Coding in the Cloud - Capturing Programming Behaviors at Scale

EduHPC14: Workshop on Education for High-Performance Computing

Dan Connors and Kyle Dunn
Department of Electrical Engineering
University of Colorado Denver
Overview

- Cloud Motivation
- IPython Notebook
- PyBox- Tracking Programmer Behaviors
- Parallel and Distributed Computing (PDC) Notebooks:
  - Multiprocessing: Mandelbrot, Amdahl’s, Speedup, Efficiency, and Profiling
  - ParallelSchedule: Job Scheduling
  - Synthetic Speedup: Interactive
  - Computer Vision (CV): GPU
Cloud Motivation

- Access to high-performance machines still a barrier to some education programs and reaching out to potential students in Parallel and Distributed Computing (PDC)
- Virtual machine environments are a solution
  - Limitations to libraries and data archives
- MOOC environments scale available content, access, and interaction
  - On-line models have advantage of tracking student learning outcomes and the process of learning
- Examples of good MOOC or PDC environments
  - Hwu [University of Illinois], CUDA Parallel Programming, WebGPU
  - Rixner, Wong, Warren, and Greiner [Rice University] Interactive Python http://www.codeskulptor.com
The IPython Notebook is a web-based interactive computational environment

- A well-structured code development environment
- A framework for observing and recording and results of code execution
- Linking text such as comments, equation generators for mathematics
- Embedded plots and other rich media formatting options
- The cloud coding advantage is that the IPython Notebook Viewer renders the code as a web page and users can read and interact with a remote system without having to install anything on their device.
PyBox Motivation

- Python environments in the cloud have scaled, evolved, and are changing availability
  - www.pythonanywhere.com
  - www.wakario.io
- Blackbox project [4], launched in 2013, collects large amounts of data about student programming activities in the BlueJ IDE (Java source code, compilations, debugger usage & compiler interactions.)
- Course: Computer Vision Acceleration Using GPU and Multicore Architectures
  - Substantial amount of GPU/parallel coding
- Access to resources
  - NVIDIA GPU systems
  - Open Computer Vision (OpenCV) Library
  - Large image dataset (faces, stereoscopic left/right, videos)
PyBox – IPython Notebook Server
IPython

- Python Parallelism
  - Multithreading and multiprocessing support
  - Locks, queues, process pool

- IPython Notebooks
  - Interactive lectures
  - Code and see: embed visualization of parallel performance trade-offs
  - Assignment template and turnin
Mandelbrot Calculation : Block Size

- Overhead of work assignment

```python
import time
size = 1000

for block in [500, 5000, 50000, 500000]:
    tic = time.time()
    z = create_mandel(-2.2, 0.8, -1.5, 1.5, size, block)
    print(block, time.time() - tic)
```

- 500 8.53376102448
- 5000 4.11379504204
- 50000 3.75533103943
- 500000 6.21821498871
Profiling

```
p = pstats.Stats('profile.dat')
p.sort_stats('cumulative').print_stats(10)
```

Mon Nov 10 01:01:21 2014    profile.dat

15 function calls in 6.708 seconds

Ordered by: cumulative time
List reduced from 11 to 10 due to restriction <10>

<table>
<thead>
<tr>
<th>ncalls</th>
<th>ttotal</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.002</td>
<td>0.002</td>
<td>6.708</td>
<td>6.708</td>
<td>&lt;string&gt;:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>1</td>
<td>0.001</td>
<td>0.001</td>
<td>6.706</td>
<td>6.706</td>
<td>&lt;ipython-input-4-3a3f526c3eab&gt;:1(create_mandel)</td>
</tr>
<tr>
<td>1</td>
<td>6.017</td>
<td>6.017</td>
<td>6.022</td>
<td>6.022</td>
<td>&lt;ipython-input-3-b02ba84dd904&gt;:1(mandel_z)</td>
</tr>
<tr>
<td>1</td>
<td>0.680</td>
<td>0.680</td>
<td>0.682</td>
<td>0.682</td>
<td>&lt;ipython-input-2-1df0d7a34369&gt;:1(grid)</td>
</tr>
<tr>
<td>2</td>
<td>0.004</td>
<td>0.002</td>
<td>0.004</td>
<td>0.002</td>
<td>{numpy.core.multiarray.zeros}</td>
</tr>
<tr>
<td>1</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>{method 'copy' of 'numpy.ndarray' objects}</td>
</tr>
<tr>
<td>2</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>/Users/dconnors/Library/Enthought/Canopy_64bit/User/lib/python2.7/site-packages/numpy/core/function_base.py:6(linspace)</td>
</tr>
<tr>
<td>2</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>{numpy.core.multiarray.arange}</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>{range}</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>{method 'disable' of '_lsprof.Profiler' objects}</td>
</tr>
</tbody>
</table>
Parallelism

**Parallel: 3 steps**

**Step 1: import**

```
In [15]: import multiprocessing
```

**Step 2: Create a pool of workers.**

```
In [16]: num_cores = multiprocessing.cpu_count()
pool = multiprocessing.Pool(num_cores)
print num_cores
```

8

**Step 3: Modify the serial map.**

```
In [17]: def create_mandel(min_x, max_x, min_y, max_y, size, block_size, pool):
    x_vals = np.linspace(min_x, max_x, num=size)
    y_vals = np.linspace(min_y, max_y, num=size)

    mandelbrot = grid(x_vals, y_vals, block_size)

    results = pool.map(mandel_z, mandelbrot)  # <- Modification

    output = np.concatenate(results)

    return output
```
Speedup and Efficiency

\[
S = \frac{t_1}{t_n}
\]

Measures processor utilization

\[
\text{efficiency} = \frac{\text{sequential time}}{(\text{number of processors} \cdot \text{parallel time})} = \frac{\text{speedup}}{\text{num of processors}}
\]

\[
e = \frac{S}{n}
\]

In [21]: import pandas as pd
df = pd.DataFrame({'n': cpus, 't': t})

In [22]: df['speedup'] = df[df['n'] == 1]['t'].values / df['t']
df['eff'] = df['speedup'] / df['n']
Execution Time

![Graph showing the relationship between execution time and number of cores]

- **Y-axis:** Time (sec)
- **X-axis:** Number of cores

The graph illustrates a decrease in execution time as the number of cores increases.
Speedup

-mandel
-ideal

number of cores

speedup

0 2 4 6 8 10 12

0 2 4 6 8 10 12
Efficiency
Amdahl Law's Evaluation

**Amdahl's Law**

```python
In [49]:
parallel_ = 0.9
serial_ = 0.1
total_time = df.ix[0,'t']

np = range(1, 13)
t = [ total_time * (serial_ + parallel_/float(n)) for n in np ]
```

```python
In [50]:
fig, ax = plt.subplots(figsize=(8,6))
ax.plot(df['n'], df['speedup'].values, label='actual')
ax.plot(np, total_time/t, label='Amdahl')
ax.set_xlabel('number of cores')
ax.set_ylabel('speedup')
ax.legend(loc='upper left')
fig.show()
```
Amdahl Law’s Evaluation

![Graph showing Amdahl Law's Evaluation]
Single Job Timeline
Parallel Job Scheduling Timeline (8 cores)
Worker Workload Summary
Base Graph

Serial fraction = 0.0%
Overhead fraction = 0.00%

- Speedup vs. Number of cores graph with a linear trend line.
Parallelism Exploration

In [6]: `StaticInteract(plot_speedup, s=RangeWidget(0, 10), o=RangeWidget(0, 10))`

Out[6]:

Slider control in IPython figure
Parallel Exploration

- Serial fraction

- Overhead
Summary and Future Plans

- New opportunities for introducing parallelism
  - Revision of CSI
  - Foundations of computation
  - Parallel programming

- Tracking programmer behaviors
  - Currently log student programmers on a number of code problems

- Looking for colleagues to share framework and collaborate on ideas