Development of Fast Reconstruction Techniques for Prompt Gamma Imaging during Proton Radiotherapy

REU Site: Interdisciplinary Program in High Performance Computing

Johnlemuel Casilag[1], James Della-Giustina[2], Elizabeth Gregorio[3], Aniebiet Jacob[1],
Graduate assistant: Carlos Barajas[4], Faculty mentor: Matthias K. Gobbert[4],
Clients: Dennis S. Mackin[5], and Jerimy Polf[6]

[1]Department of Computer Science and Electrical Engineering, UMBC,
[2]School of Information Technology & Computer Science,
Community College of Baltimore County,
[3]Department of Physics, Hamline University,
[4]Department of Mathematics and Statistics, UMBC,
[5]Department of Radiation Physics, The University of Texas MD Anderson Cancer Center,
[6]Department of Radiation Oncology, University of Maryland — School of Medicine

Technical Report HPCF–2017–16, hpcf.umbc.edu > Publications

Abstract

Proton beam radiation treatment was first proposed by Robert Wilson in 1946. The advantage of proton beam radiation is that the lethal dose of radiation is delivered by a sharp increase toward the end of the beam range. This sharp increase, known as the Bragg peak, allows for the possibility of reducing the exposure of healthy tissue to radiation when comparing to x-ray radiation treatment. As the proton beam interacts with the molecules in the body, gamma rays are emitted. The origin of the gamma rays gives the location of the proton beam in the body, therefore, gamma ray imaging allows physicians to better take advantage of the benefits of proton beam radiation. These gamma rays are detected using a Compton Camera (CC) while the SOE algorithm is used to reconstruct images of these gamma rays as they are emitted from the patient. This imaging occurs while the radiation dose is delivered, which would allow the physician to make adjustments in real time in the treatment room, provided the image reconstruction is computed fast enough. This project focuses on speeding up the image reconstruction software with the use of of parallel computing techniques involving MPI. Additionally, we demonstrate the use of the VTune performance analyzer to identify bottlenecks in a parallel code.

Key words. Proton beam therapy, Image reconstruction, SOE algorithm, Parallel computing, High performance computing.

AMS subject classifications (2010). 90C15, 97M60, 97R60.

1 Introduction

In order for physicians to ensure accurate treatment, it is essential for them to have images of the patients anatomy taken throughout the administration of radiation therapy. This is

necessary because, as a patient undergoes treatment, their anatomy changes as the tumor shrinks and surrounding tissue swells. This means that each day during treatment the target for the radiation may be slightly different. Therefore, if a physician were to have the ability to see where inside of the body the proton beam is delivering its dose while in the treatment room they would be able to more accurately treat patients. It is possible for physicians to attain this information through prompt gamma imaging of the proton beam and image reconstruction.

Prompt gamma imaging works by capturing the scattered gamma rays released when a proton beam interacts patients cells and applying the Stochastic Origin Ensemble (SOE) algorithm. These gamma rays are released while the proton beam is being administered to the patient, therefore, this imaging must be done at the same time as treatment. Because the gamma rays are released where the proton beam interacts with the patient, the origins of these gamma rays are in the same position within the body as the proton beam. Therefore, if the origins of the rays can be traced back and compiled to construct an image, then physicians will have the ability to see exactly where the proton beam is delivering its dose of radiation. In a clinical setting, this imaging offers doctors the possibility of making adjustments to the treatment of patients in real time [3]. Being able to make these adjustments will allow them to better take advantage of the potential for a proton beam to deliver smaller doses of radiation to surrounding healthy tissue.

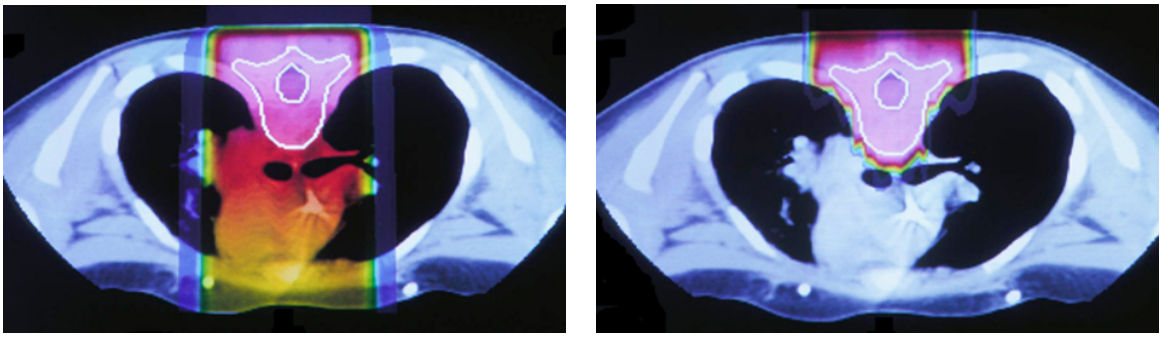
In order for the proton beam to hit the specified volume of tissue it is necessary for the patient to lie entirely still on the treatment table. It is also necessary for them to lie still during imaging as imaging occurs while the proton beam is being administrated. The position a patient needs to hold can often be difficult or awkward be in for long periods of time. Therefore, it is important that this imaging software runs as fast as possible [2,4]. This project explores the possibility for implementation of parallelism to the image reconstruction software through the development of an MPI algorithm to decrease this run time.

The remainder of this report is organized as follows: Section 2 describes proton beam therapy in greater detail and motivates the need for fast image reconstruction. Section 3 describes the SOE algorithm used for the image reconstruction and several versions of its implementation. Section 4 presents results of the reconstructions using the different versions of the code and their performance results. Section 5 summarizes our conclusions.

2 Problem

2.1 Proton Beam Therapy versus X-Ray Therapy

The idea to use proton beams during radiotherapy stemmed from Robert Wilson in 1946. He realized that while x-ray radiation delivers its lethal dose throughout the patient, proton beams reach their highest dose just before they stop, at what is called the Bragg peak with no radiation delivered to the tissue beyond. This gives physicians the ability to precisely target a tumor with high radiation doses by choosing where the proton beam will stop within the patient and can allow for the reduction of exposure of healthy tissue to radiation. The



X-ray Proton beam

Figure 2.1: X-ray radiation therapy vs. proton beam radiation therapy.

difference between radiation delivery depth can be seen in Figure 2.1.

Although there is a noticeable reduction to healthy tissue exposure to radiation with modern proton radiotherapy, this potential cannot always be fully taken advantage of because of small uncertainties in our ability to position the Bragg peak within the patient. Although the placement of the beam is carefully determined before treatment, there are variations within the atmosphere of treatment and the anatomy of the patient that can interfere with the correct placement of the proton beam.

By providing the physician with a system to image the proton beam path within the patient, it becomes possible for them to see precisely where the dose is delivered that day and to make adjustments as necessary. This imaging tool will allow physicians to better avoid delivering doses of radiation to healthy cells.

2.2 Prompt Gamma Emission

A proton beam can be imaged by studying the gamma rays emitted when the proton beam interacts with tissue. These gamma rays can only be emitted from tissue that the proton beam has come into contact with. Therefore, if the origins of the gamma rays can be determined and constructed into an image, it will be possible to see where the proton beam interacts within the patient.

In addition to imaging the full proton beam, it is also possible to reconstruct an image of what tissue in the body was effected by the beam. The gamma rays emitted have a characteristic energy based off the element that the proton beam interacted with in order to emit the ray. Therefore, it is possible to know the composition of the tissue that the proton beam has interacted with by measuring the number of gamma rays of a given energy characteristic to each element within the tissue.

2.3 Current Clinical Prototype Prompt Gamma Imaging System

This imaging system for prompt gamma emission creates 3D images using the coordinates taken by the Compton camera of the gamma rays emitted during the interaction. The

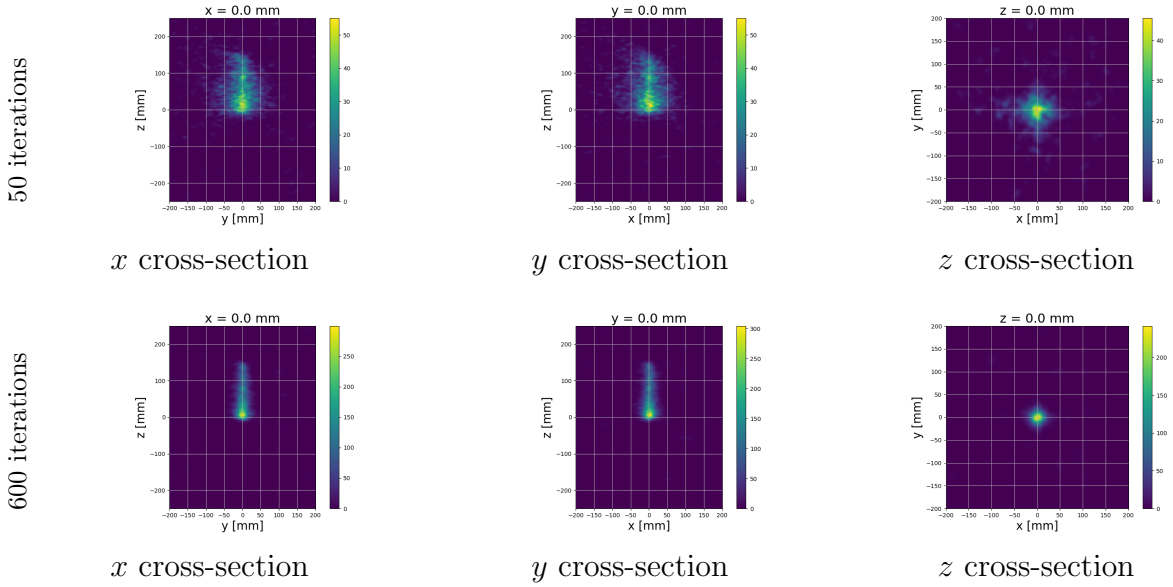


Figure 2.2: 3D image cross-sections of reconstructed proton beam after 50 and 600 iterations using the water input file.

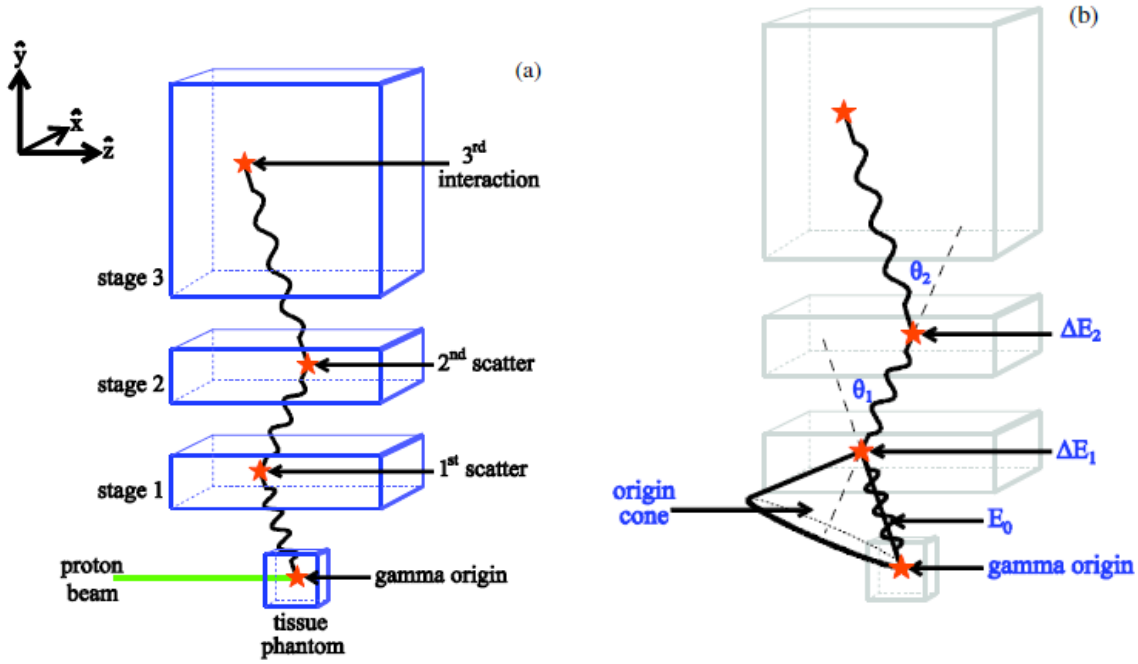
algorithm takes the information of these gamma rays from a measured data file and uses an iterative process to determine the origin of each ray. These origins are then plotted so that the output images can be used to determine the area of the patient which the proton beam interacted with. An example of the images used by a physician can be seen in Figure 2.2.

3 Algorithm and Implementation

3.1 SOE Algorithm

During data collection, gamma rays scatter into a specially designed camera known as a "Compton Camera" (CC) which records the coordinates and energy deposited by each gamma ray that interacts with the CC. Each gamma ray must interact with the camera at least two times to be useful for imaging. The 3D coordinates and energies deposited by the gamma rays are stored in a data file which is used to initialize the conic image reconstruction software. A line is drawn between the two points of ray impact and an angle is calculated and used to construct an initial cone as seen in Figure 3.1. The cone's surface encompasses all the possible origins for that ray. After these cones have been constructed, a random point from the base of the cone is chosen as an initial guess for the likely origin of the gamma ray. This initial point becomes the algorithm's first guess for the likely origin of that gamma ray. The 3D area containing the tissue phantom seen in Figure 3.1 is turned into a 3D density histogram divided into bins. Lastly the density histogram is populated by the counts of all likely origins contained in each 3D bin.

The conic reconstruction, based on the SOE algorithm [1], then improves the histogram



First stage reconstruction Second stage reconstruction

Figure 3.1: Gamma ray scatter and its image reconstruction.

iteratively by moving likely origins for each cone to more likely locations using the criterion that an already higher density in the histogram. Thus, in each iteration, for each cone, the algorithm chooses a new random point in the cone. If the random point has a higher density than the density of the current likely origin of that cone, it chooses the random point as the new likely origin of the cone. Correspondingly, the histogram is updated by decrementing the count in the old likely origin's bin and incrementing the count in the new likely origin's bin. These iterations are run until only a small fraction of likely origins are changed in one iteration.

3.2 Description of Code Implementation

Two input files are required to run this code. A configuration file controls various parameters for the code. Important examples include the total number of cones used, histogram coordinate boundaries in the x, y, and z directions, the total number of bins in the x, y, z directions, and the total number of iterations; notice that the number of iterations is fixed here by trial and error based on observing a small enough fraction of likely origin changes. The configuration file also specifies the name of the second input file, which is obtained from the gamma ray scattering into the CC. This is the measured data file containing a list of the energy deposited and x, y, and z coordinates of each gamma ray interaction occurring in the CC during the measurement. There can be two or three entries for a single gamma

ray because of the minimum set forth in the SOE algorithm.

The initial cones are created using the process described in Section 3.1 and their first likely origins are stored in an initial 3D density histogram. It then begins the iterations in which for each cone the algorithm picks a random coordinate point and checks if the histogram bin has a greater density then the cone's current bin. If so, the cones current bin is decremented, the new found bin is incremented, and the cone's likely origin is updated to be that of the new coordinate. After a set amount of varying iterations, the code writes the current iteration number, the time elapsed, the numbers of cone origins updated in the most recent iteration, and the ratio of cones updated vs. total cones to the "stdout" file. After the iterations have ceased, the code outputs the changes made into a file called `events.dat`. Additionally a file called `output.dat` is generated that provides coordinates of the last likely origin for each ray. Also the configuration file is saved for the run. All three output files are used for post-processing using Python2.7 that plots numerous figures. For comparison in Section 4, we use plots of likely origins of all cones.

3.3 OpenMP Algorithm

The code for this project was developed by Dr. Dennis Mackin. The number of cones used is typically large, so that it makes sense to speed up their processing by distributing work to several computational cores of a multi-core CPU. This was accomplished in the original version of the code by using the shared-memory parallel library OpenMP. The program first obtains the initial cones and initial density histogram via Section 3.1 in serial. At the start of the iterations, the OpenMP threads are started up using an OpenMP `parallel for` pragma applied to the loop over all cones. Then cones are then distributed to each thread, which distributes the main work of the code for a potential speedup as large as the number of threads used. The number of threads used is limited to the number of computational cores of one multi-core CPU, since OpenMP works only on a code that shares memory across all threads. This implies that the density histogram is shared between all threads at all times. Each thread will try to find a new origin, as in Section 3.2, for each cone, by comparing the density of the cone's current bin location to the new bin location. If the density of the new bin is greater, the code changes the likely origin of the cone and updates the histogram immediately by decrementing the old bin and incrementing the new bin. Since the histogram is in shared memory of all threads, OpenMP has to put a lock on the histogram, via a `critical` pragma, during this update, which forces other threads to idle during this time, thus decreasing parallel efficiency. But the latest histogram is always immediately available to all threads after the end of the lock, giving best convergence of the algorithm. After all cones have been processed the threads shut down and the output to stdout is done as stated in Section 3.2. This is repeated until all iterations have been completed and the output files are generated as they were in Section 3.2.

3.4 MPI Algorithm

The algorithm explained in Section 3.3 was changed based on the observation that the density of a likely origin of a cone is defined as its bin number's count, thus we work directly with the histogram from now on. We replaced the use of OpenMP by MPI (Message Passing Interface), so that the parallel code is not limited by the number of cores of one CPU any more, but can use several CPUs with distributed memory. The parallelism is achieved just like with OpenMP by distributing the large number of cones to the MPI processes. Each MPI process holds the cone database including the likely origin of each cone for its own cones only, hence this does not increase the overall memory usage of the code. Since the histogram is not too large in memory, it is possible to give each process a local copy of the density histogram. Doing so cuts out extra communication amongst processes that would come with each one having sections of the histogram and it avoids the need for a lock as required by the shared memory structure of OpenMP.

In each iteration, the MPI process picks a random bin for its cone and checks its density using the local histogram against the old bin number in the local histogram. However each process does not change their local density histogram; the changes are instead tracked by a local count vector and the local histogram remains unchanged. This means that all density checks are happening against the local histogram from the previous iteration rather than the latest histogram used in the OpenMP version. The local count vectors are then combined into a global count vector and scattered back to all processes using `MPI_Allreduce` every 1 iteration whereby they are merged into their local histograms. Similar to Section 3.3 the stdout is performed, but only by process 0. At present, our MPI code does not combine the cone data from all processes back into a serial data structure used for the output files `events.dat` and `output.dat` used for Python post-processing. Instead, the histogram data itself is output every 100 iterations to file, and Matlab post-processing was created that plots the histogram data, i.e., the counts of the bins, instead of the coordinates of the likely origins.

3.5 Modified MPI Algorithm

The only parallel cost of the MPI algorithm described in Section 3.4 is the use of the `MPI_Allreduce` command after every iteration. We created a modified MPI algorithm that instead of updating the histograms after 1 iteration only updates it every 10 iterations. This has the potential for speeding up the execution time of the code significantly, i.e., in principle by a factor 10. The potential downside is that the convergence might be slower, that is, that more iterations might be needed to reach a histogram of the same quality as measured by the number of changes of likely origins needed in an iteration.

3.6 Hardware Utilized

To measure performance of the software, we ran studies on the newest nodes of the Maya cluster in the UMBC High Performance Computing Facility (`hpcf.umbc.edu`). These nodes

contain two Intel E5-2650v2 Ivy Bridge (2.6 GHz, 20 MB cache) CPUs and 64 GB of memory each and are connected by a low latency quad data rate (QDR) Infiniband interconnect.

3.7 Intel VTune

To fully understand the time inefficiency of the code, we profiled the code using Intel's VTune software [5]. This performance profiler analyzes software in respect to many different areas of potential improvement. For our uses, we chose the HPC Performance Characterization analysis, as well as Hotspots analysis. These tests were run on the Texas Advanced Computing Center's (TACC) Stampede cluster, even though this report does not contain performance results from that cluster. These tests provide us data about how effectively our computationally intensive functions utilize memory, CPU, and hardware resources. This also provides information about the OpenMP and MPI parallelized performance efficiency within the code.

4 Results

The following are the results were obtained from the different algorithms that were formulated. Each algorithm utilized a deterministic configuration file utilizing 100,000 cones imaged by a histogram with $102 \times 102 \times 126$ bins.

4.1 OpenMP Algorithm

The OpenMP algorithm used to obtain the following results was described in Section 3.3 and written by Dr. Dennis Mackin in conjunction with Dr. Jerimy Polf.

4.1.1 OpenMP Algorithm Serial Run

The SOE algorithm uses iterations to progressively compute the most likely origins of the 100,000 cones that were measured by the CC during application of the proton beam. As more iterations are performed the results, or images, become more clearly defined. This convergence can be seen in Figure 4.1. Although at 100 iterations the image has some shape, it is clear that as the number of iterations increases this shape becomes more accurate. Additionally as the number of iterations increases the granularity of the results improve until the stopping point of 600 iterations whereby there is a distinct beam that can be seen.

The convergence in the algorithm happens, because with each iteration, the number of cones whose likely origin changes decreases. This means that as as the number of iterations increases, the amount of changes done to the overall histogram decreases. This behavior is quantified by the ratio of the number of changes to the total number of cones. The convergence of this ratio can be seen in Figure 4.2 which shows log output of sample iterations output to stdout. From this, we can infer that the points that the program is estimating to be the points of origin for the prompt gamma emission are becoming more accurate. This

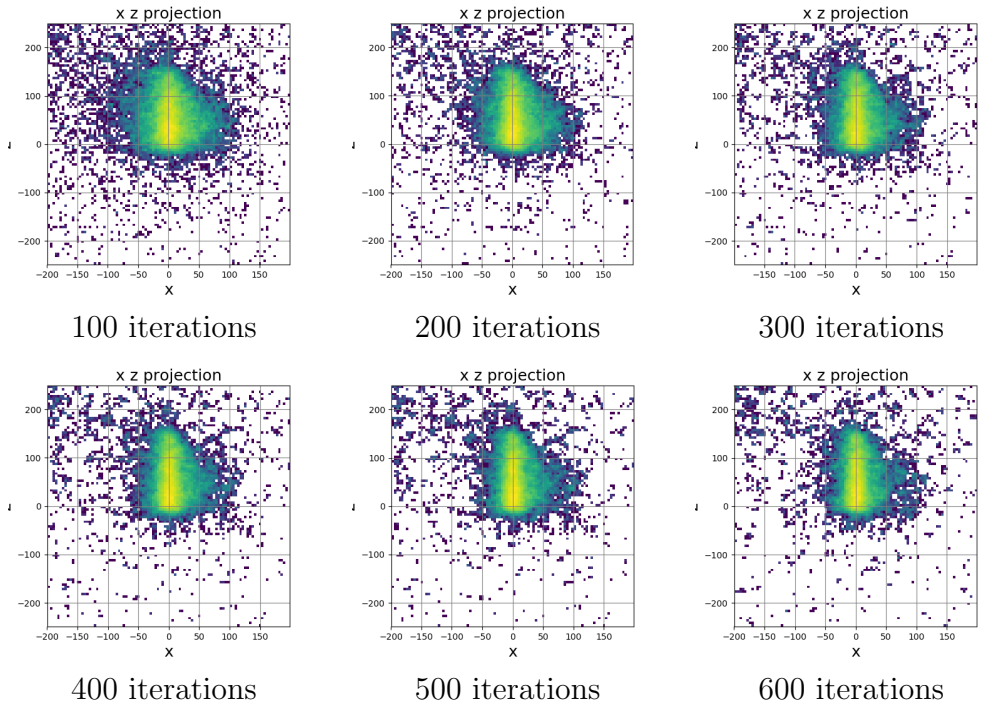


Figure 4.1: Reconstructed images computed by the OpenMP algorithm with 1 thread using Python post-processing up to 600 iterations.

would also mean that fewer changes are being made to the density matrix as more iterations are performed.

At the end of the log output in Figure 4.2, the code records the observed wall clock time in the line denoted by "Time Elapsed". This is the time that we use later in the timing studies of the code.

It is worth noting that the images in Figure 4.1 do not actually come from the same run of the code. This stems from the fact that the original code only outputs data for visualization at the final time. Hence, each plot in the figure is the final result of a run up to the specified number of iterations. It would be logical to say that if the 600 iteration run could have been post-processed at 500 iterations it should be the same as a 500 iteration run with the same parameters. This, however, is not the case because of the random number generator being used.

4.1.2 OpenMP Algorithm Multi-Threaded Runs

This code is designed to use OpenMP, which is an application programming interface that can be used on shared-memory multi-processor computers to allow for multi-threaded parallel processing. To maximize the speedup of the OpenMP algorithm it is necessary to use multiple threads. With OpenMP, memory cannot be pooled between nodes. Therefore, this algorithm only has the capability of using a single node at a time.

```
Iteration: 10, time 35, Number of Position Changes: 8799, ratio: 0.088
Iteration: 20, time 68, Number of Position Changes: 7485, ratio: 0.075
Iteration: 30, time 100, Number of Position Changes: 6498, ratio: 0.065
Iteration: 40, time 133, Number of Position Changes: 5784, ratio: 0.058
Iteration: 50, time 165, Number of Position Changes: 5339, ratio: 0.053
Iteration: 60, time 197, Number of Position Changes: 4971, ratio: 0.050
Iteration: 70, time 229, Number of Position Changes: 4469, ratio: 0.045
Iteration: 80, time 261, Number of Position Changes: 4336, ratio: 0.043
Iteration: 90, time 292, Number of Position Changes: 4119, ratio: 0.041
Iteration: 100, time 324, Number of Position Changes: 3944, ratio: 0.039
Iteration: 200, time 638, Number of Position Changes: 2990, ratio: 0.030
Iteration: 300, time 949, Number of Position Changes: 2684, ratio: 0.027
Iteration: 400, time 1257, Number of Position Changes: 2413, ratio: 0.024
Iteration: 500, time 1564, Number of Position Changes: 2195, ratio: 0.022
Iteration: 600, time 1870, Number of Position Changes: 2103, ratio: 0.021
--- Total Iterations: 600, time 1870,
Number of Position Changes: 2103, 100000, ratio: 0.021
Time Elapsed: 1884.54s
```

Figure 4.2: Iteration log output from the OpenMP algorithm with 1 thread.

Figure 4.3 shows the results of iterations of a 2 thread run. The reconstructed images have shapes similar to those seen in Figure 4.1. As was stated previously, it should be noted that the results in Figure 4.3 are not identical but the shapes of the beam are nearly indistinguishable and therefore acceptable results.

Similarly, the convergence behavior of the ratio for 2 threads in Figure 4.4 follows the same pattern as Figure 4.2. That is, as the number of iterations increases the ratio decreases and the image become more refined and suitable in shape. However, small variations of results between Figures 4.2 and 4.4 point to both the effect of differences in the random number sequences between runs and the slight differences possible between runs with different numbers of threads.

For our performance study, the code was run at 600 iterations on multiple threads ranging from 1 thread to 16 threads. It is important to note that even as the algorithm is run on more threads, the output image continues to be consistent with those obtained with serial runs. These results can be seen in Figure 4.5 in which the image holds steady for results using 4, 8, and 16 threads, also compared to the plot for 600 iterations in Figure 4.1 for 1 thread and in Figure 4.4 for 2 threads. This is an important comparison to make because although changes can be made to improve performance, to be acceptable these changes must have no effect on the quality of the results.

4.1.3 OpenMP Algorithm VTune Results

To initially understand the time inefficiency of the code, we used Intel's VTune profiling software to identify time intensive functions. The VTune Performance Characterization analysis identifies areas of OpenMP and/or MPI communication times as well as CPU and memory utilization. The Hotspots analysis was also used, which exhibits time intensive

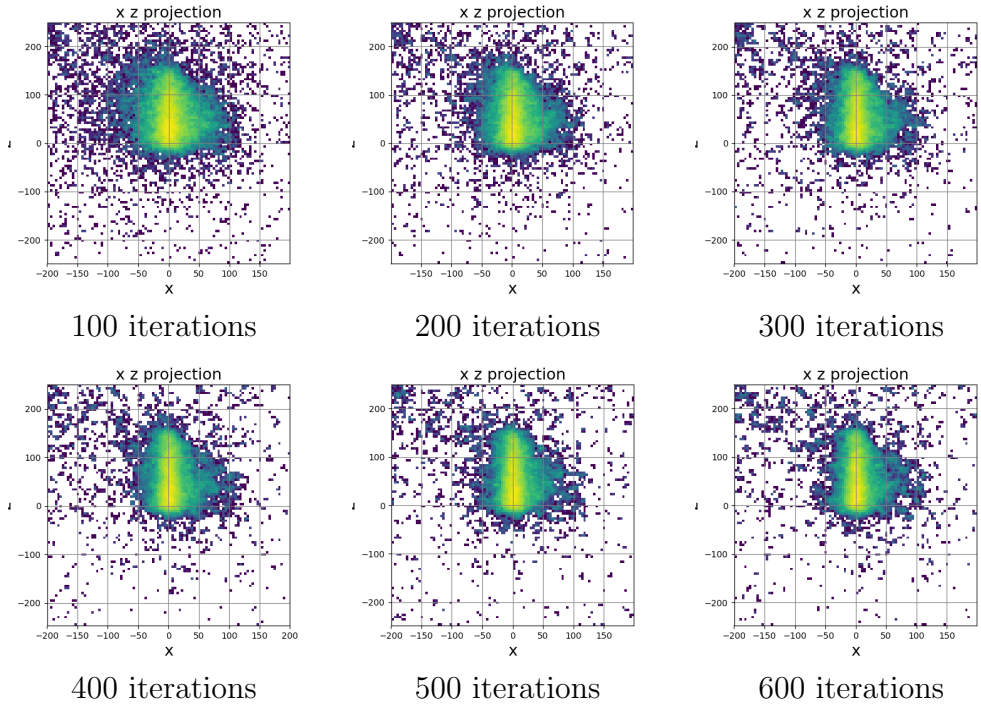
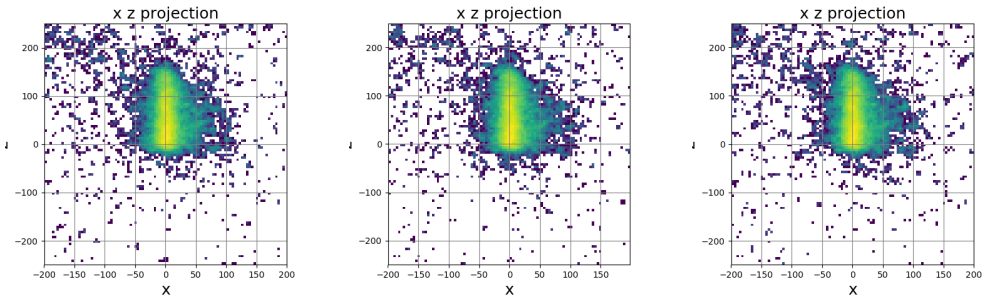


Figure 4.3: Reconstructed images computed by the OpenMP algorithm with 2 threads using Python post-processing up to 600 iterations.

```
Iteration: 10, time 14, Number of Position Changes: 8756, ratio: 0.088
Iteration: 20, time 27, Number of Position Changes: 7487, ratio: 0.075
Iteration: 30, time 39, Number of Position Changes: 6468, ratio: 0.065
Iteration: 40, time 51, Number of Position Changes: 5791, ratio: 0.058
Iteration: 50, time 63, Number of Position Changes: 5441, ratio: 0.054
Iteration: 60, time 74, Number of Position Changes: 5032, ratio: 0.050
Iteration: 70, time 86, Number of Position Changes: 4628, ratio: 0.046
Iteration: 80, time 97, Number of Position Changes: 4343, ratio: 0.043
Iteration: 90, time 109, Number of Position Changes: 4160, ratio: 0.042
Iteration: 100, time 120, Number of Position Changes: 3975, ratio: 0.040
Iteration: 200, time 231, Number of Position Changes: 3000, ratio: 0.030
Iteration: 300, time 338, Number of Position Changes: 2605, ratio: 0.026
Iteration: 400, time 444, Number of Position Changes: 2436, ratio: 0.024
Iteration: 500, time 548, Number of Position Changes: 2254, ratio: 0.023
Iteration: 600, time 652, Number of Position Changes: 2134, ratio: 0.021
--- Total Iterations: 600, time 652,
Number of Position Changes: 2134, 100000, ratio: 0.021
Time Elapsed: 661.26s
```

Figure 4.4: Iteration log output from the OpenMP algorithm using 2 threads up to 600 iterations.



4 threads 8 threads 16 threads

Figure 4.5: Comparison between reconstructed images after 600 iterations computed by the OpenMP algorithm with different numbers of threads.

functions regardless of parallelism.

Figure 4.6 shows VTune Performance Characterization for the OpenMP algorithm with 68 threads per node. Two functions are reported to have massive spin, or communication times, `kmp_wait_yield` and `kmp_barrier`, being 1838 seconds and 1521 seconds respectively. `kmp_barrier` and `kmp_wait_yield` are OpenMP library calls for communication between threads. These show us that while OpenMP may have optimized certain sections of the code, it ultimately led to longer run times due to communication times between threads. This spurred on the possibility of implementing MPI into the code to potentially cut these communication times down.

Figure 4.7 shows the VTune Hotspots Analysis with 68 threads per node. This type of analysis maps out the functions that are the most time intensive within the code without regards to communication times. We can see that the `getDensity` function called from `DensityMatrix` is solely responsible for the majority of time use when the executable is run. When investigating the underlying code for `getDensity`, we quickly realized that the other three most time intensive functions were directly related and/or called from `getDensity`. For example, `getBinNumber`, and the `ASHDensity::getDensity`, the second and fourth most time intensive function, are directly called from the `getDensity` function. This is a logical process because the `DensityMatrix` object's `getDensity` function calls `ASHDensity` object's `getDensity` which also calls their `getBinNumber` method. So this cascading spin time is a direct result of all the threads chasing down several sets of nested pointers in C++. The fourth time intensive function is the `updateMatrix` function. This function's impact is the result of a `critical` pragma that wraps around it.

Both analysis types showed time critical areas to focus our attention on. The Performance Characterization results clearly showed that OpenMP library calls were costing massive wait times in order for threads to synchronize, while the Hotspots Analysis showed functions that could potentially benefit from parallelization. With these results in hand, a clear course of action could be mapped out.

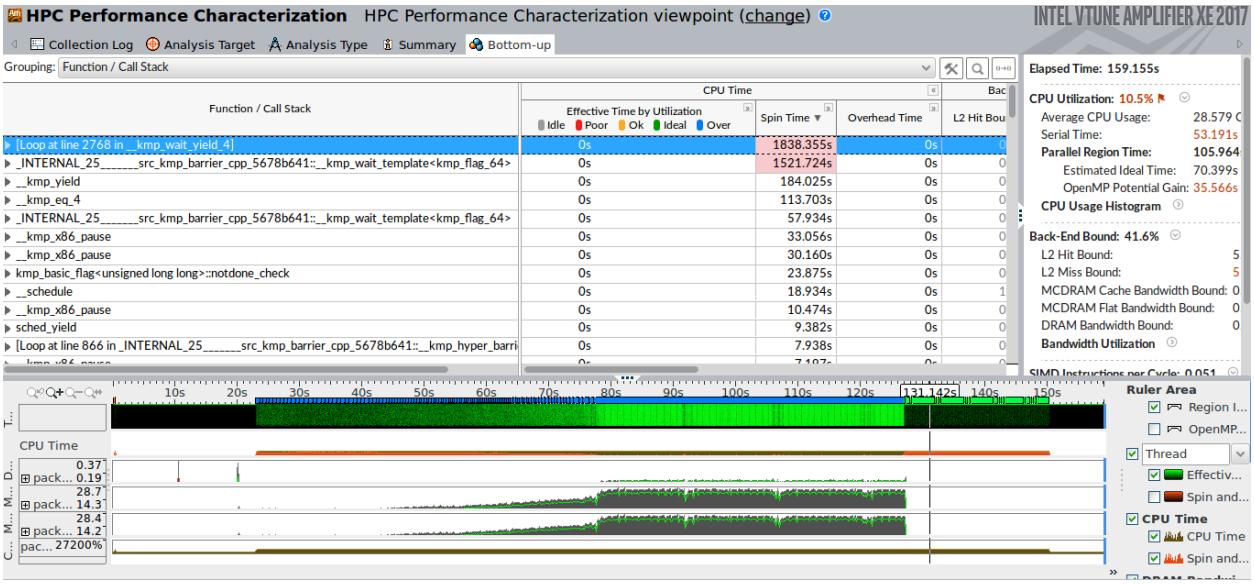


Figure 4.6: VTune Performance Characterization for the OpenMP algorithm with 68 threads per node.

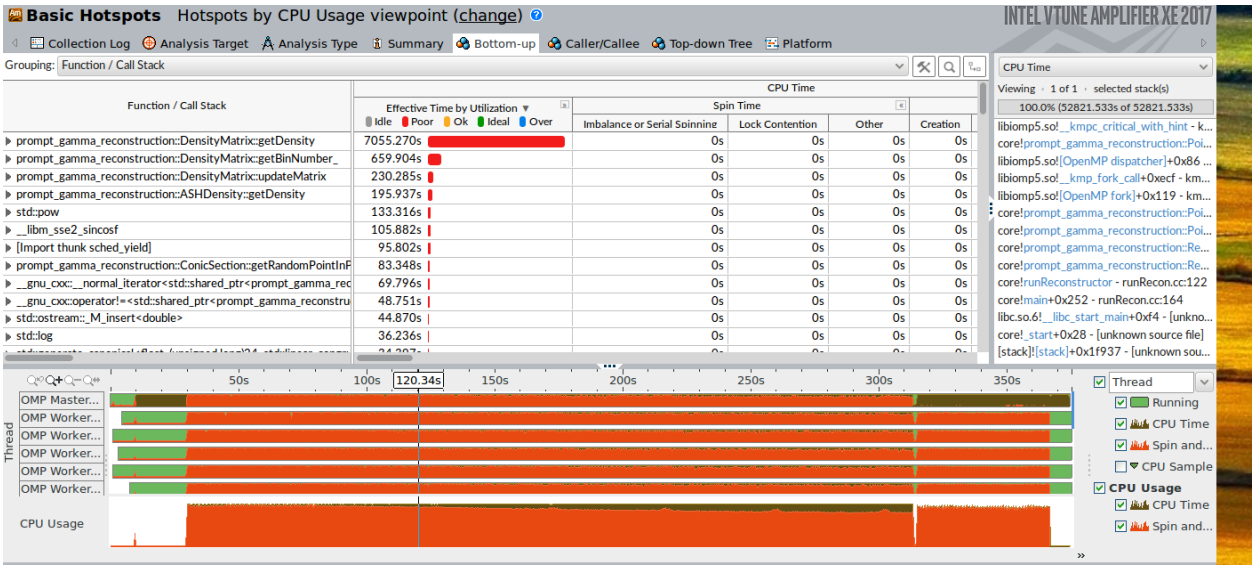


Figure 4.7: VTune Hotspots Analysis for the OpenMP algorithm with 68 threads per node.

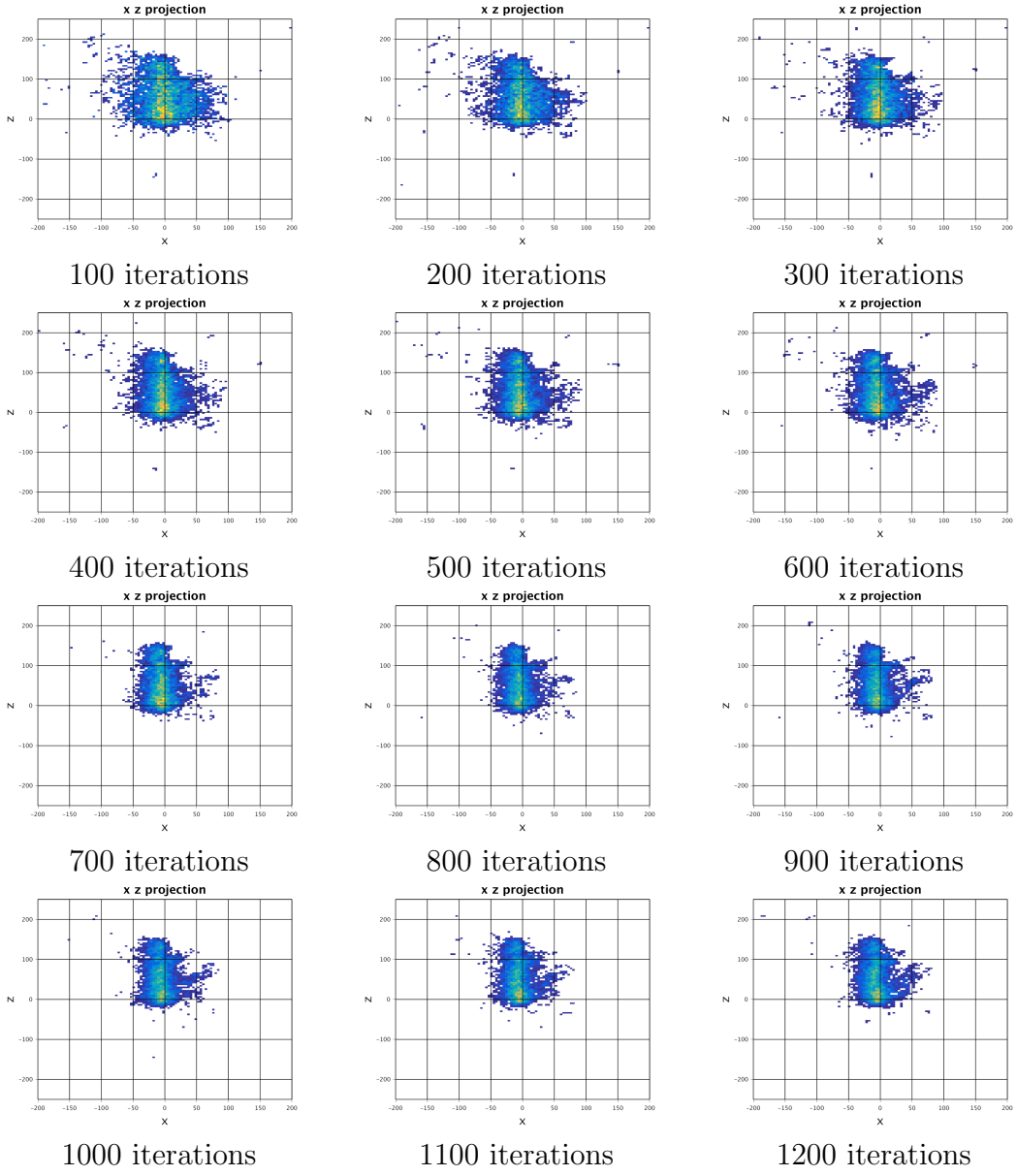


Figure 4.8: Reconstructed images computed by the MPI algorithm with 1 process using Matlab post-processing up to 1200 iterations.

4.2 MPI Algorithm

The MPI algorithm used to obtain the following results was described in Section 3.4.

4.2.1 MPI Algorithm Serial Run

Figure 4.8 represents the serial run, meaning 1 process per 1 node up to 1200 iterations. In the first few iterations, the first image shows a rather wide cloud of points. As iterations

```
Iteration: 10, time 18, Number of Position Changes: 18815, ratio: 0.188
Iteration: 20, time 34, Number of Position Changes: 19761, ratio: 0.198
Iteration: 30, time 50, Number of Position Changes: 16207, ratio: 0.162
Iteration: 40, time 66, Number of Position Changes: 13412, ratio: 0.134
Iteration: 50, time 82, Number of Position Changes: 11338, ratio: 0.113
Iteration: 60, time 98, Number of Position Changes: 9841, ratio: 0.098
Iteration: 70, time 114, Number of Position Changes: 8835, ratio: 0.088
Iteration: 80, time 129, Number of Position Changes: 7675, ratio: 0.077
Iteration: 90, time 145, Number of Position Changes: 7198, ratio: 0.072
Iteration: 100, time 161, Number of Position Changes: 6655, ratio: 0.067
Iteration: 200, time 315, Number of Position Changes: 4479, ratio: 0.045
Iteration: 300, time 466, Number of Position Changes: 3821, ratio: 0.038
Iteration: 400, time 644, Number of Position Changes: 3390, ratio: 0.034
Iteration: 500, time 893, Number of Position Changes: 3146, ratio: 0.031
Iteration: 600, time 1139, Number of Position Changes: 3018, ratio: 0.030
Iteration: 700, time 1384, Number of Position Changes: 2889, ratio: 0.029
Iteration: 800, time 1629, Number of Position Changes: 2837, ratio: 0.028
Iteration: 900, time 1873, Number of Position Changes: 2769, ratio: 0.028
Iteration: 1000, time 2116, Number of Position Changes: 2688, ratio: 0.027
--- Total Iterations: 1200, time 2598,
Number of Position Changes: 2688, 100000, ratio: 0.027
Time Elapsed:  1918s
```

Figure 4.9: Iteration log output from original MPI algorithm with 1 process up to 1200 iterations.

start to hit the midpoint, the beam shape is much narrower and better defined. The last three images of the figure look very similar in both shape and coloring. Visually these images show that the algorithm has converged because the differences between the images are hardly noticeable.

As detailed in Section 3.4, the MPI algorithm is different than the OpenMP algorithm in that the histogram is updated at the end of each iteration, not after updating each cone. This has the potential for causing slower convergence, and we therefore report studies with twice as many iterations than used for the OpenMP studies. The images in Figure 4.8 confirm a slightly slower convergence, in particular at the beginning, but certainly the final three images are very stable. We might recommend using 700 iterations instead of 600 in production runs.

We note that the images in Figure 4.8 are produced by Matlab instead of Python post-processing. It reflects the difficulty to interface the C code associated with the MPI functions with the C++ data structure that undergird the file output for Python post-processing. But we also gain the advantage that one run of the code now outputs in steps of 100 iterations, thus all images in Figure 4.8 come from the same run.

Figure 4.9 shows the log file of the run. The log confirms the slightly slower convergence in the beginning, and it eventually converges to 0.027.

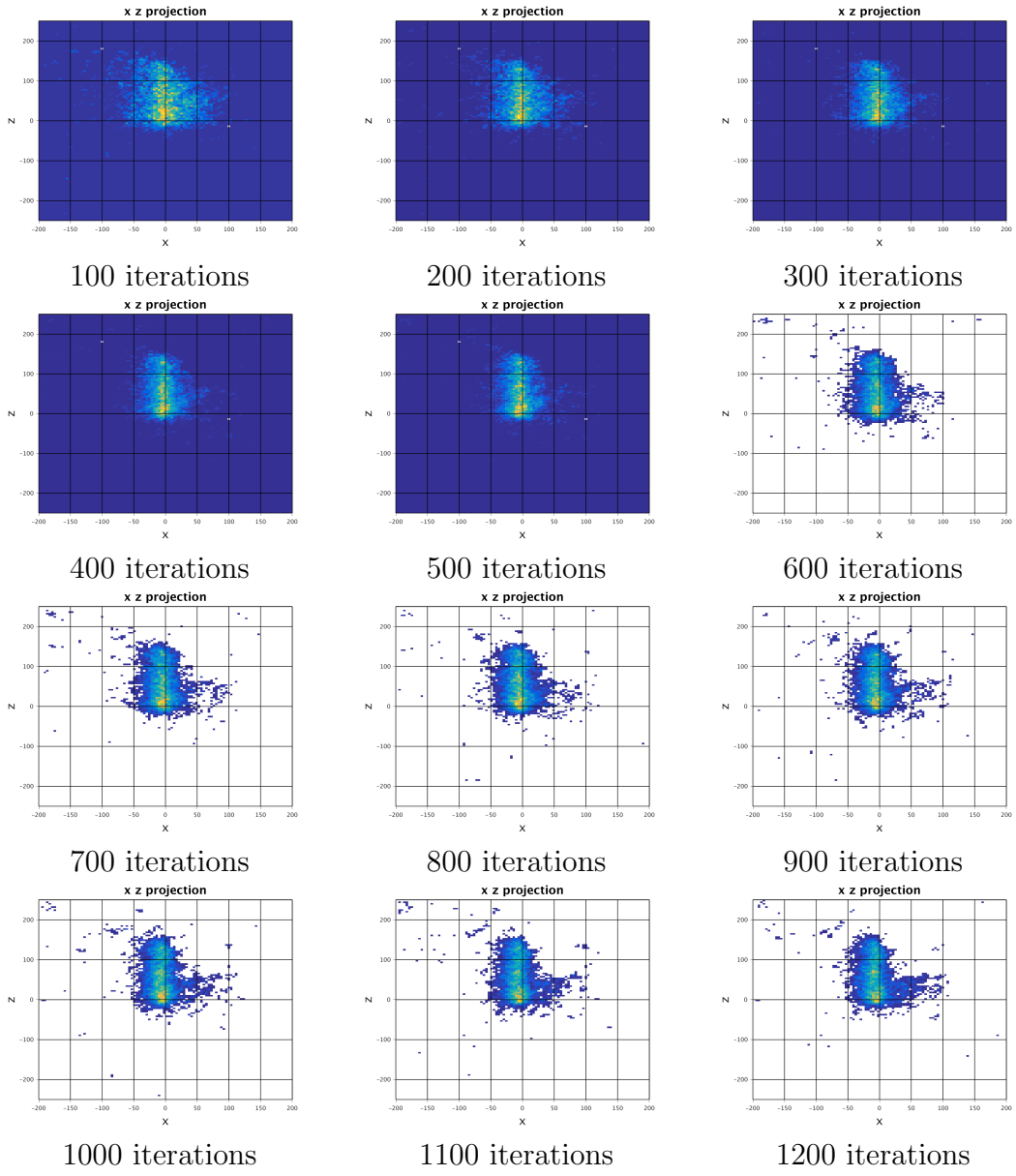


Figure 4.10: Reconstructed images computed by the MPI algorithm with 2 processes using Matlab post-processing up to 1200 iterations.

4.2.2 MPI Algorithm Multi-Process Runs

Figure 4.10 shows the iterations of the parallel MPI algorithm with 2 processes. The blue color in the first images is an artifact of the colormap used in Matlab, however, it flags that some numbers being plotted are negative. This is not a sensible value for the population of a bin in a histogram. Inspection of the initialization eventually clarified that the underlying cause are histograms on each process that are not necessarily identical to histograms on other processes. This in turn is actually caused by random number sequences that are independent

```
Iteration: 10, time 11, Number of Position Changes: 15648, ratio: 0.156
Iteration: 20, time 21, Number of Position Changes: 11890, ratio: 0.119
Iteration: 30, time 30, Number of Position Changes: 9857, ratio: 0.099
Iteration: 40, time 40, Number of Position Changes: 8425, ratio: 0.084
Iteration: 50, time 49, Number of Position Changes: 7682, ratio: 0.077
Iteration: 60, time 58, Number of Position Changes: 6984, ratio: 0.070
Iteration: 70, time 68, Number of Position Changes: 6414, ratio: 0.064
Iteration: 80, time 77, Number of Position Changes: 6039, ratio: 0.060
Iteration: 90, time 86, Number of Position Changes: 5682, ratio: 0.057
Iteration: 100, time 96, Number of Position Changes: 5574, ratio: 0.056
Iteration: 200, time 187, Number of Position Changes: 4076, ratio: 0.041
Iteration: 300, time 277, Number of Position Changes: 3607, ratio: 0.036
Iteration: 400, time 366, Number of Position Changes: 3195, ratio: 0.032
Iteration: 500, time 455, Number of Position Changes: 3022, ratio: 0.030
Iteration: 600, time 543, Number of Position Changes: 2875, ratio: 0.029
Iteration: 700, time 631, Number of Position Changes: 2740, ratio: 0.027
Iteration: 800, time 720, Number of Position Changes: 2766, ratio: 0.028
Iteration: 900, time 808, Number of Position Changes: 2637, ratio: 0.026
Iteration: 1000, time 896, Number of Position Changes: 2641, ratio: 0.026
--- Total Iterations: 1200, time 1072,
Number of Position Changes: 2572, 100000, ratio: 0.026
Time Elapsed:  1085s
```

Figure 4.11: Iteration log output from original MPI algorithm with 2 processes up to 1200 iterations.

and potentially different on each process. We do not report the results from the fixed code here, but include this fix in the results in the following Section 4.3. But the images in Figure 4.10 actually demonstrate the robustness of the SOE algorithm, in that it overcomes the erroneous negative histogram entries in the beginning and in fact converges to the correct result in the final three images.

This exact point can be seen in Figure 4.11 where the final ratio is very close to the 0.026.

4.3 Modified MPI Algorithm

The modified MPI algorithm used to obtain the following results was described in Section 3.5. This reflects the observation that the major cost of parallelism with MPI is the cost of communication. For the original MPI algorithm, this cost amounts to a call to `MPI_Allreduce` after every iteration. The modified MPI algorithm decreases this cost by communicating only every 10 iterations.

4.3.1 Modified MPI Algorithm Serial Run

The major concern with this algorithm is possible degradation of image quality. With the processes only having access to a histogram that was developed, at most, 10 iterations prior to their current iteration there was a possibility that the lack of updates cause the images to degrade to an unusable state. Also the error that was causing the negative values in

the histogram for the original MPI algorithm was corrected in this modified MPI algorithm. Running the algorithm in serial should show some minor signs of this degradation if it was present. Figure 4.12 starts out with a rough cloud of points which form into a wider beam around the 400 iteration mark. The beam starts to narrow and become more defined at around 800 iterations. After that, the image seems to remain similar to previous images. Visually the convergence can be seen from this image set, starting out with a very rough image and going to something much more refined and recognizable. The modified MPI algorithm in serial shows the same behavior as the original MPI algorithm in serial as seen in Figure 4.8.

Furthermore, the log output in Figure 4.13 for the modified serial run shows that the ratio is a little higher than original MPI serial run in Figure 4.9. Despite this, the general behavior of convergence remains similar and the time taken was about the same.

4.3.2 Modified MPI Algorithm Multi-Process Runs

Figure 4.14 shows some image degradation again in the first images, compared to the original OpenMP algorithm. The cloud starts out similar to before but persists for a couple hundred iterations beyond the first image. However by 700 or 800 iterations the beam is well formed albeit slightly wider. The images visually converge at around 900 iterations which coincides with the output in Figure 4.15. We might recommend to use again a slightly larger number of iterations in production runs, such as 700 or 800 iterations instead of the originally used 600 for the OpenMP algorithm. We note that all images in Figure 4.14 demonstrate the success of the fix to the MPI code, since no more negative numbers in the histogram appear.

4.3.3 MPI VTune

To gauge the efficiency of the new MPI implemented algorithm, the code was profiled once more with Intel's VTune software. We used the same analysis specifications to determine if communication times were indeed faster than the OpenMP algorithm, as well as watching our previous time intensive functions.

Figure 4.16 shows the VTune Performance Characterization of the modified MPI algorithm with 4 processes per node. Implementing MPI did indeed improve the communication times, or 'spin' far greater than expected. The first analysis showed that OpenMP communication took a staggering 1838 seconds, while our MPI algorithm had a maximum communication time of only 0.130 seconds. This affirmed our decision to introduce MPI into the algorithm.

Figure 4.17 shows the VTune Hotspots Analysis for the same modified MPI algorithm with 4 processes per node. This Hotspots Analysis produced similar results to the previous run. While `getDensity` and its related function calls were still responsible for being the most time critical calls, the parallelism imposed onto these helped to reduce the footprint. The introduction of MPI eliminated the waiting associated with `updateMatrix` but did not smoothly allow the elimination of the nested object calls associated with `getDensity`. Converting more C++ object code into C code would allow `getDensity` to be made much for efficient or removed alltogether.

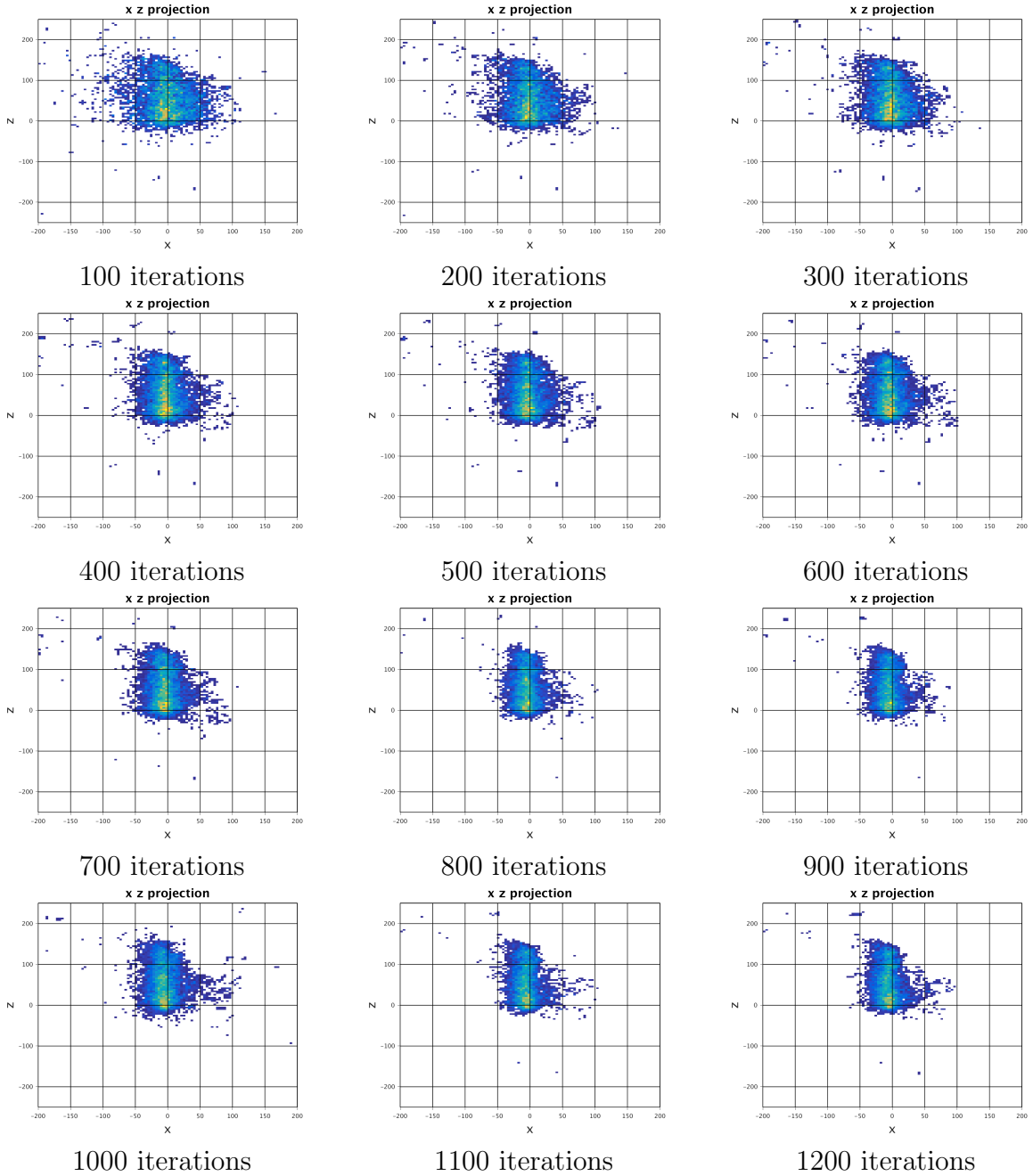


Figure 4.12: Reconstructed images computed by the modified MPI algorithm with 1 process up to 1200 iterations using Matlab post-processing.

```
Iteration: 10, time 18, Number of Position Changes: 18815, ratio: 0.188
Iteration: 20, time 34, Number of Position Changes: 19761, ratio: 0.198
Iteration: 30, time 50, Number of Position Changes: 16207, ratio: 0.162
Iteration: 40, time 66, Number of Position Changes: 13412, ratio: 0.134
Iteration: 50, time 82, Number of Position Changes: 11338, ratio: 0.113
Iteration: 60, time 98, Number of Position Changes: 9841, ratio: 0.098
Iteration: 70, time 114, Number of Position Changes: 8835, ratio: 0.088
Iteration: 80, time 129, Number of Position Changes: 7675, ratio: 0.077
Iteration: 90, time 145, Number of Position Changes: 7198, ratio: 0.072
Iteration: 100, time 161, Number of Position Changes: 6655, ratio: 0.067
Iteration: 200, time 315, Number of Position Changes: 4479, ratio: 0.045
Iteration: 300, time 466, Number of Position Changes: 3821, ratio: 0.038
Iteration: 400, time 644, Number of Position Changes: 3390, ratio: 0.034
Iteration: 500, time 893, Number of Position Changes: 3146, ratio: 0.031
Iteration: 600, time 1139, Number of Position Changes: 3018, ratio: 0.030
Iteration: 700, time 1384, Number of Position Changes: 2889, ratio: 0.029
Iteration: 800, time 1629, Number of Position Changes: 2837, ratio: 0.028
Iteration: 900, time 1873, Number of Position Changes: 2769, ratio: 0.028
Iteration: 1000, time 2116, Number of Position Changes: 2688, ratio: 0.027
--- Total Iterations: 1200, time 2598,
Number of Position Changes: 2688, 100000, ratio: 0.027
Time Elapsed:  2610s
```

Figure 4.13: Iteration log output from the modified MPI algorithm using 1 processes up to 1200 iterations.

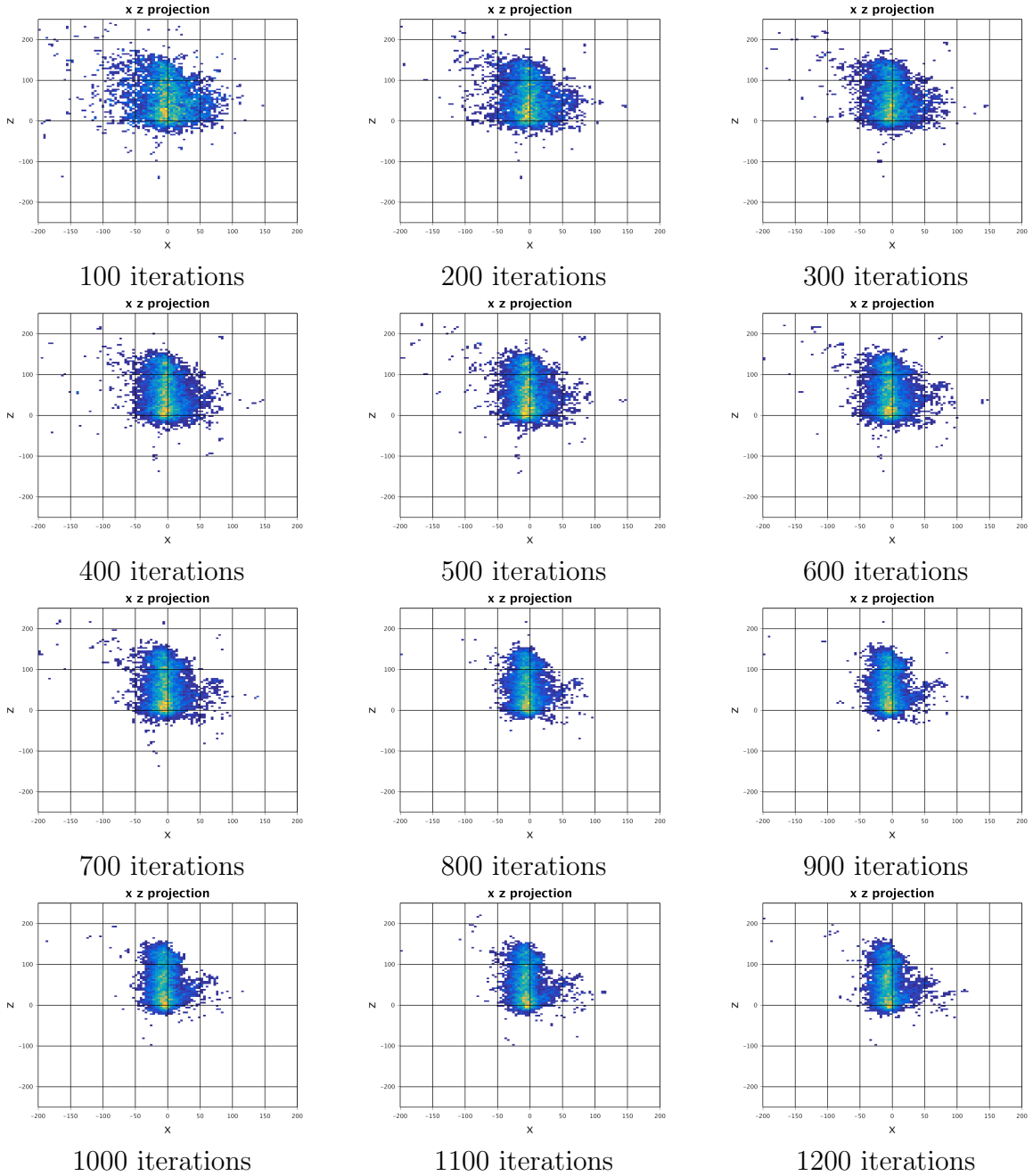


Figure 4.14: Reconstructed images computed by the modified MPI algorithm with 2 processes up to 1200 iterations using Matlab post-processing.

```
Iteration: 10, time 11, Number of Position Changes: 18727, ratio: 0.187
Iteration: 20, time 19, Number of Position Changes: 19703, ratio: 0.197
Iteration: 30, time 27, Number of Position Changes: 16209, ratio: 0.162
Iteration: 40, time 35, Number of Position Changes: 13374, ratio: 0.134
Iteration: 50, time 43, Number of Position Changes: 11165, ratio: 0.112
Iteration: 60, time 52, Number of Position Changes: 9851, ratio: 0.099
Iteration: 70, time 60, Number of Position Changes: 8658, ratio: 0.087
Iteration: 80, time 67, Number of Position Changes: 7696, ratio: 0.077
Iteration: 90, time 75, Number of Position Changes: 7139, ratio: 0.071
Iteration: 100, time 84, Number of Position Changes: 6783, ratio: 0.068
Iteration: 200, time 162, Number of Position Changes: 4493, ratio: 0.045
Iteration: 300, time 240, Number of Position Changes: 3858, ratio: 0.039
Iteration: 400, time 316, Number of Position Changes: 3493, ratio: 0.035
Iteration: 500, time 392, Number of Position Changes: 3081, ratio: 0.031
Iteration: 600, time 468, Number of Position Changes: 2936, ratio: 0.029
Iteration: 700, time 543, Number of Position Changes: 2874, ratio: 0.029
Iteration: 800, time 618, Number of Position Changes: 2748, ratio: 0.027
Iteration: 900, time 692, Number of Position Changes: 2677, ratio: 0.027
Iteration: 1000, time 767, Number of Position Changes: 2651, ratio: 0.027
--- Total Iterations: 1200, time 915,
Number of Position Changes: 2651, 100000, ratio: 0.027
Time Elapsed:   928s
```

Figure 4.15: Iteration log output from the modified MPI algorithm using 2 processes up to 1200 iterations.

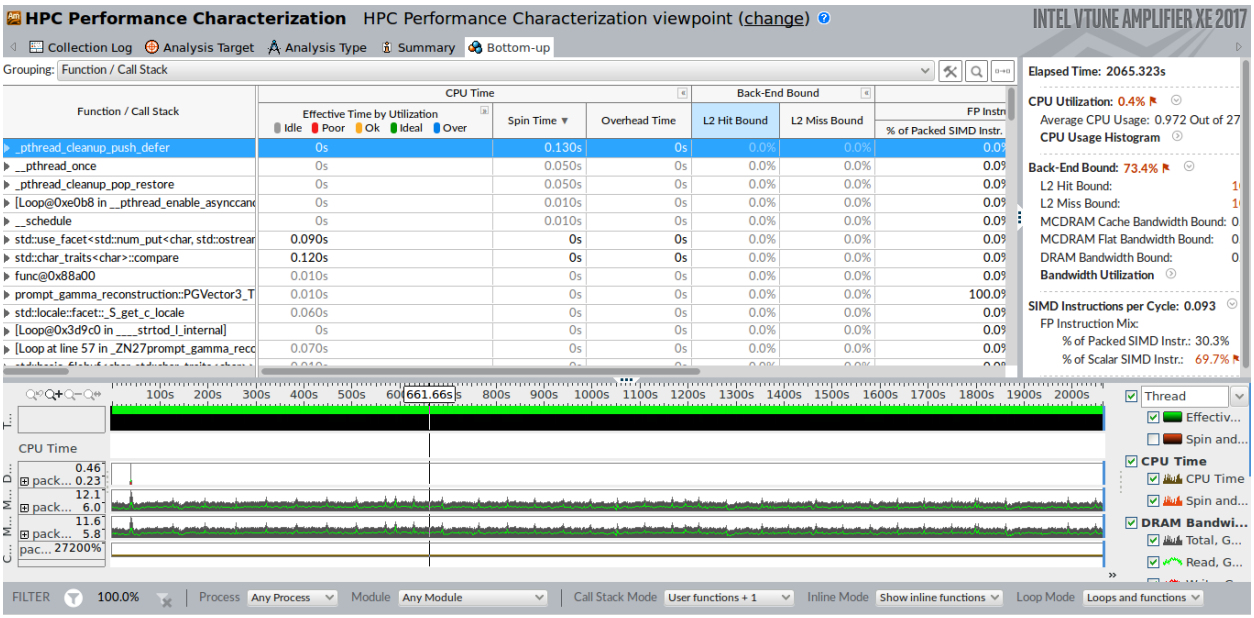


Figure 4.16: VTune Performance Characterization for the Modified MPI algorithm with 4 processes per node.

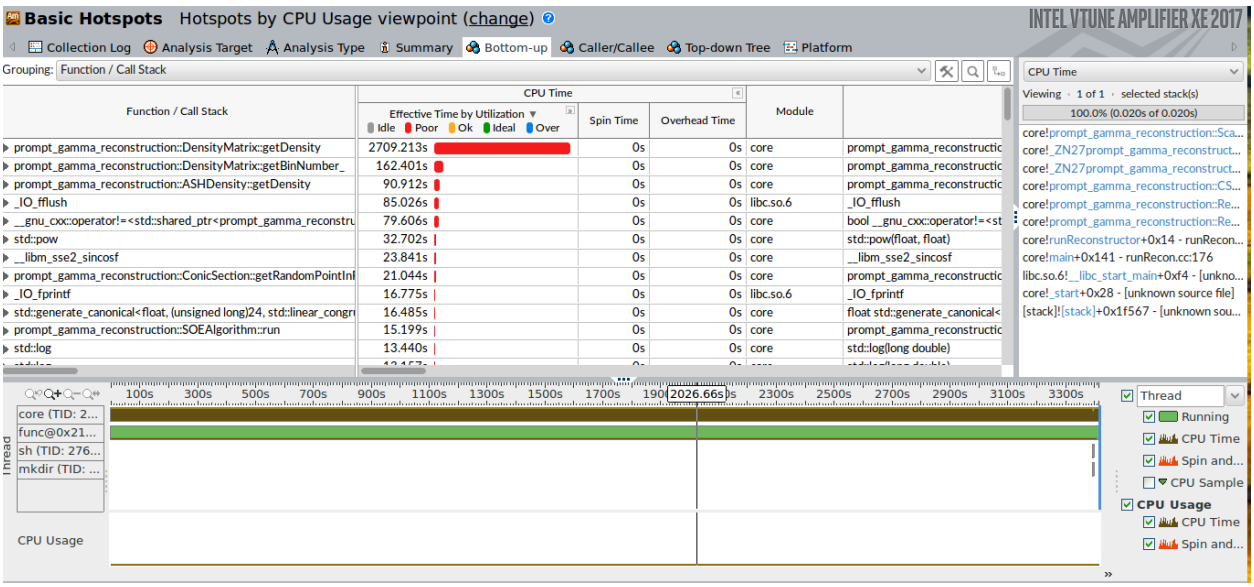


Figure 4.17: VTune Hotspots Analysis for the Modified MPI algorithm with 4 processes per node.

4.4 Performance Studies

To understand the impact of the changes we implemented on the run time, a study was created to compare the performance of the original OpenMP algorithm, our (original) MPI algorithm, and the modified MPI algorithm. Table 4.1 shows the timing results of our study. Note that the performance studies used 600 iterations for all algorithms to provide a direct comparison of the times.

The first studies ran were to determine the performance of the OpenMP algorithm. The code was tested when run on one core using 1, 2, 4, 8, and 16 threads to get an idea of the speedup. With 1 thread, the code ran a staggering 1885 seconds, or roughly 30 minutes. As the number of threads increased, the run times gained significant speedup as had been expected. For example, at 2, 4, and 8 threads, the run times had been reduced by more than fifty percent each time. While still improving at 16 threads, times decreasing from 188 seconds to 105 seconds, is not the same rate of improvement as can be observed in the first 4 increments. This speed of 105 seconds is the best possible run time observed for this algorithm, since OpenMP is limited to one node.

The first changes to the code allowed for the implementation of MPI and disabled all OpenMP pragmas. These adjustments allowed for both the use of distributed memory and the multiple processes across nodes for each job. In order to assess the benefits of MPI, tests were run using numbers of processes that mirror the number of threads in the previous study. That is, both studies use two different software libraries, but access the same hardware cores in the CPUs on one node; beyond 16 processes, MPI uses more nodes and thus more hardware. The performance of our MPI code immediately saw improvement, bringing the run time to 1592 seconds for one process due to other improvements of the code. At 2 processes, the time dropped by roughly a third down to 546 seconds, similar to the speedup seen with OpenMP. For the rest of the runs, from 8 to 64 processes, the code did not speedup as much as had been expected, reaching its fastest time at 354 seconds for 8 processes and rising to 430 seconds at 64 processes. This exhibits that parallel communications eventually overwhelm the increased efficiency of splitting up the computational work between processes.

The code was then adjusted so that the histogram would be updated after every ten iterations. This change allowed for a significant speedup. At one process, the algorithm fully ran in 985 seconds, almost half of that for the OpenMP with one thread, due to additional improvements in the code. The increases in processes showed reduction in run times, ending on 64 processes with 83 seconds.

Table 4.1: Observed wall clock time in seconds for reconstruction with 600 iterations.

Computational cores	1	2	4	8	16	32	64
OpenMP multi-threading	1885	661	344	188	105	N/A	N/A
Original MPI algorithm	1569	546	372	354	511	477	430
Modified MPI algorithm	985	480	277	194	147	113	83

5 Conclusions

The original code provided by Dr. Polf and Dr. Mackin implements an algorithm using the shared-memory parallel library OpenMP, which is constrained to all cores of 1 node which limits performance. In order to see a significant speedup, this code needed to given the ability to run on multiple nodes. In order to have this capability, the algorithm was modified so that it could be run using the distributed-memory parallel communication library MPI. Modifications to the algorithm included distributing the work associated with the large number of cones in each iteration to the parallel processes. Each process works on a section of the total number of cones, thus the work is distributed. The first version of the MPI algorithm updates the global histogram updated after every iteration. This was changed in the modified MPI algorithm to allow for further speedup by only updating the histogram every 10 iterations. Various code improvements helped already improve the serial time from 1885 seconds to 985 seconds. In parallel, OpenMP provided a run time of 105 seconds on 1 node. By implementing an algorithm designed to use MPI, we were able to move beyond 1 node and obtain the best run time of 85 seconds.

By implementing MPI in this algorithm this study has given the algorithm potential for further speedup. MPI's distributed memory and multiple-processing power has been shown to be capable to scale down run times. For these reasons, the algorithm developed here has great potential to be sped up as necessary to be used in a clinical setting with more code improvements and with the use of hybrid MPI+OpenMP, which we did not explore yet.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (hpcreu.umbc.edu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2017. This program is funded by the National Science Foundation (NSF), the National Security Agency (NSA), and the Department of Defense (DOD), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-author James Della-Giustina was supported in part, by the Math Computer Inspired Scholars program, through funding from the National Science Foundation and also the Constellation STEM Scholars Program, funded by Constellation Energy. Co-authors Johnlemuel Casilag and Aniebiet Jacob were supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. Graduate assistant Carlos Barajas was supported by UMBC. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] Andriy Andreyev, Arkadiusz Sitek, and Anna Celleri. Fast image reconstruction for Compton camera using stochastic origin ensemble approach. *Med. Phys.*, 38:429–35, 2011.

[2] Fernando X. Avila-Soto, Alec N. Beri, Eric Valenzuela, Abenezer Wudenhe, Ari Rapkin Blenkhorn, Jonathan S. Graf, Samuel Khuvis, Matthias K. Gobbert, and Jerimy Polf. Parallelization for fast image reconstruction using the stochastic origin ensemble method for proton beam therapy. Technical Report HPCF–2015–27, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[3] Dennis Mackin, Steve Peterson, Sam Beddar, and Jerimy Polf. Evaluation of a stochastic reconstruction algorithm for use in Compton camera imaging and beam range verification from secondary gamma emission during proton therapy. *Phys. Med. Biol.*, 57:3537–3553, 2012.

[4] Jerimy C. Polf and Katia Parodi. Imaging particle beams for cancer treatment. *Physics Today*, 68(10):28–33, 2015.

[5] Intel Developer Zone. Intel VTune Amplifier, 2017. Documentation at the URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe-support/documentation.