Outline

Motivation

Brief Introduction to J

Data Parallelism Opportunities

Truffle-J Interpreter

Performance Results

Introduction

- Writing sequential programs is hard
- Writing explicitly parallel programs is harder
- Instead use implicit parallelism
  - Legacy code also parallelized without rewrite

Motivation for an Array Programming Model

- Allows programmers to operate on aggregates of data
- Exposes lots of opportunities for data parallelism
  - Language constructs can expose control parallelism
- Enables extraction of available parallelism *implicitly*

Our solution

- An Abstract Syntax Tree interpreter for J [1]
- Truffle is an easy framework to implement dynamic languages
  - Previously language implementations focused on single-threaded performance
- Extract implicit parallelism via an AST interpreter
  - Focus on multi-threaded parallelism
  - Interpreter based on the Truffle API
  - AST specialized dynamically during execution

[1] http://www.jsoftware.com/

Contributions

- Identification of parallel opportunities during interpretation
  - Rank Agreement
  - Vector operations
  - Reductions on associative operators
  - Control constructs
- Implicitly parallelizing interpreter for J
  - Written entirely in Java
- Performance evaluation of interpreter
  - J programs written without parallelization in mind

Outline

Motivation

Brief Introduction to J

Data Parallelism Opportunities

Truffle-J Interpreter

Performance Results

Introducing J

- Dynamically typed
- Right-to-left evaluation
- Functional in nature
- Language constructs say what to do, not how to do it
- Terseness personified
  - Unlike anything I had seen before ☺

- Interested in getting started
  - J for C Programmers (http://www.jsoftware.com/help/jforc/contents.htm)

J Vocabulary

Nouns	
Scalars are 0-dimensional arrays	`5       NB. A scalar, 0-D array`
N-dimensional arrays	`0 1 2 3 NB. 1-D array`
Verbs	
(Noun ➔ Noun)	`i.      NB. Create array`
(Noun ✕ Noun ➔ Noun)	`+       NB. Binary addition`
Adverbs	
(Noun ➔ Verb)	`3}      NB. Extract fourth element`
(Verb ➔ Verb)	`+/      NB. Sum reduce`
Conjunctions	
(Noun ✕ Verb ➔ Verb)	`2&+     NB. Add two to argument`
(Verb ✕ Noun ➔ Verb)	`^&3     NB. Cube argument`
(Verb ✕ Verb ➔ Verb)	`- @: *  NB. Multiply then negate`
…	

Simple Example – Sum Reduce

```
plus =: +                          NB. A verb
insert =: /                        NB. An adverb
sumReduce =: plus insert           NB. A verb

a =: i. 100                        NB. 0 1 2 3 ... 99
sumReduce a                        NB. 4950
```

```
NB. Tacit version: +/ i. 100
```

Simple Example – Matrix Multiplication

```
plus =: +
times =: *
insert =: /
sumReduce =: plus insert
matrixProduct =: sumReduce . times
NB. '.' is a conjunction
a =: i. 2 3
b =: i. 3 4
a matrixProduct b
```

0	1	2
3	4	5

matrixProduct

0	1	2	3
4	5	6	7
8	9	10	11

➔

20	23	26	29
56	68	80	92

```
NB. Tacit version: a +/.* b
```

Example – Counting Example

Given a range between a and b, compare the number of values that are and are not divisible by c and return the greater of them available to the user.

```
divisionCounter =: dyad define
    NB. Compute the remainders, compare to zero, then
    NB. count the exact divisions and the inexact
    NB. divisions, return the larger of those counts
    (+/ >. (+/ @: -.)) 0 = x | y
)
range =: dyad define
    x + i. 1 + y - x
)
c divisionCounter a range b
```

```
NB. Tacit version:
NB. c ([: (+/ >. +/@:-.) 0 = |) a ([ + [: i. 1 + -~) b
```

Outline

Motivation

Brief Introduction to J

Data Parallelism Opportunities

Truffle-J Interpreter

Performance Results

Parallel Opportunity - Rank Agreement

- Verbs in J have ranks
  - Specify the type of operands
  - e.g. dyadic + has a rank of 0 for both operands
- Ranks are used for implicit looping
- Ranks of functions can be controlled by the rank conjunction (")

Rank Agreement – monadic example

```
sumReduce =: plus insert
```

```
a =: i. 2 3
```

0	1	2
3	4	5

```
sumReduce a
```

0	1	2	**plus**	3	4	5

3	5	7

```
NB. Tacit version: +/ a
```

Rank Agreement - monadic example

```
sumReduce =: plus insert

a =: i. 2 3
```

0	1	2
3	4	5

```
byRows =: "1     NB. Adverb to operate by rows
(sumReduce byRows) a
```

sumReduce	0	1	2
sumReduce	3	4	5

3
12

Perform the row computations in parallel!

Rank Agreement - dyadic example

```
(i. 2 3) (plus byRows) (i. 3)
```

0	1	2
3	4	5

plus

0	1	2

0	1	2

plus

0	1	2

3	4	5

plus

0	1	2

0	2	4

3	5	7

Perform the computations in parallel!

Merge the individual fragments at the end

Parallel Opportunity – Vector Ops

- *Scalar verbs* on non-scalar data
- Element-wise operations on corresponding elements
- Perform the operation in parallel on the partitions
- Shape of the result is always the same as the input

e.g. `(i. 4 4)` ***plus*** `(2 times i. 4 4)`

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

plus

0	2	4	6
8	10	12	14
16	18	20	22
24	26	28	30

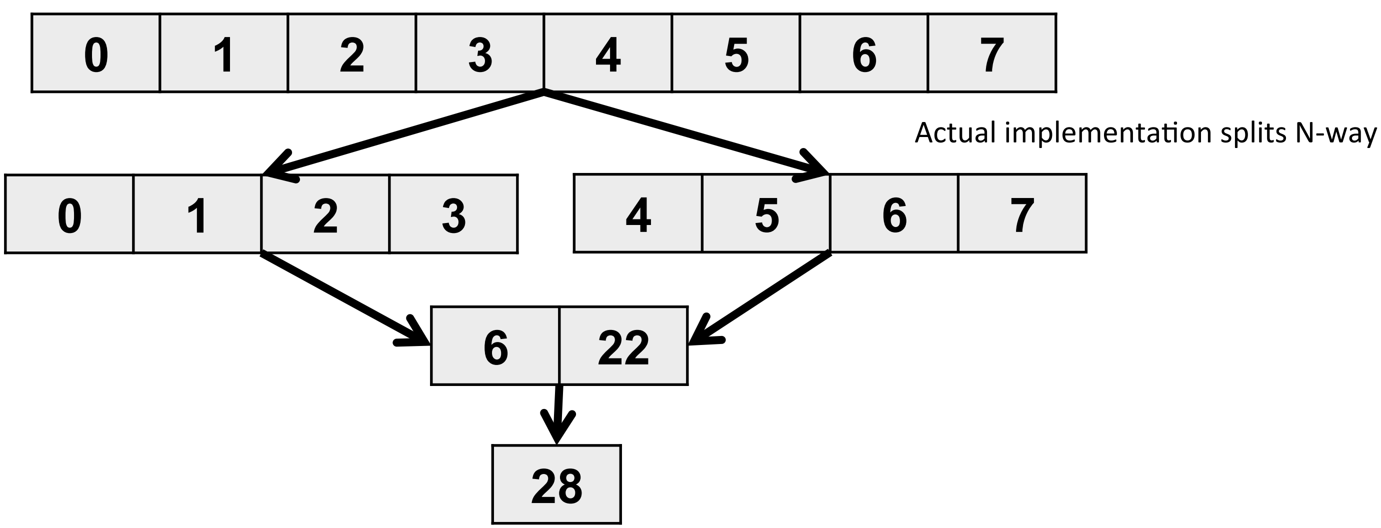
➔

0	3	6	9
12	15	18	21
24	27	30	33
34	39	42	45

Parallel Opportunity - Reductions

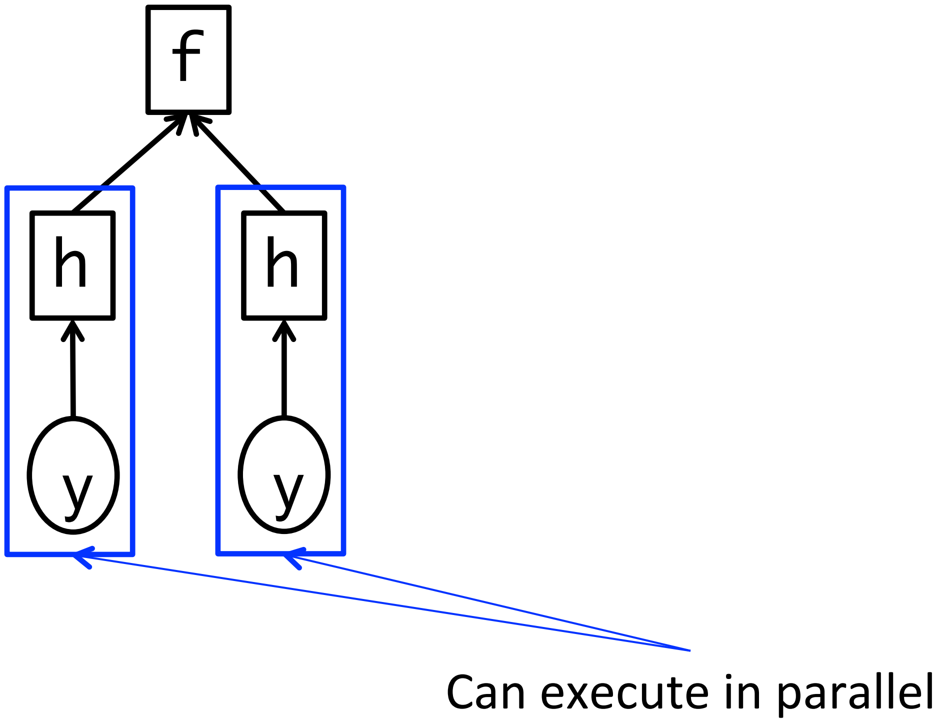
- Associative operation on the *insert adverb*
  - Operator not required to be commutative
  - Right-to-left evaluation order preserved
- Simple Fork-Join approach
  - Recursive reduction

e.g. `plus insert (i. 8)` `NB. Works with sumReduce also`



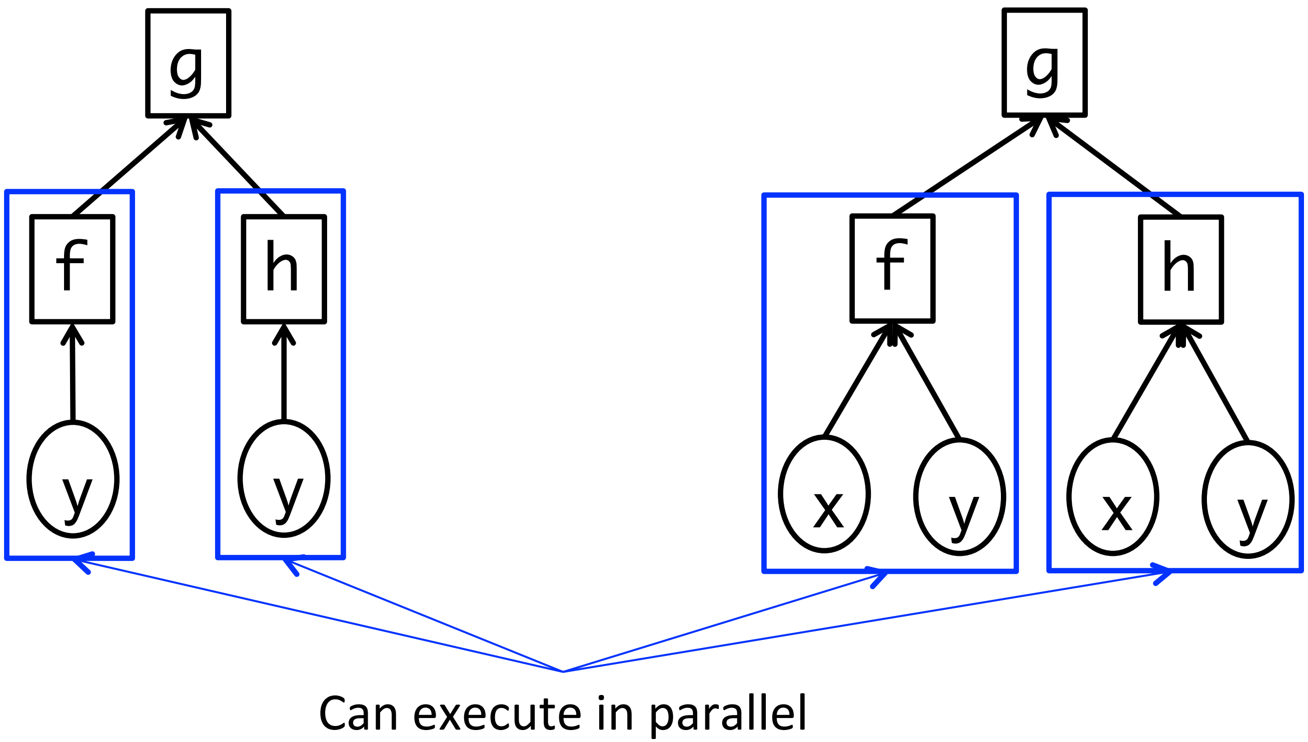
Other Parallel Opportunities – compose conjunction

```
x (f & h) y  <-->  (h x) f (h y)
```



Other Parallel Opportunities – monadic and dyadic fork construct

```
(f g h) y <--> (f y) g (h y)
x (f g h) y <--> (x f y) g (x h y)
```



Outline

Motivation

Brief Introduction to J

Data Parallelism Opportunities

Truffle-J Interpreter

Performance Results

Current Status – Implementation

- Pure Java Implementation
  - On Truffle framework
  - ~23 Kloc source (sloccount)
  - ~39 Kloc test code (sloccount)
- Most of J vocabulary supported, except
  - Data types other than int and double
  - Boxed data
  - Control words (e.g. if-else, explicit for, …)
- Source code to be open-sourced soon
  - https://java.net/projects/truffle-j

Truffle Node Specializations

- ASTs are merged to mimic inlining
- Macro expansions
  - Hooks, forks, and verb trains
  - adverb and conjunction applications
- All node rewrites and specializations happen dynamically
- Type specializations for scalars and arrays

Other Optimizations

- Rank agreement specializations on verb applications
  - Rank agreement logic bypassed for simple cases
- Function (AST) inlining
- Operand promotion
- Rank-0 verbs
- Verb fusion
- Minimize temporary Array creation
- Nested fork-join parallelism

Status – verbs/adverbs/conjunctions

Appendix E. Parts of Speech

Verbs

= Self-Classify • Equal
<. Floor • Lesser Of (Min)
> Open • Larger Than
>: Increment • Larger Or Equal
+ Conjugate • Plus
+: Double • Not-Or
*. Length/Angle • LCM (And)
- Negate • Minus
-: Halve • Match
%. Matrix Inverse • Matrix Divide
^ Exponential • Power
$ Shape Of • Shape
$: Self-Reference
~: Nub Sieve • Not-Equal
|. Reverse • Rotate (Shift)
, Ravel • Append
,: Itemize • Laminate
;: Words • Sequential Machine
#. Base 2 • Base
! Factorial • Out Of
\: Grade Down • Sort
[: Cap
{ Catalogue • From
{: Tail •
}. Behead • Drop
". Do • Numbers
? Roll • Deal
A. Anagram Index • Anagram
e. Raze In • Member (In)
i. Integers • Index Of
I. Indices • Interval Index
L. Level Of
p. Roots • Polynomial
p: Primes
r. Angle • Polar
u: Unicode
_9: to 9: Constant Functions

< Box • Less Than
<: Decrement • Less Or Equal
>. Ceiling • Larger of (Max)
_: Infinity
+. Real / Imaginary • GCD (Or)
* Signum • Times
*: Square • Not-And
-. Not • Less
% Reciprocal • Divide
%: Square Root • Root
^. Natural Log • Logarithm
$. Sparse
~. Nub •
| Magnitude • Residue
|: Transpose
,. Ravel Items • Stitch
; Raze • Link
Tally • Copy
#: Antibase 2 • Antibase
/: Grade Up • Sort
[Same • Left
] Same • Right
{. Head • Take
{:: Map • Fetch
}: Curtail •
": Default Format • Format
?. Roll • Deal (fixed seed)
C. Cycle-Direct • Permute
E. • Member of Interval
i: Steps • Index Of Last
j. Imaginary • Complex
o. Pi Times • Circle Function
p.. Poly. Deriv. • Poly. Integral
q: Prime Factors • Prime Exponents
s: Symbol
x: Extended Precision

Adverbs

~ *Reflex* • *Passive* / *Evoke*
/ *Insert* • *Table*
/. *Oblique* • *Key*
\ *Prefix* • *Infix*
\. *Suffix* • *Outfix*
} *Item Amend* • *Amend* (m} u})
b. *Boolean* / *Basic*
f. *Fix*
M. *Memo*
t. *Taylor Coeff.* (m t. u t.)
t: *Weighted Taylor*

Others

=. Is (Local)
=: Is (Global)
_ Negative Sign / *Infinity*
_. *Indeterminate*
a. *Alphabet*
a: *Ace*
NB. Comment

Conjunctions

^: **Power** (u^:n u^:v)
. **Determinant • Dot Product**
.. **Even**
.: **Odd**
: **Explicit / Monad-Dyad**
:. **Obverse**
:: **Adverse**
;. **Cut**
!. **Fit (Customize)**
!: **Foreign**
" **Rank** (m"n u"n m"v u"v)
` **Tie (Gerund)**
`: **Evoke Gerund**
@ **Atop**
@. **Agenda**
@: **At**
& **Bond / Compose**
&. **Under (Dual)**
&.: **Under (Dual)**
&: **Appose**
d. **Derivative**
D. **Derivative**
D: **Secant Slope**
H. **Hypergeometric**
L: **Level At**
S: **Spread**
T. **Taylor Approximation**

- Verbs
  - 73 out of 132
- Adverbs
  - 4 out of 18
- Conjunctions
  - 9 out of 30
- User-defined verbs supported
  - User-defined adverbs and conjunctions not yet supported

Status – nouns (N-dimensional arrays)

- Wraps a one-dimensional Java array
  - Single implementation called a StructA
- Wrapped arrays are immutable once initialized
- Subarrays are shared when items are created
  - No copying overhead for item creation
  - No copying overhead during shape promotion
- Subarrays can be targeted when merging result frames

Status – Parallel Runtime

- Based on the java.util.concurrent Executor framework
  - ThreadPool executor used
- All parallelism is from the fork-join pattern
- No work-stealing required
  - Handles nested data parallelism
  - Runtime carefully manages parallel task creation
  - Parallel tasks created when workers are available
- All data parallel opportunities mentioned earlier are exploited
- Main thread does more work than worker threads to minimize time spent waiting at the join point

Outline

Motivation

Brief Introduction to J

Data Parallelism Opportunities

Truffle-J Interpreter

Performance Results

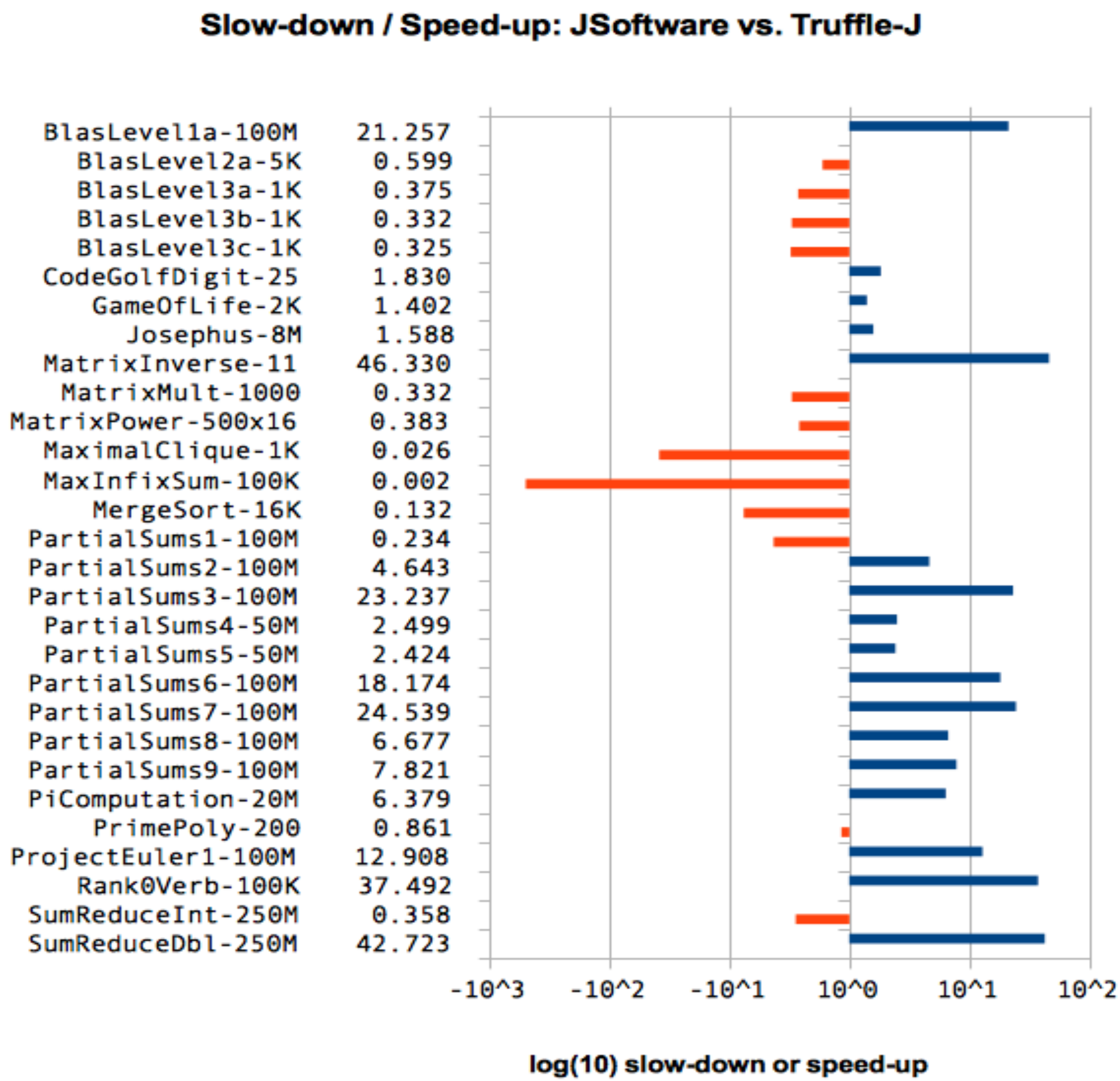
Experimental Results: Methodology

- Sequential Performance
  - Compared against JSoftware interpreter (version J801)
  - 4-core Intel Core i7 2.4 GHz system
  - 8 GB memory, 32 kB L1 cache, a 256 kB L2 cache
  - Java Hotspot JDK 1.7.0_17
- Parallel Performance
  - SPARC T5-8 Server
  - 8 processors at 3.6GHz x 16 cores x 8 threads = 1024 threads
  - 4TB of memory, a 16KB L1 data cache, a 128KB L2 cache
  - Java Hotspot JDK 1.7
  - Benchmarks run on 1, 2, 4, 8, 16, 32, 64, and 128 worker threads

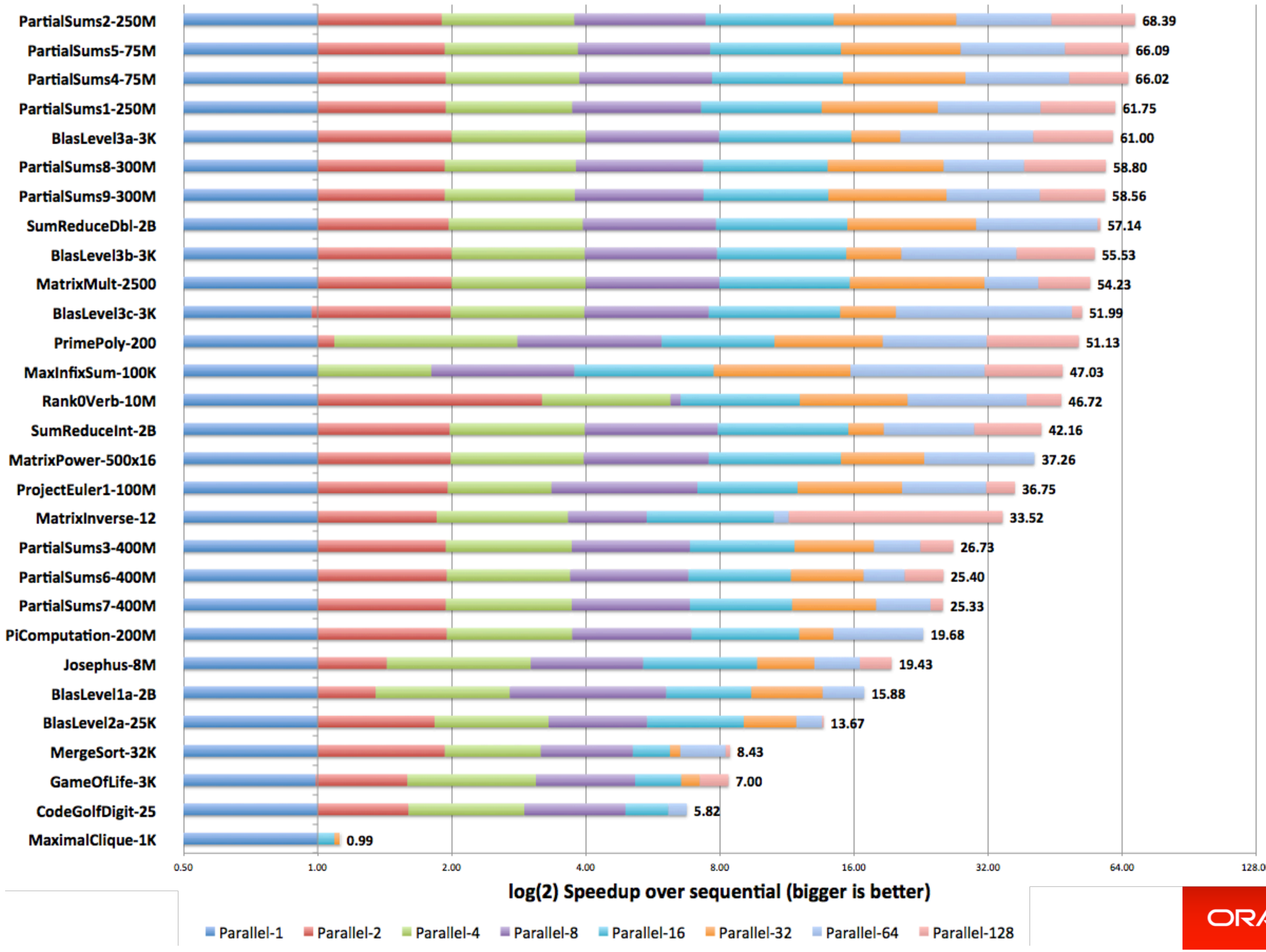
Experimental Results: Benchmarks

Name	Source, Computational Feature
BlasLevel1a	Ourselves, Linear Algebra
BlasLevel2a	Ourselves, Linear Algebra
BlasLevel3a	Ourselves, Linear Algebra
BlasLevel3b	Ourselves, Linear Algebra
BlasLevel3c	Ourselves, Linear Algebra
CodeGolfDigit	Ourselves, Scalar arithmetic
GameOfLife	C. Jenkins, Stencil Computation
Josephus	JSoftware, Scalar arithmetic
MatrixInverse	JSoftware, Linear Algebra
MatrixMult	JSoftware, Linear Algebra
MatrixPower	JSoftware, Linear Algebra
MaximalClique	JSoftware, Graph Algorithm
MaxInfixSum	JSoftware, J adverbs
MergeSort	C. Jenkins, Array indexing
PartialSums1	JSoftware, Arithmetic Series Sum
PartialSums2	JSoftware, Geometric Series Sum
PartialSums3	JSoftware, Inverse quadratic series
PartialSums4	JSoftware, Flint Hills series
PartialSums5	JSoftware, Cookson Hills series
PartialSums6	JSoftware, Harmonic series
PartialSums7	JSoftware, Riemann Zeta series
PartialSums8	JSoftware, Alternating series
PartialSums9	JSoftware, Gregory series
PiComputation	C. Jenkins, Pi Series Sum
PrimePoly	JSoftware, Scalar arithmetic
ProjectEuler1	JSoftware, Scalar arithmetic
Rank0Verb	Ourselves, Scalar Arithmetic
SumReduceInt	Ourselves, Series Sum (int)
SumReduceDbl	Ourselves, Series Sum (double)

Experimental Results: Sequential Perf.



Experimental Results: Parallel Perf.



Summary

- Implicitly exploit parallelism
  - Language constructs say **what** to do, not **how** to do it
  - Using a parallelizing interpreter
- Array language implementation on Truffle
  - Sequential interpreter gives good performance
- Array framework available for use in other projects
  - Includes support for binary and unary array operations
  - Includes the parallel runtime
  - Exploits nested parallelism
- Good parallel performance on benchmarks
  - Benchmarks were not written with parallelism in mind

Summary

- Implicitly exploit parallelism
  - Language constructs say **what** to do, not **how** to do it
  - Using a parallelizing interpreter
- Array language implementation on Truffle
  - Sequential interpreter gives good performance

```
import array.audience.questions.*;
```

- Array framework available for use in other projects
  - Includes support for binary and unary array operations
  - Includes the parallel runtime
  - Exploits nested parallelism
- Good parallel performance on benchmarks
  - Benchmarks were not written with parallelism in mind

BACKUP SLIDES