Parallel Graph Construction in Imaging Science

Abhinav Sinha
Department of Computer Science
Purdue University
West Lafayette, IN
sinha45@purdue.edu

***Abstract*—In Imaging Science, graphs derived from images and image-like data are increasingly being used to represent and analyze these images. These graphs represent data by neighborhood relationships among their voxels, where the notions of distance and similarity vary depending on the application. Since images now commonly have millions of pixels, practical applications can easily have millions of nodes and 10 million to a few billion edges, with neighborhood regions as far away as 50 to 100 voxels. Despite the computation being highly parallelizable, building and manipulating these graphs is slow, especially for three-dimensional images in medical applications, which limits the utility of methods based on these graphs. A simple implementation can take hours to build a single graph. On top of run time considerations, the sheer size of these graphs often leads to memory bottlenecks. In this paper, we present methods to parallelize these computations, allowing us to explore possibilities to speed up graph construction in practice. We improve performance by exploiting the structure of the data to achieve efficient data parallelism without race conditions or the need for locking of resources. We provide complete single-thread, multi-thread and CUDA implementations in Julia for constructing graphs from three-dimensional images. Finally, we show the effectiveness of these implementations through experimental results on LGE MRI data with a size up to millions of nodes and a few billion edges.**

I. Introduction

As the field of Imaging Science develops, graphs are emerging as a convenient and potent representation for analyzing image data. In regards to an image, constructing a graph is simply connecting each voxel (a pixel in three-dimensions) in an image to all its neighbors within a certain search distance. These types of graphs are versatile in nature, meaning that they can analyzed using many different methods depending on the problem. One such application is in the problem of local graph clustering. Local graph clustering looks to identify clusters of nodes that are tightly connected, given a set of seed vertices [10]. Other applications include manifold-based hyperspectral imaging and dimension reduction [1], image search and object retrieval [4], and pattern recognition [8]. However, the aforementioned construction process is often painstakingly slow and memory intensive, and thus, limits the viability of these analysis methods in practical applications. For instance, on our larger machine (see Section V-C for specifications), constructing a graph for the full body LGE MRI of size $576 \times 576 \times 88$ with a search distance of 10 takes slightly over 2 hours with only a single-thread. In this paper, we aim to show that it is indeed worthwhile to parallelize this construction process to improve performance on large images and their graphs.

I would like to show my deep appreciation for my advisor David F. Gleich (dgleich@purdue.edu) for his continuous guidance and support throughout this project.

II. Background

An image can be represented as a matrix $\boldsymbol{M}$, in which each element of the the matrix is represents a pixel or voxel in an image. The index of that element represents the location of the voxel, and the its value is the value of the voxel. Suppose the 3×3 tile in Figure 1 was a complete image. The matrix representation of that image is

$$\boldsymbol{M} = \begin{bmatrix} 0.97 & 0.43 & 0.00 \\ 0.70 & 0.82 & 0.00 \\ 0.12 & 0.92 & 0.11 \end{bmatrix}.$$

The same principles apply in three-dimensions.

Given an image represented as a three-dimensional matrix $\boldsymbol{M}$, a search distance d, and a similarity function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, we construct an undirected and weighted graph $G = (V, E)$. Each vertex v is a voxel in the given image (thus, an element in $\boldsymbol{M}$), and two vertices v_1, v_2 are connected by a weighted edge if $\text{dist}(v_1, v_2) \leq d$. The search distance is the Euclidean distance; i.e.

$$\text{dist}(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}.$$

The edge weight is determined by $f(v_1, v_2)$.

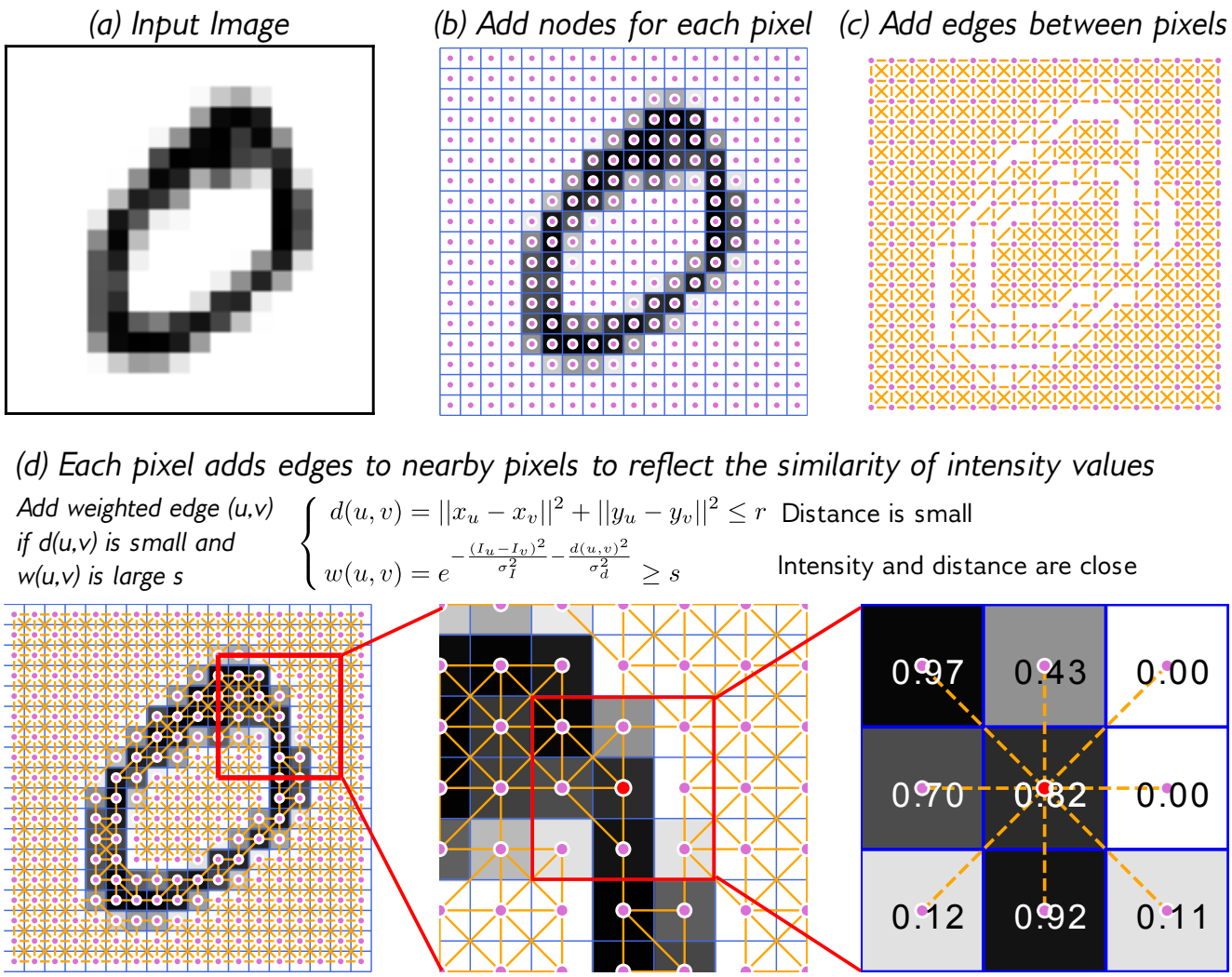


Fig. 1: A two dimensional example of constructing a graph from an image based on distance and similarity of color. We add a node in the graph for each pixel in the image. An edge is attached between two nodes if there are within a distance r and their computed similarity is at least s. Example taken from David Gleich's "Flow-based Algorithms for Improving Clusters" [3].

In our implementation, the similarity function f is a function that takes in two floats as inputs as returns a single float. For example, Figure 2 is the code for the similarity function:

$$f(v_1, v_2) = \sqrt{\frac{v_1}{63}} - \sqrt{\frac{v_2}{63}}.$$

```
function mri_diff_fn(
  v1::T,
  v2::T
)::T where T <: Float32
  return (
    min(sqrt(v1)/63,1.0) -
    min(sqrt(v2)/63,1.0)
  )
end
```

Fig. 2: Example representation of the MRI similarity function in Julia.

A. Single-thread and Graph Representation in Practice

Intuitively, constructing a graph from an image involves visiting each voxel, finding all its neighbors, and attaching each pair with a weighted edge. This process is illustrated in Figure 1. We start with a simple single-threaded algorithm, whose Julia implementation is provided in Figure 3.

As opposed to constructing a sparse adjacency matrix directly, we propose two different approaches to represent the resulting graph in our algorithms: either a set of four vectors, or a modified adjacency matrix. Because sparse matrices in Julia are stored in Compressed Sparse Column (CSC) format, incrementally creating (or, adding non-zero rows) is very inefficient [6]. Internally, memory will be constantly copied or reallocated to maintain sparsity. Dense matrices should not be used, especially for large images; much of the allocated memory will be unused.

In our first approach, we choose to represent the graph as the vectors e_i, e_j, e_d, and e_v, where e_i are the source vertices, e_j are the destination vertices, e_d are the distances between vertices, and e_v are the weights of edges. These vectors are roughly of length $|V|\cdot(2d+1)^3$ (see Section III for a more refined approach on computing the expected length of these vectors). The second approach emulates a standard adjacency matrix. We construct a modified weighted adjacency matrix $\boldsymbol{A}$ of size $(2d+1)^3 \times |V|$. All the neighbors of a voxel are contained within a cube centered around that particular voxel. Thus, we can map the location of each neighbor, relative to its source voxel, to a linear index based on this cube; we call this mapped linear index a voxel's neighbor offset. The rows of $\boldsymbol{A}$ are the possible neighbor offsets for the given search distance d. The columns are simply the voxels of the image. From $\boldsymbol{A}$, we can extract e_i, e_j, e_d, and e_v by using the neighbor offsets to determine e_j and e_d. The Julia implementation in Figure 4 shows how to use $\boldsymbol{V}$ in a single-thread implementation.

III. Size of the Image Graph

In this section, we compute bounds on the size of the graph in order to preallocate memory. Most, if not all, programming languages that allow for dynamic allocation on the stack or heap call for reallocation if the intended size of a vector is greater than its current capacity. In most cases, including Julia, reallocation usually doubles the vector's capacity [5]. Thus, a vector whose final length is n will automatically call for reallocation $\sim \log_2 n$ times. For very large vectors without size hints or a preallocated size, these repeated operations greatly impact performance. In practice, we can avoid those operations by computing an upper bound on the number of edges of the resulting graph.

A. Bounding the Number of Edges in G

Let $\boldsymbol{M}$ be the matrix representation of an image of size $s_x \times s_y \times s_z$ and $d > 0$ be the search distance.

```
function st_construct(
  image::AbstractArray{<: Unsigned},
  d::Int, diff_fn::Function
)
  # iterator for Cartesian indices of image
  R = CartesianIndices(data)

  # converge R to linear indices
  imap = LinearIndices(R)

  # find first and last index of image
  cf, cl = first(R), last(R)

  # convert d to a Cartesian index
  # i.e. d = 2 to dd = (2, 2, 2)
  dd = d * oneunit(cf)

  # allocate memory for result vectors,
  # i.e. the sparse array inputs
  ei = zeros(Int, 0) # src vector
  ej = zeros(Int, 0) # dest vector
  evd = zeros(Float32, 0) # distances
  evi = zeros(Float32, 0) # edge weights

  # iterate over all indices in image
  @inbounds for I in R
    # find the first and last index of
    # the neighbors of I
    lower = max(cf, I-dd)
    upper = min(cl, I+dd)

    src = imap[I] # linear index of I
    pi = image[I] # voxel value at I

    # iterate over all neighbors of I
    for J in lower : upper
      dst = imap[J] # linear index of J

      # do not construct self-loops
      (src == dst) && continue

      pj = image[J] # voxel value at J

      # compute distance and edge weight
      dist = norm(Tuple(I - J))^2
      sim = diff_fn(pi, pj)

      # store edge in result vectors
      push!(ei, src)
      push!(ej, dst)
      push!(evd, dist)
      push!(evi, sim)
    end
  end

  # return result vectors
  ei, ej, evd, evi
end
```

Fig. 3: Julia implementation for single-thread graph construction using e_i, e_j, e_d, and e_v to represent our graph.

```
function st_construct(
  image::AbstractArray{<: Unsigned},
  d::Int, diff_fn::Function
)
  # iterator for Cartesian indices of image
  R = CartesianIndices(image)

  # converge R to linear indices
  imap = LinearIndices(R)

  cf, cl = first(R), last(R)

  # convert d to a Cartesian index
  # i.e. d = 2 to dd = (2, 2, 2)
  dd = d * oneunit(cf)

  # compute largest cube that contains all
  # potential neighbors of a voxel
  bm = 3d * oneunit(cf)
  neighbor_box = dd : bm

  # converge box to linear indices
  jmap = LinearIndices(dd:bm)

  # allocate V: (2d+1)^3 by n
  n = length(imap)
  V = fill(NaN32, (2d+1)^ndims(image), n)

  # iterate over all indices in image
  @inbounds for I in R
    # find the first and last index of
    # the neighbors of I
    lower = max(cf, I-dd)
    upper = min(cl, I+dd)

    src = imap[I] # linear index of I
    pi = image[I] # voxel value at I

    # iterate over all neighbors of I
    for J in lower : upper
      dst = imap[J] # linear index of J

      # do not construct self-loops
      src == dst && continue

      pj = image[J] # voxel value at J

      # map J to linear index in jmap
      noffset = jmap[J-lower+oneunit(cf)]

      # store edge in V
      V[noffset, src] = diff_fn(pi, pj)
    end
  end

  # return graph as V
  # and linear map of neighbors
  V, jmap
end
```

Fig. 4: Julia implementation for single-thread graph construction using our modified adjacency matrix $\boldsymbol{V}$ to represent a graph.

Since the search distance is the Euclidean distance, we note that a point p can have neighbors along the x, y, and z-dimensions as well as diagonally with respect to p. Consider a cube $\boldsymbol{B}$ in $\mathbb{Z}^3$ with side length $2d+1$ centered about p. The total number of neighbors is given by the volume of the resulting cube minus 1 (we do not consider a point to be a neighbor of itself):

$$n_p = V(\boldsymbol{B}) - 1 = (2d+1)^3 - 1.$$

From this, a rough upper bound on the number of edges u is given by

$$u = n_p(s_x s_y s_z).$$

While this rough bound may be sufficient for small images, we can tighten this bound by considering the possibile locations of p. We have $\forall p \in \boldsymbol{M}$, p is one of the following:

- p_c – in the "center" of the $\boldsymbol{M}$,
- p_s – on the "surface" of the $\boldsymbol{M}$,
- p_e – on the "edge" of the $\boldsymbol{M}$, or
- p_r – the "corner" of $\boldsymbol{M}$.

Here, "edge" refers to the edges of a polytope in $\mathbb{R}^3$, which differs from the notion of a graph edge. The neighbors of the latter three points are constrained by the size of the $\boldsymbol{M}$. See Figure 5 for an example of these four cases. A tighter upper bound would then be

$$u = n_c c_c + n_s c_s + n_e c_e + n_r c_r,$$

where n is the number of neighbors, and c is the count of each type of p.

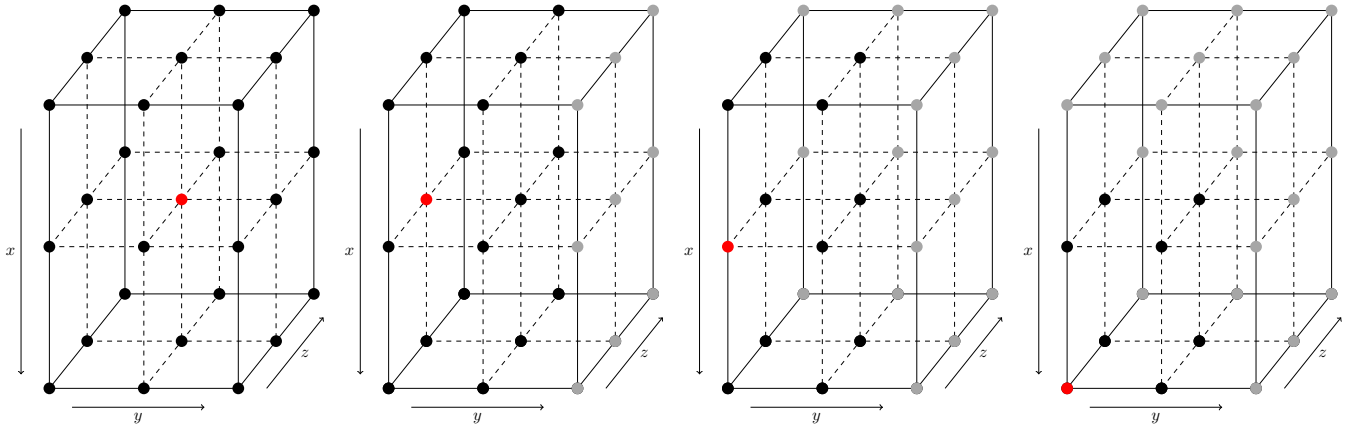
Continuing the geometric interpretation in $\mathbb{Z}^3$, we find that for each type of point p, there exists a rectangular prism that covers all possible neighbors a distance d away from p. For the sake of brevity, we will refer to rectangular prisms as simply prisms. We can construct, for each type of point:

- p_c – prisms of size $(2d+1) \times (2d+1) \times (2d+1)$,
- p_s – prisms of size $(2d+1) \times (2d+1) \times (d+1)$,
- p_e – prisms of size $(2d+1) \times (d+1) \times (d+1)$,
- p_r – prisms of size $(d+1) \times (d+1) \times (d+1)$.

It follows that

$$\begin{aligned} n_c &= (2d+1)^3 - 1, \\ n_s &= (d+1)(2d+1)^2 - 1, \\ n_e &= (d+1)^2(2d+1) - 1, \text{ and} \\ n_r &= (d+1)^3 - 1. \end{aligned}$$

The number of points in the center of the image can be characterized as the volume of a prism $\boldsymbol{P}$, whose size is that of $\boldsymbol{M}$, excluding the surface, edges, and corners.



(a) "Center" voxel (b) "Surface" voxel (c) "Edge" voxel (d) "Corner" voxel

Fig. 5: Visualization of the four voxel cases for $d = 1$ on an image of size $3 \times 3 \times 3$. The red voxel – the voxel of interest – is a neighbor to at most all the black voxels. Grey voxels, while in the image, are not neighbors to red when $d = 1$.

This is equivalent to a smaller prism of size $(s_x - 2) \times (s_y - 2) \times (s_z - 2)$, so that $c_c = (s_x - 2) \times (s_y - 2) \times (s_z - 2)$. On the surface, the number of points is equivalent to the surface area of the smaller prism, since we exclude the edges and corners; i.e. $c_s = 2[(s_x - 2)(s_y - 2) + (s_x - 2)(s_z - 2) + (s_y - 2)(s_z - 2)]$. For the number of points on the edge of the image, we have the perimeter of $\boldsymbol{P}$ minus the corners. Thus, $c_e = 4(s_x + s_y) + 4s_z - c_r = 4(s_x + s_y + s_z) - c_r$. Clearly, $c_r = 8$, and so $c_e = 4(s_x + s_y + s_z) - 8$.

This bound can be made tighter for $d > 1$ by considering points that are in the center of the image, but still less than d away from the surface. However, our proposed bound works well in practice. For the LGE MRI image of size $576 \times 576 \times 88$ and $d = 2$, we find that the number of edges is 3555589896, while the computed bound is 3577124352 – an error of 0.6%.

B. Bounding the Number of Edges in a Section of G

In the case of our multi-threaded algorithm, we need to compute bounds on the number of edges in each thread's section of the graph. Even though the indices processed by each thread are unique, there will be overlap between the indices *visited* by each thread. Namely, those indices on the outermost planes of each section will require access to indices in their neighboring block(s).

We compute this bound by treating each block as its own image and computing the bound from Section III-A. To account for the additional edges to elements in neighboring blocks, we calculate the volume of the protruding prism. Suppose the image was largest along the z-dimension, we can then divide the image along this dimension, consistent with the description from Section IV-A. The projected prism $\boldsymbol{P}$ would then be of size $s_x \times s_y \times d$. Now, consider the plane that borders the block of the next thread. We have three types of points:

- p_c – points in the center of the plane,
- p_p – points on the perimeter of the plane, and
- p_r – points on the corners of the plane.

We find that the bound u_p for the edges projecting from this plane is

$$u_p = n_c c_c + n_p c_p + n_r c_r.$$

Following similar geometric reasoning as the previous section, we find that the number of neighbors for each point is

$$\begin{aligned} n_c &= d(2d+1)^2, \\ n_p &= d(d+1)(2d+1), \\ n_r &= d(d+1)^2. \end{aligned}$$

The corresponding counts are $c_c = (s_x - 2)(s_y - 2)$, $c_p = 2(s_x + s_y) - c_r$, and $c_r = 4$. The bound for the top and bottom block (e.g. Block 1 and Block n in Figure 6), is then

$$u_s = u_b u_p,$$

and the bound for the middle blocks (Block 2 to Block $n - 1$) is

$$u_m = 2u_b u_p.$$

IV. Parallel Graph Construction – im2gr

The primary goal for *im2gr* is to provide a starting point for parallelizing the graph construction of a three-dimensional image. We design our algorithms to avoid race conditions or the need for locking of resources.

The implementation in Figure 3 contains two loops. The first loop visits each node of the graph, while the second loop visits all the neighbors for the current node of loop one. The body of the first loop (lines 7 to 15) is the same for each node index, and thus, is a good candidate for data parallelism.

A. Multi-threaded Construction

The approach we take to achieve data parallelism is to divide $\boldsymbol{M}$ into blocks, so that each thread operates on its own block, equivalent to a unique section of its data. Each thread essentially runs code presented in our single-thread implementation in Figure 3, but only its section of the data. After each thread completes its assigned block, the result vectors are merged and returned. The ordering of e_i, e_j, e_d, and e_v as a whole need not be consistent between runs, but the indices in each vector must align. In other words, for some index k, $e_d[k]$ and $e_v[k]$ must be the distance and weight for the edge between $e_i[k]$ and $e_j[k]$; k is not required to remain the same for a particular edge.

To find each block, we divide the data into blocks along its longest dimension. Suppose s_z is the longest dimension of $\boldsymbol{M}$ and t is the number of available threads. Then length of each block would be $b = \left\lceil \frac{s_z}{t} \right\rceil$ so that there are $n = \left\lceil \frac{s_z}{b} \right\rceil$ blocks of size $s_x \times s_y \times b$. In the (highly probable) case that s_z is not evenly divisble by t, we choose to relax the size of the last block as opposed to a relaxed stride. Relaxing the stride, would lead to overlap between the data processed by each thread. This overlap could lead to edges being created more than once, unnecessarily and incorrectly increasing the size of the graph. See Figure 6 for an illustration of work division. In terms of load balance, it is possible that the thread responsible for the last block will perform less work than the others.

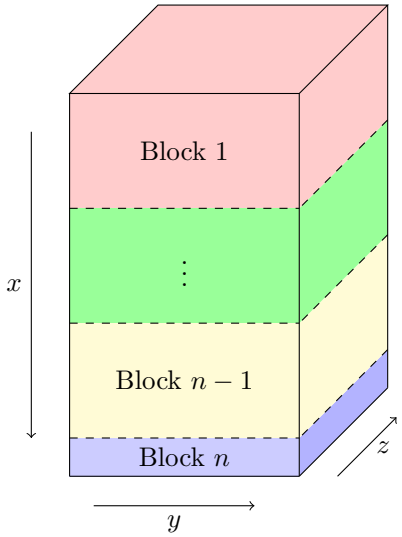


Fig. 6: Division of work across n threads along the x-dimension, with the size of the n^{th} block relaxed.

One of our goals with this multi-threaded algorithm is to forgo using any sort of locking mechanism on our data. Because of the repeated accesses to $\boldsymbol{M}$, locking would have a large overhead that would likely mitigate any performance gains from parallelism. Instead, we choose to maintain thread-local state. This also aligns nicely with Julia's functional programming paradigm to avoid side-effects and allow at most local mutation [2]. We allocate n vectors for each result vector of sizes consistent with the bounds calculated in Section III-B. Each thread is given its own references to one set of those result vectors to store its portion of the graph. Figure 7 is the implementation of this initialization function. The thread-local result vectors are concatenated to form the true e_i, e_j, e_d, and e_v after all threads have completed their task.

B. Construction in CUDA

For CUDA, we followed the natural approach, given the architecture of a GPU. Each thread in the GPU computes the neighbors for a single element. The element processed by a thread is determined by its global index. If the index of a thread is not within the image, then no computation is performed on that thread. We launch the CUDA kernel using the three-dimensional abstraction with a default block size of $(8, 8, 8)$ and the corresponding grid dimensions calculated from the size of the image. The kernel is written to perform the body of loop one from Figure 3 for the index of the thread in its block. While this approach is simple, it does culminate in noticeable performance gains. This suffices as a good starting point for implementing parallel graph construction in CUDA.

V. Experiments

We perform experiments on two different sized images and varying search distances using our implementation of new parallel graph construction methods. We show

```
function mt_init(
  sz, ax, # image size, longest dimension
  nb, bl, # number of blocks, block length
  dd, d, cf, cl
)
  # block start and stop indices, and
  # corresponding data start and stop indices
  bstarts = Vector{CartesianIndex}(undef, nb)
  bstops = Vector{CartesianIndex}(undef, nb)
  dstarts = Vector{CartesianIndex}(undef, nb)
  dstops = Vector{CartesianIndex}(undef, nb)

  # compute partial bounds on G for the
  # top/bottom blocks, and middle blocks
  sb = partialbound(sz, bl, ax, d, Side)
  mb = partialbound(sz, bl, ax, d, Middle)

  # collection of all thread-local vectors
  eis = Vector{Vector{Int}}(undef, nb)
  ejs = Vector{Vector{Int}}(undef, nb)
  evds = Vector{Vector{Float32}}(undef, nb)
  evis = Vector{Vector{Float32}}(undef, nb)
  for b in 1 : nb # initialize for each block
    # compute start/stop index along 'ax'
    start = 1 + bl * (b-1)
    stop = bl + bl * (b-1)
    # ensure we are not indexing out of image
    stop = stop > cl[ax] ? cl[ax] : stop
    bstart = Vector{Int}(undef, 3)
    bstop = Vector{Int}(undef, 3)
    # create Cartesian index from start/stop
    for a in 1 : 3
      # block size along the other two dims
      # remains the same
      bstart[a] = ax == a ? start : 1
      bstop[a] = ax == a ? stop : cl[a]
    end
    bstarts[b] = CartesianIndex(bstart...)
    bstops[b] = CartesianIndex(bstop...)
    # find needed section of data for block b
    dstarts[b] = max(cf, bstarts[b] - dd)
    dstops[b] = min(cl, bstops[b] + dd)

    # allocate memory for thread-local vector
    if b == 1 || b == nb # top or bottom block
      eis[b] = Vector{Int}(undef, sb)
      ejs[b] = Vector{Int}(undef, sb)
      evds[b] = Vector{Float32}(undef, sb)
      evis[b] = Vector{Float32}(undef, sb)
    else # middle block
      eis[b] = Vector{Int}(undef, mb)
      ejs[b] = Vector{Int}(undef, mb)
      evds[b] = Vector{Float32}(undef, mb)
      evis[b] = Vector{Float32}(undef, mb)
    end
  end

  return (
    bstarts, bstops, dstarts, dstops,
    eis, ejs, evds, evis
  )
end
```

Fig. 7: Julia implementation for initializing memory and calculating offsets to achieve thread-local state.

```
function cuda_construct_kernel!(
  V, # our modified adjacency matrix
  image, R, # image, Cartesian index iterator
  cf, cl, # first and last index
  dd, # search distance as Cartesian index
  imap, # linear indices of R
  jmap, # linear indices of neighbor cube
  diff_fn # similarity function
)
  # compute global index of current thread
  # blockIdx is offset by 1, since Julia
  # arrays are indexed starting at 1
  bIdx = blockIdx()
  bDim = blockDim()
  tIdx = threadIdx()

  x = (bIdx.x-1) * bDim.x + tIdx.x
  y = (bIdx.y-1) * bDim.y + tIdx.y
  z = (bIdx.z-1) * bDim.z + tIdx.z
  I = CartesianIndex(x, y, z)

  # check if thread index is in our image
  if I in R
    @inbounds begin
      # find the first and last index of
      # the neighbors of I
      lower = Base.max(cf, I-dd)
      upper = Base.min(cl, I+dd)

      src = imap[I] # linear index at I
      pi = image[I] # voxel value at I

      # iterate over all neighbors of I
      for J in lower : upper
        dst = imap[J] # linear index of J

        # do not construct self-loops
        src == dst && continue

        pj = image[J] # voxel value at J

        # map J to linear index in jmap
        noffset = jmap[J-lower+oneunit(cf)]

        # store edge in V
        V[noffset, src] = diff_fn(pi, pj)
      end
    end
  end

  # CUDA kernels have a void return type
  nothing
end
```

Fig. 8: Julia implementation for our CUDA kernel, where each thread on the GPU computes the neighbors for a particular voxel.

that parallelizing graph construction does lead to performance gains, in most cases. Further analysis of the results can be found in Section V-E.

A. Implementation Details

These parallel graph construction methods were implemented in Julia v.1.6.0. We make use of Julia's native support for multi-threading, as well as the package CUDA.jl (available here, https://github.com/JuliaGPU/CUDA.jl). The source code for our single-threaded, multi-threaded, and CUDA implementations are publicly available at https://github.com/sinha-abhi/im2gr (along with preliminary C++ implementations for both single-threaded and multi-threaded graph construction).

B. Problem Construction and MRI Dataset

We test our Julia implementation of *im2gr* on two datasets, the first being a randomly generated "image" of size $144 \times 144 \times 22$, and the second a full body MRI scan of size $576 \times 576 \times 88$. This MRI scan was initially obtained from the 2018 Atrial Segmentation Challenge [12]. We construct graphs for these two datasets with search distances $d = 1$ and $d = 2$, as resources permit.

C. Machine Specifications and Computing Environment

Experiments were performed on two machines. Machine 1 is a computer with 1 8-core 3.6 GHz Intel Core i7-9700K processor, and 32 GB of RAM. The GPU on this machine is a NVIDIA GeForce RTX 2070 SUPER with 8 GB of on-board memory and a compute capability of 7.5. Machine 2 is a computer with 8 24-core 2.7 GHz Intel Xeon Platinum 8186 processors, and 6340 GB of RAM. These processors are equipped with Intel Hyper-Threading technology. This machine had 8 NVIDIA Tesla P100-PCIe-16GB GPUs with compute capabilities of 6.0. We made use of only one of those units at a time in these experiments. This machine was used to test our parallel implementations on the MRI dataset because we required a larger amount of memory. Our machines were not exclusively dedicated to these experiments, so runtimes may differ based on other processes occupying CPU time.

D. Results

Tables I and II show results for running our parallel graph construction code. On Machine 1, we ran our code using 1 and 8 threads on the CPU, and using our default CUDA block size on the GPU. On Machine 2, we use 1, 56, and 384 threads on the CPU(s), and use a block size of $(8, 8, 4)$ on the GPU to accommodate the Tesla P100's resource limitations.

In Table I, we see that our parallel implementations perform better than the single-threaded implementation, with the GPU implementation giving a speed up of approximately a factor 4. For $d = 1$, we find that the multi-threaded implementation leads to a speed up of roughly a factor 2.5, but for $d = 2$ that the speed up margin is a lot smaller. The results of second machine are more interesting in that they differ from the pattern of the first machine. On this machine, with distributed cores, the multi-threaded implementation actually performed much worse than using a single thread. The CUDA implementation performed better as expected, but not as well as the CUDA implementation on the first machine. These results seem to imply a much too strong (and unwanted) dependence on hardware.

TABLE I: Results for *im2gr* on Machine 1

Size	d	Mode	Threads	Mem (GB)	Time (s)
$144 \times 144 \times 22$	1	ST	1	0.273	0.122
		MT	8	0.547	0.050
		CUDA	-	0.098	0.030
$144 \times 144 \times 22$	2	ST	1	1.288	0.573
		MT	8	2.556	0.459
		CUDA	-	0.456	0.149

TABLE II: Results for *im2gr* on Machine 2

Size	d	Mode	Threads	Mem (GB)	Time (s)
$144 \times 144 \times 22$	1	ST	1	0.273	0.259
		MT	2	0.547	0.200
		MT	56	0.547	0.456
		MT	384	0.547	0.455
		CUDA	-	0.098	0.075
$144 \times 144 \times 22$	2	ST	1	1.288	1.208
		MT	2	2.556	1.604
		MT	56	2.556	2.125
		MT	384	2.556	2.117
		CUDA	-	0.456	0.353
$576 \times 576 \times 88$	1	ST	1	18.03	18.80
		MT	2	35.56	25.33
		MT	56	35.56	33.08
		MT	384	35.56	33.24
$576 \times 576 \times 88$	2	ST	1	85.84	89.90
		MT	2	171.27	160.91
		MT	56	171.27	165.26
		MT	384	171.27	153.93

E. Difference in Hardware

Though we might expect the second machine to greatly outperform the first because of its resources, we

find that the performance hit on the second machine is actually unsurprising. In this section, we briefly highlight the contributing factors to these differences.

1) Multi-thread: On large distributed machines like Machine 2, we find that Non-Uniform Memory Access (NUMA) has profound effects on the the performance of multi-threaded code, especially code that relies on heavy memory accesses and allocations. Though all our data is stored in RAM, some of that may be present on a different NUMA node than the current CPU. The CPU must then fetch that memory from that NUMA node. In cache-coherent NUMA machines, this memory access can take a few hundred clock cycles depending on the NUMA distance between the nodes [9]. The interconnection architecture between NUMA nodes is crucial to the performance of multi-threaded algorithms in such large machines. This suggests that a fourth mode is needed for *im2gr* in which we make use of distributed cores efficiently.

From profiling, we also notice that much of the time spent constructing the graph for the MRI scan when $d = 2$ is in garbage collection. For this graph that has roughly 3.5 billion edges, about 57% of the time was spent in garbage collection. It is a known issue that Julia's garbage collector and allocation heavy multi-threaded code do not synergize well.

2) CUDA: The difference in performance here is perhaps the least surprising since the architecture of GPUs is rapidly changing. Thus, generational gaps between hardware generally leads to much different results. Tim Besard, the maintainer of CUDA.jl, and I had an interesting discussion about these results, in which he highlighted that some of this variance can be attributed to CUDA wrapper for Julia itself (CUDA.jl). Newer generations of GPUs perform better with the more general code emitted by CUDA.jl. General code, in the sense that CUDA.jl does not currently optimize for pointer chasing, integer conversions (Julia uses 64-bit integers, while CUDA's are intrinsically 32-bit), and unlikely branches (such as from exceptions). All of these operations are performance killers on older GPUs.

VI. Related Work

Our work presented in this paper is related to other graph construction methods in image classification, graph clustering, and hyperspectral image analysis through semi-supervised learning (SSL) [11], [7]. In our implementation, we followed a procedure tailored to local graph clustering by establishing neighbor relationships via search distance. There exist more general methods to create higher dimensional k-NN (k nearest neighbors) graphs that aim for approximate, but still accurate, neighborhood graphs. There are also construction methods that approach this problem via manifold learning. These methods aim to create graphs suitable for SSL by embedding geometric properties between neighborhoods within edge weights.

VII. Discussion and Future Work

In this paper we have provided a starting point for parallelizing graph construction through multi-threading and CUDA programming. The large size of these graphs makes the simple iterative approach time consuming and memory intensive in most applications. We showed that parallelizing these methods do lead to performance gains in basic computing environments.

In future work, we will improve on the performance aspects of this construction process, including more careful memory management on distributed systems and CUDA-equipped devices. This can be achieved by introducing methods targeted toward multiprocessor machines. We will also consider new methods for constructing extremely large graphs. It is not be feasible to always preallocate memory for the entire graph. Even with the large amount of memory present on Machine 2, for some search distances (like $d = 20$), we cannot preallocate memory for the entire graph. Computing a bound like in Section III-A, we find the LGE MRI scan with a search distance of $d = 20$ would have approximately 2 trillion edges. This equates to

$$2 \cdot 2e12 \cdot (8 + 4) = 4.8e13 \text{ bytes} = 48 \text{ TB}$$

of working memory (an `Int` is 64 bits and a `Float32` is 32 bits in Julia). A potential approach is to construct the graph in batches, and use files to incrementally store progress. Finally, we will explore methods of local graph construction for applications in which an entire graph may not be required for effective analysis.

References

[1] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, 2003.

[2] Jeff Bezanson, Jameson Nash, and Kiran Pamnany. Announcing composable multi-threaded parallelism in julia. https://julialang.org/blog/2019/07/multithreading, July 2019.

[3] K. Fountoulakis, M. Liu, D. F. Gleich, and M. W. Mahoney. Flow-based algorithms for improving clusters: A unifying framework, software, and performance. *arXiv*, cs.LG:2004.09608, 2020.

[4] Herve Jegou, Cordelia Schmid, Hedi Harzallah, and Jakob Verbeek. Accurate image search using the contextual dissimilarity measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):2–11, 2010.

[5] JuliaLang. julia. https://github.com/JuliaLang/julia/blob/master/src/array.c, 2021. v1.6.0: src/array.c:904.

[6] JuliaLang. *The Julia Language*, 2021.

[7] Li Ma, Melba M. Crawford, Xiaoquan Yang, and Yan Guo. Local-manifold-learning-based graph construction for semisupervised hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 53(5):2832–2844, 2015.

[8] Thomas B. Sebastian and Benjamin B. Kimia. Metric-based shape retrieval in large databases. In *Proceedings of the 16 Th International Conference on Pattern Recognition (ICPR'02) Volume 3*, ICPR '02, page 30291, USA, 2002. IEEE Computer Society.

[9] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 80–91, 1992.

[10] Nate Veldt, Christine Klymko, and David F. Gleich. Flow-based local graph clustering with better seed set inclusion. *Proceedings of the 2019 SIAM International Conference on Data Mining*, page 378–386, 2019.

[11] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. Scalable k-nn graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1106–1113, 2012.

[12] Jichao Zhao and Zhaohan Xiong. 2018 atrial segmentation challenge. http://atriaseg2018.cardiacatlas.org/, 2018.