Massively Parallel 3D Rendering with Granular Computing

Sadjad Fouladi, Alex Poms, Jiren Zhu, Alex Ozdemir, Pat Hanrahan, Kayvon Fatahalian, Keith Winstein
Compiling Google Chrome takes ~16 hours.
Compressing a 15-minute 4K video takes ~7.5 hours.
Rendering each frame of *Monsters University* took 29 hours.
The Problem

Many of these pipelines take *hours and hours* to finish.
The Question

Can we achieve interactive speeds in these applications?
An Answer

Granular Computing*

* terms & conditions may apply
Challenge ①
What is the right platform?

• The architecture?

• The infrastructure?
  • VMs (e.g. EC2, GCE)
    35-second startup
    60-second minimum bill
  • Cloud Functions (e.g. AWS Lambda)
    <1 second startup
    0.1-second minimum bill
    1,000 new cores in <4 seconds
  • Future infrastructure
    <1 millisecond startup & minimum bill

• The programming model?
Challenge ②
How would the application fit into this model?

• Examples:
  
  • In video compression, **parallelism hurts the compression efficiency**. *(ExCamera, NSDI’17)*
  
  • For software compilation, **outsourcing tools don’t scale out well**.
ExCamera:
Low-Latency Video Processing Using Thousands of Tiny Threads
"Apply this awesome filter to my video."
"Look everywhere for this face in this movie."
"Remake Star Wars Episode I without Jar Jar."
ExCamera: Low-Latency Video Processing Using Thousands of Tiny Threads

• Challenge ①
  • We built a system that starts up thousands of threads on AWS Lambda in a few seconds and manages inter-thread communication.
ExCamera:
Low-Latency Video Processing Using Thousands of Tiny Threads

• **Challenge ②**
  • We built a video codec intended for massive fine-grained parallelism.

```
decode(state, frame) → (state’, image)
encode(state, image) → interframe
rebase(state, image, interframe) → interframe’
```
Challenge ①

- An abstraction for describing jobs as a composition of transient & functional containers that can run anywhere.
- Separates the algorithm *(what functions are applied to what data)* from execution *(how is data moved, where are functions run)*.
- Frees the programmer from thinking about reliability, memoization, straggler mitigation, and data movement.
Outsourcing Everyday Jobs to Thousands of Transient Functional Containers
gg:
Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

1. **PREPROCESS**(hello.c) $\rightarrow$ hello.i

   ```
   function: {
     hash: 'VDS0_TM',
     args: [
       'gcc', -E, 'hello.c',
       '-o', 'hello.i' ],
     envvars: [ 'LANG=us_US' ] },
   objects: [
     'VLb1SuN=hello.c',
     'VDS0_TM=gcc',
     'VAs.BnH=cpp',
     'VB33fCB=/usr/stdio.h' ],
   outputs: [ 'hello.i' ]
   ```

2. **COMPILE**(hello.i) $\rightarrow$ hello.s

   ```
   function: {
     hash: 'VDS0_TM',
     args: [
       'gcc', -x, 'cpp-output',
       '-S', 'hello.i',
       '-o', 'hello.s' ],
     envvars: [ 'LANG=us_US' ] },
   objects: [ $\text{HelloMeiRL} =$hello.i',
     'VDS0_TM=gcc',
     'VMRZGH1=cc1', ],
   outputs: [ 'hello.s' ]
   ```

content hash: T0MEiRL

content hash: TRFSH91
Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

- Software compilation

### Compiling Inkscape

<table>
<thead>
<tr>
<th>Tool</th>
<th>Median time</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-core make</td>
<td>32m 34s</td>
</tr>
<tr>
<td>icecc to a warm 48-core EC2 machine</td>
<td>6m 51s</td>
</tr>
<tr>
<td>icecc to a warm 384-core EC2 cluster</td>
<td>6m 57s</td>
</tr>
<tr>
<td>gg to AWS Lambda</td>
<td>1m 27s</td>
</tr>
</tbody>
</table>

### Compiling Chromium browser

<table>
<thead>
<tr>
<th>Tool</th>
<th>Median time</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-core ninja</td>
<td>15h 58m 20s</td>
</tr>
<tr>
<td>icecc to a warm 48-core EC2 machine</td>
<td>46m 01s</td>
</tr>
<tr>
<td>icecc to a warm 384-core EC2 cluster</td>
<td>42m 18s</td>
</tr>
<tr>
<td>gg to AWS Lambda</td>
<td>18m 55s</td>
</tr>
</tbody>
</table>
Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

- Unit tests

<table>
<thead>
<tr>
<th>Tool</th>
<th>Median time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtest-parallel on a local 4-core machine</td>
<td>51m 45s</td>
</tr>
<tr>
<td>gtest-parallel on a local 48-core machine</td>
<td>4m 40s</td>
</tr>
<tr>
<td>gg to AWS Lambda</td>
<td>3m 25s</td>
</tr>
</tbody>
</table>
Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

- Object recognition

<table>
<thead>
<tr>
<th>Object Recognition</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gg local (64 cores)</td>
<td>04m 30s</td>
</tr>
<tr>
<td>gg on AWS Lambda</td>
<td>37s</td>
</tr>
<tr>
<td>Scanner local (64 cores)</td>
<td>05m 39s</td>
</tr>
<tr>
<td>Scanner on cluster (140 cores)</td>
<td>03m 14s</td>
</tr>
</tbody>
</table>
Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

- Functional abstraction makes debugging, tracing, etc. tractable.
- Backend developers can come up with the best optimizations, placement algorithms, storage systems, data-transmission methods, etc.
3D Rendering
Ray tracing

- Ray tracing is a rendering technique by tracing the path of light and simulating the effects of its encounters with virtual objects.

Sphere
- color: blue
- radius: 5
- position: (0 0 0)

Light
- position: (10 5 0)

Camera
- position: (-10 5 1)
- looking at: (0 0 0)
Forward Tracing
Forward Tracing
Forward Tracing
Forward Tracing
Backward Tracing

Each path corresponds to a pixel on the image plane.
Backward Tracing

one path, three rays
Paths are independent
Paths are independent
Ray tracing is embarrassingly parallel
Paths are independent
Ray tracing is embarrassingly parallel

Rays, even for a small tile, can end up all over the scene.

Each rendering thread has access to the whole scene.
Moana Island Scene

- ~50 GB of scene files
- 200 million objects
- Needs ~50 GB of memory to render
- Takes 1.5 hours to render with 24 vCPUs
- ~500 million path
Ray tracing on Lambda

- Lambdas wouldn’t work out-of-the-box because the low memory capacity.
Idea ①
Page in/out parts of the scene as necessary

- Each worker is responsible for rendering a small tile.
- The worker pages in the parts of scene that are necessary.

- **bandwidth cost** $\propto$ **scene size** $\times$ **number of workers**
Idea ②

Workers collectively load the scene, and rays are moved between them.
Idea ②
Workers collectively load the scene, and rays are moved between them

• Each worker is assigned with a small part of the scene, the rays are moved between the workers.

• **bandwidth cost** $\propto$ **number of rays**

• Communication cost is independent of the number of workers.
Modeling the cost

- **Baseline:**
  $N$ 24-core machines, each with enough memory for the whole scene

- **Lambda:**
  $N$ lambda workers, each holding a small part of the scene
Current challenges: Load imbalance

- Not all the scene parts are created equal.
Current challenges: Inter-Lambda communication
Takeaways

Encoding
Compressing this video will take a long time. How do you want to execute this job?

Locally (~5 hours)

Remotely (~5 secs, 50¢)

Cancel
• We want to make it easy for programmers to write 10,000-core workloads and actually execute them on commodity infrastructure.

• 3D rendering, software compilation, unit testing, physics simulations, etc.