Distributed Ray Tracing: Parallelizing graphical computation

by #ConcurrentSwag

Arthur Berman
Tufts University

Maxwell Bernstein
Tufts University

Thomas Colgrove
Tufts University

Kate Wasynczuk
Tufts University

Abstract

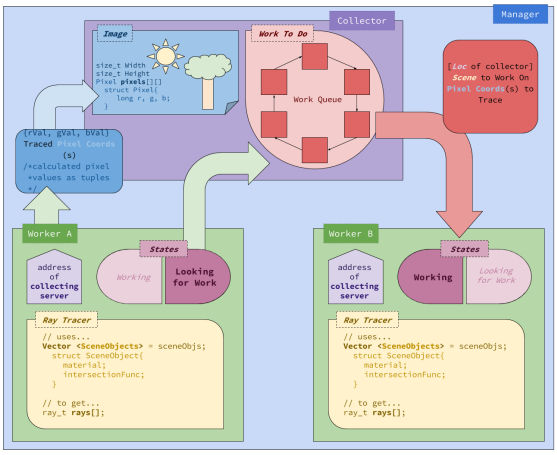
We want to parallelize the ray tracing algorithm across multiple machines instead of just multiple processes on one machine. These machines and processes need to communicate over a network.

1 Design

We have dubbed our distribution/concurrency model a Worker/Collector Architecture. The Collector in this model has knowedge of work to be done, and divides this work into chunks to be put on a work queue, and consumed by worker processes. The Woker processes consume work from the work queue, and produce results. Finally, the collector collects (consumes) these results, and assembles them into a cohesive whole.

We chose this design because it offers high extensibility and keeps mutable resource sharing to a minimum. We could extend this same architecture to many different divide-and-conquer problems. Furthermore, at no point in our model do the workers or collector have access to a shared mutable state.

Our model is also the easiest to employ in a practical manner. Based on our current archtecture, we could create a wrapper for both Collectors and Workers, called Manager. This manager could orchestrate multi-scene rendering efforts, spin up remote workers, and more.



This diagram shows two Workers and one Collector. The Collector produces an image with an arbitrary width and height. The Collector can be in one of two states: assigning work, and outputting image. When the Collector is assigning work, it waits for messages from Workers, then allocates work to each Worker that asks. The Collector selects work based on the current front of a circular queue. Once work is complete (read: returned by a Worker), it is removed from the queue.

Work consists of a named scene description file and a set of pixels. Once every pixel in the image has been rendered, the Collector transitions to outputtting an image.

When the Collector is outputting an image, the Collector writes a file to the filesystem containing the rendered image.

There are two Workers connected to the server. Worker A is looking for work. Worker B has received work and is working. Worker A has sent a message to the server requesting a set of pixels. When the server responds, Worker A will transition to the working state.

Worker B, in the working state, is iterating through the set of pixels in its packet of work. For each pixel, it uses a traced ray to determine the correct color. Once all pixels have been rendered, it returns the set of rendered colors

to the server and transitions to the look for work state.

2 Reflection on design

The two best design decisions we made work hand in hand: the work queue, and the stateless message protocol.

The work queue allows for clients to fail for whatever reason (network issues, segfaults, etc) and the image to continue to be rendered successfully.

The stateless message protocol accomplishes the following:

- Limit number of open connections.
- Reduce complexity in both Worker and Collector state machines.
- Limit technology requirements (i.e. no keep-alive connections, long-polling, etc)

3 Analysis of outcome

We were able to achieve our minimum and maximum deliverables from the initial proposal and further. We can, by means of a Python script, also create moving GIFs of renders of scenes. See the example of snowman with snow falling.

The math, while difficult, did not prove nearly as intractable as we initially thought, *even though* we do not have backgrounds in mathematics. Additionally, the math required for ray tracing allows for easy frame splitting. We were not sure if this was possible.

4 Reflection on development plan

We operated in two-week sprints as pairs. Each pair followed the COMP 40 pair-programming paradigm, with two members of the group on one computer, with one driving and the other providing continual feedback.

Throughout each sprint, each pair pushed the code to a feature branch, and then pair reviewed the other pair's code. After pair review (and appropriate revisions), feature branches were merged into the development branch.

4.1 Sprints

(1) Thomas and Max: single-threaded ray tracer, Kate and Arthur: scene description parser

(2) Thomas and Arthur: integrate ray tracer and scene description parser, Kate and Max: networking stack, message passing, and work queue

(3) Arthur and Max: integrate work queue and message passing into ray tracer, Thomas and Kate: slide deck and presentation materials

4.2 Reflection

For the most part, the pair-sprint format worked well. It forced every member of the group to have a solid understanding of the code and the architecture. Additionally, it ensured punctuality by way of peer pressure.

Occasionally, one team would require assistance from another in order to finish a sprint.

4.3 Interface management

The weekly full-group meetings to decide interfaces worked well. We found that in practice, a team would make the call to change an interface in order to write better code. This was not a problem.

5 Bug report

5.1 JSON? What JSON?

Occasionally, the Worker would have trouble parsing larger scene files sent by the Collector. JsonCpp would complain about poorly formed documents, that it expected an open brace here, quote there, etc. Even after very detailed manual inspections, the transmitted JSON appeared completely fine.

One time, however, we noticed a non-ASCII character rendered as a question mark in the JSON output. This kind of output only appears when the terminal can't render the (presumably invalid) character. On larger JSON files, these question mark characters appeared at very regular intervals - about every 512 bytes, the amount we `recv()` each call.

It turned out that the amount we `recv()`'d completely filled our buffer, leaving room for no NULL-terminator, so we would read past the bounds of buffer whenever we loaded JSON. If we `recv()`'d one less character, the problem disappeared.

5.2 SIGBUS...?

After debugging the JSON issue mentioned above, the Worker could parse JSON, render pixels, and then send to the Collector. However, the Collector would exit with SIGBUS on images larger than roughly 200 by 300 pixels. When we ran the Collector under gdb and unrolled the stack, there were nearly 500 stack frames of the same function - `read_from_sock`.

`Read_from_sock`, given a client socket and string buffer reference, will read until there is nothing more to

read, and append the buffer to the string all the while. It was initially written recursively because it was the cleanest way - however, it looked like on large messages (which could be up to five (5) megabytes) and chunk sizes of 512 bytes, there would be a stack overflow that only got noticed when `read_from_sock` tried to `bzero()` a variable.

`Read_from_sock` was rewritten as an iterative function, and order was restored.

5.3 JsonCpp keeps global state

We use JsonCpp as our JSON serialization and deserialization library. This is for the most part an excellent library, but when we started running into errors when ray tracing with multiple Workers. We noticed that JsonCpp would occasionally throw a runtime error about blowing the stack. This error message only cropped up when using two or more Workers.

We decided to track down where this error message came from, and found out that there was an artificial cap on the recursion allowed in the JSON parser. Interestingly enough, the recursion count was kept in a *global variable*, allowing all instances of the `Json::Reader` class to increment it at the creation of each new stack frame. Naturally, we "blew the stack" early and often.

The fix (submitted as a pull request to the JsonCpp project) is to localize the `stackDepth` variable to each instance of the `Json::Reader` class by making it a member variable. The problem disappears after patching the library.

The team submitted a pull request to the JsonCpp library with a fix for the issue.

5.4 Sockets randomly closing

Currently, Workers will fail to `send()` messages, exiting the process with an error about using the wrong protocol for that type of socket. After some preliminary internet research, this error can occasionally appear when trying to `send()` large messages. Our messages can be very large, so we wrote some code to split the larger messages into 512 byte chunks and `send()` those instead.

Instead of fixing the problem, the Workers then exited with SIGPIPE. At some point during the repeated `send()`s, the pipe was closing and Workers could no longer write.

While researching the SIGPIPE errors after the presentation, we realized that a SIGPIPE indicates that the connection is no longer open, and the socket is invalid. Perhaps it was not some bizarre system issue after all! After some investigation into the 'TCPServer' class, it became evident that the unforgiving timeout in `read_from_sock` closed the connection too early. The timeout was increased from 0.01 seconds to 1 second and all was well.

6 Code overview

We have a significant number of modules, utility classes, and some bundled requirements.

6.1 Collector

`Collector` is a networked producer of coordinate ranges and consumer of pixels. It acts as a TCP server and serves coordinates to `Workers` that request it. It also fields requests from `Workers` submitting pixels.

6.2 Worker

`Worker` is a networked consumer of coordinate ranges and producer of pixels. It acts as a TCP client and sends pixels to `Collectors`.

6.3 WorkQueue

`WorkQueue` is a circular queue with constant-time insert, remove, and lookup. It ensures that `Collector` processes work with extreme fault tolerance and speed.

6.4 RayTracer

`RayTracer` is the class that controls all of the actual rendering, `PnmImage` manipulation, and file writing.

6.5 Parser

`Parser` is responsible for the management of scene description files. Our scene descriptions are stored in JavaScript Object Notation (JSON), and `Parser` loads, parses, and creates in-memory representations of the scenes. `RayTracer` uses `Parser`.

6.6 SceneObject

`SceneObject` is the overarching virtual class in the 3D object class heirarchy. It contains traits such as `loc` for location, `material` for material, and methods such as `get_location()` and `get_normal()`. `Sphere`, `PointLight`, and `Box` all inherit from this class.

6.7 Our File Format

In JSON, we represent our scene as two lists: `scene_objects` and `scene_lights`. `Scene_objects` contains objects representing figures in the scene, while `scene_lights` describes the illuminations.

6.8 TCPServer

C++ does not have a built-in or easily bundled TCP server implementation, so we wrote our own. It supports only basic functions like `start`, `serve_request`, and `stop`. It is created with a callback function of signature `string -> string` that handles all requests.

6.9 TCPClient

C++ also does not have a built-in or easily bundled TCP client implementation, so we wrote our own. It supports only basic functions like `connect`, `send_data`, and `receive`.

6.10 PnmImage

`PnmImage` is responsible for the creation and writing of collections of pixels to disk. `RayTracer` manages a `PnmImage`, both alone and in `Collector`.

6.11 Miscelleaneous other

There are numerous other modules like `vector3_t`, `pixel_t`, and `material_t`, that are little more than their types and are fairly self-explanatory. Upon request, further information about any requested module can be produced.

6.12 The work of others

The contents of ./dist are the work of others, with some small patches as described in our bug report.

7 Building and Running our Work

To build our project, run `make worker collector`. This should produce two executables, `worker` and `collector`.

To start a collector that will render "scene.json" and place the result in "output.pnm," run `./collector -s scene.json -o output.pnm` .

Use the `-vpw <width>` and `-vph <height>` options to set the width and height. Use the `-spl <splits>` option to set the number of columns to divide the image into.

Once a collector is running, start a worker using `./worker`. To connect to a remote collector, use `./worker -h <host> -p <port>`.

8 Update from final design

In this section, we update the grading reader on the project's progress since the last submitted document. Sections that have not changed are omitted, and some new sections are added.

8.1 Progress

We can fully render spheres with diffuse color, reflection, and refraction. We are able to render a single image across multiple processes and multiple hosts.

9 Update from initial design

9.1 Diagrams

We have updated the vocabulary in the diagram objdiagram.png to reflect an accurate description of the roles of various processes in the system.

9.2 Progress

We can adequately render spheres with diffuse color, and load a scene from an external description. We are able to render frames in independent processes, but cannot yet render a single image across multiple processes.

9.3 Vocabulary

We have transitioned from the terms "Client" and "Server" to the terms "Worker" and "Collector." Worker processes handle rendering of an arbitrary chunk of the image, which it then sends to the collector process, which collects rendered chunks and assembles them into a finished image.