Final Project

CSE 597

Amatur Rahman

Friday, December 7, 2018

Abstract

The idea of relating Poisson equation into image processing was motivated from [15]. Image blending is one of the important techniques in image processing. In this project, we want to seamlessly cut and paste a source image on a target image, where the source and target are visually different [16]. This problem falls in the category of gradient domain processing of image, where a gradient vector corresponds to the change of pixel value. The process is computationally expensive, more so in case of gigabyte sized images. So sequential algorithms do not scale well to solve these problems. There comes a need for more computer resources. Therefore parallel programming is greatly useful to provide such compute power. In this report, one parallelizing method of solving the Poisson Image Blending problem will be presented using MPI and another one using GPU programming with CUDA coupled with cuBLAS library.

We formulated the problem as a classical Ax=b problem in linear algebra. Our A matrix is sparse, so indirect solver Jacobi does better. We chose jacobi as a starting iterative solver, because of its highly parallel data structure. Sparse matrices do not generally have sparse LU decomposition, so it gets harder to fit the matrices in memory as the problem size goes larger. For Jacobi method, we store the A matrix in sparse representation. We modified the residual computation of the initial Jacobi code to make the code faster. After optimizing Jacobi in its serial implementation, we tried to improve its performance by parallelizing it using MPI. For Image size = 40 x 40 and 10 processors, this gives us 8x speedup. For making it even faster, we coupled our code to thrust and cuBLAS library. After adding GPU kernel code and library coupling, we get 5x speedup over a parallel MPI implementation with 8 processors.

1 Problem of Interest

1.1 The problem

In this report, the problem that we are trying to solve is from two sources of similar images, how to blend them in the most efficient way. We are finding out the blended image using Poisson solver, which is an important component of an advanced image editing application. In Poisson image blending, we use gradient domain operations at first to reduce visible discontinuities across seams. Then to convert the image back to original domain, we need to solve Poisson equation. For that, an iterative solver (Jacobi) is used.

1.2 General overview

Poisson image blending is a process of inserting a source image into a target image [11]. Here two image of similar appearance are stitched along a seam to give the effect of smooth blending as in Fig 1.3. The seam is not noticeable as we convert the image to gradient domain using image processing libraries like [8], do the computations of stitching and generate the blended image (the rightmost one) using inverse gradient transform. To do the inverse gradient transform, we have solve a Poisson equation. This is the memory and time intensive part of our project, so making efficient use of resource is a must here.

1.3 Scientific merit

We have Source image S, and a target image T, we want to seamlessly paste the source image to target so that it matches its surroundings and does not stand out. Say f(x, y) is the pixel value at (x, y) coordinate of the pasted image.

Now we want to optimize

$$\min_f \iint_\Omega |\nabla f - \mathbf{v}|^2 \text{ with } \; f|_{\partial\Omega} = f^*|\partial\Omega$$

Where v is the gradient of a region in an image, g is a selected region of source, which for simplification we assumed the whole picture, $f*$ is a set of known functions that exist in domain S (source image), f is an

unknown function that exist in domain Ω, Ω is the region g after placing it on source image. $\partial\Omega$ are the boundaries between the source and target regions.

So given a vector field, we want to find the value of f in unknown region that optimize.

Now, the solution of the above equation is the solution of Poisson equation:

$$\Delta f = \text{ div v over } \Omega, \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega} \tag{1}$$

Here,

$$\text{div } \mathbf{v} = \partial v/\partial x + \partial v/\partial y \tag{2}$$

Δ is the Laplacian operator:

So we get the formulation of divergence using

$$divG = -4f(X,y) + f(X-1,y) + f(X,y-1) + f(X+1,y) + f(X,y+1) \tag{3}$$

Now we get the b matrix by,

$$b[i] = div(G(Source(x,y))) + Neighbor(target[i])$$

for all i = 1 to N

where N is the problem size. If we deal with 100x100 pixel image N = 100.

The equations were adopted from [3].

A is a matrix of size NxN, which is obtained by discretizing Poisson 2D equation. It is a sparse matrix having -4 in diagonals, and 1 in off-diagonals.

To get the test problem of B vector from image, In order to generate the test problem, we cut and pasted a source image in grayscale from to the target image, and then applied the blending process using the equation. We tested our solver with images of 100x100 pixels. All the other scalability tests were done on a randomly generated B vector.

We want to be able to solve the seamless Poisson image blending for gigapixel images. So there would $10^6 x 10^6$ pixels in production problem. Now, for image processing applications, depending on the field where it is used, we will almost always have some pre-defined level of clearness of the image that matters to us. For example, if we are stitching up large panoramic images for a video game that is played in mobile phone, we would require less quality of image. In that case, we can terminate our iteration early, when the difference is around the order of tenth of magnitude of pixels. On the other hand, for generating images for movie production, this error tolerance needs to be more precise. For simplification, we take the pixel values of RGB channel and convert them to grayscale to get single float value of pixel intensity. Then we blend the grayscale channel separately, using the formulas discussed.

Figure 1: Overview of image blending problem [2]

1.4 Relevant fields

Mainly Digital Image Processing, Data augmentation in deep neural networks, Medical Imaging.

Poisson image processing and image blending has wide scale application in motion and object detection. Whenever a pattern matching or machine learning algorithm is used, a large number of data is required to train the machine learning algorithm. Large scale image processing is also required in X-ray image stitching [14]. When the image size gets larger, say around 1000000 pixels, it gets computationally unfeasible to solve using sequential methods. So a numerical parallel approach is worthy of doing.

1.5 Other methods used to solve Poisson Image Blending

Jacobi iterative method that we are using has the advantage of allowing termination when a pre-specified level of accuracy has been reached. However other methods exist too.

Table 1: Direct Methods

Method	**Complexity**
Gaussian Elimination	N^3
LU Decomposition	N^3
FFT-based	N log N

Table 2: Indirect Methods

Method	**Complexity**
Jacobi	N^2
SOR	$N^{\frac{3}{2}}$
Conugate Gradient	$N^{\frac{3}{2}}$
Multigrid	N

1.6 Discussion of known solution (analytic/published)

Conjugate Gradient (CG) algorithms typically exhibit much faster convergence, so it is almost always used with a preconditioner, and preconditioned CG (PCG) usually requires many fewer iterations than Jacobi to reach the same accuracy. But the preconditioning is hard to do.

Typically Fast Fourier Transform is used to solve image blinding [7]. While there are many sequential algorithms implemented in practice, parallel implementation studies is quite limited. In [17], a simple framework for the parallel gradient domain processing is given for gigapixel images. Methods exist to find a direct Poisson solution using Fast Fourier Transforms (FFT) [13]. FFT methods can make use of parallel computation. However, these methods have not yet been tested in modern out-of-core gradient domain image processing.

Direct techniques have a much higher cost. For example, Cholesky decomposition and Gaussian elimination has runtime complexity $O(n^3)$, where n is the number of pixels. When n becomes large, such direct solvers require too much memory and computational becomes impractical.

Often the Poisson problem is simplified by discretization into a large linear system whose dimension is typically the number of pixels in an image. Methods exist to find a direct solution to this linear system.

Often, especially in distributed systems, it is simpler to implement an iterative method to find a solution [18, 6]. However for larger linear systems, memory consumption is a limiting factor and iterative methods such as Jacobi is preferred.

1.7 Numerical Set-up

Now this yields a system of linear equations The boundary conditions are Dirichlet.

- **How the A-matrix and b-vector are formed**

We have Source image S, and a target image T, we want to seamlessly paste the source image to target so that it matches its surroundings and does not stand out. Say f(x, y) is the pixel value at (x, y) coordinate of the pasted image.

Now we want to optimize

$$\min_f \iint_\Omega |\nabla f - \mathbf{v}|^2 \text{ with } f|_{\partial\Omega} = f^*|\partial\Omega$$

Where v is the gradient of a region in an image, g is a selected region of source, which for simplification we assumed the whole picture, $f*$ is a set of known functions that exist in domain S (source image), f is an unknown function that exist in domain Ω, Ω is the region g after placing it on source image. $\partial\Omega$ are the boundaries between the source and target regions.

So given a vector field, we want to find the value of f in unknown region that optimize.

Now, the solution of the above equation is the solution of Poisson equation:

$$\Delta f = \text{ div v over } \Omega, \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega} \tag{4}$$

Here,

$$\text{div}\,\mathbf{v} = \partial v/\partial x + \partial v/\partial y \tag{5}$$

Δ is the Laplacian operator:

So we get the formulation of divergence using

$$divG = -4f(X,y) + f(X-1,y) + f(X,y-1) + f(X+1,y) + f(X,y+1) \tag{6}$$

Now we get the b matrix by,

$$b[i] = div(G(Source(x,y))) + Neighbor(target[i])$$

for all i = 1 to N

where N is the problem size. If we deal with 100x100 pixel image N = 100.

The equations were adopted from [3].

A is a matrix of size NxN, which is obtained by discretizing Poisson 2D equation. It is a sparse matrix having -4 in diagonals, and 1 in off-diagonals.

To get the test problem of B vector from image, In order to generate the test problem, we cut and pasted a source image in grayscale from to the target image, and then applied the blending process using the equation. We tested our solver with images of 100x100 pixels. All the other scalability tests were done on a randomly generated B vector.

- **Problem sizes for problems of interest:**

  We want to be able to solve the seamless image blending for gigapixel images. So there would $10^6 x 10^6$ pixels in production problem.

- **Discretization method and Boundary conditions:**

  Poisson's equation is discretized by mapping it into a finite differenced spatial grid. From that, we obtain the results in the Equation 6. We use Dirichlet condition, where all boundary values are zero.

2 Solvers

2.1 Direct Solver

- **Direct solver being used:**

LU Decomposition.

- **Justification for direct solver being used:**

  Since most of the element of the A matrix is zero, doing the backfill in LU decompostion is computationally inexpensive than other direct solvers like Cramer's method.

  In LU factorization, we can separate the factorization and solving for x in two steps. Since the image processing application we are working on keeps A unchanged and is supposed to apply repetitively to different datasets, this decoupling saves time. Because factorization itself is the most expensive computation (O(N^3)). This only requires matrix A. Once this is done, this factored A can be used with different B matrices to compute at a faster rate, because the cost of solving for x is only O(N^2) which is way less than the cost of decompisition as seen by our analysis to follow. For larger problem size, this effect is even more pronounced. [1]

- **List of and justification for optimization flags**

  We are using "O2" flags because they needed 0.007 seconds on average, whereas other take 0.015 seconds. It massively improves LU decomposition implementation too.

- **Timing for test problem**

  - For a problem size of 1000 pixels:
    Total CPU ticks for forward substitution: 1561
    Total CPU ticks for backward substitution: 1625
    Total CPU ticks for LU decomposition: 1287851

- **Projection of time for production problem:** In production, we will be using gigapixels of image. Considering N = 1000000 (that results in N*N = 1gigapixel), we can obtain time for production problem by multiplying the results of problem size of 1000 pixels by 1000. (2/3ncube)

  Eliminating the first column will require n additions and n multiplications for n1 rows. Therefore, the number of operations for the first column is 2n(n1). For the second column, we have n1 additions and n1 multiplications, and we do this for (n2) rows giving us 2(n1)(n2). Therefore, the total number of operations required for the full decomposition can be written as O($\frac{2}{3}n^3$). For N = 1000, time 0.270089 seconds. Let $\frac{2}{3} \times n^3 = 0.270089$. From this we calculate n. Now, if this n is multiplied by 1000, we get the projected time for gigapixels image, which is approx. 7716 days! This is not feasible computationally.

- **Memory being allocated for test problem:**

  Maximum resident set size (kbytes): 63896 (memory allocated in RAM)

  Number of page faults: 16016

- **Comparison with memory being used by task** Maximum resident set size when no method is running is 32492 kbytes, whereas LU decomposition takes extra 63896 Kbytes in memory.

- **Projection of memory required for production problem:**

  In production, we will be needing gigapixels of image. Considering integer size to be 2 bytes and floating point size to be 8 bytes. For N = 1000000, memory for A is 1000000*1000000*2 bytes, for B, x and intermediate matrix is 1000000*8000000*3 bytes. So that sums up to 22400 MBytes.

2.2 Iterative Solver

In iterative method error is decreased in each iteration. We ran the solver for maximum 2*N^2 iteration, or when the error is significantly small (error tolerance 1e-.04).

- **Iterative solver being used:** Jacobi
- **Justification for iterative solver being used** The linear system in discussion has a strictly diagonally dominant A matrix. Jacobi method converges when we have a strictly diagonally dominant matrix. In this case, since we are discretizing 2D Poissons equation, we get a diagonally dominant A matrix. Thats why Jacobi method was chosen. The matrix is also positive definite. There are other methods such as Gauss-Siedel, Successive Over-relaxation which are also applicable in this context. However, our final goal in this project is efficient parallelization, and it is achievable quickly and easily using Jacobi. This is because Jacobi algorithm does calculations of rows separately. This can be exploited in both multicore-programming and accelerators. Gauss-Siedel performs poorly in such case because of the row dependencies.
- **Justification for convergence criteria, and residual/norm being used:**

  From the analysis of 2D Poisson equation in [18], it can be proved that Jacobi method converges in $O(N^2)$ steps. So we chose maximum iteration number to be in that order.
- **List of and justification for optimization flags:**

  We are using "O2" flags because it needed 0.007 seconds on average, whereas other take 0.015 seconds. It massively improves my Jacobi implementation too.
- **Timing for test problem for 3 different initialization methods (random, good guess, all zeros or ones):**

  Random: 51.902 ms
  All Zeroes: 54.456 ms
  Good Guess: 46.046 ms

  Here "good guess" is the solution 'x' vector obtained from an earlier solution. If we deal with images with similar patterns, then from the earlier solution database, we can make a good guess.
- **Plot showing convergence, comparison of convergence rate with expected rate**

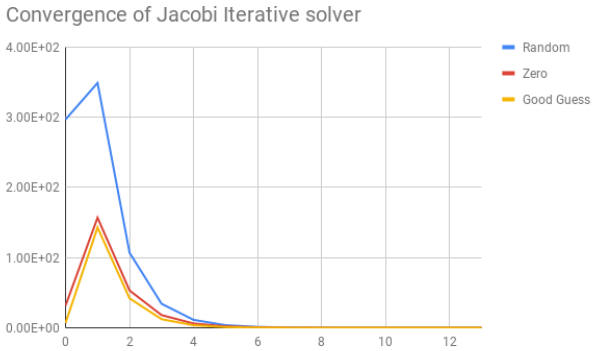


Figure 2: Convergence rate of Jacobi for Different Initialization Techniques

In the figure x-axis is number of iteration and y-axis is the maximum error in each iteration.

The expected rate is cos(pi/(n+1)) [4]. Compared to the expected rate, the convergence of jacobi iteration is shown as follows [Fig. 3].

- **Projection of time required for production problem:** Jacobi has running time $O(N^4)$. Therefore we would need 106.8890046 days if an image blending process of gigapixels of images is done in a

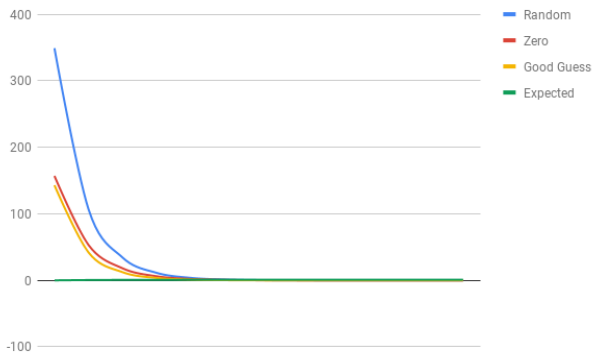


Figure 3: Convergence compared to the expected rate. In the figure x-axis is number of iteration and y-axis is the maximum error in each iteration.

sequential approach. We calculated this by scaling from the smallest runtime (which is 310 ms for N = 100) to larger runtimes.

- **Memory being allocated for test problem:**

  Maximum resident set size (kbytes): 48176

  Minor (reclaiming a frame) page faults: 12087

- **Comparison with memory being used by task:**

  For a problem size of N = 100, Jacobi method uses 48228 kbytes, whereas for initialization only we require 32492 kbytes. So in total we need 63896 kbytes of memory. So 75% of memory is contributed to Jacobi computation.

  Considering integer size to be 2 bytes and floating point size to be 8 bytes. For N = 1000, memory for A is 1000*1000*2 bytes, for B, x and intermediate matrix is 1000*8000*3 bytes. So that sums up to 22400 kbytes. The calculated estimated memory from integer and floating point size is 22400 kbytes, which is comparable to 48228 kbytes obtained using the command `/usr/bin/time -v ./main.out`.

- **Projection of memory required for production problem:** Considering integer size to be 2 bytes and floating point size to be 8 bytes. For N = 1000000, memory for A is 1000000*1000000*2 bytes, for B, x and intermediate matrix is 1000000*8000000*3 bytes. So that sums up to 22400 MBytes. We did not use the sparse representation in our implementation in progress report 1, but doing so saves 2/3rd of space contributed by A.

3 Solver Comparison

- **Which solver does better?** In my implementation Jacobi iterative solver does better both in compute time and memory.

- **Which solver will do better for a production scale problem?**

  Since our A matrix is sparse it is better to use Jacobi method. Sparse matrices do not generally have sparse LU decomposition, so it gets harder to fit the matrices in memory as the problem size goes larger. For Jacobi method, we can just store the A matrix in sparse representation.

- **Discuss how the production problem projections will be constrained and what will need to be taken into account for parallelization:**

  The Jacobi iterative solver can be implemented in parallel in the following way. Let p be the number of processors, f be the time for a floating point operation, α be the startup time, and β be the time

per word in a message. So the time required is going to be Time = number of steps * cost per step = $O(N) * ((N/p) * f + \alpha + (n/p) * \beta) = O(N^2/p) * f + O(N) * \alpha + O(N^{3/2}/p) * \beta$.
So, O(N/p) floating point operations are needed to do startup and O(n/p) boundary values are passed to other nodes [4].

4 Parallelization

4.1 Parallelization method being used

MPI is used as the parallelization framework. Here we do data decomposition, by distributing the A and B matrices across nodes. If N is the size of problem, p is the number of cores available, we stripe the problem evenly across p nodes by making the problem to a size of (N/p) subproblems.

4.2 Justification for method being used

- Standardization and Universality: MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries. MPI is an "industry standard" for writing message passing programs on HPC platforms. [5]
- Portability: There is little or no need to modify your source code when you port your application to a different platform that supports the MPI standard.
- MPI runs on either shared or distributed memory architectures. I used gather scatter. This bring more flexibility in coding.
- Flexibility: MPI offers more control on memory and data locality management,
- Although MPI puts a lot of responsibility on the programmer to manage memory and communicatiom, but since I am interested in learning the intrinsics of parallelization, and communication MPI is suitable for me both because of its ease of use and detailedness. [12]
- The other methods like Map-reduce, PGAS, OpenMP, Threads require less programming, but will be less efficient. OpenMP is not suitable for multicomputers. It depends on the scheduling of the OS whether different threads would be scheduled to different cores. Using OpenMP, we could only do loop parallelization which did not improve as much performance as MPI did.

4.3 Discussion of what in code is being parallelized

In code, the A matrix and B matrix is subdivided into p strips. Here p is a factor of number of processors N. When each iteration of jacobi is executed, the computation is performed in separate nodes.

Percentages parallelized (approx.):

After the serial code is profiled, we look at the portion taking a lot of time. We notice the structure in the matrix that can be parallelized. From Fig. 4, we see that the parallelizable part (jacobi main iteration) takes around $98 - 98.5\%$ of time.

Percentages not parallelized (approx.):

The serial portion is initialization of matrices, and adding up the sum from the cells of the matrix. This accounts for about 1.5% of the total time.

4.4 Discussion of any differences with setting up the A-matrix and b-vector and initial condition

A matrix is converted from a 2d array to a 1d array. This was done to make efficient and easier use of `MPI_Scatter` and `MPI_AllGather`. `MPI_Scatter` takes an array of elements and distributes the elements in the order of process rank. It is easier to decompose a linear array rather than a 2d array.

5 Profiling

For profiling we use both `MPI_Wtime` and `tau` to get the runtime.

5.1 Serial

5.1.1 Profile of serial code for sample problem size

If sample problem size is taken as 15*15=225,

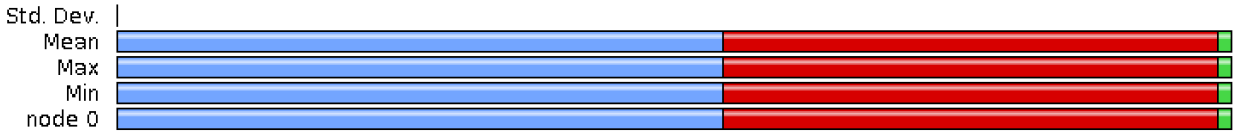


Figure 4: TAU Profile for Serial

The table corresponding to this profile is:

Table 3: Serial (Slow) Version for Problem Size 15*15=225

Percentage	Exclusive Time (msec)	Inclusive Time (msec)	Function Name
100	3	9,878	TAU
100	0.088	9,875	main
100	5,377	9,874	jacobiSolve
45.5	119	4,497	getError
44.3	4,377	4,377	matmul
0	0.378	0.378	init2d
0	0.111	0.111	fillA
0	0.015	0.015	init1d
0	0.011	0.011	getTime

The blue part corresponds to the proportion of exclusive runtime of `main()` function mentioned in Table 3. The red part corresponds to the proportion of exclusive runtime of `getError()` function mentioned in Table 3. The green part corresponds to the proportion of exclusive runtime of `init2d()` function mentioned in Table 3. It shows that the error calculation step in jacobi is taking a lot of time in this code. Initializations are very trivial compared to that.

What's taking a lot of time?

The first step of parallelizing an application is to find the most resource/time intensive part of the program. From Fig. 4 and Table 3, it is seen that we are being inefficient in calculating error in jacobi iteration.

Is this what you expect?

This is expected because matrix multiplication is a costly operation. It is done in quadratic time complexity.

5.1.2 Discuss potential steps to take to increase efficiency

In stead of using matrix multiplication to calculate error, it is possible to multiple inside the jacobi iteration main loop.

5.1.3 Modify your code with 1+ of your potential steps

The error is calculated in the main jacobi loop, and after that we get the following profile:

Table 4: Serial (Fast) Version for Problem Size 15*15=225

Percentage	Exclusive Time (msec)	Inclusive Time (msec)	Function Name
100	3	31	TAU
90	0.071	28	main
88.3	27	27	jacobiSolve
1.1	0.338	1	init2d
0.4	0.11	0.11	fillA
0.1	0.016	2	init1d

5.1.4 Show the updated profile and discuss the differences

In Table 4, we see a large improvement in the runtime. It is done in 31 msec now, whereas previously in Table 3 it was around 10 seconds. So we see a 320% speedup. This is expected, because we removed the costly looping matmult function, and locally calculated the error of jacobi iterations.

5.2 Parallel

5.2.1 Profile your parallel code, sample or production problem size

Sample problem size is 20*20=400. Production size could not be done due to limitations in ACI compute.

Table 5: Parallel (Initial) for Processor 10, Problem size 400

Percentage	Exclusive Time (msec)	Inclusive Time (msec)	Function Name
100	50	791	.TAU
93.6	9	741	main
74.5	590	590	MPI_Init()
15.9	125	125	MPI_Finalize()
1.7	13	13	MPI_Allgather()
0.1	0.863	0.863	MPI_Recv()
0.1	0.704	0.704	getError
0.1	0.645	0.645	MPI_Scatter()
0	0.0134	0.0134	MPI_Send()

What's taking a lot of time?

It is seen that establishing the MPI communication is dominating over other times.

Is that you expect?

This is expected, because this was done on 10 processors. For that, a lot of communication initiation happens.

Compare to serial

When this is compared to dimension 40*40=1600 with serial, it shows 10.6 sec as runtime. Whereas the parallel is 1.21 sec. So, parallel version is performing better than serial with 10 processor nodes. This is because the problem size is much smaller to make effective use of parallelism.

For dimension 20*20=400, non-MPI serial version actually does better. This leads us to another observation. For trivial problems, setting up the overhead of costly parallel communication is not helpful, rather it increases runtime.

Compare time/memory to estimate from PR#1

Time projection from PR1: The Jacobi iterative solver can be implemented in parallel in the following way. Let p be the number of processors, f be the time for a floating point operation, α be the startup time, and β be the time per word in a message. So the time required is going to be Time = number of steps * cost per step = $O(N) * ((N/p) * f + \alpha + (n/p) * \beta) = O(N^2/p) * f + O(N) * \alpha + O(N^{3/2}/p) * \beta$.

So, if we make an estimate using the vlaues, we get: 1600*(1600/10) + (1600/10)*2 = 256.2 second

Where the actual value is 351.307 second.

Memory projection from PR1: Considering integer size to be 4 bytes and floating point size to be 8 bytes. For N = 10000, memory for A is 10000*10000*4 bytes, for B, x and intermediate matrix is 10000*8*5 bytes. So that sums up to (10000*10000*4+5*10000*8) = 8GBytes of heap memory. By looking at the `tau` memory profile, by executing

```
tau_exec -memory ./jacobi_s 100
```

We find of heap memory allocated. For 40 cores, we get 8.19 GBytes memory allocated for 100 problem size.

5.2.2 Discuss potential steps to take to increase efficiency

The main bottleneck in parallel optimization is the MPI calls as seen in the figure described. The root process sends the data to everyone else while the others receive from the root process. If the MPI call from root is decreased, and the responsibility of passing message is handed over to one node to the other, then this can result in better load balancing.

5.2.3 Modify your code with 1+ of your potential steps

We use `MPI_BCast` to use broadcast collective call rather than single `MPI_Send` and `MPI_Recv`. If we use broadcast, process one will help process 0 pass data by forwarding it to process 2, and process 2 to the next one. process one is now helping out the root process by forwarding the data to process three. During the second stage, two network connections are being utilized at a time. The network utilization doubles at every subsequent stage of the tree communication until all processes have received the data.

5.2.4 Show the updated profile and discuss the differences

After modifying the code, to use `MPI_Bcast` instead of `MPI_Send` and `MPI_Recv`, we get the Table 6.

As you can see, there is no difference between the two implementations at two processors. This is because MPIBcasts tree implementation does not provide any additional network utilization when using two processors. However, the differences can clearly be observed when going up to even as little as 16 processors.

From the figure, it is clear that the nodes are well balanced. This was because instead of using `MPI_Send` and `MPI_Recv`. All the nodes have the same amount of overheads, and the same computation time.

But for a larger problem size,the modified version showed slight improvement (Fig. 5):

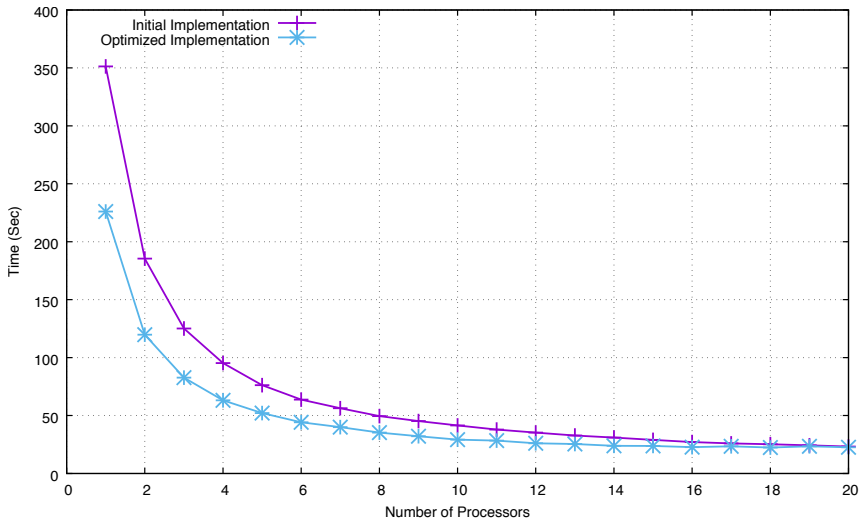


Figure 5: Comparsion of Optimized Parallel and Unoptimized Parallel Code

5.2.5 Show and describe the profile

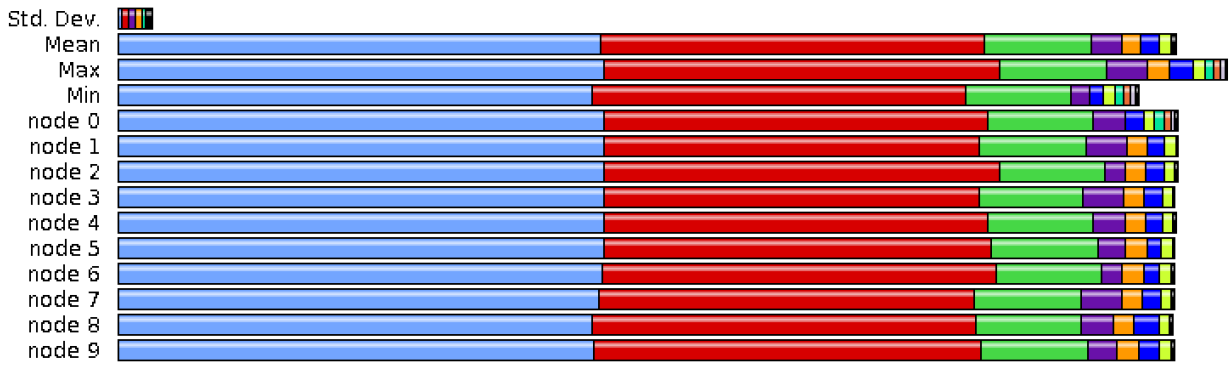


Figure 6: TAU Profile

Table 6: Modified Parallel Implementation

Percentage	Exclusive Time (msec)	Inclusive Time (msec)	Function Name
100	21	1,233	.TAU
98.3	447	1,211	main
45.7	563	563	MPI_Init()
10	123	123	MPI_Finalize()
3	37	37	MPI_Allgather()
1.8	21	21	MPI_Bcast()
1.1	13	13	MPI_Scatter()

The blue part corresponds to the proportion of exclusive runtime of `TAU` function mentioned in Table 6. The red part corresponds to the proportion of exclusive runtime of `main()` function mentioned in Table

6. The green part corresponds to the proportion of exclusive runtime of `MPI_Init()` function mentioned in Table 6. We see that profiling and parallel initialization is still causing the bottleneck. However, although we expected that adding MPI_Bcast will improve performance, it did not have much impact on overall runtime. This is because the amount of data to broadcast is very minimal in size, so it did not make much of an improvement. If the same technique was applied for larger communication, we might notice significant runtime improvement.

6 Strong Scaling

There are two basic ways to test the parallel performance: Strong Scaling and Weak Scaling.
In strong scaling case the problem size stays fixed but the number of processing elements are increased.

6.1 Strong scaling results for your code

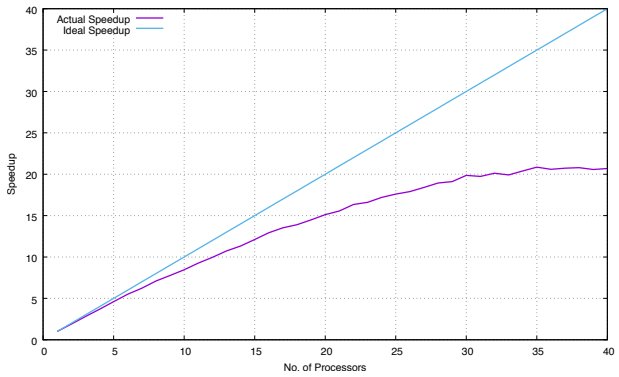


Figure 7: Strong Scaling: Speedup relative to 1 processor

6.2 Justification for starting point and size

Profiling with TAU shows that if we choose a problem size of 6400, 8 GB of memory is needed. Now for a larger production level problem where we have gigapixels of images, it is not possible to fit the computation into 1 node. For that, the 'data' which is the input A and B matrix need to be divided across nodes. It is possible to store the images in distributed systems, and stream portions of it to get around the problem of running out of memory.

Since we are able to run 6400 pixels using 8 GB memory, if we scale to to gigapixels, 8*1000 = 80000 GB memory is needed to store such a large scale problem. Per node we have 128 GB available. So we have to allocate total 80000/128 = 625 nodes.

When the problem dimension is 6400, it is not possible to complete the code in real time using 1 or 2 processor. We needed to allocate minimum 20 processor to make it run in reasonable time. A lot of image processing is needed as online systems. So it does not make sense to wait a long time before getting results. So for dimension 6400 we choose 25 processors to complete it in a reasonable time. We put more focus on time, rather than on efficiency.

For 6400 dimension, the total memory required is 8 GB. So, the need for more than one need is not because of runtime, but also because of memory.

To ensure that each node got a resonably sized sub-image to solve, the tests were limited to 20 nodes. Still the strong scaling plots show that its performance is not that good after 10 processors.

6.3 Overview of results

What size would you pick for a fast answer?

In Figure 8, we see that even if the processor number is increasing, it does not help at all with reducing runtime. The overhead of communication become more dominant. The problem size is so small here (N = 400, only 400 pixels), that adding communication overheads surpasses the benefit of paralelization.

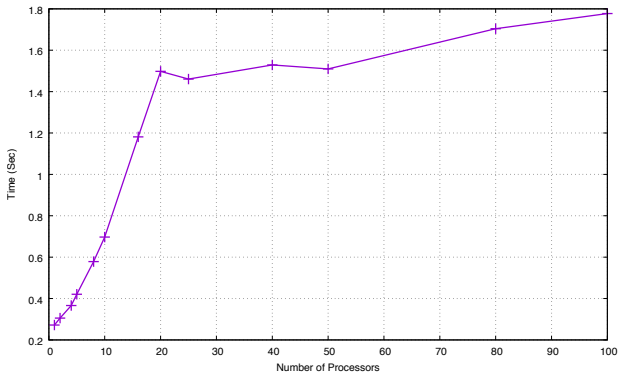


Figure 8: Runtime (seconds) of Parallel Implementation (N=400). Here parallelization does not help much in improving runtime.

So, if we apply parallelization technique, it should be on larger problem size. So, we increased problem size to sample problem size 80*80=6400

Also, from the timing plot at Fig. 10, it is clear that we should pick processor number around 25. Because it does the computation in under 20 seconds. This would be picked for faster solution.

What size would you pick for efficient resource use?

From Fig. 9, it is seen that efficiency drops as number of processors increase. So we are just bringing more resources, while a lot of those resources are just sitting idle and wasting their compute time. This results in loss of efficiency. We would want to be as efficient as possible, because with more processors come more cost. So around 70-80% efficiency is good enough. For that we pick the number of processing units from 15-20.

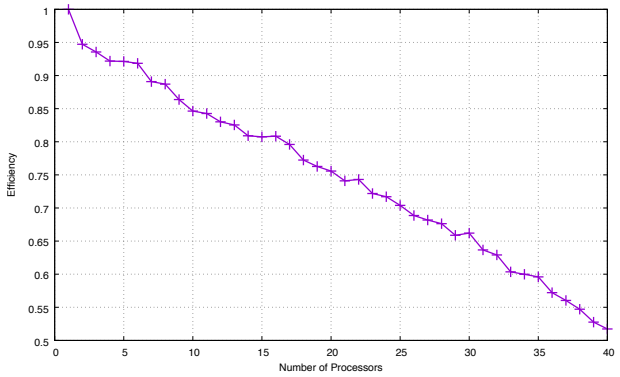


Figure 9: Efficiency of Parallel Implementation (N=6400)

What size would you use for research?

Digital images need 1 byte per pixel for simple storage (ranging from 0-255) in red, blue and green channel. This is approximately two orders of magnitude. But for high quality images, it is not enough. As

there are texture, lighting and gradient effects present in real pictures. The range can have about 108th order of magnitude, so it requires floats to be accurately represented. Since we will be dealing with gigapixels of images in production, for research an appropriate approximation can be in megapixels. This was chosen based on the availability of computing resources to us. In ACI, we can get upto 1024 GB of memory. So it is not possible to run production size images here.

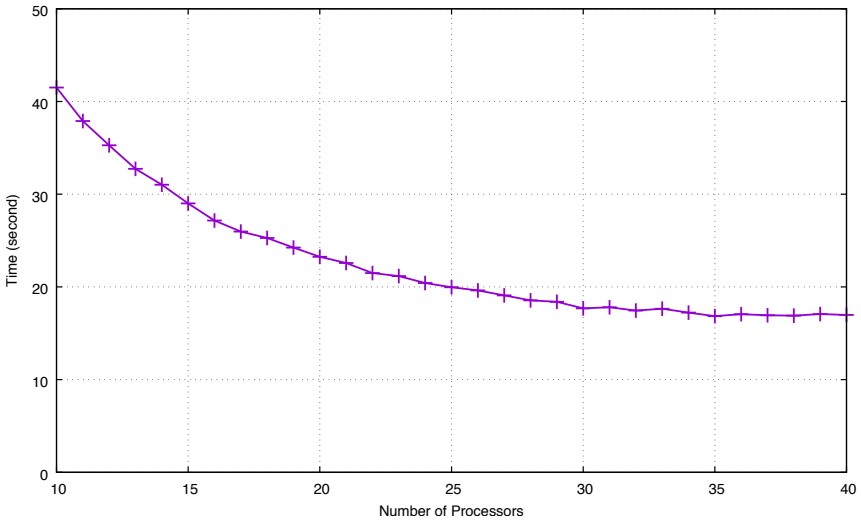


Figure 10: Time for N=6400, Optimized Parallel Implementation

At low values of N, parallel algorithm does not have a utility because of communication overheads Because of communication overhead, lower number of processors have decreasing speedup. But after some point (as p exceeds 20) it starts to improve.

So for research purpose in working with gigapixels, we can use more processors, while bringing down the time reasonably to do online image processing. In production system, such image processing algorithm are expected to give real-time results.

6.4 Plot scaling results next to Amdahl's law with your strong scaling

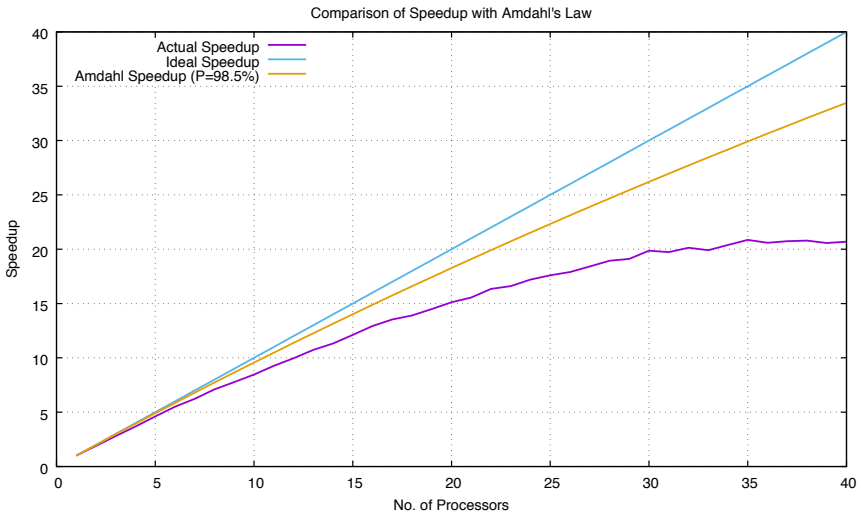


Figure 11: Comparsion with Amdahl's Law

Are you close to 'ideal'?

The results are quite close, but as the number of processors grow, they start to decrease. After 30 processors, it is almost flat. This implies that after a certain threshold, we won't see any significant performance improvement. Not close to ideal because performance of MPI code is limited by the communcation between the nodes.

As the parallel profile show that load balancing across nodes is pretty good, the standard deviations are low as well. This property made the code work well, and made the speedup curve comparable to Amdahl's law curve.

7 Weak Scaling

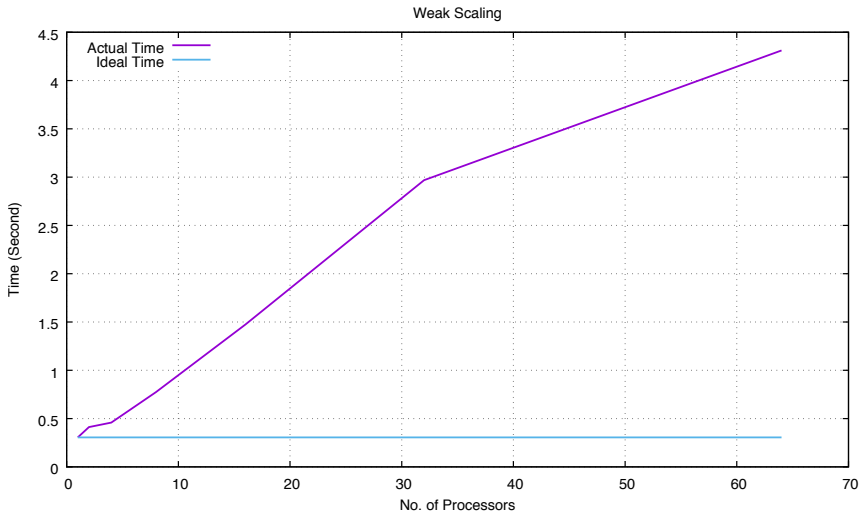


Figure 12: Weak Scaling

Strong scaling spread the same size problem across more nodes. Weak scaling increases the number of nodes, but keep the per node workload size constant, thus solving a larger problem size. So larger problem size will be solved by more processors, which is justification for programs take a lot of memory. This gives some justification to the extra CPU time the process is taking. In Fig. 12, running time of parallel jacobi is increasing almost linearly for the first 30 processors. But then the time So in 2 processors, 16 dimension matrix is given as input. For 4, it is 32 and so on. Ideally we would expect the time to same if we think linear speedup happens. But in real world, linear speedup is impossible. Because of the instrumentation code and MPI communication overheads, as number of processors increase, runtime increases as well.

8 Coupling to an External Library

8.1 Coupling

- **Justification for what you are replacing (why use a library for this section of code?)**

  I modified my MPI parallel code to run it in GPU. For that, I used CUDA. I chose CUDA as I wanted to add the power of GPU to my code, and make it faster. As for Jacobi methods, it is easier to parallel, as each iterations are relatively independent, so using GPU seemed promising to me.

  The main point of modification in the parallel code was the `getError()` function, which is called in each Jacobi iteration.

- **Justification for library being used (why this library and not another?)**

  Understanding the advantages of GPU in image processing, and observing a wide range of literature using GPU for this purpose, I decided to use GPU for my code. GPU has high bandwidth main memory. As we see from our experimental results, GPU offers massive parallesism, and thereby reduce latency significantly. While doing MPI code, we were limited to 60 nodes, however, with a single GPU, we can get the illusion of having thousands of separate compute resources.

  `CUDA Programming`: First of all, among all the methods of GPU programming covered in class, it is easier to improve performance by coding in CUDA. CUDA is comparatively low-level API for GPU programming that is exclusive to NVIDIA. Its learning curve is steeper, but since we are using pointers to pass memory back and forth, it was easier to keep track of which data went where. With OpenACC and OpenCL, the memory management seemed ambiguous. They also offered less flexibility in coding, as their motto was to make syntax simpler.

  `Thrust`: I decided to modify my parallel code using a library called "Thrust", mainly because it is released as part of the CUDA toolkit. So, it was easier to use the library without using any external linking. Since Thrust is an integral part of CUDA C++, I decided to use it. Also, it is easier to code in it, as memory management is cleaner. We do not have to keep doing memcpy back and forth explicitly. Thrust will significantly reduce the number of lines of code. So it is a nice tool to prototype CUDA code.

  `CUDA Kernels + CUBLAS`: Since CUDA kernels and cuBLAS offered more lower level support, I was comfortable optimizing them. cuBLAS has optimized implementations of common vector operations. That's why I decided to couple to this library as well.

- **Discussion of what you had to do to your code**

  Following the logic in MPI code done in Progress Report 2, I converted the code to CUDA code at first. As the profile obtained from Progress Report 2 showed that the error aggregation part of Jacobi was taking bulk of the time, I decided to modify the getError part of jacobi iteration.

  For this, first I converted this part using thrust. However, while library use simplifies programming, in this case, because of my inexperienced use, it degraded performance as seen in Fig. 13. It is seen that as problem size increases, the CPU time incurred for passing data back and forth to GPU device takes over. It pushes the runtime to higher up. For a mere 12x12 pixel size, it shoots up to as much as 35 seconds runtime. All of these had equal number of threads as problem size (so fully parallelized).

  The challenge in coding CUDA programming was realizing the optimum point of transferring data from memory to device. Performance changes drastically depending on this.

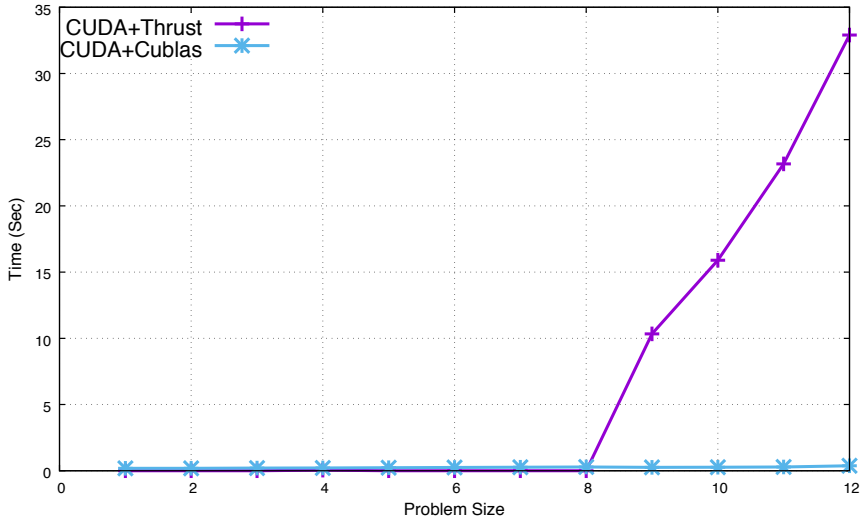


Figure 13: Comparison of CUDA+Thrust and CUDA Parallel Jacobi Code

After observing the poor performance of thrust, I decided to spend more time in optimizing this portion, and used cuBLAS. NVBLAS is a GPU-accelerated version of BLAS that further accelerates BLAS routines.

- **Updated profile or scaling with the library**

  After using cuBLAS with CUDA, we got the results in Fig. 14. It shows that even with 8 cores, the MPI code could not beat the performance of GPU code coupled with cuBLAS library. This was obtained for problem size 32x32. We see upto 5x speedup for 8 cores.

  Scaling Results of CUDA+CUBLAS:

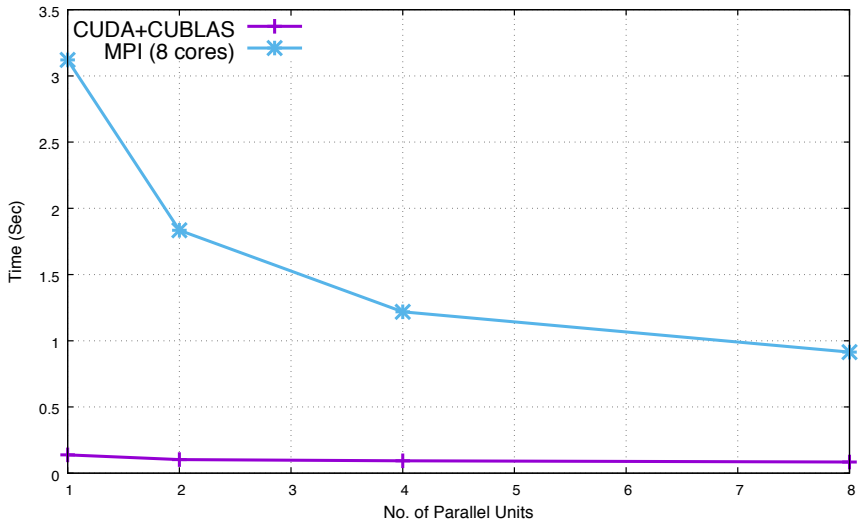


Figure 14: Comparison of Optimized CUDA+CUBLAS and MPI Parallel Jacobi Code

Memory Requirement:

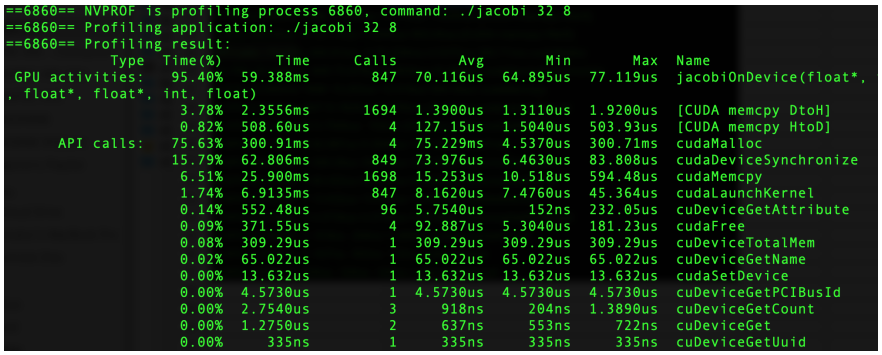
Using the command `nvidia-smi` For 10x10 pixels and 10 threads, The GPU memory usage for the thrust implementation is 289MB. Using CUBLAS: 329MB Jacobi: 443 MB

So in terms of memory management, using external library does better.

- **Discussion for how change can be seen in profile/scaling data**

  Scaling Results of CUDA+CUBLAS: In Fig. 14, we see that after coupling to external library and using GPU, the runtime has reduced. One thing to observe here is that MPI code is scaling better as the number of parallel processing unit is increasing. However in GPU, we are changing the degree of threading. By design, a well-written GPU code will strong scale to fill whatever GPU that is available there without any programmer or user intervention. So, the change in speedup or time improvement is less visible here.

  Updated Profile:



```
==6860== NVPROF is profiling process 6860, command: ./jacobi 32 8
==6860== Profiling application: ./jacobi 32 8
==6860== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   95.40%  59.388ms       847  70.116us  64.895us  77.119us  jacobiOnDevice(float*,
, float*, float*, int, float)
                   3.78%  2.3556ms      1694  1.3900us  1.3110us  1.9200us  [CUDA memcpy DtoH]
                   0.82%  508.60us         4  127.15us  1.5040us  503.93us  [CUDA memcpy HtoD]
      API calls:   75.63%  300.91ms         4  75.229ms  4.5370us  300.71ms  cudaMalloc
                   15.79%  62.806ms       849  73.976us  6.4630us  83.808us  cudaDeviceSynchronize
                    6.51%  25.900ms      1698  15.253us  10.518us  594.48us  cudaMemcpy
                    1.74%  6.9135ms       847  8.1620us  7.4760us  45.364us  cudaLaunchKernel
                    0.14%  552.48us        96  5.7540us     152ns  232.05us  cuDeviceGetAttribute
                    0.09%  371.55us         4  92.887us  5.3040us  181.23us  cudaFree
                    0.08%  309.29us         1  309.29us  309.29us  309.29us  cuDeviceTotalMem
                    0.02%  65.022us         1  65.022us  65.022us  65.022us  cuDeviceGetName
                    0.00%  13.632us         1  13.632us  13.632us  13.632us  cudaSetDevice
                    0.00%  4.5730us         1  4.5730us  4.5730us  4.5730us  cuDeviceGetPCIBusId
                    0.00%  2.7540us         3     918ns     204ns  1.3890us  cuDeviceGetCount
                    0.00%  1.2750us         2     637ns     553ns     722ns  cuDeviceGet
                    0.00%     335ns         1     335ns     335ns     335ns  cuDeviceGetUuid
```

Figure 15: Profiling with Optimized CUDA+CUBLAS Jacobi Code

The profile obtained using `nvprof` in Fig. 15 shows that, the GPU time is more utilized than CPU time. 95% of GPU time is actually spent in parallelizing code, and 5% is spent in overhead. On the other hand, if we look at the "API calls" section, we see that the overhead is higher. We also note that CPU time is taking longer (the highest portion taking 300.91ms) than GPU (61ms), so to utilize the full potential of the GPU, more computation needs to be done there.

Our implementation of CUDA code has a lot of room for improvements. We are using costly locking mechanism "cudaDeviceSynchronize" function, which adds 15.79% overhead. If we could design without the need for such synchronization, we could get better performance.

8.2 Potential Coupling

- **Discussion of what other parts of your code you could replace with other libraries**

  `cuRAND`:

  For testing, I used random numbers as matrix elements. Since my code had a random number generation for matrix in a part, I believe that optimizing it can be beneficial. As external library, I also attempted to use thrust and cuRAND with CUDA. Thrust is a framework for easier coding in CUDA. cuRAND is used to generate random numbers fast. However, due to time constraint, I could not optimize the code. I believe this is a scope for future work. Random number generation for a large matrix is a costly operation. That's why I think it will be worthwhile to use cuRAND library in generating random elements in the matrix.

- **Discuss the applicability of each topic below to your code**

  - **Advanced decomposition**

    For image processing, devising a partitioning scheme to convert image into tiles and apply an operation in a block of tiles is a common approach [10]. Depending on the relationship of image pixels (i.e. how close they are in pixel values) one or a group of image tiles are executed in a thread block in a GPU.

- **Solver libraries**

  I used thrust and cuBLAS. Some other numerical libraries are MKL (Math Kernel Library), PETSc, boost, etc. For the Jacobi code, in each iteration we need to calculate error. It involves repetitive numerical operation. The summing, squaring and taking square root of the error values to get global error are some common mathematical operations. So, instead of doing them manually, it is possible to use MKL to link this part. Boost can serves the same purpose. However, since I am using MPI and CUDA programming, PETSc can be a better choice. Because takes advantage of many other libraries, including BLAS and LAPACK. It also has support for MPI and MPI-GPU parallelism.

- **Parallel I/O libraries**

  One scope for improvement is using parallel I/O libraries like HDF5 and NetCDF to make the initial image loading time faster. This is an essential feature of the image blending problem we are trying to solve. If we do online image processing, where the input images change dynamically, we will be saving time by doing parallel I/O. Because no matter how fast the disks are becoming, they fall behind in speed by a great deal compared to processors. This gap creates bottleneck in time improvement. So, for our application in reading massive images from disk, using faster parallel I/O libraries will definitely be applicable.

- **Accelerators**

  One of the biggest group of manycore processors are GPU. GPU have been re-purposed from graphics-accelerated content, often found in video games. Prior to 2006, GPUs were difficult to program due to the difficult of mapping graphics functions to general calculations. They became more accessible in 2007 with the release of CUDA. I used accelerator in my code.

- **Discuss next steps for your code development**

  - **Which of the above do you think is worth your time and why?** If I had to choose one of the above four, I would choose "advanced decomposition". Using CUDA, if I can optimize my code using my own kernel, that can give me better performance. This is what I observed while adding libraries myself. The optimization that was lacking my coding was the decomposition part. I naively divided my matrix into stripes, which is the easiest decomposition method. I believe that it is possible to take advantage of the square grid structure of my matrix. Also, for 3-D image, there are several studies supporting decomposition.

    Also, I could not run my code for higher problem size using GPU. One possible reason is that some of the larger runs of my code was being pre-empted. Preemption writes the current memory to disk if the resource is required for another job. I did observe that some nodes ran faster than others. So, by being aware of the environment the program is running, we need to add code segments to make it fail-safe. This is another reason to use advanced libraries, they take care of this inherently. Using parallel I/O might solve this problem. In that, we could store the larger image pixels in multiple nodes, and perform parallel seek, thereby not overflowing the limited GPU memory.

9 Discussion and Conclusions

- **Basic overview of what you have done (include PR1, PR2 and final)**

  In the initial phase of this project, we implemented two solver for the Ax=B problem: Jacobi and LU Decomposition. For accurate solution, direct method is more appropriate. It does not need any estimation, and it is insensitive to all of the randomness from the model. Comparing LU decomposition with Jacobi iterative method we realized that LU decomposition is more resource hungry and takes more clock ticks than Jacobi iteration.

In the 2nd phase of the project, we describe a parallel iterative technique for parallel gradient domain processing of massive images. We implemented the code using MPI. As it is implemented in MPI, it is scalable in other High Performance Computing system as well. So the research done in this project, can be extended to different systems. We did both strong and weak scaling. It showed that although our method does not scale in an impressive manner, it improves some performance. The most notable one being using only 10 cores it improves runtime of jacobi from 1 second to 10 second for problem size 40.

Based on the insights from phase 1 and 2, it was realized that implementing the fast poisson image blending application in real world has major resource constraints, the crucial one being the constraints are the size of node available in real world. Another one of the constraints is storing a large image file in a single node.

In the final phase of the project, we coupled an external library to our code to exploit the potential of GPU programming. We wanted to utilize the highly parallel structure of Jacobi iterative method to the fullest. We observe that although library use makes coding simpler, using it inefficiently especially for parallel programming can decrease performance because of the overheads related in moving data in and out form GPU. So, we changed the code to use less movement of data, and updated it using cuBLAS library. It shows 5x speedup over a parallel MPI implementation with 8 processors. In both GPU and MPI programming, we find the use of library make code and memory management cleaner.

- **Discussion of any constraints and future work**
  - **Future work:**

    Because of the limited resources, we could test our code upto 60 nodes only. It is possible to further see the effect of runtime improvement using higher number of nodes.

    As iterative method, I used Jacobi, which is the simplest and easier to implement in parallel. However, for better performance, another iterative method called SOR can be used.

    We provided a basic framework and guideline for image blending application. For now, our code will be applicable for image pixels given in greyscale value. It is possible to support color images by extending this code, and instead of keeping one B and X vector, 3 B and X vectors can be added for three color channels.

    There might be scope for better optimization based on the input image. For example, if we deal with a perfect square number matrix dimension, the decomposition of the matrices could be done more efficiently by distributing $\sqrt{N}$ pixels in p processors. In future, we will do better decomposition to make the efficiency better.
  - **How might your work be modified with thoughts to advancing technology, i.e. Quantum Computing, clusters on a chip, exascale computing, etc.**

    As exascale computing approaches, the research towards fast parallel I/O is accelerated. Now is the era of high performing computing. A powerful computing resource such as this can create realistic images and models from natural scenario. Both in virtual reality and in high definition animations, our proposal of fast image blending proves to be useful. So when we will have this much resource available to us, it makes sense to faciliate study to make full use of them. By doing this project, I realized that the more such high clusters are emerging by the innovation of hardware scientists, the more the responsibility of the programmaers increase - to come up with ways to efficiently use them. Without using parallelization techniques, we will be only having faster hardwares, but not receive any true benefit at all. Another area worth explorint is quantum computing. According to the principles of quantum physics, the computing power of a quantum machine is monumental if we just compare it with our high performing production machines. So currently, important applications of quantum computation to computer science have been developed (e.g. faster algorithms, secure transmissions). In the study [9], some of the image manipulation techniques such as histogram computation and histogram equalization and

show are expressed using the quantum formalism. Inspired by their results, I think as a future work it will be interesting to exploit special properties of quantum computation to achieve better performance.

Appendices

A Acknowledgements

I took help from the template code provided by Professor Lavely and Professor Blanton for Jacobi iterative method implementation, and the script to cycle through optimization flags. The resources provided by them for our course were of immense help. All of the computations are performed on my ACI account, which sponsored by PSU CSE department. I would also like to thank my classmates Sahithi and Chirag for helping me with ACI commands.

B Code

B.1 Final Report Code

- **Where can you find the code**

  `https://github.com/amatur/cuda597`

- **File names and descriptions**

  - `jacobi.cu`: The parallel (optimized) CUDA implementation of the iterative solver.
  - `jacobi_cublas.cu`: CUDA implementation of the iterative solver Jacobi with coupling to cuBLAS library.
  - `jacobi_rand.cu`: The parallel (unoptimized) implementation of the iterative solver with coupling to cuRAND library.
  - `jacobi_thrust.cu`: The unoptimized implementation of the iterative solver Jacobi with coupling to thrust framework.
  - `Makefile`: Make file to compile CUDA code using nvcc.
  - `sc.sh`: Shell script that was used to generate strong scaling data.
  - `wc.sh`: Shell script that was used to generate weak scaling data.
  - `poster.pdf`: Poster about this project.
  - `poster.txt`: Text for accessibility for poster about this project.
  - `gpl.txt, license.txt`: Licensing information.

- Instructions for compiling and reproducing your results

  First, login to xsede, and get connected to bridges.

  ```
  ssh -l USERID login.xsede.org
  gsissh bridges
  ```

  After logging into bridges, you need to request GPU resources using the command:

  ```
  interact -gpu -t 02:00:00
  ```

  Now, after you are in the node which has a name like `USERID@gpu045`, load the following modules by typing:

  ```
  module load cuda/10.0
  ```

Set the environment variables for both CUDA path and library path:

```
export PATH=/usr/local/cuda-10.0/bin\${PATH:+:\${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64\${LD_LIBRARY_PATH:+:\${LD_LIBRARY_PATH}}
```

Download the code:

```
git clone git@github.com:amatur/cuda597.git
```

Change directory into `cuda597` and "Make" the code after cleaning up:

```
make clean
make
```

Run the optimized CUDA code, the last line of the output is runtime in seconds:

```
make runcu
```

Run the unoptimized thrust code, the last line of the output is runtime in seconds:

```
make runth
```

Run the code coupled with cuBLAS:

```
make runcublas
```

After compiling the code of your choice, run `nvprof ./jacobi 16 2` to get the profile of the code as presented in the report.

The first argument after `./jacobi` is the problem size, N. The second argument is the block size for parallelization.

- Listing of which nodes types you ran on
  - `br018`
  - `br008`
  - `gpu045, gpu046` (Results were generated with Nvidia Tesla 28C GPU).

B.2 Project Report 2 Code

Where can you find the code

The code is publicly available at `https://github.com/amatur/cse597_parallel_solver`.

File names and descriptions

- `jacobi.cpp`: The parallel (optimized) implementation of the iterative solver.
- `jacobi_s_opt.cpp`: The serial (optimized) implementation of the iterative solver.
- `jacobi_p_unopt.cpp`: The parallel (unoptimized) implementation of the iterative solver.
- `jacobi_s.cpp`: The serial (unoptimized) implementation of the iterative solver.
- `test.cpp`: Test code to check the correctness of jacobi implementation.
- `Makefile`: Make file to compile using tau.
- `sc.sh`: Shell script that was used to generate strong scaling data.

- `wc.sh`: Shell script that was used to generate weak scaling data.
- `serial.sh`: Shell script that was used to compare serial data.
- `poster.pdf`: Poster about this project.

Instructions for compiling and reproducing your results

After logging into ACI, you need to request a compute node with desired number of processors and nodes. For example, to run the code on maximum 20 nodes:

```
qsub -I -A open -l walltime=2:00:00 -l nodes=1:ppn=20:scivybridge
```

Now, after you are in the compute node which has a name like `comp-sc-0161`, load the following modules by typing:

```
module load gcc/5.3.1
module load openmpi/1.10.1
module use /storage/work/a/awl5173/toShare/tauPdt/tau
module load adamsTau_2.27
```

Download the code:

```
git clone git@github.com:amatur/cse597_parallel_solver.git
```

Change directory into `cse597_parallel_solver` and Make the code after cleaning up:

```
make clean
make
```

Run the optimized serial code:

```
make runs
```

Run the unoptimized serial code:

```
make runsu
```

Run the optimized parallel code, the last line of the output is runtime in seconds:

```
make runp
```

Run the unoptimized parallel code, the last line of the output is runtime in seconds:

```
make runpu
```

After running the code of your desired choice, run `pprof` to get the profile of the code as presented in the report.

List of node types we ran on

- Standard compute (Ivy Bridge): comp-sc-0106
- High memory: comp-hc-0015

B.3 Project Report 1 Code

- **Where to find the code:**
  `https://github.com/amatur/cse597_solvers`
- **File names and descriptions:**
  - `cycleThrough.sh`: The shell script to iterate through all of the optimization flags to find out the best one.
  - `gpl.txt, license.txt`: Licensing information
  - `main.cpp`: source file containing Jacobi and LU decomposition solver
  - `Makefile`: makefile for generating executable main.out
- **Instructions for compiling and reproducing your results:**
  Instructions can also be found at `https://github.com/amatur/cse597_solvers`

  Log in to ACI, using ssh: `ssh USERID@aci-b.aci.ics.psu.edu`

  After logging into ACI, you need to request a compute node with desired number of processors and nodes. For example, to run the code on maximum 1 node:

  ```
  qsub -I -A open -l walltime=2:00:00 -l nodes=1:ppn=1
  ```

  Now, after you are in the compute node which has a name like `comp-sc-0161`, load the following modules by typing:

  ```
  module load gcc/7.3.1
  ```

  Download the code:

  ```
  git clone git@github.com:amatur/cse597_solvers.git
  ```

  Change directory into `cse597_solvers` and Make the code after cleaning up:

  ```
  make clean
  make
  ```

  Run the code:

  ```
  ./main.out 4000
  ```

  You should see output with runtime for both direct and iterative solvers.

  ```
  Total time(seconds) taken by CPU for LU Decomposition Only: 18.7073
  Total time(seconds) taken by CPU for Forward Substitution: 0.01055
  Total time(seconds) taken by CPU for Backward Substitution: 0.0106
  Total time(seconds) taken by CPU for LU Decomposition: 18.7366
  Total time(seconds) taken by CPU for Jacobi: 0.365309
  ```
- **Listing of which nodes types we ran on:**
  - Standard compute (Ivy Bridge): comp-sc-0106
  - High memory: comp-hc-0015
  - Basic node: comp-bc-0352

C Poster - separate document

Available in: `https://github.com/amatur/cuda597/blob/master/poster.pdf`
Poster text for accessibility: `https://github.com/amatur/cuda597/blob/master/poster.txt`

References

[1] Lu decomposition takes more computational time than gaussian elimination! Retrieved from `https://autarkaw.org/2008/06/04/lu-decomposition-takes-more-computational-time-than-gaussian-elimination-what-gives/`, accessed 2018-09-23.

[2] Poisson blending. Retrieved from `http://eric-yuan.me/poisson-blending/`, accessed 2018-09-23.

[3] Poisson image editing. Retrieved from `http://www.ctralie.com/Teaching/PoissonImageEditing/`, accessed 2018-09-23.

[4] Solving the discrete poisson equation using jacobi, sor, conjugate gradients, and the fft. Retrieved from `https://people.eecs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html`, accessed 2018-09-23.

[5] Barker, B. Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede* (2015), vol. 262.

[6] Berisha, S., and Nagy, J. G. Iterative methods for image restoration. In *Academic Press Library in Signal Processing*, vol. 4. Elsevier, 2014, pp. 193–247.

[7] Bhat, P., Curless, B., Cohen, M., and Zitnick, C. L. Fourier analysis of the 2d screened poisson equation for gradient domain problems. In *European Conference on Computer Vision* (2008), Springer, pp. 114–128.

[8] Bradski, G., and Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[9] Caraiman, S., and Manta, V. Image processing using quantum computing. In *2012 16th International Conference on System Theory, Control and Computing (ICSTCC)* (Oct 2012), pp. 1–6.

[10] Chapman, W., Ranka, S., Sahni, S., Schmalz, M., Majumder, U., Moore, L., and Elton, B. Parallel processing techniques for the processing of synthetic aperture radar data on gpus.

[11] Daks, A. Object detection data augmentation via poisson blending. Retrieved from `http://alondaks.com/2017/12/01/object-detection-data-augmentation-via-poisson-blending/`, accessed 2018-09-23.

[12] Gropp, W. D., Gropp, W., Lusk, E., Skjellum, A., and Lusk, A. D. F. E. E. *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[13] Hockney, R. W. A fast direct solution of poisson's equation using fourier analysis. Tech. rep., STANFORD UNIV CA STANFORD ELECTRONICS LABS, 1964.

[14] Paudel, K. Stitching of x-ray images, 2012.

[15] Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. *ACM Transactions on graphics (TOG) 22*, 3 (2003), 313–318.

[16] Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 313–318.

[17] Philip, S., Summa, B., Bremer, P.-T., and Pascucci, V. Parallel gradient domain processing of massive images. In *EGPGV* (2011), pp. 11–19.

[18] Saad, Y. *Iterative methods for sparse linear systems*, vol. 82. siam, 2003.