Parallel Computing for the Optimization of Mesh Consolidation

Jacob Pessin
pessij@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

Alexandra Vest
vesta@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

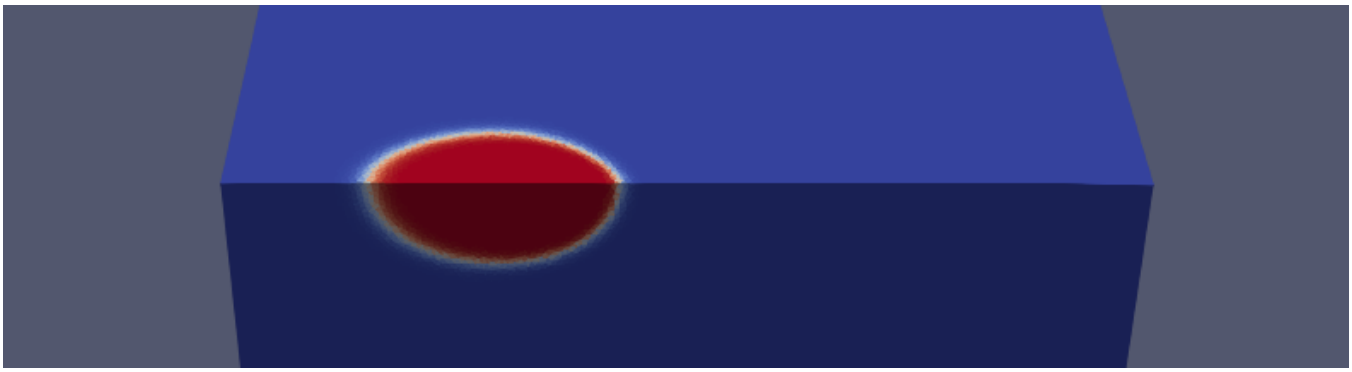

Figure 1: Melt pool caused by laser beam incident on the edge of the block.

ABSTRACT

Additive manufacturing simulations are becoming increasingly prevalent in research due to the need for more accurate predictors of mechanical properties of the ever-growing quantity of parts produced. Simulations that model the manufacturing of each layer are used to help predict these mechanical properties. An important step that is the focus of this paper is the consolidation of the porous powder layer as it is melted by the laser. The re-consolidation of the mesh following melting can be tracked through a state variable which measures whether or not the powder has been melted. This variable must then be accurately tracked and related throughout the simulation, and applied to the next layer's mesh. A version of code which completes this has previously been written in MATLAB, and a prototype of this code has been supplied for this project to be written in C and used on the AiMOS supercomputer at RPI. This will allow for better compatibility between the simulation software and the consolidation code. This prototype code was then put under extensive testing to identify the performances of GPUs, MPI, and parallel I/O under varying input parameters. The parameter which showed the most critical influence on performance times was threads assigned to a block for the GPU computations. By increasing the threads per block alongside problem size, a substantial decrease in run times can be achieved. Additionally, by attempting to limit the ranks used in MPI/parallel I/O application, the communication overhead time can be greatly reduced. While the code presented in this paper is the groundwork for complete conversion from MATLAB to C language, work must continue on the code to convert all functions in the original MATLAB code to allow for the accurate representation of the consolidated mesh.

ACM Reference Format:
Jacob Pessin and Alexandra Vest. 2020. Parallel Computing for the Optimization of Mesh Consolidation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Additive manufacturing has become much more prevalent in the manufacturing world due to its rapid production capability, ease of prototyping, and the creation of intricate designs and otherwise impossible to manufacture geometries. One of the major setbacks that many companies and organizations have in more widely adopting additive manufacturing technologies is the uncertainty in the strength of its created parts. To help mitigate this issue and increase its use, simulations are implemented to predict the mechanical properties of additive manufactured parts and components.

These simulations return properties of the printed material such as temperature, laser source, and state variables. A pertinent result from the Albany code simulations evaluated in this paper is the phase variable. This variable stores the degree to which the powder region has been melted for the current laser pass. For the code in question for this paper, the variable is referred to as *psi*1. This variable ranges from 0, never melted, to 1, completely melted (Roy et al., 2017). As a history variable if a region has ever been melted it will be stored in this variable.

Throughout the simulation this variable changes as the laser passes over the mesh elements. This variable is important to be

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

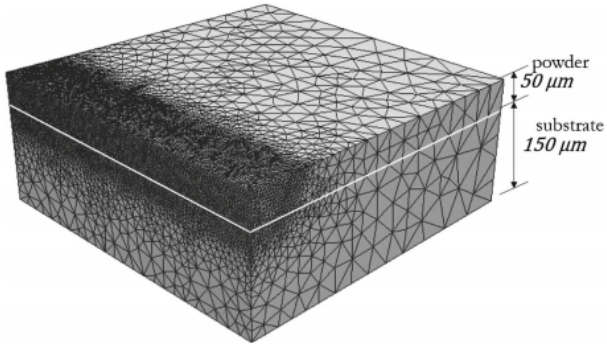

Figure 2: Undeformed mesh with a single layer of powder on top. The bottom block is the build plate, while the top layer is the powder of which the additive manufactured part will be made (Roy et al., 2017).

able to predict the final porosity thereby affecting the processing of the next layer. Therefore, this variable must be considered when creating the next geometry to build a powder layer upon (Roy et al., 2017). The challenge presented from this variable is creating unity between neighboring elements so that a contiguous geometry is created for the next layer's mesh.

A way this is often managed is by consolidating the mesh following each individual layer of powder. This accounts for the melting and re-solidification for the next interval. One such way to implement this step is through parallel computing. of each mesh region as well as with the individual mesh elements.

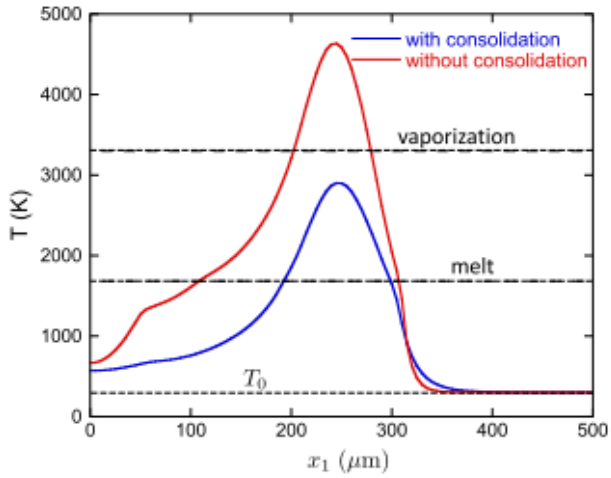

Figure 3: Temperature graph on top surface of powder region with and without consolidation implemented (Roy et al., 2017).

Previous researchers have integrated parallel computation in various ways into their finite element simulation processes. Parallelization of large-scale fluid mechanics problems has become arguably necessary due to their size and complexity (Tezduyar Sameh, 2005). Johnson and Tezduyar implemented parallel computation in their fluid dynamics simulations to decrease computation time. In their simulations, the mesh required updates that incorporated the moving domain and its respective boundaries and interfaces. In order to accomplish this for their 3D simulations, they utilized separated the domain into bounded regions that could compute mesh updates in parallel (Johnson Tezduyar, 1994).

Implementing parallel methods can produce shorter computation times, however, they must be efficiently incorporated into the existing methodology. In order to increase the parallel efficiency seen in re-meshing for fluid mechanics problems, the time interval for the re-meshing must be optimized between the time it takes to re-mesh, and the time it takes for the fluid mechanics solver to complete its task. If too long of a period is taken between re-meshing, enhanced discretization is produced causing increased computation for the solver. Conversely, if too short of a period has passed, the re-meshing causes time delays in the acquisition of the solution (Tezduyar Sameh, 2005).

There have also been changes outside of the fluid dynamics field of finite element simulations. Parallelization has also been used to completely rewrite finite element simulation code for a more efficient solution process. Law developed a method which "computes directly the element distortions, as opposed to solving a system of global equations" (1986, p. 1). This method allowed for each element to be mapped onto a processor which then handled the calculations and information regarding that element. This allowed for the elements to all be computed in parallel, therefore increasing computation times. Another improvement was more time was spent on the calculations than on the interprocesser communications (Law, 1986).

Lastly, parallel methods have been utilized for mesh refinement and then redistribution, similar to methods proposed in this paper. Özturan and peers used the "mesh topological hierarchy" as the inherent data structure to formulate the information needed for adjacent mesh elements (1993, pg. 1). This relationship was then used to connect processors to share information to create the mesh for the next time step. This method provided good convergence results when tested (Özturan et al., 1993).

Lian and partners also devised a parallel adaptive mesh refinement scheme for their unstructured tetrahedral mesh. Their methodology focused on refining the initial elements before they were manipulated into smaller pairs. They aimed to stray from the edge based pairing of neighboring elements, due to the high memory allocation demands of this approach. Instead, an algorithm was implemented which utilized the relation between neighboring elements to further adjoin them to other elements (Lian et al., 2006).

MPI is used to to bring in the data from the previous grid mesh, and the refinement method desired. A host processor holds the the connected data between the mesh sub-groups required for the refinement, since neighboring data is needed to define the next mesh. Sub-domains are then allocated to individual processors for refinement in parallel until a satisfactory limit has been reached. The host then receives the information from the individual processors, and arranges the sub-groups in the necessary order for the mesh domain (Lian et al., 2006).

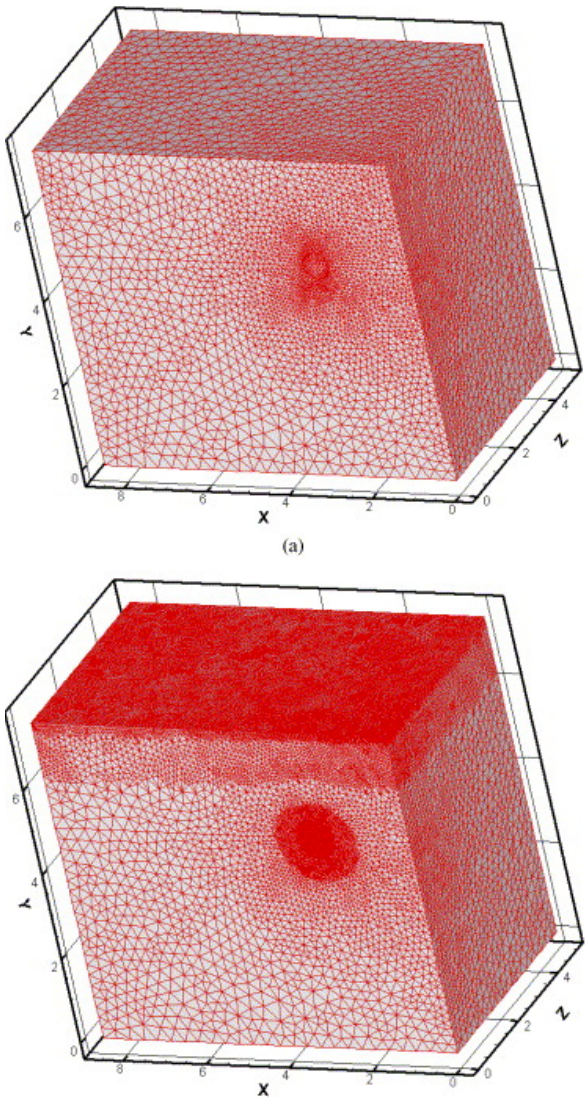

Figure 4: Top figure is not refined, bottom figure is refined using the Level 6 parallel adaptive mesh refinement (Lian et al., 2006).

This model was tested for a varying number of processors, from 2 to 64, increasing in increments of 2 to an exponential. For only 2 processors, the mesh refinement time for the total time was 1103 seconds. When 32 processors were used, the time was decreased to 37.1 seconds, only 3.4 % of the time taken with 2 processors (Lian et al., 2006).

As can be seen through these previous implementations, the inclusion of parallel computing within finite element simulations can provide benefits for decreased computation time and an increase in efficiency for the overall experiment. These benefits include decreased run time and increased accuracy. For these reasons, a prototype of code for the Albany simulations has been created which consolidates the mesh from the previous step of the simulation. This code has previously been formulated in MATLAB, and was not compatible with the simulation software used by the Maniatty group for simulations and used for parallel programming on AiMOS.

The motivation for this project is to reduce the overall time spent running a multiple layer simulation through the use of multiple processors. Additionally, by creating code which can run in parallel there will no longer be a need to switch software interfaces to consolidate the mesh in MATLAB, which will also a decrease in the overall simulation time.

2 BACKGROUND

Previously, a serial version of the code was created by James Dolan to be run in MATLAB for the simulation being considered in this paper. This code reads in each part of the simulation file as a single file then determines the displacement of the finite element mesh nodes for the powder layer.

However, this code takes several minutes to run, causing a delay in the overall simulation experiment. Additionally, it requires the user to change software to utilize MATLAB to complete the mesh consolidation further increasing the overall run time of the simulation. This code prototype has been adapted to create a basis for future parallel implementation of the code. The final code will avoid these delays and allow for the accurate and timely consolidation of the mesh between powder layers.

2.1 Operating System

The work done for this project was completed on AiMOS (Artificial Intelligence multiprocessing Optimized System). This supercomputer is currently the most powerful in New York state with 8-petaflop IBM POWER9 CPUs. It also makes use of NVIDIA GPU (SCER Staff, 2019).

2.2 Original Mesh Consolidation

The mesh consolidation works through the incorporation of finite element methods. Finite element simulations are created from deconstructing a model into multiple pieces which are referred to as elements. These elements are then assigned nodes at their corners. A common type of element is the tetrahedral element, which has 4 nodes. These nodes are described in both local and global coordinates (Kattan, 2008). Local coordinates are with respect to the element, and global coordinates are with respect to the entire system.

The equations which govern these elements, and therefore simulations, are often repeated boundary value problems. These equations are solved through the use of the variational, or weak, form of the boundary value problem which can be solved on a local (elemental) level. The boundary value problem uses finite element functions in order to solve the variational. This step is repeated for all elements within a mesh, and then repeated continuously for the amount of time steps assigned (Hughes, 1987).

The form of the equations used for analysis in this code belong to the Galerkin method of finite element methods. The Galerkin method uses multiple linear equations to solve the boundary value problems. The creation of these linear equations occurs through

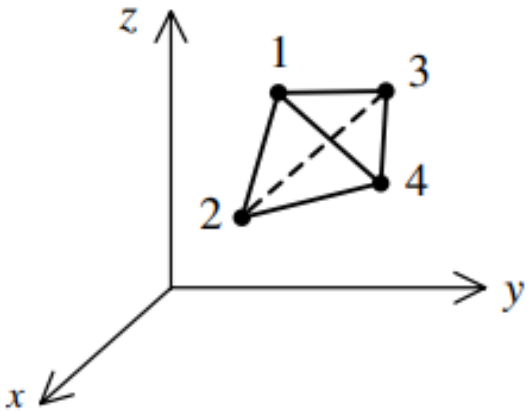

Figure 5: Example of a tetrahedral element in 3D space (Kattan, 2008).

the stiffness matrix, **K**. In order to construct this stiffness matrix to solve the given problem, shape functions are used which cause the value of the linear system to be 1 at a given node (Hughes, 1987). For tetrahedral elements, the shape functions are linear in nature (Kattan, 2008). The following is an example of the 1-D Galerkin equation (Hughes, 1987).

$$a(N_a, N_b)d_b = (N_a, f) + N_a(0)h - a(N_a, N_{n+1})g \tag{1}$$

In this equation, the values of the stiffness matrix are described by the weak form solution of the shape functions. The weak formulation is denoted by $a()$, and the shape functions are denoted by N_a, N_b.

$$K_{ab} = a(N_a, N_b) \tag{2}$$

The right hand side of the Galerkin equation can be further described as the force vector, **F**. g are the boundary conditions on the displacement equations which describe the motion of the elements. Furthermore, h are the boundary conditions on the first derivative of the displacement equations (Hughes, 1987).

$$F_a = (N_a, f) + N_a(0)h - a(N_a, N_{n+1})g \tag{3}$$

Finally, that leaves one unknown in the Galerkin equations, **d**. This is figured to as the displacement vector, and is solved for using the stiffness matrix and the force vector.

$$K_{ab}d_b = F_a \tag{4}$$

For the mesh consolidation code created in this paper, the above equations are altered to be with respect to the boundary conditions being prescribed from `psi1`, and for a 3D space.

The number of nodes is read from the output file provided from the simulation with the deformed mesh before consolidation. The code is read through in ranks, whose multitude is determined by the number of output files from the simulation. This requires the the offset to be later calculated for the element locations in order to offset them by the appropriate amount ensuring elements indices are associated with the correct element and are in the correct location.

The nodes, with respect to their elements, is then further defined into their local coordinates taken again from the output file. The nodes are then organized to the coordinates of their respective element. The state variable, `psi1`, which is used to solve the Galerkin equation for displacement of the nodes, must also be read in from the output file.

Four linear shape functions are then created which each have their respective node they highlight. The shape functions are then multiplied with the local coordinates of a singular element, creating the Jacobian matrix for that element. The inverse of the Jacobian is then used to find the distortion of the elements within that time step.

For the element, the experimental porosity of the powder layer following the laser pass must be checked. If this value returns as less than 0, it must corrected to be 0. This is because it represents the porosity which cannot physically be less than 0.

Then, the element stiffness matrix, element force vector, and the stress strain displacement matrix, **B**, must be initialized. The stress strain displacement matrix is an addition to the 1D version of the Galerkin equations, and is used to solve for the stiffness matrix further along in the calculations. The stress strain displacement matrix is assigned the values of distortion of the local z coordinates. The element stiffness matrix is then updated.

$$\mathbf{k}_e = \mathbf{k}_e + \mathbf{N}^T * \mathbf{B} * w * J \tag{5}$$

For the equation above, w is the weighting factor set to be 1, and J is the determinant of the Jacobian matrix. Then the force vector for the element can be updated using the updated element stiffness matrix.

$$\mathbf{f}_e = \mathbf{f}_e - \mathbf{N}^T * \frac{por * psie * w * J}{1 - psie} \tag{6}$$

psie is the *psi*1 variable recorded for the current element, and *por* is the term used to govern whether or not there is an effect from the porosity. It was previously found to be 0 if the change in the local z coordinates was greater than the powder thickness. If it is 0, there is no effect to the force vector from the term since there is no porosity from which to bring change. If not, it allowed the equation to function as if it was not there.

The ID array is then aligned, The ID array contains "relationship between global node numbers and global equation numbers as well as nodal boundary condition information" (Hughes, 1987, pg. 71). This will be used to map the local coordinates and *psi*1 variable to the entire system.

Additionally, the LM array must be found following the ID array. This takes the global assignments in the ID array, and connects them to the local coordinates so that they have the correct boundary conditions and element number (Hughes, 1987). This will store the results, and later be used to map the results to the global ID array.

Then a linear solver is used to find the solution to the Galerkin equations generated through Equation 4. This step is the final solution to the problem which will allow for generation of the final mesh. Once this is completed, the new consolidated mesh can be outputted to a file to be read back into the simulation.

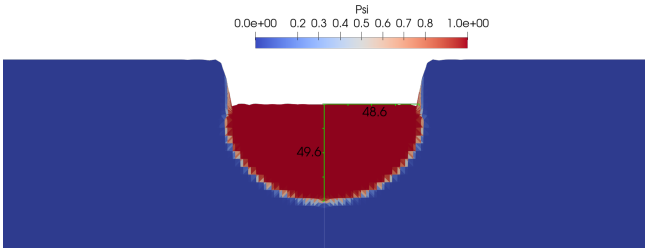

Figure 6: An example of the updated mesh following consolidation of the powder layer that was melted from the previous research of Jacob Pessin.

2.3 Current Mesh Consolidation Prototype

For the current prototype of the code evaluated, the number of nodes per rank is prescribed by the user. From this value, the number of elements is found in the function `num_ElementsNodes`, and later arranged into an array in `read_elements`. The nodes, with respect to their elements, are then further defined into their local coordinates in `read_coordinates`.

The state variable, `psi1`, which is used to solve the Galerkin equation for displacement of the nodes is currently assigned in function `read_psi`. The rest of the functions are handled within the CUDA function `gol_runKernel`.

Within the kernel, the ID matrix is arranged with respect to the number of elements within a rank. For any value of of a distorted z coordinate which returns a value less than 0, the value is set at 0 for the powder thickness. If not, the value is set at 1.

The array `a_bar`, meant to represent the solution **d** in Equation 4 prior to mapping, is set to be the values of the element force vector. This is then used to run the CUDA function `gen_corrector`. `gen_corrector` handles the mapping of the value assigned to `a_bar` to **d**. The last function of the code is then used to free the arrays used in CUDA computations.

2.4 CUDA Devices

GPUs are often used to enhance the execution of code compared to the traditional CPU. The speedups seen through the use of these CPU devices can be anywhere from 10 to 1000 times that of a multi-core CPU (Lee et al., 2010). To incorporate the use of GPUs in parallelism, the Nvidia CUDA platform can be used.

CUDA uses threads which encompass a thread block. These blocks are chunks of data, or arrays, that are run on different GPUs. These threads are synchronized in order at the launch of each kernel, and following with a synchronize command (Harris).

In this prototype, there is one function which was run on GPUs. The function is the `gen_corrector` function which corrects the numerical solver array to make sure it is within bounds and returns the updated matrix to be outputted as the displacement of the mesh in the powder layer region.

2.5 MPI

MPI, or message passing interface, is a program designed for the communication between tasks or processes. This allows for memory to be shared between processors through the send and receive operations. It increases the portability of the data analyzed (Barker, 2015).

It offers several advantages in parallelization. It is compatible with various hardware systems, allowing for nearly universal implementation. MPI helps control the speeds between processes, not allowing one process to become backed up or waiting for data. Lastly, it improves performance by allowing the user to allocate specific data to a process (Gropp, Lusk, Skjellum, 2014).

MPI is used in conjunction with CUDA in this program. It allows message passing between the clusters to obtain the neighboring data. Specifically, MPI is used to send and receive the different mesh parts for consolidation. The input mesh parts are then used within CUDA program to generate the new mesh parts, which are outputted by MPI processes.

2.6 Parallel I/O

Parallel I/O is "multiple processes of a parallel program accessing data ... from a common file" (Carothers, 5). While the use of parallel I/O is not as simple as sequential I/O, it offers high performance unseen by sequential I/O. Additionally, it can be utilized to produce a single output file from the data (Carothers).

In the early 1990's, researchers were looking to develop a basis for parallel I/O that allowed for the adequate transfer of files. Eventually, it was proposed that parallel I/O be based off of MPI. However, instead of sending and receiving messages, parallel I/O would read files and write to files (Corbett et al., 1995).

MPI libraries were updated to be a basis for I/O shortly thereafter. Since then, it has been optimized to allow for fast, unique uses. One such use is collective I/O, which allows for the combination of small requests for file data to be combined into large, conjoined access to the data to decrease the amount of total reads necessary. In collective I/O, each process receives only the data it requested (Carothers).

In this code, parallel I/O is used to read and write to the mesh part files. It creates the output file which harnesses the data for the new mesh created after consolidation of the powder.

3 PERFORMANCE EVALUATION

The performance of the code was determined through multiple test cases which varied the problem size,GPUs/compute nodes count, and threads per block count for each of the cases. In order to encompass the scope of parallel performance capabilities, CUDA, MPI, and parallel I/O portions were tested. This includes weak and strong scaling studies, as well as a comparison between sequential and parallel performance. Overall comparison was completed through examining the communication overhead time seen for each test case.

3.1 Experimental Setup

In order to complete testing, several meshes configurations were used from a previous single layer of powder experiment. In the slurmSpectrum file, the number of threads per block and the number of nodes per rank were specified, respectively. In the `sbatch` command the number of GPUs and compute node counts were altered. For all test cases, program overhead time was also recorded and compared to find the optimal system.

3.2 CUDA Analysis

The CUDA analysis was completed through a weak-scaling and strong-scaling study of the CUDA run times. The weak scaling study was done by increasing the number of GPUs and compute nodes while keeping the number of nodes per rank constant and threads per block constant. Since the node count input is for each rank, this causes the problem size to increase while the GPU and compute node count remained the same.

The strong scaling study was completed by the alteration of node count per rank along with the GPU/compute node count. Test trials were completed so that the total nodes for the entire problem were the same.

Additional tests were also run to test the CUDA performance alone. A set of tests were completed which altered the thread counts per block, while holding all variables steady for weak scaling analysis of the thread count. An additional set was completed which also altered the world size with thread count for strong scaling analysis purposes.

3.3 MPI Analysis and Parallel I/O Analysis

The MPI and parallel I/O analysis was completed through the use of test trials from the strong scaling CUDA tests. Since this was the scaling for the MPI ranks as well, the MPI read and write times were considered for these tests to make complete comparisons.

3.4 Overhead Analysis

For all the test cases completed, communication overhead times were also found using the `getticks(void)` function. This function utilizes the cycle counter of the IBM POWER9 CPU to measure the amount of time spent completing communication overhead.

This allows for introspection into the amount of time actually spent doing computational work (the amount of run time left after subtracting out communication overhead time). By analyzing the runs with the lowest communication overhead, it can determined which tests were the most efficient.

4 RESULTS

The results for the individual components are explained in the following subsections. Four examinations were completed for the CUDA portion of the code. One of those tests also holds applicable data to the MPI tests. The final section contains data on the overhead communication times for all tests ran.

4.1 CUDA Results

For the weak scale study, the results are summarized in Table 1. The results showed an optimal performance rate at 2 GPUS/compute nodes for a total world size of 65,536 nodes. Following this result, there was a decrease in performance with increasing GPUs/compute nodes count.

The one GPU/compute node performance is poor, which can be attributed to an influx of data on one compute node and one GPU, therefore causing slowdowns. For the 4 and 6 GPUs/compute nodes tests, the program overhead should be considered. For 2 GPUs/compute nodes, the average overhead communication time was approximately 0.55 seconds. However, for the 4 GPUs/compute nodes test it was approximately 0.86 seconds, and for 6 GPUs/compute nodes it was as high as 1.7 seconds. This shows that there was an increase in communication time for the latter two tests, causing an overall increase in the time it took to complete the calculations within the kernel.

Table 1: CUDA Weak Scaling Tests

Ranks	Nodes/Rank	Nodes	Update/Node(s^{-1})
1	32,768	32,768	178,924
2	32,768	65,536	254,504
4	32,768	131,072	209,665
6	32,768	196,608	142,107

Table 2: CUDA Strong Scaling Tests

Ranks	Nodes/Rank	Nodes	Update/Node(s^{-1})
1	32,768	32,768	61,439
2	16,384	32,768	51,804
4	8,192	32,768	32,269
8	4,096	32,768	27,979

For the strong scaling study, the test case with 1 GPU/compute node had the best performance rate. This is due to the relatively small size for a simple parallel computation. Since the calculations are not as complex as they will be in the final version, the kernel can be completed fairly quickly. Therefore, for small world sizes such as 32,768, one compute node is enough for optimal time.

However, as shown in the weak scaling study, when the calculations become more demanding, either with size or complexity, it will not always be the case that the one GPU/compute node configuration will be optimal.

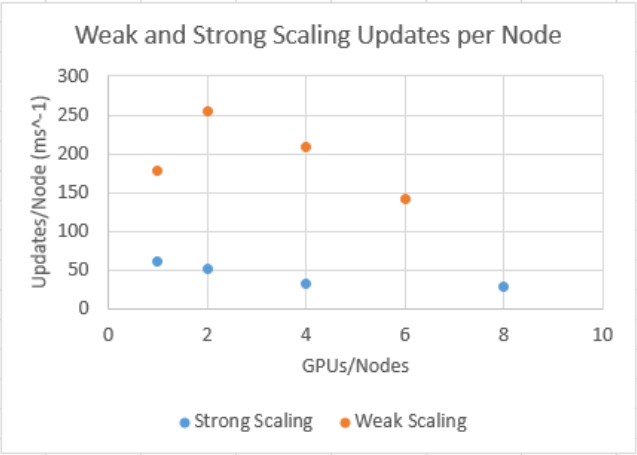

Figure 7: Updates per node for both weak and strong scaling studies.

As seen in Figure 7, the trend for the weak scaling updates per node appears to be linear following the initial surge in updates at 2 GPUs/2 compute nodes. For the strong scaling tests, the decrease in updates per node are linear from the beginning up until 4 GPUs/4

Table 3: Threads/Block Strong Scaling Tests

Problem Size	Threads/Block	Update/Node($s^{(}-1$))
32,768	128	199,084
32,768	256	216,125
32,768	512	192,740
32,768	1,024	209,665

compute nodes where it seems to reach a plateau continued by 8 GPUs/compute nodes.

In addition, a serial performance test case was run with 1 GPU/compute node and 1 thread per block for a total node size of 32,768. This had an update rate per node of 70,424 updates/node. This was higher than any of the strong scaling results, but not of the same magnitude as the weak scaling results. This further ascertains that for the larger problem sizes, the GPU/compute node count optimization is imperative, but for less significant problems it may not cause much of an influence.

An additional test was run to study the influence of threads per block on the performance of the code. Both strong and weak scaling testing was completed. For the strong scaling tests, ranks, nodes/rank, and total problem size were held constant. The thread size per block was then altered from 128 to 1024.

The most efficient thread/block size was 256 threads/block. Following this, there was a significant decline in the cell updates per second. Using 512 threads/block gave the lowest performance rate overall.

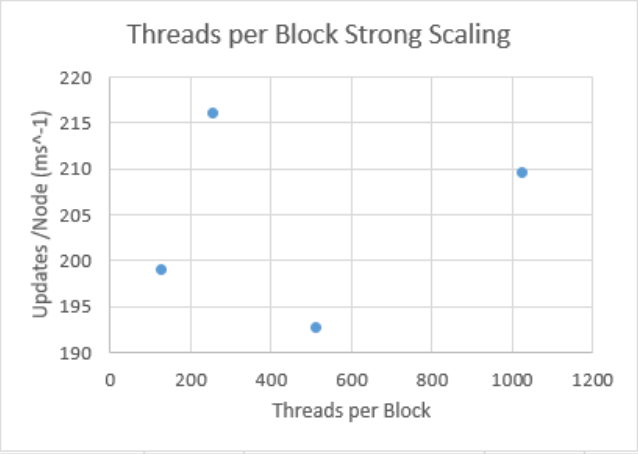

Figure 8: Strong scaling results for threads per block testing.

The weak scaling tests increased the nodes per rank while holding the rank number at 4 therefore increasing the problem size. The performance rate the for the weak scaling tests showed continuous increase as the threads per block and problem size increased.

Unlike the strong scaling results, there appears to be a correlation within the weak scaling results. The increase appears to be almost linear in nature. The updates of the largest scaling with 1,024 threads per block and a problem size of 65,536 is almost double that of the 512 threads per block and world size 32,768 problem size.

Table 4: Threads/Block Weak Scaling Tests

Problem Size	Threads/Block	Update/Node(s^{-1})
4,096	128	6,617
8,192	256	12,396
32,768	512	53,033
65,536	1,024	103,340

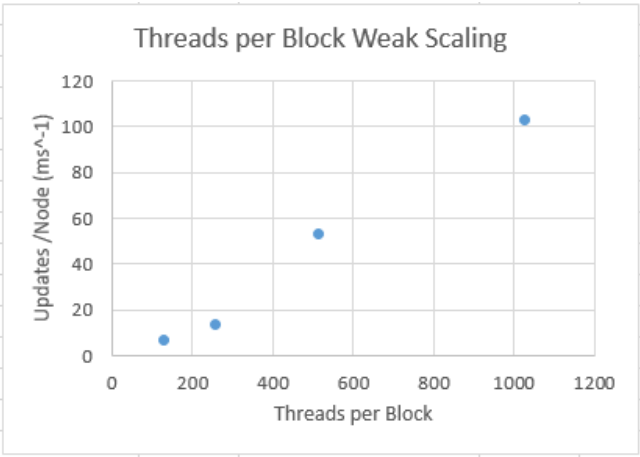

Figure 9: Weak scaling results for threads per block testing.

Table 5: MPI/Parallel I/O Tests

Ranks	Nodes/Rank	MPI Write Time (s)	MPI Read Time (s)
1	32,768	0.0344	0.00121
2	16,384	0.0478	0.00185
4	8,192	0.0466	0.00259
8	4,096	0.0966	0.01113

4.2 MPI and Parallel I/O Results

Universally, all tests had a longer time to write than time to read. All cases, with the exception of the 8 ranks test case, had a write time an order of a magnitude larger than its read time.

The MPI read time continued to increase with rank number. Small increases in time to complete were seen for 2 and 4 ranks, however, the 8 ranks test had an increase much larger than the previous ranks.

The MPI write times had a similar behavior. There was again a small rise in time for the 2 and 4 rank cases. However, for the MPI write portion of testing, the test case with 4 ranks had a time approximately 1 *ms* faster than that of the test case with 2 ranks. The test case with 8 ranks once again had a much more significant increase, uncharacteristic of the previous test cases.

For the overall run time for MPI operations, the single rank test case had the lowest completion time. The 2 ranks and 4 ranks test cases had nearly the same overall run time, with the 4 rank case only being 0.4 ms faster. The total run time for the 8 rank case was

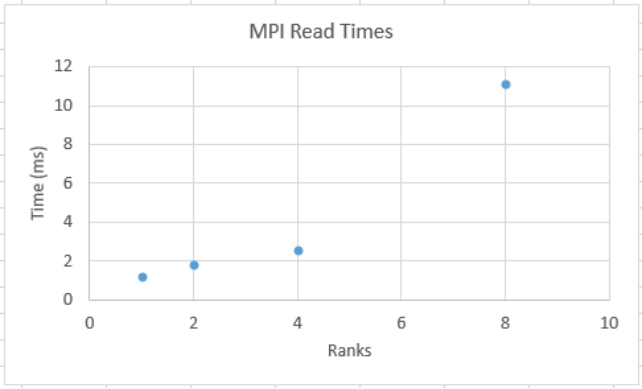

Figure 10: Time taken to complete MPI read operations for varying rank number.

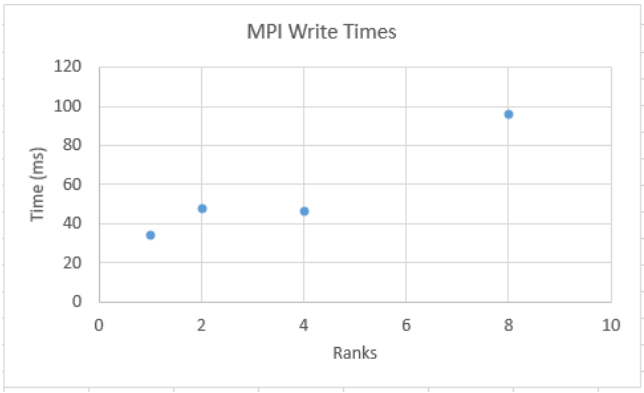

Figure 11: Time taken to complete MPI write operations for varying rank number.

219 % that of the 2 ranks and 4 ranks test cases, and 302 % that of the single rank test case.

From this data it can be concluded that the single rank case is the fastest to complete since there is only one file to consider for the operations. The effects of adding on a limited amount of ranks causes an increase which seems to be steady for some time. However, with larger amounts of ranks, it appears that the effects of reading and writing to various files is fully felt, causing significant time increases.

4.3 Program Communication Overhead

As aforementioned, for all tests communication overhead was also found. This value was returned for each rank run for a test case. Since there is little change between the times received for each rank, they have been averaged for analysis purposes. This averaged time was then divided by updates to normalize all tests for comparison. As a control, the communication overhead time per node for the serial run of the code was 20.91 us.

For these values, the most desirable is the lowest value. The lowest value indicates that for the relative amount of nodes in

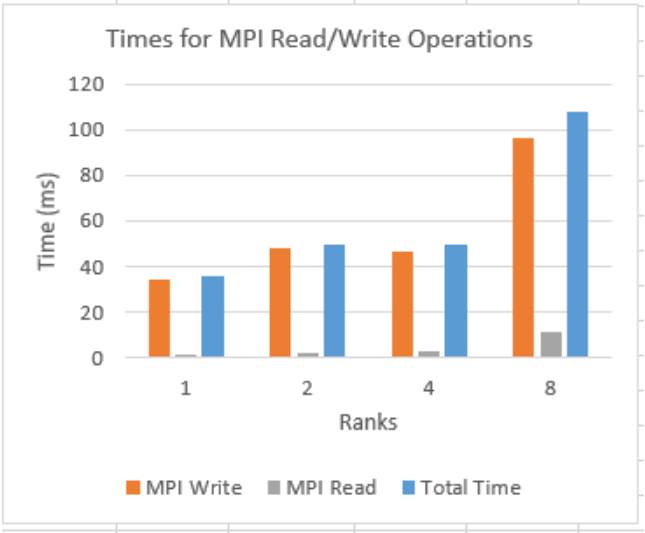

Figure 12: MPI write, MPI read, and overall times for each of the test cases.

the test case, that amount of time was allocated to each node for overhead communication purposes.

The first four columns in the left column are the table are the strong scaling tests for the GPUs/compute nodes tests (also the MPI/Parallel I/O tests). It can be seen that communication overhead time per node increases as the number of ranks increases. This is logical as the problem size is remaining the same and more ranks are being added causing more communication that must occur for the reads and writes.

The last four cases in the left column are the strong scaling tests for the threads per block analysis. These tests showed a low amount of communication overhead, with no clear correlation between the threads per block and time per update.

The top four of the right hand column are the weak scaling results for the GPUs/compute nodes tests. The first value for 1 rank was so small, it was rounded down to 0. It is believed that this was the result of an issue with the run since it is an apparent outlier. For the remaining three test cases, they are all similar. There is a maximum difference of 2.2 us between the cases. This leads to the conclusion that when the rank size and problem size increases, the communication time remains fairly constant for its assigned nodes per rank value.

The last four in the right hand column are the weak scaling test cases for the threads per block case. These cases showed a dramatic decrease in time per update as the threads per block and problem size increased. The initial value for 128 threads per block and 1,024 nodes per rank is 211.8 us, however that drops to 13.24 us by the final test of 1,024 threads per block and 16,384 nodes per rank. This can be attributed to more calculations being completed within the code following communication. Communication overhead will not increase since there are only 4 ranks being communicated for test case. To conclude, as the calculation amount increases the communication necessary remains constant causing an overall low

Table 6: Program Overhead Times for All Tests

Ranks	Threads	Nodes/Rank	Time per Node (us)	Ranks	Threads	Nodes/Rank	Time per Node(us)
1	1,024	32,768	22.25	1	1,024	32,768	0
2	1,024	16,384	26.34	2	1,024	32,768	8.497
4	1,024	8,192	39.92	4	1,024	32,768	6.596
8	1,024	4,096	47.69	6	1,024	32,768	8.726
4	128	32,768	7.440	4	128	1,024	211.8
4	256	32,768	6.414	4	256	2,048	104.6
4	512	32,768	8.190	4	512	8,192	26.35
4	1,024	32,768	6.596	4	1,024	16,384	13.24

time per update for large thread per block and nodes per rank counts.

5 CONCLUSIONS AND FUTURE WORK

Throughout this study, many observations have been made regarding the influence of several parameters on the overall function of the mesh consolidation code prototype that has been created. When considering both the weak and strong scaling CUDA tests that were completed, the optimal performance range was seen for the use of 2 GPUs/compute nodes. Increasing the GPUs/compute nodes input past this initial range resulted in slow downs for both of the scaling studies. Utilizing 1 GPU/compute node gave results secondary to that of the 2 GPUs/compute nodes overall performance, however, this configuration should be used with caution when dramatically increasing problem size as it can easily become overwhelmed with increased complexity.

Throughout the threads per block analysis of the CUDA portion, a conclusion can be made from the weak scaling tests. As the problem size and the threads per block increased, the updates per node also increased. A fairly significant increase was seen in the updates per node. Therefore, for greater problem sizes the threads per block should be increased accordingly to allow for maximum computation rate. No significant conclusions can be assumed from the strong scaling thread per block tests as no direct correlation was observed.

The MPI/Parallel I/O tests displayed much more substantial write times compared to read times. It follows that it takes the program longer to write the same amount of data than to read the data. Optimization should be focused on this area. There was a general increase in MPI/parallel I/O time with rank increase. A single rank provided the least amount of time for overall completion. The 2 rank test case and 4 rank test case were nearly steady with one another, and the 8 rank test case had a much higher overall run time. From this is can be deduced that the minimal amount of ranks should be used to limit time spent on reading and writing to files.

The final conclusion which can be drawn from testing is regarding the scaling of threads per block to problem size. From the study on communication overhead time, it was seen that a direct correlation was seen between these two variables and overhead communication time per node. Since the decrease in time per node was substantial for increasing the two variables simultaneously, the scaling of these two variables together is encouraged to increase performance. This further strengthens the previous recommendation from the CUDA weak scaling test results for the threads per block.

The prototype code for mesh consolidation in parallel discussed in this paper proved the feasibility of the transfer of the code from MATLAB to the C language compatible with AiMOS. As the project continues to move towards the super-computing stage, it is imperative to ensure that all aspects of the simulation are designed to work on the same interface for simplicity and reduction of error. Through the combination of GPUs, MPI, and Parallel I/O the original mesh consolidation code can become an integrated part of the super-computer computations. While this prototype provided a basis for our simulations in parallel, there is work which must be completed to have the code functioning at maximum accuracy.

One aspect which must be included in future versions of the C language code is the reading in of nodes and elements from the finished previous time step file. Currently, these values are assigned. In order to allow for the true consideration of these variables, they must be brought in from previous data. The number of ranks will also be need to be read in from the number of part files the mesh is separated into.

Additionally, the full finite element approach must be included. Equations 5 and 6 must be incorporated into the code which use the read in variables to update the Galerkin equations for each local element. Then these local values must be assigned to the global ID array to assemble the mesh.

Finally, within the original MATLAB version of the consolidation code, there is a numerical solver implemented. This solver evaluates the system of linear equations which describe displacement. For this code, displacement was not correctly solved for, but assumed from the coordinates. A future direction for the C language version of this code would be to integrate the linear equation solver to allow for the most accurate new mesh.

ACKNOWLEDGMENTS

We would like to acknowledge James Dolan, who wrote the original code in MATLAB for the consolidation.

Also, we would like to acknowledge Prof. Maniatty for her assistance and encouragement for this project.

REFERENCES

Barker, B. (2015, January). Message Passing Interface. *High Performance Computing on Stampede.*

Carothers, Chris. *MPI File I/O*.

Corbett, P., Feitelson, D., Hsu, Y., Prost, J.-P., Snir, M., Fineberg, S., … Wong, P. (1995). MPI-IO: A Parallel File I/O Interface for MPI Version 0.3. *NAS Technical Report*.

Gropp, W., Lusk, E., Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-passing-Interface* (Vol. 1). Cambridge, MA: The MIT Press.

Harris, Mark. *Optimizing Parallel Reduction in CUDA*.

Hughes, T. J. R. (1987). *The Finite Element Method Linear Static and Dynamic Finite Element Analysis*. Englewood Cliffs, NJ: Prentice-Hall.

Johnson, A. A., Tezduyar, T. E. (1994). Mesh update strategies in parallel finite element computations of flow problems with moving boundaries and interfaces. *Computer Methods in Applied Mechanics and Engineering, 119*(1-2), 73–94. doi: 10.1016/0045-7825(94)00077-8

Kattan, P. I. (2008). The Linear Tetrahedral (Solid) Element. *MATLAB Guide to Finite Elements*, 337–365. doi: 10.1007/978-3-540-70698-4_15

Law, K. H. (1986). A parallel finite element solution method. *Computers Structures, 23*(6), 845–858. doi: 10.1016/0045-7949(86)90254-3

Lee, V. W., Hammarlund, P., Singhal, R., Dubey, P., Kim, C., Chhugani, J., … Chennupaty, S. (2010). Debunking the 100X GPU vs. CPU myth. *Proceedings of the 37th Annual International Symposium on Computer Architecture - ISCA 10*. doi: 10.1145/1815961.1816021

Lian, Y. Y., Hsu, K. H., Shao, Y. L., Lee, Y. M., Jeng, Y. W., Wu, J. S. (2006). Parallel adaptive mesh-refining scheme on a three-dimensional unstructured tetrahedral mesh and its applications. *Computer Physics Communications, 175* (11-12), 721–737. doi: 10.1016/j.cpc.2006.05.010

Özturan, C., Decougny, H. L., Shephard, M. S., Flaherty, J. E. (1993). Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Computers. doi: 10.21236/ada290451

Roy, S., Juha, M., Shephard, M. S., Maniatty, A. M. (2017). Heat transfer model and finite element formulation for simulation of selective laser melting. Computational Mechanics, 62(3), 273–284. doi: 10.1007/s00466-017-1496-y

SCER Staff. (2019, December 5). "AiMOS, Most Powerful Supercomputer at a Private University, To Focus on AI Research". Retrieved from https://news.rpi.edu/content/2019/12/05/aimos-most-powerful-supercomputer-private-university-focus-ai-research

Tezduyar, T. E., Sameh, A. (2006). Parallel finite element computations in fluid mechanics. *Computer Methods in Applied Mechanics and Engineering, 195*(13-16), 1872–1884. doi: 10.1016/j.cma.2005.05.038

A RESEARCH METHODS

A.1 Individual Contributions

The following are the contributions for individual members of this group.

Jacob Pessin converted the MATLAB code into C language, worked on debugging, ran some tests, reviewed the paper, and provided moral support to Alex.

Alexandra Vest worked on debugging, created the test cases and ran the simulations, ran some test cases, worked on the paper, and provided moral support to Jacob.