MASSIVE IMAGE EDITING ON THE CLOUD

Brian Summa[*]
SCI Institute
University of Utah
USA

Huy T. Vo[†]
SCI Institute
University of Utah
USA

Valerio Pascucci[‡]
SCI Institute
University of Utah
USA

Claudio Silva[§]
SCI Institute
University of Utah
USA

ABSTRACT

Processing massive imagery in a distributed environment currently requires the effort of a skilled team to efficiently handle communication, synchronization, faults, and data/process distribution. Moreover, these implementations are highly optimized for a specific system or cluster, therefore portability or improved performance due to system improvements is rarely considered. Much like early GPU computing, cluster computing for graphics is a highly-specialized field for few experts.

In this work, we experiment using the cloud as a possible alternative to the status quo, abstracting away much of the complexity associated with current implementations. As gigapixel images increase in prevalence, the need for a higher level of abstraction for broadly accessible deployment is clear, much like the emergence of CUDA, OpenCL and DirectCompute for multicore and general purpose GPU computing. The increased availability of cloud resources as a commodity offers a unique opportunity to adopt this level of abstraction and extend the distribution and development of large image algorithms to a much wider community. This can potentially lead to a drastic decrease in deployment time for algorithms allowing for faster testing of new ideas. The abstraction of the cloud can allow simple, system oblivious implementations which are more portable, fault-tolerant, and likely to scale as hardware improves.

In this paper, we detail how to reformulate graphics algorithms to perform well on the cloud and what considerations need to be made for an efficient implementation. Specifically, we show the use of gradient domain techniques to stitch large panoramas on Apache Foundation's open source implementation of Google's MapReduce framework called Hadoop. With the proper redesign of current algorithms, we show how one can balance processing and data movement to achieve implementations well suited for the cloud.

KEY WORDS

Gradient domain editing, Cloud computing, Image stitching, MapReduce, Hadoop

[*]e-mail: bsumma@sci.utah.edu
[†]e-mail: hvo@sci.utah.edu
[‡]e-mail: pascucci@sci.utah.edu
[§]e-mail: csilva@sci.utah.edu

1 Introduction

Due to the availability of high-resolution cameras and inexpensive robots to automatically capture large image collections [19], gigapixel images are becoming increasingly more prevalent. Tools to create and share large, stitched panoramas are easily accessible over the internet. Larger images, gigapixels in size, are freely distributed online such as satellite imagery from the United States Geological Survey (USGS) website and planetary images from the High Resolution Imaging Science Experiment (HiRISE).

Methods have been recently developed to process these images on commodity hardware, but even the most clever out-of-core method cannot hope to achieve the performance of a large distributed system. Despite the continued emergence of these massive format images, distributed methods still remain complicated to implement due to the need to distribute data and computation efficiently, handle faults, and coordinate the cluster nodes. This often results in highly specialized code written in many months and with great effort by a group of programmers. Often, this code is so specifically tailored to a platform or system that porting requires too much duplication of effort and is frequently not considered. However this specialization typically leads to outstanding performance, taking problems that once took hours to compute down to mere minutes.

We owe the inspiration for this paper to the emergence of GPU programming in the 1990s where the move from hardware specific implementations by experts to open standards revolutionized research and the industry. One can also draw from multicore programming's more recent history with the emergence of APIs such as CUDA SDK, OpenCL and DirectCompute which offer a level of abstraction above the graphics pipeline for general purpose GPU (GPGPU) computing. Much like the current use of distributed systems, GPGPU programming was once an area of expertise of a dedicated few.

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. In this paper we explore the cloud 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. For example, the Salt Lake City example in this paper would have cost a mere $50 to compute using Amazon's Elastic Reduce [6].

In the following, we outline how to extend a particularly expensive imaging technique, gradient domain image processing, to the cloud. Processing in the gradient domain offers a very specific challenge for parallelized solutions since it requires the capture of large scale trends that are only typically found in full, global solutions. In Section 2 we cover the related work in gradient domain methods, their solution, computing the solution in a distributed or out-of-core environment, and our cloud implementation of choice, Hadoop. Section 3 gives details on Hadoop and the MapReduce framework. In Section 4, we outline the basics of gradient domain processing and detail our panorama stitching implementation for MapReduce and Hadoop. Section 5 discusses the results for our test images and scalability of our implementation. We conclude with a discussion about the work in Section 6.

In particular, our contributions are the following:

- The first distributed Poisson solver for imaging implemented in Hadoop;
- A new tiling method to solve a Poisson system for image editing that captures large scale trends and only requires a small memory footprint with no additional disk storage requirements except where required by Hadoop;
- A practical example of extending a modern graphics algorithm into the MapReduce framework, detailing the main challenges that must be addressed for an efficient implementation.

2 Related Work

Poisson Image Processing. Gradient-based editing is a computationally expensive yet fundamental piece of any advanced image editing application. Given a guiding gradient field constructed from one or multiple source images, these methods attempt to find a smooth image that is closest fit in a least squares sense to the guiding gradient. This concept has been adapted for seamless cloning [36], drag-and-drop pasting [26] as well as matting [40]. Furthermore, gradient-based techniques can reduce the range of HDR (High Dynamic Range) images for display on standard monitors [17] or hide the seams in panoramas [2, 29, 32, 36]. Other applications include detecting lighting [25] or shapes from images [43], removing shadows [18] or reflections [5], and painting in the gradient domain [34]. An alternative to gradient based methods using Mean-Value Coordinates has been introduced to smoothly interpolate the boundary difference between images to mimic Dirichlet boundary conditions [16]. Currently, this technique does not extend to parallel solutions on distributed systems for applications such as panorama stitching due to the dependency between solved images and their unsolved neighbors.

Poisson Solution. Gradient based image processing typically requires the solution to a 2D Poisson problem. Computing the solution to Poisson equations efficiently, in parallel, or on distributed systems has been the focus of a large body of work; therefore we only offer a cursory review in this paper. Methods exist to find a direct Poisson solution using Fast Fourier transforms [3, 4, 24, 33]. With our chosen framework, direct FFT methods would require two full MapReduce jobs to compute. In the next section we will outline why it is desirable to have a minimum number of passes. Often the Poisson problem is simplified by discretization into a large linear system whose dimension is typically the number of pixels in an image. Methods exist to find the direct solution to the linear system. We refer the reader to Dorr [15] who provides an extensive review on direct methods and Heath et al. [23] who provide a survey on parallel algorithms. Often, especially in distributed systems, it is a far simpler implementation to use an iterative method to find a solution. Iterative Krylov subspace methods, such as conjugate gradient, are often used due to their fast convergence. However for larger linear systems, memory consumption is the limiting factor and iterative methods such as Successive Over-Relaxation (SOR) [7] are preferred.

If accuracy is not a driving factor, a coarse approximation to the Poisson solution may be sufficient. Extending a coarse solution to finer resolutions using Bilateral upsampling [31] has been shown to produce good results for applications such as tonemapping. Such methods have not yet been shown to handle applications such as panorama stitching where the solution is typically not smooth at the seams between images.

Often multigrid methods are used to aid the convergence of an iterative solver. These techniques include preconditioners [20, 41] and multigrid solvers [10, 11]. There exist different variants of multigrid algorithms using either adaptive [1, 8, 9, 28, 37] or non-adaptive meshes [27, 29]. There has been much work in distributed multigrid methods and we refer the reader to [12] for a survey of current methods.

Recently, it has been shown that combining upsampling and the coarse-to-fine half of a multigrid cycle gives quality results for imaging [39]. In this paper, it was shown that an initial coarse solution, when upsampled and used as the initialization of an iterative solver, produces results visually indistinguishable from a direct solution.

Out-of-Core. We refer the reader to Toledo [42] for a survey of general out-of-core algorithms for linear systems. Most algorithms surveyed assume that at least the solution can be kept in main memory, though this is rarely the case for large images. For out-of-core processing of large images, the streaming multigrid method of Kazhdan and Hoppe [29] and the progressive Poisson method [39] have so far provided the only solutions. Recently, streaming multigrid has been extended to a distributed environment [30] and has reduced the time to process gigapixel images from hours to minutes. Out-of-core methods often achieve a low memory footprint at the cost of significant disk storage requirements. For example, the multigrid method [29] requires auxiliary storage an order of magnitude greater than the input size, almost half of which is due to gradient storage. 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 their terapixel example, the method had a minimum requirement of 16 nodes in order to accommodate the needed disk space for fast local caching. In contrast, our approach needs only 6 bytes/pixel of temporary storage to work well with Hadoop and no additional storage if chosen to be implemented with standard MPI. Our method could solve, albeit slowly, any image on one node as long as the original and solved data in byte format could be saved to disk (with an additional temporary copy of the original data when using Hadoop). Streaming multigrid also assumes that the image gradient is precomputed, which would require a separate distributed pass. Our approach solves directly from the original image data in one MapReduce pass by computing gradients on the fly. The multigrid method [29, 30] 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.

Systems for large imagery often store images as a collection of tiles at the highest resolution; therefore methods that use this structure would be desirable. Stookey et al. [38] use a tiled base approach to compute an over-determined Laplacian PDE. 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 single pass method cannot account for large scale trends beyond a single overlap and therefore can only be used on problems which only have local trends. This method also maintains a single pass at the cost of a 8 times increase in solver computation and transfers 4 times the original data size.

MapReduce and Hadoop. MapReduce [14] 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:

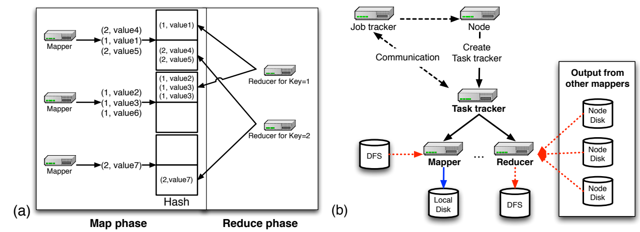

Figure 1: *(a) The two phases of a MapReduce job. In the figure, three map tasks produce key/values pairs that are hashed into two bins corresponding to the two reduce tasks in the job. When the data is ready, the reducers grab their needed data from the mapper's local disk (b) A diagram of the job control 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.*

- 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. [35] found that Hadoop was easy to deploy and use, offered adequate 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 [6]. Companies have begun to use Hadoop as a simple alternative for data processing on large clusters [22]. For instance, The New York Times has used Hadoop for large scale image to PDF conversion [21] . Google, IBM, and NSF have also partnered to provide a Hadoop cluster for research [13].

3 Hadoop

This section 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 1(a)). Hadoop's scheduler will interleave their execution as data is 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. In Section 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.

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.

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 1(a)) 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 is stored to a node's local disk. Each mapper informs the control node that its data is 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 1). 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 1(b)).

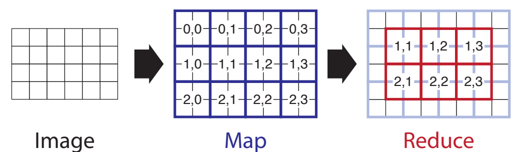

Figure 2: *Our tile-based approach: An input image is divided into equally spaced tiles. In the map phase after a symbolic padding by a column and row in all dimensions, a solver is run on a collection of 4 tiles labeled by numbers above. After the mapper finishes, it assigns a key such that each reducer runs its solver a collection of 4 tiles that have a 50% overlap with the previous solutions.*

4 MapReduce Image Processing

This section briefly reviews the Poisson system at the core of gradient domain image processing. We then detail our algorithm to produce a Poisson solution on the cloud.

4.1 Gradient Domain Methods

Rather than operating directly on the pixel values, gradient based techniques manipulate an image based on the value of a gradient field. 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:

$$\min_P \iint_\Omega \|\nabla P - \vec{G}\|^2 \tag{1}$$

Minimizing equation (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 $\vec{G}$.

For images, we discretize the differential form $\Delta P = div\ \vec{G}(x, y)$ using finite differences into the following sparse linear system: $L\mathbf{p} = \mathbf{b}$. Each row of the matrix L stores the weights of the standard five point Laplacian stencil, $\mathbf{p}$ is the vector of pixel colors, and $\mathbf{b}$ encodes the guiding gradient field. Both L and $\mathbf{b}$ encode boundary conditions as well. Gradient domain techniques are typically defined by how the guiding gradient field is computed and what boundary conditions are chosen. For instance, seamless cloning uses Dirichlet boundaries set to the color values of the background image and the foreground image's gradient as the guiding field (see Perez et al. [36] for a detailed description). For panorama stitching, Neumann boundary conditions are used and the guiding gradient field is computed as a composite of the gradients from the source images. At image boundaries, where an unwanted large gradient exists (the seams), the gradient between images is

Figure 3: *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 4: *The 512 x 512 tiles used in our Edinburgh (top), Redrock (middle), and Salt Lake City (bottom) examples.*

considered to be zero and the gradient on either side of the boundary is averaged across the seam [2, 29, 32, 36]. Another interesting example is gradient domain painting [34] which uses artistic input as the guiding field.

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 3). Therefore we add upsampling of a coarse solution similar to the approach used in Summa et al. [39] 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.

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. [39] 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 4 shows the tile layout for our test images. Each phase operates on 4 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 2 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 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.

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. [39], 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.

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 4 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 a 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:

$$key_row = x * 2 + i; \quad (2)$$

$$key_col = y * 2 + j; \quad (3)$$

Below we provide pseudocode for the map phase and Figure 2 provides an example.

```
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]);
```

Second (Reduce) phase. The second phase now gathers the 4 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 3 tiles since the nodes could be placed such that one tile is always stored locally. After the data is 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. 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 4 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);
```

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

5 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 [13] cluster, which consists of 275 nodes each with dual Intel Xeon 2.8GHz processors with Hyper-Threading and 8GB of memory. While still a valuable resource for research, as modern clusters are concerned CLuE's hardware is outdated being a retired system based on a 6 year old technology since it was 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 8 mappers and 4 reducers. The Redrock panorama consists of 9 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 9 mappers and 9 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 6 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 3444 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 min-

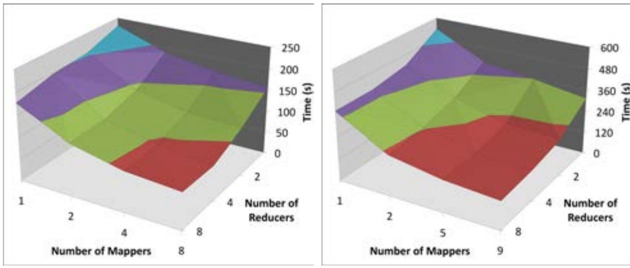

Figure 5: *(left) The scalability plot for the Edinburgh (50-megapixel) panorama on our one node 8-core test desktop; (right) the scalability plot for Redrock (87-megapixel) panorama on the same machine*

utes 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 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 3 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 7 for our results. 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 [6]. 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.

Scalability. Due to the shared nature of the CLuE cluster, we restricted our scalability tests to only the single node test desktop. Figure 5 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.

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.

Figure 6: *The results of our cloud implementations, from top to bottom: Edinburgh, 25 images,* $16,950 \times 2,956$*, 50-megapixel; the solution to Edinburgh from our cloud implementation; Redrock, 9 images,* $19,588 \times 4,457$*; 87-megapixel; the solution to Redrock from our cloud implementation.*

6 Conclusions

In this paper, we detailed how the cloud can be used as a possible alternative to a traditional distributed implementation for processing massive imagery. We introduced a new tiled-based method to solve a Poisson system for image applications that captures large scale trends and does not require large memory resources, nor does it accomplish this at the cost of significant data proliferation. We outlined how this method can be extended to the cloud and what needs to be carefully considered in order to ensure an efficient implementation. Finally, we presented the first Poisson solver for image editing implemented in the MapReduce framework.

We have shown how our abstracted, general implementation scales and runs well on not only a large distributed cluster but also a single node. Due to the high level implementation, this portable code is already able to take advantage of improvements in hardware or in the underlying Hadoop system. We demonstrated good running times for images that range from megapixels to gigapixels in size.

One can imagine new researchers and developers using similarly designed algorithms to perform new complex imaging operations on massive images quickly and cheaply. For instance, our Salt Lake City example would

Figure 7: *Salt Lake City Panorama: 611 images,* $126,826 \times 29,633$*, 3.27-gigapixel, (top) Original image (bottom) Our cloud solution.*

have cost approximately $50 to compute on a commodity Hadoop system [6]. As the history of GPU programming has shown, abstraction layers which remove some of the more tedious and complex elements of an implementation can have significant implications in accelerating new innovations.

Acknowledgements

This work is supported in part by the National Science Foundation awards OCI-0904631, OCI-0906379, IIS-1045032, IIS-0844572, CNS-0751152, and CCF-0702817. This work was also performed under the auspices of the U.S. Department of Energy BER and ASCR and under contract DE-SC0001922 and DE-FC02-06ER25781 by the University of Utah and DE-AC52-07NA27344 by Lawrence Livermore National Laboratory .

References

[1] Aseem Agarwala. Efficient gradient-domain compositing using quadtrees. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 94, New York, NY, USA, 2007. ACM.

[2] Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. Interactive digital photomontage. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, New York, NY, USA, 2004. ACM.

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

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

[5] Amit Agrawal, Ramesh Raskar, Shree K. Nayar, and Yuanzhen Li. Removing photography artifacts using gradient projection and flash-exposure sampling. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 828–835, New York, NY, USA, 2005. ACM.

[6] Amazon. Amazon web services - elastic mapreduce,. http://aws.amazon.com/elasticmapreduce.

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

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

[9] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Multilevel streaming for out-of-core surface reconstruction. In *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 69–78, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

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

[11] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. SIAM, 2nd edition, 2000.

[12] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. A survey of parallelization techniques for multigrid solvers. In Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors, *Parallel Processing for Scientific Computing*, volume 20 of *Software, Environments, and Tools*, pages 179–201. SIAM, Philadelphia, November 2006. ch. 10,.

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

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

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

[16] Zeev Farbman, Gil Hoffer, Yaron Lipman, Daniel Cohen-Or, and Dani Lischinski. 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.

[17] Raanan Fattal, Dani Lischinski, and Michael Werman. 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.

[18] Graham D. Finlayson, Steven D. Hordley, and Mark S. Drew. Removing shadows from images. In *ECCV '02: Proceedings of the 7th European Con-*

ference on Computer Vision-Part IV, pages 823–836, London, UK, 2002. Springer-Verlag.

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

[20] S. Gortler and M. Cohen. Variational modeling with wavelets. In *Symposium on Interactive 3D graphics*, pages 35–42, 1995.

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

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

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

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

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

[26] Jiaya Jia, Jian Sun, Chi-Keung Tang, and Heung-Yeung Shum. Drag-and-drop pasting. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 631–637, New York, NY, 2006. ACM.

[27] Michael Kazhdan. Reconstruction of solid models from oriented point sets. In *Eurographics Symposium on Geometry Processing*, pages 73–82, 2005.

[28] Michael Kazhdan, M. Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Eurographics Symposium on Geometry Processing*, pages 61–70, 2006.

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

[30] Michael Kazhdan, Dinoj Surendran, and Hugues Hoppe. Distributed gradient-domain processing of planar and spherical images. *ACM ToG. to appear*, 2010.

[31] Johannes Kopf, Michael F. Cohen, Dani Lischinski, and Matthew Uyttendaele. Joint bilateral upsampling. *ACM ToG*, 26(3):96, 2007.

[32] Anat Levin, Assaf Zomet, Shmuel Peleg, and Yair Weiss. Seamless image stitching in the gradient domain. In *In Eighth European Conference on Computer Vision (ECCV 2004*, pages 377–389. Springer, 2004.

[33] James McCann. Recalling the single-FFT direct poisson solve. In *SIGGRAPH Posters*, page 71. ACM, 2008.

[34] James McCann and Nancy S. Pollard. Real-time gradient-domain painting. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–7, New York, NY, USA, 2008. ACM.

[35] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. *SIGMOD '09*, 2009.

[36] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM ToG.*, 22(3):313–318, 2003.

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

[38] Jared Stookey, Zhongyi Xie, Barbara Cutler, W. Randolph Franklin, Daniel M. Tracy, and Marcus V. A. Andrade. Parallel ODETLAP for terrain compression and reconstruction. In Walid G. Aref, Mohamed F. Mokbel, and Markus Schneider, editors, *GIS*, page 17. ACM, 2008.

[39] Brian Summa, Giorgio Scorzelli, Ming Jiang, Peer-Time Bremer, and Valerio Pascucci. Interactive editing of massive imagery made simple: Turning atlanta into atlantis. *ACM Transactions on Graphics (TOG)*, 29(5), 2010.

[40] Jian Sun, Jiaya Jia, Chi-Keung Tang, and Heung-Yeung Shum. Poisson matting. *ACM ToG.*, 23(3):315–321, 2004.

[41] R. Szeliski. Locally adapted hierarchical basis preconditioning. *ACM ToG.*, 27(3):1135–1143, 2008.

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

[43] Yair Weiss. Deriving intrinsic images from image sequences. In *International Conference on Computer Vision*, pages 68–75, 2001.