ANTICIPATING RAY TRACING ON EXASCALE COMPUTERS

By

ELLEN PORTER

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

YOUR SUBMISSION DATE

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ELLEN PORTER find it satisfactory and recommend that it be accepted.

_____
Thesis Committee Chair's Name (fix in wsuthesis.cls), Chair

_____
Committee Member No. 1

_____
Committee Member No. 2 (repeat as needed)

ACKNOWLEDGEMENT

Acknowledge whomever you want to here. (optional)

ANTICIPATING RAY TRACING ON EXASCALE COMPUTERS

Abstract

by Ellen Porter, M.S.
Washington State University
Your Submission Date

Chair: Robert R. Lewis

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 hardware and software changes for high-performance computing (HPC). Applications will need to adapt as the architecture of supercomputers change. Some studies suggest "communication avoiding" algorithms might be the most performant design for future systems. This has created an interest in the development of scalable visualization algorithms and techniques.

In this paper we explore the impact exascale hardware will have on programming models and application design. We then look at one specific model, Concurrent Collections (CnC), and explore how we can use it and Intel's Embree Ray Tracing Engine to build a scalable ray tracing system with an emphasis on communication avoidance and extension for exascale. As exascale hardware does not yet exist, this work is speculative, but we hope it serves as a foundation for future discussion and work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

This is dedicated to ... (optional)

CHAPTER ONE

INTRODUCTION

Achieving the performance expected from an exascale computer will require modifications to current hardware architecture which will in turn affect programming models and runtime[1] design. Until recent years, performance increased in keeping with Moore's "Law" (which is really more of 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 most cases, this involved a greater emphasis on parallelism. Consequently, in order to take advantage of hardware advances, applications, runtimes, and programming models have often required redesign, if not reimplementation.

As we look towards the next generation of high-performance computing (HPC) systems, a shift in application design is again anticipated, this time to reach exascale performance. On-chip parallelism along with reduced data movement will be critical for applications to make optimal use of the hardware and minimize power consumption.

Unfortunately, conventional language semantics will not be sufficient to exploit the architectural advances being developed such as inter-core message queues. Therefore, new parallel programming models and smarter runtimes are being designed. The majority of these models are "data-centric" rather than "compute-centric": They allow, for instance, the runtime scheduler to prioritize scheduling computation on nodes or cores where the required data already resides rather than the next available processor [**?**]. This kind of

[1]We use the term "runtime" in the sense of a library or libraries compiled into and running as part of anapplication which is not specific to the application but which moderates its interface (e.g. memory management, thread prioritization, etc.) with the operating system. It's not just a "library", as it may have its own threads or other execution units.

model will reduce communication which is the predicted bottle neck for exascale systems.

The data produced as output from HPC applications such as fluid simulations or finite-element models tends to scale in size with compute power. This is expected to occur with exascale systems as well and has produced a need for 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. Section **??** proposes one such design for ray tracing, a commonly used rendering technique, using the Intel Concurrent Collections (CnC) programming model.

The rest of this paper is organized as follows: We start with a description of exascale along with a description of the projected trends in programming models that will perform well on exascale. We then explore one programming model, CnC, that is expected to map well to exascale systems. After describing the CnC programming model we analyze current ray tracing algorithms and propose places for improvement for exascale. Specifically, we look at ways we can reduce communication overhead within the algorithm. We then describe the implementation details of a ray tracer developed in CnC and look at how it might perform on future exascale hardware. Finally we conclude with a section on future work.

The work we present here is an extension of that in [**?**].

CHAPTER TWO

BACKGROUND

2.1 Reaching Exascale

Until 2004, performance of single-core microprocessors increased as predicted as a result of smaller and faster transistors being developed (i.e. Moore's Law). At that time, this trend shifted as we reached an inflection point caused by a chips power dissipation [**?**]. 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 (and other) systems.

The introduction of multi-core processors on each node of a cluster caused a shift in parallel application design. Programs using the cross-platform standard Message Passing Interface (MPI) library [**?**] could not efficiently exploit parallelism on individual nodes without a rewrite of the underlying algorithms. The Open Multi-Processing (OpenMP) [**?**] library presented a cross-platform standard for parallel programming on multicore nodes, which led to the emergence of hybrid systems that mixed MPI and OpenMP. The cluster would run a collection of MPI processes, one per node, and each node would then execute an OpenMP program redesigned from the original single-threaded program which used a fixed number of threads to execute a single work-sharing construct, such as a parallel loop [**?**].

Although the exact form of an exascale ecosystem is unknown, research suggests that data movement will overtake computation as the dominant cost in the system. This results from the primary means to increase parallelism is expected to be on-chip, with some predictions suggesting hundreds or even thousands of cores per chip die. As a result, we will

would see a higher available bandwidth on chip along with lower latencies for communication within a node. The lower overhead within a chip provides a significant incentive to develop "communication avoiding" algorithms.

Two means to avoid communication are, first, to re-compute values instead of communicating results when possible and, second, to take account of the need to minimize communication when partitioning the algorithm into parallel functional units.

Many of our current programming models lack the semantics necessary to implement communication-avoiding algorithms. As a result, new languages with additional semantics are being proposed for exascale systems. A common theme among these languages is the ability to statically declare data dependencies and data locality information. These additional details can then be used by the runtime to aid in scheduling and anticipatory data movement.

2.1.1 Task-Based Programming Models

One class of programming model that may map well onto exascale systems are task-based models. They tend to be declarative: An application is broken down into chunks of work and the inputs and outputs to that work are declared in the language semantics. Their explicit data dependencies allow the runtime to optimally schedule and execute the tasks, or chunks of work, in the application.

Execution can often be further improved by the implementation of a secondary specification (a file, typically) separate from the program that provides "hints" to the runtime. The key difference between many task-based models and more traditional programming models is the movement from compute-centric to data-centric application design. Algorithms are designed around the data a task needs to execute and the data it will produce rather than designed around the computation.

2.2 Limitations of Current Ray Tracing Algorithms

Traditional parallel ray tracing algorithms are "embarrassingly" so as no ray depends on any other ray, at least on the same level of the ray tracing tree. The data needed by each individual ray, however, varies widely as its path is traced. Acceleration structures such as k-d trees have been developed to increase ray tracing performance. As the size of the scene increases, 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, secondary, and subsequent 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 [**?**].

Load balancing, a significant bottleneck on todays systems, may not be easily implemented on exascale systems. The proposed smarter programming models and runtimes, on the other hand, will allow for scheduling and data movement decisions to be made at runtime which will help reduce imbalance in a system. Data and computation can be dynamically migrated off of overworked nodes (assuming a properly sized granularity for tasks and data).

CHAPTER THREE

PREVIOUS WORK

Cite previous work (using bibtex).

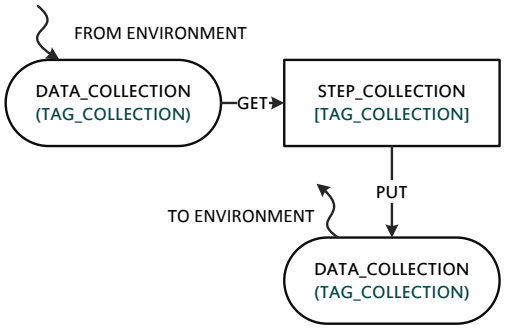


Figure 4.1: CnC Graph Semantics

CHAPTER FOUR

DESIGN

4.1 The CnC Programming Model

The Concurrent Collections Programming Model (CnC), developed by Intel, is one such data-centric programming model. Its deterministic semantics allow a task-based runtime to programmatically exploit parallelism. In addition, it allows for a secondary file, called a tuning spec, to provide additional hints to improve performance.

The CnC model can be thought of as a producer-consumer paradigm where data is produced and consumed by tasks, or "steps" in CnC terminology. The produced and consumed data is declared explicitly in an input file, known as a "graph file". The steps themselves are also entities that can be produced. When a step produces another step, this is known as a "control dependency" and is also declared in the graph file.

Figure **??** shows a graph file. The rectanglees are "step collections", the "data collections" are ovals, and the dependencies are directed edges between them. The title of a step collection is usually a descriptive verb and the title of a data collection is usually a descriptive noun. The control dependencies are not shown. A text description of Figure **??** (which

includes control information) is provided to CnC when designing a CnC application. Section **??** shows an example of this.

By declaring all dependencies between steps and data, the specifics regarding how the algorithm is executed is abstracted out of the implementation. For example: It is clear what data is needed by a given step, so if that data has not been produced yet, the step will not be scheduled. This allows the runtime to optimally decide when and where to schedule computation. By way of contrast, in a conventional multithreaded program, it is the responsibility of the programmer to guarantee that a thread's inputs are available and consequently start the thread.

For some more complicated semantics, additional hints can be provided to the runtime through a "tuning specification". As the tuning specification usually is in a separate file, this makes it easy to run the same program on different architectures, as no rewrites of the application are necessary to switch platforms: just the tuning specification.

4.1.1 Language Specifics

The CnC model is built on three key constructs; step collections, data collections, and control collections [**?**]. A step collection defines computation, an instance of which consumes and produces data. The consumed and produced data items belong to data collections. Data items within a data collection are indexed using item "tags": tuples that, like primary keys, can uniquely identify a data item in the data collection. Finally, the control collection describes the prescription, or creation, of step instances. The relationship between these collections as well as the collections themselves are defined in the graph file.

Developing a CnC application then begins with designing the 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 the graph. Each step can optionally consume data, produce data, and/or prescribe additional computation.

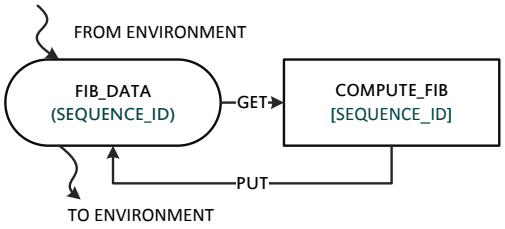


Figure 4.2: A CnC Graph to Compute the Fibonacci Sequence

These relationships: producer, consumer, and control, define the edges of 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 step is: consume, compute, and produce. This ordering is required as there is no guarantee the data a step needs will be ready when the step begins executing. This is due to steps being preemptively scheduled when they are prescribed. Most of the time the data *will* be ready when a step begins execution, but occasionally and often due to an implementation error, a steps data may never be available. Internally, if the data is not ready when a step begins execution it will halt execution and try again later. To improve performance, hints can be provided through the tuning specification to increase the likelihood that steps are schedule for execution when their required input data is ready.

4.1.2 Example

Figure **??** shows an example of a simple iterative implementation of the Fibonacci sequence. This application consists of one step, COMPUTE_FIB, which takes the previous two computed values as input and produces the next value in the sequence. One data collection, FIB_DATA, exists for the application. Data within the collection is indexed by a tag consisting of the sequence number. Tags 1-5, then index the values 1, 1, 2, 3, 5, respectively. The first two values of the data collection are produced by the environment, the rest of the values in the collection are produced as needed by COMPUTE_FIB. A tag exists for

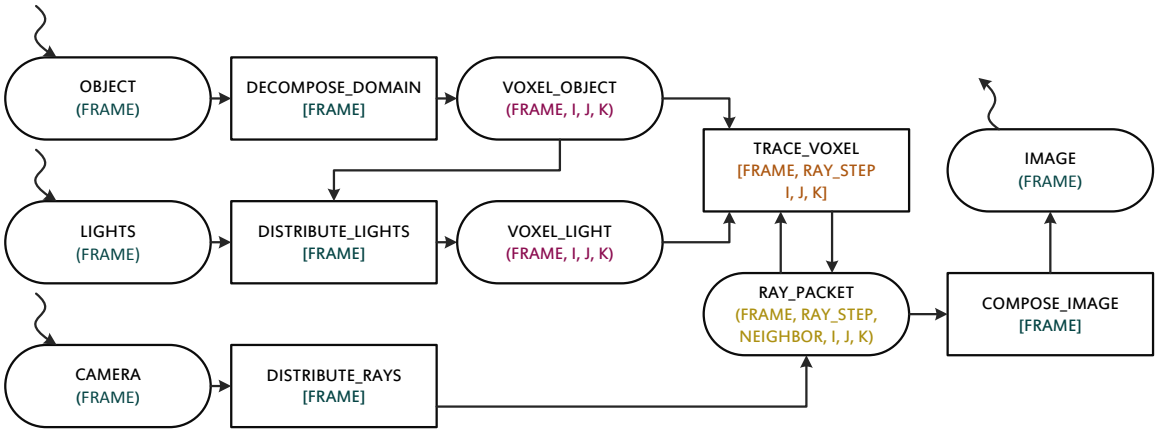


Figure 4.3: CnC Graph

COMPUTE_FIB as well. We can index this collection by the integer sequence a particular instance will produce. For example, the step instance at tag 3 will consume the data at tag 1 and 2, and produce data at tag 3. Specifically it will consume 1, 1 and produce 2. The number of steps executed in this example is provided by the environment.

CHAPTER FIVE

IMPLEMENTATION

5.1 Exascale Ray Tracing

When designing our ray tracing prototype for exascale we focused on two key aspects: (a) producing a task based application with (b) an emphasis on avoiding communication. For simplicity, we consider here only ray tracing scenes without reflection or refraction, although our proposed algorithm can be extended to handle either in the future. Our algorithm uses a simple voxel decomposition to split the work required to trace a scene (which we will henceforth refer to as a "domain").

Primary camera rays are then sent into the system and propagate through the domain. As secondary rays introduce most of the uncertainty in the amount of communication necessary in a ray tracing algorithm, we introduce a pre-processing technique that distributes light information to each voxel prior to tracing the domain. This allows the ray tracing step within each voxel to be completely independent of the data in the rest of the domain. Using this algorithm we can predict an upper bound on the amount of communication necessary (a domain that contains no data), and extrapolate from there rough estimates on how our algorithm might perform on an exascale system.

5.1.1 Implementation

We chose to implement our ray tracer using Intels implementation of CnC, which is built on top of their Thread Building Blocks (TBB) library. This runs on todays multicore systems but has the potential to do so on anticipated exascale systems.

Figure **??** shows the graph for our distributed ray tracer. It shows data collections, step collections, and the dependencies between them. The tags corresponding to each collection are also shown. The control collection for the proposed model is static for all steps and

defined in an initialization step. The graph begins execution when the object, lights, and camera data are provided, and terminates when it produces an image.

Let us consider the parts of the graph individually.

5.1.2 Tags

The tags are different for many of the data and step collections but share common elements. The FRAME tag refers to one specific frame in the case of an animation. The INSTANCE tag refers to the current iteration. The I, J, and K tags are iterators over 3D spatial data, selecting a specific voxel. The RAY_STEP tag allows for multiple traversals of the same voxel in the same frame for secondary rays.

5.1.3 Data Collections

The OBJECT data collection contains input data for the domain from the environment. Currently, this data is extracted from WaveFront "`.obj`" files. The DECOMPOSE_DOMAIN step collection partitions this data into voxels, producing the VOXEL_OBJECT data collection. Objects that span multiple voxels are duplicated. The LIGHTS data collection contains data pertaining to light sources. The VOXEL_LIGHT data collection contains the same information as LIGHTS plus a traced light mesh for each wall of a voxel. The CAMERA data collection contains the location and direction of the camera. The RAY_PACKET data collection contains all the rays that intersect a voxel wall for a given wall and iteration. The IMAGE data collection contains the final image data.

5.1.4 Step Collections

Recall that these are where the computation is done. They may be implemented in any programming language CnC supports, which is most of them.

DECOMPOSE_DOMAIN

As mentions in Section **??** the DECOMPOSE_DOMAIN step 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 uniformly-gridded voxel decomposition is sufficient. The number of voxels produced is set at runtime and should be more than the number of nodes available.

DISTRIBUTE_LIGHTS

In order to reduce the communication of secondary rays, the DISTRIBUTE_LIGHTS step 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 uniformly spaced points along a voxel wall, see Figure **??**. Where the rays intersect the wall, new point or directional light sources are created. If the ray is blocked, the node in the light mesh is tagged as in shadow.

DISTRIBUTE_RAYS

The DISTRIBUTE_RAYS step is responsible for sending each voxel its first iteration of ray information. This is an empty set of data for all voxels except the voxel containing the camera if the camera is positioned within the domain. If the camera is outside the domain (e.g. in an orthographic view), multiple voxels may be initialized.

TRACE_VOXEL

The TRACE_VOXEL step is the heart of the application. This step executes multiple times for each voxel, depending on the size of the domain. Each time TRACE_VOXEL executes, it 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 or directional light source from the voxels light mesh

is not in shadow.

Rays that do not intersect objects within the voxels and reach the voxels walls are collected and passed to the corresponding neighbor on the next iteration. See Figure **??**. Because we are not considering reflection or refraction, we know the maximum amount of times we will have to communicate a single ray across voxel borders in the worst case is proportional to domain size. This allows us to prescribe the a maximum number of instances of TRACE_VOXEL in an initialization step. For the example in Figure **??**, that maximum is 3. As each step will eventually be executed on a single node of a cluster we plan to implement TRACE_VOXEL using Embree, Intels ray tracing kernel, in order to optimize per node performance.

PRODUCE_IMAGE

When all steps in TRACE_VOXEL have completed, the final set of ray packets is produced. Each ray in that packet contains the information that may be necessary to produce the final image. Merging these is the responsibility of the PRODUCE_IMAGE step.

5.1.5 Textual CnC Graph File

To give a more concrete example of how the CnC graph file is implemented, we include a simplified version of the textual representation of TRACE_VOXEL in Figure **??**. In addition to the step declaration, the code includes the SCENE and RAY_PACKET data collections as well as the control dependencies from the environment for TRACE_VOXEL.

Under the *Item collection declaration* section we see two item collections being declared. Once for the domain and one for the ray packets. Instances of the domain indexed using frame, i, j, and k which correspond to the frame in the case of an animation and the 3-dimensional identifier for a given voxel. Ray packets are indexed similarly but also include an entry for the current ray step as well as a neighbor identifier.

Under *CnC steps* we see the declaration for TRACE_VOXEL. Each TRACE_VOXEL

step is indexed using the frame, the voxels i, j, k and the current ray step. The specific instances of the domain and ray data consumed and produced by TRACE_VOXEL can be declared using the RAY_STEP tag. The step will always consume its scene data. It may then consume an incoming ray packet and/or produce an outgoing ray packet for each of the voxel's interior walls.

Under *Input output relationships from environment* we see the control for TRACE_VOXEL. In the initialize step we will produce an instance of TRACE_VOXEL for every frame in our animation, for every step in our ray steps, and for each voxel in our scene's decomposition.

.49

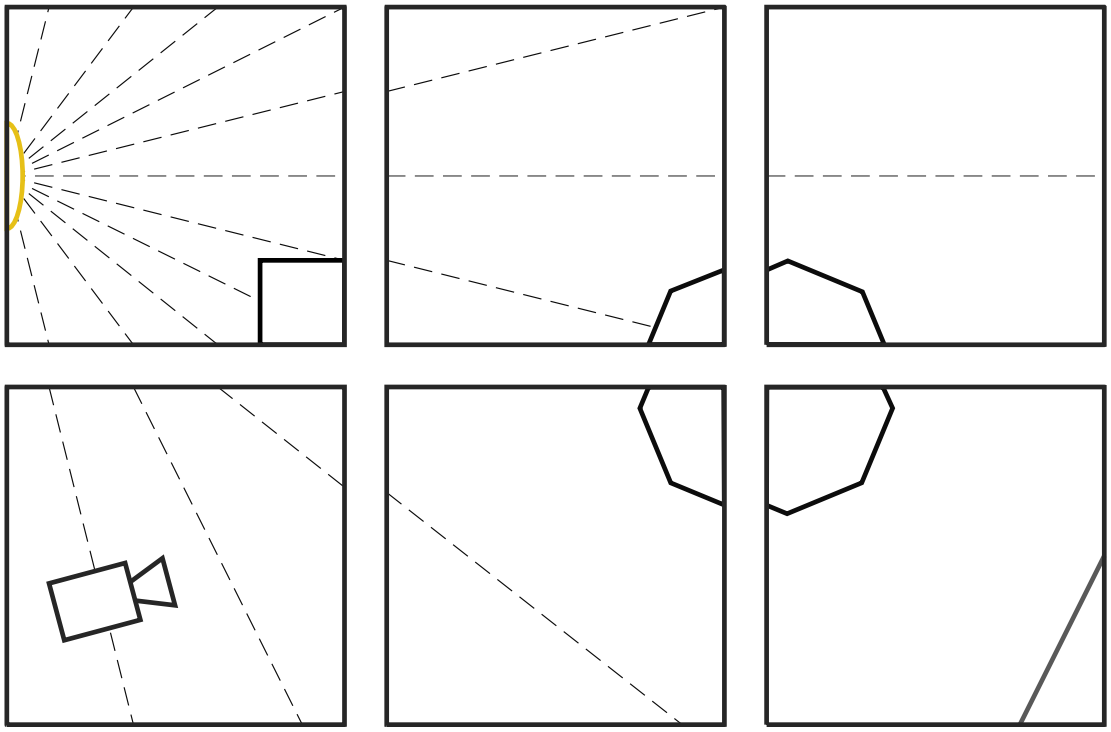


Figure 5.1: Initial light rays

.49

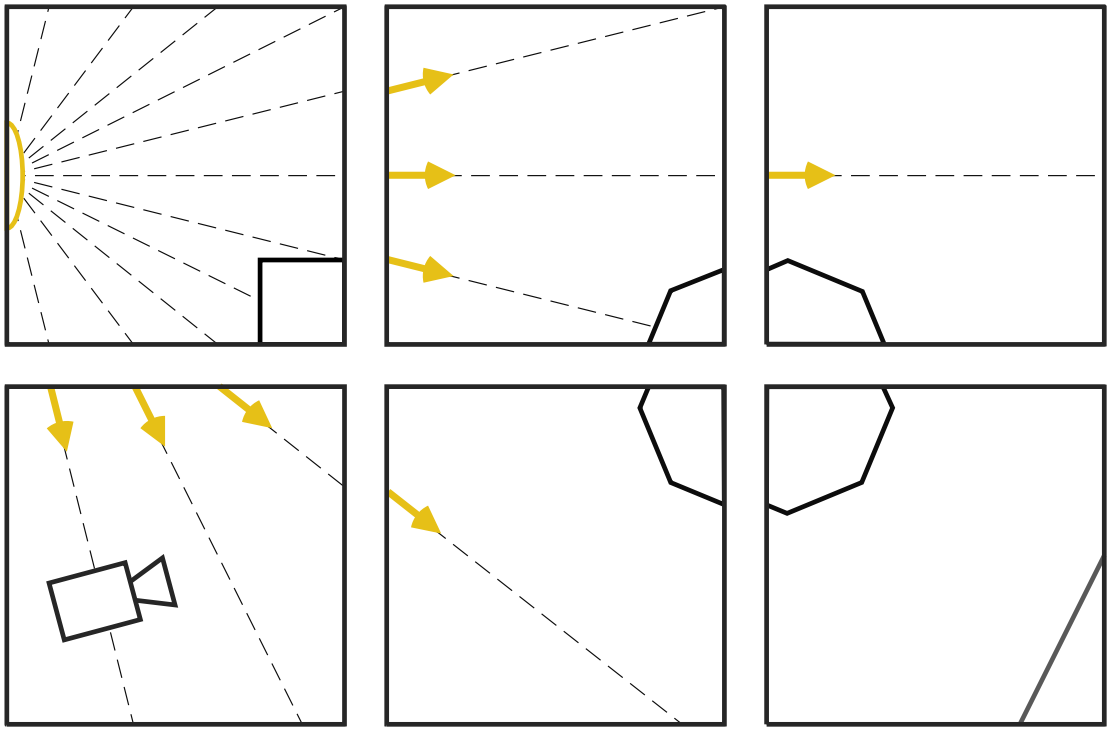


Figure 5.2: Point light sources

Figure 5.3: Light Ray Distribution

.49

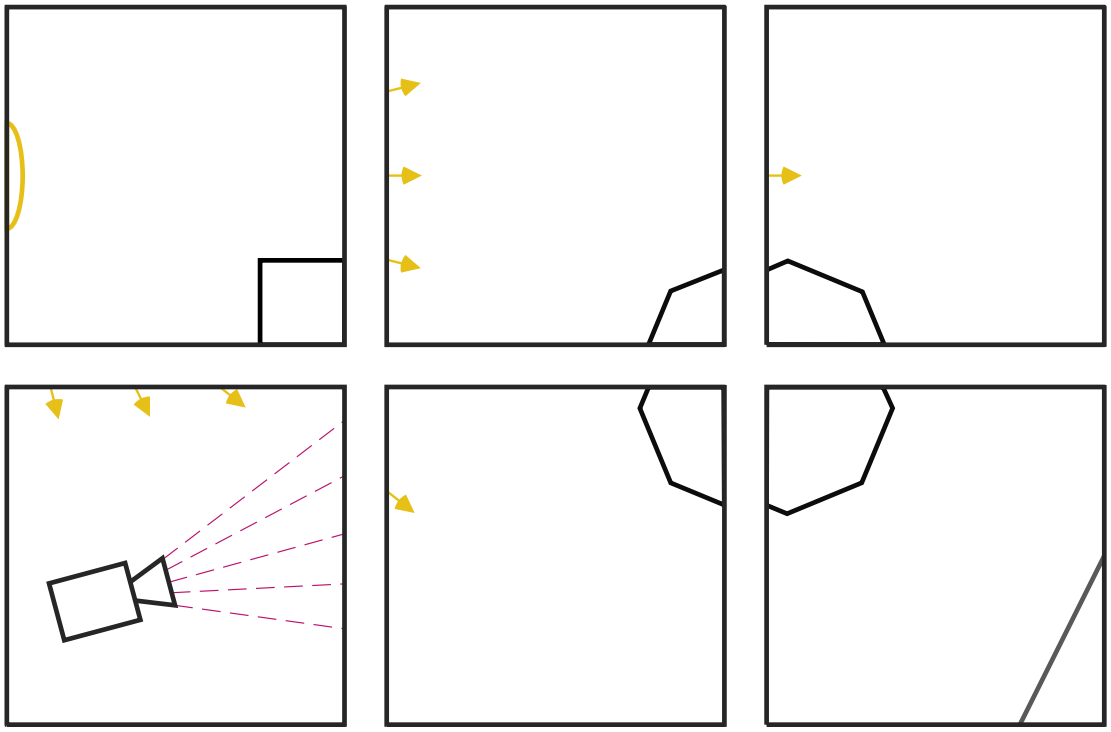


Figure 5.4: Distribute rays

.49

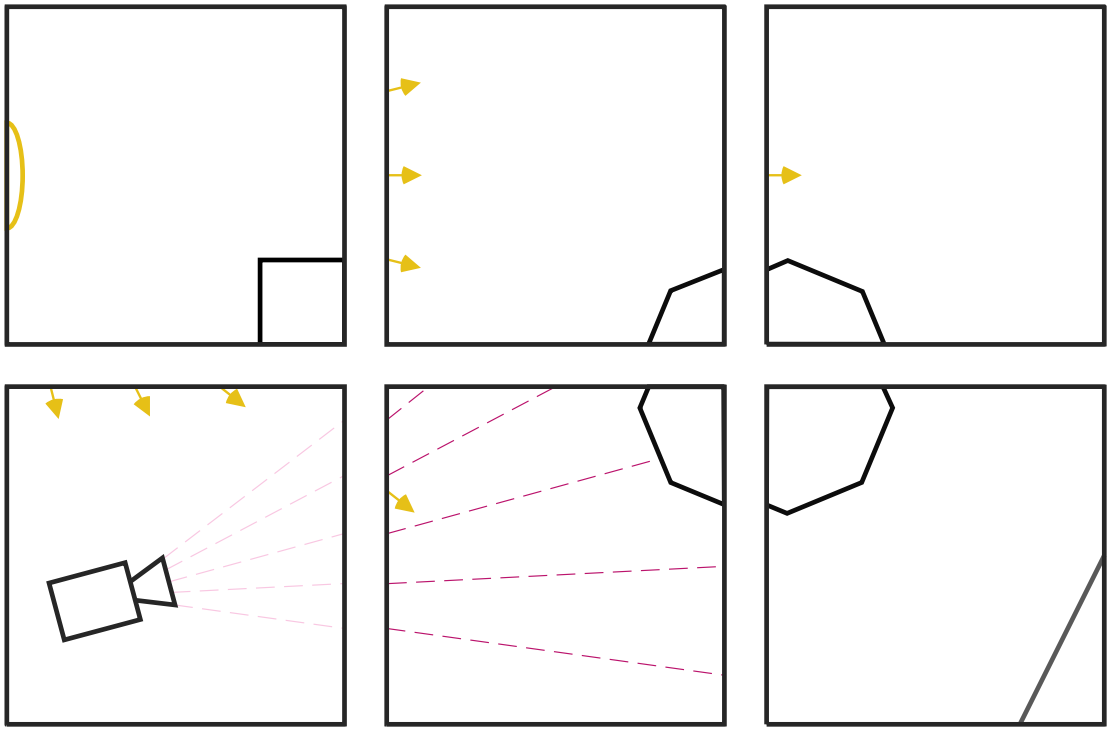
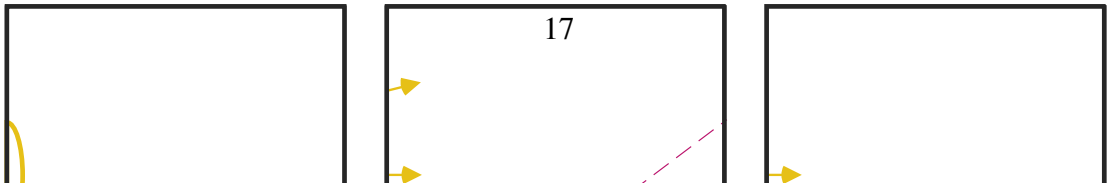


Figure 5.5: Trace voxel; ray step 1

.49

17



```
/*****************************************************************
//* Item collection declarations */

// data for each voxel, produced by decompose_domain
[ voxel_object *voxel : frame, i, j, k ];

// ray data passed to neighbors, produced by camera and trace_voxel
[ ray_packet *rays : frame, ray_step, neighbor, i, j, k ];

/*****************************************************************
//* CnC steps */

( trace_voxel : frame, ray_step, i, j, k )
<- [ voxel: frame, i, j, k],
   [ rays : frame, ray_step  , 0, i  , j  , k
] $when(i<#voxels_i-1),
   [ rays : frame, ray_step  , 1, i  , j  , k
] $when(i>0),
   [ rays : frame, ray_step  , 2, i  , j  , k
] $when(j<#voxels_j-1),
   [ rays : frame, ray_step  , 3, i  , j  , k
] $when(j>0),
   [ rays : frame, ray_step  , 4, i  , j  , k
] $when(k<#voxels_k-1),
   [ rays : frame, ray_step  , 5, i  , j  , k   ] $when(k>0)
-> [ rays : frame, ray_step+1, 0, i-1, j  , k
] $when(i>0),
   [ rays : frame, ray_step+1, 1, i+1, j  , k
] $when(i<#voxels_i-1),
   [ rays : frame, ray_step+1, 2, i  , j-1, k
] $when(j>0),
   [ rays : frame, ray_step+1, 3, i  , j+1, k
] $when(j<#voxels_j-1),
   [ rays : frame, ray_step+1, 4, i  , j
, k-1 ] $when(k>0),
   [ rays : frame, ray_step+1, 5, i  , j
, k+1 ] $when(k<#voxels_k-1);

/*****************************************************************
//* Input output relationships from environment */

( $initialize: () )
-> (trace_voxel : $range(0, #num_frames), $range(0, #ray_steps),
           $range(0, #voxels_i), $range(0, #voxels_j), $range(0, #voxels_
```


Figure 5.9: The TRACE_VOXEL Section of the CnC Graph File. This has been somewhat simplified for readability.

CHAPTER SIX

EVALUATION

6.1 Example: San Miguel

Although purely theoretical, we can walk through a potential execution of our algorithm and draw some conclusions on how it may perform. Using the San Miguel data set and assuming we decompose the domain into 27 equal parts, we get the distribution shown in Figure **??**. This will be the cost of communicating the initial datasets to the nodes where execution will take place. We will incur these costs again if data needs to be moved once the algorithm starts executing.

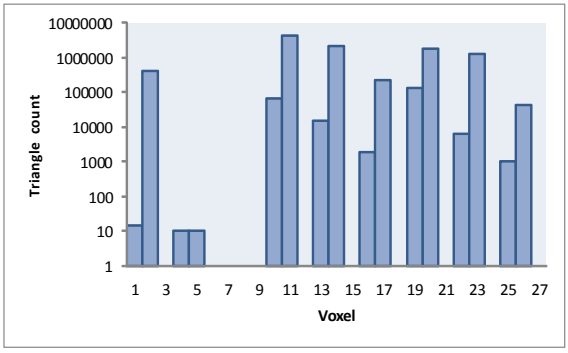


Figure 6.1: San Miguel Data Decomposition

If we consider a machine with 8 cores, we might hope to get a distribution such as that shown in Figure **??**. This roughly distributes the data evenly putting an emphasis on positioning neighbors on the same cores. Depending on the lights in the scene, we also have to consider the cost of distributing them, building the light mesh, and sending the light mesh data to each node. For this example I am assuming the costs associated with the lighting is negligible next to the cost of distributed the data in the scene.

¿¿¿ RESUME

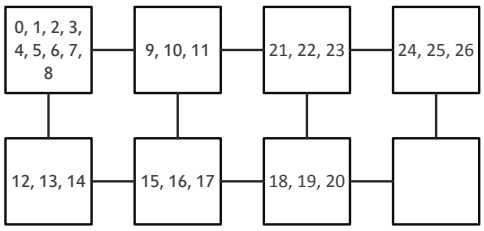


Figure 6.2: Sample Data Distribution

Once the algorithm begins execution we can assume a worst case of 3 ray steps being necessary to trace rays from the camera through the scene. This means each voxel will need to communicate with its neighbors a maximum of 3 times. Some of this communication will be within a node, some will be between nodes. The cost of this communication needs to be factored into our estimates for total time to trace a given scene. Acquiring accurate estimates for this equation will be the focus of our future work as exascale systems do not yet exist.

CHAPTER SEVEN

CONCLUSIONS

7.1 Conclusion

We have taken the first steps in designing and implementing a ray algorithm for exascale. Although we do not have exascale hardware yet, we are hoping to emulate our implementation on todays systems. Integrating our CnC graph file with Embree to produce a fully funcitonal distributed ray tracer will be the focus of our future work, but we also hope to explore other aspecs of exascale such as hybrid systems and examine how they may impact our design.

CHAPTER EIGHT

FUTURE WORK

What sort of things could follow from this work?