Parallel Image Inpainting

15-418/618 Parallel Computer Architecture

Saileshwar Karthik (saileshk)
Ayushi Bansal (ayushib)

URL: https://astroskape.github.io/parallel-inpainting/

Summary
Image inpainting is a technique that can reconstruct missing regions in an image using the pixel values of remaining regions within the image so that the result looks natural. We aim to parallelise one such inpainting algorithm and analyze the different tradeoffs and bottlenecks.

Background
There are different approaches to performing image inpainting, some of which are:

1. Partial Differential Equation (PDE) based - The missing regions of an image are filled through a diffusion process which smoothly propagates information from the boundary towards the interior of the missing region. The simplest form uses the Laplace equation, averaging the four direct neighbors iteratively until convergence.
2. Exemplar based - The missing regions are filled in by copying patches of content from the existing parts of an image to maintain structural and textural components. The gap reconstruction is done serially by picking a tile which has a high confidence score (ratio of filled pixels to unfilled) and a high data score (favours edges).
3. Nearest neighbor field (NNF) based - Similar to exemplar based, but instead of filling one patch at a time according to a greedy order based on the priority values, a global nearest neighbor field is computed that assigns every patch in the image an offset pointing to its best matching patch elsewhere in the image.

Criminisi - An exemplar-based inpainting algorithm that fills large missing regions by copying patches from elsewhere in the image. The algorithm has three main components:

1. Priority computation - Each patch on the boundary of the hole is assigned a priority score that determines the fill order. The priority is the product of two terms:
    1. Confidence term C(p): Measures how much of the patch consists of known, original pixels. Patches on the outer boundary of the hole with more original neighbors get higher confidence. Patches deeper in the hole with fewer known neighbors get lower confidence. This enforces an outside-in fill order similar to onion peeling.

    2. Data term D(p): Measures the strength of image edges flowing into the patch from outside the hole. Patches that lie on the continuation of strong edges get boosted priority, ensuring edges are propagated into the hole before surrounding texture fills in around them.
2. Patch search and copy - The highest priority boundary patch is selected and the entire source region is searched for the most visually similar patch using sum of squared differences (SSD). The best matching patch is copied into the hole, filling in both texture and structure in one step.
3. Confidence update - After a patch is filled, the confidence values of the newly filled pixels are updated to reflect the confidence of the patch that filled them. Confidence decays as filling progresses inward, reflecting that synthesized pixels are less reliable than original ones. The boundary of the hole shrinks and the process repeats until the hole is fully filled.

PatchMatch - An algorithm that is based on the NNF approach. The algorithm has three main components:

1. Initialization - the NNF is initialised with random offsets for each patch.
2. Iterative refinement - Each iteration either scans from top-left to bottom right or bottom-right to top-left depending on the iteration parity. For each patch f(x, y) in scan order, two operations are applied.
    a. Propagation: The current offset f(x, y) is compared against the offsets of its scan direction neighbors f(x-1, y) and f(x, y-1) (or f(x+1, y), f(x, y+1) when doing reverse scan). The offset yielding lowest SSD (Sum of squared differences) for the current patch is kept. The insight here being that natural coherence in imagery allows neighboring patches to have similar matches.
    b. Random search: Attempts to improve the current offset f(x,y) by testing a sequence of candidate offsets at an exponentially decreasing distance from the current patch. This allows the algorithm to discover better matches than what the propagation phase alone can find.
3. Patch voting and reconstruction - At this stage, each hole pixel is assigned a final color value. This final pixel value is computed as a weighted average of the pixel values pointed to by the NN offsets computed for each of the overlapping patches, where better-matching patches receive higher weight.

Both algorithms present unique and interesting parallelisation challenges. We plan to explore parallelization strategies within PatchMatch in particular.

The Challenge

For PatchMatch:

1. There are clear data dependencies in the propagation phase which will be difficult to parallelise. Identifying approaches that ensure computational speedup and fast convergence while ensuring the image's visual quality will be a challenge. We could

explore strategies like red-black ordering (for the propagation phase) and jump-flood schemes (for the random search phase)
2. Random search has divergent execution. The search termination depends on how soon a good match could be found for a given patch. This phase also has bad spatial locality across patch searches - the patch SSD computation at each random probe reads a full patch at a random location in the image while consecutive pixel probes point to completely different patches (and location in the image).
3. There is load imbalance and this imbalance varies as the EM (Expectation-Maximization) iterations progress. The pixels adjacent to the hole boundary have many adjacent non-masked pixels to compare against and will converge to good matches quickly as opposed to pixels deep inside holes.
4. The reconstruction phase is susceptible to write conflicts. Multiple processors computing different patch contributions may try to write to the same pixel simultaneously. This will add an additional synchronization overhead.

Resources

https://github.com/CDOrtona/Image_Inpainting
For the initial implementation and debugging, we will be using the GHC machines. Once we have refined our approach, we will run benchmark tests on PSC machines to test against a larger number of compute cores.

PatchMatch:
We will develop the starting code from scratch while referring to the original PatchMatch paper and looking at online resources for the sequential implementation of the same

References
https://gfx.cs.princeton.edu/pubs/Barnes_2009_PAR/patchmatch.pdf
Sequential code reference: https://github.com/younesse-cv/PatchMatch

Goals and Deliverables:

Plan to Achieve:

1. Implement a parallelised version of an image inpainting algorithm using OpenMP and perform extensive analysis on
    a. how hole size affects performance
    b. how different propagation strategies can affect convergence rate and output quality
    c. how patch size affects the communication to computation ratio
2. Achieve a computational speedup of at least 4x on GHC machines (8 cores)
3. Build an application that allows users to submit images with holes in them and have it go through our inpainting algorithm and display the reconstructed image

Hope to Achieve:

1. Allow users to paint over any image of their choice to apply the masks

2. Implement a parallelised version using CUDA

Platform Choice

We will primarily require access to the GHC Clusters for development and the PSC machines to run large-scale experiments. This is well justified for the following reasons:

1. **HPC Computing Resources** - The PSC machines provide access to systems with large numbers of processing cores which is essential for testing the scalability of our parallel algorithm. At this stage we are not sure how computationally heavy the algorithm might be, but we would like to test this out.
2. **Language and Framework support** - We plan to use C++ and OpenMP primarily (maybe even CUDA if we reach our stretch goals). These frameworks are readily available on these platforms

Schedule

Mar 26, 2026 - Apr 1, 2026

Checkpoint 1 - Project Setup and Literature Survey

- Read more papers and do more research on ways to parallelize the problem
- Set up a code repo with starter code
- Find out the C++ libraries required to setup the sequential version of the algorithm
- Start work on the sequential version

Apr 2, 2026 - Apr 8, 2026

Checkpoint 2 - Sequential Implementation

- Complete the sequential version of the code in C++
- Verify correctness of the solution

Apr 9, 2026 - Apr 15, 2026

Checkpoint 3 - Project Milestone

- Get a basic parallel version of the code up and running
- Start with the OpenMP implementation
- Start building a simple UI to be able to submit images (with holes) and view reconstructed image post inpainting
- Finish up with the Project Milestone Report

Apr 16, 2026 - Apr 21, 2026

Checkpoint 4 - Parallelization with OpenMP

- Complete the OpenMP implementation
- Complete the UI work
- Start benchmarking the algorithm and different approaches taken on PSC
- See if any of the stretch goals can be achieved

Apr 22, 2026 - Apr 30, 2026

Checkpoint 5 - Final Report Preparation

- Prepare the final report and presentation
- Include graphs, charts and comparisons
- Address any remaining issues that could improve performance
- Prepare for the poster session