BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY

ArrayUDF: User-Defined Scientific Data Analysis on Arrays

Bin Dong[1], Kesheng Wu[1], Surendra Byna[1], Jialin Liu[1]

Weijie Zhao[2], Florin Rusu[1,2]

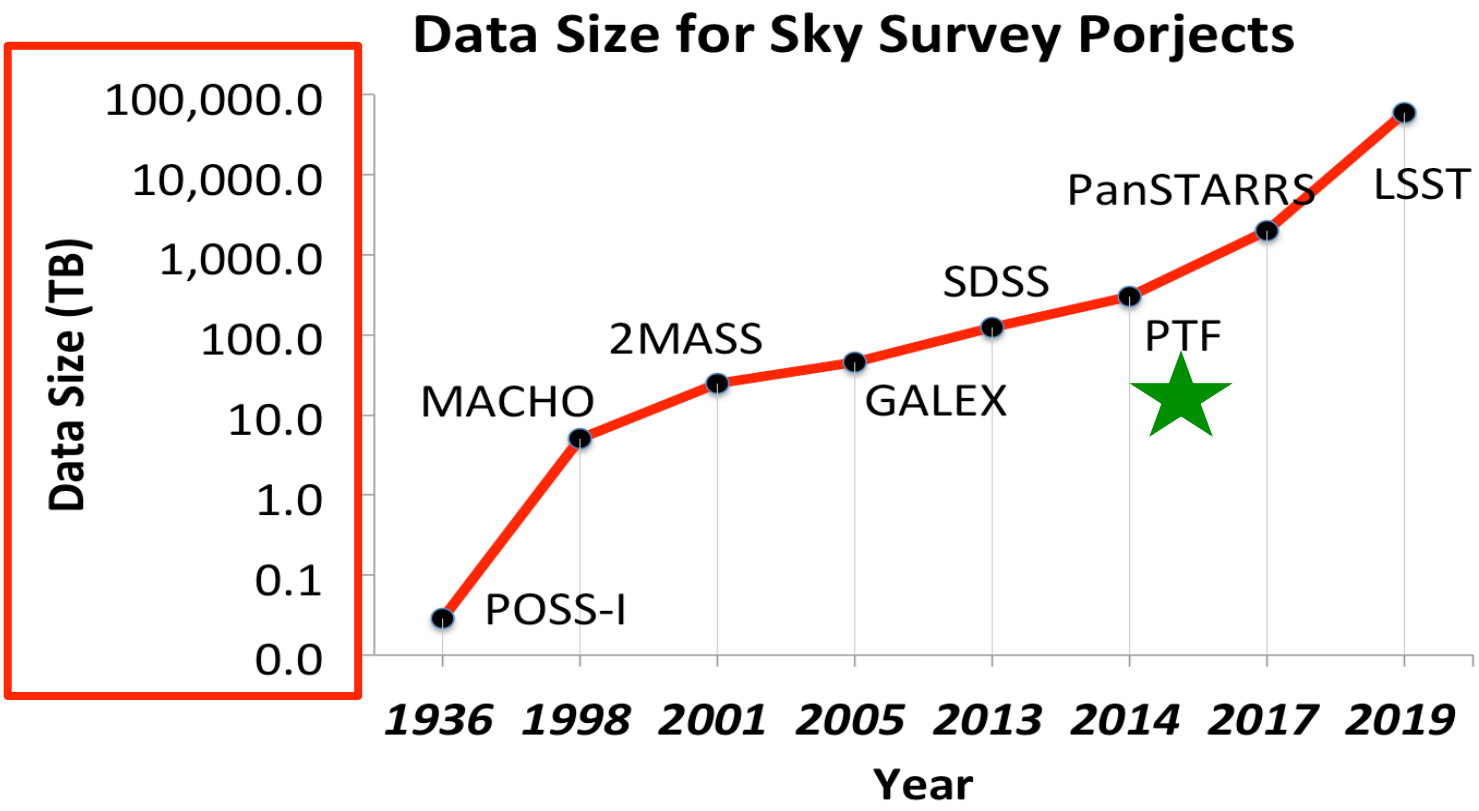
[1]LBNL, Berkeley, CA

[2]UC Merced, Merced, CA

HPDC 2017, Washington D.C. June 28, 2017

Scientific activities evolve into big data analysis

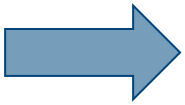
Example: scientific projects for supernovae, dark matter/energy, etc.



Data source:
Rick White,
J. Hart,
R. Cutri,
Ian Foster,
C. J. Grillmair,
etc.

Q1: How many data analysis operations are being or will be developed ?

Q1: How many data analysis operations are being or will be developed ?



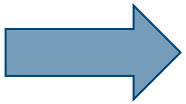
Large population

Implication from popular data analysis languages

	Avg Growth
Python modules	69/day
R packages	8/day
Java packages	102/day

*Data from http://www.modulecounts.com/ on June 21 2017

Q1: How many data analysis operations are being or will be developed ?



Large population

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

Q1: How many data analysis operations are being or will be developed ?

Large population

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

Variety

Two common methods to develop data analysis operations with large population and variety

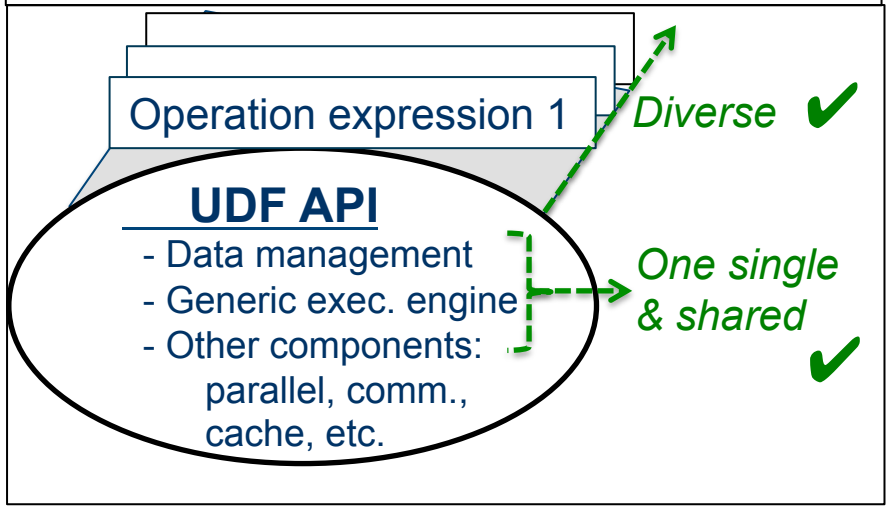
Customized Solutions

For each operation *P* **Do**
Develop *P*'s :
- Data management ---> *Redundant* ✗
- Expression execution ---> *Diverse* ✔
- Other components: ---> *Redundant* ✗
  parallel,
  communication
  cache,
  etc.

End For

May lack expertise of the underlying systems to tune its performance ✗

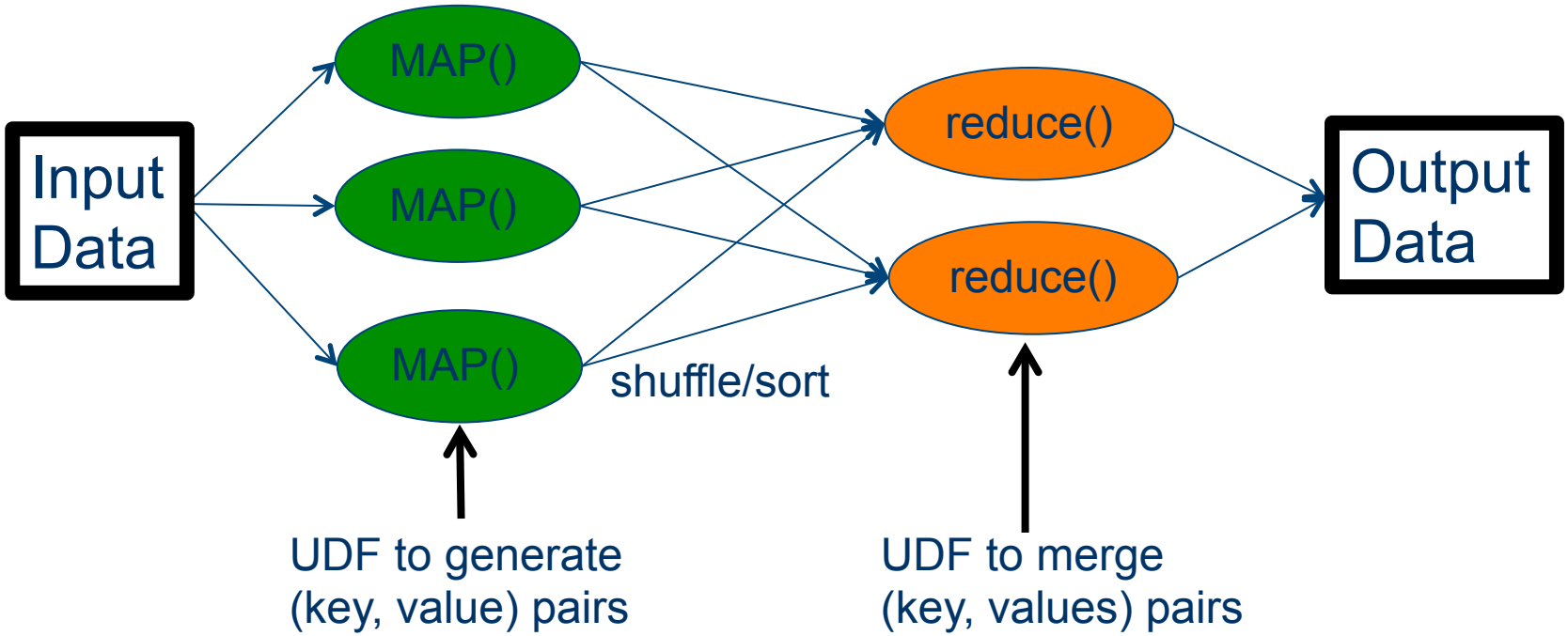
User-defined Functions (UDF)



Professionally tuned ✔

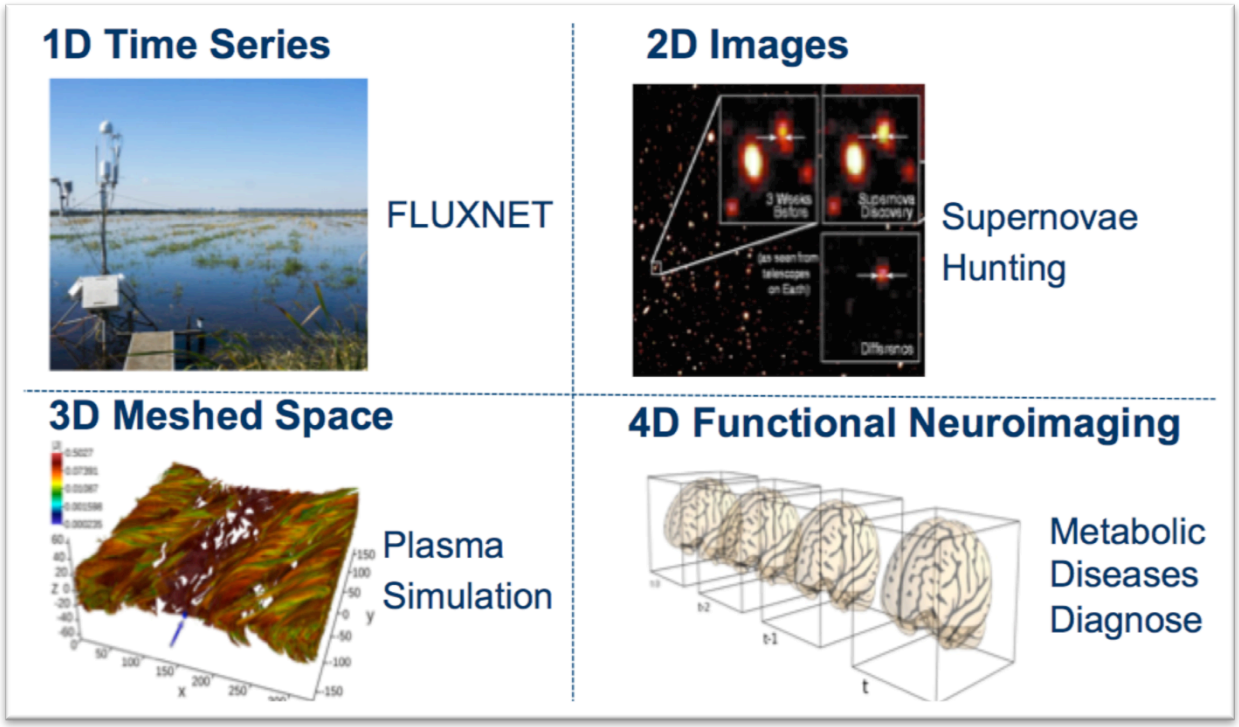
UDF is at heart of modern big data system

Examples: MapReduce in Apache Hadoop and Spark



MapReduce is not an optimal fit for scientific data analysis

Reason 1: most scientific data are multi-dimensional arrays



Pictures Credit: Kyle Hemes, Peter Nugent, Suren Byna, etc.

Converting array to (key, value) is expensive because of explicitly handling coordinate

MapReduce is not an optimal fit for scientific data analysis (continued)

Reason 1: most scientific data are multi-dimensional arrays

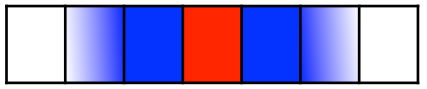
→ Converting array to (key, value) is expensive

Reason 2: most scientific data analysis operations own structure locality property

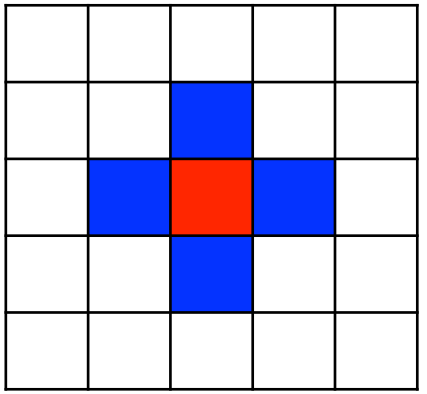
Structure locality:

The analysis operation on a single cell accesses its neighborhood cells

Moving Average



2D Poisson Equation Solver (Discrete)



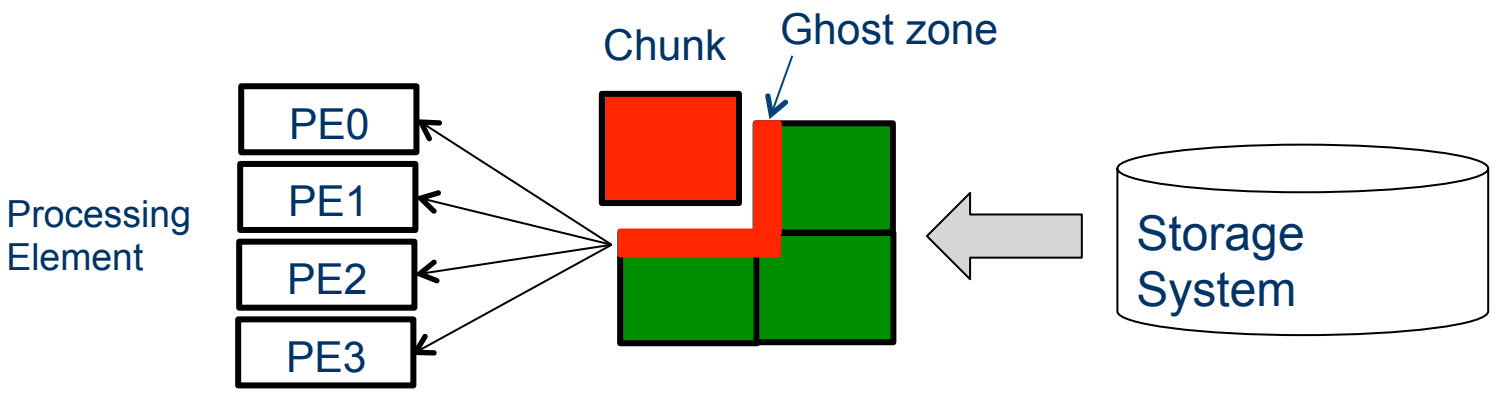
→ Map deals with a single element at a time

→ Reduce requires to duplicate each cell for all neighborhood cells

→ Reduce only happens after expensive shuffle

ArrayUDF: user-defined scientific data analysis on arrays

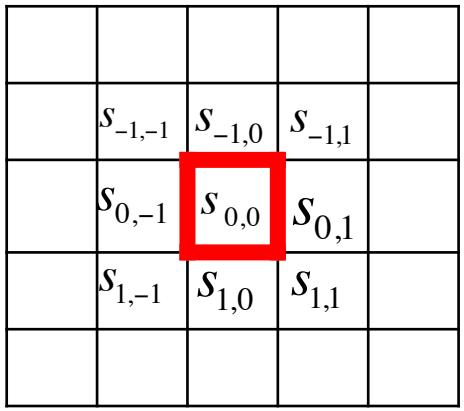
- Stencil-based user-defined function
  - ➡ Structural locality aware array operations
- Native multidimensional array data model
  - ➡ In-situ data processing in scientific data formats, e.g., HDF5
- Optimal and automatic chunking and ghost zone handling method
  - ➡ Fast large array processing in parallel & out-of-core manner



Stencil-based UDF

- Stencil is a set (*S*) of neighborhood cells
  - *The S has a* center where computing happens
  - The size of |*S*| is not fixed
  - Notations for set member
    $s_{\delta 1,\delta 1,\bullet\bullet\bullet}$ stands for the cell at *offset* $\delta_1,\delta_2,\bullet\bullet\bullet$ *from center point* $i,j,\bullet\bullet\bullet$

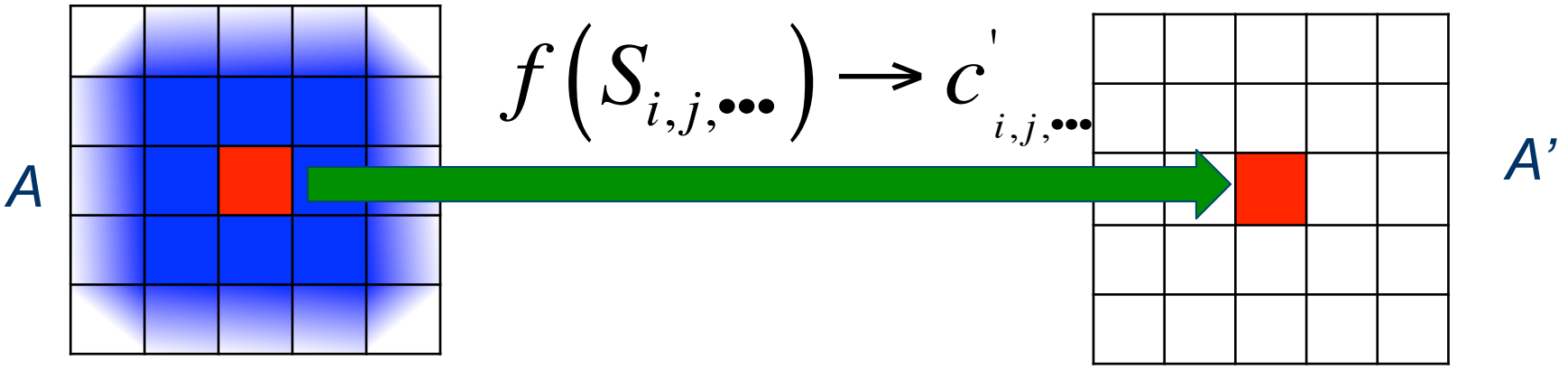
2D Example:



➡ Materialized structure locality

➡ Flexible UDF expression by manipulating each neighborhood cell independently

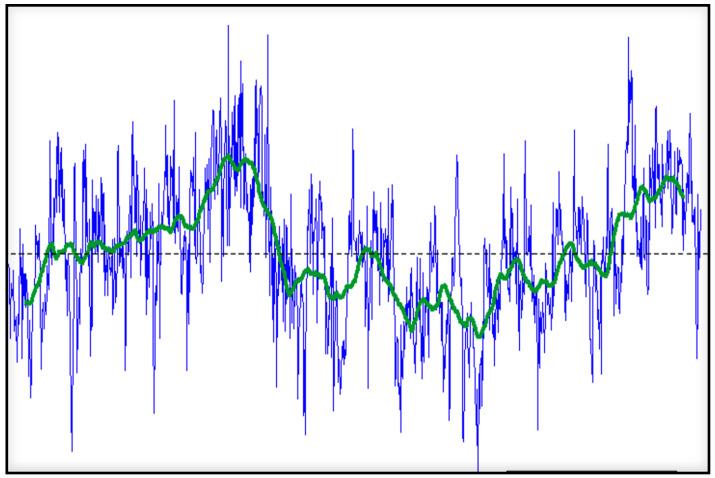
Stencil-based UDF(continued)



- f is arbitrary user-defined function
- *Input S* is Stencil representing set of neighborhood cells
  - |S| = 1, user-defined function of a single cell
    i.e., map in MapReduce
  - |S| > 1, user-defined aggregation of a set of cells,
    i.e., reduce in MapReduce

Examples of using ArrayUDF

Example 1: moving average in time series data



Global temperature trend filtered by moving average at 60 years' interval from 1908 to 2008

Three steps by using ArrayUDF:

Step 1: Initialize data

```
Array T("data location pointer")
```

Step 2: Define operation on Stencil

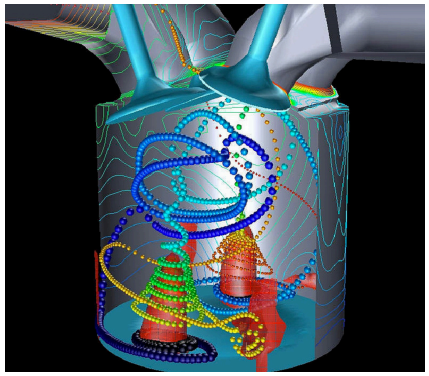
```
Tem_avg(Stencil t):
    return (t(-30)+ … t(30))/60
```

Step 3: Run & get result *T'*

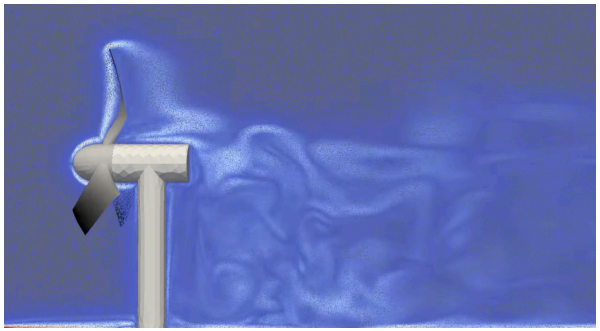
```
T.Apply(Tem_avg, T')
```

Examples of using ArrayUDF (Continued)

Example 2: vorticity computation in fluid flow



Combustion engines



Modeling renewable energy

Pictures credit to: LANL, Frank Fritz Michael Milthaler*, etc.*

Three steps by using ArrayUDF :

Step 1: Initialize data (2D example)

```
Array V_X("data location pointer")
Array V_y("data location pointer")
```

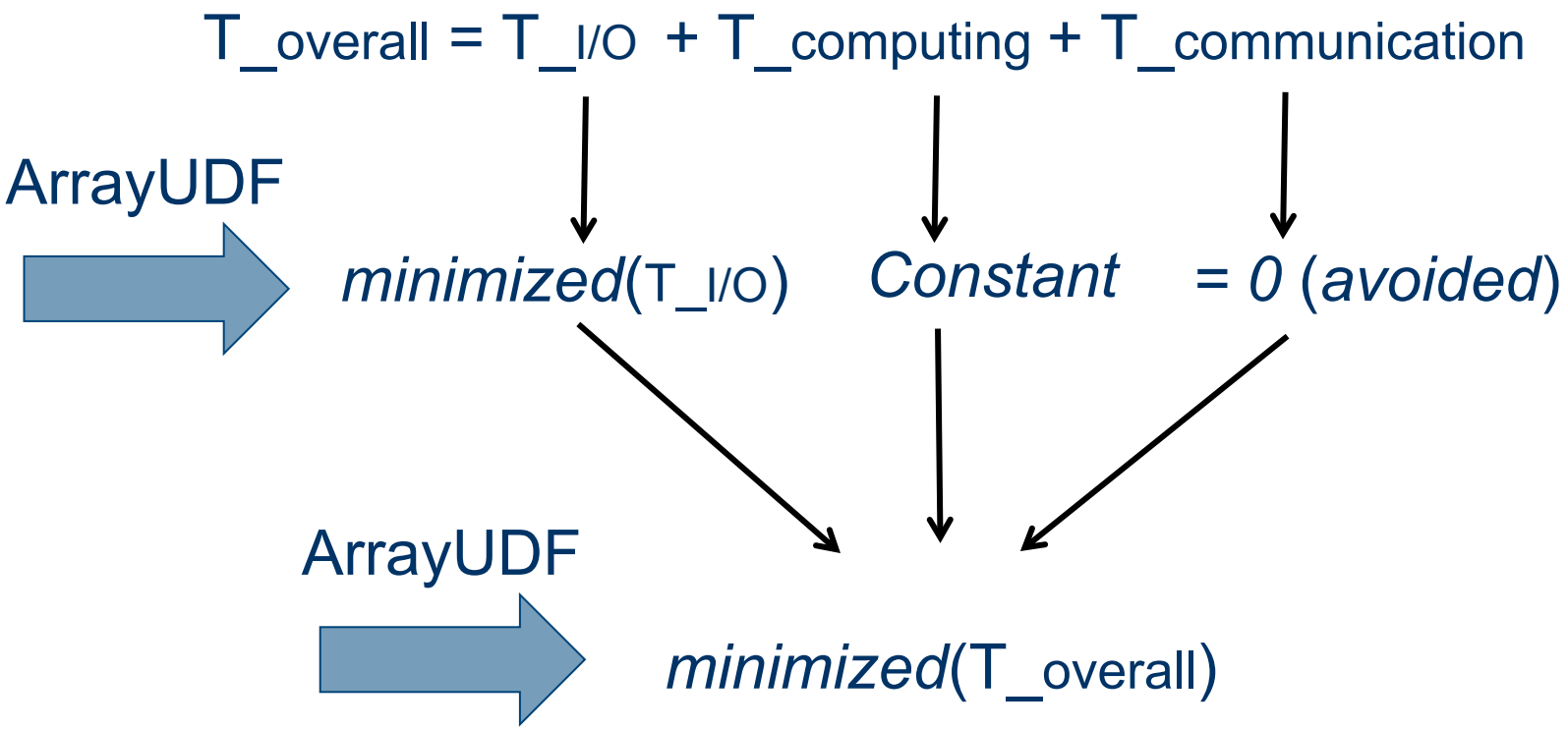
Step 2: Define operation on Stencil

```
VC_X(Stencil u):
   return u(0,1)- u(0, -1)
VC_Y(Stencil v):
   return v(1,0)- u(-1, 0)
```

Step 3: Run & get result

```
V_X.Apply(VC_X,  V_X')
V_Y.Apply(VC_Y , V_Y')
V_X'+V_Y' as vorticity
```

Optimized performance of ArrayUDF

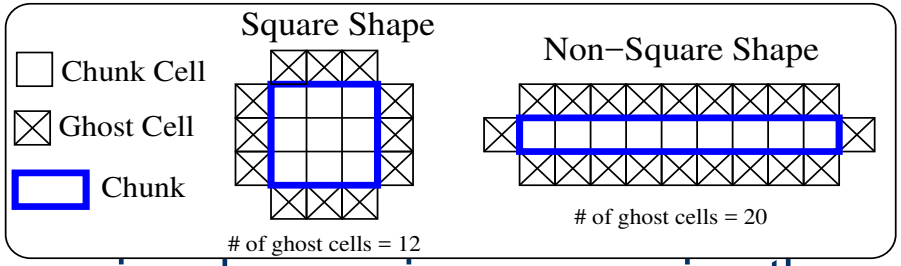


T_overall: overall time to run a data analysis operation
T_I/O: time of reading/writing data
T_computing: time of execute operation expression
T_communication: time of communications

ArrayUDF minimizes I/O cost via smart chunking

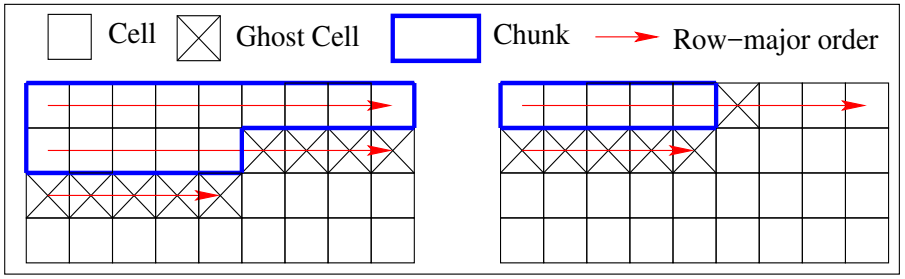
- Factors considered in chunking : physical layout, logical shape, size, etc.
- Two chunking strategies:
  - Layout unknown, i.e., average case of all possible layouts

⇒ Select **square shaped chunk** to minimize ghost cells/chunk



  - Layout known in advance, i.e., row-major, the most popular one

⇒ Select **contiguous chunking** to maximize I/O on contiguous cells, including ghost cells



See theoretical analysis in paper

ArrayUDF dynamically builds ghost zone to avoid communication

- What is ghost zone and why ?
  - Ghost zone are extra cells surrounding a chunk
  - Motivated by structure locality
- When to build ghost zone?
  - Ghost zone is built when chunk is read from disk into memory
- How to determine the size of ghost zone ?

⇨ Trail-run: execute the UDF code on a special Stencil instance to collect the offsets used by UDF

⇨ Size of ghost zone = maximum of collected offsets

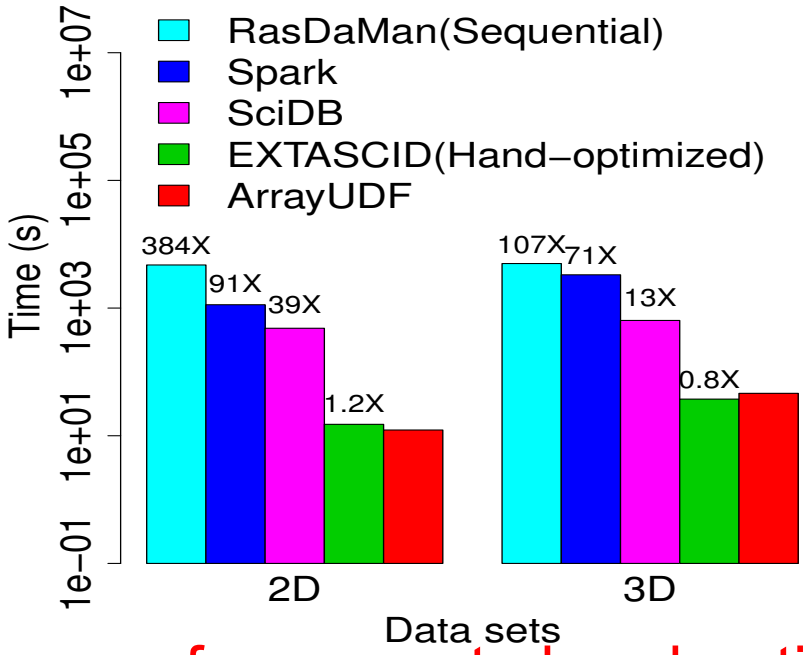
Evaluations

- Hardware:
  - Edison, a Cray XC30 supercomputer at NERSC
  - 5576 computing nodes, 24 cores/node, 64GB DDR3 Memory
- Software
  - ArrayUDF - RasDaMan 9.5 (sequential version)
  - Spark 1.5.0 - EXTASCID(hand-optimized version)
  - SciDB 16.9 - Hand-optimized C/C++ code
- Workloads
  - Two synthetic data sets (i.e., 2D and 3D) for micro benchmarks
    - Window operators, chunking strategy, trail-run, etc.
  - Four real scientific data sets (i.e., S3D, MSI , VPIC , CoRTAD)
    - Overall performance tests /w generic UDF interface

Comparison with peer systems with standard "window" operators

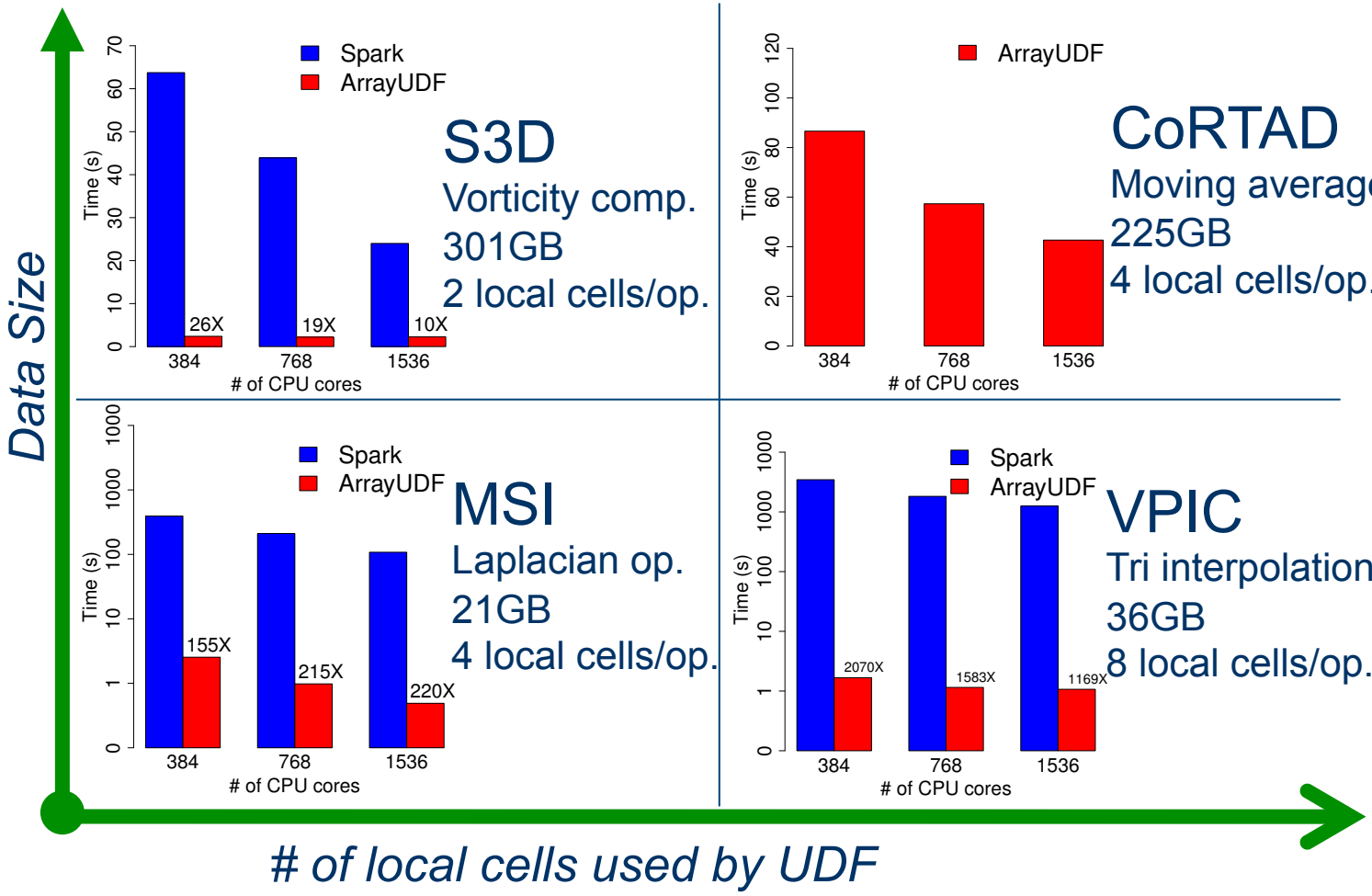
- "window" comes from SciDB and RasDaMan, where a operator is applied to all window members uniformly

Average on
- window 2x2 for 2D
- window 2x2x2 for 3D



- ArrayUDF has close performance to hand-optimized code
- ArrayUDF is as much as 384X faster than peer systems

Comparison with Spark in real scientific data analysis with generic UDF interface



Spark experiences out-of-memory:

- large data size

- more local cells

We observed ArrayUDF is 2070X faster

BERKELEY LAB

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.
  - Auto & Optimal chunking and ghost zone methods for parallel or out-of-core array processing
- ArrayUDF provides close performance to hand-optimized code
- ArrayUDF is as much as *2070X* faster than Spark
- ArrayUDF is easy-to-use data analysis system
- ArrayUDF source code: https://bitbucket.org/arrayudf/
- Future work
  - Python and other language interface (Done)
  - more in-situ formats: NetCDF, PnetCDF, ADIOS, etc.

Acknowledgments

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

U.S. DEPARTMENT OF ENERGY | Office of Science

- National Energy Research Scientific Computing Center

Thanks

Bin Dong
dbin@lbl.gov
http://crd.lbl.gov//dongbin

Backup Slides

Stencil-based computing model vs. others

	Input	Output	UDF
SQL DBMS	Tuple *t*	Tuple *t'*	$t'=f(t)$
SciDB	Cell *c*	Cell *c'*	$c'=f(c)$
MapReduce	KeyValue *kv*	KeyValue *kv"*	$kv'=Map(kv)$ $kv"=Reduce(kv'_1, kv'_{2, ..})$
ArrayUDF	Stencil *s*	Cell *c'*	$c'=f(s)$

vs. MapReduce:

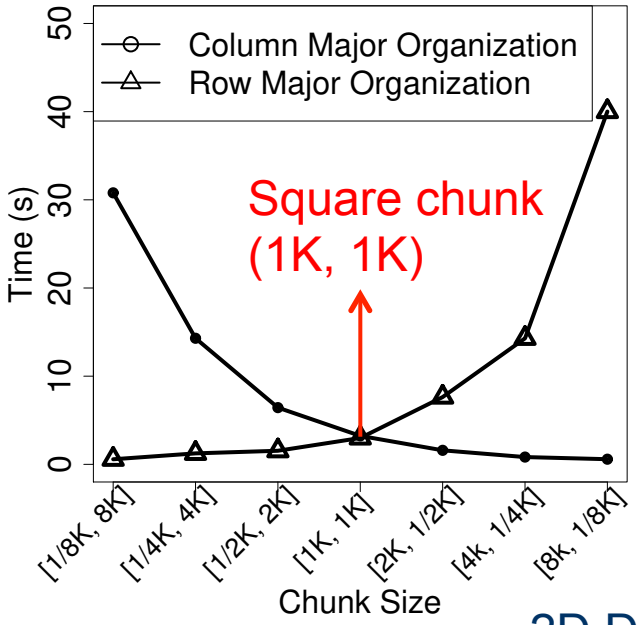
ArrayUDF generalizes map and reduce as a single operation

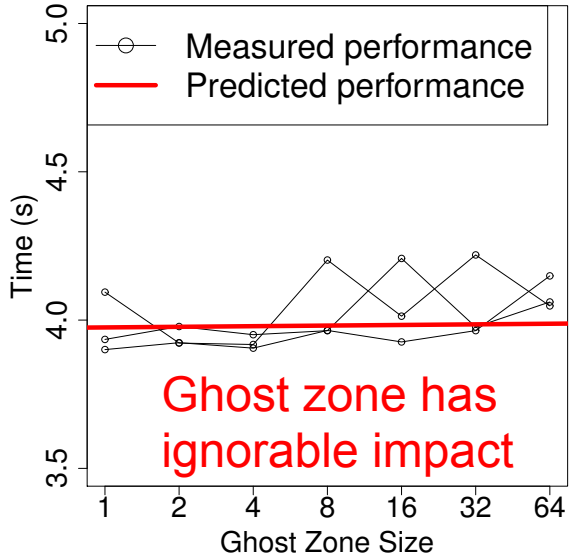
vs. SciDB:

SciDB has 'window', similar to Stencil. But, the 'window' usually applies an operator uniformly on all cells involved. In ArrayUDF, cell with stencil can be applied with different operator.

Chunking strategy evaluation

- Squared chunking (for average cases)
  - minimize ghost cells # to reduce I/O cost
- Contiguous chunking (for row-major layout)
  - maximize contiguous disk read





2D Dataset (100000, 100000)

Trail-run overhead

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

Data sets	The number of cells used by UDF						
	4	8	16	32	64	128	256
S1	0.37	0.38	0.46	0.48	0.54	0.59	0.80
S2	0.48	0.52	0.65	0.75	0.79	0.84	1.04

Unit: microsecond

≈ 1 ms when 256 cells are used in the UDF