Speedup Potential for Reconstruction Techniques for Prompt Gamma Imaging During Proton Radiotherapy

James Della-Giustina[*a], *Johnlemuel Casilag*[b], *Elizabeth Gregorio*[c], *Aniebiet Jacobs*[b]

[a]*Department of Information Technology & Computer Science, Community College of Baltimore County, Baltimore, MD*
[b]*Department of Computer Science & Electrical Engineering, University of Maryland, Baltimore County, Baltimore, MD*
[c]*Department of Physics, Hamline University, St. Paul, MN*
[d]*Department of Mathematics and Statistics, University of Maryland, Baltimore County, Baltimore, MD*
[e]*Department of Radiation Oncology, University of Maryland School of Medicine, Baltimore, MD*
[f] *Department of Radiation Physics, The University of Texas MD Anderson Cancer Center, Houston, Texas*

Students: jdella@umbc.edu, cas6@umbc.edu, anie1@umbc.edu, egregorio01@gmail.com*
Research Assistant: Carlos Barajas[d], *barajasc@umbc.edu*
Faculty Mentor: Matthias K. Gobbert[d], *gobbert@umbc.edu*
Clients: Dennis Mackin[f] , *dsmackin@mdanderson.org, Jerimy Polf*[e], *jpolf@umm.edu*

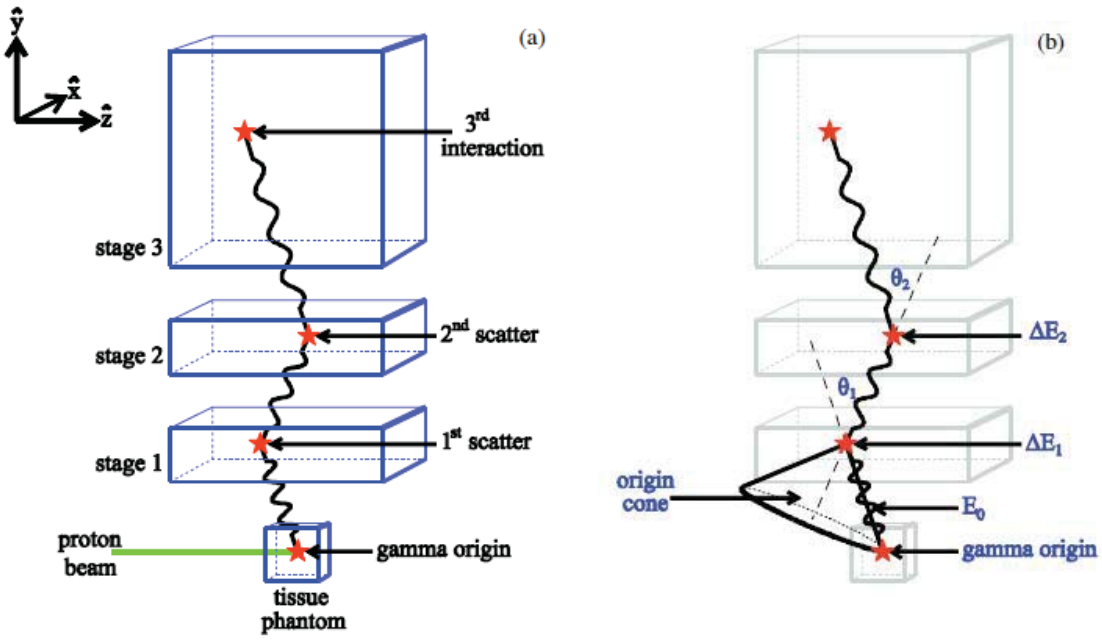
ABSTRACT
Proton beam radiation treatment was first proposed by Robert Wilson in 1946. The advantage of proton beam radiation is that the lethal dose of radiation is delivered by a sharp increase toward the end of the beam range. This sharp increase, known as the Bragg peak, allows for the possibility 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.

KEYWORDS
Proton Beam Therapy; Image Reconstruction; SOE Algorithm; Parallel Computing; High Performance Computing; Medical Imaging; Prompt Gamma Imaging; Radiotherapy

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



First stage reconstruction Second stage reconstruction

Figure 1. Gamma ray scatter and its image reconstruction.

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.[1] 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 administered. 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, 3] 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 the SOE algorithm used for the image reconstruction and different versions of its implementation. **Section 3** presents results of the reconstructions using the different versions of the code and their performance results. **Section 4** summarizes our conclusions.

2. ALGORITHM AND IMPLEMENTATION

2.1. *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 1**. The cone's surface encompasses all the possible origins for that ray. After these cones have been constructed, a random point from the surface 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 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,[4] then improves the histogram iteratively by moving likely origins for each cone to more likely locations using the criterion of an already higher density in the histogram. That is, in each iteration, for each cone, the algorithm chooses a new random point in its cone of possible origins for that ray. 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 is changed in one iteration.

2.2. *Description of code implementation*

Two input files are required to run this code. A configuration file controls various parameters for the code. Important parameters 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.

The initial cones are created using the process described in **Section 2.1** 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 than 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. The code writes the iteration number, the time elapsed, the numbers of cone origins updated, and the fraction of cones updated to the stdout. 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 3**, we use plots of likely origins of all cones.

2.3. *OpenMP algorithm*

The original code for this project was developed by Drs. Dennis Mackin and Jerimy Polf. 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 node with two multi-core CPUs. 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 2.1** 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. The 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 the two multi-core CPUs on one node, 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 2.2**, 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 2.2**. This is repeated until all iterations have been completed and the output files are generated as they were in **Section 2.2**.

2.4. *Original MPI algorithm*

The algorithm explained in **Section 2.3** 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 node anymore, but can use several nodes 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 process 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 using `MPI_Allreduce` every 1 iteration and all local histograms updated. 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.

2.5. *Modified MPI algorithm*

The only parallel cost of the MPI algorithm described in **Section 2.4** 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.

2.6. *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.

2.7. *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.

3. RESULTS

The following are the results were obtained from the different algorithms that are formulated in **Section 2**. Each algorithm uses the same deterministic configuration file utilizing 100,000 cones imaged by a histogram with $102 \times 102 \times 126$ bins.

3.1. *OpenMP algorithm*

The OpenMP algorithm used to obtain the following results was described in **Section 2.3** and written by Drs. Dennis Mackin and Jerimy Polf.

3.1.1. *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 2**. 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 3** 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 would also mean that fewer changes are being made to the density matrix as more iterations are performed.

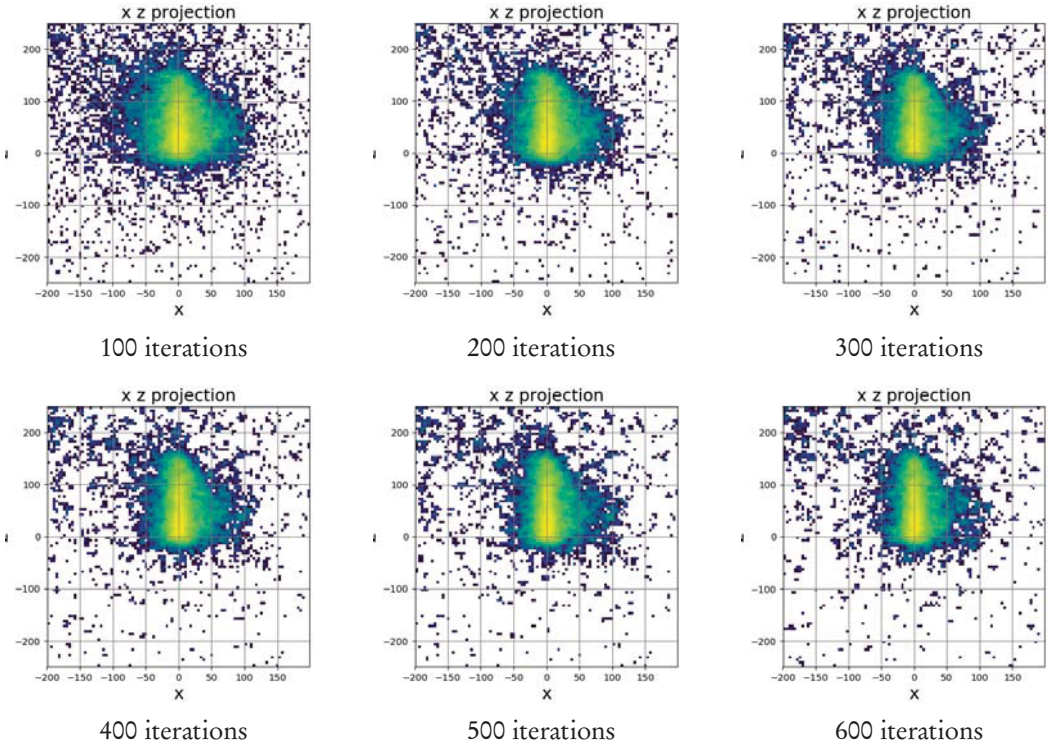


Figure 2. Reconstructed images computed by the OpenMP algorithm with 1 thread using Python post processing up to 600 iterations

```
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 3. Iteration log output from the OpenMP algorithm with 1 thread.

At the end of the log output in **Figure 3**, 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 2** 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.

3.1.2. *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.

Figure 4 shows the results of iterations of a 2 thread run. The reconstructed images have shapes similar to those seen in **Figure 2**. The shapes of the beam are nearly indistinguishable and therefore acceptable results.

Similarly, the convergence behavior of the ratio for 2 threads in **Figure 5** follows the same pattern as **Figure 3**. 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 3** and **5** 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.[6]

3.1.3. *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 functions regardless of parallelism.

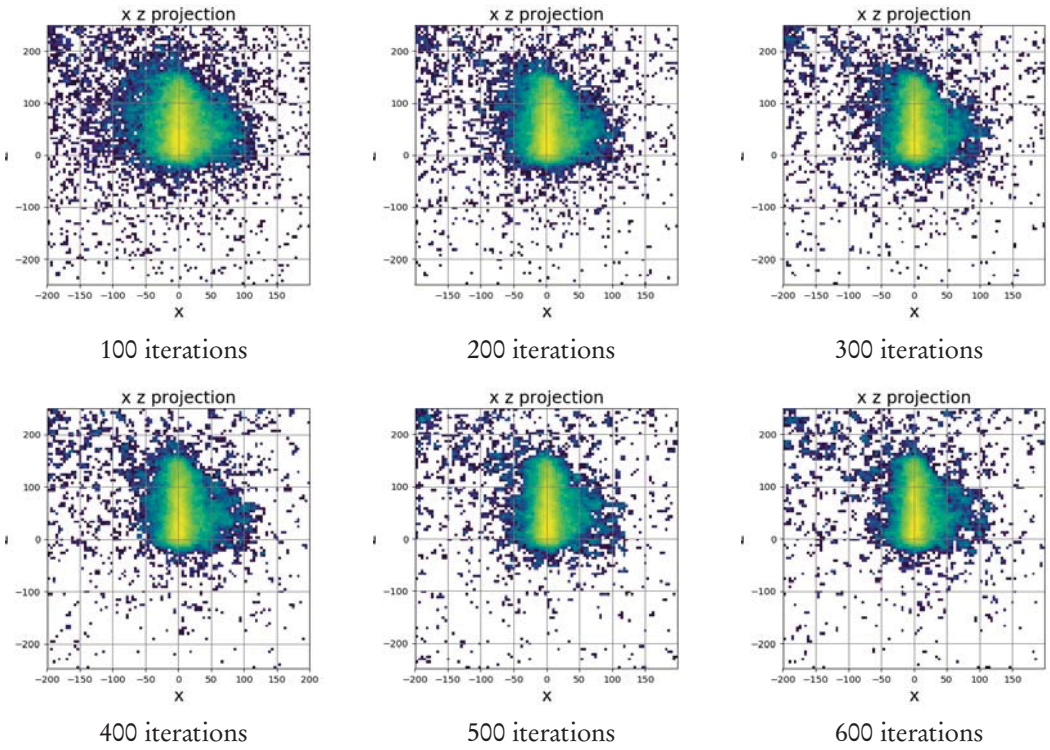


Figure 4. Reconstructed images computed by the OpenMP algorithm with 2 threads using Python post-processing up to 600 iterations.

Figure 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 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`

```
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 5. Iteration log output from the OpenMP algorithm using 2 threads up to 600 iterations.

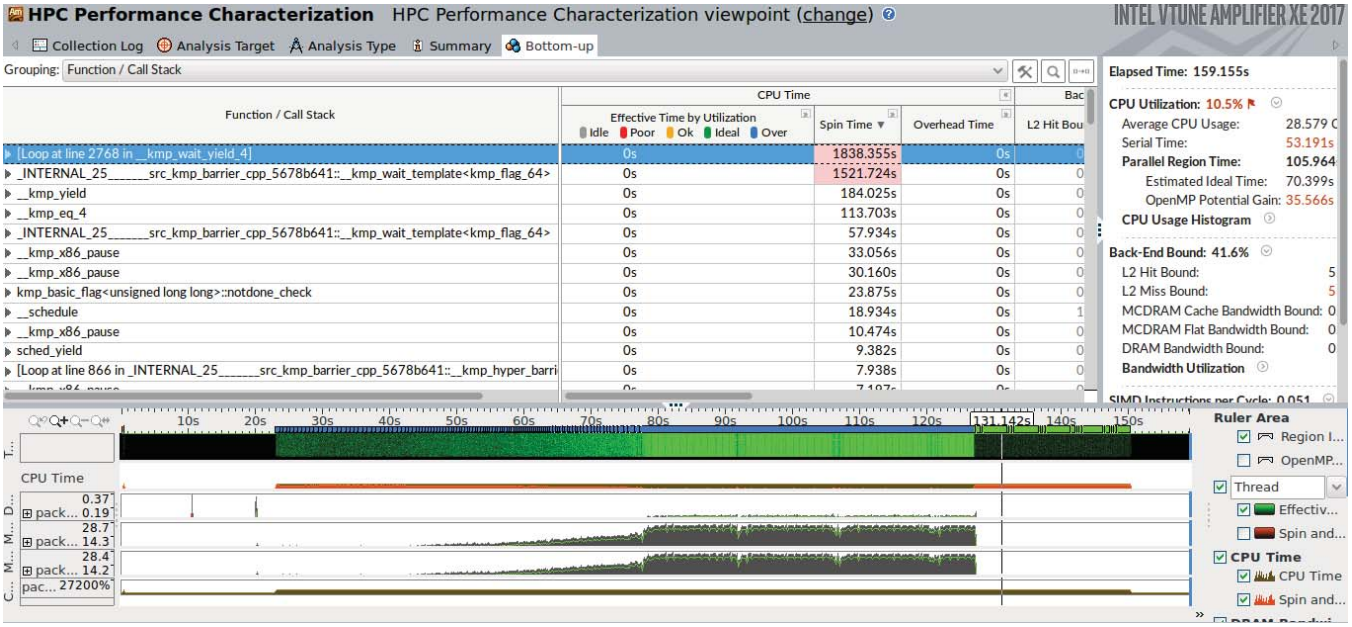


Figure 6. VTune Performance Characterization for the OpenMP algorithm with 68 threads per node.

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.

3.2. *Modified MPI algorithm*

As detailed in **Sections 2.4** and **2.5**, the MPI algorithm is different from 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 in the following. But the advantages of the MPI algorithms are two-fold: (i) Parallelism is not restricted to one node any more, but can be extended to any number of parallel nodes available, and (ii) speedup is not limited any more if progressively more nodes can be used in parallel. These two-fold advantages are the goal of both MPI algorithms.

3.2.1. *Modified MPI algorithm serial run*

A second, modified MPI algorithm is described in **Section 2.5** and 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 the communication command `MPI_Allreduce` every 1 iteration. The modified MPI algorithm decreases this cost by communicating only every 10 iterations. The major concern with this algorithm is possible further degradation of the convergence behavior. With the processes only having access to a histogram that was developed 10 iterations prior to their current iteration there is

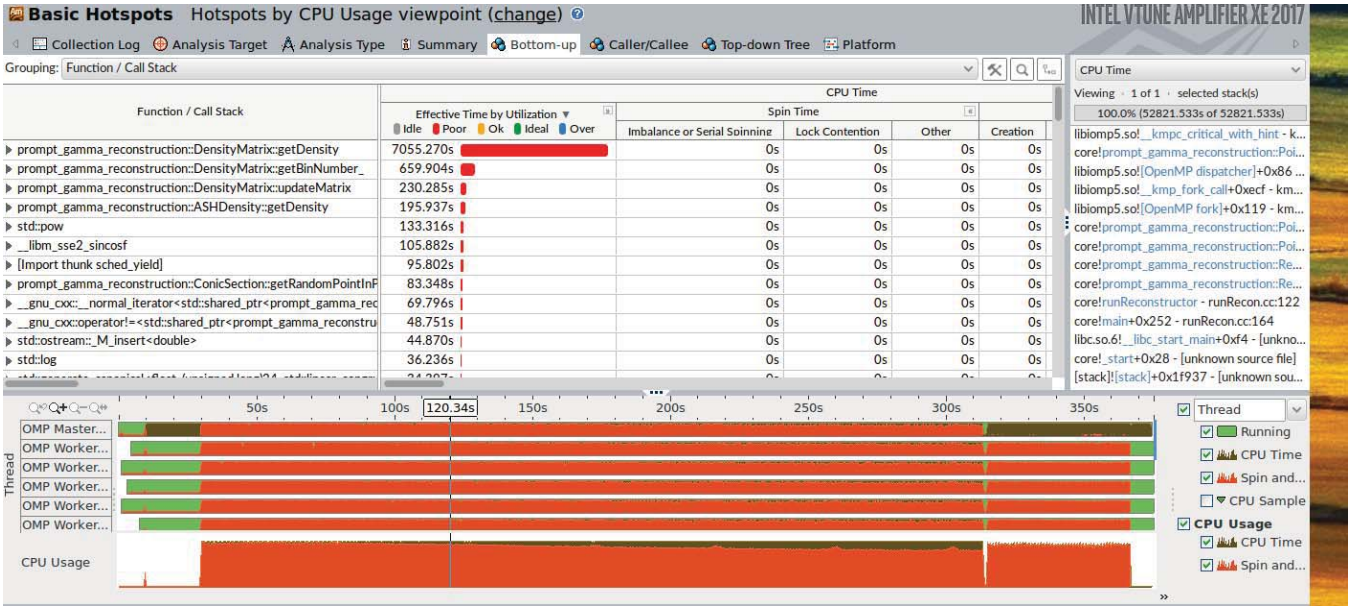


Figure 7. VTune Hotspots Analysis for the OpenMP algorithm with 68 threads per node.

a possibility that the lack of updates causes the images to degrade to unusable quality. For the modified MPI algorithm in serial, **Figure 8** 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 images 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[6] and shows only slightly slower convergence than the original OpenMP code in **Figure 2**.

Furthermore, the log output in **Figure 9** for the modified serial run shows that the ratio is a little higher than the OpenMP algorithm, but the general behavior of convergence remains similar.

We note that the images in **Figure 8** are produced by Matlab instead of Python post-processing. As discussed in **Section 2.4**, this change gains the advantage that one run of the code now outputs in steps of 100 iterations, thus all images in the figures from MPI algorithms come from the same run.

3.2.2. *Modified MPI algorithm multi-process runs*

Figure 10 shows some minor image degradation again in the first images, compared to the original OpenMP algorithm, but is apparently identical to **Figure 8** for the serial run. As in **Figure 8**, the cloud starts out similar to **Figure 2** 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 than the original MPI.[6] The images visually converge at around 900 iterations, which coincides with the log file output in **Figure 11**. This slower convergence behavior can be more clearly demonstrated comparing the modified MPI algorithm log output in **Figure 11** to the multi-threaded OpenMP **Figure 5**. We might recommend to use 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.

3.2.3. *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.

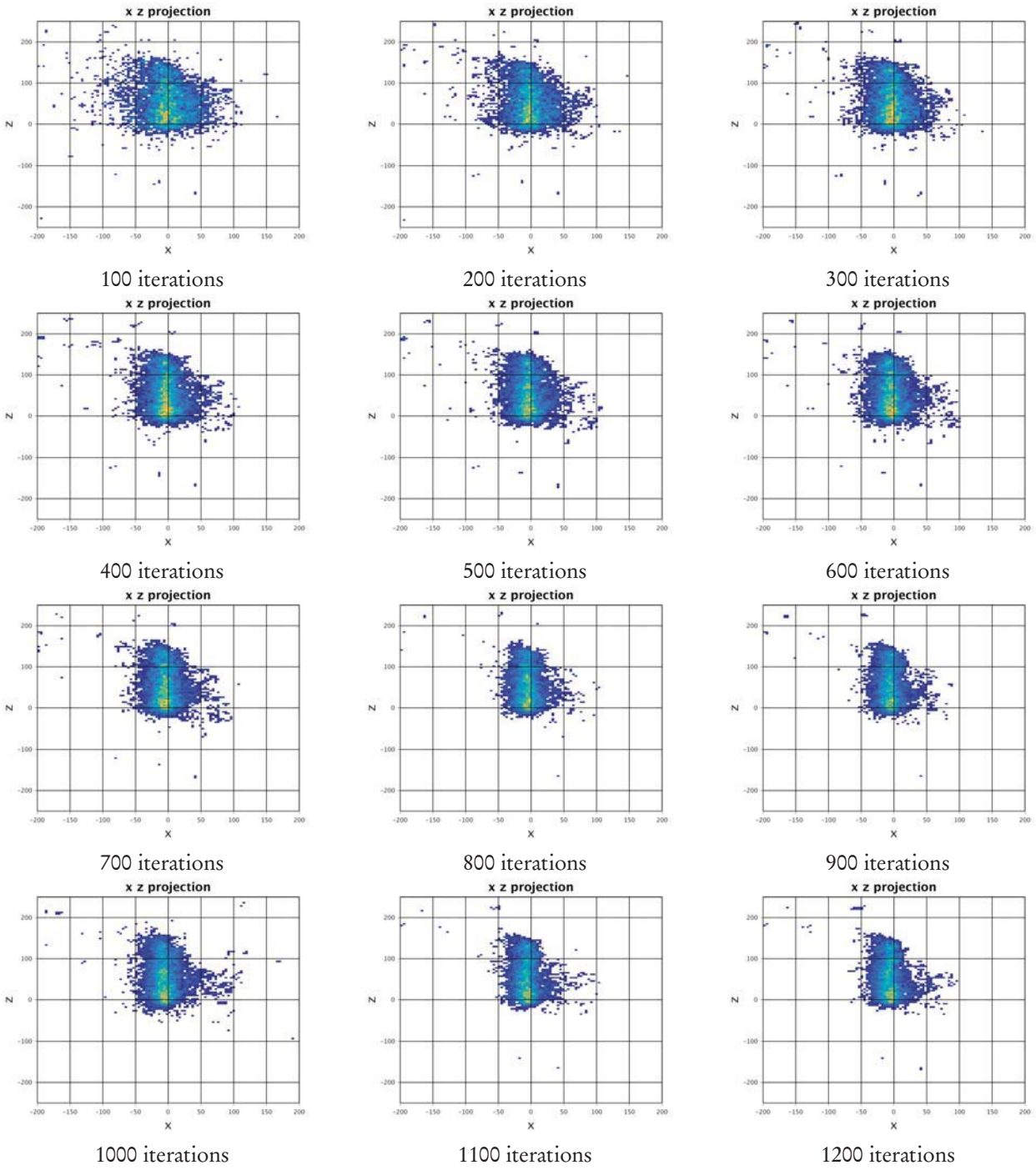


Figure 8. Reconstructed images computed by the modified MPI algorithm with 1 process up to 1200 iterations using Matlab post-processing.

Figure 12 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 13 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

```
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 9. Iteration log output from the modified MPI algorithm using 1 processes up to 1200 iterations.

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 altogether.

3.3. *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 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 shown in the first row of timings in **Table 1**. The code was tested when run on one node 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; runs for 32 and 64 cores are marked as N/A in the table.

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 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, the studies in each row of timings in **Table 1** the same hardware cores in the CPUs on one node; beyond 16 processes, MPI uses more nodes and thus more hardware, while OpenMP cannot be utilized beyond one node. Already the performance of our original MPI algorithm from **Section 2.4** 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 speed up 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

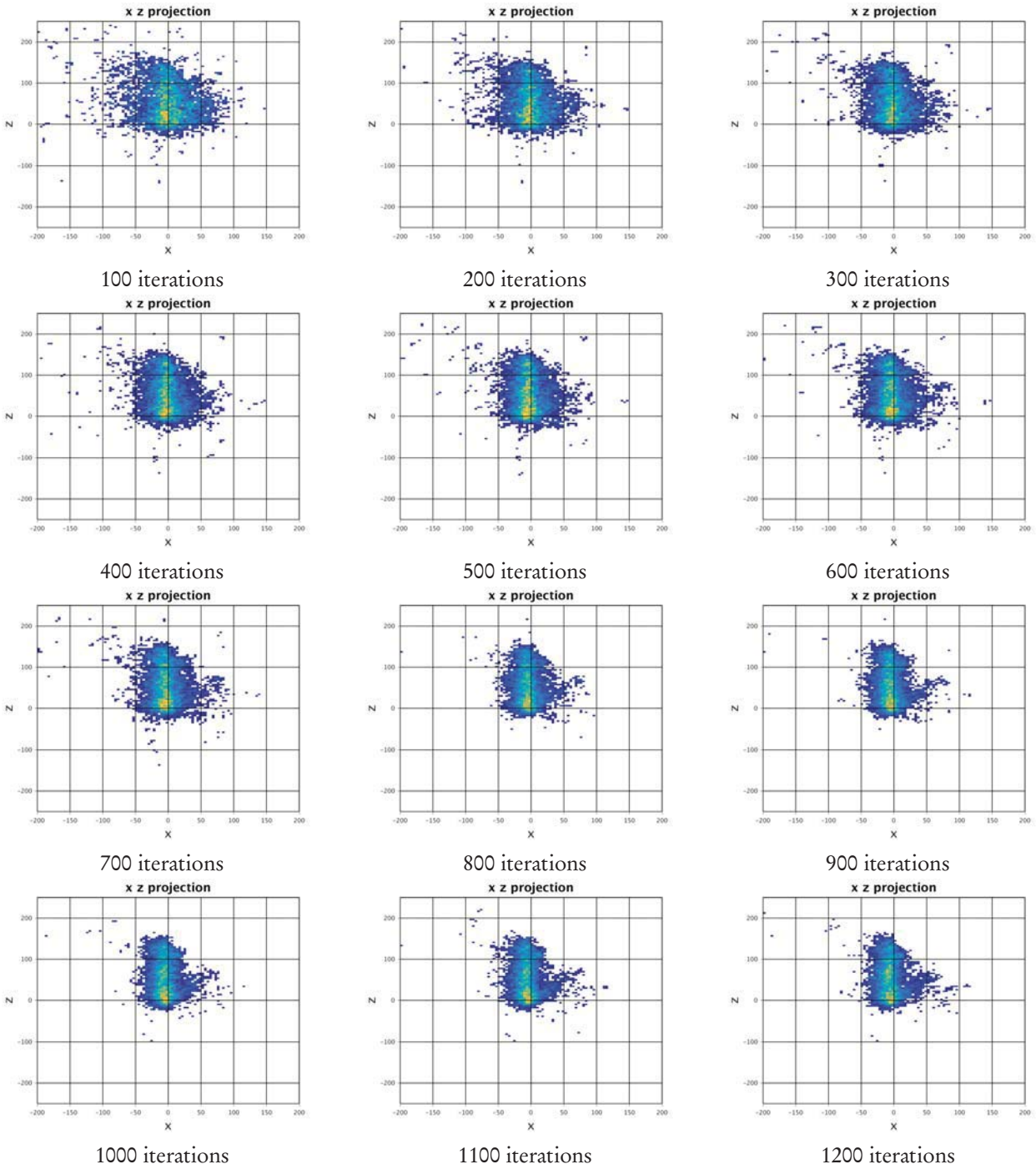


Figure 10. Reconstructed images computed by the modified MPI algorithm with 2 processes up to 1200 iterations using Matlab post-processing.

exhibits that parallel communications eventually overwhelm the increased efficiency of splitting up the computational work between processes.

In the modified MPI algorithm, the histogram would be updated every 10 iterations as described in **Section 2.5**. 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.

```
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 11. Iteration log output from the modified MPI algorithm using 2 processes up to 1200 iterations.

4. 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.[6] This was changed in the modified MPI algorithm to allow for

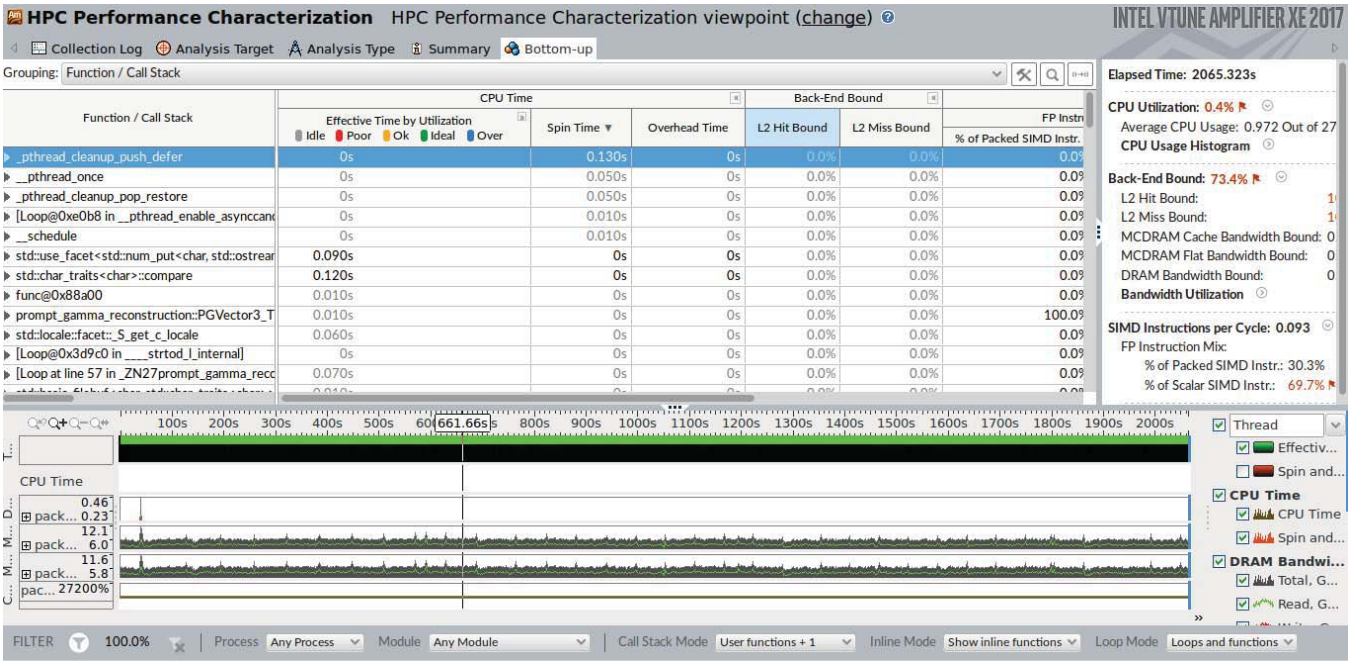


Figure 12. VTune Performance Characterization for the Modified MPI algorithm with 4 processes per node.

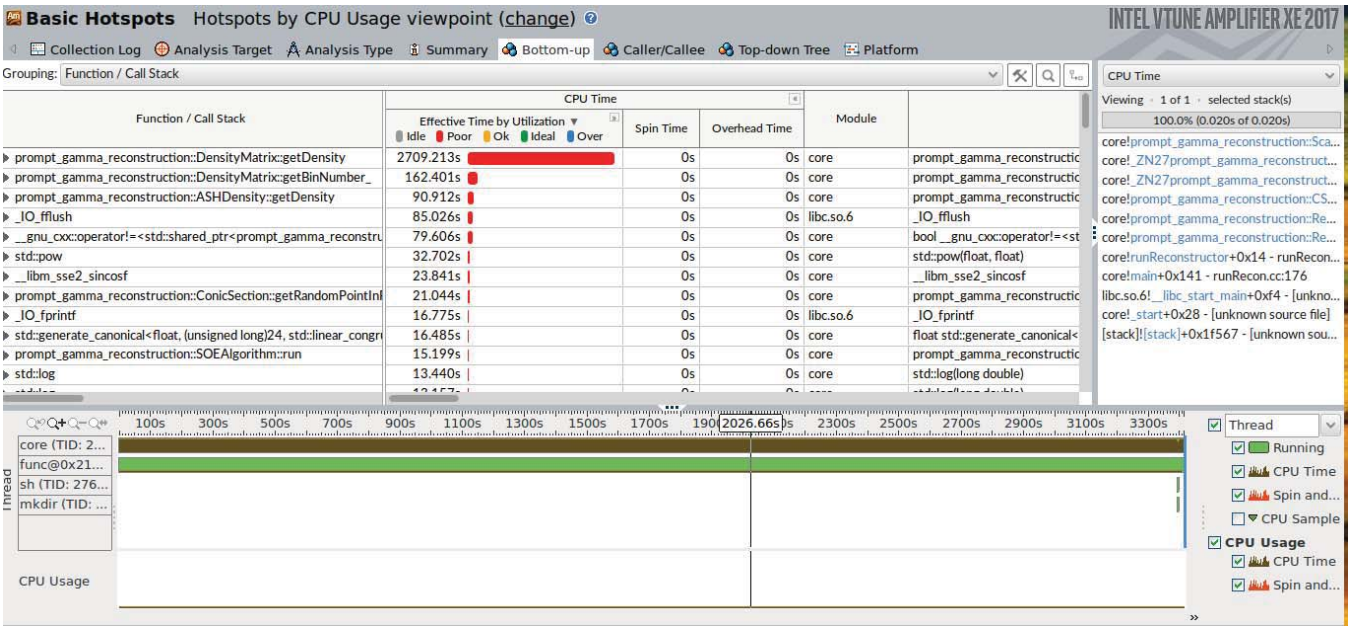


Figure 13. VTune Hotspots Analysis for the Modified MPI algorithm with 4 processes per node.

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. In particular, the formulation lays the groundwork to be used on the brand new many-core Intel Xeon Phi KNL processor with 64+ computational cores, since it is demonstrated that pure OpenMP parallelism is not optimal on that hardware.

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

Table 1. Observed wall clock time in seconds for reconstruction with 600 iterations.

ACKNOWLEDGEMENTS

These results were obtained in the REU Site: Interdisciplinary Program in High Performance Computing in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2017 (hpcreu.umbc.edu). 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. The research reported in this publication was supported by the National Institutes of Health National Cancer Institute under award number R01CA187416. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health. 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. Dennis Mackin, Steve Peterson, Sam Beddar, and Jerimy Polf. (2012) 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.
2. Jerimy C. Polf and Katia Parodi. (2015) Imaging particle beams for cancer treatment. *Physics Today*, 68(10):28–33.
3. 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. (2015) 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.
4. Andriy Andreyev, Arkadiusz Sitek, and Anna Celleri. (2011) Fast image reconstruction for Compton camera using stochastic origin ensemble approach. *Med. Phys.*, 38:429-435.
5. Intel Developer Zone. Intel VTune Amplifier. Documentation at the URL: *https://software.intel.com/en-us/intel-vtune-amplifier-xe-support/documentation* (accessed January 2018).
6. Johnlemuel Casilag, James Della-Giustina, Elizabeth Gregorio, Aniebet Jacob, Carlos Barajas, Matthias K. Gobbert, Dennis S. Mackin, and Jerimy Polf. (2017) Development of fast reconstruction techniques for prompt gamma imaging during proton radiotherapy. Technical Report HPCF–2017–16, UMBC High Performance Computing Facility, University of Maryland, Baltimore County.

ABOUT THE STUDENT AUTHORS

Johnlemuel Casilag and Aniebiet Jacobs will graduate in May 2020, both with a B.S. in computer science from University of Maryland, Baltimore. John plans on pursuing an advanced technical degree in artificial intelligence. Elizabeth Gregorio will graduate from Hamline University in May 2018 with a B.S. in Physics. After graduation she hopes to continue her education studying computational fluid dynamics. James Della-Giustina will graduate in December 2019 with a B.S. in computer science and plans to work with embedded systems.

PRESS SUMMARY

Proton beam radiation treatment has the potential to greatly reduce the amount of healthy tissue that receives radiation during treatment for cancerous tumors as opposed to x-ray radiation treatment. This paper focuses on improving performance for prompt gamma imaging software that would allow physicians to monitor the dose delivered in real time on the operating table, a tool that currently does not exist. With the implementation of the parallel computing library MPI, we increased the speed of the code as well as laid the foundation for further performance improvements to provide physicians with this much needed advantage.