An Prospective Approach to Ray Tracing on Exascale Computers

Ellen A. Porter [*] Robert R. Lewis [†]

Washington State University, Tri-Cities
Program in Engineering and Computer Science

Abstract

Exascale computers, defined as being capable of performing at least one exaflop (10^{18} floating point operations per second) are anticipated to emerge in the next several years. Reaching this scale of computation will require significant hardware and software changes for high-performance computing (HPC). Applications will need to adapt as the architecture of supercomputers change. In this paper we explore those changes and the impact they will have on programming models and application design. We then look at one specific model, the Concurrent Collections Programming Model (CnC) and explore how we can use CnC and Intel's Embree Ray Tracing Engine to build a scalable ray tracing system ready for exascale.

Index Terms: [Computing methodologies]: Ray tracing [Computing methodologies]: Parallel programming languages [Hardware]: Emerging architectures

1 Introduction

Achieving the performance expected from an exascale computer (defined as capable of performing at least one exaflop (10^{18} floating point operations per second) will require modifications to current hardware architecture which will in turn affect programming models and runtime design. Until recent years, performance has increased in line with Moore's law (which is not so much a law as an observation): the number of transistors within an integrated circuit doubled approximately every two years. As we reached a limit on the number of transistors a single chip could contain, hardware architects had to look for other ways to keep up with performance advancement expectations. In order to take advantage of these hardware advances, applications often required redesign.

In addition to the changing algorithms, the data produced as output from high-performance computing (HPC) applications tends to scale in size with compute power. This has produced a need for rendering and visualization algorithms that can take advantage of distributed systems as well as an opportunity to design algorithms that can be integrated into HPC applications to produce results during execution, i.e. in "real time". This paper proposes one such design for ray tracing, a commonly-used rendering technique, using the Intel Concurrent Collections (CnC) programming model, which is being targeted at exascale computing.

2 Reaching Exascale

For the past two decades high performance computing (HPC) progression has been driven by Moore's law. Until 2004, performance of single-core microprocessors increased as predicted as a result of smaller and faster transistors being developed. In 2004, this advancement trend shifted as we reached an inflection point caused by a chips power dissipation [3]. Unable to sufficiently and inexpensively cool a chip, chip designers looked for other ways to increase performance. This came in the form of multi-core processors which are now the building blocks of many HPC systems.

The introduction of multi-core processors on a single node of a cluster caused a shift in parallel application design. Programs using the standard Message Passing Interface (MPI) library [6] could not exploit parallelism on a single node without a rewrite of the underlying algorithms. This resulted in the emergence of hybrid systems that mix MPI and the Open Multi-Processing (OpenMP) libraries. OpenMP [5] being designed for shared memory multiprocessors, each node would execute an OpenMP program controlled overall by MPI using a fixed number of threads to execute a single work-sharing construct such as a parallel loop [2].

As we look towards the next generation of HPC systems a shift in application design will once again be necessary to reach exascale performance. On-chip parallelism along with reduced data movement will be critical for an applications success. Unfortunately, conventional language semantics will not be sufficient to exploit the architecture advances being developed such as inter-core message queues. Therefore, new high-performance parallel programming models and smarter runtimes are being developed.

The majority of these models are data-centric rather than compute-centric, which allows the runtime scheduler to prioritize scheduling computation on nodes or cores where the required data already resides rather than the next available processor [3]. This kind of model will reduce communication which is the predicted bottle neck for exascale systems.

3 The CnC Programming Model

Concurrent Collections (CnC) is one such data-centric programming model, the deterministic semantics of which allow a task-based runtime to programmatically exploit parallelism. In this model, algorithms are designed based on their data and control dependencies [1]. The specifics regarding the execution of the algorithm is then abstracted out of the implementation. This allows the runtime to optimally decide when and where to schedule computation. Hints can also be provided to the runtime through a separate file called a "tuning specification".

The CnC model is built on three key constructs: step collections, data collections, and control collections [1]. A step collection defines computation, an instance of which consumes and produces data. The consumed and produced data, or data items, belong to data collections. Data items within a data collection are indexed using item tags. Finally the control collection describes the prescription, or creation, of step instances. The relationship between these three collections is defined statically in an input file called a "CnC graph".

Developing a CnC application begins with designing the CnC graph file. An algorithm is broken down into computation steps, instances of which correspond to different input arguments. These steps along with the data collections become nodes in a graph. Each step can optionally consume data, produce data, and prescribe additional computation. These relationships, producer, consumer, and control, define the edges in the graph and will dynamically be satisfied as the program executes.

The next and final required step in producing a CnC application is to implement the step logic. The flow within a single CnC

[*]e-mail: ellen.porter@wsu.edu
[†]e-mail: bobl@tricity.wsu.edu

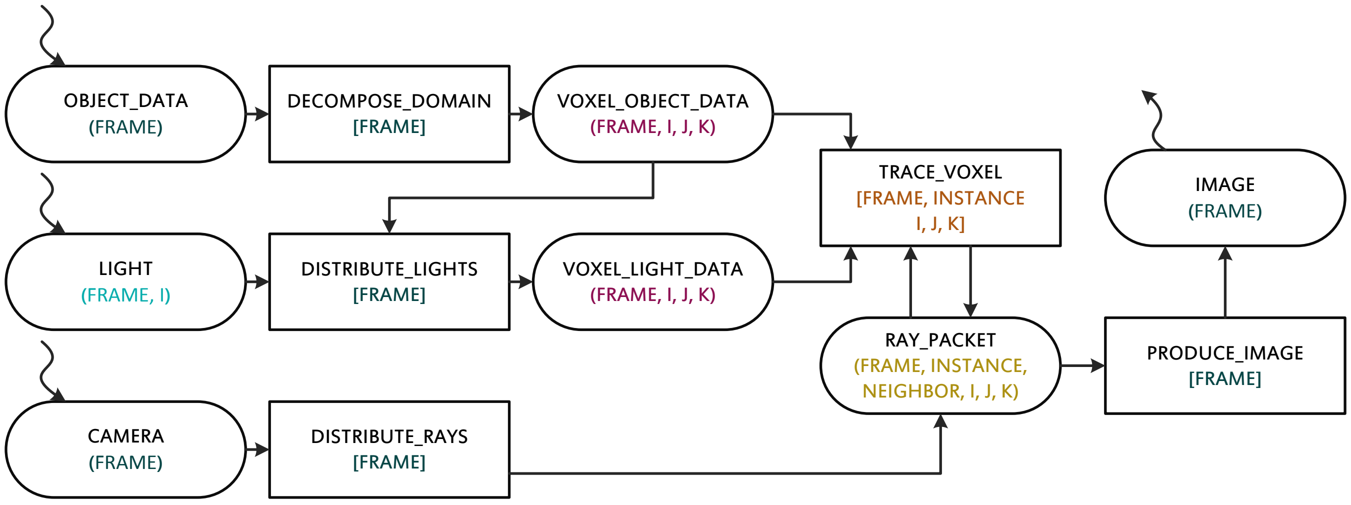


Figure 1: CnC Graph

step is as follows: consume, compute, and produce. This ordering is required as there is no guarantee the data a step needs will be ready when the step beings executing. Internally, CnC will attempt to retrieve the data, if it is not ready, the step will halt execution and try again later. To improve performance, hints can be provided through the tuning specification to ensure steps are only prescribed and scheduled for execution when their required input data is ready.

4 Limitations of Current Ray Tracing Algorithms

Traditional ray tracing algorithms are embarrassingly parallel as no primary ray depends on any other ray. The data needed by each individual ray, however, varies widely as its path is traced, especially as regards secondary or other rays. Acceleration structures, such as k-d trees have been developed to increase ray tracing performance. As the data scales up however, it is no longer possible to store an entire data set in an acceleration structure in shared memory.

One solution is to implement data decomposition. Each node on a distributed system is then responsible for a subset of the domain. Primary rays and secondary rays are then communicated across nodes as the algorithm executes. These types of models typically rely on expensive preprocessing steps that help to balance both the data distribution and rendering work evenly across nodes [4].

5 CnC Ray Tracing Implementation

Implementing ray tracing using CnC allows the algorithm to run on a distributed system which is planned to be extensible to exascale computers and reduces the need for expensive preprocessing seen with many current systems. This is due to the dynamic nature of CnC execution and is similar to the algorithm proposed by Navratil et al. [4]. The CnC implementation beings by partitioning data into voxels, this data is then distributed dynamically at runtime. The ray tracing portion of the algorithm is iterative. Primary rays are sent into the system from the camera and may give rise to secondary rays (shadow, reflection, refraction, etc.) until all rays are fully traced.

Figure 1 shows the proposed CnC graph for distributed ray tracing. It represents data collections as ovals and step collections as rectangles, each with a collection tag (in upper case on its first line) and associated item tag(s) (in square brackets on its subsequent lines). It also shows the dependencies between them. The control collection for the proposed model is static for all steps except TRACE_VOXEL and defined in an initialization step. The graph begins execution when the object data, light data, and camera data are provided, and terminates when the image is produced.

5.1 Tag Collections

The tag collections are different for most data and step collections but share common elements. FRAME refers to one specific frame in the case of an animation. INSTANCE refers to the current iteration. I, J, K are iterators over spatial data, or in the case of LIGHT, the light index.

5.2 Data Collections

OBJECT_DATA contains input data for the scene from the environment. In our case, it is in the form of a Wavefront "`.obj`" file. DECOMPOSE_DOMAIN splits this data into voxels and produces VOXEL_OBJECT_DATA. Triangles that span multiple voxels are duplicated. The LIGHT data collection contains data pertaining to any light sources. VOXEL_LIGHT_DATA contains the same information as LIGHT plus a traced light mesh for each wall of a voxel. CAMERA contains the location and direction of the camera. RAY_PACKET contains all the rays that intersect a voxel wall for a given wall and iteration. IMAGE contains the final image data.

5.3 Step Collections

5.3.1 Decompose Domain

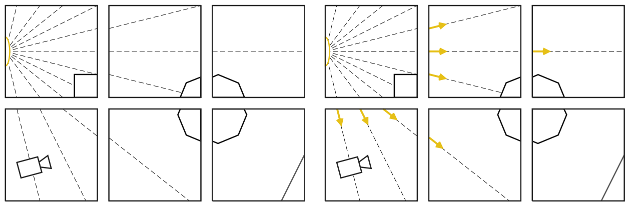
DECOMPOSE_DOMAIN takes the data to be traced as input and produces subsets of that data based on voxel decomposition. As load balancing is not a concern a fast uniformly spaced geometric distribution is sufficient. The number of voxels produced is set at runtime and should be more than the number of nodes available.

5.3.2 Distribute Lights

In order to reduce communication of secondary rays, DISTRIBUTE_LIGHTS lights is responsible for distributing light information to each voxel. This guarantees each voxel will only need to communicate with its direct neighbors. The light information produced for each voxel contains the original light sources as well as a light source mesh for each light and each wall of the voxel. The mesh is produced by tracing rays from each light source to a uniform grid on a voxel wall. Where the rays intersect the wall, a new directional light source is created to light the cell. If the ray is blocked, the point light source will be tagged as in shadow.

5.3.3 Distribute Rays

DISTRIBUTE_RAYS is responsible for sending each voxel its first iteration of ray information. This is an empty set of data for all



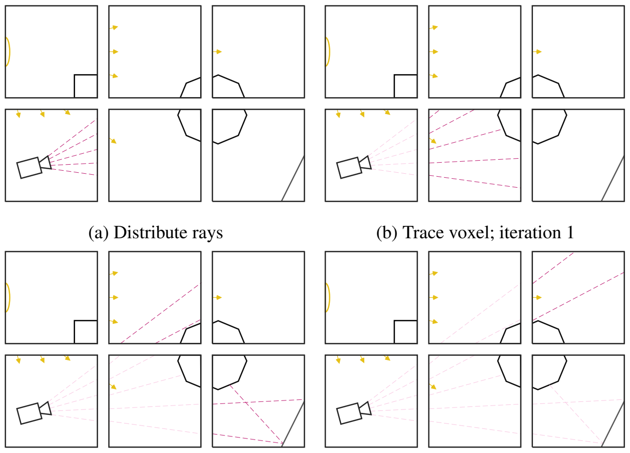
(a) Initial light rays (b) Point light sources

Figure 2: Light Ray Distribution

voxels except the voxel containing the camera if the camera is positioned within the domain. If the camera is outside the domain, multiple voxels may receive data.

5.3.4 Trace Voxel

TRACE_VOXEL is the heart of the application. This step is iterative and prescribes the next iteration as long as there are rays still to trace and it has not reached a maximum threshold. Trace voxel consumes ray packets from each of its neighbors. It then traces the rays over its subset of the domain. If a ray intersects with an object, secondary rays from each light source are considered if the corresponding point light source from the voxels light mesh is not in shadow. Rays that reach the voxel walls are collected and passed to the corresponding neighbor on the next iteration. As each CnC step instance will eventually be executed on a single node of a cluster the step code is integrating with Embree, Intels ray tracing kernel, in order to optimize per node performance.



(a) Distribute rays (b) Trace voxel; iteration 1

(c) Trace voxel; iteration 2 (d) Trace voxel; iteration 3

Figure 3: Iterative Trace Voxel

5.3.5 Produce Image

When trace voxel has converged, a final set of ray packets will be produced. Each ray in that packet contains the information necessary to produce the final image; merging these is the responsibility of the PRODUCE_IMAGE step.

6 Future Directions

6.1 Hierarchy

6.2 Dynamic scenes

6.3 checkpoint and restart

6.4 Global illumination

7 Conclusion

Acknowledgements

The authors wish to thank A, B, C. CnC team at RICE? Intel?

References

[1] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasırlar. The concurrent collections programming model.

[2] W. Gropp and M. Snir. Programming for exascale computers. *Computing in Science & Engineering*, 15(6):27–35, 2013.

[3] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[4] P. A. Navrátil. Dynamic scheduling for large-scale distributed-memory ray tracing. In *EuroGraphics Symposium on Parallel Graphics and Visualization*, 2014.

[5] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.