NumPyWren

Storage-enabled Scaling of Serverless Supercomputing

Eric Jonas
Postdoctoral Researcher
jonas@eecs.berkeley.edu
@stochastician
PyWren: Scale For Everyone
Not just computer scientists
PyWren: Scale For Everyone

Not just computer scientists
PyWren: Scale For Everyone
Not just computer scientists

Neuroscientists

Microscopy and Optics
PyWren: Scale For Everyone

Not just computer scientists

Neuroscientists

Microscopy and Optics

Geophysicists
PyWren: Scale For Everyone
Not just computer scientists

Neuroscientists

Microscopy and Optics

Geophysicists

Astronomers
PyWren: Scale For Everyone

Not just computer scientists

Neuroscientists

Microscopy and Optics

Geophysicists

Astronomers

Finance and Credit

305 Million Solutions to The Black-Scholes Equation in 16 Minutes with AWS Lambda

Originally Posted May 28, 2017

The research I'm working on involves estimating a firm's probability of default over a variety of time horizons using the Merton Distance to Default model. The dataset contains daily financial information for more than 24,000 firms over the past 30 years. Given that I am calculating the probability of default over five time horizons, applying the Merton model will require solving the Black-Scholes equation roughly 305 million times. Luckily, the model is easily parallelized because the only data needed for the model are the firm's own equity and debt, and the stock and bond yields of the firm. The model also assumes that the firm's assets are perfectly liquid.
PyWren: Scale For Everyone

Not just computer scientists

Neuroscientists
Microscopy and Optics
Geophysicists
Astronomers
Finance and Credit
Developmental Economists
With PyWren, AWS Lambda Finds an Unexpected Market in Scientific Computing

AWS Dev Day

PyWren Web Scraping

I was tasked with scraping information of houses for sale in Massachusetts for my data science class. The target site in question was realestate.com, and they do not tolerate web scraping and will give you a rate limit if you exceed some unknown threshold of pages per minute or had a faulty user-agent.

Not to be deterred by a captcha, I used Selenium and Chromedriver to write a scraper that worked pretty well and importantly: was not caught by realestate’s algorithm. Each page took ~2-3 seconds to scrape.

TerraFlops - Extracting 25 TFLOPS from AWS Lambda

AWS re:INVENT

Massively Parallel Data Processing with PyWren and AWS Lambda

November 30, 2017

Occupy the Cloud: Distributed Computing for the 99% [VISION]

Eric Jonas, Qifan Pu, Shivaram Venkatakrishnan, Ion Stoica, Benjamin Recht (UC Berkeley)
map(function, data)

and... that’s mostly it
map(function, data)
and... that's mostly it

def myfunc(x):
    return x + 1
map(function, data)
and... that's mostly it

def myfunc(x):
    return x + 1

futures = pwex.map(myfunc, [1, 2, 3])
map(function, data)
and... that's mostly it

def myfunc(x):
    return x + 1

futures = pwex.map(myfunc, [1, 2, 3])
print pywren.get_all_results(futures)
map(function, data)
   and... that's mostly it

def myfunc(x):
    return x + 1

futures = pwex.map(myfunc, [1, 2, 3])
print pywren.get_all_results(futures)
[2, 3, 4]
map(function, data)
and... that's mostly it

def myfunc(x):
    return x + 1

futures = pwex.map(myfunc, [1, 2, 3])
print pywren.get_all_results(futures)
[2, 3, 4]
Beyond map?
MOTIVATING LINEAR ALGEBRA
MOTIVATING LINEAR ALGEBRA

High Performance Computing (HPC)
MOTIVATING LINEAR ALGEBRA

High Performance Computing (HPC)

Expensive capital outlay
High speed interconnect
Speed is #1 job
Older technology stack
MOTIVATING LINEAR ALGEBRA

High Performance Computing (HPC)
Expensive capital outlay
High speed interconnect
Speed is #1 job
Older technology stack

Machine Learning
MOTIVATING LINEAR ALGEBRA

High Performance Computing (HPC)

- Expensive capital outlay
- High speed interconnect
- Speed is #1 job
- Older technology stack

Machine Learning

- Focus on deep method
- Everything is streaming
- Does this really work?
MOTIVATING LINEAR ALGEBRA

High Performance Computing (HPC)
- Expensive capital outlay
- High speed interconnect
- Speed is #1 job
- Older technology stack

Machine Learning
- Focus on deep method
- Everything is streaming
- Does this really work?
- "It’s easier to train a deep neural bidirectional LSTM with attention than it is to compute the SVD of a large matrix" - Chris Re
TRENDS AND OBSERVATIONS

Compute more precious

Fast cheap disaggregated state

Algorithms with dynamic parallelism

Operations where compute dominates IO

$O(n^3) > O(n^2)$
NO MOORE FREE LUNCH

NO MOORE FREE LUNCH

DATA CENTER DISAGGREGATION

Cost to store 1 TB
DATA CENTER DISAGGREGATION

Cost to store 1 TB

0
1.75
3.5
5.25
7
DATA CENTER DISAGGREGATION

Cost to store 1 TB

x1e.8xlarge AWS Instance

0
1.75
3.5
5.25
7
DATA CENTER DISAGGREGATION

$6.67/hr

Cost to store 1 TB

x1e.8xlarge

AWS Instance
DATA CENTER DISAGGREGATION

Cost to store 1 TB

- AWS S3: 7
- AWS Instance (x1e.8xlarge): 5.25
- AWS Instance: 3.5
- AWS Instance: 1.75

$6.67/hr
DATA CENTER DISAGGREGATION

- **AWS Instance**
  - x1e.8xlarge
  - Cost to store 1 TB: $6.67/hr

- **AWS S3**
  - Cost to store 1 TB: $0.04/hr
DATA CENTER DISAGGREGATION

Cost to store 1 TB

<table>
<thead>
<tr>
<th>AWS Instance</th>
<th>AWS S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1e.8xlarge</td>
<td></td>
</tr>
</tbody>
</table>

$6.67/hr

$0.04/hr

Read/Write rates per worker

S3 throughput per lambda

Read: 0.4
Write: 0.2

MB/sec

20 40 60 80
DATA CENTER DISAGGREGATION

Cost to store 1 TB

$6.67/hr

$0.04/hr

x1e.8xlarge AWS S3

AWS Instance

AWS S3

S3 throughput in total

S3 throughput

read

write

GB/sec

0 100 200 300 400

0 25 50 75 100 125 150

time (sec)
Linear algebra operations have

DYNAMIC PARALLELISM AND WORKING SET SIZE

![Graph showing dynamic parallelism and working set size over time](image-url)
Linear algebra operations have
Linear algebra operations have

Compute

$O(n^3)$
Linear algebra operations have

\[
\text{Compute } O(n^3) > \text{ Communication } O(n^2)
\]
Linear algebra operations have

Compute

\[ O(n^3) \]

Communication

\[ O(n^2) \]

Matrix-Matrix product
Singular Value Decomposition
Least Squares Solve
Cholesky Factorization
TRENDS AND OBSERVATIONS

- Compute more precious
- Fast cheap disaggregated state
- Algorithms with dynamic parallelism
- Operations where compute dominates IO $O(n^3) > O(n^2)$
NUMPYWREN GOALS
NUMPYWREN GOALS

• No expensive setup (ala PyWren)
NUMPYWREN GOALS

• No expensive setup (ala PyWren)

• Decouple computation and storage
NUMPYWREN GOALS

- No expensive setup (ala PyWren)
- Decouple computation and storage
  - More cores->faster
NUMPYWREN GOALS

• No expensive setup (ala PyWren)

• Decouple computation and storage
  • More cores -> faster
  • More storage -> bigger
NUMPYWREN GOALS

• No expensive setup (ala PyWren)

• Decouple computation and storage
  • More cores -> faster
  • More storage -> bigger

• Elastic parallelism — be careful with compute
NUMPYWREN GOALS

- user facing numpy/matlab-like interface: numpywren
- Low Level IR aimed at LA primitives: lambdapack
- Execution Framework: pywren
NUMPYWREN GOALS

- Usable by anyone who knows Numpy

- User facing numpy/matlab-like interface
  - `numpywren`

- Low Level IR aimed at LA primitives
  - `lambdapack`

- Execution Framework
  - `pywren`
NUMPYWREN GOALS

- Usable by anyone who knows Numpy
- All big matrices live transparently in S3

- User facing numpy/matlab-like interface: numpywren
- Low Level IR aimed at LA primitives: lambdapack
- Execution Framework: pywren
NUMPYWREN GOALS

- Usable by anyone who knows Numpy
- All big matrices live transparently in S3
- All intermediate state is retained
NEAREST NEIGHBOR
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = numpywren.default_executor()
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()
    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2

    norms_train = npwex.linalg.norm(X_train, axis=1)
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()
    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2
    norms_train = npwex.linalg.norm(X_train, axis=1)
    norms_test = npwex.linalg.norm(X_test, axis=1)
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()
    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)
    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2
    norms_train = npwex.linalg.norm(X_train, axis=1)
    norms_test = npwex.linalg.norm(X_test, axis=1)
    distances = norms_train + XYT + norms_test.T
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2

    norms_train = npwex.linalg.norm(X_train, axis=1)
    norms_test = npwex.linalg.norm(X_test, axis=1)
    distances = norms_train + XYT + norms_test.T
    argmins = npwex.argmin(distances, axis=0).numpy()
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2

    norms_train = npwex.linalg.norm(X_train, axis=1)
    norms_test = npwex.linalg.norm(X_test, axis=1)

    distances = norms_train + XYT + norms_test.T
    argmins = npwex.argmin(distances, axis=0).numpy()
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2
    norms_train = npwex.linalg.norm(X_train, axis=1)
    norms_test = npwex.linalg.norm(X_test, axis=1)
    distances = norms_train + XYT + norms_test.T
    argmins = npwex.argmin(distances, axis=0).numpy()

    return metrics.accuracy_score(y_train[argmins], y_test)
def nearest_neighbor_numpywren(X_train, X_test, y_train, y_test):
    npwex = npywren.default_executor()

    X_train_sharded = npwex.matrix_init(X_train)
    X_test_sharded = npwex.matrix_init(X_test)

    XYT = npwex.dot(X_train_sharded, X_test_sharded.T)
    XYT *= -2

    norms_train = npwex.linalg.norm(X_train, axis=1)
    norms_test = npwex.linalg.norm(X_test, axis=1)
    distances = norms_train + XYT + norms_test.T
    argmins = npwex.argmin(distances, axis=0).numpy()

    return metrics.accuracy_score(y_train[argmins], y_test)
SOLVING A LINEAR SYSTEM  \[ \mathbf{A} \mathbf{x} = \mathbf{B} \]
SOLVING A LINEAR SYSTEM \( Ax = B \)

1. Compute Cholesky Factorization \( LL^T = A \)
SOLVING A LINEAR SYSTEM $Ax = B$

1. Compute Cholesky Factorization $LL^T = A$
2. Forward substitution to solve $Lz = B$
SOLVING A LINEAR SYSTEM \( \mathbf{A} \mathbf{x} = \mathbf{B} \)

1. Compute Cholesky Factorization \( \mathbf{L} \mathbf{L}^T = \mathbf{A} \)
2. Forward substitution to solve \( \mathbf{L} \mathbf{z} = \mathbf{B} \)
3. Backward substitution \( \mathbf{L}^T \mathbf{x} = \mathbf{z} \)
SOLVING A LINEAR SYSTEM \[ Ax = B \]

1. Compute Cholesky Factorization \[ LL^T = A \] \( O(n^3) \)
2. Forward substitution to solve \[ Lz = B \]
3. Backward substitution \[ L^Tx = z \]
1. Compute Cholesky Factorization $LL^T=A$ $O(n^3)$
2. Forward substitution to solve $Lz = B$ $O(n^2)$
3. Backward substitution $L^Tx = z$
SOLVING A LINEAR SYSTEM \( Ax = B \)

1. Compute Cholesky Factorization \( LL^T = A \) \( O(n^3) \)
2. Forward substitution to solve \( Lz = B \) \( O(n^2) \)
3. Backward substitution \( L^T x = z \) \( O(n^2) \)
SOLVING A LINEAR SYSTEM \( Ax = B \)

1. Compute Cholesky Factorization \( LL^T = A \) \( O(n^3) \)
2. Forward substitution to solve \( Lz = B \) \( O(n^2) \)
3. Backward substitution \( L^T x = z \) \( O(n^2) \)

Figure 2: First 4 time steps of parallel Cholesky decomposition: 0) Diagonal block Cholesky decomposition 1) Parallel column update 2) Parallel submatrix update 3) (subsequent) Diagonal block Cholesky decomposition
cholesky

```python
cholesky(iter=0)
    0 = LOAD BigMatrix(X)[0, 0]
    1 = CHOL 0
    2 = WRITE chol(BigMatrix(X))[0, 0]

col_update(row=1, col=0)
    3 = LOAD chol(BigMatrix(X))[0, 0]
    4 = LOAD BigMatrix(X)[1, 0]
    5 = TRSM 3 4
    6 = WRITE chol(BigMatrix(X))[1, 0]

col_update(row=2, col=0)
    ...

low_rank_update(iter=0, row=1, col=2)
    15 = LOAD chol(BigMatrix(X))[0, 0]
    16 = LOAD chol(BigMatrix(X))[1, 0]
    17 = LOAD chol(BigMatrix(X))[2, 0]
    18 = SYRK 15 16 17
    19 = WRITE temp(BigMatrix(X))[0, 1, 2]

low_rank_update(iter=0, row=1, col=3)
    ...
```
cholesky(\text{iter}=0)
\begin{align*}
0 &= \text{LOAD BigMatrix}(X)\{0, 0\} \\
1 &= \text{CHOL} \ 0 \\
2 &= \text{WRITE} \ \text{chol}(\text{BigMatrix}(X))\{0, 0\}
\end{align*}
\text{col_update}(\text{row}=1, \text{col}=0)
\begin{align*}
3 &= \text{LOAD} \ \text{chol}(\text{BigMatrix}(X))\{0, 0\} \\
4 &= \text{LOAD} \ \text{BigMatrix}(X)\{1, 0\} \\
5 &= \text{TRSM} \ 3 \ 4 \\
6 &= \text{WRITE} \ \text{chol}(\text{BigMatrix}(X))\{1, 0\}
\end{align*}
\text{col_update}(\text{row}=2, \text{col}=0)
\text{...}
\text{low_rank_update}(\text{iter}=0, \text{row}=1, \text{col}=3)
\begin{align*}
15 &= \text{LOAD} \ \text{chol}(\text{BigMatrix}(X))\{0, 0\} \\
16 &= \text{LOAD} \ \text{chol}(\text{BigMatrix}(X))\{1, 0\} \\
17 &= \text{LOAD} \ \text{chol}(\text{BigMatrix}(X))\{2, 0\} \\
18 &= \text{SYRK} \ 15 \ 16 \ 17 \\
19 &= \text{WRITE} \ \text{temp}(\text{BigMatrix}(X))\{0, 1, 2\}
\end{align*}
\text{low_rank_update}(\text{iter}=0, \text{row}=1, \text{col}=3)
\text{...}
cholesky(iter=0)
  0 = LOAD BigMatrix(X)[0, 0]
  1 = CHOL 0
  2 = WRITE chol(BigMatrix(X))[0, 0]
col_update(row=1, col=0)
  3 = LOAD chol(BigMatrix(X))[0, 0]
  4 = LOAD BigMatrix(X)[1, 0]
  5 = TRSM 3 4
  6 = WRITE chol(BigMatrix(X))[1, 0]
col_update(row=2, col=0)
  7 = LOAD chol(BigMatrix(X))[0, 0]
  8 = LOAD chol(BigMatrix(X))[1, 0]
  9 = LOAD chol(BigMatrix(X))[2, 0]
low_rank_update(iter=0, row=1, col=2)
  10 = LOAD chol(BigMatrix(X))[0, 0]
  11 = LOAD chol(BigMatrix(X))[1, 0]
  12 = LOAD chol(BigMatrix(X))[2, 0]
low_rank_update(iter=0, row=1, col=3)
  13 = LOAD chol(BigMatrix(X))[0, 0]
  14 = LOAD chol(BigMatrix(X))[1, 0]
  15 = LOAD chol(BigMatrix(X))[2, 0]
low_rank_update(iter=0, row=1, col=3)
  16 = LOAD chol(BigMatrix(X))[0, 0]
  17 = LOAD chol(BigMatrix(X))[1, 0]
  18 = SYRK 15 16 17
  19 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  20 = LOAD chol(BigMatrix(X))[0, 0]
  21 = LOAD chol(BigMatrix(X))[1, 0]
  22 = LOAD chol(BigMatrix(X))[2, 0]
  23 = SYRK 15 16 17
  24 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  25 = LOAD chol(BigMatrix(X))[0, 0]
  26 = LOAD chol(BigMatrix(X))[1, 0]
  27 = LOAD chol(BigMatrix(X))[2, 0]
  28 = SYRK 15 16 17
  29 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  30 = LOAD chol(BigMatrix(X))[0, 0]
  31 = LOAD chol(BigMatrix(X))[1, 0]
  32 = LOAD chol(BigMatrix(X))[2, 0]
  33 = SYRK 15 16 17
  34 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  35 = LOAD chol(BigMatrix(X))[0, 0]
  36 = LOAD chol(BigMatrix(X))[1, 0]
  37 = LOAD chol(BigMatrix(X))[2, 0]
  38 = SYRK 15 16 17
  39 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  40 = LOAD chol(BigMatrix(X))[0, 0]
  41 = LOAD chol(BigMatrix(X))[1, 0]
  42 = LOAD chol(BigMatrix(X))[2, 0]
  43 = SYRK 15 16 17
  44 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  45 = LOAD chol(BigMatrix(X))[0, 0]
  46 = LOAD chol(BigMatrix(X))[1, 0]
  47 = LOAD chol(BigMatrix(X))[2, 0]
  48 = SYRK 15 16 17
  49 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  50 = LOAD chol(BigMatrix(X))[0, 0]
  51 = LOAD chol(BigMatrix(X))[1, 0]
  52 = LOAD chol(BigMatrix(X))[2, 0]
  53 = SYRK 15 16 17
  54 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  55 = LOAD chol(BigMatrix(X))[0, 0]
  56 = LOAD chol(BigMatrix(X))[1, 0]
  57 = LOAD chol(BigMatrix(X))[2, 0]
  58 = SYRK 15 16 17
  59 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  60 = LOAD chol(BigMatrix(X))[0, 0]
  61 = LOAD chol(BigMatrix(X))[1, 0]
  62 = LOAD chol(BigMatrix(X))[2, 0]
  63 = SYRK 15 16 17
  64 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  65 = LOAD chol(BigMatrix(X))[0, 0]
  66 = LOAD chol(BigMatrix(X))[1, 0]
  67 = LOAD chol(BigMatrix(X))[2, 0]
  68 = SYRK 15 16 17
  69 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  70 = LOAD chol(BigMatrix(X))[0, 0]
  71 = LOAD chol(BigMatrix(X))[1, 0]
  72 = LOAD chol(BigMatrix(X))[2, 0]
  73 = SYRK 15 16 17
  74 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  75 = LOAD chol(BigMatrix(X))[0, 0]
  76 = LOAD chol(BigMatrix(X))[1, 0]
  77 = LOAD chol(BigMatrix(X))[2, 0]
  78 = SYRK 15 16 17
  79 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  80 = LOAD chol(BigMatrix(X))[0, 0]
  81 = LOAD chol(BigMatrix(X))[1, 0]
  82 = LOAD chol(BigMatrix(X))[2, 0]
  83 = SYRK 15 16 17
  84 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  85 = LOAD chol(BigMatrix(X))[0, 0]
  86 = LOAD chol(BigMatrix(X))[1, 0]
  87 = LOAD chol(BigMatrix(X))[2, 0]
  88 = SYRK 15 16 17
  89 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  90 = LOAD chol(BigMatrix(X))[0, 0]
  91 = LOAD chol(BigMatrix(X))[1, 0]
  92 = LOAD chol(BigMatrix(X))[2, 0]
  93 = SYRK 15 16 17
  94 = WRITE temp(BigMatrix(X))[0, 1, 2]
low_rank_update(iter=0, row=1, col=3)
  95 = LOAD chol(BigMatrix(X))[0, 0]
  96 = LOAD chol(BigMatrix(X))[1, 0]
  97 = LOAD chol(BigMatrix(X))[2, 0]
  98 = SYRK 15 16 17
  99 = WRITE temp(BigMatrix(X))[0, 1, 2]
EXECUTION

Instruction Queue

Time

0
EXECUTION

- Local Queue (per core)
- Read Thread
- Compute Thread
- Write Thread

Time

Instruction Queue
EXECUTION

- Local Queue (per core)
- Read Thread
- Compute Thread
- Write Thread

Instruction Queue

- S3 Load
- Chol
- S3 Write
- S3 Load
- SYRK
- S3 Write
- S3 Load
- SYRK
- S3 Write
- S3 Load
- TRSM
- S3 Write

Time

0
EXECUTION

Local Queue (per core)  Read Thread  Compute Thread  Write Thread

S3Load  CHOL  S3Write

S3 Load  SYRK  S3 Write
S3 Load  SYRK  S3 Write
S3 Load  TRSM  S3 Write

Time

1

Instruction Queue
EXECUTION

Local Queue (per core)  Read Thread  Compute Thread  Write Thread

S3Write  S3Load  CHOL  S3Write
SYRK  S3Load  SYRK  S3Write
S3Load  SYRK  S3Load  SYRK
S3Write  S3Write  S3Write  S3Write

Time
3

Instruction Queue

S3 Load  TRSM  S3 Write
EXECUTION

Local Queue (per core)

- S3Load
- TRSM
- S3Write
- S3Write
- SYRK
- S3Write

Read Thread

- S3Load
- SYRK
- S3Write

Compute Thread

Write Thread

Full pipelining

Instruction Queue

Time

4
EXECUTION

Local Queue (per core)

Read Thread

Compute Thread

Write Thread

New Instructions Enqueued (based on task graph)

Time

Instruction Queue

5
PERFORMANCE

Efficiency

How efficiently did I use my resources

End to end runtime

How long did it take to get an answer
TOTAL CORE SECONDS USED

- Theoretical lower bound
- Numpywren
- Scalapack

CPU Time (million seconds)

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>128k</th>
<th>256k</th>
<th>512k</th>
<th>1m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numpywren</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalapack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NumPyWren

- Serverless linear algebra is possible, performant, elastic, and easy

Eric Jonas
jonas@eecs.berkeley.edu
@stochastician
NumPyWren

- Serverless linear algebra is possible, performant, elastic, and easy
- Releasing code this month

Eric Jonas
jonas@eecs.berkeley.edu
@stochastician

Vaishaal Shankar  Karl Krauth  Qifan Pu
Shivaram Venkataraman  Ion Stoica  Ben Recht  Jonathan Ragan-Kelly
NumPyWren

• Serverless linear algebra is possible, performant, elastic, and easy
• Releasing code this month
• Next steps: Op fusion, straggler mitigation, even higher-level interfaces

Eric Jonas
jonas@eecs.berkeley.edu
@stochastician
NumPyWren

- Serverless linear algebra is possible, performant, elastic, and easy
- Releasing code this month
- Next steps: Op fusion, straggler mitigation, even higher-level interfaces
- Questions?

Eric Jonas
jonas@eecs.berkeley.edu
@stochastician

Vaishaal Shankar  Karl Krauth  Qifan Pu  Shivaram Venkataraman  Ion Stoica  Ben Recht  Jonathan Ragan-Kelly
DISCUSSION SLIDE

• What additional services need to be truly elastic to make these sorts of applications possible?

• How much control do we want/need over queues, timing, latency, etc?

• What is the equilibrium price for serverless architectures?

• How can we expand this as a development platform for others