ArrayUDF: User-Defined Scientific Data Analysis on Arrays

Bin Dong\textsuperscript{1}, Kesheng Wu\textsuperscript{1}, Surendra Byna\textsuperscript{1}, Jialin Liu\textsuperscript{1}
Weijie Zhao\textsuperscript{2}, Florin Rusu\textsuperscript{2}

\textsuperscript{1}LBNL, Berkeley, CA
\textsuperscript{2}UC Merced, Merced, CA

Scientific Activities Evolved into Big Data Analysis

Example: scientific projects for dark matter/energy, supernovae, etc.

Data Size for Sky Survey Projects

Data source: Rick White, J. Hart, R. Cutri, Ian Foster, C. J. Grillmair, etc.
Q1: How many data analysis operations that have been and will be developed?
Q1: How many data analysis operations that have been and will be developed?

Large population

Implication from popular data analysis languages

<table>
<thead>
<tr>
<th></th>
<th>Total amount</th>
<th>Avg Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td>110,486</td>
<td>69/day</td>
</tr>
<tr>
<td>R packages</td>
<td>10,877</td>
<td>8/day</td>
</tr>
</tbody>
</table>

*Data from [http://www.modulecounts.com/](http://www.modulecounts.com/) on June 21 2017*
Q1: How many data analysis operations that have been and will be developed?

Large population

Q2: What are the functions of these data analysis operations?
Q1: How many data analysis operations that can be used to extract scientific meaning?

Large population

Q2: What are the functions of these data analysis operations?

Variety
Two Methods to Embrace Large Population & High variety in Data Analysis Operations

### Customized Solutions

**For each operation $P$ Do**

- Develop $P$'s:
  - Data management
  - Expression execution
  - X components: parallel, communication, cache, etc.

**End For**

### User-defined Functions (UDF)

**Operation expression 1**

- **UDF API**
  - Data management
  - Generic exec. engine
  - X components: parallel, comm., cache, etc.

#### Amateurishly tuned

- Data management
- Expression execution
- X components: parallel, communication, cache, etc.

#### Professionally tuned

- Data management
- Expression execution
- X components: parallel, communication, cache, etc.

<table>
<thead>
<tr>
<th>For each operation $P$ Do</th>
<th>Redundant</th>
<th>Diverse</th>
<th>Redundant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Develop $P$’s:</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>- Data management</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>- Expression execution</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- X components:</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>parallel, communication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cache, etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation expression 1</th>
<th>Redundant</th>
<th>✔</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UDF API</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>- Data management</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>- Generic exec. engine</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>- X components:</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>parallel, comm., cache,</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>
UDF is at Heart of Modern Big Data system

Examples: MapReduce in Apache Hadoop and Spark

Input Data

MAP()

MAP()

MAP()

shuffle/sort

UDF to generate (key, value) pairs

reduce()

reduce()

UDF to merge (key, values) pairs

Output Data
MapReduce Doesn’t Fit Scientific Data Analysis

Reason 1: Most Scientific Data are Multi-dimensional Arrays

Converting array to (key, value) is expensive

Pictures Credit: Kyle Hemes, Peter Nugent, Suren Byna, etc.
MapReduce Doesn’t Fit Scientific Data Analysis

Reason 1: Most Scientific Data are Multi-dimensional Arrays

Reason 2: Most Scientific Data Analysis Operations Own Structure Locality Property

Structure Locality:
The analysis operation on a single cell accesses its neighborhood cells

- Map deals with a single element at a time
- Reduce requires to duplicate an each cell for all neighborhood cells
- Reduce only happens after expensive shuffle

Pictures Credit: Kyle Hemes, Peter Nugent, Suren Byna, etc.
ArrayUDF: User-Defined Scientific Data Analysis on Arrays

- Stencil based User-defined Function
  - Structural locality aware array operations
- Native Multidimensional Array Data Model
  - In-situ data processing in scientific data formats, e.g., HDF5
- Optimal Chunking and Ghost Zone Method
  - Efficiently parallel array processing on HPC system
Stencil Definition

- Stencil S Definition for $d$-dimensional array

$$S = \{ c_{i_1+\delta_1, \ldots, i_d+\delta_d}, \quad |\delta| \in [L, R], \quad L \in [-i, 0], \quad R \in [0, N-i] \}$$

Materialized structure locality

Flexible neighborhood expression

- Centre of Stencil: $S_{0, \ldots, 0} = c_{i_1, \ldots, i_d}$
- Element at offsets $\delta_1, \ldots, \delta_d$:

$$S_{\delta_1, \ldots, \delta_d} = c_{i_1+\delta_1, \ldots, i_d+\delta_d}$$

2D Example:
Stencil based Computing Model

\[ C'_{i_1, \cdots, i_d} = f\left( \{ S_{\delta_1, \cdots, \delta_d} \mid \delta \in [L, R] \} \right) \]

- The cell in array \( A' \) at coordinate \( i_1 \cdots i_d \)
- The stencil set in array \( A \) at offset: \( \delta_1 \cdots \delta_d \)
- \( A = A' \) when in-situ updates
- \( \text{Dim}(A) = \text{Dim}(A') \) in most cases
<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>Output</th>
<th>UDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>Tuple $t$</td>
<td>Tuple $t'$</td>
<td>$t'=f(t)$</td>
</tr>
<tr>
<td>SciDB</td>
<td>Cell $c$</td>
<td>Cell $c'$</td>
<td>$c'=f(c)$</td>
</tr>
<tr>
<td>MapReduce</td>
<td>KeyValue $kv$</td>
<td>KeyValue $kv''$</td>
<td>$kv'=Map(kv)$ [kv''=Reduce(kv'_1, kv'_2, ...)]</td>
</tr>
<tr>
<td>ArrayUDF</td>
<td>Stencil $s$</td>
<td>Cell $c'$</td>
<td>$c'=f(s)$</td>
</tr>
</tbody>
</table>

vs. MapReduce: ArrayUDF generalizes its two steps as a single step on array

vs. SciDB: ArrayUDF supports structure-locality based computing on array
Examples of ArrayUDF

Example 1: Moving average

\[ V' = \frac{w_t V_{t-k} + \ldots + w_t V_t + \ldots + w_{t+m} V_{t+m}}{k + m + 1} \]

UDF_MV(Stencil s){
    return (s(-k)*w_{t-k} + \ldots s(0)*w_t + \ldots s(m)*w_{t+m})/(k+m-1)
}
V.Apply(UDF_MV)

Example 2: Vorticity computation

\[ \xi_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x} + \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta y} \]

UDF_VC_U(Stencil u){
    return u(0,1)- u(0, -1)
}
U.Apply(UDF_VC_1)

UDF_VC_V(Stencil v){
    return v(1,0)- u(-1, 0)
}
V.Apply(UDF_VC_V)

ArrayUDF’s C++ implementation
UDF Support System: Chunking

- Chunking enables parallel and out-of-cores processing
  - general chunking (layout unknown)
    - minimize ghost cells
  - layout-aware chunking (row-major)
    - maximize contiguous disk read

See theoretical proof in paper
UDF Support System: Ghost Zone handling

- ArrayUDF processes chunks in parallel and/or in out-of-core manner
- Ghost zone avoids communications between chunks
- Ghost zone size might be unknown in advance
  - UDF source code might be unavailable
- Trail-run is used to find ghost zone size:
  - Run UDF on a sample Stencil instance, that collects the offsets applied within UDF
Evaluations

• Hardware:
  - Edison, a Cray XC30 supercomputer at NERSC
  - 5576 computing nodes, 24 cores/node, 64GB DDR3 Memory

• Software
  - ArrayUDF 0.0.1
  - Spark 1.5.0
  - SciDB 16.9
  - RasDaMan 9.5 (sequential version)
  - EXTASCID, hand-optimized version
  - Hand-optimized C/C++ code

• Workloads
  - Two synthetic data sets (i.e., 2D and 3D array) for micro benchmarks
    ▪ Chunking strategy, trail-run, etc.
  - Four real scientific data sets (i.e., S3D, MSI, VPIC, CoRTAD)
    ▪ Overall performance tests /w generic UDF interface
Chunking Strategy Evaluation

- **general chunking** (for average cases)
  - minimize ghost cells # to reduce I/O cost
- **layout-aware chunking** (for layout special case)
  - maximize contiguous disk read
  - ignore the impact of ghost zone

2D Dataset (100000, 100000)

![Graph showing chunking strategy evaluation](image-url)
Comparison with peer systems with standard “window” operators

- “window” comes from SCiDB and RasDaMan
- “window” supports certain structure locality but lack the link to UDF function

Poisson equation solver \(w\)
Stencil S of ArrayUDF
2D : \[4S(0,0)−S(−1,0)−S(0,1)−S(1,0)−S(−1,0)\]
3D : \[6S(0,0,0)−S(−1,0,0)−S(0,1,0)−S(1,0,0)−S(−1,0,0)−S(0,0,−1)−S(0,0,1)\]

- ArrayUDF has close performance to hand-optimized code
- ArrayUDF is at least 13X faster than peer systems
Comparison with Spark in supporting real applications operations

Data Size

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Spark</th>
<th>ArrayUDF</th>
<th>Data Size</th>
<th>Spark</th>
<th>ArrayUDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3D Vorticity comp.</td>
<td>301GB</td>
<td>2 local cells/op.</td>
<td>CoRTAD Moving average</td>
<td>225GB</td>
<td>4 local cells/op.</td>
</tr>
<tr>
<td>MSI Laplacian op.</td>
<td>21GB</td>
<td>4 local cells/op.</td>
<td>VPIC Tri interpolation</td>
<td>36GB</td>
<td>8 local cells/op.</td>
</tr>
</tbody>
</table>

Spark’s Out-Of-Memory:
- large data size
- more local cells

ArrayUDF is as much as 2070X faster than Spark

# of local cells used by an UDF /w generic interface
Conclusions

• ArrayUDF: User-Defined Scientific Data Analysis on Arrays
  • Stencil based UDF for structural locality-aware operations
  • Native array model & In-situ array processing in HDF5, etc.
  • Optimal chunking and ghost zone methods for large array
• ArrayUDF provides close performance to hand-optimized code
• ArrayUDF is as much as 2070X faster than Spark
• ArrayUDF source code: https://bitbucket.org/arrayudf/
• Future work
  • Python and other language interface
  • Communication optimizations
Acknowledgments

• Nicholas Chaimov from University of Oregon for suggestions to set up Spark on Editon at NERSC
• Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, support for the SDS project (Program manager: Dr. Lucy Nowell) under contract number DE-AC02-05CH11231
• National Energy Research Scientific Computing Center
Thanks

Bin Dong
dbin@lbl.gov
http://crd.lbl.gov//dongbin
## Trail-run overhead

- Detect ghost zone size automatically
- Run the UDF on a single Stencil but the UDF might access more neighborhood cells

<table>
<thead>
<tr>
<th>Data sets</th>
<th>The number of cells used by UDF</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td>0.37</td>
<td>0.38</td>
<td>0.46</td>
<td>0.48</td>
<td>0.54</td>
<td>0.59</td>
<td>0.80</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td>0.48</td>
<td>0.52</td>
<td>0.65</td>
<td>0.75</td>
<td>0.79</td>
<td>0.84</td>
<td>1.04</td>
</tr>
</tbody>
</table>

Unit: microsecond

≈ 1 ms when 256 cells are used in the UDF