Distributed Seams for Gigapixel Panoramas

Sujin Philip, Brian Summa, Julien Tierny, Peer-Timo Bremer, Valerio Pascucci

Distributed Seams for Gigapixel Panoramas

Sujin Philip, Brian Summa, Julien Tierny, Peer-Timo Bremer, and Valerio Pascucci

Abstract—Gigapixel panoramas are an increasingly popular digital image application. They are often created as a mosaic of many smaller images. The mosaic acquisition can take many hours causing the individual images to differ in exposure and lighting conditions. A blending operation is often necessary to give the appearance of a seamless image. The blending quality depends on the magnitude of discontinuity along the image boundaries. Often, new boundaries, or seams, are first computed that minimize this transition. Current techniques based on multi-labeling Graph Cuts are too slow and memory intensive for gigapixel sized panoramas. In this paper, we present a parallel, out-of-core seam computing technique that is fast, has small memory footprint, and is capable of running efficiently on different types of parallel systems. Its maximum memory usage is configurable, in the form of a cache, which can improve performance by reducing redundant disk I/O and computations. It shows near-perfect scaling on symmetric multiprocessing systems and good scaling on clusters and distributed shared memory systems. Our technique improves the time required to compute seams for gigapixel imagery from many hours (or even days) to just a few minutes, while still producing boundaries with energy that is on-par with Graph Cuts.

Index Terms—Panorama, Seams, Gigapixel, Parallel, Scalable, Out-Of-Core, MPI

1 Introduction

Panoramic images, composed as a mosaic of many smaller images, are an increasingly popular digital photography application. These images can range from a few megapixels to many gigapixels in size and can contain hundreds or thousands of individual images. Recently, the trend has been towards gigapixel sized panoramas due to the availability of high resolution cameras and inexpensive robots for automatic capture.

The robots capture the individual images one by one in a grid pattern and typically take a few seconds per image. Therefore, the capture of gigapixel sized panoramas can take many hours. This results in each image having different lighting and exposure conditions. The resulting panorama is an unappealing patchwork with visible transitions between the images. To combat this, blending operations are performed to give the appearance of a single seamless image. The quality of blending depends on the magnitude of discontinuity along the transition boundaries. Therefore, another step is usually performed where new boundaries between the images are computed such that the magnitude of transition between them is minimized. These boundaries are often called *seams*.

Seam computation determines which image provides the pixel value in the final panorama for regions where multiple images overlap, where each image is from a collection of well-registered and transformed images. The most widely used technique for this problem is Graph Cuts [1], [2], [3]. This technique computes a k-labeling to the nodes of a graph in order to minimize an energy function defined on the nodes and edges of the graph. For the seam computation problem, the panorama is represented as a graph where the pixels are the nodes and the edges connect a pixel to its neighbors [4], [5]. The labels specify the source images that provide the corresponding pixel values. The energy function is typically pixel based and is used to minimize the color or color gradient variations between images. Graph Cuts is only an approximation since the globally optimal solution for more than two labels is known to be NP-hard [1].

Graph Cuts is not suitable for gigapixel sized panoramas due to its high computational cost and memory requirements. A common technique employed to reduce both the memory and computational cost of Graph Cuts is to use a hierarchical scheme [6], [7]. Hierarchical Graph Cuts has been shown to produce good results for only two to three levels in practice. Figure 1 demonstrates a typical problem encountered at higher levels. In this figure, at four levels the dynamic scene element is small enough in the coarse solution for the seam to simply pass through it. The dilatation, which is the region around the upsampled coarse seams where refinement is performed, is not large enough (10 pixels is typical) to allow the seam to exit this local minima. Larger dilatation may help in this case although at an increased computational and memory cost. This interplay between a shallow hierarchy and dilatation parameters makes the technique inherently unscalable and not sufficient for direct application to gigapixel sized panoramas. A moving window based

- *Sujin Philip, Brian Summa and Valerio Pascucci are with the Scientific Computing and Imaging Institute, University of Utah. E-mail: sujin, bsumma, pascucci@sci.utah.edu.*
- *Julien Tierny is with the CNRS - LIP6 UPMC Sorbonne Universites, LTCI Telecom ParisTech, email: tierny@telecom-paristech.fr.*
- *Peer-Timo Bremer is with Lawrence Livermore National Laboratory. E-mail: bremer5@llnl.gov.*

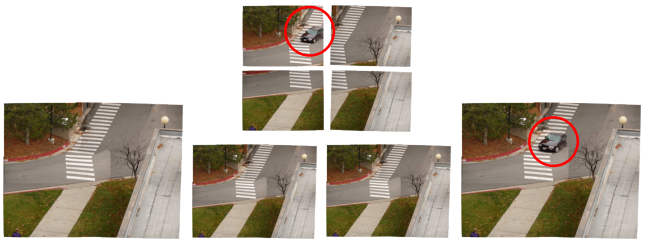


Fig. 1. An example case where a deep hierarchy for Hierarchical Graph Cuts produces poor results. A car that appears in only one of the input images (top) causes a region with high energy which a pixel based energy will avoid. The results for up to 3 levels are good. At 4 levels, in the coarse solution, the car is small enough for the seam to pass through it. The dilation parameter (10 pixels in the typical case) is not large enough to exit this local minima in the refinement stages.

out-of-core technique is presented in [8] where the Graph Cuts algorithm is sequentially applied over the footprint of each individual image of the panorama. Therefore, at each step the window includes only the current image and the overlapping regions of its neighbors. This technique has been shown to work on gigapixel sized panoramas but it is sequential and is still computationally and memory intensive, even over the domains of individual images. This can be sped up by solving the windows hierarchically but the problems related Hierarchical Graph Cuts will be introduced.

For faster seam computation, we look to the recently introduced alternative to Graph Cuts: Panorama Weaving [9]. This work presented a novel technique to combine independently computed pairwise seams into a global seam network. It has been shown to be fast, light-weight and easily parallelized. However, it is an in-core, parallel technique. In-core parallel algorithms do not often translate to efficient and scalable, out-of-core parallel nor to distributed parallel implementations. The original algorithm is not an exception. It has scalability issues since the memory usage increases with the size and number of input images. Hence, it has been shown to work only on panoramas of less than 100 megapixels. In addition, since all the threads have access to all the buffers via shared memory, it is not easily portable to large, distributed systems.

In this work, we introduce a scalable version of the Panorama Weaving technique. In doing this, we present the first truly out-of-core, parallel and scalable panorama seams technique that can handle arbitrary sized panoramas, yet requires limited memory independent of the total size of the panorama. We present the following contributions over the previous work:

- Enable processing of large panoramas, irrespective of the number of images, by minimizing memory usage to a fixed set of buffers active at any time. This enables the seams to be computed on a wide range of systems from single-core machines to many-core shared memory workstations to large distributed and distributed shared memory clusters.
- Minimize communications among distributed nodes, yet fully utilize shared memory buffers within each node, for efficient parallelization on modern distributed and distributed shared memory systems.
- Support configurable maximum memory usage to enable it to run more efficiently on systems with more memory by reducing redundant I/O and computations.

2 Related Work

Once the images of a panorama are registered into a common global frame, it is desirable to smooth the transitions between the images to give the impression of a single seamless image. A simple approach is to perform an alpha-blend over the overlap areas. Szeliski [10] provides an introduction to this and other blending techniques. Although highly scalable, these techniques are not suitable for cases where there are registration errors, dynamic elements, or varying lighting and exposure conditions across the images. These are very common for larger panoramas. Hard seams between the images that minimize the transition can often hide registration errors and dynamic elements and provide a good input for techniques such as gradient domain blending [11], [12].

Panoramas where images are acquired in a single sweep of the scene is a simplified case where only boundaries between pairwise images need to be considered [13], [14], [15], [16], [17], [18]. The optimal boundary between a pair of overlapping images can be computed quickly and exactly using the min-cut/max-flow algorithm. There has been some work to combine such pairwise seams for more complex panorama configurations. Gracias et al. [19] use an image distance based metric to combine the pairwise seams for more complex panoramas. Efros et al. [20] combine the seams by adding them together sequentially for the purpose of texture synthesis. Summa et al. [9] present a novel technique of combining these pairwise seams in a general panorama configuration by introducing the concept of a driving adjacency mesh to encode the boundary relations and intersections in a panorama.

Graph Cuts builds on the min-cut/max-flow algorithm [1], [2] to efficiently compute an approximate optimal solution for k-labeling of a graph. Graph Cuts has been adapted to the panorama seams problem [4], [5] and it is currently a widely used technique. However, it is a computationally expensive and memory intensive technique and is not suitable for gigapixel

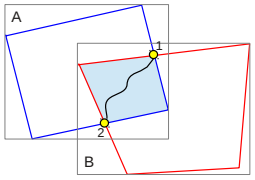


Fig. 2. Pairwise overlap between two images, A and B, with their boundaries intersecting at two points. The optimal pairwise seam lies in the overlap region and connects these points.

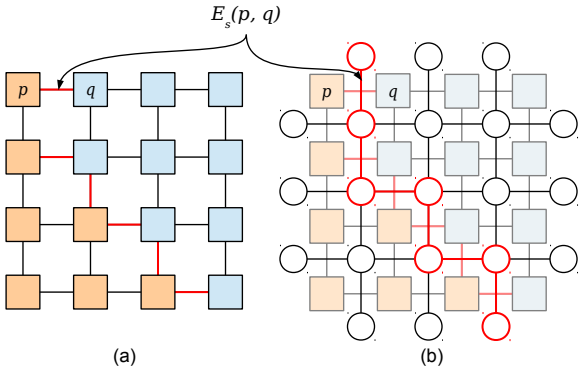


Fig. 3. (a) Min-cut solution on a 4-connected graph with an energy function E_s defined on its edges. (b) Its equivalent min-path solution on the dual.

sized panoramas. As mentioned in the previous section, a hierarchical scheme can be used to alleviate some of these problems but it is inherently unscalable. The only truly scalable technique is an out-of-core scheme where the Graph Cuts algorithm is sequentially applied over the footprint of each individual image [8]. As we will show in Section 5, this serial computation can take several hours to compute a solution for gigapixel sized panoramas.

There have been works that parallelize min-cut/max-flow on multicore systems [21], [22], [23] and GPUs [24]. The system in [23] is also capable of handling graphs that are larger than the available memory. However, min-cut/max-flow is not directly applicable to the panorama seams problem and extending it to multi-label Graph Cuts is non-trivial.

3 Panorama Weaving

In this section we give an overview of Panorama Weaving [9] which is the basis of our scalable seam computing technique.

Panorama Weaving produces a global seam solution by combining independently computed pairwise seams. Consider a pair of overlapping images as shown in Figure 2 with the boundaries of the images intersecting at two points. The optimal pairwise seam between the images lies in the overlap region and connects the intersection points. The overlap region can be represented as a graph with the pixels being

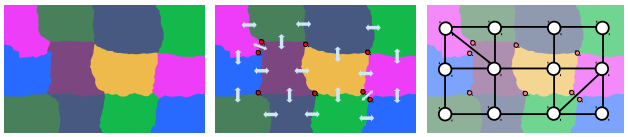


Fig. 4. (a) An example labeling of a panorama. (b) The labeling forms a set of connected regions separated by pairwise seams. Groups of seams join at points called Branching Points (shown in red). (c) A mesh representation of the panorama. The vertices correspond to the images, the edges correspond to the pairwise seams and are orthogonal to them, and the faces correspond to the branching points.

the nodes and edges connecting each pixel to its four-neighborhood. The optimal seam will be a min-cut of this graph based on some energy function E_s defined on the edges (Figure 3-a). It has been shown [25] that there is an equivalent min-path to a min-cut solution on its dual graph (Figure 3-b). This is true for all single-source, single destination paths. The dual graph is created with nodes between the pixels and edges orthogonal to the edges of the four-connected graph. One can then compute the min-path on this graph, with the intersection points as the source and destination, using a shortest path algorithm such as Dijkstra's [26].

The energy function is defined as $E_s(p, q)$ where $p, q \in \mathcal{N}$ and $\mathcal{N}$ is the set of all neighboring pixels. We would like to minimize the sum of the energy of all neighbors, E, with a labeling L. For the panorama boundary problem, this energy is typically [5] 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. The label of a pixel specifies the source image that provides the pixel value. Notice that $L(p) = L(q)$ implies $E_s(p, q) = 0$.

To illustrate how the pairwise seams combine into a global seam network, consider a typical labeling of a panorama as shown in Figure 4-a. The labels form a collection of connected regions (shown in solid colors) that are separated by pairwise seams. Groups of these seams meet each other at a common point, called the *branching point* of the seams. Figure 4-b shows the pairwise seams and the branching points. Note that this is a simplified model, but as the previous work has shown this assumption still provides quality seam solutions. The global seam network as described

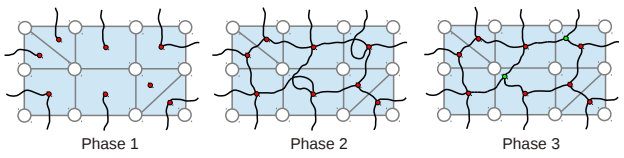


Fig. 5. The three phases to compute the global seam network. The faces can be processed independently of each other in each phase. Phase 1 computes the branching points and boundary seams, phase 2 computes the shared seams and phase 3 detects and resolves seam intersections.

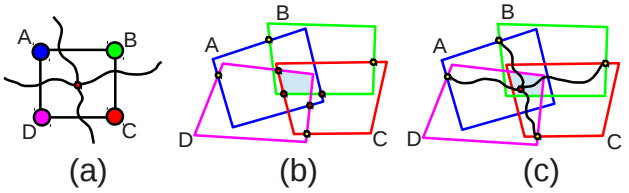


Fig. 6. (a) A quad face of a panorama mesh with its branching point and seams. (b) The collection of images that form the face. These images have a non-empty intersection called as the Multi-Overlap region (light blue). Consider the pairwise overlaps corresponding to the edges. An overlap's intersection point that is closer to the multi-overlap region is labeled as the inside point (red) and the other is labeled as the outside point (yellow). (c) The inside points are adapted into a branching point. The branching point is located inside the multi-overlap region and minimizes the sum of the min-path distances from the outside points.

above can be represented by an abstract mesh data structure which is the dual of the seam network as shown in Figure 4-c. The vertices of this mesh represent the images of the panorama, the edges represent the pairwise seams, to which they are orthogonal, and the faces represent the branching-points. Given such a mesh representation of a panorama, its global seam network can be computed by processing the faces of the mesh independently of each other in logically three phases. Figure 5 gives an overview of these phases. In the first phase the branching points of the faces are computed. The seams on the boundary of the mesh, called *boundary seams*, can then be found since their computation is independent of other faces. In the second phase the seams connecting the branching points, called *shared seams*, are computed. These seams are each shared by two faces, but only one needs to perform the computation. The seams of a face can intersect and cross-over each other causing areas of inconsistent labeling. In the third phase, intersections between seams of a face are detected and resolved.

The seams corresponding to the edges of a face meet at the face's branching point. Thus the branching point should be located in the region of intersection of all the pairwise overlaps represented by the edges, which is the same as the region of intersection of all the images corresponding to the vertices. Therefore, the images of a face must overlap each other. In short, the vertices of a face form a clique of overlap relationship. The intersection region is called as the face's *Multi Overlap* region. Figure 6-a shows an example configuration for a quad face. Only the pairwise overlaps represented by the edges of the face need to be considered. For each overlap, the intersection point that is closest to the multi-overlap region is labeled as the *inside* point and the other is labeled as the *outside* point (Figure 6-b). The pairwise seams are combined by replacing the inside points with the branching point as shown in Figure 6-c. A single-source/all-destinations min-path tree, called *Seam Tree*, is computed on the dual graph of each overlap with the corresponding outside point as the source. Each node of a seam-tree gives the cost of the path from the source to that node. Within the multi-overlap region, the node for which the sum of its costs from each of the seam-trees is minimized, gives a good location for the branching point.

For faces that contain the mesh-boundary edges, the boundary seams (orthogonal to those edges) are computed by a simple lookup in each of the edge's seam-tree from the face's branching point. Once the branching points for all the faces are computed, the shared seams, orthogonal to the interior edges, are computed by connecting the branching points of each edge's adjacent faces with a min-path on the energy field defined over its overlap.

There is a possibility that the paths of the seams of a face will intersect each other. The details about how and why the seams intersect, its implications and resolution are beyond the scope of this paper. Interested readers are referred to [9] for further details. At a high level, intersections between pairs of seams are handled by truncating the seams up to the furthest intersection point and computing a new seam path to the intersection point over a proper energy function.

The mesh representation of a panorama can be generated from the collection of its input images by creating an adjacency graph where the nodes represent the images and the edges represent pairwise overlaps between the images. All non-overlapping maximal cliques of this graph are identified and the edges that form the boundary of the cliques are activated. The cliques then become the faces and the active edges become the edges of the mesh.

4 Scalable Seams

Panorama Weaving is an in-core technique that needs all its images and intermediate buffers, such as overlap energies and seam trees, in memory. This is not

scalable to gigapixel panoramas as memory usage greatly increases with the total number of images. In this section, we describe our scalable seams technique.

One of the strengths of Panorama Weaving is that faces of the mesh representation can be processed independently. We use this feature to process one face at a time and only use the memory required per face. Moreover, processing within a face is ordered such that all the images and buffers of the face need not be active at all times and can be acquired and released as needed. There is a trade off between speed and memory as acquiring and releasing results in increased disk I/O and/or recomputation of buffers. Therefore, it is necessary to reduce the number of times a particular buffer is acquired for good performance. We use a caching scheme for this purpose. The size of this cache can be configured based on the amount of memory available in a particular system or user's preference. The rest of the section describes the Scalable Seams technique in detail:

Input. The input is typically a collection of images that have already been registered, transformed and rasterized along with their bounding boxes in the panorama. The invalid pixels of the images have an alpha value of zero and this property is used to identify the shape of the image.

Preprocessing. A pair of images is assumed to be overlapping if their bounding boxes overlap. This information is used to build the full adjacency graph of the images. The mesh representation is then computed from this adjacency graph as described in section 3. In case of gigapixel panoramas, since the images are typically acquired by robots on a grid, a simplifying assumption can be made that the panorama forms a quad mesh layout. The size of the mesh representation is negligible, only a few hundred kilobytes for panorama with thousands of images.

Pass 1. For each face of the mesh, its branching point and boundary seams (if any) are computed (Figure 5-left). The edges of a face are assigned a winding order (Figure 7 top) and we process the edges in that order. Figure 7 shows the various data buffers required for the computation of the branching point, and their dependencies. For each edge, the images corresponding to their vertices are loaded and their overlap energy is computed. The image buffers can now be released. Next, the seam tree is computed after which the energy buffer can be released. The costs of the nodes of the tree in the multi-overlap region are accumulated in a cost buffer, at which point the seam tree can be released. After the costs from all seam trees of the face have been accumulated, the location of the minimal value in the

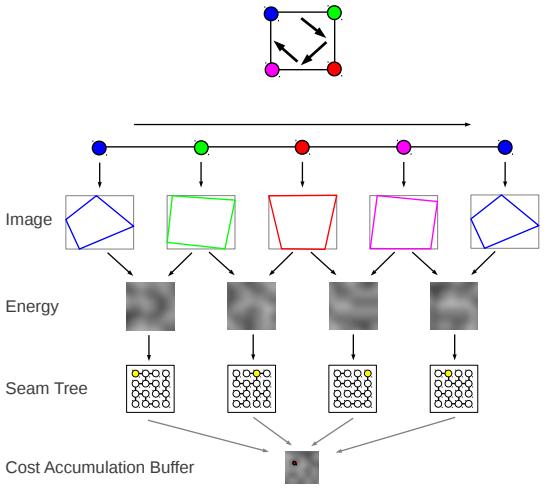


Fig. 7. Computation of the branching point of a face and the dependencies between the required data structures. The edges are iterated over based on a winding order (top). For each edge, the images corresponding to the two vertices are loaded and the overlap energy is computed. A single-source/all-destinations min-path tree (seam-tree) is computed on the energy. The costs to its nodes within the multi-overlap region are accumulated into an accumulation buffer. After the costs from all the edge's seam-trees have been accumulated, the minimal point in the accumulation buffer gives the location of the branching point. Buffers can be freed once their dependents have been computed. Edges are processed sequentially so only one set of these buffers need to be active at a time.

cost buffer will be the location of the branching point. Another iteration through the edges is performed and for mesh-boundary edges, the corresponding seam path can be found using the edge's seam tree. A cost-memory trade-off can be made at this point, if the seam trees of these edges are not freed in the first iteration. In this way reloading of the images and recomputation of the overlap energy and seam-tree can be avoided. Another such trade-off can be made for image loading. Since the second image of an edge becomes the first of the next edge, reloading of the image can be avoided by maintaining a circular buffer of length two for the images. The first vertex of the first edge is also the second vertex of the last edge. By keeping the first image around for the entire iteration on the edges, the number of times that each image of a face is loaded can be reduced to just once.

Pass 2. For each interior edge, the corresponding shared seam is computed as a min-path between the branching points of the adjacent faces, over the overlap energy (Figure 5-middle). The seam only needs to be computed for one of the faces, so there should be a consistent criteria to choose that face.

Passes 1 and 2 do not strictly need to be two separate passes. During the first pass a face can compute the proper seam path immediately after its branching point is computed if the adjacent face's branching point is available by then. Merging the two passes also allows us to save the overlap energy (computed during the first iteration over the edges) of the shared edges and reuse that for the min-path computation.

Pass 3. For each face, pairs of seams are checked for intersections (Figure 5-right). If detected, the furthest intersection point is identified and the seams are truncated to this point. Note that seams that are shared can have intersections in both of its faces. To be able to process this pass for the faces independently, the seam paths are implemented as double ended queues. To truncate a shared seam one face only needs to update its head and the other only updates its tail with no conflict between the two. A new seam is then computed from the branching point to the intersection point. The required images and overlap energies are loaded and computed as required.

Output. The output of our system is a set of seams. Each seam is represented by the labels of the images it separates and its path. The contributing region of each image in the panorama can be rasterized from the seams that have the image as one of their labels.

Pass 1 is the most computationally expensive part of the processing. Most of the time is spent in performing I/O to load images and in computing the seam-trees (single-source, all-destinations Dijkstra's algorithm) for each edge/overlap of a face. Though it uses a good number of buffers, only a few, fixed number of them need to be active at a time. Pass 2 can be eliminated by merging it with pass 1. It computes single-source, single-destination paths for each of the shared seams, however, its cheaper than computing seam-trees. During pass 3, only in cases where seam intersection occurs, the corresponding pair of images need to be loaded and buffers allocated for the images and overlap energy. Dijkstra's algorithm is run again to compute the non-intersecting paths.

To demonstrate its scalability and portability, we have implemented multiple versions of our technique—a single threaded out-of-core version, three shared memory multithreaded versions and a hybrid version that uses both message passing and multithreading.

4.1 Single Threaded Implementation

The single threaded version sequentially iterates over each individual face. Processing is performed in two passes. The computation of branching points and boundary seams (pass 1) and shared seams (pass 2) are merged into a single pass as described previously. Of the two adjacent faces that share a seam, the face which is processed second computes the shared seam. Intersection resolution is still performed in a separate pass. We chose to implement the cost-memory trade-offs mentioned in *Pass 1* above to achieve a balance between memory usage and performance.

4.2 Multithreaded Implementation

We have implemented three different versions of our technique using multithreading.

The first version is a *direct* parallelization of the sequential out-of-core version. A pool of threads is maintained and each thread picks its next work from a global queue of tasks. Initially, one task per face is put in the queue for branching point and seams computation. Atomic counters are used to determine which of the two adjacent faces of a shared seam will compute it. There is no separate pass for intersection resolution in this version. As soon as all the seams of a particular face have been computed, a new task is put in the queue for intersection resolution on that face. This way we avoid a stall in the pipeline that would have resulted from separate passes.

The second version is an extension of the direct parallel method. Note that nodes and edges are shared between neighboring faces. This corresponds to sharing of images and pairwise overlaps. Due to dynamic scheduling, it is likely that parallel threads are working on adjacent faces. Given that the buffers are read-only data, we can reduce I/O and computation time by sharing them among the threads. We call this the *sharing* version. The image and overlap buffers are tracked by reference counters. Whenever a thread requires a buffer, it increments its reference count and decrements it when it is done. A buffer is only freed when its reference count becomes zero.

The third is the *caching* version which further improves upon the previous version by not immediately freeing the buffers whose reference count have reached zero. Instead, it can be configured with a user defined cache size. When memory usage starts exceeding this threshold, a cache manager starts freeing buffers with reference count zero in a *Least Recently Used* manner until the memory usage returns to the set threshold or there are no buffers left with reference count of zero. This allows the implementation to use more memory in systems that can afford it to further reduce disk I/O and redundant computations. Note that the minimum memory usage will be the same as sharing version's memory usage. If the cache size is configured to a lesser value, this version will behave like the sharing version. In this version the task queue is implemented as a priority queue. When an intersection resolution task for a face is put in the queue, there is a good chance that the required buffers are in memory since the seams computation

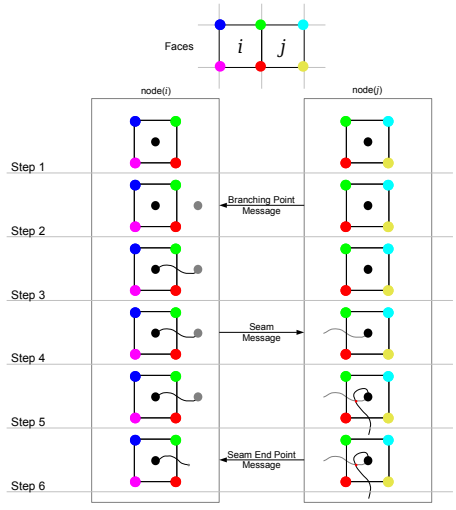


Fig. 8. Two adjacent faces i and j that are assigned to two different nodes, $node(i)$ and $node(j)$. Here $i < j$, so $node(i)$ is the owner node of the shared seam between i and j. Once the branching point of j is computed, it is sent to $node(i)$ so that it can compute the shared seam. The seam is then sent to $node(j)$ since it is required for intersection resolution. If the seam is found to be intersecting in j, the new end-point is sent to $node(i)$, as it is the owner node. The computations and communications are performed asynchronously for improved performance.

for that face would have been run recently. Therefore, intersection resolution tasks have a higher priority than branching point and seams computation tasks, so that they are executed as soon as possible.

4.3 Hybrid Implementation

The hybrid implementation extends the caching version of the multithreaded implementation to run on large systems with distributed memory compute nodes by adding a message passing layer. It uses message passing for communicating among the nodes and shared memory for communicating among the processors/cores within a node. The faces of the panorama mesh representation are statically distributed among the nodes during initialization. Within each node, the faces are dynamically processed with a task queue and thread pool, like the multithreaded implementations. The cache in this version is local to each node and can be configured according to the amount of memory available per node. The various buffers are only shared among the threads within a node. Each node does its own I/O and computations of buffers to minimize communication.

4.3.1 Inter-Node Communication

Nodes containing adjacent faces need to communicate certain data among them via messages, as shown in Figure 8. Shared seams between adjacent faces that are assigned to different nodes only need to be computed by one of the nodes. For consistency, we denote the node which has the face with a lower *id* value as the *owner* of the corresponding shared seam and is the node that maintains the seam. The node with the face with higher id value, called the *partner* node, sends its branching point to the owner node, which then computes the seam. After the seam is computed it is communicated to the partner node since both the faces need it for the intersection resolution phase. Intersection resolution may change the end points of a seam. For shared seams, the owner node receives the opposite end-point from the partner node and updates the seam.

The sending and receiving of messages are handled by a message passing engine, which internally uses MPI. Though the MPI standard supports multithreaded programs, the support for safe and fast multithreaded calls to MPI functions is not mature enough in many implementations and therefore, is not portable. Instead, threads pass control to the message passing engine at strategic locations by calling its send/receive routines which are protected by locks. When data required by another node is ready, it should be immediately sent so as to reduce latency. Therefore, these are good points to pass control to the engine. When a thread wants to send a message, it puts the message in a global send queue and the engine's send routine is called. The engine tries to acquire a lock and if successful, sends the messages in the queue. If the lock is owned by another thread, the engine just returns instead of blocking and the messages in the send queue are sent the next time the engine receives control from any thread. After a thread has finished executing a task, it again passes control to the engine where it sends any outstanding messages and polls for messages to be received. Since we are using asynchronous send/receive routines, this guaranties their progress in a portable manner by passing control to MPI routines regularly.

4.3.2 Computation Overview

In each node, the task queue is initialized with one task per assigned face for branching point and seams computation. This task first computes the branching point of the corresponding face. If the branching point is required by other nodes, it is immediately put in the send queue and the message passing engine's send routine is called as described above. The task then computes the seams of the face. Computation of shared seams, for which the adjacent faces are on the same node, are tracked using atomic counters. Shared seams with adjacent faces on different nodes are computed in their corresponding owner nodes. Such a seam can be computed only if the opposite branching point has already been received, else the computation is skipped instead of waiting. The receipt of the branching points are also tracked using atomic

variables. When the required branching point for a skipped shared seam is eventually received, a new task is put in the queue that just computes the seam. Immediately after a shared seam is computed, if it is required by another node it is put in the send queue and the message passing engine's send routine is called. Once all the seams of a face have been computed and/or received, a task for intersection resolution for the face is put in the task queue. After all the tasks have finished, the new end-points of remote shared seams are sent to their respective owner nodes. Points to be sent to the same node are coalesced into a single message. In the end, each node dumps its share of seams into separate files.

4.3.3 Task Distribution and Scheduling

Computation is distributed among the nodes by assigning equal number of faces to each of them. This results in some load imbalance since the faces can take different amounts of processing time. The load imbalance can be reduced by assigning weights to the faces based on their processing time and distributing them accordingly. The effectiveness of this scheme depends on how well the weights predict the actual execution time of the faces. One good heuristic is the sum of the sizes of the overlapping buffers corresponding to the edges of the face. This heuristic is based on the fact that a significant amount of processing time of a face is spent in computing the seam-trees which in turn depends on the area of the overlapping regions. We implemented a load balancing technique using the above heuristic for weights and an optimal chain-on-chain partitioning algorithm [27] for assigning the faces to the nodes. The load balanced version did not show consistent improvement over the trivial distribution version. We found that even with the trivial distribution the load imbalance is not very high and implementing a load balancing system is not worth the trouble.

Two important factors that affect the performance of our system are—the reuse of buffers already in memory/cache and the amount of time the threads are idling waiting for work. These factors depend on how the faces of the mesh are partitioned, the order in which they are processed and the priorities of the different types of tasks that processes the faces.

For good performance, the faces should be processed in an order that utilizes images that are already active or in the cache. For gigapixel sized panoramas, the faces can be assumed to form a grid-of-quads layout and they can be easily sorted according to *Z-order* indexing. Processing the faces in Z-order gives good caching performance, but it is not ideal for minimizing waits. Prioritizing faces that produce data required by neighboring nodes can minimize idling times by making such data available as soon as possible. A node should first process those faces whose branching points are required by neighboring nodes. This reduces latency of branching point messages but it also reduces the chance of redundant disk I/O and computations. As mentioned previously, computation of a shared seam between adjacent faces on different nodes is skipped in the seams computation task if the opposite branching point has not been received. This can result in extra I/O and computation as a new task is spawned to compute the seam, which may have to reload the required images and recompute the corresponding buffers. Faces that need to communicate their seams should be processed next so that they are available for intersection resolution in the neighboring nodes as soon as possible. Finally, all the remaining faces should be processed. This order reduces idling time in the threads but its pattern of accessing images is not good for caching. For our implementation we use column major order for partitioning and processing the faces. This order processes the faces that need to send their branching points first but the faces that need to send their seams are processed last. This may introduce stalls in the neighboring nodes that have exhausted their local tasks and are waiting for the shared seams for more intersection resolution tasks. But overall, this order seems to provide a good balance between caching performance and minimizing idle time.

Task priorities are also used for good caching performance and to minimize idling times. Tasks that compute skipped seams have the highest priority since the task that skipped the corresponding seam's computation may have run recently and the required buffers might still be in the cache. The seams generated by these tasks are also required by neighboring nodes for intersection resolution tasks, so prioritizing them decreases latency for the seam messages. The branching point and seams computing tasks have the next priority since their results are required for further computations. The intersection resolution tasks have the lowest priority since their results are final and not required for further computations.

5 Results

We have conducted tests to compare our technique with a sweeping window implementation of Graph Cuts [8] with additional strong scaling tests of our multithreaded and hybrid implementations. The following datasets were used for testing:

- **SLC.** $122,625 \times 26,632$, 3.27 gigapixels panorama composed of 624 individual images acquired in a 48×13 grid. Figure 9, top.
- **Lake Louise.** $187,068 \times 40,201$, 7.52 gigapixels panorama composed of 1512 individual images acquired in a 72×21 grid. Figure 9, center.
- **Campus.** $294,040 \times 109,080$, 32.07 gigapixels panorama composed of 5500 individual images acquired in a 100×55 grid. Figure 9, bottom.

Fig. 9. Seams computed on the three different datasets. Top, SLC panorama, 3.27 gigapixels with 624 images. Computed in 3.76 minutes using 24 cores. Center, Lake Louise panorama, 7.52 gigapixels with 1512 images. Computed in 26.72 seconds using 1024 cores. (courtesy of City Escapes Nature Photography). Bottom, Campus panorama, 32.07 gigapixels with 5500 images. Computed in 60.45 seconds using 1024 cores.

SLC			
	Time (min.)	Max MB	Total Energy
GC1	1533.97	3324	1.070×10^8
GC3	318.21	867	1.119×10^8
SS	64.6	290	1.036×10^8

Lake Louise			
	Time (min.)	Max MB	Total Energy
GC1	8037.82	4022	2.927×10^8
GC3	1029.88	1067	3.279×10^8
SS	266.64	382	2.841×10^8

Fig. 10. Results of our single threaded out-of-core implementation (SS) and the two sliding-window based out-of-core Graph Cuts implementations—Full resolution version (GC1) and Hierarchical version with 3 levels (GC3). GC1 is too slow to be practical. GC3 is faster and uses lesser memory but produces poorer results. Our method is much faster than GC3 and uses much less memory while producing seams that are better than even GC1.

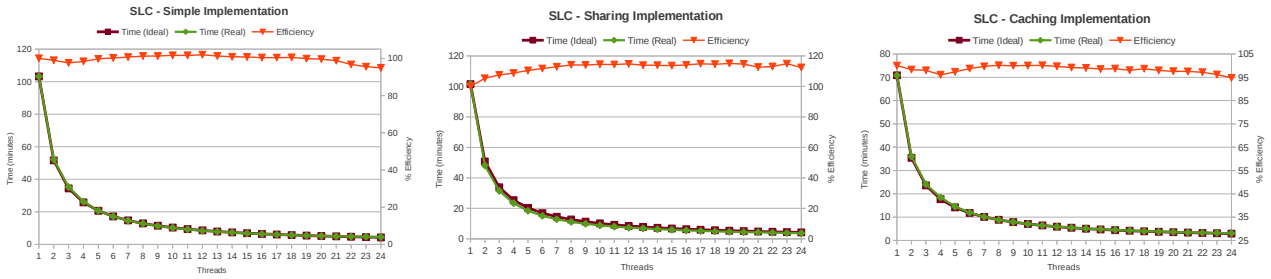
The results show that our sequential implementation is much faster, uses much less memory and produces seams with energy comparable to Graph Cuts (5.1). Section 5.2 shows that the multithreaded implementations scale almost linearly, even up to the full machine, while still maintaining the memory usage per thread. The performance is further improved with the use of caching at the cost of increased memory usage. Section 5.3 shows that the hybrid implementation also has good scalability, even up to 1024 processors.

5.1 Comparison with Graph Cuts

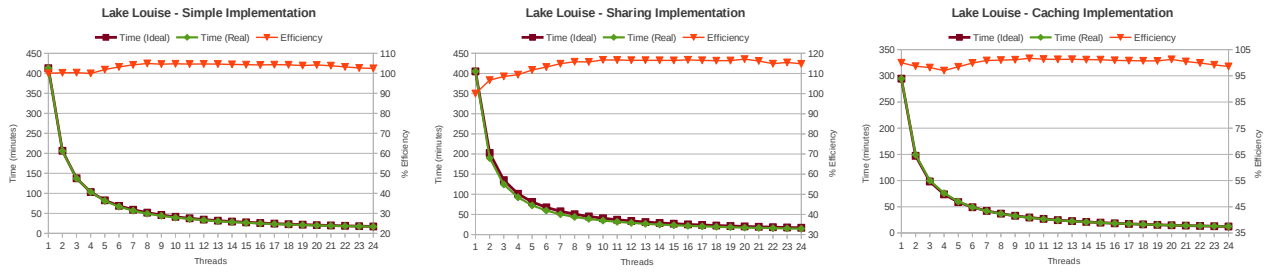
As mentioned in Section 2, the out-of-core, sweeping Graph Cuts window [8] approach is the only minimal seam approach that has been shown to work with gigapixel sized panoramas and therefore is used for our comparison. We also extended the implementation by adding support for hierarchical solving. For each image, it and its overlapping images are loaded and the energy function is computed over the domain of the image. The Graph Cuts algorithm is then applied on the energy for the solution of the window. Overlapping portions from solutions of previous windows are locked so that the seams are consistent across window boundaries. For hierarchical implementation an image pyramid with the specified number of levels is created per each window. Graph Cuts is first applied to the energy computed at the deepest level. The solution is then upsampled to the next level using bilateral upsampling. A dilation is applied on the upsampled seams and Graph Cuts is again performed on the dilated region. This step is repeated till the top level of the pyramid. For Graph Cuts we use the widely used implementation provided by [1], [2], [3], [28].

Each alpha-expansion of Graph Cuts can be possibly parallelized for non-overlaping image neighborhoods. Since this is a local approach, only the image being processed and the overlapping regions of its neighbors need to be kept in memory. In our work, we have found no research or publications on the scalability and efficiency of such an approach for out-of-core or distributed settings so direct comparison is difficult. Although, when compared to our technique this approach has some critical drawbacks. For each alpha-expansion step, the results of the overlapping regions need to be communicated with an image's neighbors. This requires the transfer of an image buffer. Synchronization would also be required after each expansion step and each iteration. Moreover, there may be multiple, possibly many, iterations to produce an acceptable result. In contrast, our technique only needs to communicate points and/or polylines and only requires a single pass. Graph Cuts communication would be quite expensive on large distributed systems and be a critical bottleneck in high-latency, low-bandwidth, distributed systems now seen in cloud computing infrastructures. Finally, like the initial Panorama Weaving technique, the scalability of this algorithm is limited by the amount of solution buffers that can be held in memory. Schemes, similar to what we have presented in this work, would be required for this technique to scale efficiently.

We tested the hierarchical, sweeping-window Graph Cuts based implementation on the two smaller datasets, SLC and Lake Louise, with two different configurations—one running the solver at full resolution and the other with 3 levels of hierarchy. We compared the running times, memory usage and total



SLC												
	Direct				Shared				Cached (8 GB)			
Threads	Ideal	Actual	Eff. %	MB	Ideal	Actual	Eff. %	MB	Ideal	Actual	Eff. %	MB
1	103.26	103.26	100.00	290	101.53	101.53	100.00	290	70.86	70.86	100.00	8294
4	25.82	26.23	98.41	784	25.38	23.34	108.75	769	17.72	18.44	96.08	8323
8	12.91	12.75	101.26	1412	12.69	11.12	114.17	1306	8.86	8.85	100.13	8323
12	8.61	8.44	101.99	2207	8.46	7.37	114.79	1869	5.91	5.92	99.70	8383
16	6.45	6.43	100.43	2636	6.35	5.56	114.10	2179	4.43	4.49	98.69	8386
20	5.16	5.18	99.63	3280	5.08	4.43	114.70	2624	3.54	3.63	97.56	8403
24	4.30	4.53	94.93	4018	4.23	3.76	112.41	2991	2.95	3.12	94.72	8415



Lake Louise												
	Direct				Shared				Cached (8 GB)			
Threads	Ideal	Actual	Eff. %	MB	Ideal	Actual	Eff. %	MB	Ideal	Actual	Eff. %	MB
1	412.85	412.85	100.00	382	405.10	405.10	100.00	382	294.62	294.62	100.00	8304
4	103.21	103.26	99.95	1129	101.27	92.60	109.37	1052	73.65	75.92	97.01	8393
8	51.61	49.19	104.91	2099	50.64	43.73	115.79	1891	36.83	36.46	101.01	8413
12	34.40	32.86	104.69	3079	33.76	28.97	116.52	2536	24.55	24.24	101.27	8481
16	25.80	24.77	104.16	3953	25.32	21.69	116.71	3471	18.41	18.25	100.90	8526
20	20.64	19.82	104.17	4860	20.25	17.30	117.07	4395	14.73	14.55	101.28	8861
24	17.20	16.79	102.46	5820	16.88	14.71	114.76	4750	12.28	12.45	98.58	8599

Fig. 11. Scaling results of our three parallel implementations on the SLC (top) and Lake Louise (bottom) datasets (All timings are in minutes). Due to space limitations the tables only show data for core counts in multiples of 4. The plots show all the data points. All implementations show good efficiency (Eff. %) throughout, with the shared implementation achieving super-linear speedup due to data sharing. The maximum memory usage (MB) of the direct and shared implementations do increase with the number of threads but memory/thread is always maintained bellow the usage of single threaded runs. The cached implementation can be configured with a cache size (8 GB in this case) that allows it to scale to systems with larger memory. It provides the best performance among all three implementations by further reducing disk I/O and computations.

energy of the results with the single threaded, out-of-core implementation of our technique. The tests were run on a system with a quad core Intel Core i7 920 CPU @ 2.67 GHz and 6 GB of memory.

The results are detailed in Figure 10. Even though the out-of-core Graph Cuts implementation can handle gigapixel sized panoramas, it is too slow for practical purposes. The hierarchical solver is much faster and requires lesser memory at the cost of a lower quality solution. In contrast, our system is much faster, uses much less memory and the final energy computed is lower than the full resolution Graph Cuts implementation.

5.2 Multi-Threaded Scalability Tests

We have performed scalability tests on our three multi-threaded, out-of-core implementations. The system for these tests was an Intel Xeon based workstation with 4 Intel Xeon E7540 CPUs @ 2.00 GHz, having 6 physical cores and 12 HT threads each and 128 GB of RAM. The tests were run on the SLC and Lake Louise datasets. Figure 11 shows the performance results.

The *direct* threaded implementation is fast and scalable and exhibits near linear scaling throughout. Even for up to the maximum number of physical cores the efficiency is maintained around 95% for SLC dataset.

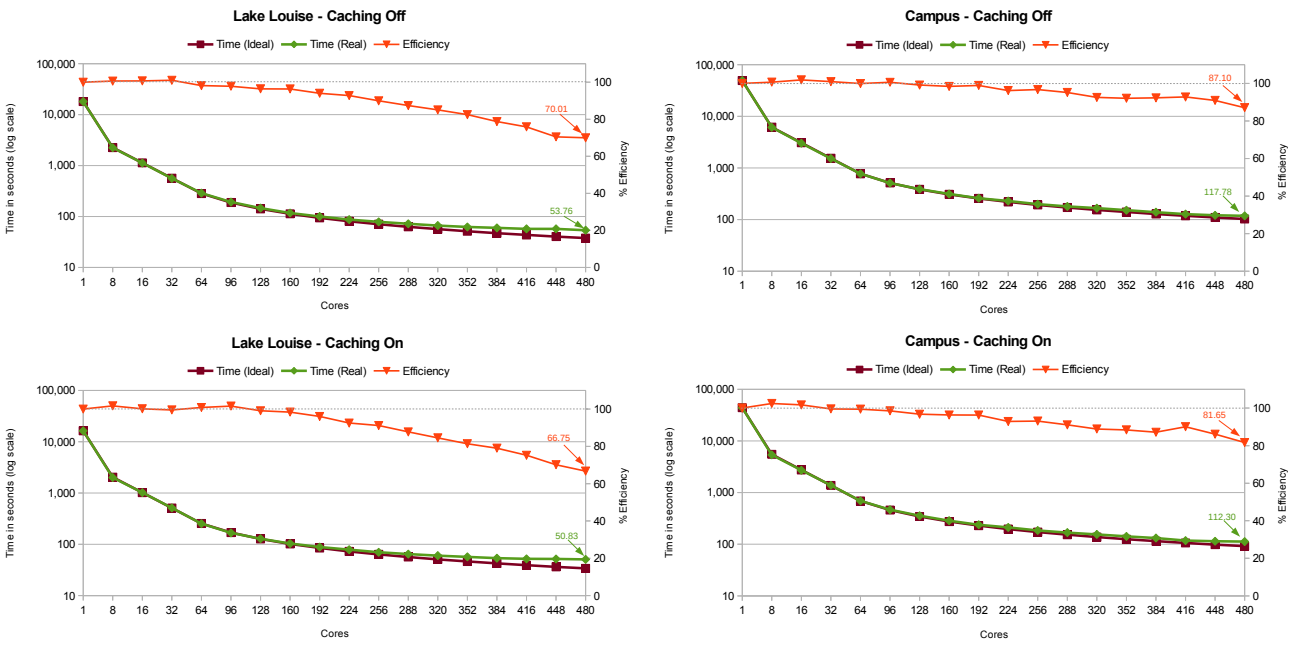


Fig. 12. Scaling results on the NVIDIA cluster for the Lake Louise (left) and Campus (right) datasets. Time in seconds and percent efficiency are shown for caching disabled and enabled runs. The tests were run on up to 60 nodes with 8 threads/node for a total of 480 threads. The cache size was configured to be 16 GB/node for the caching enabled runs. The Campus dataset scales better as it has more faces that keep the threads busy without idling. For both datasets the performance is better with caching enabled but the efficiency drops on higher nodes because caching is not as effective due to each node having to process fewer number of faces.

The slight reduction in efficiency at 24 threads is due to the smaller size of the panorama and it is not seen in the larger Lake Louise dataset, which is maintained at 99% efficiency. From the table, it can be seen that the maximum memory usage increases with the number of threads for the direct and shared implementations, but memory/thread is always maintained bellow the usage of single threaded runs.

The *sharing* implementation shows an overall improvement in performance compared to the *direct* implementation. By sharing the buffers among the threads it reduces their disk I/O and recomputations. This results in a super-linear speed-up as shown in the graph with efficiency reaching as high as 114% for SLC and 117% for Lake Louise datasets. The data sharing also reduces the maximum memory usage of this implementation. As shown in the table, for SLC dataset it only requires 3.0 GB of memory with 24 threads compared to 4.0 GB for the *direct* implementation. Similarly for Lake Louise dataset it only requires 4.8 GB of memory with 24 threads compared to 5.8 GB for *direct*.

For the *caching* implementation, the tests were configured to use 8 GB of cache which is a reasonable size for modern workstations. With this implementation we are able to further improve the performance while still maintaining its scalability. As the table shows, the efficiency doesn't go bellow 94% for SLC and 98% for Lake Louise datasets. The maximum memory usage is maintained around the configured 8 GB size.

5.3 Hybrid Scalability Tests

We performed scalability tests for the hybrid implementation on three different systems. Two of the larger datasets—Lake Louise and Campus were used for these tests. Two sets of tests were run with caching enabled and disabled. The cache sizes were configured based on the amount of memory available in the corresponding systems. All three systems are shared machines. To discount the effects of other jobs on the shared resources such as the file system, several runs were made for each configuration and the best times were recorded.

The first machine is the *NVIDIA* cluster. Each node of the cluster contains two quad core Xeon X5550 processors @ 2.67 GHz and 24 GB of RAM. The tests were run on up to 60 nodes (480 cores). For runs with caching enabled, the cache size was configured to 2 GB per thread. The conservative size of 16 GB, of the avialable 24 GB, was chosen because some amount of memory is used up by the operating system and system buffers and we wanted to avoid swapping to disk which would have been detrimental to the performance. The results from our tests are shown in Figure 12. Note that for the first few core counts we get more than 100% efficiency. This is due to the sharing of the buffers among the threads within a node which reduces some disk I/O and computations.

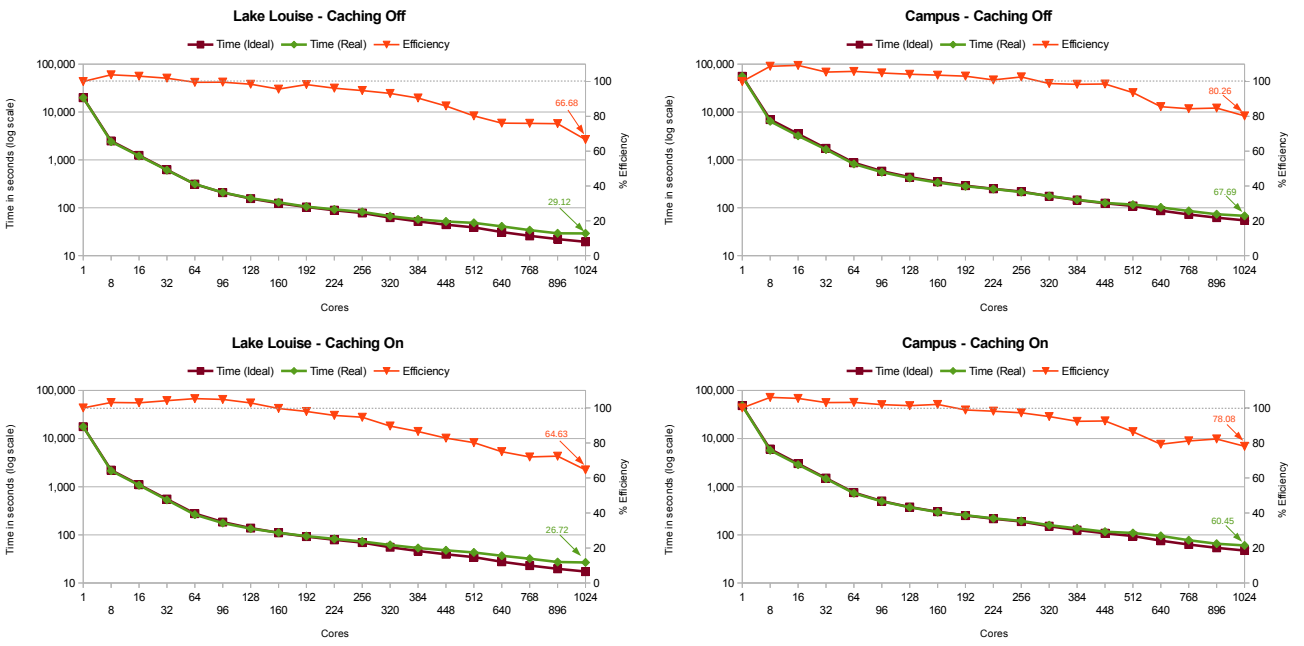


Fig. 13. Scaling results on the Longhorn cluster for the Lake Louise (left) and Campus (right) datasets. Time in seconds and percent efficiency are shown for caching disabled and enabled runs. The tests were run on up to 128 nodes with 8 threads/node for a total of 1024 threads. The cache size was configured to be 32 GB/node for caching enabled runs. Our system scales much better on this cluster compared to the NVIDIA cluster due to its use of the high performance Lustre file system. Similar to the NVIDIA cluster, the bigger dataset, Campus, scales better than Lake Louise dataset and the caching version doesn't scale as well as the non caching version though it is faster.

From the plots, we can see that the system scales quite well with efficiency around 70% for the Lake Louise dataset on 480 cores with caching disabled. The efficiency is much better for the larger Campus dataset, around 87% on 480 cores without caching. This is because there are more faces to be processed which reduces the effect of load imbalance, and the chance of threads idling waiting for tasks, on higher number of nodes. The results for the runs with caching enabled, show that the performance is consistently better than the caching disabled runs, but it doesn't scale as well. This is because the effectiveness of the per node cache reduces with increasing number of nodes since they have to process fewer faces. Still we can see good scaling performance with the efficiency being around 66% for the Lake Louise dataset and 81% for the Campus dataset on 480 cores.

The second system is the *Longhorn* cluster, with nodes containing 8 Nehalem cores @ 2.5 GHz and at least 48 GB of RAM. For cache enabled runs, a conservative cache size of 4 GB per thread or 32 GB per node was chosen. The scaling tests were run on up to 128 nodes or 1024 cores. The results are given in Figure 13. Similar to the NVIDIA cluster we can see that the scaling is better for the larger Campus dataset than the Lake Louise dataset. The efficiency of the caching enabled runs are again lower than the runs with caching disabled for higher number of cores but they are still consistently faster. Overall, our system scales better on the Longhorn cluster than the NVIDIA cluster with the efficiency on 1024 cores for the Lake Louise dataset with caching enabled and disabled being 66.23% and 66.68% respectively, and 80.26% and 78.08% for the Campus dataset. The scalability is better on Longhorn primarily due to its faster *Lustre* file system compared to the NFS based file system on the NVIDIA cluster.

The third system is the *SGI UV 1000* which has 264 Xeon X7542 cores @ 2.67 GHz and 2.8 TB of RAM. The UV is a distributed shared memory ccNUMA system where a single process can address the entire memory of the system. It consists of 22 blades, each containing two six-core processors and 128 GB of RAM. The blades are connected to each other using the SGI NUMAlink 5 interconnect. Although it is possible to run just one process with 264 threads using shared memory, the NUMA nature of the system means that it would not scale very well. For good efficiency on higher number of cores it is important to use shared memory only among the cores within a blade and use message passing between the blades. Our hybrid implementation maps nicely to this problem. Our implementation uses SGI's MPT library for MPI calls on this system. To get good scaling it is important that each thread that is spawned is allocated a unique physical core, all the threads of a single process are

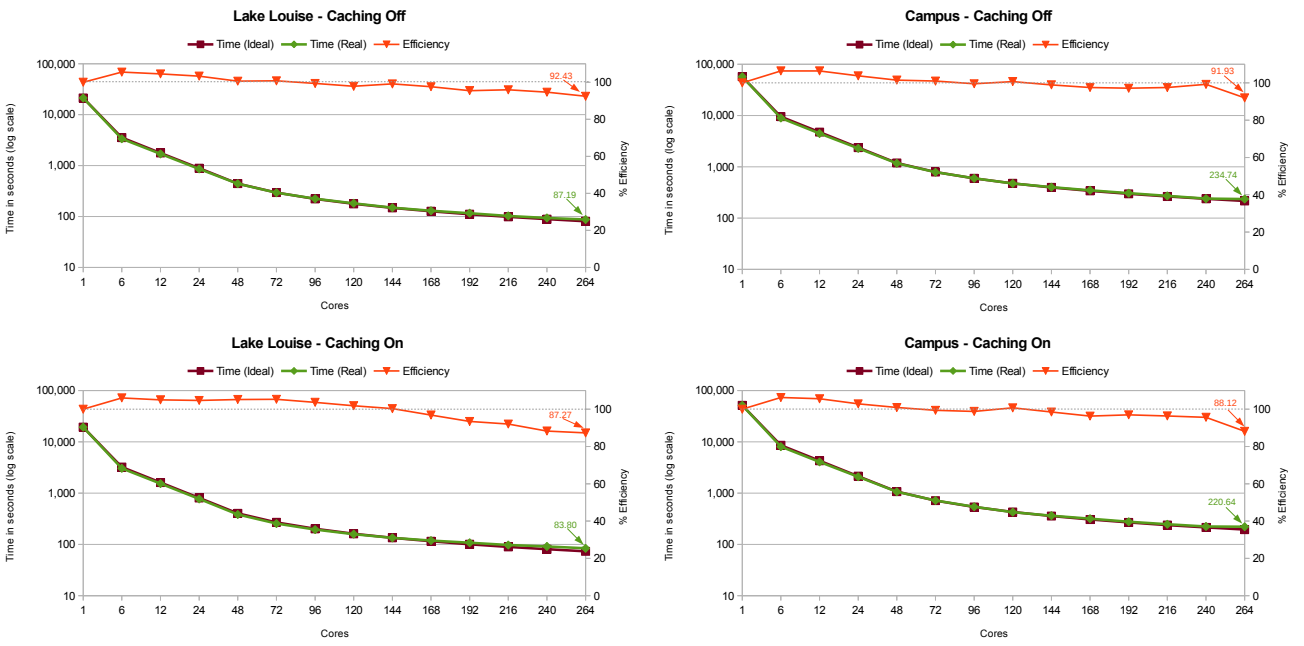


Fig. 14. Scaling results on the SGI UV 1000 system for the Lake Louise (left) and Campus (right) datasets. Time in seconds and percent efficiency are shown for caching disabled and enabled runs. One process was assigned to one physical processor with six threads per process mapped to each of its six cores. The tests were run with up to 44 processes and 6 threads/process for a total of 264 threads. The cache size was configured to be 36 GB/process. The drop in efficiency at 264 cores (full machine) is due to some resources being used by the operating system and other services.

assigned to cores in the same blade and the assigned physical memory also resides on the same blade. The MPT library can be configured for this using some environment variables as described in [29]. The scaling results are given in Figure 14. One process was spawned for each physical CPU and each of them spawned six threads, one each for the six cores of the CPU. For the caching enabled runs the cache size was configured to 36 GB per process. The plots show that we are getting good scaling with the efficiency being above 90% even for the full machine (264 cores) for both data sets with caching turned off. Similar to the other machines the efficiency of the caching enabled runs drop on higher core counts (87.27% for Lake Louise and 88.12% for Campus) due to the diminishing value caching provides with higher number of processes. The drop in efficiency on the full machine (264 cores) for both types of runs is due to some of the cores being assigned to the OS and other services running on the system.

6 Conclusion

In this work, we have presented a technique for computing seams for gigapixel sized panoramas that is fast, light and scalable. On resource constrained system it is able to run in an out-of-core fashion using very little memory and still produce a solution orders of magnitude faster than the previous state-of-the-art: a moving window Graph Cuts scheme. On multi-core systems it can run in parallel and achieve super-linear speed-ups by sharing data among the threads and reducing redundant disk I/O and computation. On systems with higher memory resources, it can utilize some of it as a cache to further reduce disk I/O and computations for improved performance. On clusters and distributed memory systems, it utilizes hybrid multithreading/message-passing techniques to scale to hundreds of cores. Moreover the energy of the seams produced by our technique is comparable, if not better, to the previous work.

Panorama Weaving allows for interactive editing of seams. For future work, we plan to extend our technique to allow local editing of seams for large panoramas in a scalable and interactive manner.

Acknowledgments

References

[1] Y. Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 23, no. 11, Nov. 2001.

[2] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 26, no. 9, pp. 1124–1137, 2004.

[3] V. Kolmogorov and R. Zabih, "What energy functions can be minimized via graph cuts?" *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 26, no. 2, pp. 147–159, 2004.

[4] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, "Graph-cut textures: Image and video synthesis using graph cuts," *ACM Trans. Graph*, vol. 22, no. 3, pp. 277–286, July 2003.

[5] A. Agarwala, M. Dontcheva, M. Agrawala, S. M. Drucker, A. Colburn, B. Curless, D. Salesin, and M. F. Cohen, "Interactive digital photomontage," *ACM Trans. Graph*, vol. 23, no. 3, pp. 294–302, 2004.

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

[7] A. Agarwala, K. C. Zheng, C. Pal, M. Agrawala, M. F. Cohen, B. Curless, D. Salesin, and R. Szeliski, "Panoramic video textures," *ACM TOG*, vol. 24, no. 3, pp. 821–827, 2005.

[8] J. Kopf, M. Uyttendaele, O. Deussen, and M. F. Cohen, "Capturing and viewing gigapixel images," *ACM Trans. Graph*, vol. 26, no. 3, p. 93, 2007.

[9] B. Summa, J. Tierny, and V. Pascucci, "Panorama weaving: fast and flexible seam processing," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 83:1–83:11, Jul. 2012.

[10] R. Szeliski, "Image alignment and stitching: A tutorial," *Foundations and Trends in Computer Graphics and Vision*, vol. 2, no. 1, 2006.

[11] P. Pérez, M. Gangnet, and A. Blake, "Poisson image editing," *ACM Trans. Graph*, vol. 22, no. 3, pp. 313–318, 2003.

[12] A. Levin, A. Zomet, S. Peleg, and Y. Weiss, "Seamless image stitching in the gradient domain," in *ECCV 2004*.

[13] D. L. Milgram, "Computer methods for creating photomosaics," *IEEE Trans. Computer*, vol. 23, pp. 1113–1119, 1975.

[14] ——, "Adaptive techniques for photomosaicking," *IEEE Trans. Computer*, vol. 26, pp. 1175–1180, 1977.

[15] R. S. Szeliski, "Video mosaics for virtual environments," *IEEE Computer Graphics and Applications*, vol. 16, no. 2, Mar. 1996.

[16] H. Y. Shum and R. S. Szeliski, "Construction and refinement of panoramic mosaics with global and local alignment," in *ICCV*, 1998, pp. 953–956.

[17] J. E. Davis, "Mosaics of scenes with moving objects," in *CVPR*, 1998, pp. 354–360.

[18] M. T. Uyttendaele, A. Eden, and R. S. Szeliski, "Eliminating ghosting and exposure artifacts in image mosaics," pp. II:509–516, 2001.

[19] N. R. E. Gracias, M. H. Mahoor, S. Negahdaripour, and A. C. R. Gleason, "Fast image blending using watersheds and graph cuts," *Image and Vision Computing*, vol. 27, no. 5, pp. 597–607, Apr. 2009.

[20] A. A. Efros and W. T. Freeman, "Image quilting for texture synthesis and transfer," in *SIGGRAPH*, 2001.

[21] J. Liu and J. Sun, "Parallel graph-cuts by adaptive bottom-up merging," in *CVPR*. IEEE, 2010, pp. 2181–2188.

[22] O. Jamriska, D. Sykora, and A. Hornung, "Cache-efficient graph cuts on structured grids," in *2012 CVPR*.

[23] A. Delong and Y. Boykov, "A scalable graph-cut algorithm for N-D grids," in *CVPR*. IEEE, 2008.

[24] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *Computer Vision on GPU*, 2008, pp. 1–8.

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

[26] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[27] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *J. Parallel Distrib. Comput.*, vol. 64, no. 8, pp. 974–996, Aug. 2004.

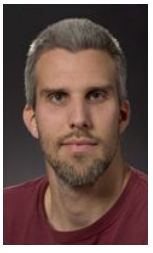
[28] A. Delong, A. Osokin, H. Isack, and Y. Boykov, "Fast approximate energy minimization with label costs," in *2010 IEEE Conference on CVPR*, june 2010, pp. 2173 –2180.

[29] SGI, *Message Passing Toolkit (MPT) User Guide*.

Sujin Philip is a graduate student at the Scientific Computing and Imaging Institute, currently pursuing his Masters degree in computer science at the School of Computing, University of Utah. He got his Bachelor's degree from the University of Pune, India, in 2006. His interests include hybrid and heterogeneous parallel computing, out-of-core processing, computer graphics and visualization.

Brian Summa is a postdoctoral researcher at Scientific Computing and Imaging Institute at the University of Utah. He received his BSE and MSE in Computer Science from the University of Pennsylvania in 2008 and his Ph.D. in Computer Science from the University of Utah in 2012. His interests include large scale data and image processing, scientific visualization, and computer graphics.

Julien Tierny received the Ph.D. degree in Computer Science from Lille 1 University in October 2008. He is currently a CNRS permanent researcher, affiliated with Sorbonne Universities (LIP6 UPMC, Paris France) since September 2014 and with Telecom ParisTech from 2010 to 2014. Prior to his CNRS tenure, he held a Fulbright fellowship (U.S. Department of State) and was a post- doctoral research associate at the Scientific Computing and Imaging Institute at the University of Utah. His research interests include topological and geometrical data analysis for scientific visualization and graphics.



Peer-Timo Bremer is a computer scientist and project leader at the Center for Applied Scientific Computing at the Lawrence Livermore National Laboratory (LLNL) and Associated Director for Research at the Center for Extreme Data Management, Analysis, and Visualization at the University of Utah. Prior to his tenure at CASC, he was a postdoctoral research associate at the University of Illinois, Urbana-Champaign. Peer-Timo earned a Ph.D. in Computer science at the University of California, Davis in 2004 and a Diploma in Mathematics and Computer Science from the Leipniz University in Hannover, Germany in 2000. He is a member of the IEEE Computer Society.

Valerio Pascucci received the EE laurea (master) degree from the University La Sapienza in Rome, Italy, in December 1993, as a member of the Geometric Computing Group, and the Ph.D. degree in computer science from Purdue University in May 2000. He is a faculty member at the Scientific Computing and Imaging (SCI) Institute at the University of Utah. Before joining SCI, he served as a project leader at the Lawrence Livermore National Laboratory (LLNL), Center for Applied Scientific Computing (from May 2000) and as an adjunct professor at the Computer Science Department of the University of California, Davis (from July 2005). Prior to his tenure at CASC, he was a senior research associate at the University of Texas at Austin, Center for Computational Visualization, CS, and TICAM Departments.