Parallel Image Upscaling
using Bilinear Interpolation

By - Siddhant Gupta (sgupta54)
Under guidance of instructor - Dr. R. Miller

Image Scaling

- Resizing to a smaller (downscaling/downsampling) or bigger size (upscaling/interpolation)
- Interpolation involves estimation of unknown data based on the known data
- Interpolation is generally performed by fitting the known data points to a function and using the function to estimate the unknowns

Image interpolation algorithms

- Nearest Neighbor: Copies the value of nearest neighbour
- Bilinear: uses linear function, 2x2 pixel grid
- Bicubic: uses cubic function, 4x4 pixel grid
- Spline: eg. Cubic-B spline, Catmull-Rom spline
- Lanczos: uses sinc function
- Machine learning: eg. NVIDIA’s DLSS

Comparing interpolation algorithms

Original

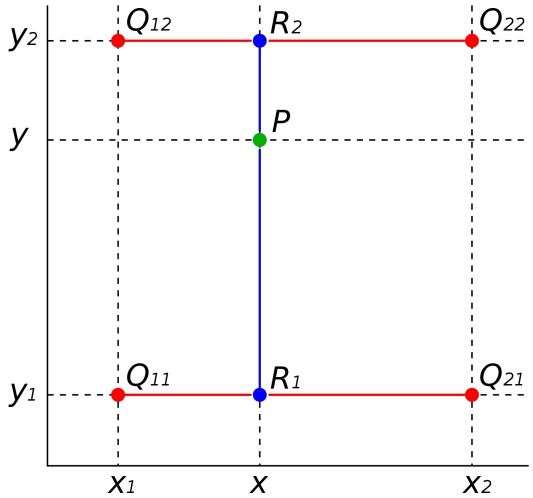
Nearest neighbor

Bilinear

Bicubic

Wiki Wiki Wiki Wiki

Bilinear Interpolation



$$f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}),$$

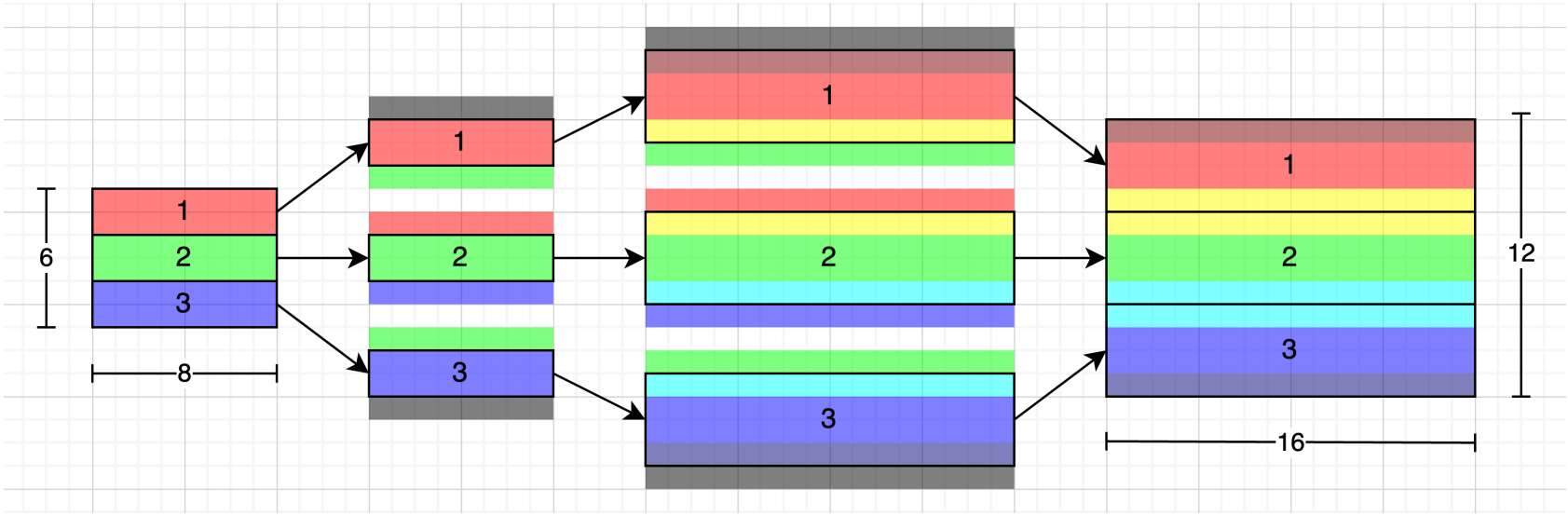
$$f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}).$$

$$f(x, y) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

Parallel Algorithm: domain decomposition

1. Decompose image into domains
2. Scatter domains to the processors
3. Exchange the boundary halos between consecutive domains
4. Simultaneously and in parallel perform the local interpolation on each domain
5. Gather local results into a final output.

Parallel Algorithm



Scatter step

```c
// Compute the local subdomain size
// Split across height; horizontally; each processor stores 'sub_height' rows
int sub_width = width;
int sub_height = height / size;

// Allocate memory for the local subdomain and halo cells
uint8_t *sub_image = calloc(sub_width * (sub_height + 2), sizeof(uint8_t)); // +2 for halos

// Scatter the image to all processes
MPI_Scatter(image,
            sub_width * sub_height,
            MPI_UINT8_T,
            (sub_image + sub_width), // leave first row empty
            sub_width * sub_height,
            MPI_UINT8_T,
            0,
            MPI_COMM_WORLD);
```

Halo exchange

```
// Exchange halo with adjacent processes
if (rank > 0)
{
    // Receive up halo from process (rank - 1)
    MPI_Recv(sub_image, sub_width, MPI_UINT8_T, rank - 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // Send up halo to process (rank - 1)
    MPI_Send(sub_image + sub_width, sub_width, MPI_UINT8_T, rank - 1, 0, MPI_COMM_WORLD);
}
if (rank < size - 1)
{
    // Send down halo to process (rank + 1)
    MPI_Send(sub_image + sub_height * sub_width, sub_width, MPI_UINT8_T, rank + 1, 1, MPI_COMM_WORLD);
    // Receive down halo from process (rank + 1)
    MPI_Recv(sub_image + (sub_height + 1) * sub_width, sub_width, MPI_UINT8_T, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Local interpolation algorithm

```
int sub_out_width = sub_width * 2;
int sub_out_height = sub_height * 2;
uint8_t *sub_out_image = calloc(sub_out_width * (sub_out_height + 2), sizeof(uint8_t)); // +2 to account for the halos

// Perform the local bilinear interpolation
for (int y = 0; y < sub_out_height + 2; y++)
{
    for (int x = 0; x < sub_out_width; x++)
    {
        float X = x / 2.0;
        float Y = y / 2.0;

        // Get the vertices from original sub_image to interpolate between
        int x0 = (int)X;
        int y0 = (int)Y;
        int x1 = x0 + 1;
        int y1 = y0 + 1;

        // Get the pixel values for these vertices
        uint8_t v00 = *(sub_image + sub_width * y0 + x0);
        uint8_t v01 = *(sub_image + sub_width * y1 + x0);
        uint8_t v10 = *(sub_image + sub_width * y0 + x1);
        uint8_t v11 = *(sub_image + sub_width * y1 + x1);

        float xfrac = X - x0;
        float yfrac = Y - y0;

        *(sub_out_image + sub_out_width * y + x) = lerp2D(v00, v01, v10, v11, xfrac, yfrac);
    }
}
```

Interpolation function

```
// 2D linear interpolation function
float lerp2D(uint8_t v00, uint8_t v01, uint8_t v10, uint8_t v11, float xfrac, float yfrac)
{
    float a = v00 * (1 - xfrac) * (1 - yfrac);
    float b = v01 * (1 - xfrac) * yfrac;
    float c = v10 * xfrac * (1 - yfrac);
    float d = v11 * xfrac * yfrac;
    return (uint8_t)a + b + c + d;
}
```

Gather step

```
int out_width = width * 2;
int out_height = height * 2;
uint8_t *out_image;
if (rank == 0)
{
    out_image = calloc(out_width * out_height, sizeof(uint8_t));
}

if (debug)
    printf("(rank %d) Debug: Local bilerp complete \n", rank);

MPI_Gather(sub_out_image + sub_out_width,
           sub_out_width * sub_out_height,
           MPI_UINT8_T,
           out_image,
           sub_out_width * sub_out_height,
           MPI_UINT8_T,
           0,
           MPI_COMM_WORLD);
```

Limitations for reducing complexity

- Grayscale images only
- Scaling factor 2x only
- Depth of halo = 1 pixel (since using bilerp)
- Domain split only across rows
- Not considering the topology of processors
- Image loading and saving times are not considered
- Images are stored as space separated pixel values in text

Results:

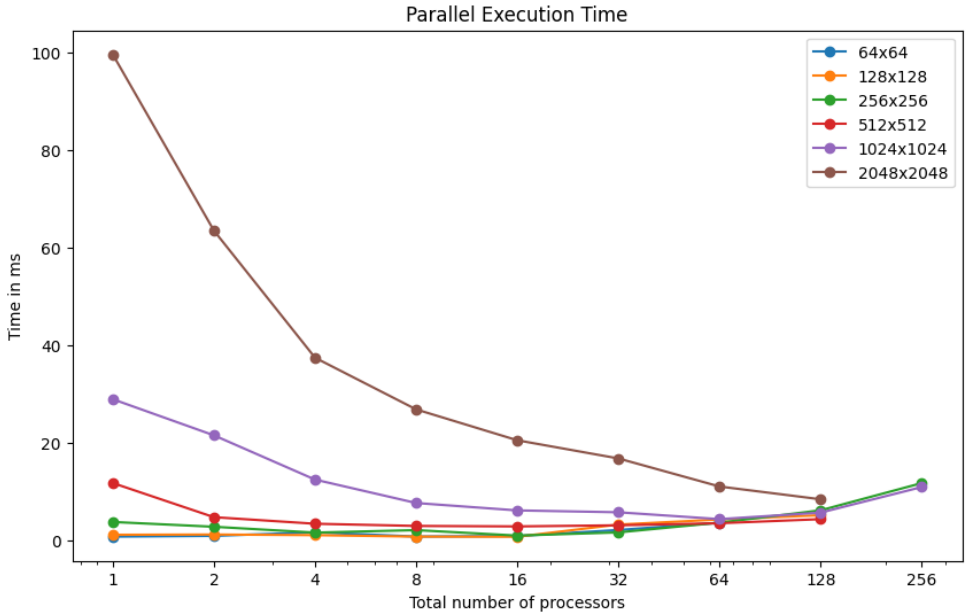
Original 512x512 image

Resized down to 256x256 then back to 512x512

Results:

<manually specifying # of nodes>

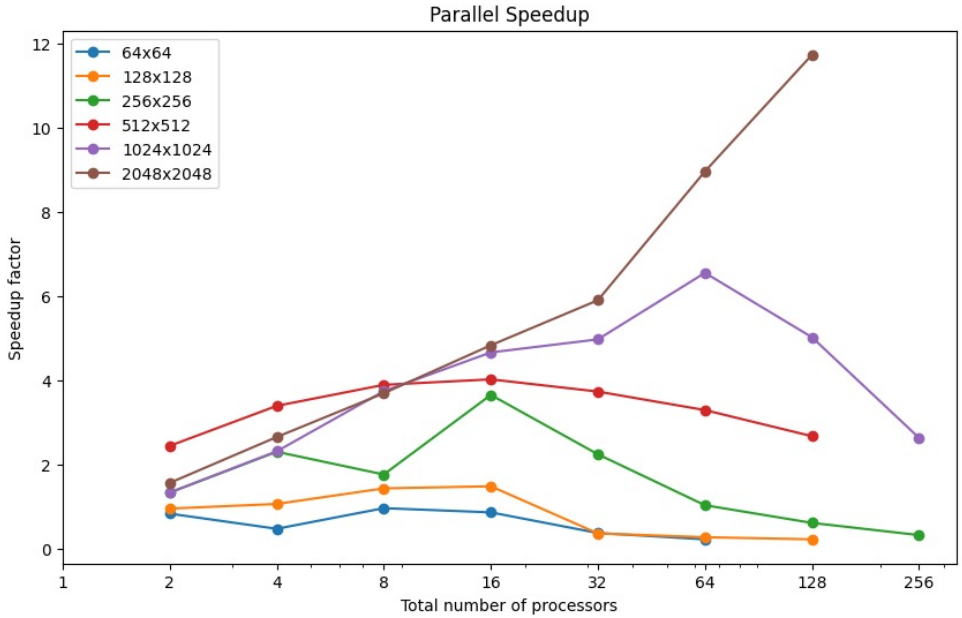
		Time taken in ms					
nodes	**ntasks-per-node**	**64x64**	**128x128**	**256x256**	**512x512**	**1024x1024**	**2048x2048**
1	1	0.834	1.213	3.837	11.807	28.970	99.548
1	2	0.989	1.270	2.855	4.817	21.573	63.484
1	4	1.723	1.137	1.661	3.474	12.513	37.464
1	8	0.857	0.841	2.166	3.025	7.717	26.883
1	16	0.955	0.816	1.047	2.927	6.203	20.563
1	32	2.198	3.306	1.704	3.155	5.820	16.842
2	32	3.682	4.343	3.700	3.582	4.416	11.098
4	32	-	5.224	6.227	4.412	5.755	8.480
8	32	-	-	11.800	-	10.991	-



Results:

<manually specifying # of nodes>

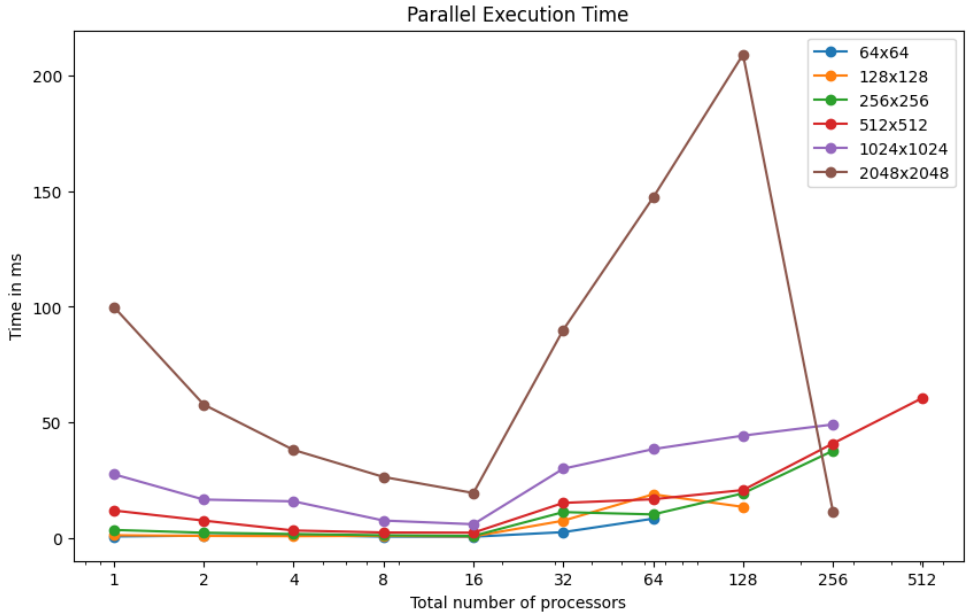
		Parallel Speedup					
nodes	ntasks-per-node	64x64	128x128	256x256	512x512	1024x1024	2048x2048
1	1	1.00	1.00	1.00	1.00	1.00	1.00
1	2	0.84	0.96	1.34	2.45	1.34	1.57
1	4	0.48	1.07	2.31	3.40	2.32	2.66
1	8	0.97	1.44	1.77	3.90	3.75	3.70
1	16	0.87	1.49	3.66	4.03	4.67	4.84
1	32	0.38	0.37	2.25	3.74	4.98	5.91
2	32	0.23	0.28	1.04	3.30	6.56	8.97
4	32	-	0.23	0.62	2.68	5.03	11.74
8	32	-	-	0.33	-	2.64	-



Results:

<automatically assigned # of nodes>

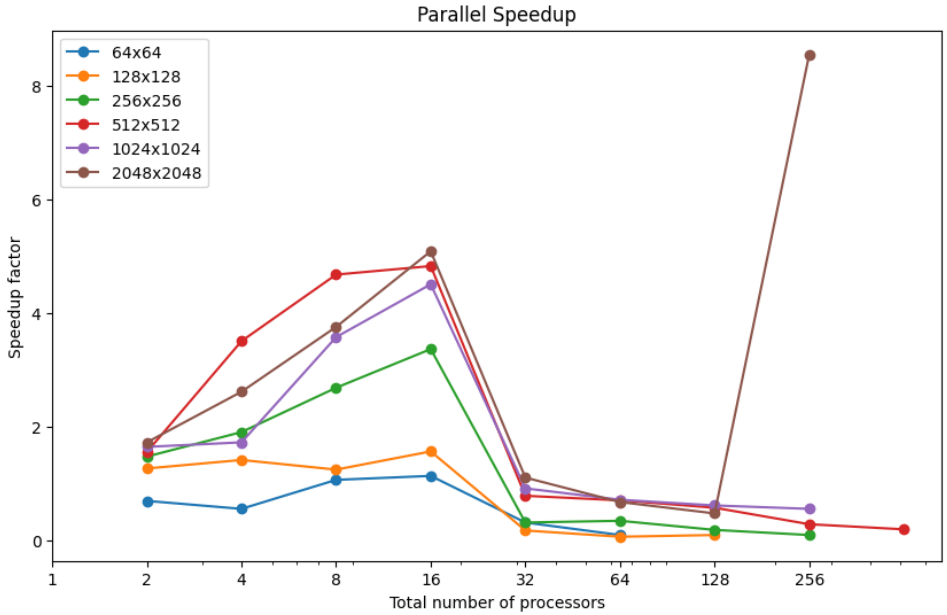
		Time taken in ms					
(# nodes received)	ntasks	64x64	128x128	256x256	512x512	1024x1024	2048x2048
1	1	0.864	1.404	3.683	12.079	27.717	99.954
1	2	1.236	1.105	2.482	7.720	16.790	57.814
1	4	1.539	0.990	1.924	3.445	16.029	38.214
1	8	0.807	1.123	1.368	2.580	7.739	26.554
2	16	0.756	0.892	1.094	2.503	6.151	19.627
2	32	2.729	7.681	11.387	15.353	30.116	90.051
4	64	8.547	19.053	10.377	16.936	38.570	147.180
8	128	-	13.629	19.483	20.900	44.392	208.889
16	256	-	-	37.849	41.001	49.210	11.693
32	512	-	-	-	60.732	-	-



Results:

<automatically assigned # of nodes>

		Parallel Speedup					
(# nodes received)	ntasks	64x64	128x128	256x256	512x512	1024x1024	2048x2048
1	1	1.00	1.00	1.00	1.00	1.00	1.00
1	2	0.70	1.27	1.48	1.56	1.65	1.73
1	4	0.56	1.42	1.91	3.51	1.73	2.62
1	8	1.07	1.25	2.69	4.68	3.58	3.76
2	16	1.14	1.57	3.37	4.83	4.51	5.09
2	32	0.32	0.18	0.32	0.79	0.92	1.11
4	64	0.10	0.07	0.35	0.71	0.72	0.68
8	128	-	0.10	0.19	0.58	0.62	0.48
16	256	-	-	0.10	0.29	0.56	8.55
32	512	-	-	-	0.20	-	-



Further experimentation

1. Fractional scaling factor
2. Alternate interpolation algorithms.
3. Using advanced features of MPI to improve performance. Eg. setting up mesh/cartesian topology of nodes
4. Looking into CUDA implementation and comparison

References

- http://www.archer.ac.uk/training/course-material/2014/07/MPI_Edi/Exercises/MPP-casestudy.pdf
- https://ubccr.freshdesk.com/support/solutions/articles/13000026245-tutorials-workshops-and-training-documents
- https://theailearner.com/2018/12/29/image-processing-bilinear-interpolation/
- https://en.wikipedia.org/wiki/Image_scaling