Chapter 6
Optimization of Tile Creation

The algorithms for creating tile sets presented in the previous chapter represent basic approaches. There are many possible optimizations to make the process more efficient. Some geospatial image sets are small enough that these optimizations are not needed. However, very large image sets will almost always require optimization to make their computation a tractable problem. In this chapter we will present algorithms for the following tiling optimizations:

- Caching tile sets in memory to improve performance
- Partial reading of source images to conserve memory
- Multi-threading of tile creation algorithms
- Tile creation algorithms for distributed computing
- Partial updating of existing tiled image sets

Each of these techniques should be thoroughly considered by those developing a tile creation system. They have been reduced to practice and are essential improvements in an otherwise resource-inefficient process.

6.1 Caching Tile Sets in Memory to Improve Performance

Because reading from and writing to disk are often the most time consuming steps in the tile creation process, we will present an optimized algorithm that minimizes both of these. In the previous chapter we presented two approaches to tile creation: pull-based and push-based. Push-based had the advantage of having to read source images only one time. Pull-based allowed us to write tiled images only one time. We decided to use the pull-based system because, with the addition of a source image cache, we could reduce some of the re-reading of source images. We are still left with the problem of having to re-read the tiled images as we create the lower resolution zoom levels.

However, if our computer system has enough memory to hold all (or a significant subset) or our tiled images, we can use a very efficient push-based approach. We

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_6,

can loop through the source images, read each one once and only once, and apply the data from the source images to our cached tiled images. Only when we have completed looping through the source images will we write our tiled images to memory. Furthermore, since our tiled images remain in memory, there is no need to re-read them when creating the lower resolution levels.

In practice, few systems will have enough RAM to hold a complete tile set, uncompressed, in memory. Therefore, we will need some logical scheme to sub-divide our tile sets into manageable pieces. Recall an earlier example of a high-resolution aerial imagery collection of the world's 50 largest cities. This dataset has a logical separation of tiles built into its structure. We could separately process, in memory, the tiles for each city and merge the results later. For source image sets without logical groupings, we would have to develop some method for geographically partitioning the tile sets. In the next chapter, we will discuss in some detail a general method for solving this problem. For the purposes of the current section assume that such a system is in place.

The algorithm for push-based tile creation with in memory tile cache is as follows:

1. Choose the base level for the tile set.
2. Determine the geographic bounds of the tile set. (This can be based on the bounds of the source images.)
3. Determine the bounds of the tile set in tile coordinates.
4. Initialize the tile cache.
5. Iterate over the source images. For each source image, do the following:
    a. Compute the bounds of the source image in tile coordinates.
    b. Read the source image into memory.
    c. Iterate over the tile set coordinates. For each tile do the following:
        i. Compute the geographic bounds of the tile.
        ii. Check the cache for the tile image. If it is not in the cache, create an empty image and put it in the cache.
        iii. Extract the required image data from the source image and store it in the tiled image.
6. For each level from (`base_level` - 1) to 1, do the following.
    a. Determine the bounds of the current tile level in tile coordinates.
    b. Iterate over the tile set coordinates. For each tile, do the following:
        i. Determine the four tiles from the higher level that contribute to the current tile.
        ii. Retrieve the four tile images from the cache or as many as exist.
        iii. Combine the four tile images into a single, scaled-down image.
        iv. Save the completed tiled image to the tile cache
7. Finalize the tile cache and store the images on disk

Before presenting the computer code for executing these steps, we will define the data types in Listing 6.1. TileCache represents the mechanism for holding tiled

Listing 6.1 TileCache class.

```
1  abstract class TileCache {
2
3      public abstract BufferedImage getTile(TileAddress ta);
4
5      public abstract void putTile(TileAddress ta, BufferedImage image);
6
7  }
```

images in memory. Additionally, we will make use of the data types defined in the previous chapter. Listing 6.6 shows the algorithm for creating tiles with a memory cache.

6.2 Partial Reading of Source Images

Each of the previously defined algorithms for tile creation assumed that source images can be read and held completely in memory. In some cases this is either not possible or not practical. Some image formats, like MrSID or JPEG2000, support very large images. It is not unusual to encounter images that are several gigabytes compressed.

Uncompressed versions could be 10 times the original size. Even if sufficient memory exists to hold the entire image, we may want to only process a part of the image. Therefore, we need to examine techniques for partial reading of images. Five logical methods for reading images are as follows:

- Whole Image: Only allows users to read entire images in one step.
- Scanlines: Allows users to read one or more scanlines in one step. This is the most common method for low-level access to image pixels.
- Tiles: Allows users to read tiled subsections of images. This is usually only available with image formats that natively store images in subdivided tiles. The reader should note that the concept of ”tile” in this context is slightly different from how we have used it throughout the book. In this context, tiles represent rectangular blocks of an image. They are not full images by themselves.
- Random Areas: Allows users to read user-defined rectangular areas of images.

The ability to read only a part of an image is dependent on both the file format used to store the image and the software library used to decode the image. The process is straightforward for uncompressed images. However, it may be impossible with compressed image formats. Java and Python support reading a variety of image formats. However, they do not allow partial reading of images. In general, the most flexible methods for reading images can be found in their C/C++ reference implementations. LIBJPEG, LIBPNG, and LIBTIFF are all open source libraries for reading JPEG, PNG, and TIFF images, respectively. Both LIBJPEG and LIBPNG support scanline based reading. LIBTIFF supports scanline and tile-based reading

depending on how the image was stored. The LizardTech GeoExpress Software Development Kit (SDK) supports random area reading of MrSID and JPEG2000 images.

Two things are needed to integrate partial reading into our existing tile creation algorithms. First, we need a method for reading random areas out of our image. This can be done by directly reading the random areas where supported or by adapting scanline or tile-based reading to provide random areas. Second, we need to adapt our tile creation algorithm to account for a partial image instead of the full image.

6.2.1 Reading Random Areas from Source Images

We define a random area as a rectangular region within an image. It is defined by the coordinates for the origin: x and y, and a width and height (See Figure 6.1). Before we can extract the image data from the random area we have to determine the geographic bounds of the intersection between our source image and our tiled image. We then have to convert the geographic bounds of the intersection area into source image coordinates.

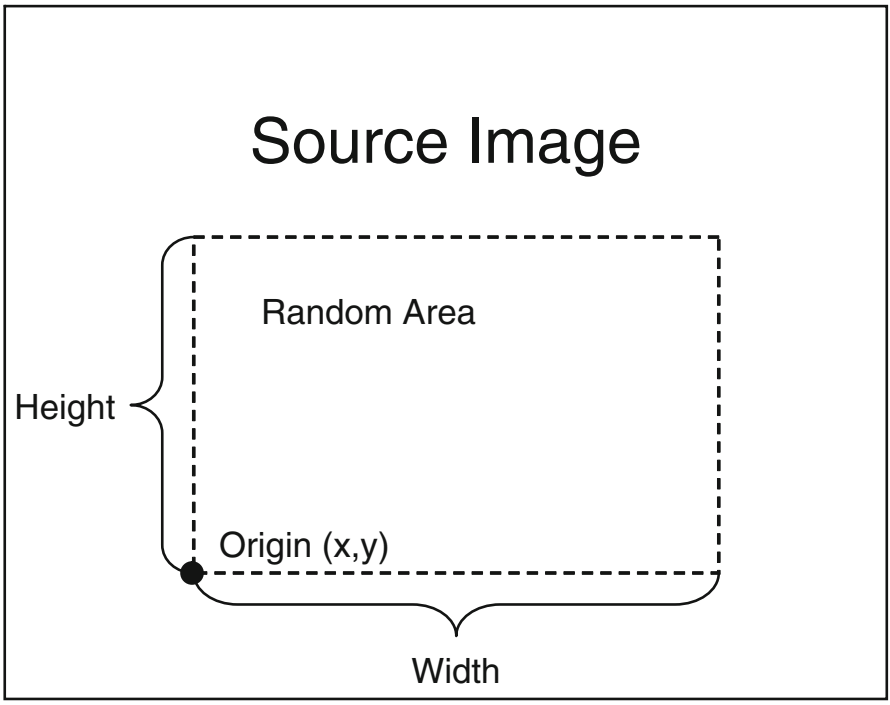

Fig. 6.1 Random area from a source image.

Listing 6.2 Compute the intersection of two bounding rectangles.

```
public BoundingBox getIntersection(BoundingBox bb1, BoundingBox bb2) {
  if (!bb1.intersects(bb2.minx, bb2.miny, bb2.maxx, bb2.maxy)) {
    return null;
  }
  double minx = Math.max(bb1.minx, bb2.minx);
  double miny = Math.max(bb1.miny, bb2.miny);
  double maxx = Math.min(bb1.maxx, bb2.maxx);
  double maxy = Math.min(bb1.maxy, bb2.maxy);
  BoundingBox out = new BoundingBox(minx, miny, maxx, maxy);
  return out;
}
```

Listing 6.3 Convert geographic bounds to image bounds.

```
public Rectangle convertCoordinates(BoundingBox imageBounds, BoundingBox
    subImageBounds, int imageWidth, int imageHeight) {

    int x = (int) Math.round((imageBounds.minx - subImageBounds.minx) / (
        imageBounds.maxx - imageBounds.minx) * imageWidth);
    int y = imageHeight - (int) Math.round((imageBounds.miny - subImageBounds.
        miny) / (imageBounds.maxy - imageBounds.miny) * imageHeight) - 1;
    int width = (int) Math.round((subImageBounds.maxx - subImageBounds.minx) /
        (imageBounds.maxx - imageBounds.minx) * imageWidth);
    int height = (int) Math.round((subImageBounds.maxy - subImageBounds.miny) /
        (imageBounds.maxy - imageBounds.miny) * imageHeight);
    Rectangle r = new Rectangle(x, y, width, height);
    return r;
}

class Rectangle {
  public Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
  }

  int x;
  int y;
  int width;
  int height;
}
```

The algorithms shown in Listings 6.2 and 6.3 compute the intersection of two bounding rectangles and convert that intersection from geographic coordinates to image coordinates.

The result of Listing 6.3 is a rectangle in image coordinates. These coordinates can be used to extract a region of pixels from a source image. The algorithms in Listings 6.7 and 6.8 demonstrate how to extract a partial image region with either scanline or tile-based access to a tiled image. Because we are demonstrating low-level access to image data, we will not use a memory image object like Java's BufferedImage. To represent in-memory images, we will use a simple array of "byte" values that represent RGB pixel values packed in 3 byte triplets and stored in row-major

order. Also, the reader should note that the algorithms presented in the following section are written for maximum clarity, not necessarily efficiency. For example, we use ”for” loops to copy blocks of bytes while many programming languages have built-in functions that perform this step much faster.

Listing 6.7 presents Java code for reading an image region with scanline based access. The steps for scanline reading of an image region are as follows:

1. Skip or seek to the first scanline needed.
2. For each scanline needed, do the following:
   a. Read the entire scanline into a temporary buffer.
   b. Copy the required subsection of the scanline into the final image buffer.
3. Return the final image buffer.

The algorithm uses the class ”ImagePointer” to represent a handle on a file or input stream with encoded pixel data. The algorithm assumes the following functions are available:

- ReadScanline: This function decodes a scanline of pixel data and copies it to the provided buffer. After completion, it positions the image pointer for reading the next available scanline.
- SkipScanlines: This function skips scanlines in the image and positions the image point for reading at the next available scanline. Some image decoding implementations allow random access to scanlines, while others will have to decode the scanlines that are skipped. This difference may affect performance and should be considered by developers.

Listing 6.8 presents Java code for reading partial image regions with tile-based image access. The algorithm assumes that our decoding implementation allows random access to image tiles; this is modeled after LIBTIFF’s access routines that do allow random access to image tiles for images that are stored in a certain way.

The steps for tile-based reading of an image region are as follows:

1. Determine the range of tiles that will need to be read.
2. Construct a temporary buffer with sufficient size to hold all of the needed tiles.
3. Iterate over the tiles, in row-major order. For each tile needed, do the following:
   a. Position the image pointer to read at the needed tile.
   b. Read the tile into the temporary buffer.
4. Trim the temporary buffer to match the desired region.

The algorithm also uses the class ”ImagePointer” to represent a handle on a file or input stream with encoded pixel data. The algorithm assumes that the following functions are available:

- SeekToTile: This function positions the image pointer to read the indicated tile.
- ReadTile: This function reads pixels from the current tile into the provided buffer.

6.2.2 Tile Creation with Partial Source Image Reading

Listing 6.9 shows our previous algorithm for creating tiles in an adapted form to handle partial reading of source images. The steps in this algorithm are as follows:

1. Chose the base level for the tile set.
2. Determine the geographic bounds of the tile set. (This can be based on the bounds of the source images.)
3. Determine the bounds of the tile set in tile coordinates.
4. Initialize the tile storage mechanism.
5. Iterate over the tile set coordinates. For each tile, do the following:
   a. Compute the geographic bounds of the specific tile.
   b. Iterate over the source images. For each source image do the following:
      i. Determine if the specific source image intersects the tile being created.
      ii. If the source image and tile intersect,
         A. Determine the intersection of tile and source image.
         B. Convert the intersection from geographic to image coordinates.
         C. Read the partial image data.
         D. Convert the partial image data to a BufferedImage.
         E. Draw the converted pixels to the tile image.
   c. Save the completed tiled image to the tile storage mechanism.
6. Finalize the tile storage mechanism.

Within Listing 6.9, the abstract method ”ReadPartialImage” is meant as a placeholder for the implementation specific techniques presented in the previous section. The abstract method ”ConvertBytes” simply moves the pixel data from the byte array into a Java BufferedImage. Listing 6.9 is a modified form of pull-based tile creation, which should be used if all image tiles cannot be held in memory at the same time. Push-based methods are still preferred if all tiles can be held in memory.

6.3 Tile Creation with Parallel Computing

Parallel computing is the use of multiple computing resources at the same time to execute a given task. This can be realized by a using a group of computer systems or by using a single computer system with multiple CPUs. In most cases, the tile creation process must be parallelized to operate efficiently. The next two sections present techniques for parallelization of the tile creation process from two very different perspectives.

Listing 6.4 Synchronized drawing.

```
1 synchronized (tileImage) {
2       drawImageToImage(bi, currentBounds, tileImage, tileBounds);
3 }
```

6.3.1 Multi-Threading of Tile Creation Algorithms

Multi-threading is a programming technique that, if supported by the underlying operating system and computer hardware, allows multiple tasks to execute at the same time within the same process. A process is an instance of computer program that is being executed. The requirement that the multiple threads of execution exist within the same process is a critical constraint. This allows the multiple threads to share memory with each other. For a detailed tutorial on threading models and usage, the reader is encouraged to see [2].

Multi-threading has three common uses. First, multi-threading is used to manage blocking input/output (I/O). Reading and writing from disk or network is relatively slow, and multi-threading allows the program to perform other tasks while waiting for I/O. A second use is to allow systems with multiple CPUs to process multiple tasks at the same time. The final common use of multi-threading is to make programs with graphical user interfaces more responsive. Multi-threading is useful for this because one thread can be dedicated to updating the graphical interface, while others perform the program's work. The first two of these uses, managing blocking I/O and processing multiple tasks, will be very useful in the tile creation process.

Our first algorithm is derived from our previous push-based tile creation algorithms but adds multi-threading to reduce waiting for I/O. In this algorithm, we have two threads: a reader thread and a tiler thread. The reader thread reads a source image into memory and waits for it to be retrieved by the tiler thread. The tiler thread retrieves decoded source images from the reader thread and creates tiles from it. When the tiler thread takes an image from the reader thread, the reader thread decodes another image and waits for it to be taken. Java code for this process is provided by Listing 6.10. This algorithm should result in a performance improvement, even on systems with just one CPU.

If our processing system has multiple CPUs, and current commodity systems can have up to 48, we can use more than one thread to perform the tiling. This requires two adjustments to the previous algorithm. First, we must create and start more than one tiler thread. Second, we need to make sure that multiple tiling threads are not accessing and writing to the same tile at the same time. To accomplish this, we will change how we call the method for drawing from source image to buffered images. We can wrap the call to the "drawImageToImage" method in a synchronized block that is synchronized on the target BufferedImage, see Listing 6.4. We need to synchronize on only tileImage because that is the only thing getting changed by the various threads. Listing 6.5 shows the method for starting and controlling multiple tiling threads.

Listing 6.5 Controlling multiple tile creation threads.

```
1   public void createTilesMultipleThreads(TileCache cache, SourceImage[]
        sourceImages, int baseLevel, int numberOfThreads) {
2       ReaderThread reader = new ReaderThread(sourceImages);
3       reader.start();
4       TilerThread[] tilerThreads = new TilerThread[numberOfThreads];
5       for (int i = 0; i < tilerThreads.length; i++) {
6           TilerThread tiler = new TilerThread(cache, baseLevel, reader);
7           tiler.start();
8           tilerThreads[i] = tiler;
9       }
10      for (int i = 0; i < tilerThreads.length; i++) {
11          try {
12              tilerThreads[i].join();
13          } catch (InterruptedException e) {
14              e.printStackTrace();
15          }
16      }
17  }
```

It is common to use a number of threads equal to the number of CPUs available. However, in many cases the optimum number of threads can be larger or smaller. This depends on the I/O bandwidth, speed of processors, amount of memory, and other computer specific parameters. Only through trial and error can a developer determine the optimal number of threads.

6.3.2 Tile Creation for Distributed Computing

In the previous section, our multiple lines of execution had the advantage of shared memory to communicate and exchange data. This is not the case for distributed computing since we are spreading our processing across multiple systems called compute nodes. Groups of compute nodes are often called clusters. With distributed computing, communication between nodes is done via a network. In some cases, clusters are connected with dedicated high speed networks like InfiniBand or 10 Gigabit Ethernet. In other cases, compute nodes may be spread out geographically and connected via the Internet. Clusters vary greatly in composition and use. Some clusters fill the traditional role of supercomputers, while others are used to provide services to the public. Some clusters are specially configured groups of identical computer systems while others are ad hoc groupings. Others are made of virtualized systems, dynamically allocated to meet on-demand needs. For the purposes of tile creation, the issues to be considered are nearly the same irrespective of the composition of the cluster.

The two primary tasks related to tile creation using computational clusters are creating a system for breaking the tile creation process into smaller, independent tasks and choosing a software framework for developing the solution. The exact physical configuration of a cluster is less important than these two issues.

The problem of dividing the tile creation process into smaller, independent tasks has already been introduced in Section 6.1. In that section we discussed the requirement for smaller tile creating tasks so that all the tiles could be held in memory. In the context of distributed computing, we have to subdivide our tasks so that we can distribute the source images in smaller collections to individual compute nodes and then collect the created tiles from each node. In the next chapter, we will present a tile storage solution that ties all of these requirements together and presents a general solution for sub-dividing tile creation tasks.

Tile creation is typically an I/O bound problem. As discussed, reading and writing to disk or network is far slower than computing the content of tiles. Tiled image calculations are relatively simple, linear pixel transformations. Given this property, we must minimize data movement to make distributed tile creation a beneficial technique.

In the next sections, we will discuss several software frameworks for distributed computing. In the context of distributed computing, the framework is a software application programming interface (API) that facilitates sharing of data and managing control flow of parallel programs. The chosen software framework usually drives the logical configuration of the computational process. We will introduce the basic concept of each framework and discuss how its properties relate to the tile creation process.

6.3.2.1 MPI

A very common cluster framework is called MPI (Message Passing Interface). MPI is a language independent communications protocol for parallel computing. MPI is just a specification. To be used, a developer must select a concrete implementation of the specification. Fortunately there are several implementations, both open source and commercial. It is most commonly used with the C and FORTRAN languages, although bindings exist for other languages like Java, Python, and the Matlab environment.

MPI provides low-level mechanisms for moving data between nodes, control of execution, and synchronization between independent processes. MPI implementations are very efficient and are a good choice for parallel applications that require a lot of interaction between nodes, as is common for some types of scientific supercomputing. It is also a good candidate for parallel applications that are primarily CPU bound. That is, those that require extensive computations with little I/O. In contrast, tile creation is often I/O bound, especially when it utilizes multi-threading techniques discussed in the previous section.

A fully functional tile creation system could be created utilizing MPI for node-to-node communication and control. However, the relatively low-level nature of MPI commands provides unneeded functionality and would make development a very tedious process. For this reason, we recommend a higher level framework.

6.3.2.2 MapReduce

MapReduce is a distributed computing model created by Google and designed to allow computing problems to be easily solved in a multi-processing environment, from a single shared-memory machine up to a large cluster of heterogeneous networked computers [1]. Users provide `map` and `reduce` functions specific to their problem. The MapReduce framework implementation coordinates the distributed computing environment using these functions. The original Google MapReduce implementation is not available to the public. Hadoop[1] is an open source implementation which is commonly used outside of Google.

The MapReduce model is derived from common functional programming techniques. The `map` function takes an input record in the form of a key/value pair. The `map` function then processes that input and creates a new intermediate key/value pair. The `reduce` function takes one of these intermediate keys as input as well as its associated values. The function then merges these values, usually into a single value. The MapReduce framework distributes input records to the `map` function and receives its output. It then distributes those intermediate values to the `reduce` function and receives the merged results. Communication and errors are also managed by the MapReduce framework.

MapReduce is most effectively used when the target problem is computationally bound, the input has a large number of records, or the distributed computing platform is large and complex. The MapReduce framework handles most of the management while a user only need implement the details for a specific task process. The benefits of MapReduce are dependent on the implementation being used. Different MapReduce implementations may provide different I/O capabilities, management capabilities, and error handling capabilities. For example, the Hadoop implementation does not allow random writes within files in its filesystem.

MapReduce may be used as a framework for tiling in a cluster, however, its capabilities are not necessarily well aligned to the task because the tiling process is I/O bound. The source data, large image files, and the output data, large tile files, must be moved to and from the processing systems. The computational cost of processing the imagery is much smaller than the I/O cost. The distributed file system used by the MapReduce framework (HDFS for Hadoop) will incur as much or more I/O cost by positioning files throughout the network. Retrieving data for use elsewhere will incur the same penalties. Additionally, large clusters are not necessary to tile map imagery. Fewer than 20 (potentially fewer than 10) processing nodes need be used to process even the largest imagery datasets in a reasonable amount of time (a few days). Given that the tile creation process is perfectly parallelizable, the complexity of the overall tile processing system is not large enough to support the use of a MapReduce framework.

The MapReduce model works well for a user such as Google because they have a large number of diverse distributed computing problems that may be solved using one framework. They also have a large and geographically diverse computing clus-

[1] http://hadoop.apache.org/

ter, which would be difficult to manage without a model such as MapReduce. Tiling is not computationally bound, has a relatively limited number of image inputs, and may be run on smaller cluster systems. Unless a MapReduce framework is useful to other applications in the enterprise, it is not necessary to use for tiling. The cost of installing the MapReduce framework combined with the cost of implementing the tile processing in using the MapReduce framework will not be significantly lower than simply creating an ad hoc clustering system.

6.3.2.3 Ad Hoc Clustering

Ad hoc clustering refers to distributed computing with no specific software framework. Software frameworks often provide useful tools, but they also introduce overhead either at execution time or development time. There are many ways to control program execution and data sharing between networked computers. REXEC or Secure Shell (SSH) can be used to remotely start and control processes on networked systems. Server Message Block (SMB) and Network File System (NFS) can be used to share data across a network through remotely mounted file systems.

Given the data intensive nature of tile creation processes, developers should create distributed systems with very minimal interactions between systems. With this constraint, sophisticated communication and control frameworks should be needed only in cases where truly large numbers of CPUs are being controlled. We typically create multi-terabyte tile sets on a cluster with 64 CPUs in just a few hours of compute time. Those CPUs are controlled with SSH commands and share data via NFS.

6.4 Partial Updating of Existing Tiled Image Sets

In the previous sections on tile creation, we have assumed all tile sets are created from source images in one single and final step. How, then, should we handle cases in which new source images need to be added to an existing tile set? This is a common problem for tile sets based on satellite or aerial imagery. These sensing platforms can image only a small portion of the earth's surface. A complete picture of a sizable area will include source images taken over an extended time period.

The most basic approach to handling updated images is to simply discard the previous tile set and create a new one each time new source images are available. In some cases, this is the best approach. If a majority of the source images have been updated or if the tile set is rather small, it may be just as efficient to start over. However, if the existing tile set is large and the updates are relatively small, starting over would be expensive or even impossible. Consider a very large example tile set that takes two weeks to create. If some source images are updated every week, we would have to start processing a new tile set as soon as the previous one finished. To keep up-to-date, we would always be processing the large tile set, and most of

our processing would be redundant. This would be expensive both in our time and computational resources.

A better approach is to integrate updated source images into existing tile sets by altering the contents of only the tiles that are affected by the new source images. Logically, the change in our tile algorithm is very simple. Instead of creating a new empty image for a tile, we simply retrieve the existing tile from storage, update its contents, and store the new image. The main difficulty lies in developing a tile storage system capable of handling updated image files. Another challenge is that we must maintain sufficient source image and tile metadata so we can detect which source images should be added to the tile set. Both of these problems relate directly to tile storage and will be discussed in the next chapter.

Listing 6.6 Push-based tile creation with a memory tile cache.

```
1  public void createCachedTiles(TileCache cache, SourceImage[] sourceImages, int
       baseScale) {
2      //Determine the geographic bounds of the tile set.
3      //This can be based on the bounds of the source images.
4      BoundingBox[] sourceImageBounds = new BoundingBox[sourceImages.length];
5      for (int i = 0; i < sourceImageBounds.length; i++) {
6    sourceImageBounds[i] = sourceImages[i].bb;
7      }
8      BoundingBox tileSetBounds = BoundingBox.union(sourceImageBounds);
9      //Determine the bounds of the tile set in tile coordinates.
10     long tilesetMincol = (long) Math.floor((tileSetBounds.minx + 180.0) /
           (360.0 / Math.pow(2.0, (double) baseScale)));
11     long tilesetMaxcol = (long) Math.floor((tileSetBounds.maxx + 180.0) /
           (360.0 / Math.pow(2.0, (double) baseScale)));
12     long tilesetMinrow = (long) Math.floor((tileSetBounds.miny + 90.0) / (180.0
           / Math.pow(2.0, (double) baseScale - 1)));
13     long tilesetMaxrow = (long) Math.floor((tileSetBounds.maxy + 90.0) / (180.0
           / Math.pow(2.0, (double) baseScale - 1)));
14
15     //Iterate over the source images
16     for (int i = 0; i < sourceImages.length; i++) {
17   BoundingBox currentBounds = sourceImages[i].bb;
18   //Compute the bounds of the source image in tile coordinates
19   long mincol = (long) Math.floor((currentBounds.minx + 180.0) / (360.0 / Math.
         pow(2.0, (double) baseScale)));
20   long maxcol = (long) Math.floor((currentBounds.maxx + 180.0) / (360.0 / Math.
         pow(2.0, (double) baseScale)));
21   long minrow = (long) Math.floor((currentBounds.miny + 90.0) / (180.0 / Math.
         pow(2.0, (double) baseScale - 1)));
22   long maxrow = (long) Math.floor((currentBounds.maxy + 90.0) / (180.0 / Math.
         pow(2.0, (double) baseScale - 1)));
23   //Read the source image into memory
24   BufferedImage bi = readImage(sourceImages[i].name);
25   for (long c = mincol; c <= maxcol; c++) {
26       for (long r = minrow; r <= maxrow; r++) {
27     TileAddress address = new TileAddress(r, c, baseScale);
28     //Compute the geographic bounds of the specific tile.
29     BoundingBox tileBounds = address.getBoundingBox();
30     //Check the TileCache for the tiled image
31     BufferedImage tileImage = cache.getTile(address);
32     if (tileImage == null) {
33         tileImage = new BufferedImage(TILE_SIZE, TILE_SIZE, BufferedImage.
               TYPE_INT_ARGB);
34         cache.putTile(address, tileImage);
35         //Extract the required image data from the source image and store it in
               the tiled image.
36         drawImageToImage(bi, sourceImages[i].bb, tileImage, tileBounds);
37         //Note that since tileImage is a pointer to the bufferedimage already
               in the cache,
38         //we don't have to put it back in after each use.
39     }
40       }
41   }
42     }
43     for (int scale = baseScale - 1; scale <= 1; scale--) {
44   //Determine the bounds of the current tile scale in tile coordinates.
45   //ratio will be used to reduce the original tile set bounding coordinates to
         those applicable for each successive scale.
46   int ratio = (int) Math.pow(2, baseScale - scale);
47   long curMinCol = (long) Math.floor(tilesetMincol / ratio);
48   long curMaxCol = (long) Math.floor(tilesetMaxcol / ratio);
49   long curMinRow = (long) Math.floor(tilesetMinrow / ratio);
50   long curMaxRow = (long) Math.floor(tilesetMaxrow / ratio);
51   //Iterate over the tile set coordinates.
52   for (long c = curMinCol; c <= curMaxCol; c++) {
53       for (long r = curMinRow; r <= curMaxRow; r++) {
```

```
54      //For each tile, do the following:
55      TileAddress address = new TileAddress(r, c, scale);
56      //Determine the FOUR tiles from the higher scale that contribute to the
            current tile.
57      TileAddress tile00 = new TileAddress(r * 2, c * 2, scale + 1);
58      TileAddress tile01 = new TileAddress(r * 2, c * 2, scale + 1);
59      TileAddress tile10 = new TileAddress(r * 2, c * 2, scale + 1);
60      TileAddress tile11 = new TileAddress(r * 2, c * 2, scale + 1);
61      //Retrieve the four tile images, or as many as exist.
62      BufferedImage image00 = cache.getTile(tile00);
63      BufferedImage image01 = cache.getTile(tile01);
64      BufferedImage image10 = cache.getTile(tile10);
65      BufferedImage image11 = cache.getTile(tile11);
66      //Combine the four tile images into a single, scaled-down image.
67      BufferedImage tileImage = new BufferedImage(TILE_SIZE, TILE_SIZE,
            BufferedImage.TYPE_INT_ARGB);
68      Graphics2D g = (Graphics2D) tileImage.getGraphics();
69      g.setRenderingHint(RenderingHints.KEY_INTERPOLATION, RenderingHints.
            VALUE_INTERPOLATION_BILINEAR);
70      boolean hadImage = false;
71      if ((image00 != null)) {
72          g.drawImage(image00, 0, Constants.TILE_SIZE_HALF, Constants.
                TILE_SIZE_HALF, Constants.TILE_SIZE, 0, 0, Constants.TILE_SIZE,
73          Constants.TILE_SIZE, null);
74          hadImage = true;
75      }
76      if ((image10 != null)) {
77          g.drawImage(image10, Constants.TILE_SIZE_HALF, Constants.TILE_SIZE_HALF
                , Constants.TILE_SIZE, Constants.TILE_SIZE, 0, 0,
78          Constants.TILE_SIZE, Constants.TILE_SIZE, null);
79          hadImage = true;
80      }
81      if ((image01 != null)) {
82          g.drawImage(image01, 0, 0, Constants.TILE_SIZE_HALF, Constants.
                TILE_SIZE_HALF, 0, 0, Constants.TILE_SIZE,
83          Constants.TILE_SIZE, null);
84          hadImage = true;
85      }
86      if ((image11 != null)) {
87          g.drawImage(image11, Constants.TILE_SIZE_HALF, 0, Constants.TILE_SIZE,
                Constants.TILE_SIZE_HALF, 0, 0, Constants.TILE_SIZE,
88          Constants.TILE_SIZE, null);
89          hadImage = true;
90      }
91      //save the completed tiled image to the tile storage mechanism.
92      if (hadImage) {
93          cache.putTile(address, tileImage);
94      }
95        }
96    }
97      }
98  }
```

Listing 6.7 Read an image region with scanline based access.

```
abstract void skipScanlines(ImagePointer im, int num);

abstract void readScanline(ImagePointer im, byte[] scanlineBuffer);

byte[] readScanlines(ImagePointer im, int imageWidth, int imageHeight, int x,
     int y, int height, int width) {
    byte[] outputImage = new byte[imageWidth * imageHeight * 3];
    int startScanline = y - 1;
    skipScanlines(im, startScanline);
    byte[] tempBuffer = new byte[imageWidth * 3];
    int imageCounter = 0;
    int scanlineOffset = x * 3;
    for (int i = 0; i < height; i++) {
  readScanline(im, tempBuffer);
  for (int j = 0; j < (width * 3); j++) {
      outputImage[imageCounter] = tempBuffer[j + scanlineOffset];
      imageCounter++;
  }
    }
    return outputImage;
}
```

Listing 6.8 Read a partial image region with tile-based image access.

```
    abstract void seekToTile(ImagePointer im, int i, int j);

    abstract void readTile(ImagePointer im, byte[] tileBuffer);

    byte[] readTiles(ImagePointer im, int imageWidth, int imageHeight, int
         tileWidth, int tileHeight, int x, int y, int height, int width) {

        //Determine the range of tiles that will need to be read.
        double numtiles = (Math.ceil((double) imageWidth / tileWidth)) * (Math.
             ceil((double) imageHeight / tileHeight));

        int startXTile = (int) Math.floor((double) x / tileWidth);
        int startYTile = (int) Math.floor((double) y / tileHeight);
        int endx = x + width - 1;
        if (endx > imageWidth) {
            endx = imageWidth;
        }
        int endy = y + height - 1;
        if (endy > imageHeight) {
            endy = imageHeight;
        }
        int endXTile = (int) Math.floor((double) endx / tileWidth);
        int endYTile = (int) Math.floor((double) endy / tileHeight);

        int tileSizeBytes = tileWidth * tileHeight * 3;

        int numtilesToDecode = (endXTile - startXTile + 1) * (endYTile -
             startYTile + 1);

        //Construct a temporary buffer with sufficient size to hold all of the
             needed tiles.
        byte[] tempImage = new byte[numtilesToDecode * tileSizeBytes];

        int tempImageRowWidth = (endXTile - startXTile + 1) * 3 * tileWidth;

        byte[] tileBuffer = new byte[tileSizeBytes];

        int startYTileCoord = startYTile * tileHeight;
        int startXTileCoord = startXTile * tileWidth;
        int bufferOffset = 0;
        //Iterate over the tiles, in row-major order.
```

```
        for (int ty = startYTile; y <= endYTile; y++) {
            for (int tx = startXTile; x <= endXTile; x++) {
                //Position the image pointer to read at the needed tile.
                seekToTile(im, tx, ty);
                //Read the tile into the temporary buffer.
                readTile(im, tileBuffer);
                int bufferStartYTile = (ty - startYTile);
                int bufferStartXTile = (tx - startXTile);
                int bufferStartYPixel = bufferStartYTile * tileHeight;
                for (int m = 0; m< tileHeight; m++) {
                    int startRow = (bufferStartYPixel + m) * tempImageRowWidth;
                    int startColumn = bufferStartXTile * 3 * tileWidth;
                    for (int n = 0; n < tileWidth * 3; n++) {
                        tempImage[startRow + startColumn + n] = tileBuffer[m *
                            tileWidth * 3 + n];
                    }
                }
            }
        }
        //Trim the temporary buffer to match the desired region.
        int xOffset = x - startXTile * tileWidth;
        int yOffset = y - startYTile * tileHeight;
        byte[] outputImage = new byte[imageWidth * imageHeight * 3];
        int imageCounter = 0;
        for (int i = 0; i < imageHeight; i++) {
            int rowOffset = (yOffset + i) * tileWidth;
            for (int j = 0; j < imageWidth; j++) {
                int columnOffset = j + yOffset;
                outputImage[imageCounter] = tempImage[rowOffset + columnOffset
                    ];
                imageCounter++;
            }
        }
        return outputImage;
    }
```

Listing 6.9 Tile creation with partial source image reading.

```
    abstract byte[] readPartialImage(String name, int x, int y, int width, int
        height);

    abstract BufferedImage convertBytes(byte[] pixels);

    public void createTilesWithPartialReading(SourceImage[] sourceImages,
        TileOutputStream tileOutputStream, int baseLevel) {

        //Determine the geographic bounds of the tile set.
        //This can be based on the bounds of the source images.
        BoundingBox[] sourceImageBounds = new BoundingBox[sourceImages.length];
        for (int i = 0; i < sourceImageBounds.length; i++) {
            sourceImageBounds[i] = sourceImages[i].bb;
        }
        BoundingBox tileSetBounds = BoundingBox.union(sourceImageBounds);
        //Determine the bounds of the tile set in tile coordinates.
        long mincol = (long) Math.floor((tileSetBounds.minx + 180.0) / (360.0 /
            Math.pow(2.0, (double) baseLevel)));
        long maxcol = (long) Math.floor((tileSetBounds.maxx + 180.0) / (360.0 /
            Math.pow(2.0, (double) baseLevel)));
        long minrow = (long) Math.floor((tileSetBounds.miny + 90.0) / (180.0 /
            Math.pow(2.0, (double) baseLevel - 1)));
        long maxrow = (long) Math.floor((tileSetBounds.maxy + 90.0) / (180.0 /
            Math.pow(2.0, (double) baseLevel - 1)));

        //Iterate over the tile set coordinates.
        for (long c = mincol; c <= maxcol; c++) {
            for (long r = minrow; r <= maxrow; r++) {
```

```
23                  TileAddress address = new TileAddress(r, c, baseLevel);
24                  //Compute the geographic bounds of the specific tile.
25                  BoundingBox tileBounds = address.getBoundingBox();
26                  //Iterate over the source images.
27                  BufferedImage tileImage = new BufferedImage(TILE_SIZE,
                        TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
28                  for (int i = 0; i < sourceImages.length; i++) {
29                      //Determine if the specific source image intersects the
                            tile being created.
30                      if (sourceImages[i].bb.intersects(tileBounds.minx,
                            tileBounds.miny, tileBounds.maxx, tileBounds.maxy)) {
31                          //Determine intersection of tile and source image
32                          BoundingBox partialBB = getIntersection(sourceImages[i
                                ].bb, tileBounds);
33                          //Convert geographic coordinates to image coordinates
34                          Rectangle rectangle = convertCoordinates(sourceImages[i
                                ].bb, partialBB, sourceImages[i].width,
                                sourceImages[i].height);
35                          //Read partial image data
36                          byte[] data = readPartialImage(sourceImages[i].name,
                                rectangle.x, rectangle.y, rectangle.width,
                                rectangle.height);
37                          //convert the pixel bytes to a BufferedImage
38                          BufferedImage bi = convertBytes(data);
39                          //Draw the converted pixels to the tile image
40                          drawImageToImage(bi, partialBB, tileImage, tileBounds);
41                      }
42                  }
43                  //Save the completed tiled image to the tile storage mechanism.
44                  tileOutputStream.writeTile(address, tileImage);
45              }
46          }
47      }
```

Listing 6.10 Tile creation with a reader and tiler threads.

```
1   public void createTilesTwoThreads(TileCache cache, SourceImage[] sourceImages,
        int baseLevel) {
2           ReaderThread reader = new ReaderThread(sourceImages);
3           reader.start();
4           TilerThread tiler = new TilerThread(cache, baseLevel, reader);
5           tiler.start();
6           tiler.join();
7       }
8
9       class TilerThread extends Thread {
10
11          private TileCache cache;
12          private int baseLevel;
13          private ReaderThread reader;
14
15          public TilerThread(TileCache tileCache, int baseLevel, ReaderThread
                reader) {
16              this.cache = tileCache;
17              this.baseLevel = baseLevel;
18              this.reader = reader;
19          }
20
21          public void run() {
22
23              ImageWrapper image = reader.getImage();
24              while (image != null) {
25                  BoundingBox currentBounds = image.si.bb;
26                  long mincol = (long) Math.floor((currentBounds.minx + 180.0) /
                        (360.0 / Math.pow(2.0, (double) baseLevel)));
```

```
27                  long maxcol = (long) Math.floor((currentBounds.maxx + 180.0) /
                        (360.0 / Math.pow(2.0, (double) baseLevel)));
28                  long minrow = (long) Math.floor((currentBounds.miny + 90.0) /
                        (180.0 / Math.pow(2.0, (double) baseLevel - 1)));
29                  long maxrow = (long) Math.floor((currentBounds.maxy + 90.0) /
                        (180.0 / Math.pow(2.0, (double) baseLevel - 1)));
30                  BufferedImage bi = image.bi;
31                  for (long c = mincol; c <= maxcol; c++) {
32                      for (long r = minrow; r <= maxrow; r++) {
33                          TileAddress address = new TileAddress(r, c, baseLevel);
34                          BoundingBox tileBounds = address.getBoundingBox();
35                          BufferedImage tileImage = cache.getTile(address);
36                          if (tileImage == null) {
37                              tileImage = new BufferedImage(TILE_SIZE, TILE_SIZE,
                                    BufferedImage.TYPE_INT_ARGB);
38                              cache.putTile(address, tileImage);
39
40                          }
41                          drawImageToImage(bi, currentBounds, tileImage,
                                tileBounds);
42                      }
43                  }
44                  image = reader.getImage();
45              }
46          }
47      }
48
49      class ReaderThread extends Thread {
50
51          List<SourceImage> images = Collections.synchronizedList(new ArrayList<
                SourceImage>());
52
53          ImageWrapper currentImage = null;
54
55          public ReaderThread(SourceImage[] images) {
56              for (int i = 0; i < images.length; i++) {
57                  this.images.add(images[i]);
58              }
59          }
60
61          public void run() {
62
63              while (images.size() > 0) {
64                  if (currentImage == null) {
65                      SourceImage si = images.remove(0);
66                      BufferedImage bi = readImage(si.name);
67                      ImageWrapper iw = new ImageWrapper(si, bi);
68                      currentImage = iw;
69                  }
70                  try {
71                      Thread.sleep(200);
72                  } catch (InterruptedException e) {
73                      e.printStackTrace();
74                  }
75              }
76
77          }
78
79          public synchronized ImageWrapper getImage() {
80              ImageWrapper returnVal = null;
81              while (currentImage == null) {
82                  if (images.size() == 0) {
83                      return null;
84                  }
85                  try {
86                      Thread.sleep(400);
87                  } catch (InterruptedException e) {
```

```
88                      e.printStackTrace();
89                  }
90              }
91              returnVal = currentImage;
92              currentImage = null;
93
94              return returnVal;
95          }
96      }
97
98      class ImageWrapper {
99
100         BufferedImage bi;
101         SourceImage si;
102
103         public ImageWrapper(SourceImage si, BufferedImage bi) {
104             super();
105             this.si = si;
106             this.bi = bi;
107         }
108
109     }
```

References

1. Dean, J., Ghemawat, S.: Map Reduce: Simplified data processing on large clusters. Communications of the ACM-Association for Computing Machinery-CACM **51**(1), 107–114 (2008)
2. Oaks, S.: Java Threads. O'Reilly (2004)