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

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

This is dedicated to ... (optional)

CHAPTER ONE

INTRODUCTION

"Ray tracing is the future and always will be". [TODO: look up quote, use idea as first introduction paragraph]

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,

[1] We use the term "runtime" in the sense of a library or libraries compiled into and running as part of an application 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.

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.

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.

1.1 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 [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 (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 [5] could not efficiently exploit parallelism on individual nodes without a rewrite of the underlying algorithms. The Open Multi-Processing (OpenMP) [4] 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 [2].

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.

1.2 Ray tracing

Ray tracing is one of the rendering techniques often used in computer graphics to render three-dimensional scenes into two dimensional images [Shirley]. A ray tracing renderer takes a set of objects in 3D space as input and then casts viewing rays into the scene to determine the color of each pixel in an output image.

As outlined in Shirley [ref], ray tracing has three main components, ray generation, ray intersection and shading. The first component is responsible for computing viewing rays, which are rays from an origin position to a point on an image plane. An image plane is the plane that will contain the final output image; it is positioned between the eye, or origin, location and the scene to be rendered. [graphic would be good here?].

Each viewing ray is then cast into the scene where we apply the second component, ray intersection. For each ray we need to know what object in the scene it intersects first. This tells us which object can be seen by that viewing ray, allowing us to color the pixel of the image that the viewing ray passed through by shading, the third component, our intersected object. Most shading models require information from secondary rays in order to compute the correct color. These secondary rays include light rays which directional rays pointing from light sources towards the intersection point as well as reflected and refracted rays depending on the type of material of the object at the intersection point.

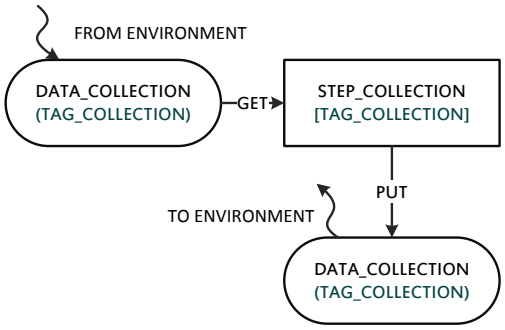

Figure 2.1: CnC Graph Semantics

CHAPTER TWO

PREVIOUS WORK

2.0.1 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 2.1 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 2.1 (which includes control information) is provided to CnC when designing a CnC

application. Section 1.2 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.

Language Specifics

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 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. These relationships: producer, consumer, and control, define the edges of the graph and

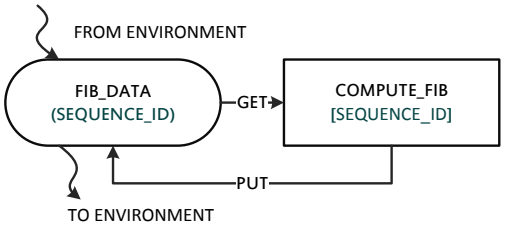

Figure 2.2: A CnC Graph to Compute the Fibonacci Sequence

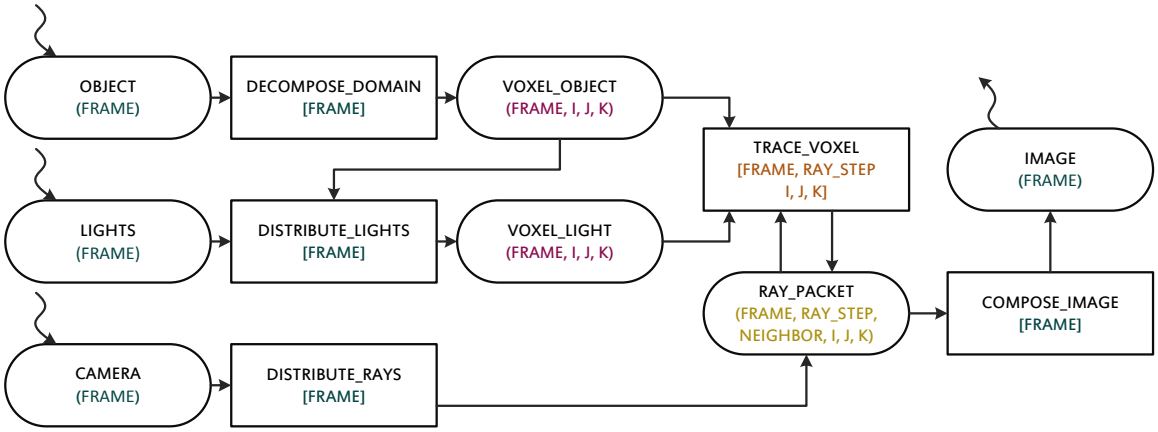

Figure 2.3: CnC Graph

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.

Example

Figure 2.2 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 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.

BIBLIOGRAPHY

[1] Zoran Budimlic, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasırlar. The concurrent collections programming model.

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

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

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

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.