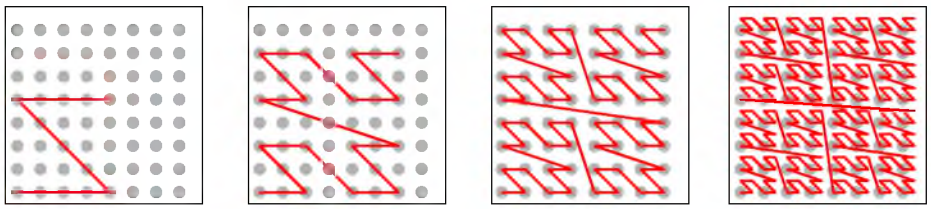
SCALABLE AND EFFICIENT DATA ACCESS

At the core of any large data processing system is an efficient scheme for data access. In this chapter, I will detail the technology used in the systems outlined in this work. In Section 3.1, I will review the fundamentals of the *hierarchical Z-order (HZ-order)* for two-dimensional arrays, our chosen format for large image processing. I will also provide a new, simple algorithm in Section 3.2 for accessing data organized in HZ-order, while avoiding the repeated index conversions used in [128]. Section 3.3 will provide a new parallel write scheme for HZ-order data. Finally, in Section 3.4 I will give an outline of the ViSUS software infrastructure, the core system behind much of the massive image processing outlined in this dissertation. Conversion of large images into theViSUS format requires no additional storage, compared to the typical 1/3 data increase common for typical tiled image hierarchies. From our test data, we have found that there is only a 27% overhead due to the conversion compared to just copying the raw data which makes this conversion very light. The conversion requires no operations on the pixel data and will outperform even the most simple tiled hierarchies, which require some manipulation of the pixel data. Section 5.2.2 will show that this new I/O infrastructure reduces the overhead by 28%-40% when compared to a standard tiled image hierarchy. These numbers reflect the theoretical bound of 1/3 overhead, made worse by the inability to constrain real queries to perfect alignment with the boundaries of a quadtree.

3.1 Z- and HZ-Order Background

The data access routine of our system achieves high performance on our image data by utilizing a hierarchical variant of a standard Z-order (Lebesgue) space filling curve to lay out our two-dimensional data in one-dimensional memory. In the two-dimensional case the Z-order curve can be defined recursively by a Z shape whose vertices are replaced by Z shapes half its size (see Figure 3.1 (a)). Given the binary row-major index of a pixel

(a)



(b)

Sample index	Binary representation Z-order	i	j
0	0000	00	00
1	0001	01	00
2	0010	00	01
3	0011	01	01
4	0100	10	00
5	0101	11	00
6	0110	10	01
7	0111	11	01
8	1000	00	10
9	1001	01	10
10	1010	00	11
11	1011	01	11
12	1100	10	10
13	1101	11	10
14	1110	10	11
15	1111	11	11

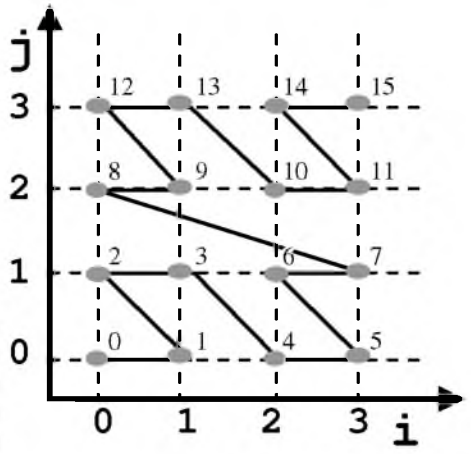
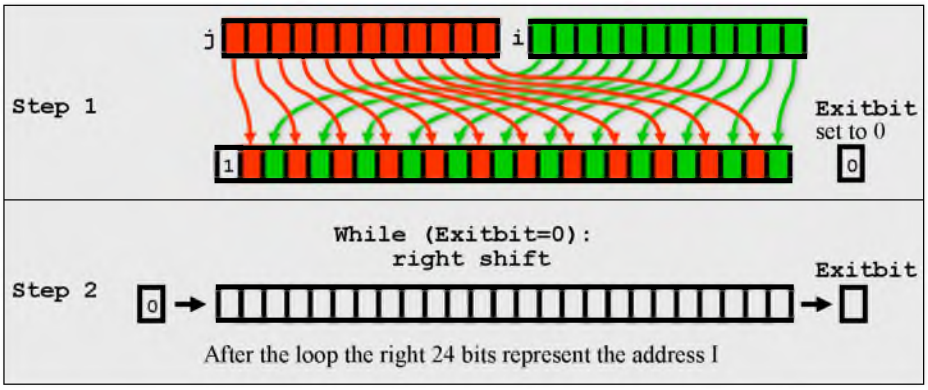


Figure 3.1: (a) The first four levels of the Z-order space filling curve; (b) 4x4 array indexed using standard Z-order

$(i_n \ldots i_1 i_0, j_n \ldots j_1 j_0)$ the corresponding Z-order index I is computed by interleaving the indices $I = j_n i_n \ldots j_1 i_1 j_0 i_0$ (see Figure 3.2 (a) step 1).

While Z-order exhibits good locality in all dimensions, it does so only at full resolution and does not support hierarchical access. Instead, our system uses the hierarchical variant, called HZ-order, proposed by Pascucci and Frank [128]. This new index changes the standard Z-order of Figure 3.1 (b) to be organized by levels corresponding to a subsampling binary tree, in which each level doubles the number of points in one dimension (see Figure 3.2 (b)). This pixel order is computed by adding a second step to the index conversion. To compute an HZ-order index $\hat{I}$, the binary representation of a given Z-order index I is shifted to the right until the first 1-bit exits. During the first shift, a 1-bit is added to the left and 0-bits are added in all following shifts (see Figure 3.2 (a)). This conversion could have a potentially very simple and efficient hardware implementation. The software C++ version can be implemented as follows:

(a)



(b)

	Binary representation				Level of resolution and its samples in red
Sample index	HZ-order	Z-order	i	j	
0	0000	0000	00	00	0
1	0001	1000	00	10	1
2	0010	0100	10	00	2
3	0011	1100	10	10	
4	0100	0010	00	01	3
5	0101	0110	10	01	
6	0110	1010	00	11	
7	0111	1110	10	11	
8	1000	0001	01	00	4
9	1001	0011	01	01	
10	1010	0101	11	00	
11	1011	0111	11	01	
12	1100	1001	01	10	
13	1101	1011	01	11	
14	1110	1101	11	10	
15	1111	1111	11	11	

Figure 3.2: (a) Address transformation from row-major index (i, j) to Z-order index I (Step 1) and then to hierarchical Z-order index (Step 2); (b) Levels of the hierarchical Z-order for a 4x4 array. The samples on each level remain ordered by the standard Z-order.

```
inline adhocindex remap(register adhocindex i){
  i |= last_bit_mask;  // set leftmost one
  i /= i&-i;           // remove trailing  zeros
  return (i>>1);       // remove rightmost one
}
```

We store the data in a way guaranteeing efficient access to any subregion without internal caching and without opening a data block more than once. Furthermore, we allow for storage of incomplete arrays. In our storage format, we first sort the data in HZ-order and group consecutive samples in blocks of constant size. A sequence of consecutive blocks is grouped

into a record and records are clustered in groups, which are organized hierarchically. Each record has a header specifying which of its blocks are actually present and if the data are stored raw or compressed. Groups can miss entire records or subgroups, implying that all their respective blocks and records are missing.

The file format is implemented via a header file describing the various parameters (dimension, block size, record size, etc.) and one file per record. The hierarchy of groups is implemented as a hierarchy of directories each containing a predetermined maximum number of subdirectories. The leaves of each directory contain only records. To open a file, one needs only to reconstruct the path of a record and defer its search to the file system. In particular, the path of a record is constructed as follows: we take the HZ-address of the first sample in the record, represent it as a string, and partition it into chunks of characters naming directories, subdirectories, and the record file. Note that, since blocks, records and groups can be missing, one is not restricted to arrays of data that cover the entire index space. In fact, we can easily store even images with different regions sampled at different resolutions.

3.2 Efficient Multiresolution Range Queries

One of the key components of our framework is the ability to quickly extract rectangular subsets of the input image in a progressive manner. Computing the row-major indices of all samples residing within a given query box is straightforward. However, efficiently calculating their corresponding HZ-indices is not. Transforming each address individually results in a large number of redundant computations by repeatedly converting similar indices. To avoid this overhead, we introduce a recursive access scheme that traverses an image in HZ-order, while concurrently computing the corresponding row-major indices. This traversal implicitly follows a kd-tree style subdivision, allowing us to quickly skip large portions of the image.

To better illustrate the algorithm I will first describe how to recursively traverse an array in plain Z-order using the 4x4 array of Figure 3.1 (b) as example. Subsequently, I will discuss how to restrict the traversal to a given query rectangle and finally, how the scheme is adapted to HZ-order.

We use a stack containing tuples of type *(split_dimension, I_start, min_i, max_i, min_j, max_j, num_elements)*. To start the process we push the tuple $t0$ =*(1,0,0,3,0,3,16)* onto the stack. At each iteration we pop the top-most element t from the stack. If t contains only a single element we output the current *I_start* as HZ-index and fetch the correspond-

ing sample. Otherwise, we split the region represented by t into two pieces along the axis given by *split_dimension* and create the corresponding tuples $t1$ =*(0,0,0,3,0,1,8)* and $t2$ =*(0,8,0,3,2,3,8)*. Note that all elements of $t1$ and $t2$ can be computed from t by simple bit manipulation. In case of a square array, we simply flip the split dimension each time a tuple is split. However, one can also store a specific split order to accommodate rectangular arrays. Figure 3.3 shows the first eight iteration of the algorithm outputting the first four elements in the array of Figure 3.1 (b).

To use this algorithm for fast range queries, each tuple is tested against the query box as it comes off the stack and discarded if no overlap exists. Since the row-major indices describing the bounding box of each tuple are computed concurrently, the intersection test is straightforward. Furthermore, the scheme applies, virtually unchanged, to traverse samples in Z-order that sub-sample an array uniformly along each axis, where the sub-sampling rate along each axis could be different.

Finally, to adapt the algorithm to HZ-order (see Figure 3.2 (b)), one exploits the following two important facts:

- One can directly compute the starting HZ-index for each level. For example, in a squared array level 0 contains one sample and all other levels h contain 2^{h-1} samples. Therefore the starting HZ-index of level h, I^h_{start}, is 2^{m-h}, where m is the number of bits of the largest HZ-index.

- Within each level, samples are ordered according to plain Z-order and can be traversed with the stack algorithm described above, using the appropriate subsampling rate.

Using these two facts one can iterate through an array in HZ-order by processing one level at a time, adding I^h_{start} to the *I_start* index of each tuple.

In practice, we avoid subdividing the stack tuples to the level of a single sample. Instead, depending on the platform, we choose a parameter n and build a table, with the sequence of Z-order indices for an array with 2^n elements. When running the stack algorithm, each time a tuple t with 2^n elements appears, we loop through the table instead of splitting t. By accessing only the necessary samples in strict HZ-order, the stack-based algorithm guarantees that only the minimal number of disk blocks are touched and each block is loaded exactly once.

For progressively refined zooms in a given area, we can apply this algorithm with a minor variation. In particular, one would need to reduce the size of the bounding box represented

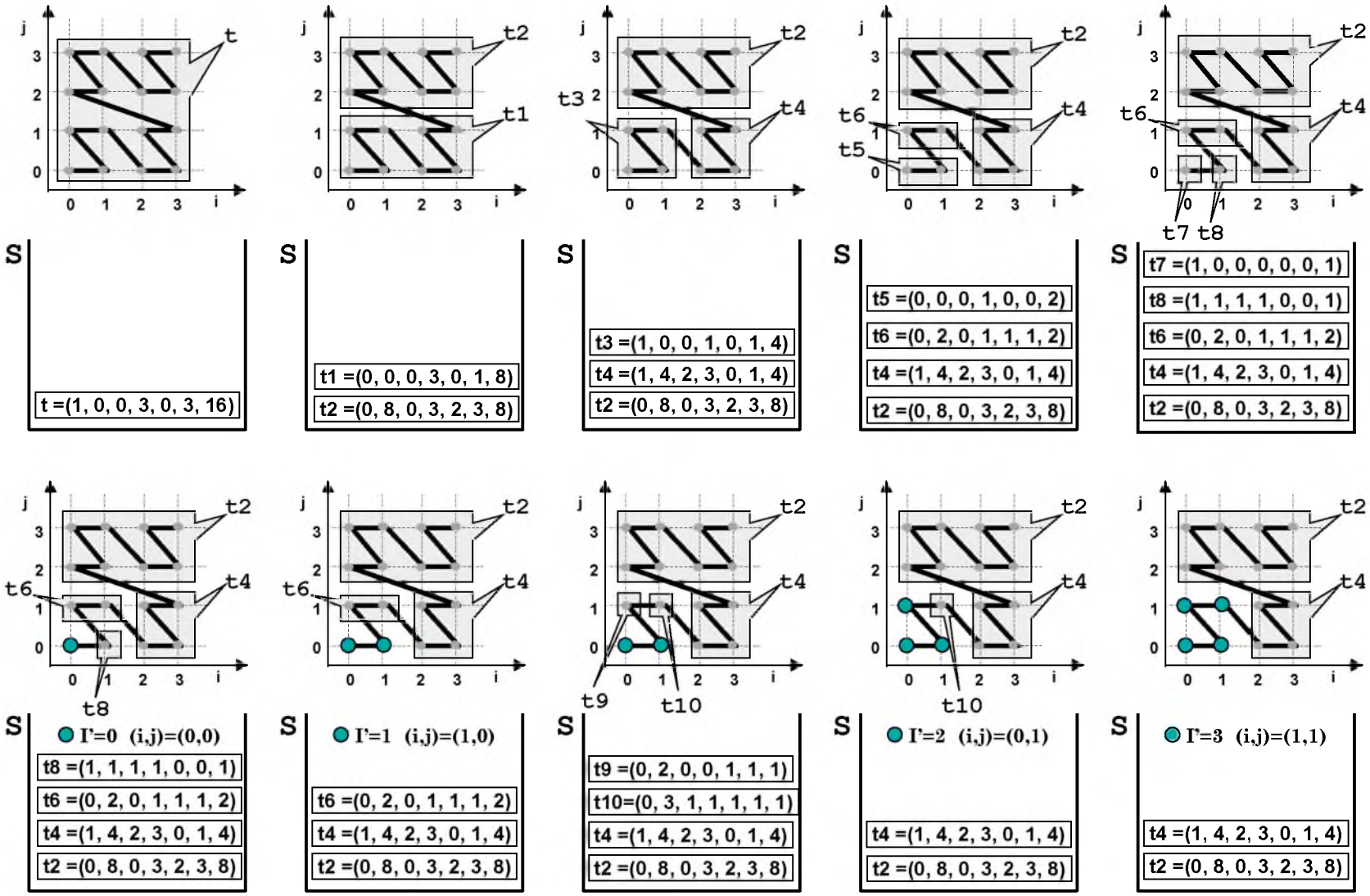


Figure 3.3: Our fast-stack Z-order traversal of a 4x4 array with concurrent index computation

in a tuple each time it is pushed back into the stack. In this way, even for a progressively refined zoom, one would access only the needed data blocks, each being accessed only once.

3.3 Parallel Write

The multiresolution data layout outlined above is a progressive, linear format and therefore has a *write* routine that is inherently serial. When processing a large image on a distributed system, or even on a single multicore system, it would be ideal for each node, or process, to be able to write out its piece of the data directly in this layout. Therefore a parallel write strategy must be employed. Figure 3.4 illustrates different possible parallel strategies. As shown in Figure 3.4 (a), each process can naively write its own data directly to the underlying binary file. This is inefficient due to the large number of small file accesses. As data gets large, it becomes disadvantageous to store the entire dataset as a single, large file and typically the entire dataset is partitioned into a series of smaller more manageable files. This disjointness can be used by a parallel write routine. As each simulation process produces simulation data, it can store its piece of the overall dataset locally and pass the data on to an aggregator process. These aggregator processes can be used to gather the individual pieces and composite the entire dataset. In Figure 3.4 (b), each process transmits each contiguous data segment to an intermediate aggregator. Once the aggregator's buffer is complete, the data are written to disk using a single large I/O operation. Figure 3.4 (c), illustrates a strategy where several noncontiguous memory accesses from each process are bundled into into a single message. This approach reduces the number of small network messages needed to transfer data to aggregators. This last strategy has been shown to exhibit good throughput performance and weak scaling for S3D combustion simulation applications when compared to standard Fortran I/O benchmark [98, 99].

3.4 ViSUS Software Framework

The ViSUS (Visual Streams for Ultimate Scalability) software framework[6] has been designed as an environment to allow the interactive exploration of massive datasets on a variety of hardware, possibly over platforms distributed geographically. The system and I/O infrastructure is designed to handle n-dimensional datasets but is typically used on two-dimensional and three-dimensional image data. This two-dimensional portion of this

[6] `http://visus.co` or `http://visus.us`

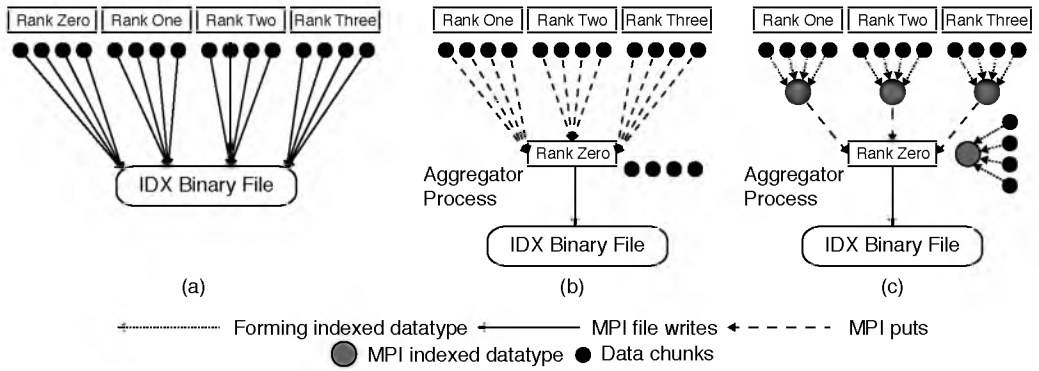


Figure 3.4: (a) Naive parallel strategy where each process writes its piece of the overall dataset into the underlying flle, (b) each process transmits each contiguous data segment to an intermediate aggregator. Once the aggregator's buffer is complete, the data are written to disk, (c) several noncontiguous memory accesses are bundled into a single message to decrease communication overhead.

system is the core application on which the massive applications outlined in this dissertation are built.

The ViSUS software framework was designed with the primary philosophy that the visualization and/or processing of massive data need not be tied to specialized hardware or infrastructure. In other words, a visualization environment for large data can be designed to be lightweight, highly scalable and run on a variety of platforms or hardware. Moreover, if designed generally such an infrastructure can have a wide variety of applications, all from the same code base. Figure 3.5 details example applications and the major components of the ViSUS infrastructure. The components can be grouped into three major categories. First, a lightweight and fast out-of-core data management framework using multiresolution space filling curves, which I have outlined Sections 3.1, 3.2, and 3.3. This allows the organization of information in an order that exploits the cache hierarchies of any modern data storage architectures. Second, ViSUS contains a dataflow framework to allow data to be processed during movement. Processing massive datasets in their entirety would be a long and expensive operation which hinders interactive exploration. By designing new algorithms to fit within this framework, data can be processed as it moves. The *Progressive Poisson* technique outlined in Section 5.2 is one such new algorithm. Third, ViSUS provides a portable visualization layer that was designed to scale from mobile devices to powerwall displays with the same code base.

Figure 3.5 provides a diagram of the ViSUS software architecture. In this section I

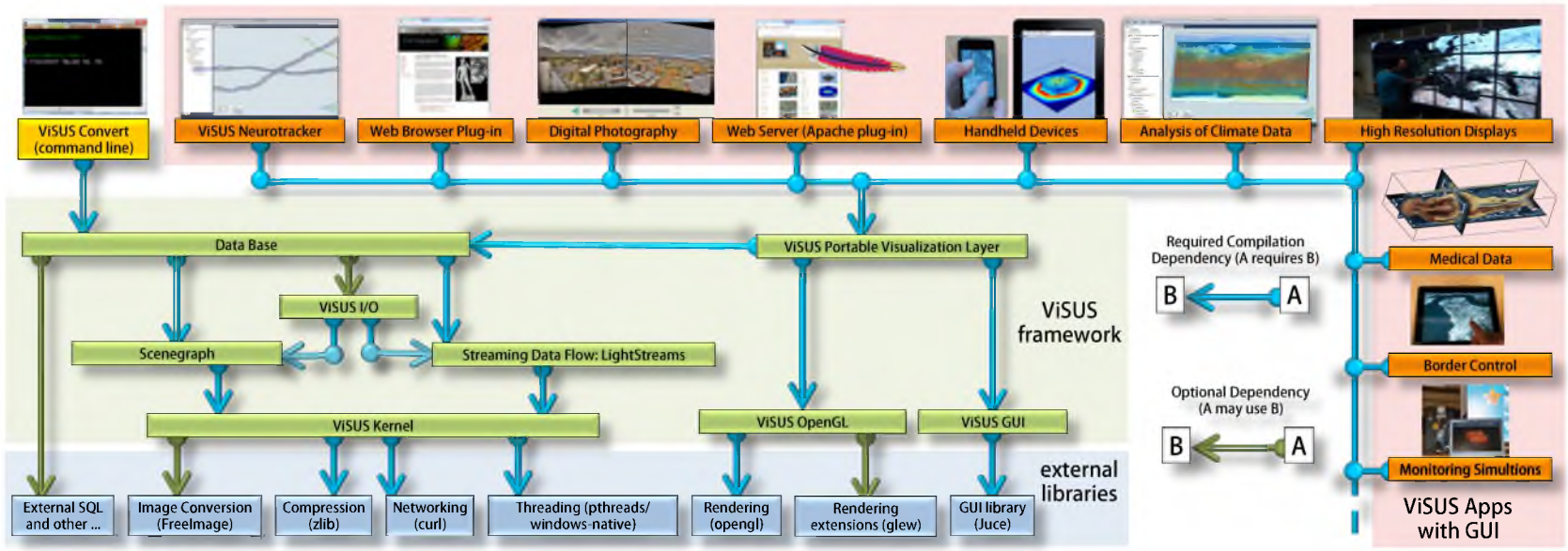


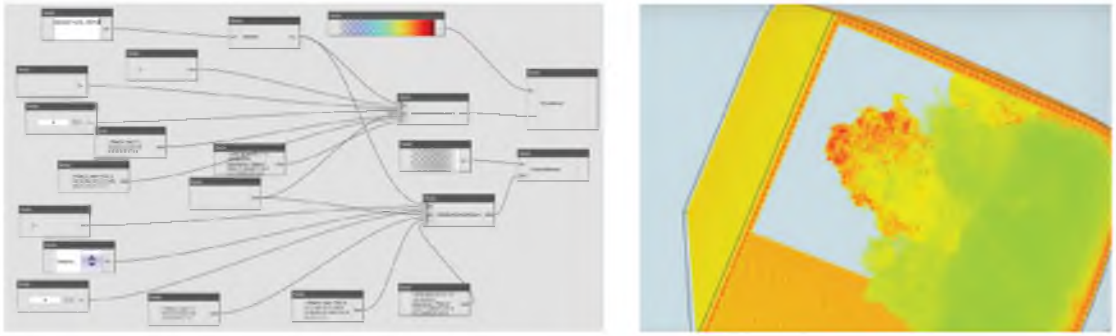
Figure 3.5: The ViSUS software framework. Arrows denote external and internal dependences of the main software components. Additionally this figure illustrates the relationship with several example applications that have been successfully developed with this framework.

will detail three of ViSUS's major components and how they couple with the efficient data access detailed in the previous sections to achieve a fast, scalable, and highly portable data processing and visualization environment. Finally, I will illustrate an important additional use of this infrastructure, real-time monitoring of scientific simulations.

3.4.1 LightStream Dataflow and Scene Graph

Even simple manipulations can be overly expensive when applied to each variable in a large scale dataset. Instead, it would be ideal to process the data based on need by pushing data through a processing pipeline as the user interacts with different portions of the data. The ViSUS multiresolution data layout enables efficient access to different regions of the data at varying resolutions. Therefore different compute modules can be implemented using progressive algorithms to operate on these data stream. Operations such as binning, clustering, or rescaling are trivial to implement on this hierarchy given some known statistics on the data, such as the function value range, etc. These operators can be applied to the data stream as-is, while the data are moving to the user, progressively refining the operation as more data arrives. More complex operations can also be reformulated to work well using the hierarchy. For instance, using the layout for image data produces a hierarchy which is identical to a subsampled image pyramid on the data. Moreover, as data are requested progressively, the transfer will traverse this pyramid in a coarse-to-fine manner. Techniques such as gradient-domain image editing can be reformulated to use this progressive stream and produce visually acceptable solutions which will be detailed in Section 5.2. These adaptive, progressive solutions allow the user to explore a full resolution solution as if it was fully available, without the expensive, full computation.

ViSUS LightStream facilitates this steam processing model by providing definable modules within a dataflow framework with a well understood API. Figure 3.6 gives an example of a dataflow for the analysis and visualization of a scientific simulation. This particular example is the dataflow for a Uintah combustion simulation used by the C-SAFE Center for the Simulation of Accidental Fires and Explosions at the University of Utah. Each LightStream module provides streaming capability through input and output data ports that can be used in a variety of data transfer/sharing modes. In this way, groups of modules can be chained to provide complex processing operations as the data are transferred from the initial source to the final data analysis and visualization stages. This data flow is typically driven by user demands and interactions. A variety of "standard" modules, such



(a) (b)

Figure 3.6: The LightStream Dataflow used for analysis and visualization of a three-dimensional combustion simulation (Uintah code). (a) Several dataflow modules chained together to provide a light and flexible stream processing capability. (b) One visualization that is the result from this dataflow.

as data differencing (for change detection), content based image clustering (for feature detection), or volume rendering with multiple, science-centric transfer functions, are part of the base system. These can be used by new developers as templates for their own progressive streaming data processing modules.

ViSUS also provides a scene graph hierarchy for both organizing objects in a particular environment, as well as the sharing and inheriting of parameters. Each component in a model is represented by a node in this scene graph and inherits the transformations and environment parameters from its parents. Three-dimensional volume or two-dimensional slice extractors are children of a data set node. As an example of inheritance, a scene graph parameter for a transfer function can be applied to the scene graph node of a data set. If the extractor on this data set does not provide its own transfer function, it will be inherited.

3.4.2 Portable Visualization Layer - ViSUS AppKit.

The visualization component of ViSUS was built with the philosophy that a single code base can be designed to run on a variety of platforms and hardware ranging from mobile devices to powerwall displays. To enable this portability, the basic draw routines were designed to be OpenGL ES compatible. This is a limited subset of OpenGL used primarily for mobile devices. More advanced draw routines can be enabled if a system's hardware can support it. In this way, the data visualization can scale in quality depending on the available hardware. Beyond the display of the data, the underlying graphical user interface (GUI) library can hinder portability to multiple devices. At this time ViSUS has

made use of the Juce [7] library which is lightweight and supports mobile platforms such as iOS and Android in addition to major operating systems. ViSUS provides a demo viewer that contains standard visualizations such as slicing, volume rendering and isosurfacing. Similarly to the example LightStream modules, these routines can be expanded through a well-defined application programming interface (API). Additionally, the base system can display two-dimensional and three-dimensional time varying data. As mentioned above, each of these visualizations can operate on the end result of a LightStream dataflow. The system considers a two-dimensional dataset as a special case of a slice renderer and therefore the same code base is used for two-dimensional and three-dimensional datasets. Combining all of the above design decisions allows the same code base to be used on multiple platforms seamlessly for data of arbitrary dimensions. Figure 3.7 shows the same application and visualization running on an iPhone 3G mobile device and a powerwall display.

3.4.3 Web-Server and Plug-In

ViSUS has been extended to support a client-server model in addition to the traditional viewer. The ViSUS server can be used as a standalone application or a web server plugin module. The ViSUS server uses HTTP (a stateless protocol) in order to support many clients. A traditional client/server infrastructure, where the client established and maintained a stable connection to the server, can only handle a limited number of clients robustly. Using HTTP, the ViSUS server can scale to thousands of connections. The ViSUS client keeps a number (normally 48) of connections alive in a pool using the "keep-alive" option of HTTP. The use of lossy or lossless compression is configurable by the user. For example, ViSUS supports JPEG and EXR for lossy compression of byte and floating point data, respectively. The ViSUS server is an open client/server architecture, therefore it is possible to port the plugin to any web server which supports a C++ module (i.e., Apache, IIS). The ViSUS client can be enabled to cache data to local memory or to disk. In this way, a client can minimize transfer time by referencing data already sent, as well as having the ability to work offline if the server becomes unreachable. The ViSUS portable visualization framework (Appkit) also has the ability to be compiled as a Google Chrome, Microsoft Internet Explorer, or Mozilla Firefox web browser plugin. This allows a ViSUS framework based viewer to be easily integrated into web visualization portals.

---

[7] http://www.rawmaterialsoftware.com

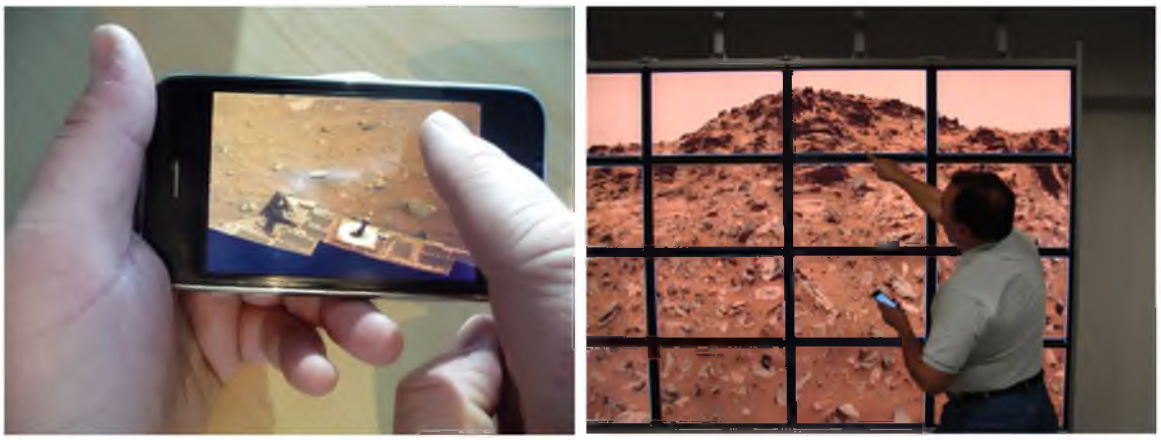


Figure 3.7: The same application and visualization of a Mars panorama running on an iPhone 3G mobile device (a) and a powerwall display (b). Data courtesy of NASA.

3.4.4 Additional Application: Real-Time Monitoring

In addition to the ones provided in this dissertation, the ViSUS framework has an additional ideal application in the real-time monitoring of large scientific simulations. Ideally, for these simulations a user-scientist would like to view a simulation as it is computed, in order to steer or correct the simulation as unforeseen events arise. Simulation data are often very large. For instance, a single field of a time-step from the S3D combustion simulation in Figure 3.8 (a) is approximately 128 gigabytes in size. In the time needed to transfer this single time-step, the user-scientist would have lost any chance for significant steering/correction of an ongoing simulation or at least the ability to save wasting further resources by terminating a simulation early. By using the parallel ViSUS data format in simulation checkpointing [98, 99], we can link this data directly with an Apache server using a ViSUS plug-in running on a node of the cluster system. By doing this, user-scientists can visualize simulation data as checkpoints are reached. ViSUS can handle missing or partial data; therefore, the data can be visualized even as it is being written to disk by the system.

ViSUS's support for a wide-variety of clients (a stand-alone application, a web-browser plug-in, or an iOS application for the iPad or iPhone) allows the application scientist to monitor a simulation as it is produced, on practically any system that is available without any need to transfer the data off the computing cluster. As mentioned above, Figure 3.8 (a) is an S3D large-scale, combustion simulation visualized remotely from an high performance

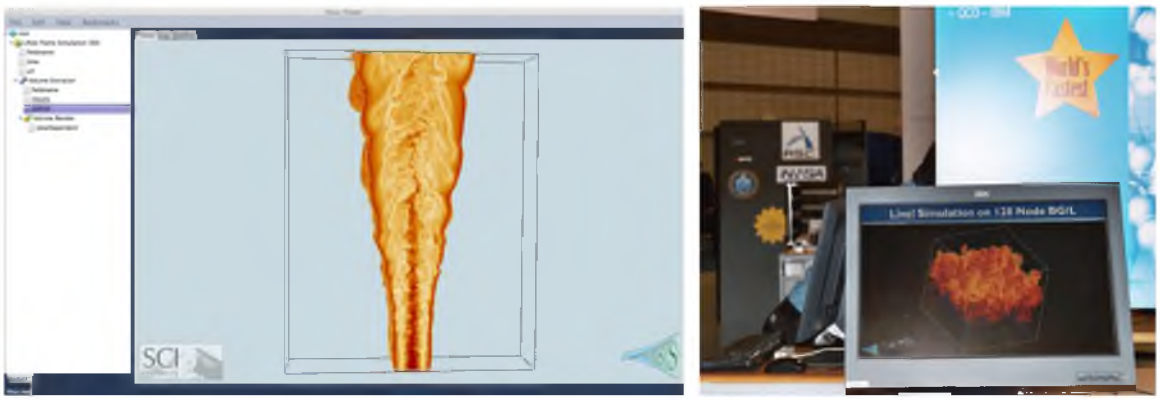


Figure 3.8: Remote visualization and monitoring of simulations. (a) An S3D combustion simulation visualized from a desktop in the Scientific Computing and Imaging (SCI) Institute (Salt Lake City, Utah) during its execution on the HOPPER 2 high performance computing platform in Lawrence Berkeley National Laboratory (Berkeley, California). (b) Two ViSUS demonstrations of LLNL simulation codes (Miranda and Raptor) visualized in real-time while executed on the BlueGene/L prototype installed at the IBM booth of the Supercomputing exhibit.

computing platform[8].

This work is the natural evolution of the ViSUS approach of targeting practical applications for out-of-core data analysis and visualization. This approach has been used for direct streaming and real-time remote monitoring of the early large scale simulations such as those executed on the IBM BG/L supercomputers at Lawrence Livermore National Laboratory (LLNL) [130] shown in Figure 3.8 (b). This work continues its evolution towards the deployment of high performance tools for in situ and postprocessing data management and analysis for the software and hardware resources of the future including exascale DOE platforms of the next decade[9].

[8]Data are courtesy of Jackie Chen at Sandia National Laboratories, Combustion Research Facility `http://ascr.sandia.gov/people/Chen.htm`

[9]Center for Exascale Simulation of Combustion in Turbulence (ExaCT) `http://science.energy.gov/ascr/research/scidac/co-design/`

CHAPTER 4

INTERACTIVE SEAM EDITING AT SCALE

This chapter outlines the *Panorama Weaving* technique which brings the boundary computation phase of panorama creation pipeline into an interactive environment. Section 4.1 gives the relevant background and formulation for the computation of optimal image boundaries. Section 4.2 discusses how to achieve interaction with pairwise boundaries. Section 4.3 introduces the adjacency mesh data structure and how it can be used to bring pairwise seams to a global seam solution. Section 4.4 details how to extend the *Panorama Weaving* technique to an out-of-core environment, thereby scaling the technique to gigapixel images. In Section 4.5, I will detail how to design an interactive system using this technique and scale it to large images in Section 4.6. Finally, Section 4.7 provides results for the technique and in Section 4.8 I discuss its limitations.

4.1 Optimal Image Boundaries

In this section, we discuss the technical background for boundary calculations of both pairwise and many-image panoramas.

4.1.1 Optimal Boundaries

Given a collection of n panorama images $I_1, I_2..I_n$ and the panorama P, the image boundary problem can be thought of as finding a discrete labeling $L(p) \in (1...n)$ for all panorama pixels $p \in P$, which minimizes the transition between each image. If $L(p) = k$, this indicates that the pixel value for location p in the panorama comes from image I_k. This transition can be defined by an energy on the piecewise smoothness $E_s(p, q)$ of the labeling of neighboring elements $p, q \in \mathcal{N}$, where $\mathcal{N}$ is the set of all neighboring pixels. We would like to minimize the sum of the energy of all neighbors, E. For the panorama boundary problem, this energy is typically [4] defined as:

$$E(L) = \sum_{p,q \in \mathcal{N}} E_s(p,q)$$

If minimizing the transition in pixel values:

$$E_s(p,q) = \|I_{L(p)}(p) - I_{L(q)}(p)\| + \|I_{L(p)}(q) - I_{L(q)}(q)\|$$

or if minimizing the transition in the gradient:

$$E_s(p,q) = \|\nabla I_{L(p)}(p) - \nabla I_{L(q)}(p)\| + \|\nabla I_{L(p)}(q) - \nabla I_{L(q)}(q)\|$$

where $L(p)$ and $L(q)$ are the labeling of the two pixels. Notice that $L(p) = L(q)$ implies $E_s(p,q) = 0$. Minimizing the change in pixel value works well in the context of poor registration or moving objects in the scene, while minimizing the gradient produces a nice input for techniques such as gradient domain blending. In addition, techniques can use a linear combination of the two energies.

4.1.2 Min-Cut and Min-Path

When computing the optimal boundary between two images, the binary labeling is equivalent to computing a min-cut of a graph whose nodes are the pixels and arcs connect a pixel to its neighbors. The arc weights are then the energy function being minimized, see Figure 4.1 (a). If we consider a four-neighborhood and the dual-graph of the planar min-cut graph, as we show in Figure 4.1 (b), we can see that there is an equivalent min-path to the min-cut solution on the dual-graph. This has been shown to be true for all single source, single destination paths on planar graphs [72]. The approaches are equivalent in the sense

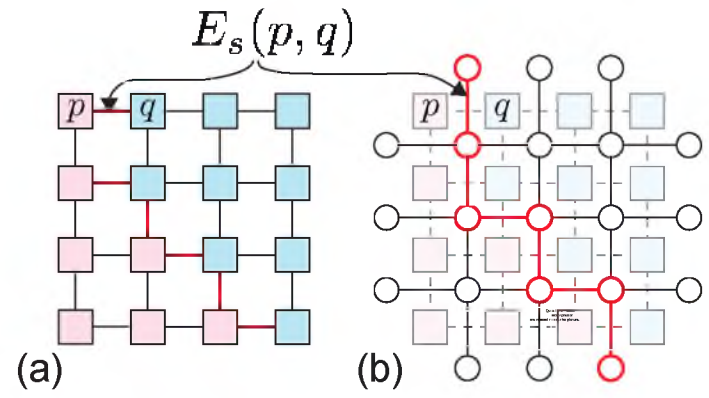


Figure 4.1: The four-neighborhood min-cut solution (a) with its dual min-path solution (b). The min-cut labeling is colored in red/blue and the min-path solution is highlighted in red.

that the final solution of a min-cut calculation defines the pixel labeling $L(p)$ while the min-path solution defines the path that separates pixels of different labeling.

4.1.3 Graph Cuts

This technique provides good solutions to pixel labeling problems for more than two images. The intricacies of the algorithm [26, 24, 92] are beyond the scope of this dissertation, but at a high level, Graph Cuts finds a labeling L which minimizes an energy function $E'(L)$. This function consists of term $E_s(p, q)$ augmented with energy associated with individual pixel locations $E_d(p)$.

$$E'(L) = \sum_{p} E_d(p) + \sum_{p,q \in \mathcal{N}} E_s(p, q)$$

For the panorama boundary problem, this data energy E_d is typically [4] defined as being 0 if location p is a valid pixel of $I_{L(p)}$. Otherwise, it has infinite energy.

4.2 Pairwise Seams and Seam Trees

Figure 4.2 illustrates two example pairwise image seams. In the simplest and most common case, Figure 4.2 (a), the boundary lines of the two images intersect at two points u and v connected by the seam s. The other simple, but more general case in Figure 4.2 (b) shows two overlapping images, where the intersection of their boundary lines results in an even number of intersection points. A set of seams can be built by connecting pairs of points with a consistent winding. The seams computed in this way define a complete partition of the space between the two images. In nonsimple cases, i.e., with co-linear boundary intervals, we can achieve the same result by choosing one representative point (possibly optimized to minimize an energy). Notice that the case in Figure 4.2 (b) produces more than a single set of valid seams, denoted by the purple and grey dashed lines. For clarity in the discussion, we will focus on the case in Figure 4.2 (a) since we can treat each seam of the case in Figure 4.2 (b) as independent.

Assuming the dual-path energy representation in Figure 4.1 (b), a seam is a path that connects the intersection points (u, v). Computing the minimal path of a given energy function will give an optimal seam s, which can be computed efficiently with Dijkstra's algorithm [50]. With minimal additional overhead, we can compute both min-path trees T_u and T_v from u and v (single source all paths). These trees provide all minimal seams which originate from either endpoint and define the *dual seam tree* of our technique. Given a point in the image overlap, we can find its minimal paths to u and v with a linear walk

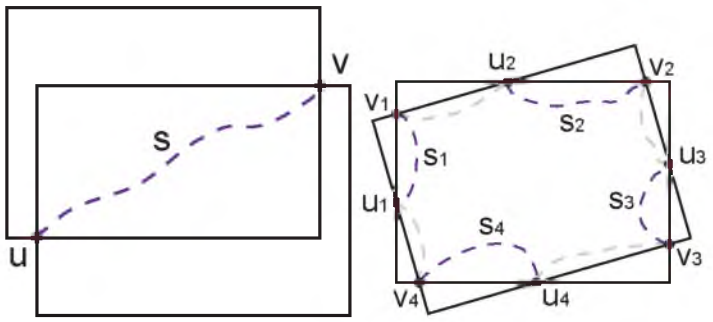


Figure 4.2: (a) Given a simple overlap configuration a seam can be thought of as a path s that connects pairs of boundary intersections u and v. (b) Even in a more complicated case, a valid seam configuration is still computable by taking pairs of intersections with a consistent winding about an image boundary. Note that there is an alternate configuration denoted in gray.

up the trees T_u and T_v, as shown in Figure 4.3. If this point is a user constraint, the union of the two minimal paths forms a new constrained optimal seam. Due to the simplicity of the lookup, this path computation is fast enough to achieve interactive rates even for large image overlaps. Note that two min-paths on the same energy function are guaranteed not to cross. Although, since each dual-seam tree is computed independently, the minimal paths from a constraint (to u and v) can cross. In particular, if the trees computed by Dijkstra's algorithm are dependent on the order in which the edges are calculated and there are multiple paths in an overlap that share the same energy, the paths on each tree to a user constraint can cross. To avoid this problem we enforce an ordering based on the edge index and we are guaranteed to achieve noncrossing solutions.

Moving an endpoint is also a simple walk up its partner endpoint's seam tree. Therefore

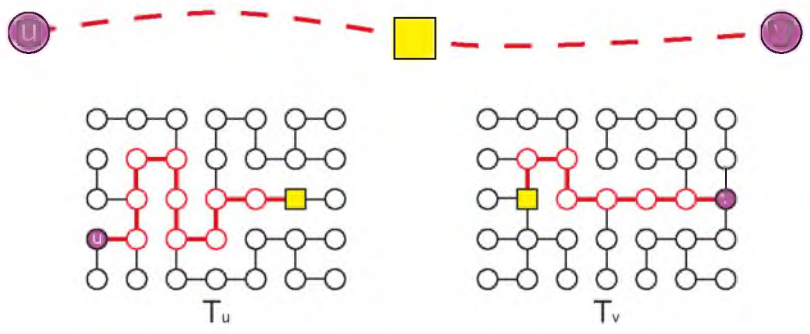


Figure 4.3: Given two min-path trees associated with a seam's endpoints (u,v), a new seam that passes through any point in the overlap (yellow) is a simple linear walk up each tree.

a user can change an endpoint location at-will, interactively. Although after the movement, the shifted endpoint's seam tree is no longer valid since it was based on a previous location. If future interactions are desired, the tree must be recomputed. This can be computed as a background process after the users finish their initial interaction without any loss of responsiveness to the system.

4.3 From Pairwise to Global Seams

To avoid incurring the cost associated with the solution of a global optimization, we build the panorama as a proper collection of pairwise seams. This is based on the observation, illustrated in Figure 4.4 (a), that the label assignment in a Graph Cut optimization mostly forms a simple collection of regions partitioned by pairwise image seams (denoted in the picture by the double-arrows).

Our technique is designed with this property in mind and independently computes each seam constrained by the pairwise intersections called *branching points*. These are colored in red in Figure 4.4 (b).

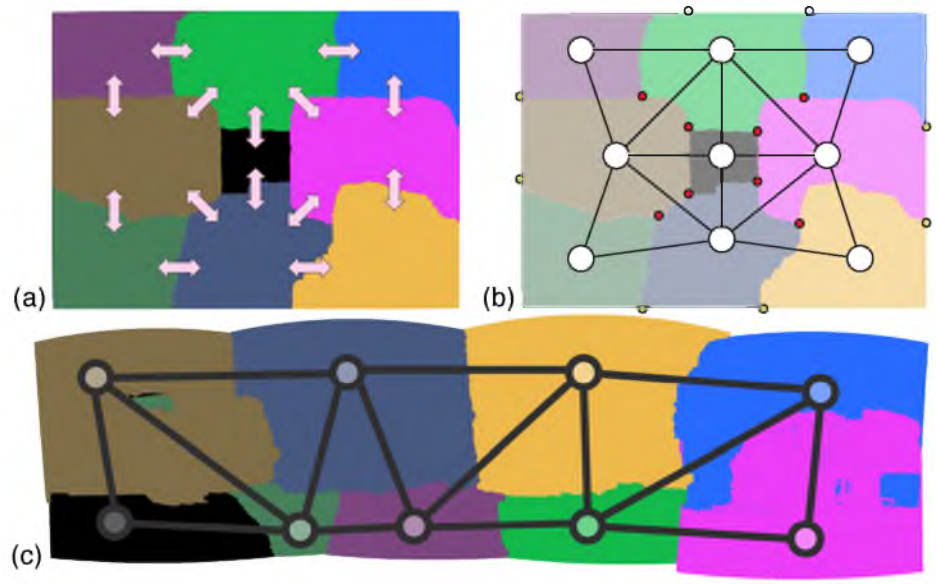


Figure 4.4: (a) A solution to the panorama boundary problem can be considered as a network of pairwise boundaries between images. (b) Our adjacency mesh representation is designed with this property in mind. Nodes correspond to panorama images, edges correspond to boundaries and branching points (intersections in red) correspond to faces of the mesh. (c) Graph Cuts optimization can provide more complex pixel assignments where "islands" of pixels assigned to one image can be completely bounded by another image. Our approach simplifies the solution by removing such islands.

Note that the solution of a Graph Cuts optimization can provide more complex pixel assignments, where "islands" of pixels assigned to one image can be completely bounded by another image, as shown in Figure 4.4 (c). Obviously, our approach simplifies the solution by removing such islands and makes each region simply connected. We have checked how the energy optimized by our technique would change with this assumption (see Section 4.7). In all cases we have noticed that the energy of the seams produced by our system remains in the same order of magnitude as Graph Cuts, actually being reduced in all cases but one. Limitations on this assumption are detailed in Section 4.8.

4.3.1 The Dual Adjacency Mesh

To construct a seam network, our computations are driven by an abstract structure that we call the *dual adjacency mesh*. We draw the inspiration for our adjacency mesh representation from the traditional region adjacency graph used in computer vision, as well as the regions of difference (ROD) graphs of Uyttendaele et al [165]. In Figure 4.4 (b and c), we have the adjacency graph for a global, Graph Cuts computation. This graph can be considered the dual to the seam network: each node corresponds to an image in the panorama, whereas each edge describes an overlap relation between images. Edges are then *orthogonal* to the seam they represent. If we consider this graph as having the structure of a mesh, the dual of the panorama branching points are the *faces* of this mesh representation. In Figure 4.4 (b), the branching points are highlighted in red. Seams which exit this mesh representation correspond to pairwise overlaps on the panorama boundary. These are illustrated in Figure 4.4 (b) with a single yellow endpoint. Connecting the branching points on adjacent faces in the mesh and/or the external endpoints gives a global seam network of pairwise image boundaries.

In addition to the branching points in the seam network, the faces of the adjacency mesh are also an intuitive representation for *overlap clusters*. Specifically, clusters are groups of overlaps that share a common area that we call a *multioverlap*. These multioverlaps are areas where branching points must occur. The simplest multioverlap beyond the pairwise case consists of three overlaps and is represented by a triangle, see Figure 4.5 (a). A multioverlap with four pairwise overlaps, can be represented by a quadrilateral, indicating that all four pairwise seams branch at a mutual point. An important property of this representation is that this quadrilateral can be split into two triangles, a classic subdivision, see Figure 4.5 (b). Any valid (no edge crossing) subdivision of a polygon in this mesh will result in

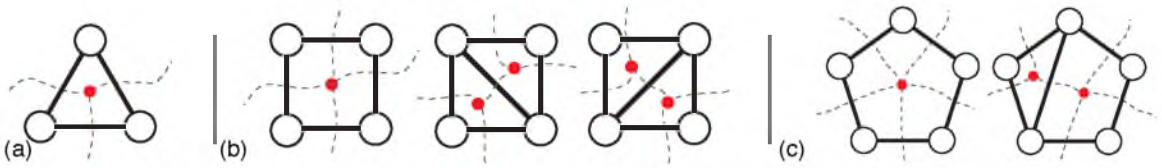


Figure 4.5: (a) A three overlap adjacency mesh representation. (b) A four overlap initial quadrilateral adjacency mesh with its two valid mesh subdivisions. (c) A five overlap pentagon adjacency mesh with an example subdivision.

a valid seam configuration. In this way, the representation can handle a wide range of seam combinations, but keep the overall network valid. Figure 4.5 (c) shows an example subdivision of a five-way intersection.

As a precomputation, we calculate the initial adjacency mesh consisting of simple n-gon face representations for every n-way cluster. This precomputation stage enables the conversion of the initial nonplanar full neighborhood graph into a planar mesh representation, see Figure 4.6. Clusters (and their corresponding muti-overlaps) by definition are nonoverlapping, maximal cliques of the full neighborhood graph. This computation is a classic clique problem and is known to be NP-complete [43]. For most panoramas, we have found the neighborhood graph is small enough that a brute-force search can be computed quickly. Although, previous work has shown that given a graph with a polynomial bound on the number of maximal cliques, they can be found in polynomial time [141]. This is indeed the case for the neighborhood graph which has maximal boxicity [140] dimension of two [36]. After the maximal cliques have been found, each n-gon face is extracted by finding the fully spanning cycle of clique vertices on the boundary in relation to the centroids of the images. The boundary edges of the n-gon face are marked as *active*, while the interior (intersecting) edges are marked as *inactive* as shown in Figure 4.6.

This adjacency mesh is used to drive the computation and combination of the pairwise boundaries as well as user manipulation. As we will illustrate in Section 4.5, it can be completely hidden from a user of the interactive system with intuitive editing concepts.

4.3.2 Branching Points and Intersection Resolution

Given a collection of seam trees that correspond to active edges in the adjacency mesh, we can now combine the seams into a global seam network. To do this, we need to compute the branching points which correspond to each adjacency mesh face, adjust the seam given a possible new endpoint, and resolve any invalid intersections that may arise (in order to maintain consistency).

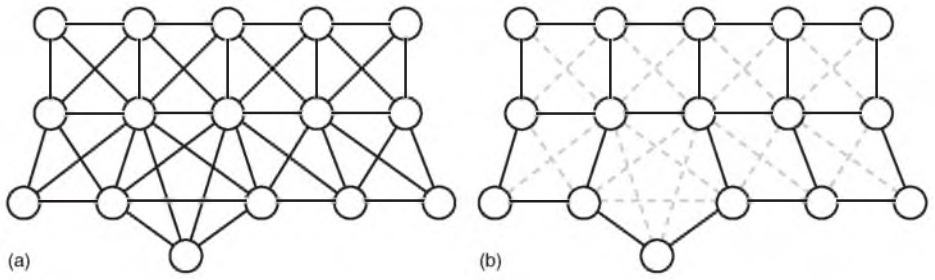


Figure 4.6: Considering the full neighborhood graph of a panorama (a), where an edge exists if an overlap exists between a pair of images, an initial valid adjacency mesh (b) can be computed by finding all nonoverlapping, maximal cliques in the full graph, then activating and deactivating edges based on the boundary of each clique.

4.3.2.1 Branching points. Assuming for each pairwise seam there exists only two endpoints, for each multioverlap one endpoint must be adapted into a branching point. We refer to this endpoint as being *inside* in relation to the adjacency mesh face. The other seam endpoint is considered to be *outside* in relation to the multioverlap. These can be computed by finding the endpoints which are closest (euclidean distance) to the multioverlap associated with the face. Figure 4.7 (a) displays these endpoints with the color red and the multioverlap area with a blue shading. Although it is possible to create a pathological overlap configuration where this distance metric fails, we have found that this strategy works well in practice.

If we use the dual seam tree distances, i.e., the path distance values associated with the *outside* endpoints, we can compute a branching point which is optimal with respect to these paths, as illustrated in Figure 4.7 (b). This can be accomplished with a simple lookup of the distance values in the trees. We have found that minimizing the sum of the least squared error provides a nice low energy solution. The new path associated with a moved endpoint is determined by a simple walk up the dual seam tree, see Figure 4.7 (c). Additionally, each seam tree associated with the branching point is recalculated given its location. As Figure 4.7 (d) illustrates, the branching point is always computed using the distance field of the initial endpoint location even if this point had been previously adjusted by an adjacent face. In practice, we have found the contribution of the root location is minimal to the overall structure of the seam tree towards the leaves of the tree. Since using the initial starting endpoints allows the branching points to be computed independently and in a single parallel pass, we have adopted this into our technique.

The seams produced by this initial process in the four-overlap case are similar to the

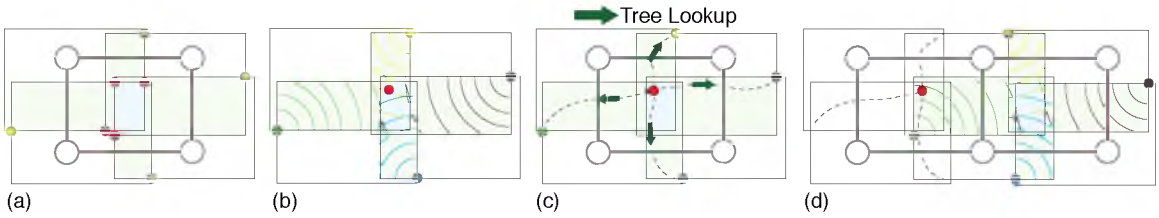


Figure 4.7: (a) Pairwise seam endpoints closest to a multioverlap (red) are considered a branching point. (b) This can be determined by finding a minimum point in the multioverlap with respect to min-path distance from the partner endpoints. (c) After the branching point is found, the new seams are computed by a linear lookup up the partner endpoint's seam tree. (d) To enable parallel computation, each branching point is computed using the initial endpoint location (green) even if it was moved via another branching point calculation (red).

sequential techniques introduced by Efros and Freeman [53] and Cohen et al. [42]. With the additional adjacency mesh, our technique is much more expressive in the possible seam configurations (especially allowing arbitrary valence branching points). In addition, as we will illustrate next, for panoramas and especially in an interactive setting one cannot assume that a seam's path to a branching point respects the paths of other seams.

4.3.2.2 Removing invalid intersections. Since each seam is computed using a separate energy function, seam-to-seam intersections beyond the branching points are possible. Small intersections of this type must be allowed to ensure solutions are computable in a four-neighborhood configuration. For instance, there would be no nonintersecting way to combine five seams into a single branching point. This allowance is defined by an ϵ-neighborhood around the branching point which can be set by the user. We have found allowing an intersection neighborhood of one or two pixels gives good results with no visible artifacts from the intersection. The intersections in this neighborhood are collapsed to be co-linear to the shortest of the intersecting paths.

Intersections that occur outside of this ϵ-neighborhood must be resolved due to the inconsistent pixel labeling that they imply. Figure 4.8 (a, c) shows an example of intersections in a four-way image overlap. The areas highlighted in gray have conflicting image assignments. Enforcing no intersections at the time of the seam computation would complicate parallelism and be overly expensive. This corresponds to a k-way planar escape problem with multiple energies (where k is the number of seams incoming to the branching point) for which variants have been shown to be NP-complete [179]. This could also lead to possible unstable interactions since small movements may lead to extremely large changes in the overall seam paths. The simplest solution is to choose one assignment per conflict

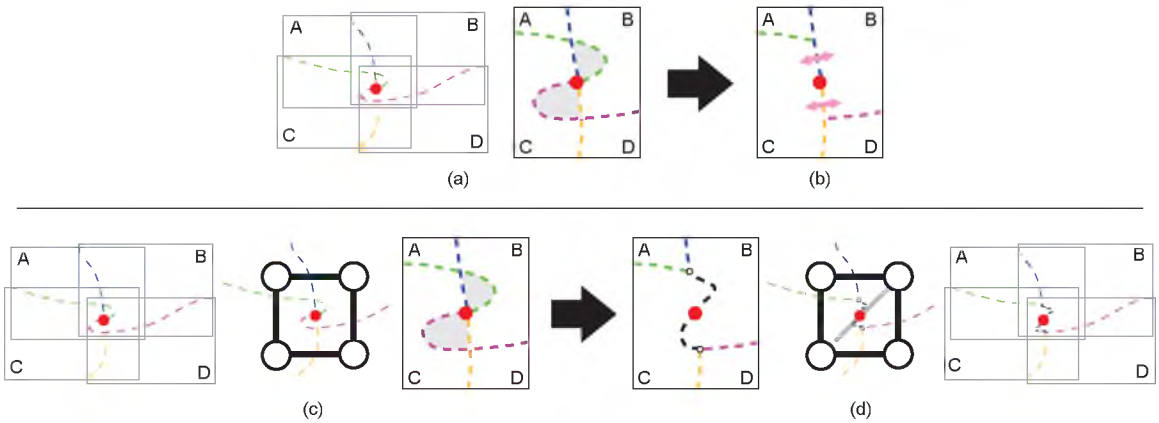


Figure 4.8: (a) Pairwise seams may produce invalid intersections or crossings in a multioverlap, which leads to an inconsistent labeling of the domain. The gray area on the top can be given the labels A or B and on the bottom either C or D. (b) Choosing a label is akin to collapsing one seam onto the other. This leads to new image boundaries, which were based on energy functions that do not correlate to this new boundary. The top collapse results in a B-C boundary using an A-B seam (C-D seam for the bottom). (c and d) Our technique performs a better collapse where each intersection point is connected to the branching point via a minimal path that corresponds to the proper boundary (B-C). One can think of this as a virtual addition of a new adjacency mesh edge (B-C) at the time of resolution to account for the new boundary.

area. This is equivalent to collapsing the area and making the two seams co-linear at points where they "cross." Each collapse introduces a new image boundary for which the wrong energy function has been minimized, Figure 4.8 (a, b). In our technique, we perform a more sophisticated collapse.

For a given pair of intersecting seams, multiple intersections can be resolved by taking into account only the furthest intersection from the branching point in terms of distance on the seam. Given that each seam divides the domain, this intersection can only occur between seams that divide a common image. If presented with a seam-to-seam intersection, we can easily compute the new boundary that is introduced during the collapse. This is simply a *resolution* seam on an overlap between the images which is not common between the intersection seams. The resolution seams connect the intersection points with the branching points. Often, if multiple resolution seams share the same overlap, as in Figure 4.8, only one min-path calculation from the branching point is needed to fill in all min-paths. The new resolution seams are constrained by the other seams in the cluster in order to not introduce new intersections with the new paths. The constraints are also given the ability to gradually increase the allowed intersection neighborhood beyond the user defined ϵ-neighborhood in

the chance that no solution path exists. The crossings and intersections are collapsed in this neighborhood. Due to the rarity of this occurrence, the routine adds minimal overhead to the overall technique in practice. Order matters in both finding the intersections and computing the resolution seams and therefore must be consistent. We have found that ordering based on the overlap size works well. Resolution seams and expanded ϵ-neighborhood are considered to be temporary states. Figure 4.8 (c, d) shows an example of an intersection resolution.

This technique robustly handles possible seam intersections at the branching points. Most importantly, since we are only adjusting the seam from the intersection point on, we can resolve each adjacency mesh face in parallel. In addition, since the seam is not changed outside of the multioverlap within a cluster, the resolution is local and will not cascade to a global resolution. However, it is possible for a user to introduce unresolvable intersections through added constraints, as we will discuss in Section 4.5.

4.4 Out-of-Core Seam Processing

While designed to be light on resources, the technique outlined in the previous sections assumes all images can be stored in-core. This assumption holds true for many panoramas, but not images gigapixels in size. In this section, I will detail how the original technique can be modified to handle large panoramas.

As illustrated in Figure 4.9, the initial seam solution for the *Panorama Weaving* technique can be thought to occur in three phases. First, for each adjacency mesh face, the corresponding branching point must be computed. After the branching point is found, any seams which occur on the panorama boundary can be computed. These correspond to edges in our mesh that belong to only one face. See Figure 4.9 (a). As mentioned previously, the branching point computations for each face given our simple pairwise seam assumption are completely independent and can be computed in parallel. As a second phase, the seams for the shared edges can be found by connecting the newly computed branching points. Each shared edge can be processed independently in parallel. See Figure 4.9 (b). Finally, once all edges for a face are computed the seam intersections for the given face can be resolved. This resolution is also independent and parallel for each adjacency mesh face. See Figure 4.9 (c). Note that each phase need not be entirely distinct since the can be interleaved.

If we bundle the branching point and shared seam calculation into a single operation, the logic of our out-of-core computation would only deal with the adjacency mesh faces. This can drastically reduce the system complexity even when working in a multicore environment.

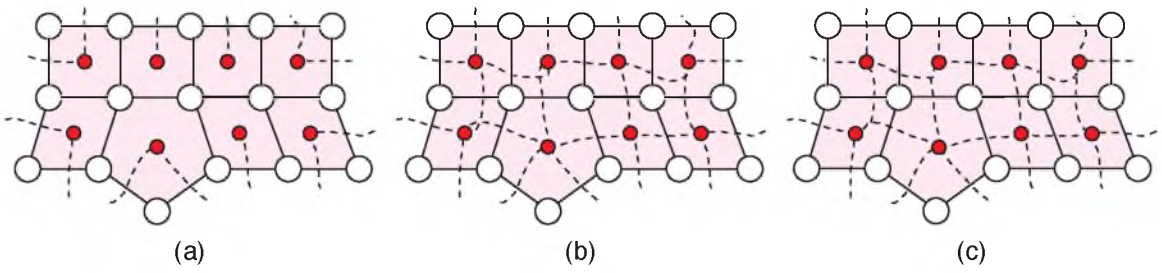


Figure 4.9: The phases of out-of-core seam computation. (a) First, branching points are computed. The seams for all unshared edges can also be computed during this pass. (b) Second, once the corresponding branching points are computed, all unshared edges can be computed with a single min-path calculation. (c) Third, once all the seams for the edges for a given face have been computed, the intersections can be resolved. Note, the three passes do not necessarily need to be three separate phases since they can be interleaved when the proper input data are ready.

By doing this, we also cast our problem into the simple problem of graph traversal for the two remaining phases of our seam processing system. For our traversal strategy, we chose a simple row-major traversal of the mesh faces.

4.4.1 Branching Point and Shared Edge Phase

The design goal for this phase is to keep the memory requirement low and predictable. The reasoning for this approach is twofold. First, low memory requirements enable the portability of our out-of-core system to a wide variety of systems. Such a technique has the ability to run on systems from laptops to HPC computers. Second, when moving into a multicore implementation being able to have a low, predictable memory requirement per face would make the logic and scheduling of the many threads operating on the adjacency mesh faces very simple. Given the resources of the system, we can predict how many faces the system can compute in parallel. We achieve our low memory footprint by computing the branching point for a given face in a "round-robin" fashion, as shown in Figure 4.10. For each edge, the images that correspond to its endpoints are loaded and the overlap energy is calculated. Next, the seam tree needed for the branching point is calculated. After this calculation, the overlap energy is no longer needed and is therefore ejected from memory. The calculation then moves onto the next edge that shares an image with the previous computation. The rest of the branching point calculation proceeds in the same way until all seam trees have been computed to compute the location of the new branching point.

As mentioned above, we couple the shared edge phase into this phase of calculation. This

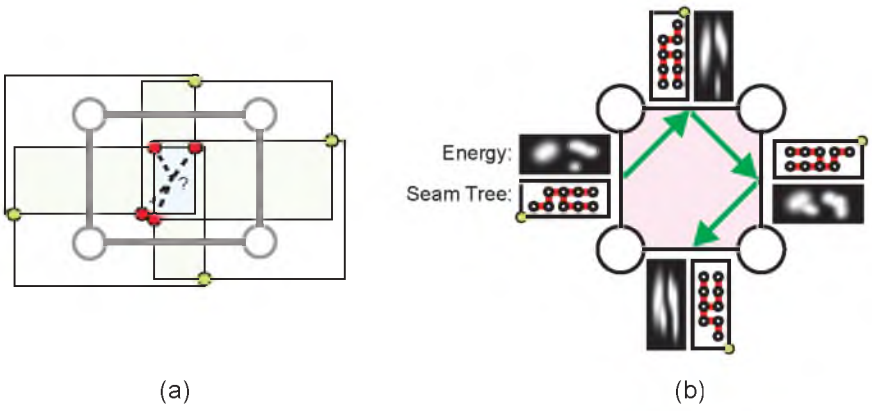


Figure 4.10: The low memory branching point calculation for our out-of-core seam creation technique. (a) Given a face for which a branching point needs to be computed, (b) the computation proceeds "round-robin" on the edges of the face to compute the needed seam trees. The images that correspond to the edge endpoints and overlap energy are only needed during the seam tree calculation for a given edge on the face. Therefore by loading and unloading these data during the "round-robin" computation, the memory overhead for the branching point computation is the cost of storing two images, one energy overlap buffer, and one for the seam trees for the given face.

is done with a flag per face to indicate whether the branching point has been computed. After the branching point has been computed for a face, the flag is set and the process checks to see if the flag is also set for the other faces on a face's shared edges. If the face calculation is the second to set this flag, it knows it can fill in the new seam for the shared edge. For the multicore implementation, this check is made atomic with locks to ensure there is always a clear first- and second-face calculation when setting flags for the endpoints of a shared seam. Note that the memory overhead for this computation is equal to the space required to store two images, one buffer for the overlap energy, and one for the seam trees for the edges of the mesh face. As you can see, this overhead is quite low and given an average image size and overlap percentage, very predictable. The geometry of the seams are stored in-core for the final phase of the calculation.

4.4.2 Intersection Resolution Phase

When all seams for a face have been computed, the seam intersections that correspond to the face can be resolved. The intersection test is computed on the seam geometry which is already in memory. If an intersection occurs within a threshold distance on the seam from the branching point it is considered small and collapsed. This is an equivalent operation to the intersection collapse from the in-core *Panorama Weaving* technique. For larger

intersections, a resolution seam must be computed and the two images that correspond to the overlap of the resolution seam need to be loaded before computation. See Figure 4.11 for an example.

4.5 Weaving Interactive System

In this section, we outline how to create a light and fast interactive system using the *Panorama Weaving* technique. A simplified diagram of the operation of the system is given in Figure 4.12. In Section 4.7, we provide examples of this application editing a variety of panoramas.

4.5.1 System Specifics

In this subsection, I will detail specifics for the prototype system.

4.5.1.1 Input. The system inputs for our prototype are flat, registered raster images with no geometry save the image offset. Any input can be converted into this format and therefore it is the most general. The initial image intersection computation is computed using the rasterized boundaries. Due to aliasing, there may be many intersections found. If the intersections are contiguous, they are treated as the same intersection and a representative point is chosen. In practice, we have found this choice has little effect on the seam outside a small neighborhood (less than 10 pixels from the intersection). Therefore, the system picks the minimal point in this group in terms of the energy. Pairs of intersections that are very close in terms of euclidean distance (less than 20 pixels) are considered to be infinitesimally small seams and are ignored.

The user is also allowed to dictate the energy function for the entire panorama, image, or overlap. This can be done as an initial input parameter or within the interactive program

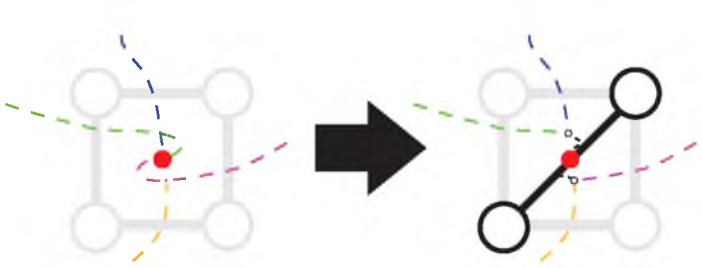


Figure 4.11: For intersections that require a resolution seam, the two images which correspond to the overlap needed for the seam must be loaded. In the figure above, these images are the ones that correspond to the endpoint of the diagonal, resolution adjacency mesh edge.

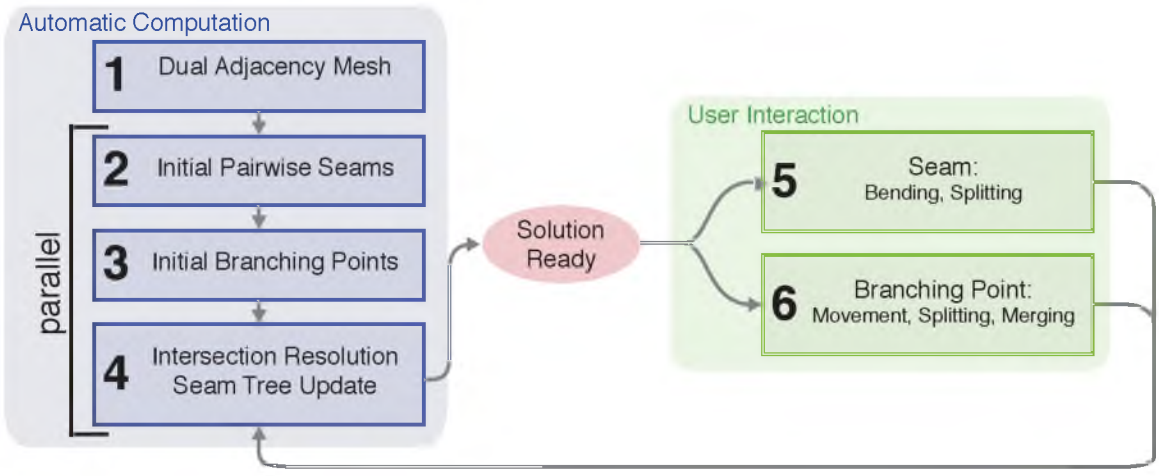


Figure 4.12: Overview of *Panorama Weaving*. The initial computation is given by steps one through four, after which the solution is ready and presented to the user. Interactions, steps five and six, use the tree update in step four as a background process. Additionally, step six updates the dual adjacency mesh.

itself. Specifically, our prototype allows the user to switch between pixel difference or gradient difference energies.

4.5.1.2 Initial parallel computation. Parallel computation is accomplished using a thread pool equal to the number of available cores. The initial dual seam tree and branching points computation can be run trivially in parallel. In the presence of two adjacent faces in the adjacency mesh, a mutex flag must be used on their shared seam since both faces may attempt to write this data simultaneously. As a final phase, each adjacency mesh face resolves intersections in parallel. In order to compute these resolutions in parallel, we split a seam's data into three separate structures for the *start*, *middle*, and *end* of the seam. The *middle* seam contains the original seam before intersection resolution and its extent is maintained by pointers. The structure's *start* and *end* are updated with the intersection resolution seams by the faces associated with their respective branching points. Either vector can be associated only with one face; therefore, we run no risk of multiple threads writing to the same location.

Each seam tree is stored as two buffers: one for node distance and one which encodes the tree itself. The tree buffer encodes the tree with each pixel referencing its parent. This can be done in 2 bits (rounded to 1 byte for speed) for a four-pixel neighborhood. Therefore, for float distances we need only 5 bytes per pixel to store a seam tree.

4.5.1.3 Seam network import. It is possible to import a seam network computed with an alternative technique (such as Graph Cuts, see Figure 4.13), and edit it with our

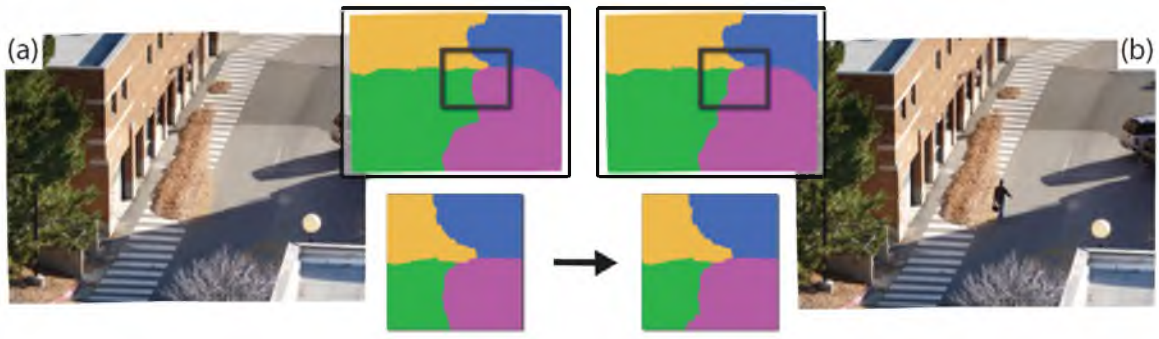


Figure 4.13: Importing a seam network from another algorithm. The user is allowed to import the result generated by Graph Cuts (a) and adjust the seam between the green and purple regions to unmask a moving person (b). Note that this edit has only a local effect, and that the rest of the imported network is unaltered.

system. Our import procedure works as follows. Given a labeling of the pixels of the panorama, the algorithm first extracts the boundaries of the regions. Then, branching points (boundary intersections) are extracted. Next, each boundary segment (bounded by two branching points) is identified as a seam and the connected components of the resulting seam network are identified. To be compatible with our framework, only the seam networks made of a single connected component can be imported. Thus, we only consider the biggest connected component of the network and small islands are discarded. Finally, our seam data-structures are fed with the seam network and the adjacency mesh is updated if necessary. Since the editing operations do not cascade globally, a user can edit a problem area locally and maintain much of the original solution if desired.

4.5.2 Interactions

In this subsection, I will detail some possible interactions that can be accomplished with our system.

4.5.2.1 Seam bending. The adding of a constraint and its movement is called a *bending* interaction in our system and operates as outlined in Section 4.2. A user is allowed to add a constraint to a seam and is provided instantly the optimal seam which must pass through it. The constraint can be moved interactively to explore the seam solution space. Intersections in any adjacency mesh face containing the corresponding edge are resolved, which can be done in parallel. Most importantly, given how the technique resolves intersections, seams cannot change beyond the multioverlap area in these faces. Therefore, the seam resolution does not cascade globally.

4.5.2.2 Seam splitting. Adding more than one constraint is akin to *splitting* the seam into segments. After a *bending* interaction, the seam trees are split into four, where there were previously two. Two of the trees (corresponding to the endpoints) are inherited by the new seams. The two trees associated with the new constraint are identical, therefore only one tree computation is necessary. Splitting occurs in our prototype when a user releases the mouse after a bending interaction. Editing is locked for this seam until the corresponding trees are resolved. This is a quick process and it is very rare for a user to be fast enough to beat the computation.

4.5.2.3 Branching point movement. The user is given the ability to grab and move the branching point associated with a selected face of the adjacency graph As I have detailed in Section 4.2, a movement of an endpoint is a simple lookup on its partner's dual seam tree. As the user moves a branching point, intersections for both the selected face and all adjacent faces are resolved. Given that the intersection resolution does not adjust seam geometry beyond the multioverlap, we need only to look at this one-face neighborhood and not globally. To enable further interaction, the seam trees associated with this endpoint need to be recalculated after movement. When the user releases the mouse, the seam tree data for all the endpoints associated with the active seams for the face are recomputed as a background process in parallel. Like splitting, editing is locked for each seam until it completes the seam tree update.

4.5.2.4 Branching point splitting and merging. The user can add and remove additional panorama seams by splitting and merging branching points. Addition and removal of seams is equivalent to subdividing and merging faces of the adjacency mesh. Improper requests for a subdivision or merge correlate to a non-valid seam network and are therefore restricted. If splitting is possible for a selected branching point, the user can iterate and choose from all possible subdivisions of the corresponding face. To maintain consistent seams, merging is only possible between branching points resulting from a previous split. In other words, merging faces associated with different initial adjacency mesh faces would lead to an invalid seam configuration since the corresponding images do not overlap. If a seam is added, its dual seam tree is computed. In addition, the other active seams associated with this face will need to be updated much like a branching point movement.

4.5.2.5 Improper user interaction. Given the editing freedom allowed to users, they may move a seam into a inconsistent configuration. Figure 4.14 illustrates some examples. Rather than constrain the user, the prototype system either tries to resolve the

improper seams or if that is not possible give the user visual feedback indicating a problem configuration. For example, if the user introduces a seam intersection, our intersection routine is launched to resolve it, see Figure 4.14 (a). Crossing branching points, Figure 4.14 (b), can be resolved similarly. Figure 4.14 (c) illustrates a configuration with no resolution. In this instance, the crossing edges are collapsed and the user is given a visual hint that there is a problem.

4.6 Scalable Seam Interactions

Given the locality of the interactions in the *Panorama Weaving* technique, extending the interactions to gigapixel sized images does not require a large change to the base interactions. Our interaction scheme is based on a standard large image viewer and on-the-fly loading and computation of the data needed for seam editing. As Figure 4.15 shows, due to the technique's simple pairwise seam assumption, the data which needs to be loaded and computed is local given an interaction. Our system works as follows: we leverage the large image, out-of-core ViSUS system outlined in Chapter 3 to provide exploration of a flattened gigapixel image created from the seams from our initial out-of-core seam computation. If a user wishes have a seam or branching point interaction, she/he can initiate this edit my selecting a seam area she/he wished to edit. The action is determined similarly to the in-core seam system in that the overlap bounding boxes are tested against the user selected area. If all the overlap bounding boxes for a given face are selected, then it is assumed the user wishes to have a branching point manipulation. Otherwise, it is assumed the user wishes a seam bending interaction and a single overlap from the selection is chosen. A user can cycle through the single selection options if more than one is present for a given input. A brute-force bounding box intersection test has the possibility of being overly expensive for panoramas which contain thousands of images and overlaps, therefore, we have designed two

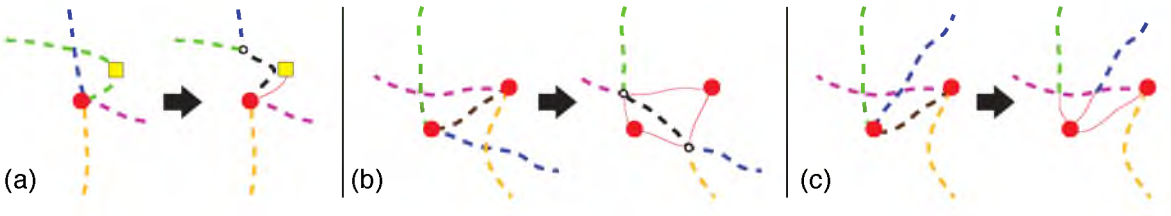


Figure 4.14: Improper user constraints are resolved or if resolution is not possible, given visual feedback. (a) Resolution of an intersection caused by a user moving a constraint. (b) Resolution of an intersection caused by a user moving a branching point. (c) A non-resolvable case where a user is just provided a visual cue of a problem.

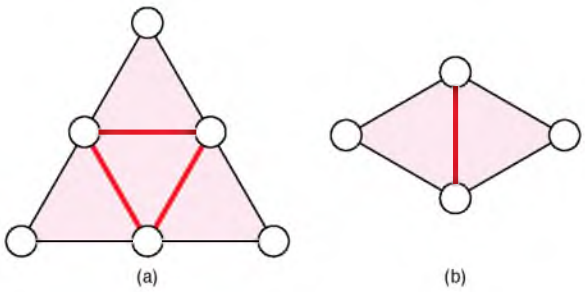


Figure 4.15: Given the inherent locality of the seam editing interactions, only a very small subset of the adjacency mesh needs to be considered. (a) For operations on an adjacency mesh face (i.e., branching point operations) only the images and overlaps of the corresponding face and its one face neighborhood need to be loaded and computed. (b) For edge operations (i.e., bending), we need consider only the faces that share the edge.

more scalable options for the selection. As Figure 4.16 (a) illustrates, a bounding hierarchy of overlaps can be built by merging the bounding boxes of pairs of neighboring adjacency mesh faces. During a user selection, the hierarchy is traversed to determine which faces need to be considered for selection. Once this is determined, the selected face's overlaps are testing for user selection. This provides a selection runtime that is logarithmic in the number of faces in our panorama. Alternatively, as Figure 4.16 (b) shows, if there is a pixel to image map this can be leveraged to determine the neighborhood of overlaps that need to be tested for selection. This neighborhood consists of the edges of faces that share the node that corresponds to the pixel-map image. After the user finishes their interaction, the

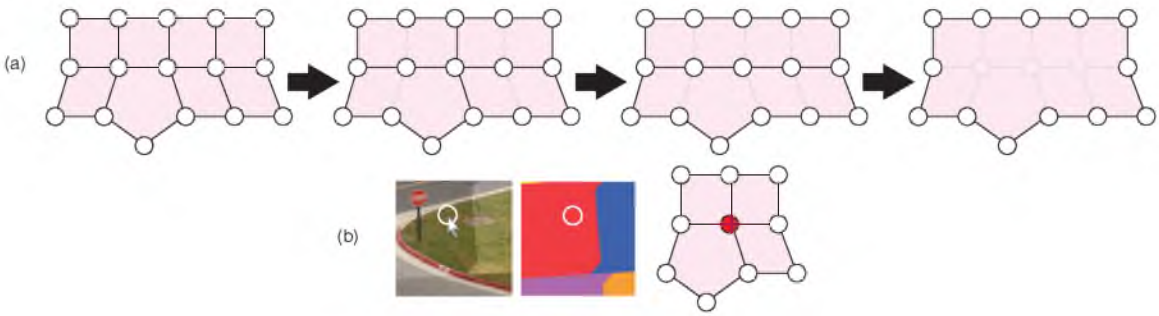


Figure 4.16: When a user selects an area of a panorama to edit, the system must determine which overlaps intersect with the selected area. This can be accomplished with a (a) bounding hierarchy of the overlaps. During selection this hierarchy is traversed to isolate the proper overlaps for the selection. This gives a logarithmic lookup with respect to the number of adjacency mesh faces in the panorama. Alternatively, (b) if a pixel-to-image labeling is provided, this can be used to isolate a fixed neighborhood that needs to be tested for overlap intersection. This labeling is commonly computed if the panorama is to be fed into a color correction routine after seam computation.

seams are saved and the loaded images are masked and saved to the flattened image.

4.7 Results

In this section, we detail the results in both the creation and editing phases of our system. In-core results were performed on a 3.07 GHz Intel i7 four-core processor (with Hyperthreading) with 24 gigabytes of memory. The large system memory was required in order to run the Graph Cuts implementation, as is, on all datasets. *Panorama Weaving* performed well for all datasets on test systems including laptops with only 4 gigabytes of memory. Out-of-core results were run on 2.67GHz Xeon X5550 (eight cores) system with 24 gigabytes of memory.

4.7.1 Panorama Creation

This subsection details the results for in-core and out-of-core initial seam creation.

4.7.1.1 In-core results. We compare the panorama creation phase of our system to the implementation provided by the authors of the Graph Cuts technique [26, 24, 92], which many consider the exemplary implementation. Both α expansion and swap algorithms were run until convergence to guarantee minimal errors and the best time is reported. Since this implementation has various ways of passing data and smoothness terms, we tested all and report the fastest, which is precomputed arrays for the costs with a function pointer acting as a lookup. Not having an equally well-established in-core parallel implementation for Graph Cuts, we use a serial version of our algorithm for comparison. Timings for Graph Cuts are based on the implementation's reported runtime. Due to the parallel option of *Panorama Weaving*, its timings are based on wall-clock time. Datasets which contain more than simple pairwise overlaps were run at full resolution and the running times and energy comparisons are provided in Table 4.1. Our technique produces lower energy seams for all but one example, Fall-5way, and even in this case the techniques have comparable energy. In terms of performance, serial *Panorama Weaving* computes its solution faster than the Graph Cuts for all datasets (at the same resolution). As the Graph Cuts results show, a hierarchical approach would be necessary to achieve similar performance by trading quality for speed. Parallel *Panorama Weaving* further reduces the runtime down to mere seconds for all datasets at full resolution. On average, we see that the scaling performance between *Panorama Weaving's* serial and parallel implementations to be about a five times speedup. This is in sync with the number of physical cores in the test system. Hyperthreading is effective when data access is a main bottleneck. A speedup corresponding to the number

Table 4.1: Performance results comparing *Panorama Weaving* to Graph Cuts for our test datasets that contain more than simple pairwise overlaps. *Panorama Weaving* run serially (PW-S) computes solutions quickly. When run in parallel, runtimes are reduced to just a few seconds. The energy ratio (E. ratio) between the final seam energy produced by *Panorama Weaving* and Graph Cuts (PW Energy / GC Energy) is shown. For all but one dataset (Fall-5way), *Panorama Weaving* produces a lower energy result. It is comparable otherwise. Panorama image sizes are reported in megapixels (MP).

Dataset	Megapixel	Images	PW Parallel	PW Serial	GC Serial	E. Ratio
Crosswalk	16.7	4	1.3	7.2	369.6	0.995
Fall-5way	30.0	5	2.4	12.1	735.4	1.220
Skating	44.7	6	3.2	16.8	734.0	0.851
Lake	9.4	22	0.5	2.9	337.2	0.503
Graffiti	36.6	10	4.3	19.6	983.7	0.707
Nation	49.1	9	4.6	23.2	1168.7	0.800

of physical cores should be expected when an algorithm is compute-bound which is true for *Panorama Weaving*. Therefore our implementation is scaling quite well on our test system.

4.7.1.2 Out-of-core results. To test the performance for our out-of-core implementation, we computed the seams for two large panoramas on our eight core test system. The Fall Salt Lake City panorama consist of 611 overlapping images and is $126,826 \times 29,633$, 3.27 gigapixel, when combined. The final image that is the result of our seam computation is provided in Figure 4.17 (b). Additionally, in Table 4.2 we provide strong scaling test of our implementation for this panorama as we vary the core count from one to eight. As this table illustrates, our implementation shows very good efficiency up to the number of cores of our test system. At eight cores, our efficiency takes a slight dip due to our implementation using a dedicated thread to schedule the face computation for each phase of the out-of-core *Panorama Weaving* technique. On a single core, our system can produce a seam solution in only 68.5 minutes. As I have discussed in Section 2.1 Hierarchical Graph Cuts [2] does not scale beyond two to three levels of the hierarchy. At three levels, the coarse version of this panorama is still approximately 100 megapixel in size with 611 labels and could not be run on our test system. Therefore, to provide context for our running times, we compare our technique to the predicted runtime of a similar technique [94] which relies on a moving window of a Graph Cuts solution. Figure 4.17 (a) provides an example of one of these windows. In our tests, a Graph Cuts solution took 3003.86 seconds to converge. Even more problematic is that the first iteration for this window still took a very long time to compute: 612.99 seconds. Therefore, if this window is a good representation for this

Figure 4.17: Fall Salt Lake City, $126,826 \times 29,633$, 3.27 gigapixel, 611 images. (a) An example window computed with out-of-core Graph Cut technique introduced in Kopf et al. [94]. This single window took 50 minutes for Graph Cuts to converge, with the initial iteration requiring 10.2 minutes. Since the full dataset contains 495 similar windows, using the windowed technique would take days (85.15 hours) at best, and weeks (17.2 days) in the worst case. (b) The full resolution *Panorama Weaving* solution was computed in 68.4 minutes on a single core and 9.5 minutes on eight cores. Our single core implementation required a peak memory footprint of only 290 megabytes while using eight cores had peak memory of only 1.4 gigabytes.

Table 4.2: Strong scaling results for the Fall Salt Lake City panorama, $126,826 \times 29,633$, 3.27 gigapixel, 611 images. Our out-of-core *Panorama Weaving* technique scales very well in terms of efficacy percentage compared to ideal scaling up to the physical cores of our test system (eight cores). At eight cores our technique loses a slight amount of efficiency due to our implementation having a dedicated thread to handing the seam scheduling. Using the full eight cores to process this panorama provides a full resolution seam solution in just 9.5 minutes. The system is extremely light on memory and uses at most 1.4 gigabytes.

Cores	Time(s)	Ideal(s)	Efficiency	Max Mem.
1	4109	NA	NA	290 MB
2	2079	2054.5	98.8%	443 MB
3	1403	1369.7	97.6%	599 MB
4	1049	1027.3	97.9%	791 MB
5	840	821.8	97.8%	881 MB
6	706	684.8	97.0%	1.1 GB
7	601	587.0	97.7%	1.2 GB
8	573	513.6	89.6%	1.4 GB

panorama dataset (which our testing indicates that it is) computing all 495 windows in this dataset would take days (85.15 hours) at best, and weeks (17.2 days) in the worst case. Our implementation can compute a full resolution solution in a little over an hour for a single core and only a few minutes when run on eight cores. Also of note is the small use of memory inherent with our scheme. At any given time, we need only hold the cost of computing the seams for a face per core. Therefore the per-core memory footprint is only the cost of holding two images, an energy buffer, and the seam tress for a given face in memory. Even with floating point precision and eight cores, for this dataset our technique uses at most 1.4 gigabytes of memory. To further test the scalability of our system, our

implementation was run on even larger image. The Lake Louise panorama consists of 1512 images and is $187,069 \times 40.202$ (7.52 gigapixel) when combined. In Table 4.3 we provide a strong scaling test for our implementation for this dataset. Like the previous example, our implementation shows very good efficiency for one to eight cores on our test system. The system is very light on memory resources and needs only 2.0 gigabytes of memory to operate on all eight cores. When using all cores, our implementation can provide a seam solution in only 37.7 minutes. The final image resulting from our seam calculation is provided in Figure 4.18.

4.7.2 Panorama Editing

We provide additional results of the interactive portion of our technique editing a variety of panoramas. Images which are color-corrected were processed using gradient domain blending [133, 103].

4.7.2.1 Editing bad seams. In Figure 4.19, the Nation dataset is a highly dynamic scene of a busy intersection with initial seams that pass through moving cars/people, see Figure 4.19(d). In addition, there are various registration artifacts, see Figure 4.19(e). Before our technique, a user would consider this panorama unsalvageable or be required to manually edit the boundary masks pixel-by-pixel. In just a few minutes, using our system, a user can produce an appealing panorama by adjusting seams to account for the moving objects and pulling registration artifacts into areas which are less noticeable. Figure 4.20 (a, b and c) shows the initial seam configuration for the Skating dataset with two problem areas. The initial seams pass through people who change position on the ice and produce

Table 4.3: Strong scaling results for the Lake Louise panorama, $187,069 \times 40.202$, 7.52 gigapixel, 1512 images. Like the smaller Fall Salt Lake city panorama, our implementation shows very good efficiency up to the physical number of cores on our test system. Using the full eight cores for the full resolution seam solution for this panorama requires 37.7 minutes of compute time and at most 2.0 gigabytes of memory.

Cores	Time(s)	Ideal(s)	Efficiency	Max Mem.
1	16279	NA	NA	382 MB
2	8263	8139.5	98.51%	627 MB
3	5516	5426.3	98.37%	877 MB
4	4132	4069.8	98.49%	1.1 GB
5	3306	3255.8	98.48%	1.4 GB
6	2778	2713.2	97.67%	1.6 GB
7	2383	2325.6	97.59%	1.8 GB
8	2259	2034.9	90.08%	2.0 GB

Figure 4.18: Lake Louise, $187,069 \times 40.202$, 7.52 gigapixel, 1512 images. The *Panorama Weaving* results for the Lake Louise panorama. Our out-of-core seam computation produces this full resolution solution in as little as 37.7 minutes while requiring at most only 2.0 gigabytes of memory. Panorama courtesy of City Escapes Nature Photography

Figure 4.19: *Panorama Weaving* on a challenging data-set (Nation, 12848 x 3821, nine images) with moving objects during acquisition, registration issues and varying exposure. Our initial automatic solution (b) was computed in 4.6 seconds at full resolution for a result with lower seam energy than Graph Cuts. Additionally, we present a system for the interactive user exploration of the seam solution space (c), easily enabling: (d) the resolution of moving objects, (e) the hiding of registration artifacts (split pole) in low contrast areas (scooter) or (f) the fix of semantic notions for which automatic decisions can be unsatisfactory (stoplight colors are inconsistent after the automatic solve). The user editing session took only a few minutes. (a) The final, color-corrected panorama.

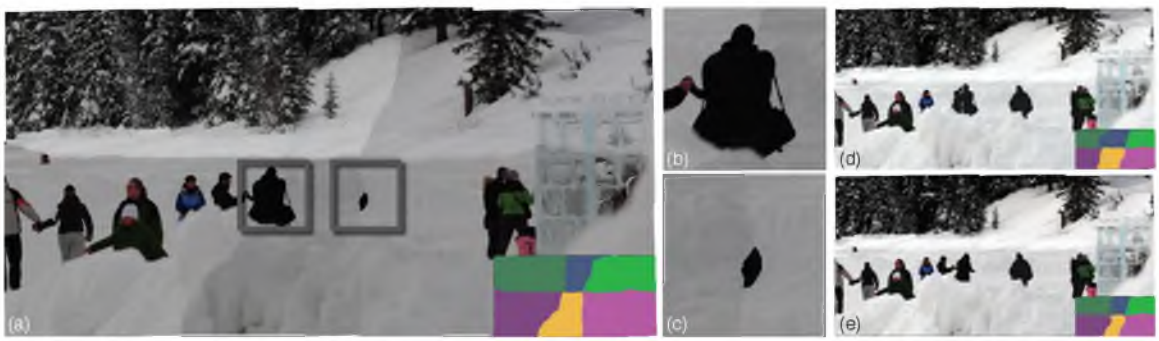


Figure 4.20: Repairing non-ideal seams may give multiple valid seam configurations. (a) The initial seam configuration for the Skating dataset (9400 x 4752, six images) based on gradient energy. (b and c) Its two major problem areas. (d and e) Using our technique a user can repair the panorama, but also has the choices of two valid seam configurations. Panorama courtesy of City Escapes Nature Photography.

either an amalgamation of two positions of a single person or a partial person. As shown in the companion video, repairing these seams only takes a few seconds of interaction, see Figure 4.20 (d and e) for edited results. Figure 4.21 illustrates how a user can correct registration artifacts that appear on the moon's horizon in the Apollo-Armstrong dataset.

4.7.2.2 Multiple valid seams. Along with repairing unideal seams, Figure 4.19 and 4.20 (Nation and Skating) are also examples of a user choosing between multiple valid seam configurations. In Figure 4.19 (f), the initial seam calculation for the Nation dataset produces an intersection with four red stoplights, an inconsistent configuration. With our system, a user can turn two stoplights green creating a more realistic setting. Figure 4.20 (bottom) shows 2 valid seam configurations the that user can choose while fixing the Skating dataset. Each was repaired with a simple bend of the panorama seam. In Figure 4.22, we provide an example of how a user can fix registration artifacts of the dataset (Graffiti) while tuning the seam location for improved results in the final color-correction. For gradient-domain blending, smooth, low-gradient areas provide the best results, therefore the user placed the seams in the smooth wall locations, Figure 4.22 (c). This editing session required just 2 minutes of interaction. Finally, in Figure 4.23 we show the color-corrected edits of the originally optimal, but non-visually pleasing, seams of Figure 1.7 for the two datasets: Canoe and Lake Path. Both interactions required only a few seconds of user input. Figure 4.24 is a Lake vista with multiple dynamic objects moving in the scene during acquisition. In all, there are six independent areas in the panorama where a canoe, or groups of canoes, change positions in overlap areas. Figure 4.24 shows two examples of alternative edits. A user editing with our technique would have the choice of 64 valid seam combinations of canoes. In Figure 4.25, we show a user iterating through valid splitting options of a five valence branching point of the Fall-5way dataset. In this way, we allow users the freedom to add and remove seams as they see fit. Finally, the images Crosswalk and Apollo-Aldrin in Figure 1.9 were created and edited in our system to show how panoramas can have multiple valid seam configurations.

4.8 Limitations and Future Work

Our technique is versatile and can robustly handle a multitude of panorama configurations. However, there is currently a limitation on the configurations which we can handle. The adjacency mesh data structure in its current form relies on the fact that the intersection of pairwise overlaps yields an area of exactly one connected component (which is needed

Figure 4.21: A panorama taken by Neil Armstrong during the Apollo 11 moon landing (Apollo-Armstrong: 6,913 x 1,014, eleven images). (a) Registration artifacts exist on the horizon. (b) Our system can be used to hide these artifacts. (c) The final color-corrected image. Panorama courtesy of NASA.

Figure 4.22: In this example (Graffiti: 10,899 x 3,355, ten images), (a) the user fixed a few recoverable registration artifacts and tuned the seam location for improved gradient-domain processing, yielding a colorful color-corrected graffiti. (b) Our initial automatic solution (energy function based on pixel gradients). (c) The user edited panorama. The editing session took 2 minutes.

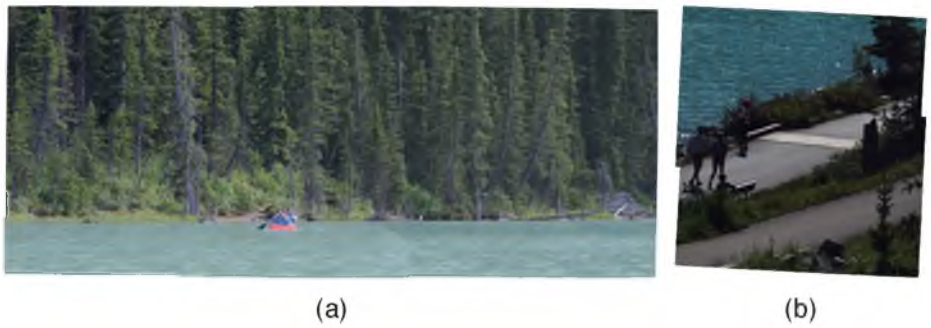


Figure 4.23: The color-corrected, user edited examples from Figure 1.7. The artifacts caused by the optimal seams can be repaired by a user. Images courtesy of City Escapes Nature Photography.

Figure 4.24: A lake vista panorama (Lake: 7,626 x 1,231, 22 images) with canoes which move during acquisition. In all there are six independent areas of movement, therefore there are 64 possible seam configurations of different canoe positions. Here we illustrate two of these configurations with color-corrected versions of the full panorama (a and c) and a zoomed in portion on each panorama (b and d) showing the differing canoe positions. Panorama courtesy of City Escapes Nature Photography.

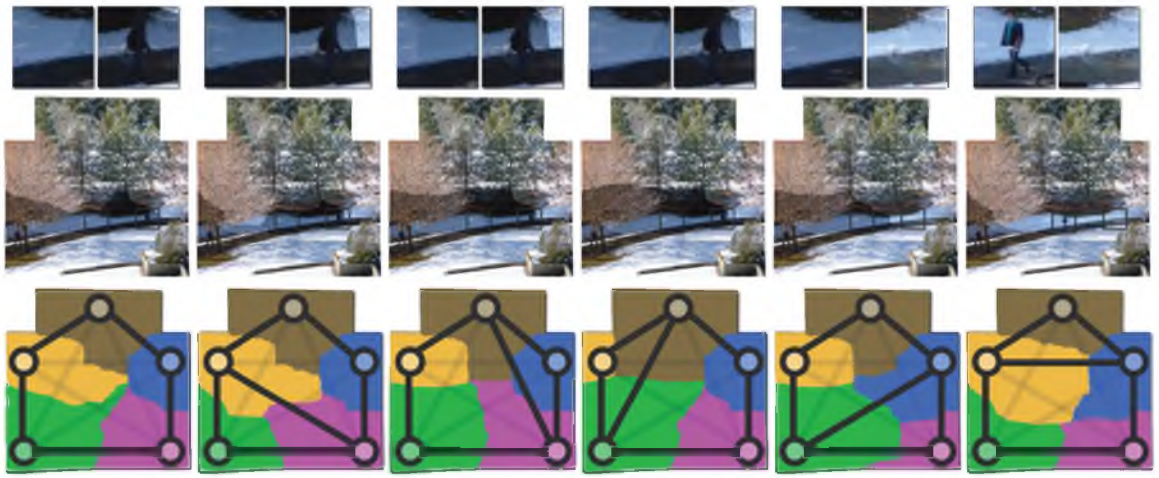


Figure 4.25: Splitting a five valence branching point based on gradient energy of the Fall-5way dataset (5211 x 5177, 5 images): as the user splits the pentagon, the resulting seams mask/unmask the dynamic elements. Note that each branching point that has a valence higher than 3 can be further subdivided.

to guarantee the manifold structure of the mesh). For example, less than one connected component would arise in a situation where one overlap is completely incased inside another and more than one can be caused by an overlap's area passing through the middle of another overlap. Both of these cases break the pairwise seam network assumption. In addition, an image whose boundary is completely enclosed by another image's boundary (100% overlap) is currently considered invalid. These are pathological cases that we have yet to encounter in practice. Overall, the authors feel that these limitations are only temporary and that the data-structures and methods outlined in this work are general enough to support these cases as a future extension.

The serial and parallel out-of-core implementations discussed above show good scale seam processing to images gigapixels in size. The schemes show good scaling for our test datasets even though there is some redundancy in the file I/O. Each image is loaded for ever multioverlap which it is a member. In other words, an image is loaded (reloaded) an equivalent number of times based on the valency of its adjacency mesh node. As future work, my collaborators and I wish to explore caching strategies for images and overlaps, as well as how these strategies interplay with different face traversal order.

CHAPTER 5

INTERACTIVE GRADIENT DOMAIN EDITING AT SCALE

This chapter introduces the *Progressive Poisson* technique which brings the color-correction phase of panorama creation pipeline into an interactive environment. This technique provides gradient domain processing interactively which is the most sophisticated and computationally expensive correction method. This operation is an inherently global and therefore, before this work, there were no known techniques on applying it to massive images interactively. Section 5.2 outlines the interactive technique and a full resolution out-of-core *Progressive Poisson* solver. Section 5.3 extends the full solver to a parallel distributed environment and Section 5.4 shows how it can be redesigned as a cloud based resource.

5.1 Gradient Domain Image Processing

Gradient domain image processing encompasses a family of techniques that manipulate an image based on the value of a gradient field rather than operating directly on the pixel values. Seamless cloning, panorama stitching, and high dynamic range tone mapping are all techniques that belong to this class. Given a gradient field $\vec{G}(x, y)$, defined over a domain $\Omega \subset \Re^2$, we seek to find an image $P(x, y)$ such that its gradient ∇P fits $\vec{G}(x, y)$.

In order to minimize $\|\nabla P - \vec{G}\|$ in a least squares sense, one has to solve the following optimization problem:

$$\min_P \iint_\Omega \|\nabla P - \vec{G}\|^2 \tag{5.1}$$

It is well known that minimizing equation (5.1) is equivalent to solving the Poisson equation $\Delta P = div\ \vec{G}(x, y)$ where Δ denotes the Laplace operator $\Delta P = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2}$ and $div\ \vec{G}(x, y)$ denotes the divergence of $\vec{G}$.

To adapt the equations shown above to discrete images, we apply a standard finite difference approach which approximates the Laplacian as:

$$\Delta P(x, y) \quad = \quad P(x + 1, y) + P(x - 1, y) +$$

$$P(x, y+1) + P(x, y-1) - 4P(x, y) \tag{5.2}$$

and the divergence of $\vec{G}(x, y) = (G^x(x, y), G^y(x, y))$ as:

$$\begin{aligned} div\ \vec{G}(x, y) &= G^x(x, y) - G^x(x-1, y) + \\ & \quad G^y(x, y) - G^y(x, y-1). \end{aligned}$$

The differential form $\Delta P = div\ \vec{G}(x, y)$ can therefore be discretized into the following sparse linear system:

$$L\mathbf{p} = \mathbf{b}. \tag{5.3}$$

Each row of the matrix L stores the weights of the standard five point Laplacian stencil given by (5.2), $\mathbf{p}$ is the vector of pixel colors, and $\mathbf{b}$ encodes the guiding gradient field, as well as the boundary constraints. The choice of guiding gradient field $\vec{G}(x, y)$ and boundary conditions for the system determines which image processing technique is applied. In the case of seamless cloning, it is necessary to use Dirichlet boundary conditions, set to be the color values of the background image at the boundaries, and the guiding gradient to be the gradient of the source image (see [133] for a detailed description). For tone mapping and image stitching, Neumann boundary condition are used. The guiding gradient field for image stitching is the composited gradient field of the original images. The unwanted gradient between images is commonly set to zero or averaged across the stitch. The guiding gradient for tone mapping is adjusted from the original pixel values to compress the high dynamic range (HDR) (see [55] for more detail). Methods such as gradient domain painting [114] allow the guiding gradient to be user defined.

5.2 Progressive Poisson Solver

This section discusses a progressive framework for solving very large Poisson systems in massive image editing. This technique allows for a simple implementation, yet is highly scalable, and performs well even with limited storage and processing resources.

5.2.1 Progressive Framework

For an image P of $n \times n$ pixels, the Laplace system (5.3) has n^2 independent variables, one per pixel. Computing the entire solution is therefore expensive both in terms of space and time. For large images, the space requirements easily exceed the main memory available on most computers. Moreover, the long computation times make any interactive application unfeasible.

Acceleration methods try to address either or both of these issues. The recent adaptive formulation by Agarwala [2] has been particularly insightful. By exploiting the smoothness of the solution, this method was the first to reduce both the cost of the computation and its memory requirements. The approach by Kazhdan and Hoppe [89] demonstrates how a streaming approach can achieve high performance by optimizing the memory access patterns.

We extend these acceleration techniques and show how to achieve high quality local solutions, without the need for solving the entire system. Moreover, we show that coarse approximations are of acceptable visual quality without the cost of a typical coarsening stage used in the V-cycle. These new features, coupled with a simple multiresolution framework, enable a data-driven interactive environment that exploits the fact that interactive editing sessions are always limited by screen resolution. At any given time, a user only sees either a low resolution view of the entire image or a high resolution view of a small area. We take advantage of this practical restriction and solve the Poisson equation only for the visible pixels. This provides performance advantages for interactive sessions, as well as tight control over the memory usage. For example, even the simple step of computing the gradient of the full resolution image can be problematic due to its significant processing time and storage requirement. In our approach, we avoid this problem by estimating gradients on-the-fly using only the available pixels.

Overall, our interactive system is based on a simple two-tier approach:

- A global progressive solver provides a near instant coarse approximation of the full solution. This approximation can be refined up to a desired solution by a lightweight process, often running in the background and possibly out-of-core. Any time the user changes input parameters, this process is restarted.

- A local progressive solver provides a quick solution for the visible pixels. This process is driven by the interactive viewer and uses as a coarse approximation the best solution available from the global solver.

These components can be coupled with different multiresolution hierarchies as discussed in the next section.

5.2.1.1 Initial solution. At launch, the system computes a coarse image for the initial view. A fast two-dimensional direct method using cosine and Fast Fourier transforms by Agrawal [7, 8] is used for this initial solve for techniques that require Neumann boundaries

(stitching, HDR compression). For methods that require Dirichlet boundaries (seamless cloning) we using an iterative method such as SOR. To provide the user with a meaningful preview, we use an initial coarse resolution of one to two megapixel depending on the physical display. We have found, in practice, that the Fast Fourier solver usually gives us this approximation in under 2 seconds. This initial solution is at the core of the progressive refinement defined in the next paragraph.

5.2.1.2 Progressive refinement. The goal of progressive refinement is to increase the resolution of our solution either locally or globally. This requires injecting color transport information from coarser to finer resolutions. In doing so, we exploit the fact that the solution, away from the seams, tends to be smooth [2] and up-sampling the coarse solution gives high quality results in large areas of the image. To improve the solution and resolve the problems at the seams, we run an iterative method, estimating new gradients from the original pixel data of the finer resolution and using the up-sampled values as the initial solution estimate. The finer resolution gradient field allows the iterative solver to reconstruct the detailed features of the original image. For the iterative method we allow the use of either conjugate gradient (for faster convergence) or SOR (for minimal memory overhead). The iterative solver is assumed to have converged when the L_2 norm of the residual between iterations is less than 1.0×10^{-3}. In practice there is no perceptible difference between iterations after this condition is met.

Figure 5.1 shows the refinement process where we assume for simplicity that each resolution doubles each dimension separately and our data is a subsampled hierarchy. In this case, computing each finer resolution is equivalent to adding new rows (or columns) to the coarse resolution. Therefore, we know that each new pixel added has two neighbors from the coarse solution. We can take the average difference from these two neighbors and apply it to the original Rgigabyte value of the pixel from the new resolution (see Figure 5.1 (a)). Since the image is subsampled, the average difference and application to the new pixel is trivial.

In a tiled hierarchy one would need to double both dimensions at the same time, requiring a simple adjustment to the interpolation. Each new resolution is treated as new data and the offset is based on the solution from the previous resolution and the transform between levels.

5.2.1.3 Local preview. Comegabyteining the coarse, global solve with a progressively refined local preview is all that is necessary for our interactive system. For data

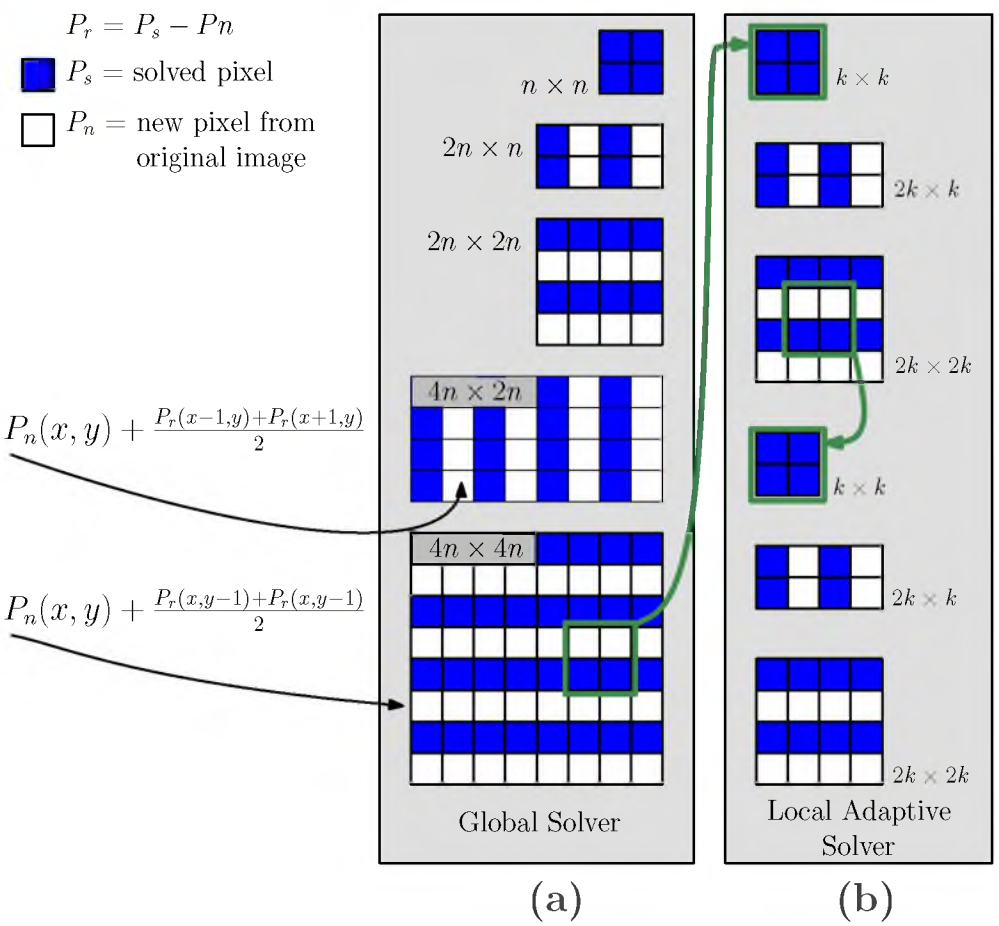


Figure 5.1: Our adaptive refinement scheme using simple difference averaging. (a) Global progressive up-sampling of the edited image computed by a background process. (b) View-dependent local refinement based on a $2k \times 2k$ window. In both cases we speedup the SOR solver with an initial solution obtained by smooth refinement of the solution.

requests at resolutions equal to or less than our coarse solution, we simply display the available data. As the user zooms into an area, the image is progressively refined in a local region. Since the resolution increase is directly coupled with the decrease in the extent of the local view, the numegabyteer of pixels that must be processed remains constant (see Figure 5.1 (b)). This results in a logarithmic run-time complexity and constant storage requirement, which allows our system to gracefully scale to images orders of magnitude larger than previously possible.

5.2.1.4 Progressive full solution. The progressive refinement can be applied globally to compute a full solution. Since the method requires a very small overhead, it can easily be run as background process during the interactive preview. When a new resolution has been solved, the interactive preview uses the solution as a new coarse approximation,

thereby saving computation during the local adaptive phase. Like other in-core methods, this progressive global solver is limited by available system memory. To address this issue, the global solver has the ability to switch modes to a moving-window out-of-core progressive solver.

5.2.1.5 Out-of-core solver. The out-of-core solver maintains strict control over memory usage by sweeping the data with a sliding window. The window traverses the finest desired resolution, which can be several levels in the hierarchy from the current available solution. If the jump in resolution is too high, the process can be repeated several times. Within each window, the coarse solution is up-sampled and the new resolution image is solved using the gradients from the desired resolution. Since the window lives at the desired resolution, we never need to expand memory beyond the size of the window. Furthermore, windows are arranged such that they overlap with the previously solved data in both dimensions to produce a smooth solution. The overlap causes the solver to load and compute some of the data multiple times. This overlap has an inherent overhead when compared to an idealized in-core solver. For instance, given a $1/x$ overlap, the four corners, each $1/x \times 1/x$ in size, are executed four times. The four strips on the edge of the window, not including the corners, each $1/x \times (1 - 2/x)$ in size are executed two times. All other pixels, size $(1 - 2/x) \times (1 - 2/x)$, are executed once. Therefore, the overhead computation for this $1/x$ overlap is given by: $4/x(1 + 1/x)$. Moreover, the I/O overhead can be reduced to $1/x$, since we can retain pixel data from the previous window in the primary traversal direction. In principle, a larger overlap between windows results in higher quality solutions, though in practice we have found that for a 1024×1024 window a $1/32$ overlap is sufficient for good results. This overlap requires only a 12.8% compute overhead and a 3.1% I/O overhead. A larger window can be used to reduce the percentage overlap while achieving the same quality results. For instance, by doubling the window size in both dimensions, a 2048×2048 window can be computed with a $1/64$ overlay, only incurring a 6.3% compute overhead and a 1.5% I/O overhead. Compared to the exact analytical solution, our method produces even higher quality results than the best known method [89] for equivalent run times.

5.2.2 Data Access

Our progressive solver can operate well on multiple hierarchical schemes. Tiled hierarchies are often used to produce smoother, antialiased images, though high contrast areas

in the original image may be lost in the smoothing. As Figure 5.2 (b) shows, the tiled image is visually pleasing, but details such as the cars on the highway are lost. This visual smoothness can also come at the cost of significant preprocessing, reduced flexibility when dealing with missing data, and increased I/O when traversing the data. The costs can be especially significant for massive data if one has to process it with very limited resources. The least costly image hierarchy can be computed by subsampling. Subsampling is simple and lightweight, but is prone to high frequency aliasing. It does, though, retain higher contrast at the coarse resolution. Figure 5.2 (a) shows how the subsampled hierarchy has aliasing artifacts, but also retains enough contrast to see the cars on the highway. This contrast may be beneficial for some applications, such as an analyst studying satellite imagery.

To show the flexibility of our interactive system, we support both a filtered tiled hierarchy and a subsampled hierarchy (see Figure 5.3). For a tiled scheme, we compute the image

Figure 5.2: Subsampled and tiled hierarchies. (a) A subsampled hierarchy. As expected, subsampling has the tendency to produce high-frequency aliasing. Though details such as the cars on the highway and in the parking lots are preserved. (b) A tiled hierarchy. This produces a more visually pleasing image at all resolutions but at the cost of potentially losing information. The cars are now completely smoothed away. Data courtesy of the U.S. Geological Survey.

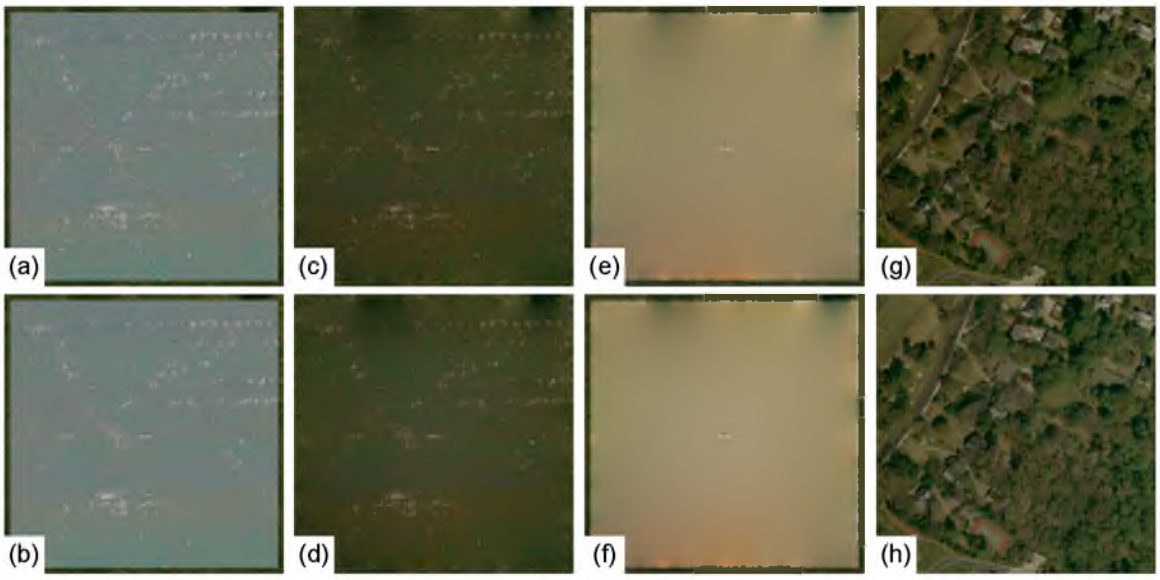


Figure 5.3: Our progressive framework using subsampled and tiled hierarchies. (a) A composite satellite image of Atlanta, over 100 gigapixels at full resolution, overlaid on Blue Marble background subsampled; (b) a tiled version of the same satellite image; (c) the seamless cloning solution using subsampling; (d) the same solution computed using a tiled hierarchy; (e) the solution offset computed using subsampling; (f) the solution computed using tiles; (g) a full resolution portion computed using subsampling; (h) the same portion using tiling. Note that even though there is a slight difference in the computed solution, both the tiled and the subsampled hierarchies produce a seamless stitch with our framework. Data courtesy of the U.S. Geological Survey and NASA's Earth Observatory.

hierarchy using a Gaussian kernel to produce a smooth, antialiased image (Figure 5.3 right column). With a minor variation to the underlying I/O layer, our system also supports a faster, subsampled Hierarchical Z-order as proposed by Pascucci and Frank [2002] (Figure 5.3 left column). For an overview of the HZ data format, see Section 3.1. To achieve the level of scalability necessary in the current system, we further simplify the HZ data access scheme. We use a lightweight recursive algorithm that avoids repeated index computations, provides progressive and adaptive access, guarantees cache coherency and minimizes the numegabyteer of I/O operations without using any explicit caching mechanisms. In particular, computing the HZ index with this new algorithm attains a thirty times speedup compared to the previous work. For example, to compute the indices for a 0.8 gigapixel image the new algorithm requires 4.7 seconds where the previous method would take 144.1 seconds. Moreover, since the traversal follows the HZ storage order exactly for any query window, we guarantee that each file is accessed only once without need of holding any block of data in cache. For details on our new recursive algorithm see Section 3.2. This

approach makes the system intrinsically cache friendly for any realistic caching architecture and, therefore, very flexible in exploiting modern hardware. Conversion into HZ-order requires no additional storage. On the other hand, for tiled hierarchies a 1/3 data increase is common. Due to our new data access scheme, conversion to HZ-order is straightforward and inexpensive. For our test data, we have found that there is only a 27% overhead due to the conversion compared to just copying the raw data. In essence, the conversion is strictly a reordering of the data and requires no operations on the pixel data. This conversion will outperform even the most simple tiled hierarchies which require some manipulation of the pixel data.

Each resolution in the HZ-hierarchy is in plain Z-order, which allows for fast, cache coherent access of subregions of the image. HZ is not tied to a specific data traversal order, such as the row-major imposed by traditional file formats, as previously observed in [89]. In fact, HZ maintains a high degree of cache coherency even during adaptive local traversals. The locality of our data access provides graceful performance degradation even in extreme conditions. In particular, we demonstrate accessing a data set, of roughly a terabyte in size, by simply mounting a remote file system over an encrypted VPN channel via a wireless connection. Even in normal running conditions, we have found that the I/O overhead caused by using a tiled hierarchy increased the running time by 39%-67%. These numegabyteers reflect the theoretical bound of 1/3 overhead, made worse by the inability to constrain real queries to perfect alignment with the boundaries of a quadtree. The effect of this overhead is detrimental to the scalability of the system under more difficult running conditions such as the one mentioned above. Moreover, HZ easily handles partially converted data, as we show in one portion of the accompanying video for the editing of the Salt Lake City panorama. In a tiled scheme, the entire hierarchy may need to be recomputed as new data is added.

5.2.3 Interactive Preview and Out-of-Core Solver Results

We demonstrate the scalability and interactivity of our approach on several applications, using a numegabyteer of images ranging from megapixels to hundreds of gigapixels in size. To further illustrate the responsiveness of our system, the accompanying video shows screen captures of live demonstrations. To highlight particular details and validate the approach, the figures in this section show previews and close-ups of our interactive system, alongside the results of our full out-of-core progressive solver. We also provide running times of our

full out-of-core solver compared with the best current method, streaming multigrid [89], which we have verified to use the same gradient information. All timings and demos were performed on a 64-bit 2.67 GHz Intel Quad Core desktop, with 8 gigabyte of memory. All streaming multigrid timings were computed from code provided by the authors and include the timing for the gradient preprocess along with the timing to produce a solution.

Our simple framework provides the illusion of a fully solved Poisson system at interactive frame rates and under continuous parameter changes with only a simple GL texture for display and no special hardware acceleration. Therefore, our code is platform independent. Our simple progressive out-of-core solver produces robust solutions with run times that rival [89]. Unlike the previous method, our out-of-core solver does not use hardware acceleration and did not undergo high code optimization to achieve the following runtimes. The solver is also sequential and uses no threading to accelerate the computation. If further optimization of the run-times is desired, there is nothing in our system to prevent the addition of these acceleration techniques.

Different from other out-of-core methods, we do not rely on large external memory data structures and we do not need to pre-compute gradients for the entire image. For the Salt Lake City panorama, for example, the streaming multigrid method [89] creates 75.2 gigabyte of auxiliary information for a 7.9 gigabyte input image. While disk space is generally assumed to be plentiful, such an explosion in disk space is unsustainable for images hundreds of gigapixel in size. The collection of satellite imagery we use in our video is more than one terabyte in size and would, therefore, require more than 9.5 terabytes of temporary storage.

The Edinburgh example is 25 images has resolution $16,950 \times 2,956$ (50 megapixel). At launch, our system performs a seamless stitch Poisson solve of a global 0.7 megapixel image in 1.26 seconds using our direct analytical solve, see Figure 5.4 (a). From this point on, the system can pan and zoom interactively as if the full-solution were already available. Our local adaptive refinement gives a solution that is visually equivalent to a solution to the entire system, see Figure 5.4 (c, d, and e). In the accompanying video, we demonstrate interactive editing and solving of the Poisson system, after the repeated user-selected replacement of pixels of a particular color. We also perform a seamless clone of a 2000×1600 airplane on Edinburgh's cloudy sky. The plane is animated along a linear path across the panorama. As evident in the video, our framework shows the entire sequence in real-time.We also demonstrate similar interactive editing with the Redrock panorama (data

Figure 5.4: The Edinburgh Panorama $16,950 \times 2,956$ pixels. (a) Our coarse solution computed at a resolution of 0.7 megapixels; (b) the same panorama solved at full resolution with our progressive global solver scaled to approximately 12 megapixel for publication; (c) a detail view of a particularly bad seam from the original panorama; (d) the problem area previewed using our adaptive local refinement; (e) the problem area solved at full resolution using our global solver in 3.48 minutes.

courtesy of Aseem Agarwala): nine images, $19,588 \times 4,457$; 87 megapixels. Given this initial coarse solution, our method can produce a full solution of Edinburg, see Figure 5.4 (b), in 3.48 minutes. The streaming multigrid method requires 3.52 minutes. Figure 5.5 (a) shows the convergence and error for our method and streaming multigrid when compared to the ideal direct solution.

The Salt Lake City example is 611 images with resolution of $126,826 \times 29,633$ (3.27 gigapixel). A significantly larger example is provided by a panorama captured with a simple camera mounted on a GigaPan robot [63]. To maximize individual image quality the pictures were taken with automatic exposure times, which inherently increases the color differences between images that need to be corrected by the Poisson solver. An initial coarse preview of 0.87 megapixel is computed by our direct analytical solver in 2.07 seconds. Figure 5.6 shows the original set of images (a), the panorama that our systems stitches in real time (b), the global solution provided by our out-of-core solver (c), and the difference image between the interactive preview and the final solution at the coarse resolution (d). There are slight deviations at some of the more challenging seams, but overall there is negligible visible difference. Our local adaptive preview mimics well the global solution, as shown in Figure 5.7. To test the accuracy of the methods, we have run a full analytical Poisson solver on a 485 megapixel subset of the panorama on a HPC-computer. Figures 5.8 (a) and (b) show how close our out-of-core solution comes to the exact analytical solution. Figure 5.8 (c) shows that the multigrid method has yet to converge to an acceptable solution given an equivalent amount of running time. All solutions were computed using the map

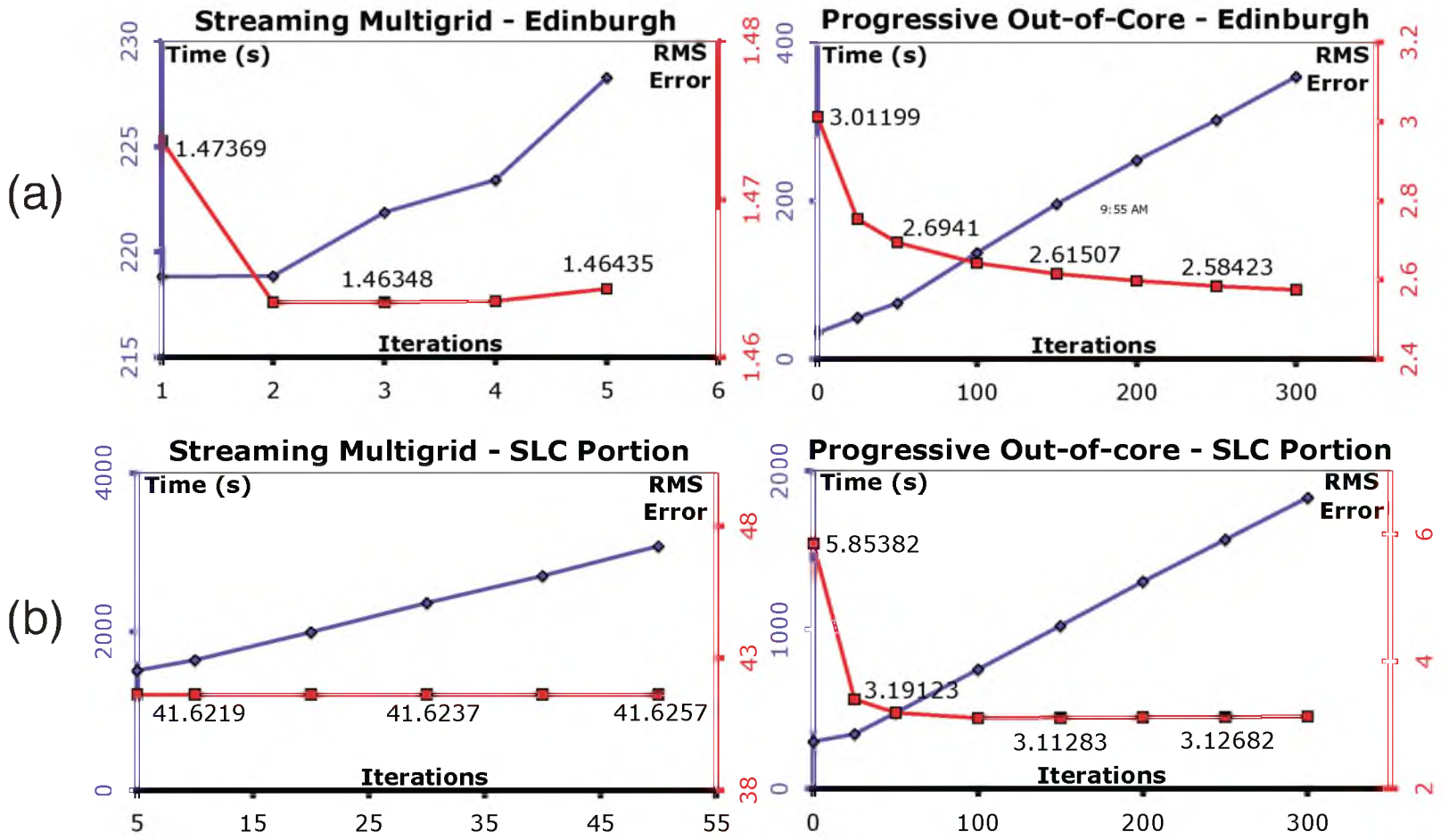


Figure 5.5: The RMS error when compared to the ideal analytical solution as we increase iterations for both methods. Streaming multigrid has better convergence and less error for the Edinburgh example (a), though our method remains stable for the larger Salt Lake City panorama (b). Notice that every plot has been scaled independently to best illustrate the convergency trends of each method.

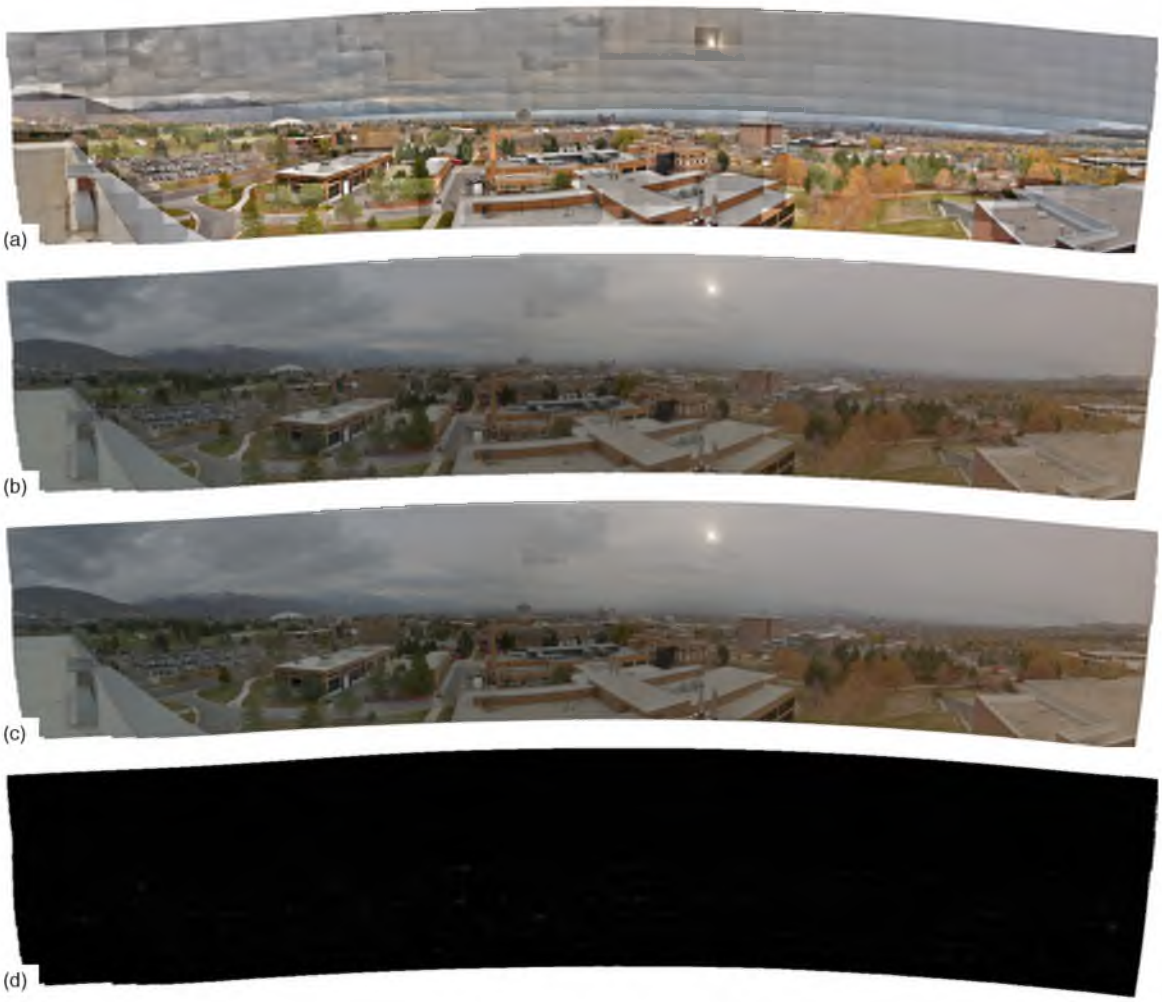


Figure 5.6: Panorama of Salt Lake City of 3.27 gigapixel, obtained by stitching 611 images. (a) Mosaic of the original images. (b) Our solution computed at 0.9 megapixel resolution. (c) The full solution provided by our global solver. (d) The difference image between our preview and the full solution at the preview resolution. Both (a) and (c) have been scaled for publication to approximately 12.9 megapixels.

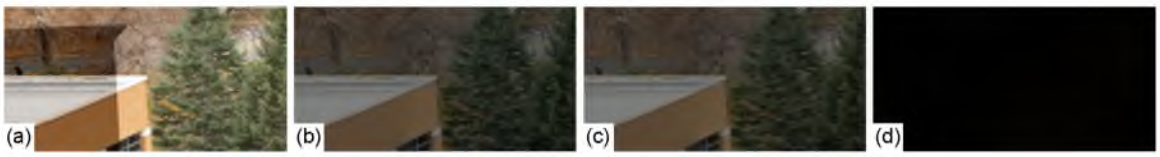


Figure 5.7: A comparison of our adaptive local preview on a portion of the Salt Lake City panorama one half of the full resolution; (a) the original mosaic, (b) our adaptive preview, (c) the full solution from our global solver, and (d) the difference image between the adaptive preview and the full solution

Figure 5.8: A comparison of our system with the best known out of core method [Kazhdan and Hoppe 2008] and a full analytical solution on a portion of the Salt Lake City panorama, 21201 × 24001 pixels, 485 megapixel (a) the full analytical solution; (b) our solution computed in 28.1 minutes; (c) solution from [Kazhdan and Hoppe 2008] computed in 24.9 minutes; (d) the analytical solution where the solver is allowed to harmonically fill the boundary; (e) our solution with harmonic fill; (f) solution from [Kazhdan and Hoppe 2008] with harmonic fill; (g) the map image used by all solvers to construct the panorama where the red color indicates the image that provides the pixel color and white denotes the panorama boundary.

increases the memory usage of the method.

Sierpinki Sponge example has resolution $128k \times 128k$ (16 gigapixel). We have tested the tone mapping application on a synthetic high dynamic range image generated with MegaPOV [118]. In this image we use a partially refined model of a Sierpinki Sponge to create high variations in level-of-detail. Such details can be completely hidden in the dark areas under projected shadows. We follow the approach introduced by Fattal [55] to reconstruct the information hidden in the dark regions. To validate the approach, we ran a typical HDR test image, the Belgium House, progressively refined from a 16 x 12 coarse solution. Even with such a coarse initial solution, we achieve results very close to the exact solution (see Figure 5.9 (c) and (d)). Figure 5.9 shows the original sponge model (a) and

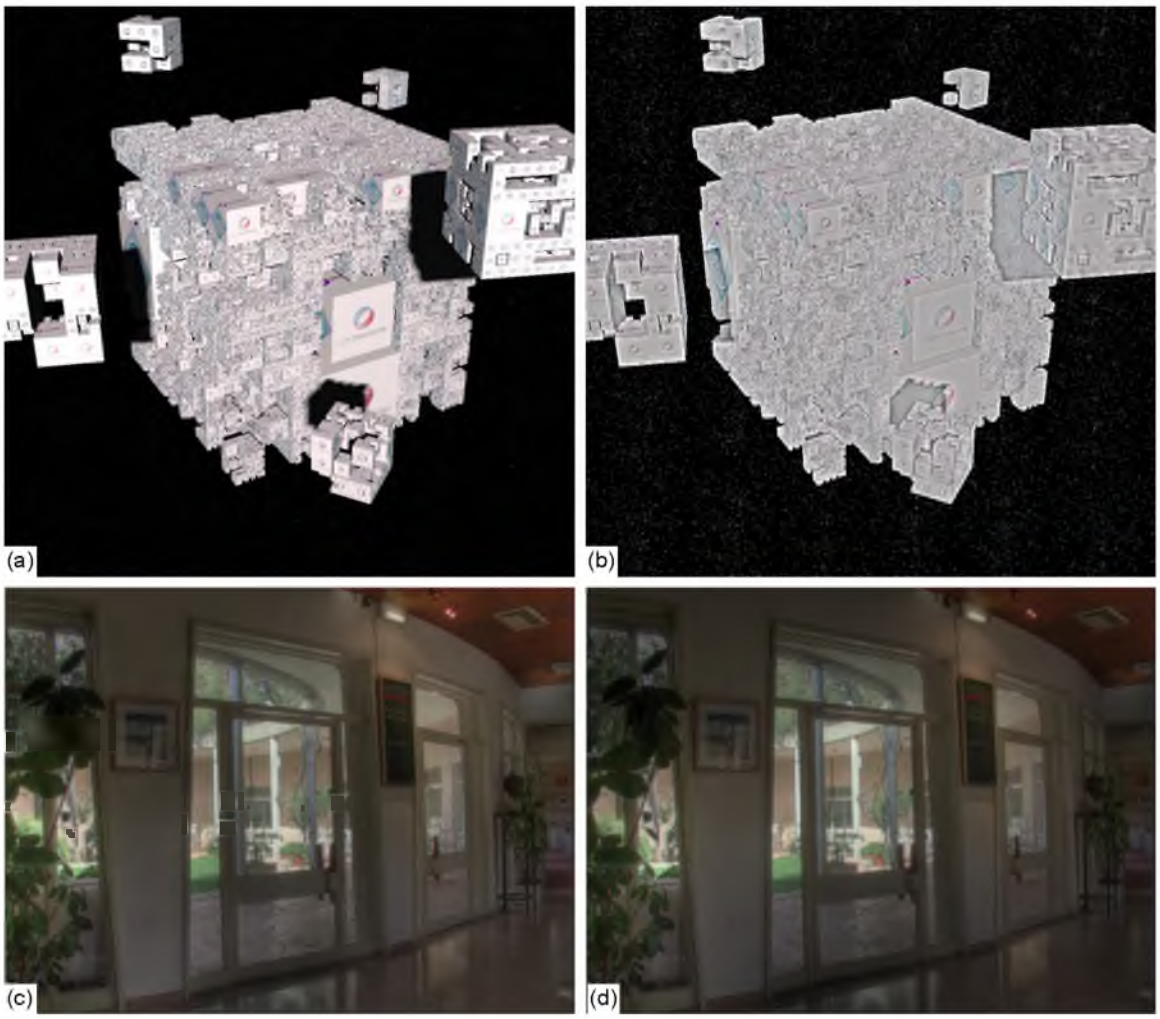


Figure 5.9: Application of our method to HDR image compression: (a) Original synthetic HDR image of an adaptively refined Sierpinki sponge generated with Povray. (b) Tone mapped image with recovery of detailed information previously hidden in the shadows. (c) Belgium House image solved using our coarse-to-fine method with an initial 16 x 12 coarse solution ($\alpha = 0.01$, $\beta = 0.7$, compression coefficient=0.5). (d) The direct analytical solution. Image courtesy of Raanan Fattal.

the processed version (b), where all the details under the shadows have been recovered.

The satellite example contains a Blue Marble background image that is 3.7 gigapixel and imagery of Atlanta and other cities which are well over 100 gigapixel. To demonstrate the scalability of our system, we have run the seamless cloning algorithm for entire cities over a variety of realistic backgrounds from NASA's Blue Marble Collection [125] (see Figure 5.10). We show how a user can take advantage of these capabilities to achieve artistic effects and create virtual worlds from real data. We also create a dynamic environment by animating

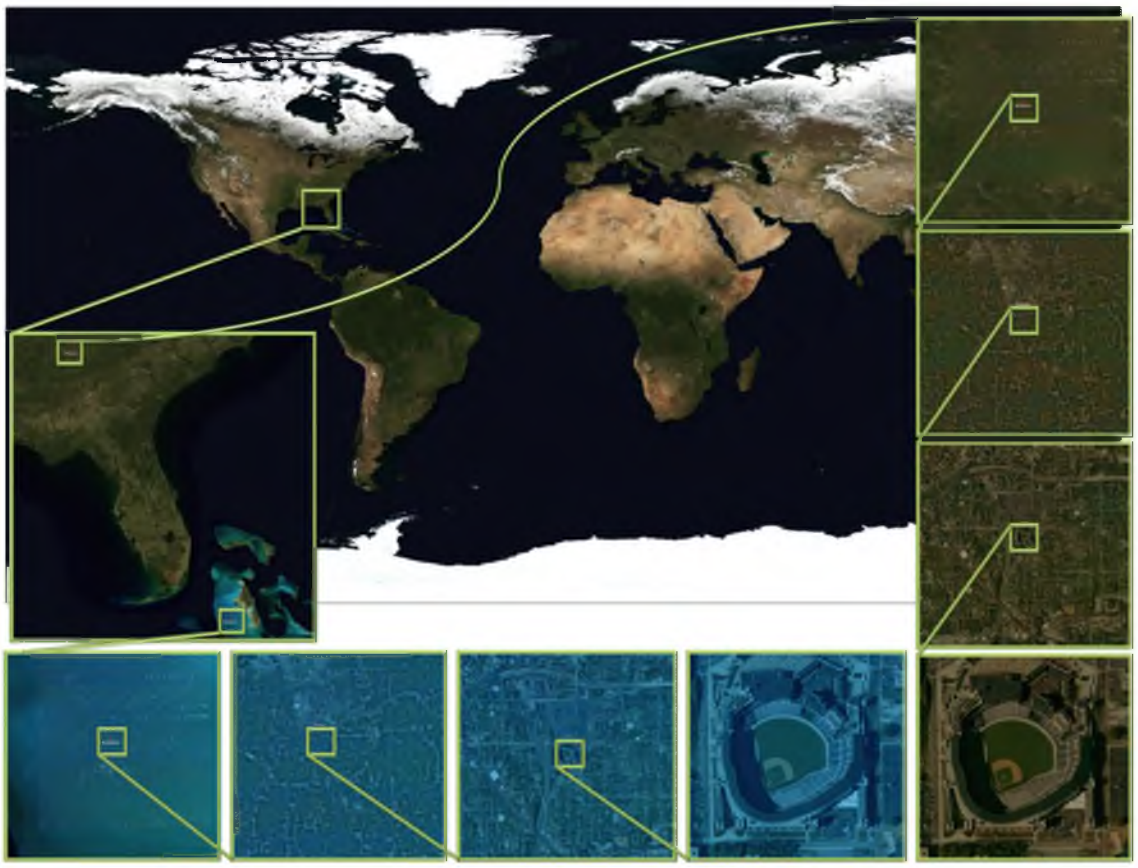


Figure 5.10: Satellite imagery collection with a background given by a 3.7 gigapixel image from NASA's Blue Marble Collection. The *Progressive Poisson* solver allows the application of the seamless cloning method to two copies of the city of Atlanta, each of 116 gigapixels. An artist can interactively place a copy of Atlanta under shallow water and recreate the lost city of Atlantis. Data courtesy of the U.S. Geological Survey and NASA's Earth Observatory.

the background world map over 12 months and concurrently use the Poisson solver to show how the appearance of a city would change across the seasons.

5.3 Parallel Distributed Gradient Domain Editing

In the following, we provide details of our parallel *Progressive Poisson* algorithm and MPI implementation. This new algorithm reduces the time to compute a full resolution gradient domain solution from hours in the case of a single, out-of-core solution to minutes when run on a distributed cluster.

5.3.1 Parallel Solver

Commonly, large images are stored as tiles, which gives one an underlying structure to divide an image amongst the nodes/processors for a distributed solver. Tile-based distributed solvers have been shown to work well when only local trends are present. Seamless stitching commonly contains large scale trends where a naive tile-based approach will provide poor results. The addition of the *Progressive Poisson* method's coarse upsampling, allows for a simple, tile-based parallel solver that can account for large trends. Our algorithm works in two phases: The first phase performs the progressive upsample of a precomputed coarse solution for each tile. The second phase solves for a smooth image on tiles that significantly overlap the solution tiles from the first phase. In this way, the second phase smooths any seams not captured or even introduced by the first phase, producing a complete, seamless image.

5.3.1.1 Data distribution as tiles. Although a tile-based approach leverages a common image storage format, it is not typically how methods are designed to handle seamless stitching of large panoramas. For instance, methods like streaming multigrid [89, 90] often assume precomputed gradients for the whole image. Our system is designed to take tiles directly as input and therefore must be able to handle the gradient computation on-the-fly. An important and often undocumented component of panorama stitching is the map or label image. Given an ordered set of images which compose the panorama, the map image gives the correspondence of a pixel location in the overall panorama to the smaller image that supplies the color. This map file is necessary to determine the difference between actual gradients and those due to seams. This map also defines the boundaries of the panorama, which are commonly irregular. This file along with each individual image that composes the mosaic are needed for a traditional, out-of-core scheme [89, 149] for gradient computation. If the gradient across the seams is assumed to be zero, which is a common technique we adopt for this solver, each tile can be composited in advance and the map file is only needed to denote image seams or boundary. As noted above, this composited tile is often already provided if used in a traditional large image system. The map file can then be encoded as an extra channel of color information, typically the alpha channel. For mosaics of many hundreds of images, such as the examples provided in this dissertation, we cannot encode an index for each image in a byte of data. Though in practice each tile has very little probability of having more than 256 individual images, each image is given a unique 0-255 number on a per tile basis.

We have chosen an overlap of 50% in both dimensions for the second phase windowing scheme of the parallel solver for simplicity in implementation. Each *window* is composed of a 2×2 collections of tiles. To avoid undefined windows in the second phase, we add a symbolic padding of one row/column of tiles to all sides of the image which the solver regards as pure boundary. Figure 5.11 gives an example of a tile layout. The overlapping window size used for our testing was 1024×1024 pixels (assuming 512×512 tiles), which we found to be a good compromise between a low memory footprint and image coverage. Each node receives a partition of windows equivalent to a contiguous subimage with no overlap necessary between nodes during the same phase. Data can be distributed evenly across all nodes in the case of a homogeneous distributed system or dependent on weights due to available resources in the case of a heterogeneous hardware. We provide a test case for a heterogeneous system in Section 5.

5.3.1.2 Coarse solution. As a first step, the first phase of our solver will upsample via bilinear interpolation a 1-2 megapixel coarse solution. Much like the *Progressive Poisson* method [149], each node computes a solution in just a few seconds using a direct FFT solver on a coarsely sampled version of our large image. In tiled hierarchies, this coarse image is typically already present and can be encoded with the map information in much the same way as the tiles.

5.3.1.3 First phase: progressive solution. This phase computes a *Progressive Poisson* solution for each window which are composed of tiles read off of a distributed

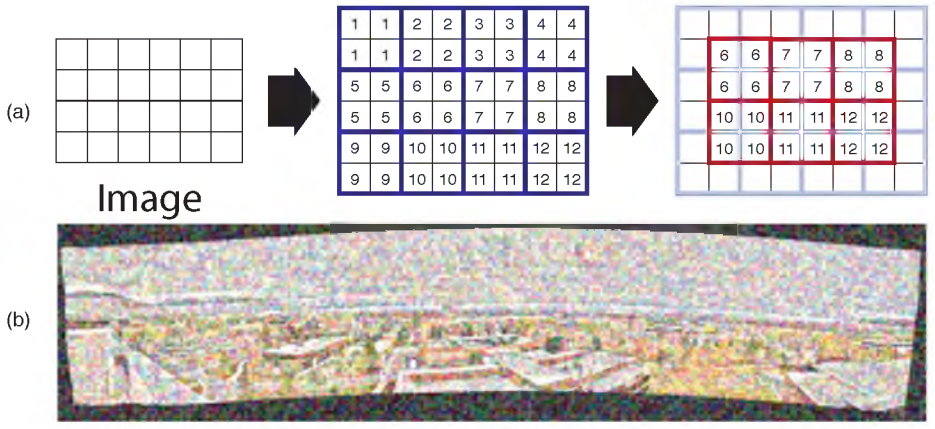


Figure 5.11: Our tile-based approach: (a) An input image is divided into equally spaced tiles. In the first phase, after a symbolic padding by a column and row in all dimensions, a solver is run on a window denoted by a collection of four labeled tiles. Data are sent and collected for the next phase to create new data windows with a 50% overlap. (b) An example tile layout for the Fall Panorama example.

file system. To progressively solve a window, an image hierarchy is necessary. For our implementation a standard power-of-two image pyramid was used. As a first step, the solver upsamples the solution to a finer resolution in the image pyramid using a coarse solution image and the original pixel values. An iterative solver is then run for several iterations to smooth this upsample using the original pixel gradients as the guiding field. This process is repeating down the image hierarchy until the full resolution is reached. The solver is considered to have converged at this resolution when the L_2 norm falls below 10^{-3} which is based on the range of byte color data. From our testing, we have found that SOR gives both good running times and low memory consumption and therefore is our default solver. As noted above, this window is logically composed of four tiles, which are computed and saved in memory for the next phase as floating point color data. This leads to 12 bytes/pixel (three floating point color data) to transfer between phases. Given the data distribution, one node may process many windows. If this is the case, only the tiles which border a node's domain are prepared to be transferred to another node, thereby keeping data communication between phases to a relatively small zone.

5.3.1.4 Second phase: overlap solution. The second phase gathers the four tiles (both solution and original) that make up the overlapping window. After the data are gathered, the gradients are computed from the original pixel values and an iterative solver (SOR) is run after being initialized with the solutions from the first phase. The iterative solver is constrained to only work on interior pixels to prevent this phase from introducing new seams at the window boundary. Technically, there may be errors at the pixels around the midpoints of the boundary edges of these windows, though we have not encountered this in practice. Again, this solver is run until convergence given by the L_2 norm. Note that even though the tile gradients are computed in the first phase, we have chosen to recompute them on the fly in the second phase. Passing the gradients would cost at least an additional 12 bytes/pixel overhead. As nodes increase, data transfer and communication becomes a significant bottleneck in most distributed schemes therefore, we chose to pay the cost of increased computation and reading the less expensive byte image data from the distributed file system instead of the costly transfer.

5.3.1.5 Parallel implementation details. Each node has one *master* thread which coordinates all processing and communication. The core component of this thread is a priority queue of windows and tiles to be processed. At launch, this queue is initialized by a separate *seeding* thread with the initial domain of windows to be solved in the first

phase. Because of the separation of the main thread from the seeding of the queue, the main thread can begin processing windows immediately. Each window is given a first phase id, which is the window's row and column location in the subimage to be processed by a node. Communication between nodes need only be one-way in our system, therefore we have chosen for communication to be "upstream" between nodes, i.e., the nodes operating on a subimage with horizontal or vertical location greater than the current node. In order to avoid starvation in the second phase, the queue is loaded with windows in reverse order in terms of the tile id. Figure 5.12 gives an example of the traversal and communication. All initially seeded windows are given equal low priority in the queue. In essence the initial queue operates much like a first-in-first-out (FIFO) queue. As windows are removed from the queue, the main thread launches a progressive solver thread which is handed off to an intra-node dynamic scheduler. Our implementation uses a HyperFlow [170] scheduler to execute the solver on all available cores. HyperFlow has been shown to efficiently schedule execution

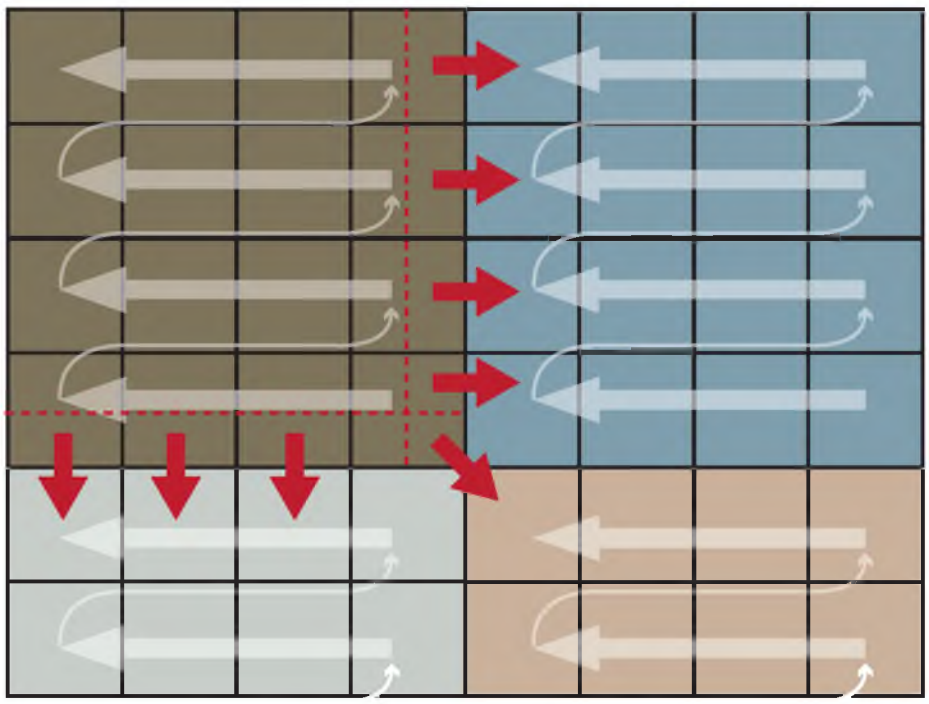


Figure 5.12: Windows are distributed as evenly as possible across all nodes in the distributed system. Windows assigned to a specific node are denoted by color above. Given the overlap scheme, data transfer only needs to occur one-way, denoted by the red arrows and boundary above. To avoid starvation between phases and to hide as much data transfer as possible, windows are processed in inverse order (white arrows) and the tiles needed by other nodes are transferred immediately.

of workflows on multicore systems and therefore is the perfect solution for our intra-node scheduling. In all there are two distinct sequential stages in each phase: (1) loading of the tile data and the computation of the image gradient and (2) the progressive solution. This flow information allows HyperFlow to exploit data, task, and pipeline parallelism to maximize throughput.

After a solution is computed, the progressive solver thread partitions the window into the tiles that comprise it. This allows the second phase to recombine the tiles needed for the 50% overlap window. All four tiles are loaded into the queue with high priority. If the main thread removes a tile (as opposed to a window) from the queue and the tile is needed by another node, the main thread immediately sends the data asynchronously to the proper node. Otherwise, if the node needs this tile for phase two, the second phase id of the window which needs the tile is computed and hashed with a two-dimensional hash function the same size as the window domain for the second phase. If all four tiles for a given second phase window have been hashed, the main thread now knows a second phase window is ready and immediately passes the window to a solver thread for processing. If the main thread receives a solved tile from another node, this is also immediately hashed.

5.3.2 Results

To demonstrate the scalability and adaptability of the approach, we have tested our implementation using two panorama datasets, gigapixels in size. To illustrate the portability of the system, we have also shown its running times and scalability on two distributed systems. Our main system, the NVIDIA Center of Excellence cluster in the Scientific Computing and Imaging Institute at the University of Utah, consists of 60 active nodes with 2.67GHz Xeon X5550 Processors (8 cores), 24GB of RAM per node, and 750GB local scratch disk space. The second system, the Longhorn visualization cluster in the Texas Advanced Computer Center at the University of Texas at Austin, consists of 256 nodes (of which 128 were available for our tests) with 2.5GHz Nehalem Processors (8 cores), 48GB of RAM per node, and 73GB local scratch disk space. Weak and strong scalability tests were performed on both systems. Given the proven scalability of Hyperflow on one node, we have tested the scalability of the MPI implementation from 2-60 and 2-128 nodes for the NVIDIA cluster and Longhorn cluster, respectively. Timings are taken as best over several runs to discount external effects to the cluster from shared resources such as the distributed file system. The datasets used for testing were:

- Fall Panorama. $126,826 \times 29,633$, 3.27 gigapixel. When tiled, this dataset is composed of 124×29 1024^2 sized windows. See Figure 5.13 for image results from a NVIDIA cluster 480 core test run.
- Winter Panorama. $92,570 \times 28,600$, 2.65 gigapixel. When tiled, this dataset is composed of 91×28 1024^2 sized windows. See Figure 5.14 for image results from a NVIDIA cluster 480 core test run.

5.3.2.1 NVIDIA cluster. To show the MPI scalability of our framework and implementation, strong and weak scaling tests were performed for 2-60 nodes. As shown in Tables 5.1 and 5.2, both datasets scale close to ideal and with high efficiency for strong

Figure 5.13: Fall Panorama - $126,826 \times 29,633$, 3.27 gigapixel. (a) The panorama before seamless blending and (b) the result of the parallel Poisson solver run on 480 cores with 124×29 windows and computed in 5.88 minutes.

Figure 5.14: Winter Panorama - $92,570 \times 28,600$, 2.65 gigapixel. (a) The result of the parallel Poisson solver run on 480 cores with 91×28 windows and computed in 6.02 minutes, (b) the panorama before seamless blending, and (c) the coarse panorama solution.

Table 5.1: The strong scaling results for the Fall Panorama run on the NVIDIA cluster from 2-60 nodes up to a total of 480 cores. Overhead (O/H) due to MPI communication and I/O is also provided along with its percentage of actual running time. The Fall Panorama, due to its larger size begins to lose efficiency at around 32 nodes when I/O overhead begins to dominate. Even with this overhead, the efficiency (Eff.) remains acceptable.

Strong Scaling - Fall Panorama - NVIDIA cluster						
Nodes	Cores	Ideal (m)	Actual (m)	Eff. %	O/H (m)	% O/H
2	16	79.35	79.35	100.0	18.80	23.7
4	32	39.68	40.08	97.1	9.05	22.2
8	64	19.84	20.83	95.2	7.28	35.0
16	128	9.92	11.43	78.9	6.50	51.7
32	256	4.96	6.20	53.8	6.20	67.3
48	384	3.31	6.40	51.7	6.40	100.0
60	480	2.65	5.88	45.0	5.88	100.0

Table 5.2: The strong scaling results for the Winter Panorama run on the NVIDIA cluster from 2-60 nodes up to a total of 480 cores. Overhead (O/H) due to MPI communication and I/O is also provided along with its percentage of actual running time. For the Winter Panorama, the I/O overhead does not effect performance up to 60 nodes and the implementation maintains efficiency (Eff.) throughout all of our runs.

Strong Scaling - Winter Panorama - NVIDIA cluster						
Nodes	Cores	Ideal (m)	Actual (m)	Eff. %	O/H (m)	% O/H
2	16	128.87	128.87	100.0	8.63	6.7
4	32	64.43	77.68	82.9	4.70	6.1
8	64	32.22	40.63	79.3	4.28	10.5
16	128	16.11	21.17	76.1	4.17	19.7
32	256	8.05	10.88	74.0	4.08	37.5
48	384	5.37	6.98	76.9	4.10	58.7
60	480	4.30	6.02	71.4	4.00	66.5

scaling. The Fall Panorama, due to its larger size begins to lose efficiency at around 32 nodes when I/O overhead begins to dominate. Even with this overhead, the efficiency remains acceptable. For the Winter Panorama, the I/O overhead does not effect performance up to 60 nodes and the implementation maintains efficiency throughout the test. Weak scaling tests were performed using a subimage of the Fall Panorama dataset. See Table 5.3 for the weak scaling results. As the number of cores increases so does the image resolution to be solved. The subimage was expanded from the center of the full image and iterations of the solver for all windows were locked at 1000 for testing to ensure no variation is due to slower converging image areas. As the figure shows, our implementation shows good weak scaling

Table 5.3: Weak scaling tests run on the NVIDIA cluster for the Fall Panorama dataset. As the number of cores, increases so does the image resolution to be solved. The image was expanded from the center of the full image. Iterations of the solver for all windows were locked at 1000 for testing to ensure no variation is due to slower converging image areas. As is shown, our implementation shows good efficiency even when running on the maximum number of cores.

Weak Scaling - NVIDIA cluster				
Nodes	Cores	Size (MP)	Time (min.)	Efficiency
2	16	100.66	5.55	100.00%
4	32	201.33	5.55	100.00%
8	64	402.65	5.53	100.30%
16	128	805.31	5.68	97.65%
32	256	1610.61	5.77	96.24%
60	480	3019.90	6.57	84.52%

efficiency even for 60 nodes with 480 cores. In all, we have produced a gradient domain solution to a dataset which in previous work the best known methods [89, 149] took hours to compute.

5.3.2.2 Longhorn cluster. To show the portability and MPI scalability of our framework and implementation, strong and weak scaling tests were performed on the largest dataset (Fall Panorama) on a second cluster. The strong scaling tests were performed from 2-128 nodes and the weak scaling tests, limited by the size of the image, were performed from 2-64 nodes. As shown in Table 5.4, our implementation maintains very good efficiency and timings for our strong scaling test up to the full 1024 cores available on the system. Much like the NVIDIA cluster, weak scaling tests were performed on a portion of the Fall

Table 5.4: To demonstrate the portability of our implementation, we have run strong scalability testing for the Fall Panorama on the Longhorn cluster from 2-128 nodes up to a total of 1024 cores. As the numbers show, we maintain good scalability and efficiency even when running on all available nodes and cores.

Strong Scaling - Fall Panorama - Longhorn				
Nodes	Cores	Ideal(m)	Actual(m)	Efficiency
2	16	84.07	84.07	100%
4	32	42.03	43.18	97%
8	64	21.02	21.85	96%
16	128	10.51	12.08	87%
32	256	5.25	6.93	76%
64	512	2.63	3.89	68%
128	1024	1.31	2.73	48%

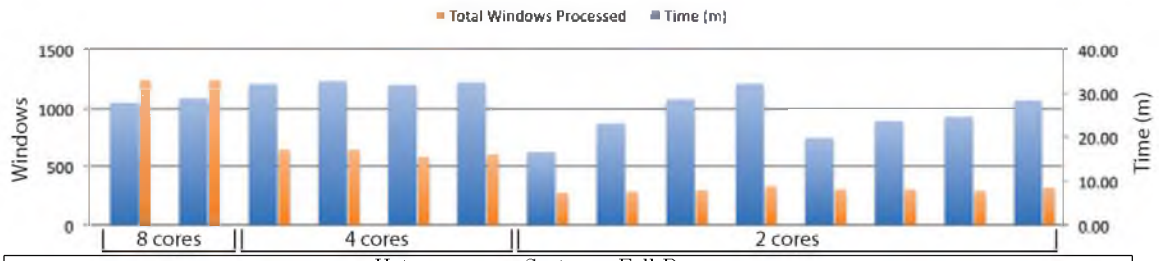
Panorama and iterations of the solver were locked at 1000. To ensure that each node got a reasonably sized subimage to solve, the tests were limited to 64 nodes. Table 5.5 demonstrates our implementations ability to weak scale on this cluster, maintaining good efficiency for up to 512 cores.

5.3.2.3 Heterogeneous cluster. As a final test of portability and adaptability, we presented our implementation with a simulated heterogeneous distributed system. Our parallel framework provides the ability to give weights to nodes which is typically even and therefore results in an even distribution of windows across all nodes. For this example, a simple weighting scheme can easily load-balance this mixed network, giving the nodes with more resources more windows to compute. Table 5.6 gives an example mixed system of two 8-core nodes, four 4-core nodes, and eight 2-core nodes. In all, this system has 48 available cores. The weights for our framework are simply the number of cores available in each node. This network was simulated using the NVIDIA cluster by overloading Hyperflow's knowledge of available resources with our desired properties. While this is not a perfect simulation since the main thread handling MPI communication would not be limited to reside on the desired cores, as shown in the strong scaling tests even with evenly distributed data on 8-16 nodes the implementation is not yet I/O bound. Therefore, we should still have a good approximation to a real, limited system. The figure details the window distribution and timings for the Fall Panorama for all nodes in this test. As is shown, we maintain good load balancing given proper node weighting when dealing with heterogenous systems. The max runtime of 32.70 minutes for this 48 core system is on par with run time for the 32 core (40.08 minutes) and 64 core (20.83 minutes) strong scaling results.

Table 5.5: Weak scaling tests run on the Longhorn cluster for the Fall Panorama dataset.

Weak Scaling - Longhorn cluster				
Nodes	Cores	Size (MP)	Time (min.)	Efficiency
2	16	75.5	5.50	100.00%
4	32	151	6.13	89.67%
8	64	302	6.15	89.43%
16	128	604	6.15	89.43%
32	256	1208	6.13	89.67%
64	512	2416	6.15	89.43%

Table 5.6: Our simulated heterogeneous system. This test example is a simulated mixed system of 2 8-core nodes, 4 4-core nodes, and 8 2-core nodes. The weights for our framework are the number of cores available in each node. The timings and window distributions are for Fall Panorama dataset. As you can see, with the proper weightings our framework can distribute windows proportionally based on the performance of the system. The max runtime of 32.70 minutes for this 48 core system is on par with timings for the 32 core (40.08 minutes) and 64 core (20.83 minutes) runs from the strong scaling test.



Heterogeneous System - Fall Panorama														
Cores	8		4				2							
Time(m)	27.9	28.9	32.1	32.7	32	32.5	16.6	23.1	28.7	32.2	20	23.6	24.6	28.4
Windows	1239	1239	640	640	580	600	276	285	300	330	304	304	290	319

5.4 Gradient Domain Editing on the Cloud

The parallel algorithm outlined in the previous section provides a full resolution gradient domain solution for massive images in only a few minutes of processing time. In this section, we explore redesigning this technique as a cloud-based application. For this work, we chose to target the MapReduce framework and its open source implementation, Hadoop. MapReduce and Hadoop have emerged in recent years as popular and widely supported cloud technologies. Therefore, they are the logical targets for this work.

5.4.1 MapReduce and Hadoop

This subsection briefly reviews some of the fundamentals of the MapReduce framework and how to design graphics algorithms to work well with Hadoop and Hadoop's Distributed File System (HDFS). We provide a high level view to justify design decisions outlined in the next section.

The map function operates on key/value pairs producing one or more key/value pairs for the reduce phase. The reduce function is a per-key operation that works on the output of the mapper (see Figure 5.15). Hadoop's scheduler will interleave their execution as data are available. Currently, Hadoop does not support job chaining. Therefore, any algorithm that requires two passes will likely require two separate MapReduce jobs. While this will likely change in the future, at this time minimizing the number of passes is an important consideration since the overhead incurred by launching new jobs in Hadoop is significant.

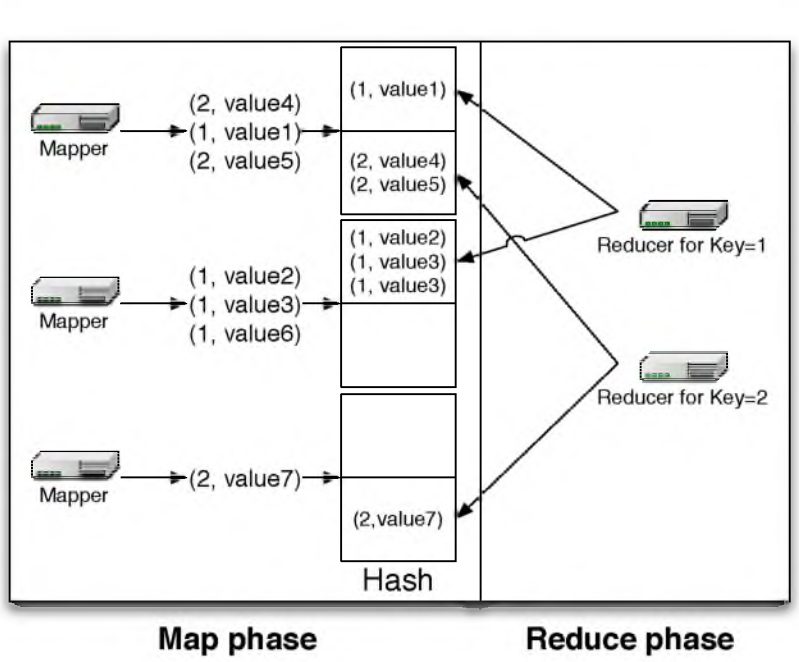


Figure 5.15: The two phases of a MapReduce job. In the figure, three map tasks produce key/values pairs that are hashed into two bins corresponding to the two reduce tasks in the job. When the data are ready, the reducers grab their needed data from the mapper's local disk.

In Section 5.4.2 we detail our algorithm, which requires only one pass.

Hadoop has been optimized to handle large files and to process/transfer small chunks of data. For many applications including the one outlined in the next section, understanding Hadoop's data flow is vital for an efficient implementation, much like random memory access must be considered in a GPU.

5.4.1.1 Input. The Hadoop distributed file system stripes data across all available nodes on a per block basis with replication to guarantee a certain level of locality for the map phase and to be able to handle system faults. When a job is launched, Hadoop will split the input data evenly for all map instances. For our example, allowing Hadoop to arbitrarily split the input data could result in fragmented images. Therefore, the system allows the developer to specialize the function reading the input which we use to constrain the split to only occur at image boundaries.

5.4.1.2 MapReduce transfer. During execution, each mapper hashes the key of each key/value pair into bins. The number of bins equal the number of reducers (see Figure 5.15) and each bin is also sorted by key. The map first stores and sorts the data in a buffer in memory but will spill to disk if this is exceeded (the default buffer size is 512

MB). This spill can lead to poor mapper performance and should be avoided if possible.

After a mapper completes execution, the intermediate data are stored to a node's local disk. Each mapper informs the control node that its data are finished and ready for the reducers. Since Hadoop assumes that any mapper is equally likely to produce any key, there is no assumed locality for the reducers. Each reducer must pull its data from multiple mappers in the cluster (see Figure 5.15 and 5.16). If a reducer must grab key/value pairs from many local disks on the cluster (possibly an N-to-N mapping), this phase can have drastic effect on the performance.

Job coordination is handled with a *master/slave* model where the control node, called the *Job Tracker* distributes and manages the map and reduce tasks. When a program is launched the *Job Tracker* initiates *Task Trackers* on nodes in the cluster. The *Job Tracker* then schedules tasks on the *Task Tracker* maintaining a communication link to handle system faults (see Figure 5.16).

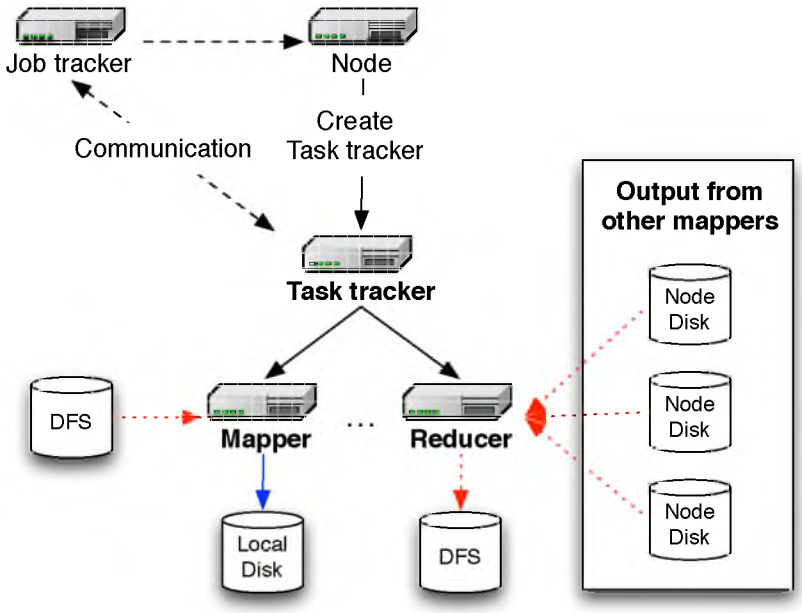


Figure 5.16: A diagram of the job control and data flow for one Task Tracker in a Hadoop job. The dotted, red arrows indicate data flow over the network; dashed arrows represent communication; the blue arrow indicates a local data write and the black arrows indicate an action taken by the node.

5.4.2 MapReduce for Gradient Domain

Commonly, large images are stored as tiles, which gives us the underlying structure for our scheme. However, a tiled-based approach by itself would not account for large scale trends common in panoramas (see Figure 5.17). Therefore, we add upsampling of a coarse solution similar to the approach used in Summa et al. [149] to capture these trends. Our algorithm works in two phases: The first phase performs the upsample of a precomputed coarse solution and solves each tile to produce a smooth solution over the extent of the tile. The second phase solves for a smooth image on tiles that significantly overlap the smoothed tiles from the first phase. In this way, the second phase smooths any seams not captured or even introduced by the first phase solvers. This algorithm can be simply implemented in one MapReduce job in Hadoop.

5.4.2.1 Tiles. We have chosen an overlap of 50% in both dimensions for the second phase due to the simplicity of implementation, although Summa et al. [149] has shown that a good solution can be found with much less. To easily accomplish this overlap, we divide the data into tiles 1/4 of the proper size. Figure 5.18 shows the tile layout for our test images. Each phase operates on four of these smaller tiles which are combined to construct the larger tiles. To avoid undefined tiles in the second phase, we add a symbolic padding of one row/column to all sides of the image. Figure 5.19 gives an example of a tile layout. An important component of panorama stitching is a map file which gives the correspondence from a pixel location in the overall panorama to the smaller image that supplies the color. This map file is necessary to determine the difference between actual gradients and those due to seams. This map also defines the boundaries of the panorama, which are commonly irregular and do not usually follow the actual image boundary. The panorama boundary is a seam we would like to preserve. We encode the map file into each individual tile as an alpha channel. For images such as the Salt Lake City example, we cannot encode an index

Figure 5.17: Although the result is a smooth image, without coarse upsampling the final image will fail to account for large trends that span beyond a single overlap and can lead to unwanted shifts in color. Notice the vertical banding denoted by the red arrows.

Figure 5.18: The 512 x 512 tiles used in our Edinburgh (a), Redrock (b), and Salt Lake City (c) examples.

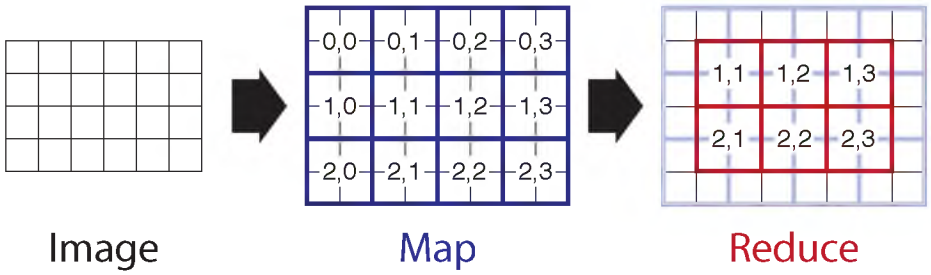


Figure 5.19: Our tile-based approach: An input image is divided into equally spaced tiles. In the map phase after a symbolic padding by a column and row in all dimensions, a solver is run on a collection of four tiles labeled by numbers above. After the mapper finishes, it assigns a key such that each reducer runs its solver a collection of four tiles that have a 50% overlap with the previous solutions.

for each image in a byte of data. However, the map is only used to denote if two pixels are from the same source image or if a pixel is on the boundary. Therefore a byte is more than enough to encode this correspondence. The symbolic padding is encoded as boundary and images that are not evenly divisible by our tile size are also padded with boundary. The overlapping window size used for our test was 1024×1024 pixels which we found was a good compromise between a low memory footprint and image coverage.

5.4.2.2 Coarse solution. As a first step, the first phase of our solver will upsample via bilinear interpolation a 1-2 megapixel coarse solution. Much like the method from Summa et al. [149], we precompute the coarse solution in just a few seconds using a direct FFT solver on a coarsely sampled version of our large image. In tiled hierarchies, this coarse image is typically already present. In Hadoop, this coarse solution is sent along with the MapReduce job when launched. The *Job Tracker* stores this image on the distributed file system for *Task Trackers* to pull and store locally.

5.4.2.3 First (map) phase. After loading/combining the smaller tiles and performing the upsample, the first phase runs an iterative solver initialized with the upsampled pixel colors. From our testing, we have found that SOR gives good running times and low memory consumption and therefore is our default solver. The solver is considered to have converged when the L_2 norm falls below 10^{-3} which is based on the range of byte data. After a smooth image is computed, the solution is split back into its four smaller tiles and sent to the next phase as byte data. Some precision is lost in the solution data by this truncation of bits and can cause slower convergence in the next phase. However, in many distributed systems, the bottleneck is data transfer; therefore it is preferable to use smaller data at the cost of increased computation. For the Hadoop implementation, this first phase of our algorithm fits well with Hadoop's map phase. Each mapper emits a key/value pair, where the value is the data from a small tile and the key is computed in such a way that we achieve the desired 50% overlap in the next phase. The key is computed as a row/column pair in the space of the larger tiles. This key is stored in 4 bytes before being emitted. The high word contains the row and the low word contains the column. For a tile at location (x, y), the key for sub-tile (i, j) is computed as:

$$key\_row = x * 2 + i; \tag{5.4}$$

$$key\_col = y * 2 + j; \tag{5.5}$$

Below we provide pseudocode for the map phase and Figure 5.19 provides an example.

5.4.2.4 Second (reduce) phase. The second phase now gathers the four smaller tiles that make up the overlapping window. These tiles sit as intermediate data on the local disks of the cluster. If the system accounts for locality, each instance would only need to gather three tiles since the nodes could be placed such that one tile is always stored locally. After the data are gathered, the gradients are computed from the original pixel values and an iterative solver (SOR) is run after being initialized with the solutions from the first

```
proc Map(blockId, image) ≡
  row := blockId >> 16;
  col := blockId & 0xFFFF;
  solver.compute_gradient(image);
  solver.upsample_coarse(image, row, col);
  solver.SOR(image);
  for i := 0 to 1 do
    for j := 0 to 1 do
      keyRow := row * 2 + i;
      keyCol := col * 2 + j;
      key := keyRow << 16 + keyCol;
      emit(key, solver.tiles[i][j]);
```

phase. The iterative solver is constrained to only work on interior pixels to prevent this phase from introducing new seams. Technically, there may be errors at the pixels around the midpoints of the boundary edges of these tiles, though in practice we have not seen this affect the solution. This second phase fits well with Hadoop's reduce phase with some considerations. Hadoop does not account for data locality for the reducers, therefore, we must assume the worst case gather of four tiles. Also, the reducers do not have access to the HDFS, nor can any task request specific data. The mappers in the first phase modify the pixel values, but the reducer needs the original values to compute the gradient vector for the iterative solver. Therefore, the mapper must also concatenate the original pixel values to the solved data before it emits the key/value pair. This leads to a 6 bytes/pixel transfer between phases. Below we provide pseudocode for the reduce phase.

```
proc Reduce(blockId, [(map1, org1), ..., (map4, org4)]) ≡
  mapper_output := merge(map1, map2, map3, map4);
  original_tile := merge(org1, org2, org3, org4);
  solver.compute_gradient(original_tile);
  solver.SOR(mapper_output);
  emit(BlockId, solver.tiles);
```

5.4.2.5 Storage in the HDFS. In Hadoop, saving the image in standard row major order would lead to poor performance in the mappers since there is good locality in only one dimension. Saving individual tiles would also not be efficient since Hadoop's HDFS is optimized for large files. Therefore, we save the data as the large tiles, comprised of the four

smaller tiles, which the mapper needs in the first phase. We concatenate the tiles together, row-by-row, into a single large file.

5.4.3 Results

We demonstrate the quality of our approach on three test panoramas which range from megapixels to gigapixels in size. We also demonstrate the generality of the abstraction by running our code, without modification, on a single desktop and on a large cluster. Finally, we test Hadoop's scalability with two of our test panoramas.

The single node tests were performed on a 2×Quad-Core Intel Xeon w5580 3.2GHz desktop with 24GB of memory. For our large distributed tests, we ran our method on the NSF CLuE [41] cluster, which consists of 275 nodes each with dual Intel Xeon 2.8GHz processors with HyperThreading and 8GB of memory. While still a valuable resource for research, as far as modern clusters are concerned, CLuE's hardware is outdated being a retired system based on a 6-year-old technology originally produced in 2004. Moreover, CLuE is also a shared resource and all timings were certainly affected by other researchers using the machines.

The Edinburgh panorama consists of 25 images with a full resolution of $16,950 \times 2,956$ pixels (50 megapixel) and was broken into 48 tiles. For our single node test, our method produced a solution in 81 seconds with eight mappers and four reducers. The Redrock panorama consists of nine images with a full resolution of $19,588 \times 4,457$ pixels (87 megapixel) and was partitioned into 96 tiles. Our method running on a single node solved the panorama in 156 seconds with nine mappers and nine reducers. The solver running on the cluster ran in 199 seconds with 96 mappers and 96 reducers. Due to the small size of the panoramas, the extra parallelization given to us by the distributed system did not increase performance. Quite the opposite was true, the runtimes were worse due to overhead of Hadoop launching and coordinating many tasks. Also, because the cluster was a shared resource, this increase in compute time could have easily come from external influences. See Figure 5.20 for the original and solved panoramas.

The Salt Lake City panorama consists of 611 images with a full resolution of $126,826 \times 29,633$ pixels (3.27 gigapixel) and was split into 3,444 tiles. Our method took 3 hours and 5 minutes to compute a solution on our one node test desktop. On the distributed cluster with 492 mappers and 492 reducers the time to compute a solution dropped to 28 minutes and 44 seconds of which 3 minutes and 24 seconds was due to Hadoop overhead and 15 minutes was due to I/O and data transfer between the map and reduce phases. Running

Figure 5.20: The results of our cloud implementation, from top to bottom: Edinburgh, 25 images, $16,950 \times 2,956$, 50 megapixel and the solution to Edinburgh from our cloud implementation; Redrock, nine images, $19,588 \times 4,457$; 87 megapixel and the solution to Redrock from our cloud implementation; Salt Lake City, 611 images, $126,826 \times 29,633$, 3.27-gigapixel and the solution to Salt Lake City from our cloud implementation.

Salt Lake City with 246 mappers and 246 reducers produced a solution in 39 minutes and 49 seconds of which 2 minutes and 7 seconds was due to Hadoop overhead and 30 minutes was due to I/O and data transfer. Note that these are all wall clock times and include activity of other people on a shared system. Moreover, the configuration, which we could not change, required running at least three processes on every node which have only two cores. Therefore, we can only hope to have 2/3 compute efficiency out of this cluster. See Figure 5.20 or the original and solved panorama. Based on our timing and the pricing available online, running the 492 mapper/reducer job would have cost approximately $50 to run on Amazon's Elastic Reduce [10]. This is orders of magnitude less expensive and time comsuming than operating and maintaining a proprietary cluster and would allow any researcher in the field to experiment with new ideas.

5.4.3.1 Scalability. Due to the shared nature of the CLuE cluster, we restricted our scalability tests to only the single node test desktop. Figure 5.21 plots the runtime to solve both the Edinburgh and Redrock panoramas as a function of number of reducers and mappers. We varied the number of mappers and reducers from one to the number of cores. The plot shows that as both the mappers and reduces increase so does our performance, but as the total number of both mappers and reducers meets or exceeds the available cores of our system, the performance gain flattens. This is an important observation and must be remembered when choosing an optimal number of mappers and reducers especially when purchasing time and cores as a commodity.

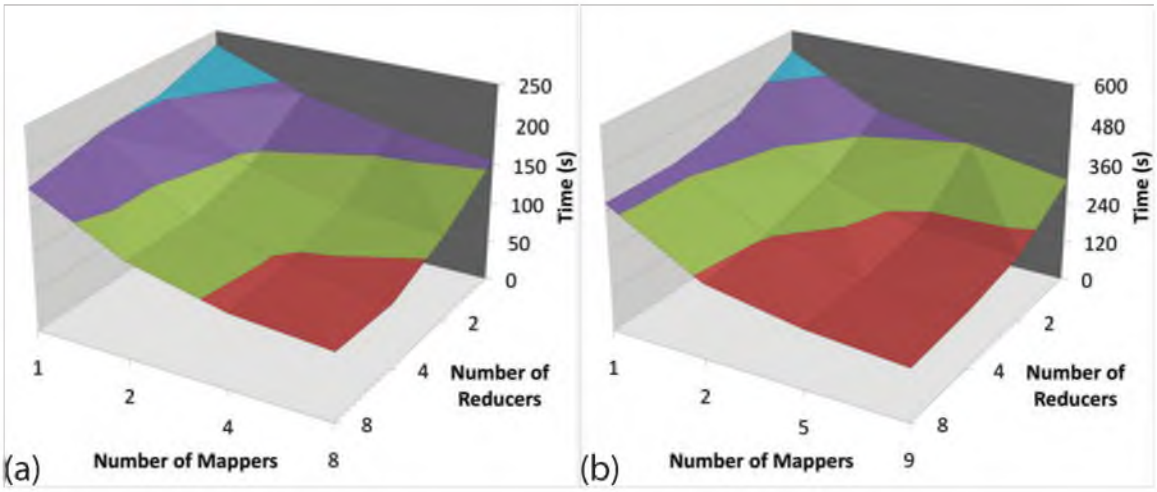


Figure 5.21: (a) The scalability plot for the Edinburgh (50 megapixel) panorama on our one node 8-core test desktop; (b) the scalability plot for Redrock (87 megapixel) panorama on the same machine

5.4.3.2 Fault tolerance. Hadoop has been developed to robustly handle failures in the cluster. Achieving a fault tolerant implementation is a major challenge on its own and is not easily available in other distributed frameworks such as MPI. The tremendous advantage of fault tolerance comes at the cost of high variability in running times, though jobs are guaranteed to finish. In fact, all runs on the distributed cluster had some kind of failure in the system at some time during the execution and still we were able to get results, which would not be available with a traditional distributed implementation. In particular, the running time stated above for the Salt Lake City example with 492 mappers/reducers was based on the job with the minimum number of failures (95 failed tasks). In practice, we have seen this example run as long as 49 minutes to account for the 133 failed tasked that occurred during the job.

CHAPTER 6

FUTURE WORK

The work outlined in this dissertation provides both the justification for and the solutions to bringing the composition stage of the panorama processing pipeline into an interactive setting. The future of this work is to bring the entire pipeline into an interactive environment. A system built with this guiding principle would allow the user to add and remove images, fix registration problems, or adjust image boundaries all while having a preview of the final color-corrected, composited panorama. This will give users an unprecedented amount of control over the creation of new panoramas, increasing both the accessibility of panorama creation and quality of the final results. Due to the work completed for this dissertation, the logical next step to achieving my ultimate goal is to provide new and interactive solutions for the registration phase of the panorama pipeline. Currently, interaction for registration is typically non-existent. Even when it is provided by a system, the interaction is rudimentary at best. For instance, the only interaction possible for a panorama processing system such as Hugin [77] is the manual selection and deletion of image feature points between pairs of images, see Figure 6.1.

This is a tedious process for small images, and completely unwieldy for larger image collections. The scaling of registration algorithms will also need further study. Despite significant previous work, many current methods have been shown to work with relatively small collections of images. Although assumed to scale well, only recently has work shown an extension of current techniques to work with extremely large collections of photographs [1]. Often such assumptions of scaling can be false; for instance some of the commercial and open-source products, although advertised to handle an arbitrary number of images, have failed when presented with panoramas with hundreds of images. General purpose algorithms for automatic registration of extremely large image collections remain an open avenue of investigation. In addition, state-of-the-art stitching software often needs a reduction of complexity by strictly enforcing that the images are acquired in a regular pattern (columns and rows) to reduce the search space for possible registrations. My collaborators and I have

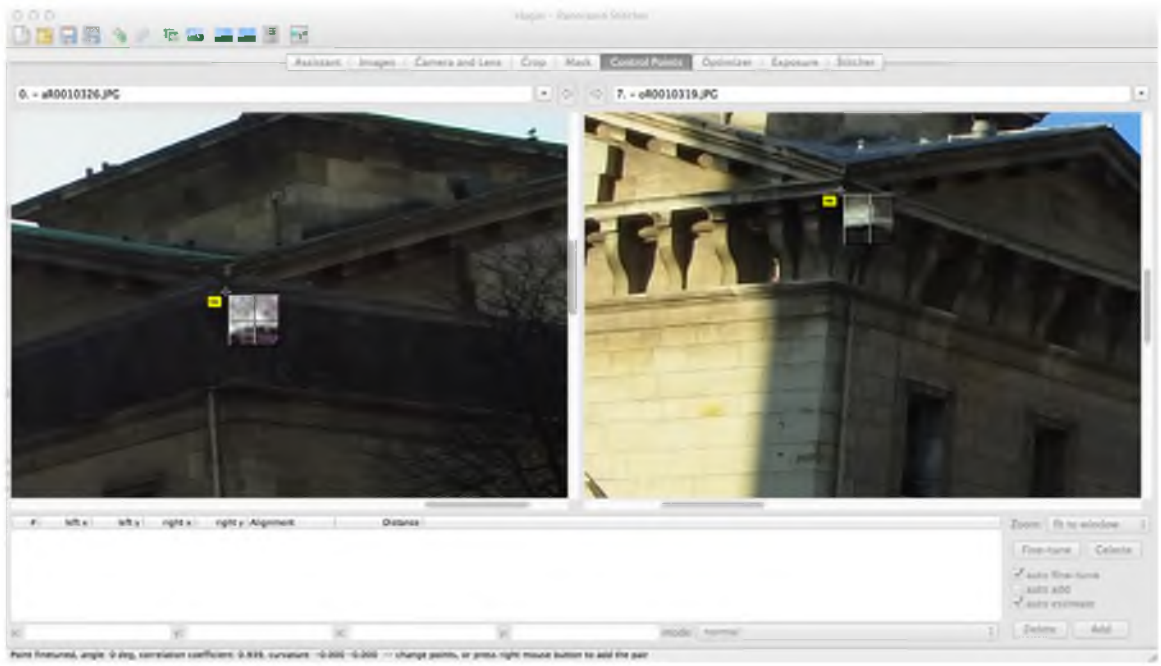


Figure 6.1: A typical example of interaction during panorama registration from the open-source Hugin [77] software tool. Current interaction is limited to the manual selection and deletion of feature points used during registration.

found that these programs will often fail when presented with large image collections with no assumed structure.

The focus of my dissertation work was to bring the composition stage of the panorama creation pipeline into an interactive setting, not only for small images, but for images massive in size. The *Progressive Poisson* and *Panorama Weaving* algorithms elegantly achieve this goal. Although panoramas were the primary focus of the work, the methods and frameworks developed throughout my dissertation provide new paradigms for interacting with high resolution imagery. For instance, the *Progressive Poisson* provides a working proof-of-concept on how to reformulate global algorithms to work in an interactive setting for large data by computing screen resolution previews in real-time and using out-of-core computation for full resolution solutions. One can envision expanding the frameworks and techniques outlined in this work with other data processing tools to allow comprehensive editing of massive datasets on regular desktop computers.

APPENDIX

MASSIVE DATASETS

Table A.1: Massive panorama data acquired and used in this dissertation work.

Dataset	Images	Format	Individual Image Size
Lake Louise Large	5794	RAW NEF 16-bit	4288 x 2848 (12 Megapixel)
Lake Louise Small 1	2805	RAW NEF 16-bit	4288 x 2848 (12 Megapixel)
Lake Louise Winter 1	1983	JPEG Fine	4288 x 2848 (12 Megapixel)
Lake Louise Winter 2	1876	JPEG Fine	4288 x 2848 (12 Megapixel)
Lake Louise Morning	1656	JPEG Fine	4288 x 2848 (12 Megapixel)
Lake Louise Small 2	1440	RAW NEF 16-bit	4288 x 2848 (12 Megapixel)
Salt Lake City Large	1311	JPEG Fine	3456 x 2592 (9 megapixels)
Lake Louise Evening	1220	JPEG Fine	4288 x 2848 (12 Megapixel)
Salt Lake City Winter	1219	JPEG Fine	3456 x 2592 (9 megapixels)
Salt Lake City Fall	624	JPEG Fine	3456 x 2592 (9 megapixels)
Mount Rushmore	300	JPEG Fine	4288 x 2848 (12 Megapixel)
Salt Lake City Small	132	JPEG Fine	3264 x 2448 (8 megapixels)

Table A.2: Massive satellite data acquired and used in this dissertation work.

Dataset	Resolution	Gigapixels
New York, NY	80000 x 80000	6.40
Chattanooga, TN	120000 x 100000	12.00
Washington, DC	131350 x 159375	20.93
Hamilton County, SC	240000 x 232000	55.68
Philadelphia, PA	250000 x 230000	57.50
Indianapolis, IN	260000 x 260000	67.60
San Diego, CA	200000 x 365000	73.00
San Francisco, CA	225000 x 330000	74.25
New Orleans, LA	330000 x 290000	95.70
Olympia, WA	501059 x 329220	164.96
San Antonio, TX	521640 x 492480	256.90
Atlanta, GA	524288 x 524288	274.88
Seattle, WA	411280 x 693528	285.23
Phoenix, AZ	720000 x 540000	388.80

REFERENCES

[1] Agarwal, S., Snavely, N., Seitz, S., and Szeliski, R. Bundle adjustment in the large. *ECCV'10: Proceedings of the 11th European conference on Computer vision: Part II* (Jan. 9), 29–42.

[2] Agarwala, A. Efficient gradient-domain compositing using quadtrees. *ACM Trans. Graph 26*, 3 (2007), 94.

[3] Agarwala, A., Agrawala, M., Cohen, M. F., Salesin, D., and Szeliski, R. Photographing long scenes with multi-viewpoint panoramas. *ACM Trans. Graph 25*, 3 (2006), 853–861.

[4] Agarwala, A., Dontcheva, M., Agrawala, M., Drucker, S. M., Colburn, A., Curless, B., Salesin, D., and Cohen, M. F. Interactive digital photomontage. *ACM Trans. Graph 23*, 3 (2004), 294–302.

[5] Agarwala, A., Zheng, K. C., Pal, C., Agrawala, M., Cohen, M. F., Curless, B., Salesin, D., and Szeliski, R. Panoramic video textures. *ACM Trans. Graph 24*, 3 (2005), 821–827.

[6] Agrawal, A., Raskar, R., Nayar, S. K., and Li, Y. Removing photography artifacts using gradient projection and flash-exposure sampling. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM, pp. 828–835.

[7] Agrawal, A. K., Chellappa, R., and Raskar, R. An algebraic approach to surface reconstruction from gradient fields. In *ICCV* (2005), pp. I: 174–181.

[8] Agrawal, A. K., Raskar, R., and Chellappa, R. What is the range of surface reconstructions from a gradient field? In *ECCV* (2006), pp. I: 578–591.

[9] Agrawal, A. K., Xu, Y., and Raskar, R. Invertible motion blur in video. *ACM Trans. Graph 28*, 3 (2009).

[10] Amazon. Elastic mapreduce, 2012. http://aws.amazon.com/elasticmapreduce.

[11] Anandan, P., Burt, P. J., Dana, K., Hansen, M. W., and van der Wal, G. S. Real-time scene stabilization and mosaic construction. In *Image Understanding Workshop* (1994), pp. I:457–465.

[12] AutoPano. http://www.autopano.net.

[13] Axelsson, O. *Iterative Solution Methods.* Cambridge Universty Press, New York, NY, 1994.

[14] Badra, F., Qumsieh, A., and Dudek, G. Rotation and zooming in image mosaicing. In *WACV* (1998), pp. 50–55.

[15] Bae, S., Paris, S., and Durand, F. Two-scale tone management for photographic look. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM, pp. 637–645.

[16] Bai, X., Wang, J., Simons, D., and Sapiro, G. Video snapcut: Robust video object cutout using localized classifiers. *ACM Trans. Graph 28*, 3 (2009).

[17] Balmelli, L., Kovacevic, J., and Vetterli, M. Quadtrees for embedded surface visualization: Constraints and efficient data structures. In *in Proc. of IEEE International Conference on Image Processing* (1999), pp. 487–491.

[18] Bay, H., Tuytelaars, T., and Gool, L. J. V. SURF: Speeded up robust features. In *ECCV* (2006), pp. I: 404–417.

[19] Ben-Ezra, M., and Nayar, S. K. Motion-based motion deblurring. *IEEE Trans. Pattern Anal. Mach. Intell 26*, 6 (2004), 689–698.

[20] Berger, M. J., and Colella, P. Local adaptive mesh refinement for shock hydrodynamics. *Journal Computational Physics 82* (1989), 64–84.

[21] Bolitho, M., Kazhdan, M., Burns, R., and Hoppe, H. Multilevel streaming for out-of-core surface reconstruction. In *SGP '07: Proceedings of the fifth Eurographics Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 69–78.

[22] Bornemann, F. A., and Krause, R. Classical and cascadic multigrid - a methodological comparison. In *In Proceedings of the 9th International Conference on Domain Decomposition Methods* (1996), Domain Decomposition Press, pp. 64–71.

[23] Boukerche, A., and Pazzi, R. W. N. Remote rendering and streaming of progressive panoramas for mobile devices. In *ACM Multimedia* (2006), K. Nahrstedt, M. Turk, Y. Rui, W. Klas, and K. Mayer-Patel, Eds., ACM, pp. 691–694.

[24] Boykov, Y., and Kolmogorov, V. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell 26*, 9 (2004), 1124–1137.

[25] Boykov, Y. Y., and Jolly, M. P. Interactive graph cuts for optimal boundary and region segmentation of objects in N-D images. In *ICCV* (2001), pp. I: 105–112.

[26] Boykov, Y. Y., Veksler, O., and Zabih, R. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Analysis and Machine Intelligence 23*, 11 (Nov. 2001), 1222–1239.

[27] Brandt, A. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation 31*, 138 (1977), 333–390.

[28] Briggs, W. L., Henson, V. E., and McCormick, S. F. *A Multigrid Tutorial (2nd Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[29] Brown, D. C. Close-range camera calibration. *Photogrammetric Engineering 37*, 8 (Aug. 1971), 855–866.

[30] Brown, M., and Lowe, D. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision* (Jan. 2007).

[31] Brown, M., and Lowe, D. G. Recognising panoramas. In *ICCV* (2003), IEEE Computer Society, pp. 1218–1227.

[32] Brown, M., Szeliski, R. S., and Winder, S. A. J. Multi-image matching using multi-scale oriented patches. In *CVPR* (2005), pp. I: 510–517.

[33] Buchanan, A., and Fitzgibbon, A. W. Combining local and global motion models for feature point tracking. In *CVPR* (2007), pp. 1–8.

[34] Capel, D., and Zisserman, A. Automated mosaicing with super-resolution zoom. *Proc. CVPR* (Jan. 1998).

[35] Cham, T. J., and Cipolla, R. A statistical framework for long-range feature matching in uncalibrated image mosaicing. In *CVPR* (1998), pp. 442–447.

[36] Chandran, L. S., and Sivadasan, N. Geometric representation of graphs in low dimension. In *Proceedings of the 12th Annual International Conference on Computing and Combinatorics* (Berlin, Heidelberg, 2006), COCOON'06, Springer-Verlag, pp. 398–407.

[37] Chartrand, G., and Harary, F. Planar permutation graphs. *Ann. Inst. Henri Poincare 3*, 4 (1967), 433–438.

[38] Chen, S. E. Quicktime VR: An image-based approach to virtual environment navigation. In *SIGGRAPH* (1995), pp. 29–38.

[39] Cho, S., and Lee, S. Fast motion deblurring. *ACM Trans. Graph 28*, 5 (2009).

[40] Chow, E., Falgout, R. D., Hu, J. J., Tuminaro, R. S., and Yang, U. M. A survey of parallelization techniques for multigrid solvers. In *Parallel Processing for Scientific Computing*, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds., vol. 20 of *Software, Environments, and Tools*. SIAM, Philadelphia, PA, Nov. 2006, pp. 179–201. ch. 10,.

[41] CluE. Clue program, 2008. http://www.nsf.gov/pubs/2008/nsf08560 /nsf08560.htm.

[42] Cohen, M. F., Shade, J., Hiller, S., and Deussen, O. Wang tiles for image and texture generation. *ACM Trans. Graph 22*, 3 (2003), 287–294.

[43] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, Cambridge, MA, 2009.

[44] Correa, C. D., and Ma, K.-L. Dynamic video narratives. *ACM Trans. Graph 29*, 4 (2010).

[45] Criminisi, A., Sharp, T., Rother, C., and Pérez, P. Geodesic image and video editing. *ACM Trans. Graph 29*, 5 (2010), 134.

[46] Davis, J. E. Mosaics of scenes with moving objects. In *CVPR* (1998), pp. 354–360.

[47] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *CACM 51*, 1 (2008), 107–113.

[48] Delong, A., and Boykov, Y. A scalable graph-cut algorithm for N-D grids. In *CVPR* (2008), IEEE Computer Society.

[49] Demmel, J. W. *Applied Numerical Linear Algebra*. SIAM, 1997.

[50] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[51] Dorr, F. W. The direct solution of the discrete poisson equation on a rectangle. *SIAM Review 12*, 2 (April 1970), 248–263.

[52] Edelsbrunner, H., and Mücke, E. P. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics 9*, 1 (Jan. 1990), 67–104.

[53] Efros, A. A., and Freeman, W. T. Image quilting for texture synthesis and transfer. In *SIGGRAPH* (2001), pp. 341–346.

[54] Farbman, Z., Hoffer, G., Lipman, Y., Cohen-Or, D., and Lischinski, D. Coordinates for instant image cloning. In *SIGGRAPH '09: Proceedings of the 36th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2009), ACM.

[55] Fattal, R., Lischinski, D., and Werman, M. Gradient domain high dynamic range compression. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), ACM, pp. 249–256.

[56] Fergus, R., Singh, B., Hertzmann, A., Roweis, S. T., and Freeman, W. T. Removing camera shake from a single photograph. *ACM Transactions on Graphics 25*, 3 (July 2006), 787–794.

[57] Finlayson, G. D., Hordley, S. D., and Drew, M. S. Removing shadows from images. In *ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part IV* (London, UK, 2002), Springer-Verlag, pp. 823–836.

[58] Flusser, J., Boldys, J., and Zitová, B. Moment forms invariant to rotation and blur in arbitrary number of dimensions. *IEEE Trans. Pattern Anal. Mach. Intell 25*, 2 (2003), 234–246.

[59] Flusser, J., and Suk, T. Degraded image analysis: An invariant approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence 20*, 6 (1998), 590–603.

[60] Ford, L. R., and Fulkerson, D. R. Maximal flow through a network. *Canadian Journal of Mathematics 8* (1956), 399–404.

[61] Freedman, D., and Zhang, T. Interactive graph cut based segmentation with shape priors. In *CVPR* (2005), pp. I: 755–762.

[62] Gall, D. L. Mpeg: A video compression standard for multimedia applications. *Communications of the ACM* (Jan 1991).

[63] GigaPan. http://www.gigapan.org/about.php.

[64] Goldman, D. B. Vignette and exposure calibration and compensation. *IEEE Trans. Pattern Anal. Mach. Intell 32*, 12 (2010), 2276–2288.

[65] Google. Google Earth http://earth.google.com/.

[66] Gortler, S., and Cohen, M. Variational modeling with wavelets. In *Symposium on Interactive 3D Graphics* (1995), pp. 35–42.

[67] Goshtasby, A. A. *2-D and 3-D Image Registration: For Medical, Remote Sensing, and Industrial Applications*. Wiley-Interscience, New York, NY, Mar. 2005.

[68] Gottfrid, D. Self-service, prorated super computing fun!, 2007. http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun.

[69] Gracias, N. R. E., Mahoor, M. H., Negahdaripour, S., and Gleason, A. C. R. Fast image blending using watersheds and graph cuts. *Image and Vision Computing 27*, 5 (Apr. 2009), 597–607.

[70] Griebel, M., and Zumbusch, G. Parallel multigrid in an adaptive pde solver based on hashing and space-filling curves. *Parallel Comput. 25*, 7 (1999), 827–843.

[71] Hadoop. Applications and organizations using hadoop, 2012. http://wiki.apache.org/hadoop/PoweredBy.

[72] Hassin, R. Maximum flow in (s, t)-planar networks. *Inform. Proc. Lett. 13* (1981), 107.

[73] Heath, Ng, and Peyton. Parallel algorithms for sparse linear systems. *SIREV: SIAM Review 33* (1991).

[74] HiRISE, High Resolution Imaging Science Experiment http://hirise.lpl.arizona.edu/.

[75] Hockney, R. W. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM 12*, 1 (Jan. 1965), 95–113.

[76] Horn, B. K. P. Determining lightness from an image. *Comput. Graphics Image Processing 3*, 1 (Dec. 1974), 277–299.

[77] Hugin. http://hugin.sourceforge.net.

[78] Ikeda, S., Sato, T., and Yokoya, N. High-resolution panoramic movie generation from video streams acquired by an omnidirectional multi-camera system. In *Proceedings of IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems* (Aug. 2003), p. 155.

[79] Jia, J., Sun, J., Tang, C.-K., and Shum, H.-Y. Drag-and-drop pasting. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, 2006), ACM, pp. 631–637.

[80] Jia, J., and Tang, C.-K. Tensor voting for image correction by global and local intensity alignment. *IEEE Trans. Pattern Anal. Mach. Intell 27*, 1 (2005), 36–50.

[81] Jia, J., and Tang, C.-K. Image stitching using structure deformation. *IEEE Trans. Pattern Anal. Mach. Intell 30*, 4 (2008), 617–631.

[82] Jia, J. Y., and Tang, C. K. Image registration with global and local luminance alignment. In *ICCV* (2003), pp. 156–163.

[83] Jia, J. Y., and Tang, C. K. Eliminating structure and intensity misalignment in image stitching. In *ICCV* (2005), pp. II: 1651–1658.

[84] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM 24*, 1 (Jan. 1977), 1–13.

[85] Joshi, N., Kang, S. B., Zitnick, C. L., and Szeliski, R. Image deblurring using inertial measurement sensors. *ACM Trans. Graph 29*, 4 (2010).

[86] Joshi, N., Szeliski, R., and Kriegman, D. J. PSF estimation using sharp edge prediction. In *CVPR* (2008), pp. 1–8.

[87] Kazhdan, M. Reconstruction of solid models from oriented point sets. In *Eurographics Symposium on Geometry Processing* (2005), pp. 73–82.

[88] Kazhdan, M., Bolitho, M., and Hoppe, H. Poisson surface reconstruction. In *Eurographics Symposium on Geometry Processing* (2006), pp. 61–70.

[89] Kazhdan, M., and Hoppe, H. Streaming multigrid for gradient-domain operations on large images. *ACM ToG. 27*, 3 (2008).

[90] Kazhdan, M., Surendran, D., and Hoppe, H. Distributed gradient-domain processing of planar and spherical images. *Transactions on Graphics (TOG 29*, 2 (Mar 2010).

[91] Kazhdan, M. M., and Hoppe, H. Streaming multigrid for gradient-domain operations on large images. *ACM Trans. Graph 27*, 3 (2008).

[92] Kolmogorov, V., and Zabih, R. What energy functions can be minimized via graph cuts? *IEEE Trans. Pattern Anal. Mach. Intell 26*, 2 (2004), 147–159.

[93] Kopf, J., Cohen, M. F., Lischinski, D., and Uyttendaele, M. Joint bilateral upsampling. *ACM Trans. Graph 26*, 3 (2007), 96.

[94] Kopf, J., Uyttendaele, M., Deussen, O., and Cohen, M. F. Capturing and viewing gigapixel images. *ACM Trans. Graph 26*, 3 (2007), 93.

[95] Kourogi, M., Kurata, T., Hoshino, J., and Muraoka, Y. Real-time image mosaicing from a video sequence. In *ICIP (4)* (1999), pp. 133–137.

[96] Kruger, S., and Calway, A. Image registration using multiresolution frequency domain correlation. *Proc. British Machine Vision Conf* (Jan 1998).

[97] Kuglin, C. D., and Hines, D. C. The phase correlation image alignment method. *Assorted Conferences and Workshops* (Sept. 1975), 163–165.

[98] Kumar, S., Pascucci, V., Vishwanath, V., Carns, P., Hereld, M., Latham, R., Peterka, T., Papka, M., and Ross, R. Towards parallel access of multi-dimensional, multi-resolution scientific data. In *Petascale Data Storage Workshop (PDSW), 2010 5th* (Nov. 2010), pp. 1 –5.

[99] Kumar, S., Vishwanath, V., Carns, P., Summa, B., Scorzelli, G., Pascucci, V., Ross, R., Chen, J., Kolla, H., and Grout, R. Pidx: Efficient parallel i/o for multi-resolution multi-dimensional scientific datasets. In *Proceedings of IEEE Cluster 2011* (Sep. 2011).

[100] Kundur, D., and Hatzinakos, D. Blind image deconvolution. *IEEE Signal Processing Magazine 13*, 3 (May 1996), 43–64.

[101] Kwatra, V., Schödl, A., Essa, I., Turk, G., and Bobick, A. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics 22*, 3 (July 2003), 277–286.

[102] Lawder, J. K., and King, P. J. H. Using space-filling curves for multi-dimensional indexing. In *LNCS* (2000), Springer Verlag, pp. 20–35.

[103] Levin, A., Zomet, A., Peleg, S., and Weiss, Y. Seamless image stitching in the gradient domain. In *ECCV* (2004), pp. Vol IV: 377–389.

[104] Li, Y., Sun, J., Tang, C.-K., and Shum, H.-Y. Lazy snapping. *ACM Trans. Graph 23*, 3 (2004), 303–308.

[105] Lischinski, D., Farbman, Z., Uyttendaele, M., and Szeliski, R. Interactive local adjustment of tonal values. *ACM ToG 25*, 3 (2006), 646–653.

[106] Liu, J., and Sun, J. Parallel graph-cuts by adaptive bottom-up merging. In *CVPR* (2010), IEEE, pp. 2181–2188.

[107] Lombaert, H., Sun, Y. Y., Grady, L., and Xu, C. Y. A multilevel banded graph cuts method for fast image segmentation. In *ICCV* (2005), pp. I: 259–265.

[108] Lowe, D. G. Object recognition from local scale-invariant features. In *ICCV* (1999), pp. 1150–1157.

[109] Lucas, B., and Kanade, T. An iterative image registration technique with an application to stereo vision. *International Joint Conference on Artificial Intelligence 3* (1981), 674–679.

[110] Maling, D. H. *Coordinate Systems and Map Projections.* Butterworth-Heinemann, Woburn, MA, 1993.

[111] Mann, S., and Picard, R. W. Virtual bellows: Constructing high quality stills from video. In *ICIP (1)* (1994), pp. 363–367.

[112] Matungka, R., Zheng, Y., and Ewing, R. Image registration using adaptive polar transform. *Image Processing, IEEE Transactions on 18*, 10 (2009), 2340 – 2354.

[113] McCann, J. Recalling the single-FFT direct poisson solve. In *SIGGRAPH Posters* (2008), ACM, p. 71.

[114] McCann, J., and Pollard, N. S. Real-time gradient-domain painting. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–7.

[115] McGuire, M. An image registration technique for recovering rotation, scale and translation parameters. *NEC Res. Inst. Tech. Rep., TR* (1998), 98–018.

[116] McLauchlan, P., and Jaenicke, A. Image mosaicing using sequential bundle adjustment. *Image and Vision Computing* (Jan. 2002).

[117] Meehan, J. *Panoramic Photography.* Amphoto, Oct. 1990.

[118] MegaPOV. http://megapov.inetart.net.

[119] Milgram, D. L. Computer methods for creating photomosaics. *IEEE Trans. Computer 23* (1975), 1113–1119.

[120] Milgram, D. L. Adaptive techniques for photomosaicking. *IEEE Trans. Computer 26* (1977), 1175–1180.

[121] Mills, A., and Dudek, G. Image stitching with dynamic elements. *Image and Vision Computing 27*, 10 (Sept. 2009), 1593–1602.

[122] Mortensen, E. N., and Barrett, W. A. Intelligent scissors for image composition. In *SIGGRAPH* (1995), pp. 191–198.

[123] Mortensen, E. N., and Barrett, W. A. Interactive segmentation with intelligent scissors. *Graphical models and image processing: GMIP 60*, 5 (Sept. 1998), 349–384.

[124] Nagahashi, T., Fujiyoshi, H., and Kanade, T. Image segmentation using iterated graph cuts based on multi-scale smoothing. In *ACCV* (2007), pp. II: 806–816.

[125] NASA. NASA Blue Marble http://earthobservatory.nasa.gov/Features/BlueMarble/.

[126] Niedermeier, R., Reinhardt, K., and Sanders, P. Towards optimal locality in meshindexings. In *Proc. Fundamentals of Computation Theory* (1997), vol. 1279 of *LNCS*, Spinger, pp. 364–375.

[127] Ojansivu, V., and Heikkila, J. Image registration using blur-invariant phase correlation. *IEEE Signal Processing Letters 14*, 7 (July 2007), 449–452.

[128] Pascucci, V., and Frank, R. J. Hierarchical indexing for out-of-core access to multi-resolution data. In *Hierarchical and Geometrical Methods in Scientific Visualization*, Mathematics and Visualization. Springer, New York, NY.

[129] Pascucci, V., and Frank, R. J. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing (SC'01)* (2001), p. 2.

[130] Pascucci, V., Laney, D. E., Frank, R. J., Gygi, F., Scorzelli, G., Linsen, L., and Hamann, B. Real-time monitoring of large scientific simulations. In *SAC* (2003), ACM, pp. 194–198.

[131] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S. R., and Stonebraker, M. A comparison of approaches to large scale data analysis. In *SIGMOD* (Providence, RI, USA, 2009).

[132] Peleg, S., Rousso, B., Acha, A. R., and Zomet, A. Mosaicing on adaptive manifolds. *IEEE Trans. Pattern Analysis and Machine Intelligence 22*, 10 (Oct. 2000), 1144–1154.

[133] Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. *ACM Trans. Graph. 22*, 3 (2003), 313–318.

[134] Philip, S., Summa, B., Bremer, P.-T., and Pascucci, V. Parallel gradient domain processing of massive images. In *Eurographics Symposium on Parallel Graphics and Visualization* (Llandudno, Wales, UK, 2011), T. Kuhlen, R. Pajarola, and K. Zhou, Eds., Eurographics Association, pp. 11–19.

[135] Philip, S., Summa, B., Pascucci, V., and Bremer, P.-T. Hybrid cpu-gpu solver for gradient domain processing of massive images. In *ICPADS '11: Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 2011), IEEE Computer Society, pp. 244–251.

[136] Pretto, A., Menegatti, E., Bennewitz, M., Burgard, W., and Pagello, E. A visual odometry framework robust to motion blur. In *ICRA* (2009), IEEE, pp. 2250–2257.

[137] PTgui, 2012. http://www.ptgui.com.

[138] Rastogi, A., and Krishnamurthy, B. Localized hierarchical graph cuts. In *ICVGIP* (2008), IEEE, pp. 163–170.

[139] Ricker, P. M. A direct multigrid poisson solver for oct-tree adaptive meshes. *The Astrophysical Journal Supplement Series 176* (2008), 293–300.

[140] Roberts, F. *On the boxicity and cubicity of a graph.* Recent Progress in Combinatorics, 1969.

[141] Rosgen, B., and Stewart, L. Complexity results on graphs with few cliques. *Discrete Mathematics & Theoretical Computer Science 9*, 1 (2007).

[142] Rother, C., Kolmogorov, V., and Blake, A. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph 23*, 3 (2004), 309–314.

[143] Sagan, H. *Space-Filling Curves.* Spinger-Verlag, New York, NY, 1994.

[144] Sand, P., and Teller, S. Video matching. *ToG 23*, 3 (2004), 592–599.

[145] Shum, H. Y., and Szeliski, R. S. Construction and refinement of panoramic mosaics with global and local alignment. In *ICCV* (1998), pp. 953–956.

[146] Simchony, T., and Chellappa, R. Direct analytical methods for solving Poisson equations in computer vision problems. *IEEE Trans. Pattern Anal. Mach. Intell. 12* (1990), 435–446.

[147] Snavely, N., Garg, R., Seitz, S. M., and Szeliski, R. Finding paths through the world's photos. In *SIGGraph-08* (2008), pp. xx–yy.

[148] Stookey, J., Xie, Z., Cutler, B., Franklin, W. R., Tracy, D. M., and Andrade, M. V. A. Parallel ODETLAP for terrain compression and reconstruction. In *GIS* (2008), W. G. Aref, M. F. Mokbel, and M. Schneider, Eds., ACM, p. 17.

[149] Summa, B., Scorzelli, G., Jiang, M., Bremer, P.-T., and Pascucci, V. Interactive editing of massive imagery made simple: Turning Atlanta into Atlantis. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 7:1–7:13.

[150] Summa, B., Tierny, J., and Pascucci, V. Panorama weaving: Fast and flexible seam processing. *ACM Trans. Graph. 31*, 4 (July 2012), 83:1–83:11.

[151] Summa, B., Vo, H. T., Silva, C., and Pascucci, V. Massive image editing on the cloud. In *IASTED International Conference on Computational Photography (CPhoto 2011)* (2011).

[152] Sun, J., Jia, J., Tang, C.-K., and Shum, H.-Y. Poisson matting. *ACM Trans. Graph. 23*, 3 (2004), 315–321.

[153] Szeliski, R. Image mosaicing for tele-reality applications. In *Proceedings of the Second IEEE Workshop on Applications of Computer Vision* (1994), pp. 44–53.

[154] Szeliski, R. Video mosaics for virtual environments. *Computer Graphics and Applications, IEEE 16*, 2 (1996), 22 – 30.

[155] Szeliski, R. Image alignment and stitching: A tutorial. *Foundations and Trends in Computer Graphics and Vision 2*, 1 (2006).

[156] Szeliski, R., and Shum, H.-Y. Creating full view panoramic image mosaics and environment maps. *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (Aug 1997).

[157] Szeliski, R. S. Video mosaics for virtual environments. *IEEE Computer Graphics and Applications 16*, 2 (Mar. 1996), 22–30.

[158] Tai, Y. W., Du, H., Brown, M. S., and Lin, S. Image/video deblurring using a hybrid camera. In *CVPR* (2008), pp. 1–8.

[159] Tasdizen, T., Koshevoy, P., Grimm, B. C., Anderson, J. R., Jones, B. W., Watt, C. B., Whitaker, R. T., and Marc, R. E. Automatic mosaicking and volume assembly for high-throughput serial-section transmission electron microscopy. *Journal of Neuroscience Methods 193*, 1 (2010), 132 – 144.

[160] Toledo, S. A survey of out-of-core algorithms in numerical linear algebra. In *External memory algorithms*, Dimacs Series In Discrete Mathematics And Theoretical Computer Science. American Mathematical Society, Boston, MA, 1999, pp. 161–179.

[161] Triggs, B., McLauchlan, P., Hartley, R. I., and Fitzgibbon, A. W. Bundle adjustment: A modern synthesis. In *Vision Algorithms Workshop: Theory and Practice* (1999), pp. 298–372.

[162] Tuytelaars, T., and Mikolajczyk, K. Local invariant feature detectors: A survey. *Foundations and Trends in Computer Graphics and Vision 3*, 3 (2007), 177–280.

[163] Tzimropoulos, G., Argyriou, V., Zafeiriou, S., and Stathaki, T. Robust fft-based scale-invariant image registration with image gradients. *Pattern Analysis and Machine Intelligence, IEEE Transactions on DOI - 10.1109/34.55103 PP*, 99 (2010), 1 – 1.

[164] USGS,. United States Geological Survey http://www.usgs.gov/.

[165] Uyttendaele, M. T., Eden, A., and Szeliski, R. S. Eliminating ghosting and exposure artifacts in image mosaics. In *CVPR* (2001), pp. II:509–516.

[166] Valgren, C., and Lilienthal, A. J. SIFT, SURF & seasons: Appearance-based long-term localization in outdoor environments. *Robotics and Autonomous Systems 58*, 2 (2010), 149–156.

[167] Vineet, V., and Narayanan, P. J. CUDA cuts: Fast graph cuts on the GPU. In *Computer Vision on GPU* (2008), pp. 1–8.

[168] Vitter, J. S. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv. 33*, 2 (2001), 209–271.

[169] Vo, H., Bronson, J., Summa, B., Comba, J., Freire, J., Howe, B., Pascucci, V., and Silva, C. Parallel visualization on large clusters using mapreduce. In *Proceedings of the 2011 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (2011), p. (to appear).

[170] Vo, H. T., Osmari, D. K., Summa, B., Comba, J. L. D., Pascucci, V., and Silva, C. T. Streaming-enabled parallel dataflow architecture for multicore systems. *Comput. Graph. Forum 29*, 3 (2010), 1073–1082.

[171] Wang, B., Summa, B., Pascucci, V., and Vejdemo-Johansson, M. Branching and circular features in high dimensional data. *Visualization and Computer Graphics, IEEE Transactions on 17*, 12 (dec. 2011), 1902–1911.

[172] Ward, G. Hiding seams in high dynamic range panoramas. In *APGV* (2006), R. W. Fleming and S. Kim, Eds., vol. 153 of *ACM International Conference Proceeding Series*, ACM, p. 150.

[173] Weiss, Y. Deriving intrinsic images from image sequences. In *International Conference on Computer Vision* (2001), pp. 68–75.

[174] Wood, D. N., Finkelstein, A., Hughes, J. F., Thayer, C. E., and Salesin, D. Multiperspective panoramas for cel animation. In *SIGGRAPH* (1997), pp. 243–250.

[175] Wu, C. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT), 2007. http://cs.unc.edu/ ccwu/siftgpu.

[176] Xiong, Y., and Pulli, K. Fast image labeling for creating high-resolution panoramic images on mobile devices. In *ISM* (2009), IEEE Computer Society, pp. 369–376.

[177] Xiong, Y., and Pulli, K. Fast panorama stitching for high-quality panoramic images on mobile phones. *IEEE Transactions on Consumer Electronics* (Jan 2010).

[178] Xiong, Y., Wang, X., Tico, M., and Liang, C. Panoramic imaging system for mobile devices. *SIGGRAPH'09: Posters* (Jan 2009).

[179] Xu, D., Chen, Y., Xiong, Y., Qiao, C., and He, X. On the complexity of/and algorithms for finding shortest path with a disjoint counterpart. *IEEE/ACM Trans. on Networking 14*, 1 (2006), 147–158.

[180] Yahoo! Yahoo! expands its m45 cloud computing initiative, adding top universities to supercomputing research cluster. `http://research.yahoo.com/news/3374`.

[181] Yoon, M.-S.-E., and Lindstrom, M.-P. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1213–1220.

[182] Yoon, S.-E., Lindstrom, P., Pascucci, V., and Manocha, D. Cache-oblivious mesh layouts. *ACM Trans. Graph. 24*, 3 (2005), 886–893.

[183] Yuan, L., Sun, J., Quan, L., and Shum, H.-Y. Image deblurring with blurred/noisy image pairs. *ACM Trans. Graph 26*, 3 (2007), 1.

[184] Zitova, B., and Flusser, J. Image registration methods: A survey. *Image and Vision Computing 21*, 11 (Oct. 2003), 977–1000.

[185] Zoghlami, I., Faugeras, O., and Deriche, R. Using geometric corners to build a 2d mosaic from a set of images. *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on* (1997), 420 – 425.