Roadmap for Efficient Parallelization of Breast Anatomy Simulation

Joseph H. Chui, David D. Pokrajac*, Andrew D.A. Maidment, Predrag R. Bakic
University of Pennsylvania, Department of Radiology, 3400 Spruce St., Philadelphia PA 19104
*Applied Mathematics Research Center, Delaware State University, Dover, DE 19901

{Joseph.Chui | Andrew.Maidment | Predrag.Bakic}@uphs.upenn.edu
dpokrajac@desu.edu

ABSTRACT

A roadmap has been proposed to optimize the simulation of breast anatomy by parallel implementation, in order to reduce the time needed to generate software breast phantoms. The rapid generation of high resolution phantoms is needed to support virtual clinical trials of breast imaging systems. We have recently developed an octree-based recursive partitioning algorithm for breast anatomy simulation. The algorithm has good asymptotic complexity; however, its current MATLAB implementation cannot provide optimal execution times. The proposed roadmap for efficient parallelization includes the following steps: (i) migrate the current code to a C/C++ platform and optimize it for single-threaded implementation; (ii) modify the code to allow for multi-threaded CPU implementation; (iii) identify and migrate the code to a platform designed for multithreaded GPU implementation. In this paper, we describe our results in optimizing the C/C++ code for single-threaded and multi-threaded CPU implementations. As the first step of the proposed roadmap we have identified a bottleneck component in the MATLAB implementation using MATLAB's profiling tool, and created a single threaded CPU implementation of the algorithm using C/C++'s overloaded operators and standard template library. The C/C++ implementation has been compared to the MATLAB version in terms of accuracy and simulation time. A 520-fold reduction of the execution time was observed in a test of phantoms with 50-400 μm voxels. In addition, we have identified several places in the code which will be modified to allow for the next roadmap milestone of the multithreaded CPU implementation.

Keywords: Digital mammography, anthropomorphic breast phantom, validation.

1. INTRODUCTION

Anthropomorphic breast phantom simulation is important for pre-clinical testing and image system analysis. Recently, a new algorithm using an octree data structure and recursive partitioning has been proposed to simulate breast anatomy [1]. In this algorithm, ellipsoids of different locations, orientations, and relative sizes are modeled inside the simulated breast. Octrees are then constructed to represent the phantom spatially while the volumes are being voxelized.

Computing devices with multiple processing cores have become mainstream and more accessible. Many of these hardware configurations consist of various multiple-core Computer Processing Units (CPU) and massively parallel processors such as Graphics Processing Units (GPU). Several studies [2] [3] using the implementations of concurrent or even parallel algorithms targeted on these hardware platforms have demonstrated the performance benefits of these processing cores. To utilize multiple processing cores to full extent, programmers are often required to explicitly create multiple threads in the software to process the data concurrently. Careful insertions of synchronization constructs are also needed to allow programmers to guard against race conditions. The proposed octree-based algorithm has shown good asymptotic complexity; performance can potentially be improved by utilizing these multiple processing cores effectively.

To maximize programming flexibility, it is necessary to migrate our implementation of the octree-based breast anatomy simulation algorithm to a software platform that is designed for data parallelism. An appropriate software platform should allow the algorithm implementation to run on hardware equipped with massively parallel processors, such as GPUs. On the other hand, implementing algorithms on these massively parallel processors is often more difficult than

Medical Imaging 2012: Physics of Medical Imaging, edited by Norbert J. Pelc, Robert M. Nishikawa, Bruce R. Whiting,
Proc. of SPIE Vol. 8313, 83134T · © 2012 SPIE · CCC code: 1605-7422/12/$18 · doi: 10.1117/12.912125

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

implementing a single-threaded version; as a result, it is important to have a reference implementation that programmers can test their concurrent implementation against it. Instead of directly migrating the MATLAB implementation into a parallel implementation, we propose a migration roadmap with well-defined milestones. In this paper, the proposed migration roadmap is defined. The roadmap is used to develop specific milestones, and strategies for achieving these milestones are reviewed. The use of the roadmap by its application to the octree modeling problem is also illustrated.

2. METHODS

2.1. Migration Roadmap

We propose a roadmap for migrating the octree algorithm into a fully parallelizable platform. Figure 1 shows a flowchart of the roadmap with well-defined milestones. In the proposed roadmap, the algorithm implemented in MATLAB will be first migrated into single threaded C++. Multiple threads of execution would then be injected into the algorithm to allow the octree nodes to be processed concurrently. Finally, the software would be migrated onto a platform that allows fine-grained control of parallelization and synchronization.

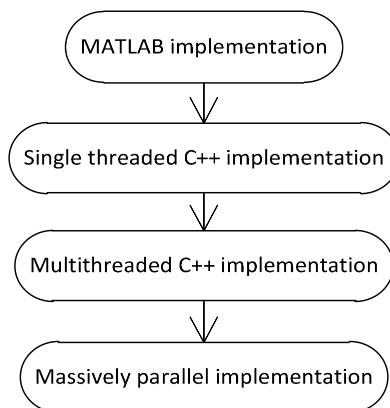


Figure 1. Migration roadmap of the octree-based algorithm implementations. With the initial implementation done in MATLAB, the roadmap defines several implementation milestones leading to a massively parallel implementation.

Regression tests are performed on each implementation before moving on to the next milestone. At each step, octrees of different resolutions constructed by the new implementation are compared against previously tested implementations to make sure the new implementation is correct. Software profiling is performed on each tested implementation to identify bottlenecks. The analysis of results is used to drive the design of the next milestone.

2.2. MATLAB Implementation to Single-threaded C++ Implementation

Figure 2 shows a flowchart of the sequential version of octree-based algorithm. At each level of the octree, an iterator processes each individual octree node to determine its tissue types.

The initial MATLAB implementation of the algorithm was optimized with respect to execution time and memory footprint. The code extensively uses global variables to store data from octree nodes, shape functions and the final voxelized phantom. To enhance speed, the matrices and vectors that depend on the parameters of shape functions but not of a particular octree node are pre-computed and stored as static global variables. The implementation does not use structures. The usage of structures, although a natural way to group variables similar in function, would impose significant performance and memory overhead. We experimented with single and double precision versions of the code: the double precision code resulted in better performance, presumably since double precision is the native way MATLAB performs computations.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

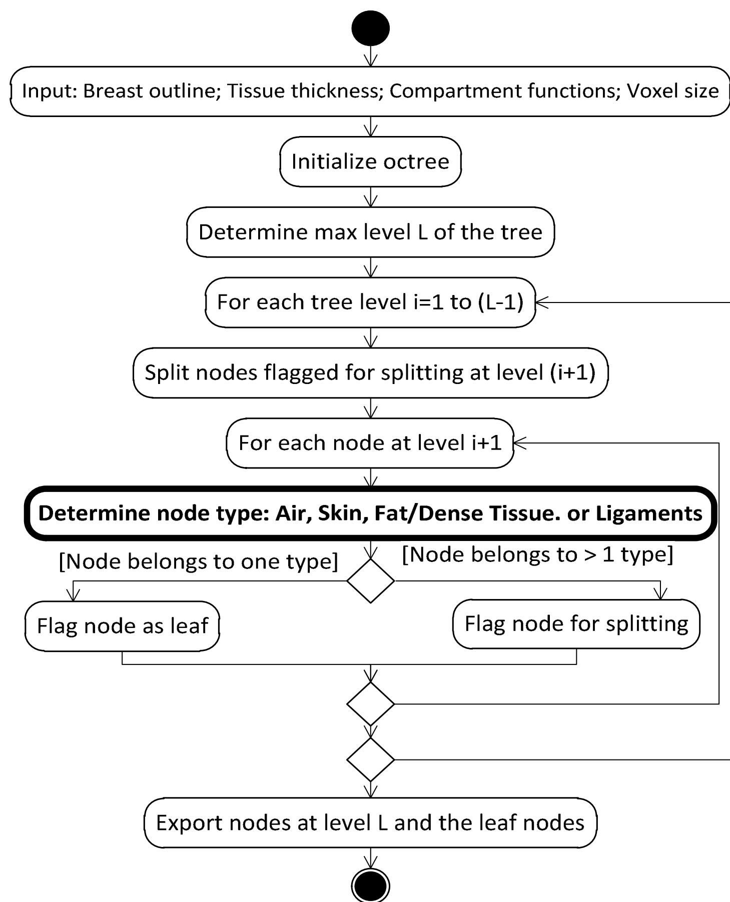


Figure 2. A flowchart of the octree algorithm.

The major functions in the implementation include:

- *recursive_partition_simulation_nostr* : main function, execute simulation and store the resulting phantom in voxel format.
- *init_oct_tree_empty_nostr* : initializes octree
- *recursive_data_split_nostr* : processes nodes from a particular level of the octree
- *split_criterion_distr_thickness_nostr/split_criterion_distr_thickness_nostr1*: determine whether a node needs be split
- *voxel_contains_boundary_thickness_nostr/ voxel_contains_boundary_thickness_nostr1:* determine whether a subvolume corresponding to the node contains only compartmental tissue, or only Cooper's ligament tissue, or both
- -*distr_max_box7:* computes the minimal and the maximal value of a shape function (that defines the position, initial orientation, and a relative size of each simulated tissue compartment.) in a given sub-volume

The profiler results, as shown in Table 1, indicate that the majority of self time (more than 882 million calls!) of the MATLAB implementation spent on a bottleneck routine *dist_max_box_7.m*, that evaluates the minimum and maximum of the shape functions (that define fat/dense compartments) in a voxel. Also, significant time in the MATLAB implementation is spent on initialization of global variables and assignment of cell-array elements.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

Table 1. Profiling result of MATLAB implementation for the top hotspots. It indicates *distr_max_box7* and *voxel_contains_boundary_thickness_nostr/voxel_contains_boundary_thickness_nostr* are the major bottlenecks of the implementation.

Function Name	Calls	Self Time (seconds)	% CPU Time
recursive_partition_simulation_nostr	1	8.792	0.01%
init_oct_tree_empty_nostr	8	11.877	0.02%
recursive_data_split_nostr	8	1762.056	2.62%
split_criterion_distr_thickness_nostr/ split_criterion_disrt_thickness_nostr1	92234064	10639.869	15.81%
voxel_contains_boundary_thickness_nostr/ voxel_contains_boundary_thickness_nostr1	76500278	24063.767	35.75%
distr_max_box7	882510190	29867.397	44.38%
Others		950.34	1.41%

2.2.1 Overloaded Operators

MATLAB's arrays and matrices and their operators are heavily used in our MATLAB implementation. To reduce human errors during migration, C++ classes such as *vector* and *matrix* were created to abstract these mathematical notations.

Figure 3 shows a snippet of MATLAB code migrated into C++ code.

```
function [int_min,int_max]=distr_max_box7(mu,R,logSqrtDetSigmalogprior,bb,A1inv,A2inv,A3inv,xlow,ylow,xhigh,yhigh,zlow,zhigh,H,U,deltax,deltaxsquared,x)

    xcmu=xc-mu;
    fc=-0.5*sum((xcmu*R).^2,2)+
        logSqrtDetSigmalogprior;
    boundary_values=fc+deltax*U*(xcmu)'+
        deltaxsquared*H;
    int_min=min(boundary_values);

    boundary_max=max(boundary_values);
    if xlow<=mu(1) && xhigh>=mu(1) &&
    ylow<=mu(2) && yhigh>=mu(2) &&
    zlow<=mu(3) && zhigh>=mu(3)

      int_max=logSqrtDetSigmalogprior;
    else
        [...]
    end
return
```

→

```
void distr_max_box7(rowVector mu, matrix
R,float logSqrtDetSigmalogprior,
columnVector bb, matrix A1inv, matrix A2inv,
matrix A3inv, float xlow, float ylow, float
xhigh, float yhigh, float zlow, float zhigh,
columnVector8 H, matrix8x3 U, float deltax,
float deltaxsquared, rowVector xc, float
&int_min, float &int_max)
{
    rowVector xcmu=xc-mu;
    float fc=0.5f*rowVector::sum((xcmu *
    R)^(xcmu*R))+logSqrtDetSigmalogprior;
    columnVector8 boundary_values= fc +
           deltax *(U*xcmu.transpose()) +
           deltaxsquared*H;
    int_min=boundary_values.min();
    float boundary_max=boundary_values.max();
    if (xlow<=mu[0] && xhigh>=mu[0] &&
        ylow<=mu[1] && yhigh>=mu[1] &&
        zlow<=mu[2] && zhigh>=mu[2])
     {
       int_max=logSqrtDetSigmalogprior;
     }else
          [...]
```

Figure 3. A snippet of MATLAB code migrated into C/C++ code. The codes are aligned to illustrate the line to line conversion of the each line of code from MATLAB into C++.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

Because C++ is a typed language, the data types such as *float* and *rowVector* have to be declared explicitly. MATLAB operators were mapped to the same overloaded operators if they are allowed in C++, otherwise, they were mapped to their respective methods, such as *transpose()*. For the indexing, while it was possible to simulate one-based indexing in C++, manual one-based to zero-based conversion is performed because it provides better consistency with the conventions in future milestones.

2.2.2 STL Containers

In the MATLAB implementation, *arrays* are frequently used to store arrays of data, such as the octree nodes at each level. Because their sizes change dynamically during the computation, the containers used in the C++ implementation are also required to resize during run time. Instead of implementing our own container class, Standard Template Library's (STL) *vector*<> class [4] is used in the C++ implementations to replace MATLAB's *arrays*. STL is part of C++ Standard Library which consists of a number of commonly used classes and functions. STL defines a collection of container templates which abstract and encapsulate common data structures such as arrays and linked lists. STL's *vector*<> is preferred to other STL container classes because of similar time complexity as MATLAB's *arrays*, better memory locality, and smaller memory footprint.

MATLAB's *cellarrays* are used for the octree-based algorithm to store arrays of arrays of homogenous data. The size of the arrays of homogenous data are dynamic during run time and also different (ie. jagged) even if they belong to the same array. In the C++ implementations, they are mapped to STL's *vector*<*vector*<>> class for the same reasons that the STL *vector*<> class is used.

2.3. Single-threaded Implementation to Multithreaded C++ Implementation

In order to utilize multiple processing cores effectively, it is necessary to identify in the algorithm where data can be processed concurrently. As shown in Figure 2, an inner loop iterator is used to walk through each node and determine its tissue type. Since the determination of each node's tissue is independent of other nodes at the same tree level, it is clear that this loop is a good candidate for data parallelism. Figure 4 shows the activity diagram of the algorithm with multiple worker threads processing the nodes concurrently:

The multithreaded C++ implementation creates a pre-defined number of worker threads in advance. Instead of creating a shared thread-safe queue for each thread to acquire nodes for it to process, the proposed algorithm assigns a pre-defined collection of nodes to each worker thread before invoking the worker threads. Because *vector*<> is thread-safe for reading; multiple threads can read the vector elements safely without synchronization until they finish processing their assigned collection of nodes.

Two node assignment schemes are tested in this study: 1) interleaved and 2) contiguous. In the interleaved scheme, each node in the array is alternately assigned to each thread in order. The n-th octree node is assigned to the (n mod T)-th thread, where T is number of worker threads. In contiguous scheme, each thread is assigned one contiguous block of nodes of equal size. In our experiment, the n-th octree node is assigned to (n/T)-th thread where T is the number of worker threads.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

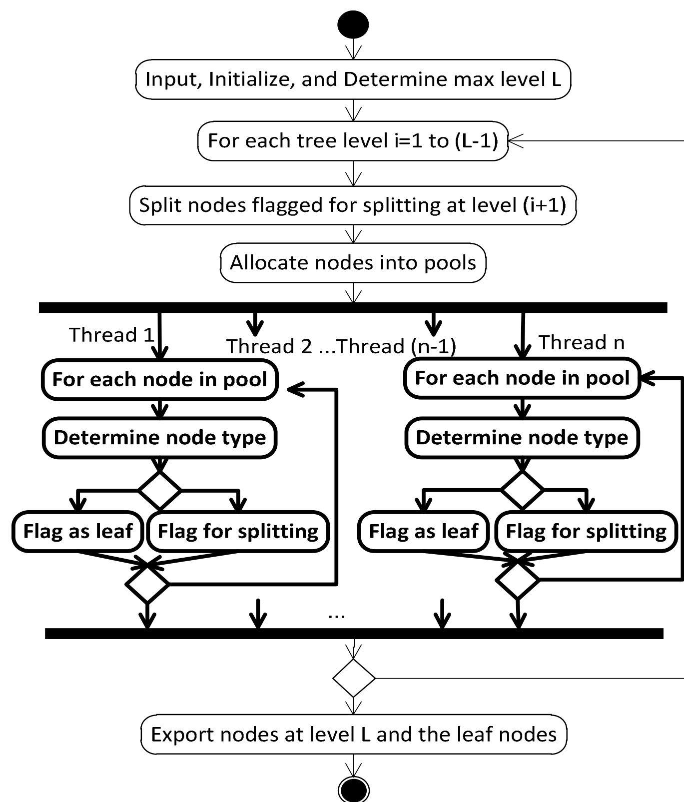


Figure 4. Flowchart of a concurrency version of the octree algorithm. Multiple concurrent flows are forked to process pre-defined sets of the octree nodes. The concurrent flows are joined after every thread finishes processing their assigned nodes.

2.4. Quantitative comparison of implementations.

The performances of the different implementations are compared by their duration times to construct an octree and voxelize a volume of tissue types. A total of 160 phantoms are simulated with various:

- Voxel sizes: 50, 100, 200, and 400 microns
- Software platforms: MATLAB, C++
- Number of threads: 1 - 8
- Octree node assignment schemes (Interleaved vs. Contiguous)

All phantoms are simulated as 450ml breasts with random distributions of 333 adipose compartments. The duration times are measured on a Windows 7 64-bit PC, with Intel Core i7 - 2600K (Quad core and Hyper-Threading enabled) and 16.0 GB of RAM. MATLAB R2011b and Microsoft Visual Studio 2010 are used to execute or compile different implementations.

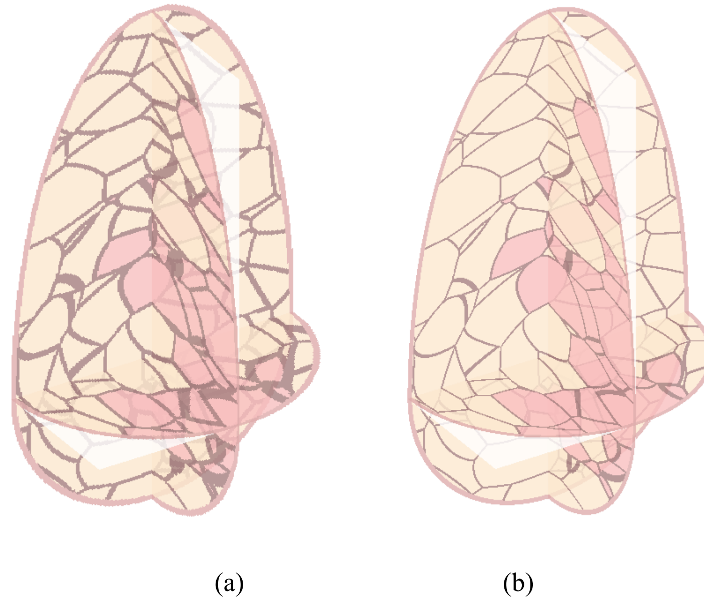
The implementation at each milestone is tested against the implementation of the previous milestone. The octrees generated with the same inputs are compared to make sure the new implementations generate the same octrees as the previously tested implementations.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

3. RESULTS

3.1. Synthetic images

Breast phantoms of different resolutions are created by using different implementations of the octrees-based algorithm. Figure 5 shows the orthogonal sections of a phantom with 400 µm and 50 µm voxel resolutions. With the same inputs, same octrees are constructed by the different implementations.



(a) (b)

Figure 5. Orthogonal sections of a simulated breast phantom of (a) 400µm and (b) 50µm resolutions

3.2. Performance Results of MATLAB, single threaded C++ and multithreaded C++ implementations.

The performance of the various implementations was assessed by comparing the duration times for generating phantoms of different voxel sizes. The duration time of each configuration is measured by the averaged duration time of 5 independent samples; each sample is generated from different set of ellipsoids modeled randomly inside the simulated breast. The empirical time complexity of the each implementation was measured by curve fitting the duration times of different voxel sizes. Figure 6 is a graph showing the duration times of different implementations at different voxel resolutions.

All implementations show a consistent quadratic time complexity against the inverse of voxel size. On average, the single-threaded C++ shows an average of 240 fold of duration time improvement over MATLAB implementation, while the multi-threaded C++ implementation shows an average of 520 fold of improvement.

3.3 Number of threads

To assess the influence of worker thread counts for different node assignment schemes, the average duration times of 5 samples was collected on simulations using up to 8 worker threads. Figure 7 shows the duration times of the multithreaded C++ implementation for different thread counts and assignment schemes.

Performance is improved by as much as 218% when the thread count is increased from 1 to 8 when testing it on a PC equipped with single quad core CPU. The duration times start to converge once the worker thread count is more than

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

four, because the number of software threads exceeds the number of processing cores on the CPU. On each thread count we tested, the interleaved scheme out-performed the contiguous scheme as much as 14%. In all cases, the interleaved scheme is at least as fast as contiguous scheme.

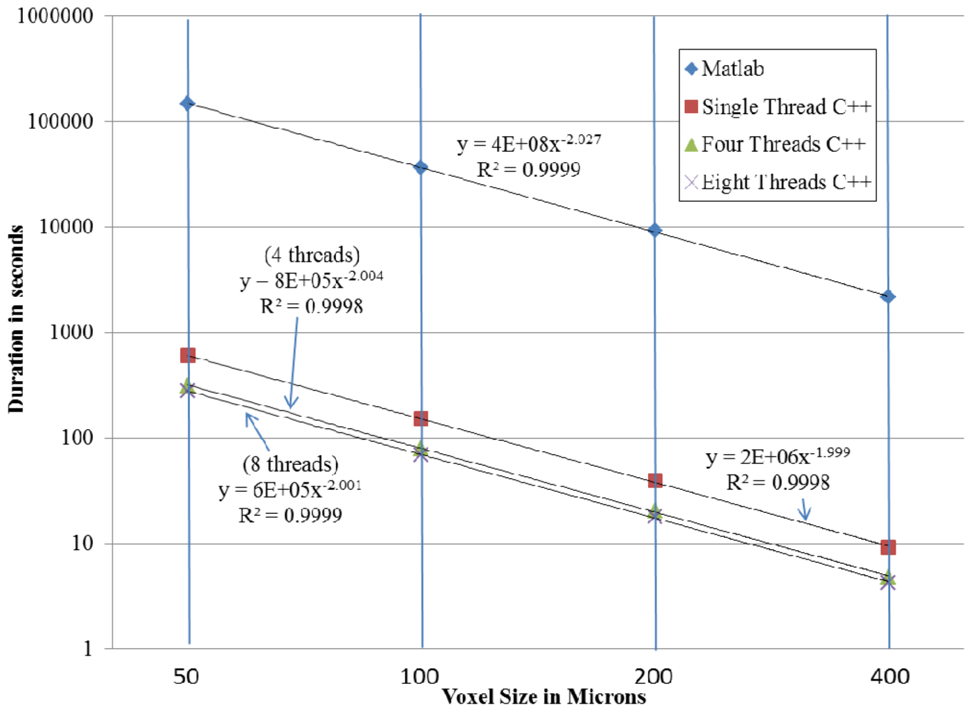


Figure 6. Average duration times of different implementations of the octree-based algorithm for various voxel sizes (50, 100, 200 and 400 microns). The times of each implementation are fitted with a line with the slope as the empirical time complexity against the voxel size.

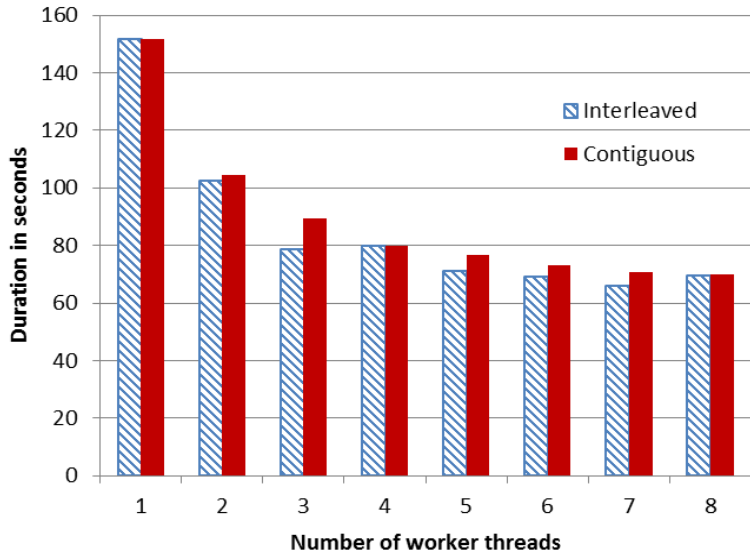


Figure 7. Average duration times of the multi-threaded C++ implementation with different numbers of worker threads forked to process the nodes concurrently. Two octree node assignment schemes: Interleaved (Blue) vs. Contiguous (Red) are compared.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

3.3. Profiling C++ implementations

After the C++ implementation had been tested, software profiling was performed using Intel VTune™ Amplifier XE 2011. Table 2 shows the profiling result of simulation of 100μm phantom using multi-threaded C++ implementation (8 threads).

Table 2. Profiler result of C++ implementation executed with 8 threads. The functions marked with asterisk are ones executed on worker threads. The self-times are the times normalized as if they are executed on a single thread. These are the same functions with the profiler results shown in Table 1.

Function Name	Self Time(seconds)	% Time
recursive_partition_simulation_nostr	0.661	0.74%
init_oct_tree_empty_nostr	3.164	3.52%
recursive_data_split_nostr	30.213	33.65%
*split_criterion_distr_thickness_nostr/ split_criterion_disrt_thickness_nostr1**	1.969229207	2.19%
*voxel_contains_boundary_thickness_nostr/ voxel_contains_boundary_thickness_nostr1**	29.23030282	32.55%
*distr_max_box7**	21.66346798	24.13%
Others	2.889	3.22%

On the multi-threaded C++ implementation, the functions executed on worker threads account for 58.87 percent of total duration time, while they account for 95.95 percent of total duration time on MATLAB implementation.

4. DISCUSSION

We have successfully completed the milestones of single-threaded and multi-threaded C++ implementation. Not only do these new implementations construct the same phantoms as the MATLAB implementation, the time performance has been improved significantly. On the single threaded C++ implementation, we observe a 240 fold improvement from the MATLAB implementation. We believe this is mainly the result of better optimization in C++ compiler and the fact that bound checking is disabled in STL containers in the released build. On the multithreaded C++ implementation, a 520 fold improvement from MATLAB implementation is observed. On a machine with a single quad-core CPU, the performance improved with the number of threads used to process the octree nodes concurrently. The improvement is not proportional to the number of threads as the bottleneck shifted to the steps where instructions were executed on the main thread.

The function, *recursive_data_split_nostr*, which runs on main thread, consumes about 33.65% of time in the multithreaded implementation, compared to 2.6% in MATLAB implementation. It indicates that the computational bottleneck is no longer dominated by numerical intensive computations, but rather it is shared by other operations such as node splitting and memory allocation and initialization. Performance improvements on these currently single threaded steps using parallelization are being investigated.

A comparison between MATLAB (double precision) and C/C++ implementations (single precision) indicate that the obtained results are similar. Note that in the C/C++ implementation the matrices utilized to compute the minimal and maximal values of the shape functions are computed as double and then stored as single precision static variables. This way we avoid any numerical instability i.e., due to performing matrix inversions. The remaining numerical operations performed on single precision variables involve simple matrix multiplications and are not sensitive to numerical error amplification.

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx

The observed acceleration is of importance for phantom applications requiring a large number of phantoms, e.g., model observer studies or statistical classifiers [5]. In addition, rapid phantom generation may reduce the need for storing and exchange of pre-fabricated phantoms; if the simulation is sufficiently fast, only the phantom parameters need to be stored, as the specific phantoms could be generated in near real-time.

5. CONCLUSION

We have created a roadmap of migrating a MATLAB implementation of the breast phantom simulation algorithm into a software platform that allows fine grained control of synchronization and parallelization. Based on the results in the multiple threaded C++implementation, real-time simulation of 3D breast anatomy using mainstream computer hardware can be realized by parallelizing our octree generalization method. The resultant multi-threaded C++ implementation is now being migrated to a platform that supports massively parallel processers such as GPUs.

ACKNOWLEDGEMENT

This work was supported in part by the US Department of Defense Breast Cancer Research Program (HBCU Partnership Training Award #BC083639), the US National Institutes of Health (R01 grant #CA154444), the US National Science Foundation (CREOSA grant #HRD-0630388), and the US Department of Defense/Department of Army (45395-MA-ISP, #54412-CI-ISP).

REFERENCES

[1] Pokrajac, D.D., Maidment, A.D.A., Bakic, P.R., "A Method for Fast Generation of High Resolution Software Breast Phantoms," *Medical Physics,* 38 (2011).
[2] Badal, A, Kyprianou, I, Banh, D.P., Badano, A, Sempau, J., "penMesh—Monte Carlo Radiation Transport Simulation in a Triangle Mesh Geometry," *IEEE Transactions on Medical Imaging* vol. 28 no. 12 (2009)
[3] Badal, A, Badano, A., "Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit," *Medical Physics* 36 (2009)
[4] Stepanov, A., Lee, M., The Standard Template Library. *HP Laboratories Technical Report 95-11(R.1)* (1995).
[5] Young, S., Park, S., Anderson, S.K., Badano, A., Myers, K.J., Bakic P., "Estimating breast tomosynthesis performance in detection tasks with variable-background phantoms," *Proc. SPIE 7258* (2009).

Downloaded From: http://spiedigitallibrary.org/ on 07/15/2016 Terms of Use: http://spiedigitallibrary.org/ss/TermsOfUse.aspx