PetaBricks: A Language and Compiler for Algorithmic Choice

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

CSAIL, MIT
The problem

• No one-size-fits-all solution for high performance algorithms.
• Optimal solutions rely on architectures, problem size, data, etc.
• Limited compiler and programming language can change parameters in algorithms.
  - coarse-grained level
  - library level selections
  - hard-code cutoffs
What’s PetaBricks

• An implicitly parallel language and compiler
  - allow algorithmic choice
  - compiler autotunes fine-grained level programs
• Techniques of autotuning algorithms
  - direct and iterative methods choosing
  - almost always near-optimal efficiency
  - compiler decides various components for different convergence rate.
Motivating Example

• Problem of sorting
• Insertion sort, quick sort, merge sort, bubble sort, heap sort, radix sort, bucket sort, etc.
• Empirical opinion:
  - small inputs, insertion sort
  - medium size, quick sort
  - very large size, radix sort
• Considering parallelism, how to determine the strategy?
PetaBricks Language

• Transform
  - defines an algorithm that can be called from other transforms. (as function)
  - header: to, from, through (outputs, inputs, data used within the transform)

• Rules
  - user encodes rules in each transform
  - rule defines how to compute a region of data
  - rule has explicit dependencies
  - the compiler should find a sequence of rules
Example of PetaBricks transform

```plaintext
1 transform MatrixMultiply
2 from A[c,h], B[w,c]
3 to AB[w,h]
4 {
5    // Base case, compute a single element
6    to (AB.cell(x,y) out)
7    from (A.row(y) a, B.column(x) b) {
8        out = dot(a,b);
9    }
10
11    // Recursively decompose in c
12    to (AB ab)
13    from (A.region(0, 0, c/2, h) a1,
14         A.region(c/2, 0, c, h) a2,
15         B.region(0, 0, w, c/2) b1,
16         B.region(0, c/2, w, c) b2) {
17        ab = MatrixAdd(MatrixMultiply(a1, b1),
18                         MatrixMultiply(a2, b2));
19    }
20
21    // Recursively decompose in w
22    to (AB.region(0, 0, w/2, h) ab1,
23        AB.region(w/2, 0, w, h) ab2)
24    from (A a,
25           B.region(0, 0, w/2, c) b1,
26           B.region(w/2, 0, w, c) b2) {
27        ab1 = MatrixMultiply(a, b1);
28        ab2 = MatrixMultiply(a, b2);
29    }
30}
31
32    // Recursively decompose in h
33    to (AB.region(0, 0, w, h/2) ab1,
34        AB.region(0, h/2, w, h) ab2)
35    from (A.region(0, 0, c, h/2) a1,
36         A.region(0, h/2, c, h) a2,
37         B b) {
38        ab1 = MatrixMultiply(a1, b);
39        ab2 = MatrixMultiply(a2, b);
40    }
41
Figure 1. PetaBricks source code for MatrixMultiply
```
Implementation

- A source-to-source compiler from the PetaBricks language to C++
- An autotuning system and choice framework to find optimal choices and set parameters
- A runtime library used by the generated code
Framework
Rolling sum example

```java
transform RollingSum
from A[n]
to B[n]
{
  // rule0: sum all elements to the left
to(B.cell(i) b) from(A.region(0, i) in) {
    b=sum(in);
  }

  // rule1: use the previously computed value
to(B.cell(i) b) from(A.cell(i) a,
               B.cell(i-1) leftSum) {
    b=a+leftSum;
  }
}
```

**Figure 3.** PetaBricks source code for RollingSum. A simple example used to demonstrate the compilation process. The output element $B_x$ is the sum of the input elements $A_0...A_x$. 
Compilation: Parsing and normalization

• Input is parsed into an abstract syntax tree.
• Rule dependencies -> region syntax
  - assign each rule a symbolic center
  - all dependencies relative to the center
• e.g. add 1 to rule 0,
  \[ b = \text{sum}(\text{in} + 1); \quad \rightarrow \text{from}(A, \text{region}(0,i+1)) \]
Compilation: Applicable regions

- The region each rule can legally be applied
- In rule 0, 
  \( b \rightarrow [0,n), \quad \text{in-} \rightarrow [0,n) \)

  In rule 1, 
  \( a \rightarrow [0,n), \quad b \rightarrow [0,n), \quad \text{leftsum-} \rightarrow [1,n) \)

  intersection of \([0,n)\) and \([1,n)\) is \([1,n)\)
Compilation: Choice grid analysis

- The choice grid divides each matrix into rectilinear regions where a uniform set of rules are applicable

\[
[0,1) = \{\text{rule 0}\} \\
[1,n) = \{\text{rule0, rule1}\}
\]
Compilation: Choice dependency graph

- Dependency graph connects the symbolic regions from the choice grid

- This high-level coarse graph is passed to the dynamic scheduler to execute in parallel at run time
Compilation: Code generation

• Default mode: choices and information for autotuning are embedded. This binary can be dynamically tuned.

• The second mode: a previously tuned configuration file is applied statically.
Parallelism in Output Code

• Dynamic scheduler
  - the dependency edges expose all available parallelism to the dynamic scheduler.
  - dynamic scheduler may change behavior based on autotuned parameters
• Continuation points, storing all needed state to heap.
• Two version of output code
  - dynamically scheduled task-based
  - entirely sequential without dynamic scheduler
Autotuning

• On the target system, autotuning is run to find optimal choices and cutoffs.
• The choice varies by architecture and number of processors.
• Autotuning library is embedded in output program if choices are not statically compiled.
• It outputs a configuration file containing choices.
Algorithm choice

• Genetic tuner manner
  - population of candidate algorithms
  - autotuner starts with small input size, doubles it at each iteration.
  - new alg. generated by adding levels to the fastest algorithms.
  - slower alg. removed
Runtime library

• Managing parallelism, data and configuration
  - runtime scheduler
distribute task across processors satisfying dependencies.

• Thread-private deque and task stealing protocol
Other concerns

• Automated consistency check
  - multiple algs must have the same results
• Deadlocks
  - no deadlocks because of elimination of cycles within the dependency graph
• Race condition
  - implicitly parallel (complier did)
Benchmarks 1: Poisson’s Equation

$$\nabla^2 \phi = f \quad \text{and} \quad Tx = b$$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Direct</th>
<th>Jacobi</th>
<th>SOR</th>
<th>Multigrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>$n^{1.5}$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

(a) Dependencies for the red cells

(b) Dependencies for the black cells
Benchmarks 1: Poisson’s Equation

• Multigrid

\[
\text{MULTIGRID-SIMPLE}(x, b) \\
1: \textbf{if } N = 3 \textbf{ then} \\
2: \quad \text{Solve directly} \\
3: \textbf{else} \\
4: \quad \text{Iterate using some iterative method} \\
5: \quad \text{Compute the residual and restrict to half resolution} \\
6: \quad \text{Recursively call MULTIGRID-SIMPLE on coarser grid} \\
7: \quad \text{Interpolate result and add correction term to current solution} \\
8: \quad \text{Iterate using some iterative method} \\
9: \textbf{end if}
\]
DP solution

• Suppose we have solved the input $A(k-1)$ for level $k-1$.
• For level $k$, try all algs in $A(k-1)$ for step 6.
• Varying parameters in other steps
Solution Plot

(a)

(b)
PetaBrick’s solution

\text{POISSON}_t(x, b)

1: \textbf{either}
2: \text{Solve directly}
3: \text{Iterate using } \text{SOR}_{\omega_{\text{opt}}} \text{ until accuracy } p_t \text{ is achieved}
4: \text{For some } j, \text{ iterate with } \text{MULTIGRID}_j \text{ until accuracy } p_t \text{ is achieved}
5: \textbf{end either}

\text{MULTIGRID}_t(x, b)

1: \textbf{if } N = 3 \textbf{ then}
2: \text{Solve directly}
3: \textbf{else}
4: \text{Compute one iteration of } \text{SOR}_{1.15}
5: \text{Compute the residual and restrict to half resolution}
6: \text{On the coarser grid, call } \text{POISSON}_t
7: \text{Interpolate result and add correction term to current solution}
8: \text{Compute one iteration of } \text{SOR}_{1.15}
9: \textbf{end if}
Figure 12. Performance for Eigenproblem on 8 cores. “Cutoff 25” corresponds to the hard-coded hybrid algorithm found in LAPACK.

Figure 14. Performance for sort on 8 cores.

Figure 15. Performance for Matrix Multiply on an 8 cores. “Strassen 256” uses strassen algorithm to decompose until n=256 when it switches to basic matrix multiply.

Figure 16. Parallel scalability. Speedup as more worker threads are added. Run on an 8-way (2 processor × 4 core) x86_64 Intel Xeon System.
### Effect of Architecture on autotuning

<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>

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>System</th>
<th>Frequency</th>
<th>Cores used</th>
<th>Scalability</th>
<th>Algorithm Choices (w/ switching points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>Core 2 Duo Mobile</td>
<td>1.6 GHz</td>
<td>2 of 2</td>
<td>1.92</td>
<td>IS(150) 8MS(600) 4MS(1295) 2MS(38400) QS(∞)</td>
</tr>
<tr>
<td>Xeon 1-way</td>
<td>Xeon E7340 (2 x 4 core)</td>
<td>2.4 GHz</td>
<td>1 of 8</td>
<td>-</td>
<td>IS(75) 4MS(98) RS(∞)</td>
</tr>
<tr>
<td>Xeon 8-way</td>
<td>Xeon E7340 (2 x 4 core)</td>
<td>2.4 GHz</td>
<td>8 of 8</td>
<td>5.69</td>
<td>IS(600) QS(1420) 2MS(∞)</td>
</tr>
</tbody>
</table>
| Niagara      | Sun Fire T200 Niagara   | 1.2 GHz   | 8 of 8     | 7.79        | 16MS(75) 8MS(1461) 4MS(2400) 2MS(∞) }
Conclusions

• No single choice of parameters can yield the best possible result.

• Sequential and parallel algorithms may be quite different.

• PetaBricks empowers the compiler to combine different algorithms in a deeper manner.

• Languages and compilers such as PetaBricks are designed architecture-independent.