PetaBricks
A Language and Compiler for Algorithmic Choice

Jason Ansel    Cy Chan    Yee Lok Wong    Marek Olszewski
Qin Zhao      Alan Edelman    Saman Amarasinghe

MIT - CSAIL

June 16, 2009
Outline

1. Introduction
   - Motivating Example
     - Language & Compiler Overview
     - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
Algorithmic choice

Mergesort (N-way)
Algorithmic choice

Mergesort (N-way)
Algorithmic choice

Mergesort (N-way)  Insertionsort
Algorithmic choice

Mergesort (N-way)

Insertionsort

Radixsort
Algorithmic choice

- Mergesort (N-way)
- Insertionsort
- Radixsort
- Quicksort
Algorithmic choice

Mergesort (N-way)

Insertionsort

Radixsort

Quicksort

STL Algorithm

N=2

@15
Algorithmic choice

Mergesort (N-way)  Insertionsort

Optimized For:  Xeon (1 core)

Radixsort  Quicksort

N=4  @75  @98
Algorithmic choice

Optimized For:
Xeon (1 core)
Xeon (8 cores)
Algorithmic choice

- Motivating Example

- Algorithmic choice:
  - Mergesort (N-way)
  - Insertionsort
  - Radixsort
  - Mergesort (N-way)
  - Quicksort

- Optimized For:
  - Xeon (1 core)
  - Xeon (8 cores)
  - Niagara (8 cores)
Algorithmic choice

- **Mergesort (N-way)**
  - N = 2, 4, 8, 16
  - Optimized for:
    - Xeon (1 core)
    - Xeon (8 cores)
    - Niagara (8 cores)
    - Core 2 (2 cores)

- **Insertionsort**
  - N = 2
  - Optimized for:
    - Xeon (1 core)
    - Xeon (8 cores)
    - Niagara (8 cores)
    - Core 2 (2 cores)

- **Radixsort**
  - N = 2, 4, 8
  - Optimized for:
    - Core 2 (2 cores)

- **Quick sort**
  - N = 4
  - Optimized for:
    - Xeon (1 core)
    - Xeon (8 cores)
    - Niagara (8 cores)
    - Core 2 (2 cores)
Algorithmic choice

Motivating Example

Introduction

Optimized For:
- Xeon (1 core)
- Xeon (8 cores)
- Niagra (8 cores)
- Core 2 (2 cores)

Quicksort
Quicksort
Quicksort
Quicksort
Quicksort
Quicksort
Quicksort
Quicksort
Quicksort

Insertionsort
Mergesort (N-way)
Radixsort

N=2,4,8
N=2
N=2,4,8,16
N=2,4,8

@98
@75
@1461
@2400

@150
@600
@1295

@1420
@38400
@600

Jason Ansel (MIT)

PetaBricks
The PetaBricks language

- The case for autotuning is obvious
The PetaBricks language

- The case for autotuning is obvious
- How should the programmer represent choice?
The PetaBricks language

- The case for autotuning is obvious
- How should the programmer represent choice?

- We present the PetaBricks programming language and compiler:
The PetaBricks language

- The case for autotuning is obvious
- How should the programmer represent choice?

- We present the **PetaBricks** programming language and compiler:
  - Choice as a fundamental language construct
The case for autotuning is obvious
How should the programmer represent choice?

We present the **PetaBricks** programming language and compiler:
- Choice as a fundamental language construct
- Autotuning performed by the compiler
The PetaBricks language

- The case for autotuning is obvious
- How should the programmer represent choice?

- We present the **PetaBricks** programming language and compiler:
  - Choice as a fundamental language construct
  - Autotuning performed by the compiler
  - Automatically parallelized
Sort in PetaBricks

1. \texttt{transform} \texttt{Sort}
2. \texttt{from} \texttt{A\[n\]}
3. \texttt{to} \texttt{B\[n\]}
4. \{
5. \texttt{from}(A \ a) \texttt{to}(B \ b) \{
6. \texttt{tunable} \texttt{WAYS;}
7. \texttt{/* Mergesort */}
8. \} \texttt{or} \{
9. \texttt{/* Insertionsort */}
10. \} \texttt{or} \{
11. \texttt{/* Radixsort */}
12. \} \texttt{or} \{
13. \texttt{/* Quicksort */}
14. \}
15. \}
Sort in PetaBricks

```java
1 transform Sort
2 from A[n]
3 to B[n]
4 {
5     from(A a) to(B b) {
6         tunable WAYS;
7         /* Mergesort */
8     } or {
9         /* Insertionsort */
10     } or {
11         /* Radixsort */
12     } or {
13         /* Quicksort */
14     }
15 }
```
Sort in PetaBricks

```plaintext
1  transform Sort
2  from A[n]
3  to B[n]
4  {
5    from(A a) to(B b) {
6      tunable WAYS;
7      /* Mergesort */
8    } or {
9      /* Insertionsort */
10     } or {
11      /* Radixsort */
12     } or {
13      /* Quicksort */
14    }
15  }
```
Sort in PetaBricks

1 transform Sort
2 from A[n]
3 to B[n]
4 {
5    from(A a) to(B b) {
6        tunable WAYS;
7        /* Mergesort */
8    } or {
9        /* Insertionsort */
10    } or {
11        /* Radixsort */
12    } or {
13        /* Quicksort */
14    }
15}

Jason Ansel (MIT)
Sort in PetaBricks

```java
1    transform    Sort
2    from    A[n]
3    to    B[n]
4    {
5        from(A a) to(B b) {
6            tunable    WAYS;
7            /* Mergesort */
8        }  or  {
9            /* Insertionsort */
10       }  or  {
11            /* Radixsort */
12       }  or  {
13            /* Quicksort */
14       }
15    }
```
Sort in PetaBricks

1. transform Sort
2. from A[n]
3. to B[n]
4. {
5.   from(A a) to(B b) {
6.     tunable WAYS;
7.     /* Mergesort */
8.   } or {
9.     /* Insertionsort */
10.   } or {
11.     /* Radixsort */
12.   } or {
13.     /* Quicksort */
14.   }
15. }

Jason Ansel (MIT)
Sort in PetaBricks

1    transform  Sort
2    from  A[n]
3    to  B[n]
4    {
5        from(A a) to(B b) {
6            tunable  WAYS;
7            /*  Mergesort  */
8        }  or  {
9            /*  Insertionsort  */
10        }  or  {
11            /*  Radixsort  */
12        }  or  {
13            /*  Quicksort  */
14    }
15    }
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
The PetaBricks compiler

- Sort is compiled into an autotuning binary
The PetaBricks compiler

- Sort is compiled into an autotuning binary
- Trained on target architecture
The PetaBricks compiler

- Sort is compiled into an autotuning binary
- Trained on target architecture
  - Structured genetic tuner
The PetaBricks compiler

- Sort is compiled into an autotuning binary
- Trained on target architecture
  - Structured genetic tuner
  - Trained with full number of threads
The PetaBricks compiler

- Sort is compiled into a autotuning binary
- Trained on target architecture
  - Structured genetic tuner
  - Trained with full number of threads
  - Under 1 minute for Sort
The PetaBricks compiler

- Sort is compiled into a autotuning binary
- Trained on target architecture
  - Structured genetic tuner
  - Trained with full number of threads
  - Under 1 minute for Sort
- Results fed back into the compiler
- Final binary created
Sort algorithm timings

On an 8-way Xeon E7340 system
Sort algorithm timings

On an 8-way Xeon E7340 system

Jason Ansel (MIT)
Sort algorithm timings\(^1\)

\(^1\)On an 8-way Xeon E7340 system
Sort algorithm timings

On an 8-way Xeon E7340 system
Sort algorithm timings\(^1\)

\(^1\)On an 8-way Xeon E7340 system
## Timings on different architectures

<table>
<thead>
<tr>
<th>Run on</th>
<th>Mobile</th>
<th>Xeon 1-way</th>
<th>Xeon 8-way</th>
<th>Niagara</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>-</td>
<td>1.09x</td>
<td>1.67x</td>
<td>1.47x</td>
</tr>
<tr>
<td>Xeon 1-way</td>
<td>1.61x</td>
<td>-</td>
<td>2.08x</td>
<td>2.50x</td>
</tr>
<tr>
<td>Xeon 8-way</td>
<td>1.59x</td>
<td>2.14x</td>
<td>-</td>
<td>2.35x</td>
</tr>
<tr>
<td>Niagara</td>
<td>1.12x</td>
<td>1.51x</td>
<td>1.08x</td>
<td>-</td>
</tr>
</tbody>
</table>
## Timings on different architectures

<table>
<thead>
<tr>
<th>Run on</th>
<th>Mobile</th>
<th>Xeon 1-way</th>
<th>Xeon 8-way</th>
<th>Niagara</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>-</td>
<td>1.09x</td>
<td>1.67x</td>
<td>1.47x</td>
</tr>
<tr>
<td>Xeon 1-way</td>
<td>1.61x</td>
<td>-</td>
<td>2.08x</td>
<td>2.50x</td>
</tr>
<tr>
<td>Xeon 8-way</td>
<td>1.59x</td>
<td>2.14x</td>
<td>-</td>
<td>2.35x</td>
</tr>
<tr>
<td>Niagara</td>
<td>1.12x</td>
<td>1.51x</td>
<td>1.08x</td>
<td>-</td>
</tr>
</tbody>
</table>
## Timings on different architectures

<table>
<thead>
<tr>
<th>Run on</th>
<th>Mobile</th>
<th>Xeon 1-way</th>
<th>Xeon 8-way</th>
<th>Niagara</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>-</td>
<td>1.09x</td>
<td>1.67x</td>
<td>1.47x</td>
</tr>
<tr>
<td>Xeon 1-way</td>
<td>1.61x</td>
<td>-</td>
<td>2.08x</td>
<td>2.50x</td>
</tr>
<tr>
<td>Xeon 8-way</td>
<td>1.59x</td>
<td>2.14x</td>
<td>-</td>
<td>2.35x</td>
</tr>
<tr>
<td>Niagara</td>
<td>1.12x</td>
<td>1.51x</td>
<td>1.08x</td>
<td>-</td>
</tr>
</tbody>
</table>
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
Early computers (and compilers) were weak
Early compilers

- Early computers (and compilers) were weak
- Parsing and code generation dominated compilation

Constrained Input Language (No choices) -> Parsing -> Code Gen
Early computers (and compilers) were weak

- Parsing and code generation dominated compilation
- Needed a constrained input language to simplify compilation
Current compilers

- Current computers are much more powerful
- Compilers can do a lot more
Current compilers

- Current computers are much more powerful
- Compilers can do a lot more
- Input language is still constraining
Current compilers

- Current computers are much more powerful
- Compilers can do a lot more
- Input language is still constraining
- Compilation dominated by exposing choices
Current compilers

- Current computers are much more powerful
- Compilers can do a lot more
- Input language is still constraining
- Compilation dominated by exposing choices
- Input language specifies only one
  - Algorithmic choice
  - Iteration order choice
  - Parallelism strategy choice
  - Data layout choice
Current compilers

- Current computers are much more powerful
- Compilers can do a lot more
- Input language is still constraining
- Compilation dominated by exposing choices
- Input language specifies only one:
  - Algorithmic choice
  - Iteration order choice
  - Parallelism strategy choice
  - Data layout choice
- Compiler must perform heroic analysis to reconstruct other choices
We propose explicit choices in the language
We propose explicit choices in the language

The programmer defines the space of legal

- Algorithmic choices
- Iteration orders (include parallel)
- Data layouts
We propose explicit choices in the language
The programmer defines the space of legal
- Algorithmic choices
- Iteration orders (include parallel)
- Data layouts
Allow compilers to focus on exploring choices
Compiler no longer needs to reconstruct choices
Future-proof programs

- The result: programs can adapt to their environment
Future-proof programs

- The result: programs can adapt to their environment
- Choices make programs less brittle
Future-proof programs

- The result: programs can adapt to their environment
- Choices make programs less brittle
- Programs change with architecture, available cores, inputs, etc
Outline

1 Introduction
   • Motivating Example
   • Language & Compiler Overview
   • Why choices

2 PetaBricks Language
   • Key Ideas
   • Compilation Example
   • Other Language Features

3 Results
   • Benchmarks
   • Scalability
   • Variable Accuracy

4 Conclusion
   • Final thoughts
Algorithmic choice in the language

- Algorithmic choice is the key aspect of PetaBricks
Algorithmic choice in the language

- Algorithmic choice is the key aspect of PetaBricks
- Programmer can define multiple rules to compute the same data
Algorithmic choice is the key aspect of PetaBricks

- Programmer can define multiple rules to compute the same data
- Compiler re-use rules to create hybrid algorithms
Algorithmic choice in the language

- Algorithmic choice is the key aspect of PetaBricks
- Programmer can define multiple rules to compute the same data
- Compiler re-use rules to create hybrid algorithms
- Can express choices at many different granularities
Synthesized outer control flow

- Outer control flow synthesized by compiler
Synthesized outer control flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
Synthesized outer control flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
Synthesized outer control flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
  - By columns?
Synthesized outer control flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
  - By columns?
  - Diagonal? Reverse order? Blocked?
  - Parallel?

Jason Ansel (MIT)
Synthesized outer control flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
  - By columns?
  - Diagonal? Reverse order? Blocked?
  - Parallel?
- Instead programmer provides explicit producer-consumer relations
Synthesized outer control flow

- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
  - By columns?
  - Diagonal? Reverse order? Blocked?
  - Parallel?
- Instead programmer provides explicit producer-consumer relations
- Allows compiler to explore choice space
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
Simple example program

```plaintext
1  transform  RollingSum
2  from  A[n]  
3  to  B[n]  
4  {
5     // rule 0: use the previously computed value
6     B.cell(i) from(A.cell(i) a,
7         B.cell(i-1) leftSum) {
8         return  a+leftSum;
9     }
10 
11     // rule 1: sum all elements to the left
12     B.cell(i) from(A.region(0, i) in) {
13         return  sum(in);
14     }
15 }
```
Simple example program

1 transform RollingSum
2 from A[n] to B[n]
3 {
4     // rule 0: use the previously computed value
5     B.cell(i) from (A.cell(i) a,
6         B.cell(i - 1) leftSum) {
7         return a + leftSum;
8     }
9 }
10
11 // rule 1: sum all elements to the left
12 B.cell(i) from (A.region(0, i) in) {
13     return sum(in);
14 }
15 }
Simple example program

...  

5   // rule 0: use the previously computed value
6   B.cell(i) from(A.cell(i) a,
7       B.cell(i-1) leftSum) {
8       return a+leftSum;
9   }

...
// rule 1: sum all elements to the left
B.cell(i) from A.region(0, i) in { 
    return sum(in);
}
## Applicable regions

<table>
<thead>
<tr>
<th>Compilation Process</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applicable regions</strong></td>
</tr>
</tbody>
</table>

- **Rule 0**: Use the previously computed value $B.cell(i)$ from $(A.cell(i), B.cell(i-1).leftSum)$.

  ```
  return a + leftSum;
  ```

  Applicable where $1 \leq i < n$

- **Rule 1**: Sum all elements to the left $B.cell(i)$ from $(A.region(0, i)in)$.

  ```
  return sum(in);
  ```

  Applicable where $0 \leq i < n$
Applicable regions

<table>
<thead>
<tr>
<th>Compilation Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicable regions</td>
</tr>
</tbody>
</table>

// rule 0: use the previously computed value
B.cell(i) from (A.cell(i) a,  
                   B.cell(i-1) leftSum) {
    return a+leftSum;
}

- Applicable where 1 ≤ i < n
Applicable regions

<table>
<thead>
<tr>
<th>Compilation Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicable regions</td>
</tr>
</tbody>
</table>

// rule 0: use the previously computed value
B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
    return a+leftSum;
}

- Applicable where $1 \leq i < n$

// rule 1: sum all elements to the left
B.cell(i) from (A.region(0, i) in) {
    return sum(in);
}

- Applicable where $0 \leq i < n$
# Choice grids

## Compilation Process

<table>
<thead>
<tr>
<th>Applicable regions</th>
<th>Choice grids</th>
<th>Choice dependency graph</th>
</tr>
</thead>
</table>
Choice grids

<table>
<thead>
<tr>
<th>Compilation Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicable regions</td>
</tr>
</tbody>
</table>

- Divide data space into symbolic regions with common sets of choices
Choice grids

<table>
<thead>
<tr>
<th>Compilation Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicable regions</td>
</tr>
</tbody>
</table>

- Divide data space into symbolic regions with common sets of choices
- In this simple example:
  - A: Input (no choices)
  - B: [0, 1) = rule 1
  - B: [1, n) = rule 0 or rule 1

```
R1  R0 or R1
0   1   n
```
Choice grids

<table>
<thead>
<tr>
<th>Compilation Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicable regions</td>
</tr>
</tbody>
</table>

- Divide data space into symbolic regions with common sets of choices
- In this simple example:
  - A: Input (no choices)
  - B: \([0, 1) = \text{rule 1}\)
  - B: \([1, n) = \text{rule 0 or rule 1}\)

Applicable regions map rules → symbolic data
Choice grids map symbolic data → rules
### Choice dependency graph

#### Compilation Process

<table>
<thead>
<tr>
<th>Applicable regions</th>
<th>Choice grids</th>
<th>Choice dependency graph</th>
</tr>
</thead>
</table>

- **A.region(0, n)**
  - (r1,<=), (r0,=)

- **B.region(0, 1)**
  - Choices: r1

- **B.region(1, n)**
  - Choices: r0, r1

- Edge annotated with directions and rules:
  - (r0,=,-1)
  - (r1,<=),(r0,=)

This graph shows the dependencies between symbolic regions and how choices are made in the compilation process.

Jason Ansel (MIT)

PetaBricks

June 16, 2009 24 / 47
Choice dependency graph

Compilation Process

<table>
<thead>
<tr>
<th>Applicable regions</th>
<th>Choice grids</th>
<th>Choice dependency graph</th>
</tr>
</thead>
</table>

- Adds dependency edges between symbolic regions

A.region(0, n) → B.region(0, 1) → B.region(1, n)

B.region(1, n) Choices: r0, r1

A.region(0, n) Choices: r1

B.region(0, 1) Choices: r0, =

(r1, <=), (r0, =)

(r0, =, -1)

(r1, <=), (r0, =)

(r0, =, -1)
Choice dependency graph

- Adds dependency edges between symbolic regions
- Edges annotated with directions and rules
Choice dependency graph

Compilation Process

<table>
<thead>
<tr>
<th>Applicable regions</th>
<th>Choice grids</th>
<th>Choice dependency graph</th>
</tr>
</thead>
</table>

- Adds dependency edges between symbolic regions
- Edges annotated with directions and rules
- Many compiler passes on this IR to:

```
A.region(0, n)
(r1,<=),(r0,=)

B.region(0, 1)
Choices: r1

(r1,<=),(r0,=)

B.region(1, n)
Choices: r0, r1

(r0,=,-1)
```
Choice dependency graph

Compilation Process

<table>
<thead>
<tr>
<th>Applicable regions</th>
<th>Choice grids</th>
<th>Choice dependency graph</th>
</tr>
</thead>
</table>

- Adds dependency edges between symbolic regions
- Edges annotated with directions and rules
- Many compiler passes on this IR to:
  - Simplify complex dependency patterns
Choice dependency graph

Compilation Process

<table>
<thead>
<tr>
<th>Applicable regions</th>
<th>Choice grids</th>
<th>Choice dependency graph</th>
</tr>
</thead>
</table>

A.region(0, n)
(r1,<=),(r0,=)

B.region(0, 1)
Choices: r1

B.region(1, n)
Choices: r0, r1
(r0,=,-1)

- Adds dependency edges between symbolic regions
- Edges annotated with directions and rules
- Many compiler passes on this IR to:
  - Simplify complex dependency patterns
  - Add choices
Code generation

1. PetaBricks source code is compiled

2. An autotuning binary is created

3. Autotuning occurs creating a choice configuration file

4. Choices are fed back into the compiler to create a final binary
Code generation

1. PetaBricks source code is compiled
2. An autotuning binary is created
Code generation

1. PetaBricks source code is compiled
2. An autotuning binary is created
3. Autotuning occurs creating a choice configuration file
Code generation

1. PetaBricks source code is compiled
2. An autotuning binary is created
3. Autotuning occurs creating a choice configuration file
4. Choices are fed back into the compiler to create a final binary
Autotuning

- Based on two building blocks:
  - A genetic tuner
  - An $n$-ary search algorithm
Autotuning

- Based on two building blocks:
  - A genetic tuner
  - An $n$-ary search algorithm

- Flat parameter space

- Compiler generates a dependency graph describing this parameter space
Autotuning

- Based on two building blocks:
  - A genetic tuner
  - An $n$-ary search algorithm
- Flat parameter space
- Compiler generates a dependency graph describing this parameter space
- Entire program tuned from bottom up
Parallel Runtime Library

- Task-based parallel runtime
- Thread-local decks of runnable tasks
Parallel Runtime Library

- Task-based parallel runtime
- Thread-local decks of runnable tasks
- Use a work-stealing algorithm similar to that of Cilk
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
More PetaBricks features

- Automatic consistency checking
- The *tunable* keyword
- Call external code
- Custom training data generators
- Matrix *versions* for iterative algorithms
- Rule priorities
- *where* (clause for limiting applicable regions)
- Template transforms
More PetaBricks features

- Automatic consistency checking
- The *tunable* keyword
- Call external code
- Custom training data generators
- Matrix *versions* for iterative algorithms
- Rule priorities
- *where* (clause for limiting applicable regions)
- Template transforms
More PetaBricks features

- Automatic consistency checking
- The *tunable* keyword
- Call external code
- Custom training data generators
- Matrix *versions* for iterative algorithms
- Rule priorities
- `where` (clause for limiting applicable regions)
- Template transforms
More PetaBricks features

- Automatic consistency checking
- The *tunable* keyword
- Call external code
- Custom training data generators
- Matrix *versions* for iterative algorithms
- Rule priorities
- *where* (clause for limiting applicable regions)
- Template transforms
Outline

1 Introduction
   • Motivating Example
   • Language & Compiler Overview
   • Why choices

2 PetaBricks Language
   • Key Ideas
   • Compilation Example
   • Other Language Features

3 Results
   • Benchmarks
   • Scalability
   • Variable Accuracy

4 Conclusion
   • Final thoughts
Eigenvector Solve

- Bisection
- QR decomposition
- Divide and conquer
Eigenvector Solve

![Graph showing time (s) vs input size for Eigenvector Solve benchmark. The x-axis represents input size, ranging from 0 to 1000, and the y-axis represents time in seconds, ranging from 0 to 0.12. The graph includes a trend line with the label "Bisection." ](image)
Eigenvector Solve

![Graph showing time vs. input size for Bisection and DC methods.]

- **Bisection**
- **DC**

Jason Ansel (MIT)

PetaBricks

June 16, 2009
Eigenvector Solve
Eigenvector Solve

![Graph showing Eigenvector Solve results with different methods: Bisection, DC, QR, and Autotuned. The x-axis represents Input Size, and the y-axis represents Time (s). The graph compares the performance of these methods over a range of input sizes.]
Eigenvector Solve

![Graph showing time vs input size for different methods: Bisection, DC, QR, Autotuned Cutoff 25]
Matrix Multiply

- Basic
- Recursive decompositions
- Strassen’s algorithm
- Iteration order (blocking)
- Transpose
Matrix Multiply

![Graph showing the relationship between input size and time for Matrix Multiply benchmarks. The x-axis represents input size, while the y-axis represents time in seconds. The graph shows a logarithmic scale for both axes, with time values ranging from $10^{-6}$ to 10000 seconds and input sizes ranging from 1 to 10000.]
Matrix Multiply

![Graph showing the comparison between Basic and Blocking for different input sizes. The x-axis represents the input size, while the y-axis represents the time in seconds. The graph demonstrates that Blocking performs better than Basic for large input sizes.](image-url)
Matrix Multiply

![Graph showing Matrix Multiply benchmarks for Basic, Blocking, and Transpose methods across varying input sizes.]
Matrix Multiply

![Graph showing the performance of different matrix multiply techniques against input size. The x-axis represents input size, and the y-axis represents time (s). The graph compares Basic, Blocking, Transpose, and Recursive methods. The time decreases as the input size increases for all methods, with Blocking showing the best performance.](image-url)
Matrix Multiply

![Graph showing the comparison of Matrix Multiply benchmarks (Basic, Blocking, Transpose, Recursive, Strassen 256) across different input sizes (1, 10, 100, 1000, 10000). The y-axis represents time in seconds, and the x-axis represents input size. The graph illustrates the performance differences among the benchmarks for various input sizes.]
Matrix Multiply

The image shows a log-log plot comparing different algorithms for matrix multiplication. The x-axis represents the input size, while the y-axis represents the time taken in seconds. Several algorithms are compared, including Basic, Blocking, Transpose, Recursive, Strassen 256, and Autotuned. The plot illustrates how the time increases with the input size for each algorithm, with varying slopes indicating different efficiencies.
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
Scalability

![Graph showing scalability](image)

- **Autotuned Matrix Multiply**

  - Speedup vs. Number of Threads
  - Data points show a linear increase with more threads.

  - Jason Ansel (MIT)

  - PetaBricks

  - June 16, 2009
Scalability

![Scalability Graph]

- Autotuned Matrix Multiply
- Autotuned Sort

Jason Ansel (MIT)
Scalability

![Graph showing scalability results for different algorithms. The graph plots speedup against the number of threads, with lines representing Autotuned Matrix Multiply, Autotuned Sort, and Autotuned Poisson.]
Scalability

![Graph showing scalability of Autotuned Matrix Multiply, Autotuned Sort, Autotuned Poisson, and Autotuned Eigenvector Solve with varying number of threads.](image-url)
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
Variable accuracy

- Most algorithms produce exact solutions
Variable accuracy

- Most algorithms produce exact solutions
- Large class of algorithms can produce approximate solutions
Variable accuracy

- Most algorithms produce exact solutions
- Large class of algorithms can produce approximate solutions
  - Iterative convergence
  - Grid coarsening
  - Others
Variable accuracy

- Most algorithms produce exact solutions
- Large class of algorithms can produce approximate solutions
  - Iterative convergence
  - Grid coarsening
  - Others
- Compiler/autotuner should be aware of variable accuracy
Variable accuracy

- Most algorithms produce exact solutions
- Large class of algorithms can produce approximate solutions
  - Iterative convergence
  - Grid coarsening
  - Others
- Compiler/autotuner should be aware of variable accuracy
- Compiler can examine optimal frontier of algorithms
Poisson’s equation

- A variable accuracy benchmark
- Accuracy level expressed as a template parameter
- Autotuner exploits variable accuracy in a general way
- Choices:
  - Direct solve
  - Jacobi iteration
  - Successive over relaxation
  - Multigrid
**Choices in Multigrid**

- SOR is an iterative algorithm

---

**Multigrid** changes grid coarseness to speed up convergence. Many standard shapes: V-Cycle, W-Cycle, etc.

- Direct solver

Jason Ansel (MIT)
Choices in Multigrid

- SOR is an iterative algorithm
- Multigrid changes grid coarseness to speed up convergence
- Many standard shapes: V-Cycle,
Choices in Multigrid

- SOR is an iterative algorithm
- Multigrid changes grid coarseness to speed up convergence
- Many standard shapes: V-Cycle, W-Cycle, etc
**Choices in Multigrid**

- SOR is an iterative algorithm
- Multigrid changes grid coarseness to speed up convergence
- Many standard shapes: V-Cycle, W-Cycle, etc
- Direct solver
Choices in Multigrid

- SOR is an iterative algorithm
- Multigrid changes grid coarseness to speed up convergence
- Many standard shapes: V-Cycle, W-Cycle, etc
- Direct solver
- Different shapes = different algorithms
Autotuned V-cycle shapes for different accuracy requirements
Autotuned V-cycle shapes for different accuracy requirements
Autotuned V-cycle shapes for different accuracy requirements
Autotuned V-cycle shapes for different accuracy requirements

![Diagram showing autotuned V-cycle shapes for different accuracy requirements.](image-url)
Dynamic programming technique for autotuning Multigrid
Dynamic programming technique for autotuning Multigrid
Dynamic programming technique for autotuning Multigrid

- Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

- Partition accuracy space into discrete levels
Dynamic programming technique for autotuning Multigrid

- Partition accuracy space into discrete levels
- Base space of candidate algorithms on optimal algorithms from coarser level
Dynamic programming technique for autotuning Multigrid

- Partition accuracy space into discrete levels
- Base space of candidate algorithms on optimal algorithms from coarser level
Dynamic programming technique for autotuning Multigrid

- Partition accuracy space into discrete levels
- Base space of candidate algorithms on optimal algorithms from coarser level
Poisson’s Equation

Results

<table>
<thead>
<tr>
<th>Variable Accuracy</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e-05</td>
<td></td>
</tr>
<tr>
<td>0.0001</td>
<td></td>
</tr>
<tr>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
</tr>
</tbody>
</table>

Jason Ansel (MIT)

PetaBricks

June 16, 2009
Poisson’s Equation

![Graph showing the time vs input size for Direct and Jacobi methods.

- Direct method is represented by blue dots.
- Jacobi method is represented by green crosses.

The x-axis represents the input size, ranging from 1 to 10,000.

The y-axis represents the time in seconds, ranging from 1e-05 to 10,000.

As the input size increases, the time for both methods also increases, with the Jacobi method generally taking more time than the Direct method.

Jason Ansel (MIT)
PetaBricks
June 16, 2009
Poisson’s Equation

The graph illustrates the time (in seconds) required to solve Poisson’s Equation for different input sizes using three different methods: Direct, Jacobi, and SOR. The x-axis represents the input size, while the y-axis shows the time taken. The data points are shown for accuracy levels ranging from $10^{-5}$ to 10,000, with the time scaling logarithmically on the y-axis.

Methods:
- Direct
- Jacobi
- SOR

Accuracy Levels:
- $10^{-5}$
- 0.0001
- 0.001
- 0.01
- 0.1
- 1
- 10
- 100
- 1000
- 10000

The graph shows how each method scales differently with respect to accuracy and input size.
Poisson’s Equation

![Graph showing the results of different methods for solving Poisson’s Equation. The x-axis represents the input size, and the y-axis represents the time in seconds. The methods include Direct, Jacobi, SOR, and Multigrid. The graph shows the relationship between accuracy and time for each method.]
Poisson’s Equation

Results

Variable Accuracy

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e-05</td>
<td></td>
</tr>
<tr>
<td>0.0001</td>
<td></td>
</tr>
<tr>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
</tr>
</tbody>
</table>

Input Size

Direct
Jacobi
SOR
Multigrid
Autotuned

Jason Ansel (MIT)
Outline

1. Introduction
   - Motivating Example
   - Language & Compiler Overview
   - Why choices

2. PetaBricks Language
   - Key Ideas
   - Compilation Example
   - Other Language Features

3. Results
   - Benchmarks
   - Scalability
   - Variable Accuracy

4. Conclusion
   - Final thoughts
Related work

- **Languages**
  - Sequoia

- **Libraries & domain specific tuners**
  - STAPL
  - ATLAS
  - FFTW
  - SPARSITY
  - SPIRAL
  - ...

---

Jason Ansel (MIT)
For more information

- PetaBricks makes programs **future-proof**, by allowing them to adapt to new architectures
- We plan to released PetaBricks at the end of summer
- Sign up for our mailing list to be notified
- For more information see: http://projects.csail.mit.edu/petabricks/
- Questions?
Thank you!