Parallel Ray Tracing using MPI: A Dynamic Load-balancing Approach

S. M. Ashraful Kadir[1] and Tazrian Khan[2]

[1] Scientific Computing, Royal Institute of Technology (KTH), Stockholm, Sweden
smakadir@csc.kth.se,
WWW home page: http://www.csc.kth.se/~smakadir/
[2] ICT Entrepreneurship, Royal Institute of Technology (KTH), Stockholm, Sweden
tazrian@kth.se

Abstract. In this paper we present an MPI implementation of dynamic load-balancing using master-worker approach for the parallel ray tracing in distributed memory systems. The dynamic load balancing technique will be advantageous for an high performance computing infrastructure where different nodes of the distributed memory system are of different processing capacities. Experimentally we show that the dynamic scheduling technique achieves better result over the previously implemented static load-balancing techniques for the presented example scene. The presented work is an extension of our previous work on parallel ray tracing in both distributed memory and shared memory systems using MPI and OpenMP respectively.

Keywords: Load-balancing, dynamic load-balancing, ray tracing, MPI

1 Summary of Previous Work

Previously, in the original paper titled "Parallel Ray Tracing using MPI and OpenMP", c.f. [1], we presented an object oriented parallel ray tracer using C++ which was designed for compile time binding to run in either sequential or parallel mode where the parallel run mode is also decided in compile time based on parameters to run in either distributed memory (using MPI) or shared memory (OpenMP) system.

For the experimental purposes we implemented random scene generator (based on boundary parameters, etc.) so that we can generate scenes with large number of objects and lights. Figure 1 shows the examples of images created with randomly generated scenes (these two scenes are often referred as "scene 1" and "scene 2"). We implemented feature as scene configurations "export to" and "import from" XML format files. We have also implemented viewport pixel-wise load (numbers of floating point operations required) measurement feature. We implemented two static load-balancing techniques. Details of the implementation is available in [1].

We presented the theoretical performance model of time and memory complexities for both MPI and OpenMP cases. The graphical representation of

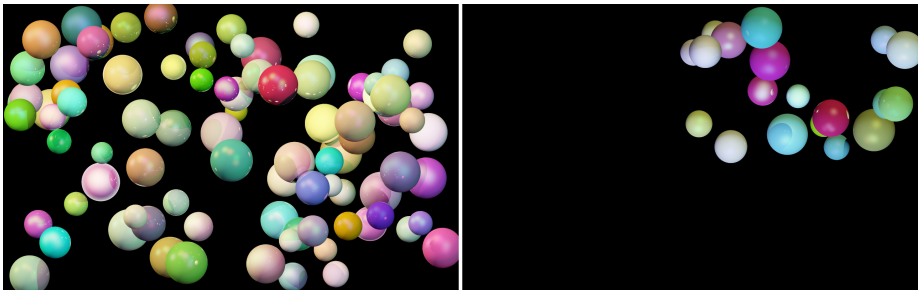

Fig. 1. Images created by scene configurations 1 (in left, scene contains 100 sphere objects) and 2 (in right, scene contains 20 sphere objects). Both the scenes contain five light sources for a viewport of 1280×800 pixels, and images created with 256 super-sampling coefficient. (Both the images are available in [1] in larger size.)

viewport-pixel-wise load distribution and processor (or thread) wise total load distribution were presented from the experimental results.

We implemented two static load balancing techniques, LB0 and LB1. The scene work load is evenly distributed among the processes (or threads) in LB0 technique, where adjacent rows are given to a processor (or a thread). In LB1 technique, process (or thread) p of total P processes (or threads) receives work load for rows $r_p, r_{p+P}, r_{p+2P}, \cdots$, i.e. adjacent rows are given to different processes (or threads) so that work load localization is handled in a better way than LB0 (see performance graph of LB0 and LB1 in 4). LB0 and LB1 are respectively referred as "no load balancing" and "static load balancing" in the previous paper (see algorithm description and figure 3 in [1]). Experimentally we have showed that both the load-balancing techniques were efficient to achieve near optimal efficiency for both the distributed memory and shared memory systems and achieved near linear speedup and near optimal efficiency for both the example scenes 1 and 2.

The implemented code was efficient in Intel C++ compiler version 10.1 as the optimization key switch didn't change the performance. However, in GNU C++ compiler version 4.1.2 the code was approximately 12.5 times faster with both level 2 and 3 optimization switching than without any optimization switching.

Although our implemented recursive doubling technique based on peer to peer communications technique performed better than the MPI function call "MPI_Gather" from the time requirements point of view, considering very small communication time requirements in comparison to the computation time and considering better memory efficiency we advocated for the use of "MPI_Gather" in the case of distributed memory systems.

We also experimented with different built-in load-balancing schemes in "for loop" parellelization using OpenMP. However, the experimental result did not seem to be impressive as the unit work load that can be dispatched is on each row basis. We advocated for "quality variant image rendering" to achieve real-time

Fig. 2. Part of the image (also zoomed in) created by scene configuration 3 (the scene contains 10,240 sphere objects each with radius 2.0, five point light sources for a viewport of 1280×800 pixels, image created with 256 super-sampling coefficient)

performance where quality can be varied with the available computing processors. Experimentally we showed that changing the super-sampling coefficient we achieved to keep the processing time requirements banded with the changes of the number of processors (MPI) or threads (OpenMP).

2 Motivation for Dynamic Load-balancing

In the original paper, dynamic load-balancing technique was suggested as a future work opportunity to improve the performance. Although we experienced near optimal efficiency from the experimental results using static load-balancing approaches, dynamic load-balancing techniques were expected to perform better than the static load-balancing approaches in the heterogeneous hardware infrastructure where the computational nodes have different processing capacities. Moreover, as we will see in this report that for scenes with certain patterns, such as stripes etc., we achieve more consistent performance using dynamic load balancing.

3 Implementation of Dynamic Load-balancing

We have followed the master-worker approach to implement dynamic load balancing where the master process is responsible for work-pool management and

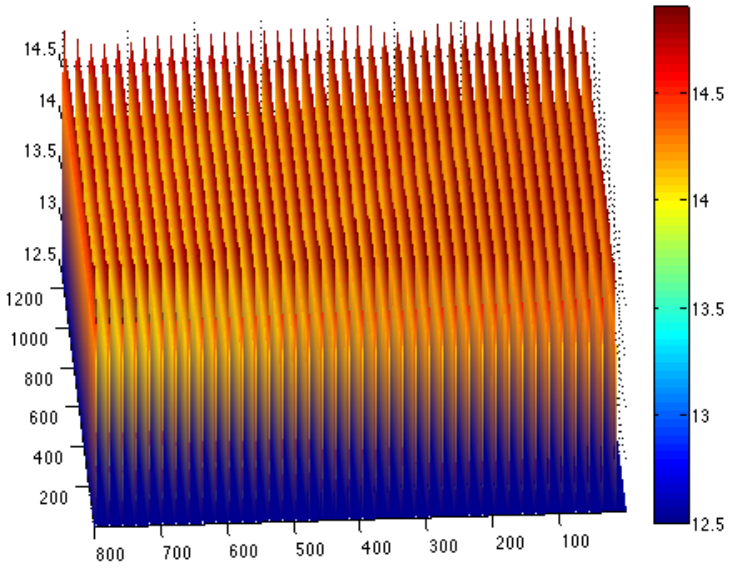

Fig. 3. Pixel wise load distribution for "scene 3" (load in logarithmic scale, image is rendered using super-sampling coefficient, c = 1), see figure 2 for the related image

work distribution, result collection through communication with the working processors, etc. Finally the complete image becomes available to the master node. At least two processors are required for applications of the implemented dynamic load-balancing technique.

The implemented dynamic load-balancing technique is a modified version of the algorithm proposed by Freisleben et al. in [2]. Initially the work load is segmented into two equal parts (+/- some small amount of workload to slightly vary the work load among the processors so that the workers don't finish their assigned works all together) where the image is stored as one-dimensional array as image width order. The first half then equally distributed among the processes. In next turns, the remaining image size is again segmented into equal two parts and evenly distributed among the worker processes "on demand" basis until the remaining work load reaches into a minimal threshold level for each worker processes. The work-demand is only raised once one processor completes the assigned task and returns the result to the master node.

The scene configurations and viewport related information is available to all nodes. For data distribution, the master process sends the start index and offset value of the segments to each worker processes that need to be rendered and remains in a listening mode from the worker processes until get a receive-data signal. The worker process then sends rendered data to master and waits for

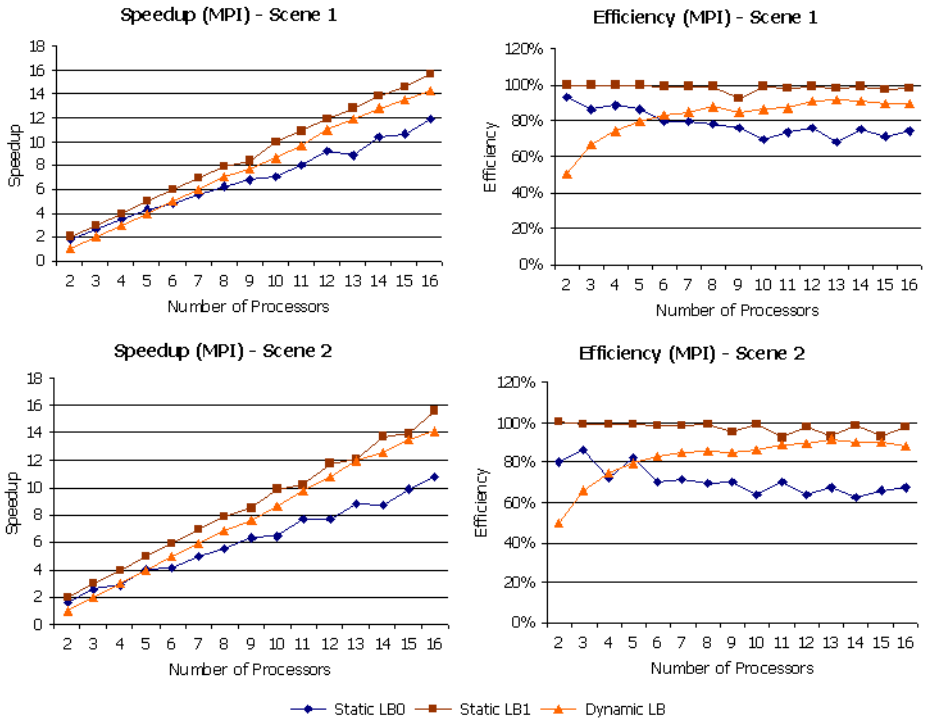

Fig. 4. Speedup and Efficiency for "scene 1" and "scene 2"

next segment information to receive for rendering until receive a no-work-left signal of (-1, -1) for starting index and offset value. The master process keeps a counter to measure how much work left and exits its waiting loop when the counter reaches to zero and then saves the output image file.

4 Result and Discussions

Figure 3 shows the pixel-wise load distribution for the scene 3 (see figure 2 for the related image). We see that the maximum load is distributed in parallel lines according to the scene. The scene contains total 10,240 spheres in parallel rows in the plane $z = 0$ parallel to the viewport. We note that the ray tracing is done for perpendicular projection. This is a newly created scene to show the effectiveness of the newly implemented dynamic load-balancing technique over the static load-balancing techniques. The other two scenes used in this experiments are taken from [1] and the related images are available in figure 1.

The experimental performance graph is available in figure 4. Although the dynamic load-balancing technique is fairly efficient to achieve better performance than LB0 case for scene 1 and scene 2, the performance is not as good as the

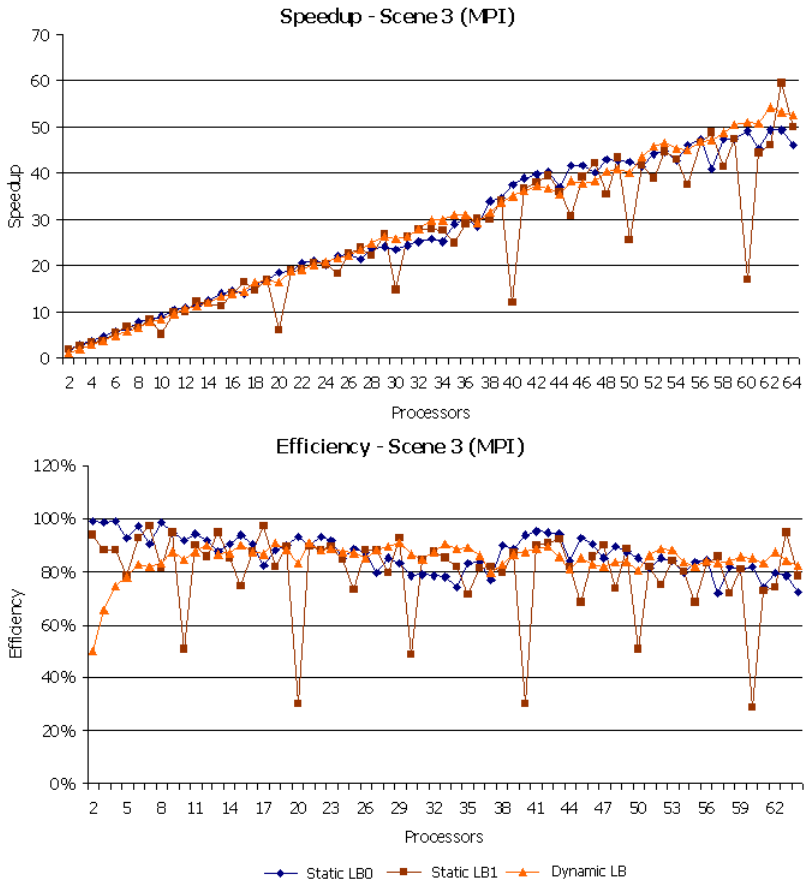

Fig. 5. Speedup and Efficiency for "scene 3"

LB1 technique. A key reason for this is that the master node in dynamic load-balancing does not contribute to the actual processing, as we see that the efficiency for dynamic-load balancing is only approximate 50% where the efficiency of LB1 is approximately 100%. However, the performance of dynamic load balancing is improved over time for both of the two cases.

The effectiveness of the dynamic load-balancing is clear in the case of scene 3 where we see the same performance graph for the dynamic load-balancing but an inconsistent speedup and efficiency demonstrated by LB1 technique. We find that the speedup and efficiency drops for LB1 mostly when the number of processors are multiple of 20 and also major performance drop is observed for number of processors multiple of 10 and moderate performance drop is observed for number of processors multiple of 5. In the case of LB1, this result is what we expected for such a scene since for the static load balancing the unit load is per

row basis which causes an uneven data distribution for number of processors. Although static load balancing LB0 seems better than the dynamic-load balancing specially for small number of processors and equal for for moderately larger number of processors, for large number of processors performance of dynamic load-balancing is better. Apparently for large number of processors the less active nature of master process in dynamic load-balancing technique will have less impact on the overall efficiency and at the same time performance of static load-balancing LB0 degrades. Cases as example where number of processors are same as number of rows of the viewport, the performance will be dropped drastically.

The memory requirements of the dynamic load balancing is reduced to 50% since initially half of the data work is done by the whole processing resources and the allocated memory is reused for the next round of processing.

5 Conclusion & Future Works

In this paper, we present dynamic load balancing technique for parallel ray tracing where the performance is more stable than the previously implemented static load balancing techniques (LB0 and LB1), specially for the newly created experimental "scene 3". However, if work load is distributed column-wise rather than row-wise in static load-balancing cases for scene 3, we can have better performance. Apparently, it will be required to have scene dependent load-balancing techniques in order to get better performance using static load balancing techniques. In that case, prior information about the pixel wise work load distribution will be required that might not be feasible for different applications, for example in the case of dynamically changing scenes. Thus, for a generic system dynamic load balancing technique will be preferred.

In the dynamic load-balancing technique, master node does not contribute to the work as the same level of the worker nodes and hence the overall efficiency is decreased, specially for small number of processors. Multithreading in master-node can be an effective solution to overcome this limitation or other scheduling algorithms can be applied. The future works are needed to improve ray tracing engine to support additional object shapes and optimization of ray tracing algorithms. Adaptive super-sampling will be effective to reduce unnecessary processing for more pixels where the image is flat, for example the black parts of the example scenes.

References

1. Kadir, S. M. A. and Khan, T.: Parallel Ray Tracing using MPI and OpenMP, Project Report, Introduction to High Performance Computing, Royal Institute of Technology, Stockholm, Sweden (2008)
2. Freisleben, B., Hartmann, D., Kielmann, T.: Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation Clusters, Thirtieth International Conference on System Sciences, Hawaii, Vol. 1, Issue 7-10, pp:596 - 605, DOI 10.1109/HICSS.1997.667407 (1997)