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

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

N-dimensional arrays

Verbs

(Noun → Noun)

(Noun × Noun → Noun)

Adverbs

(Noun → Verb)

(Verb → Verb)

Conjunctions

(Noun × Verb → Verb)

(Verb × Noun → Verb)

(Verb × Verb → Verb)

...

5 NB. A scalar, 0-D array

0 1 2 3 NB. 1-D array

i. NB. Create array

+ NB. Binary addition

3} NB. Extract fourth element

+/ NB. Sum reduce

2&+ NB. Add two to argument

^&3 NB. Cube argument

- @: * NB. Multiply then negate
Simple Example – Sum Reduce

\[
\begin{align*}
\text{plus} &=: + & \text{NB. A verb} \\
\text{insert} &=: / & \text{NB. An adverb} \\
\text{sumReduce} &=: \text{plus insert} & \text{NB. A verb} \\
\text{a} &=: i. 100 & \text{NB. 0 1 2 3 … 99} \\
\text{sumReduce a} & & \text{NB. 4950}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>0 1 2</th>
<th>1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 6</td>
<td>5 6 7</td>
</tr>
<tr>
<td>8 9 10</td>
<td>8 9 11</td>
</tr>
</tbody>
</table>

⇒

<table>
<thead>
<tr>
<th>20 23 26 29</th>
</tr>
</thead>
<tbody>
<tr>
<td>56 68 80 92</td>
</tr>
</tbody>
</table>

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

\[
\text{sumReduce} = \text{plus insert}
\]
\[
a = \text{i. 2 3}
\]
\[
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 4 & 5 \\
\end{array}
\]
\[
\text{sumReduce } a
\]
\[
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 4 & 5 \\
\end{array}
\]
\[
\text{plus}
\]
\[
\begin{array}{ccc}
3 & 5 & 7 \\
\end{array}
\]

NB. Tacit version: +/- a
Rank Agreement - monadic example

\[
\begin{align*}
\text{sumReduce} & \;=\; \text{plus insert} \\
\text{a} & \;=\; \text{i. 2 3} \\
\begin{array}{c|c|c}
0 & 1 & 2 \\
3 & 4 & 5 \\
\end{array} \\
\text{byRows} & \;=\; "1 \quad \text{NB. Adverb to operate by rows} \\
(\text{sumReduce byRows}) \text{ a} & \\
\begin{array}{c|c|c}
\text{sumReduce 0 1 2} & \text{3} \\
\text{sumReduce 3 4 5} & \text{12} \\
\end{array}
\end{align*}
\]

Perform the row computations in parallel!
Rank Agreement - dyadic example

\[(i. \ 2 \ 3) \ (\text{plus byRows}) \ (i. \ 3)\]

\[
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 4 & 5 \\
\end{array}
\quad \text{plus} \quad 
\begin{array}{ccc}
0 & 1 & 2 \\
\end{array}
\quad 
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 4 & 5 \\
\end{array}
\quad \text{plus} \quad 
\begin{array}{ccc}
0 & 1 & 2 \\
\end{array}
\quad 
\begin{array}{ccc}
0 & 2 & 4 \\
3 & 5 & 7 \\
\end{array}
\]

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*)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

 plus

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>24</td>
<td>26</td>
<td>28</td>
<td>30</td>
</tr>
</tbody>
</table>

⇒

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>39</td>
<td>42</td>
<td>45</td>
</tr>
</tbody>
</table>
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.e. 8)`** NB. Works with `sumReduce` also

```
0 1 2 3 4 5 6 7
0 1 2 3
4 5 6 7
6 22
28
```

Actual implementation splits N-way
Other Parallel Opportunities – compose conjunction

\[ x (f \& h) y \quad \longleftrightarrow \quad (h x) f (h y) \]
Other Parallel Opportunities – monadic and dyadic fork construct

\[(f \ g \ h) \ y \longleftrightarrow (f \ y) \ g \ (h \ y)\]
\[x \ (f \ g \ h) \ y \longleftrightarrow (x \ f \ y) \ g \ (x \ h \ y)\]

Can execute in parallel
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

<table>
<thead>
<tr>
<th>Verbs</th>
<th>Adverbs</th>
<th>Conjunctions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Self-Create • Equal</td>
<td>• Reflex • Passive • Evoke</td>
<td>• Powers (a^n, a^n, a^n)</td>
</tr>
<tr>
<td>≤</td>
<td>• Insert • Table</td>
<td>• Determinant • Dot Product</td>
</tr>
<tr>
<td>&gt;</td>
<td>• Oblique • Key</td>
<td>• Even</td>
</tr>
<tr>
<td>≥</td>
<td>• Prefix • Index</td>
<td>• Odd</td>
</tr>
<tr>
<td>&gt;</td>
<td>• Suffix • Outfix</td>
<td>• Explicit • Monad-Dryad</td>
</tr>
<tr>
<td>+</td>
<td>• Is Amend • Amend (m, u)</td>
<td>• Observe</td>
</tr>
<tr>
<td>×</td>
<td>• Boolean • Basic</td>
<td>• Adverse</td>
</tr>
<tr>
<td>-</td>
<td>• Fix</td>
<td>• Cut</td>
</tr>
<tr>
<td>/</td>
<td>• Memo</td>
<td>• Fit (Customize)</td>
</tr>
<tr>
<td></td>
<td>• Taylor-Coeff, (m, u, u, u)</td>
<td>• Foreign</td>
</tr>
<tr>
<td></td>
<td>• Weighted Taylor</td>
<td>• Rank (a^n, a^n, a^n, a^n, a^n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Tie (Gerund)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Evoke Gerund</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Atop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Agenda</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• At</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Bond / Compose</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Under (Dual)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Under (Dual)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Appose</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Derivative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Derivative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Secant Slope</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hypergeometric</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Level At</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Spread</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Taylor Approximation</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Source, Computational Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlasLevel1a</td>
<td>Ourselves, Linear Algebra</td>
</tr>
<tr>
<td>BlasLevel2a</td>
<td>Ourselves, Linear Algebra</td>
</tr>
<tr>
<td>BlasLevel3a</td>
<td>Ourselves, Linear Algebra</td>
</tr>
<tr>
<td>BlasLevel3b</td>
<td>Ourselves, Linear Algebra</td>
</tr>
<tr>
<td>BlasLevel3c</td>
<td>Ourselves, Linear Algebra</td>
</tr>
<tr>
<td>CodeGolfDigit</td>
<td>Ourselves, Scalar arithmetic</td>
</tr>
<tr>
<td>GameOfLife</td>
<td>C. Jenkins, Stencil Computation</td>
</tr>
<tr>
<td>Josephus</td>
<td>JSoftware, Scalar arithmetic</td>
</tr>
<tr>
<td>MatrixInverse</td>
<td>JSoftware, Linear Algebra</td>
</tr>
<tr>
<td>MatrixMult</td>
<td>JSoftware, Linear Algebra</td>
</tr>
<tr>
<td>MatrixPower</td>
<td>JSoftware, Linear Algebra</td>
</tr>
<tr>
<td>MaximalClique</td>
<td>JSoftware, Graph Algorithm</td>
</tr>
<tr>
<td>MaxInfixSum</td>
<td>JSoftware, J adverbs</td>
</tr>
<tr>
<td>MergeSort</td>
<td>C. Jenkins, Array indexing</td>
</tr>
<tr>
<td>PartialSums1</td>
<td>JSoftware, Arithmetic Series Sum</td>
</tr>
<tr>
<td>PartialSums2</td>
<td>JSoftware, Geometric Series Sum</td>
</tr>
<tr>
<td>PartialSums3</td>
<td>JSoftware, Inverse quadratic series</td>
</tr>
<tr>
<td>PartialSums4</td>
<td>JSoftware, Flint Hills series</td>
</tr>
<tr>
<td>PartialSums5</td>
<td>JSoftware, Cookson Hills series</td>
</tr>
<tr>
<td>PartialSums6</td>
<td>JSoftware, Harmonic series</td>
</tr>
<tr>
<td>PartialSums7</td>
<td>JSoftware, Riemann Zeta series</td>
</tr>
<tr>
<td>PartialSums8</td>
<td>JSoftware, Alternating series</td>
</tr>
<tr>
<td>PartialSums9</td>
<td>JSoftware, Gregory series</td>
</tr>
<tr>
<td>PiComputation</td>
<td>C. Jenkins, Pi Series Sum</td>
</tr>
<tr>
<td>PrimePoly</td>
<td>JSoftware, Scalar arithmetic</td>
</tr>
<tr>
<td>ProjectEuler1</td>
<td>JSoftware, Scalar arithmetic</td>
</tr>
<tr>
<td>Rank0Verb</td>
<td>Ourselves, Scalar Arithmetic</td>
</tr>
<tr>
<td>SumReduceInt</td>
<td>Ourselves, Series Sum (int)</td>
</tr>
<tr>
<td>SumReduceDb1</td>
<td>Ourselves, Series Sum (double)</td>
</tr>
</tbody>
</table>
Experimental Results: Sequential Perf.

Slow-down / Speed-up: JSoftware vs. Truffle-J

- BlasLevel1a-100M: 21.257
- BlasLevel2a-5K: 0.599
- BlasLevel3a-1K: 0.375
- BlasLevel3b-1K: 0.332
- BlasLevel3c-1K: 0.325
- CodeGolfDigit-25: 1.830
- GameOfLife-2K: 1.402
- Josephus-8M: 1.588
- MatrixInverse-11: 46.330
- MatrixMult-1000: 0.332
- MatrixPower-500x16: 0.383
- MaximalClique-1K: 0.026
- MaxInfixSum-100K: 0.002
- MergeSort-16K: 0.132
- PartialSums1-100M: 0.234
- PartialSums2-100M: 4.643
- PartialSums3-100M: 23.237
- PartialSums4-50M: 2.499
- PartialSums5-50M: 2.424
- PartialSums6-100M: 18.174
- PartialSums7-100M: 24.539
- PartialSums8-100M: 6.677
- PartialSums9-100M: 7.821
- PiComputation-20M: 6.379
- PrimePoly-200: 0.861
- ProjectEuler1-100M: 12.908
- Rank0Verb-100K: 37.492
- SumReduceInt-250M: 0.358
- SumReduceDbl-250M: 42.723
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