Petabricks: A language and compiler for algorithmic choice

Motivation – Many algorithms with various trade offs

<table>
<thead>
<tr>
<th>Sorting</th>
<th>MergeSort</th>
<th>QuickSort</th>
<th>InsertionSort</th>
<th>RadixSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast for medium size input</td>
<td>Fastest for medium size input</td>
<td>Fastest for small input</td>
<td>Fastest for largest input</td>
<td></td>
</tr>
<tr>
<td>Highly parallelizable</td>
<td>Exploit spatial locality</td>
<td>O(1) Memory</td>
<td>O(wn) runtime</td>
<td></td>
</tr>
</tbody>
</table>

- But gets more complicated!
- Mixing the algorithms yield better results
  - e.g. QuickSort then cut off to InsertionSort once list is small enough
    - Requires knowing an optimal cut-off point!
    - Differs from an architecture to an architecture
- Choices are left to the developer, complex, time-consuming, error prone

Hence the need for Autotuning
PetaBricks – A language and a compiler

- Self-tuning compiler for bespoke architecture
- A language that allow expressing choice in algorithms
- Implicitly parallelizable
- Auto-select the desired trade-off between accuracy and performance
Sample Code - Sorting

```c
#define SORT Sort

// #include "Bitonicsort.pbcc"
#include "Insertionsort.pbcc"
#include "Mergesort2.pbcc"
#include "Quicksort2.pbcc"
#include "Selectionsort.pbcc"

function Sort
from in[n]
to out[n]
{
    Mergesort2(out, in);
} or {
    Mergesort4(out, in);
} or {
    Quicksort2(out, in);
} or {
    Insertionsort(out, in);
} or {
    Selectionsort(out, in);
}

// or {
//    Bitonic(out, in);
//}
```

- Define multiple functions
- Functions have tunable variables
- PetaBricks choose optimal combination
PetaBricks effect

- Source code is compiled into a binary
- The binary is auto-tuned on the architecture of the system
- The compiler produces a final binary contains the optimal configuration
Tuned variables - Sample

<table>
<thead>
<tr>
<th>Line</th>
<th>Variable Name</th>
<th>Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Copy1D_sequential_cutoff</td>
<td>1455</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>2</td>
<td>Copy1D_split_size</td>
<td>1714</td>
<td>1 to 2147483647</td>
</tr>
<tr>
<td>3</td>
<td>Copy2D_sequential_cutoff</td>
<td>568</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>4</td>
<td>Copy2D_split_size</td>
<td>910</td>
<td>1 to 2147483647</td>
</tr>
<tr>
<td>5</td>
<td>InsertionsortSubArray_sequential_cutoff</td>
<td>0</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>6</td>
<td>Insertionsort_sequential_cutoff</td>
<td>0</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>7</td>
<td>Merge16_sequential_cutoff</td>
<td>1455</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>8</td>
<td>Merge2Parallel_Cutoff</td>
<td>949</td>
<td>100 to 100000</td>
</tr>
<tr>
<td>9</td>
<td>Merge2Parallel_sequential_cutoff</td>
<td>1373</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>10</td>
<td>Merge4_sequential_cutoff</td>
<td>1334</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>11</td>
<td>Merge8_sequential_cutoff</td>
<td>569</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>12</td>
<td>Mergesort16_sequential_cutoff</td>
<td>1497</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>13</td>
<td>Mergesort4_sequential_cutoff</td>
<td>1497</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>14</td>
<td>Mergesort8_sequential_cutoff</td>
<td>1458</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>15</td>
<td>MergeSortSubArray16_sequential_cutoff</td>
<td>1455</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>16</td>
<td>MergeSortSubArray4_sequential_cutoff</td>
<td>1497</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>17</td>
<td>MergeSortSubArray8_sequential_cutoff</td>
<td>1497</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>18</td>
<td>Parallel_MergesortSubArray_sequential_cutoff</td>
<td>945</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>19</td>
<td>Parallel_Mergesort_sequential_cutoff</td>
<td>1497</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>20</td>
<td>QuickSort_sequential_cutoff</td>
<td>1497</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>21</td>
<td>QuicksortSubArray_sequential_cutoff</td>
<td>949</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>22</td>
<td>RadixsortSubArray_sequential_cutoff</td>
<td>1714</td>
<td>0 to 2147483647</td>
</tr>
</tbody>
</table>
Figure 14. Performance for sort on 8 cores.

Figure taken from paper in review
PetaBricks Components Graph

Figure taken from paper in review
PetaBricks Internals – Source2Source Compiler

• Compiles from PetaBricks to C++
• Input parsed into syntax tree
• Construct a choice grid for matrix type
• Build a choice dependency graph
Compilation – 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;
}
}
```

Figures taken from paper in review
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;
    }
}
Compilation – Applicable Region

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;
}
}

Figures taken from paper in review
Compilation - Choice Grid Analysis

- Split the data into matrix (grids)
- Map data to rules
- E.g. $[0, 1) = \{\text{rule 0}\}$ only while $[1, n) = \{\text{rule 0 or 1}\}$
- Rule priority is applied here as well

Figures taken from paper in review
Compilation - Choice Dependency Graph

Figures taken from paper in review
Code Generation

• Two modes

  • Default – choices and autotuner information are embedded in the output code

  • Second mode – code generation with all choices eliminated based on autotuner results

  • Second mode is useful to produce an intermediate code for C++ to compile – it is more efficient when choices are eliminated
Auto Tuning System

• Tuning is done by running search on the available configurations

• The available configuration is described using the choice dependency graph

• Using bottom-up approach, works on smaller input and works it way up to large input
Runtime library

- Dynamically schedule tasks to distribute workload
- When task reach tunable cut-off point they stop calling the scheduler and execute sequentially
- Maximize locality using Cilk, task stealing protocol;
  - Thread operates on top of its dequeue
  - When it runs out of tasks
  - Select a random victim to steal work from bottom of their dequeue
PetaBricks – Other features

- Calling external libraries and other languages
- Template Transformation – Similar to C++
- Rule priorities and where clauses to manual tune edge cases
- Deadlock and race conditions prevention using the dependency graph
- Automated Consistency Checking - advantage of choices, you can run multiple versions and check their results for consistency
Evaluation - Performance

Figure 14. Performance for sort on 8 cores.

Figure taken from paper in review
Evaluation – Performance other algorithms

**Figure 11.** Performance for algorithms to solve Poisson’s equation up to an accuracy of $10^9$ using 8 cores. The iterated SOR algorithm uses the corresponding optimal weight $\omega_{opt}$ for each of the different input sizes.

**Figure 12.** Performance for Eigenproblem on 8 cores. “Cutoff 25” corresponds to the hard-coded hybrid algorithm found in LAPACK.
Evaluation – Parallelism

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.
POISSON$_i(x, b)$
1: either
2: Solve directly
3: Iterate using SOR$_{\omega_{opt}}$ until accuracy $p_i$ is achieved
4: For some $j$, iterate with MULTIGRID$_j$ until accuracy $p_i$ is achieved
5: end either
MULTIGRID$_i(x, b)$
1: if $N = 3$ then
2: Solve directly
3: else
4: Compute one iteration of SOR$_{1.15}$
5: Compute the residual and restrict to half resolution
6: On the coarser grid, call POISSON$_i$
7: Interpolate result and add correction term to current solution
8: Compute one iteration of SOR$_{1.15}$
9: end if

Figure 10. Pseudo code for family of functions POISSON$_i$ and MULTIGRID$_i$ where $i$ is the required accuracy, as used in the benchmark.
PetaBricks Today

- 363 citations according to Google Scholar
- Experiments and benchmarked continued for 4 years looking into tuning variables in algorithms, portability, and study the trade-offs between accuracy and performance.
- Main author (during his PhD), Jason Ansel, Director of engineering at GoDaddy since 2013
- GitHub repo, abandoned 6 years ago
- Ideas we can still use, with help of ML
  - Auto tuning between accuracy and performance
  - Auto tuning variables based on architecture
## Critique

<table>
<thead>
<tr>
<th>The Good</th>
<th>The Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>First language that explores algorithmic choice with impressive results</td>
<td>Slow compiling time – impractical for software engineering tasks (target success metric)</td>
</tr>
<tr>
<td>Ease of selecting the trade-off between accuracy and performance</td>
<td>Complex code structure – harder to adhoc debug</td>
</tr>
<tr>
<td>Portability across architectures and future proof</td>
<td>Principles of auto tuning using simple parameter search might be too slow</td>
</tr>
</tbody>
</table>
Comment – Working in a heterogenous environment

- Optimisation could be running a different algorithm choice on a different machine, storing the metadata of binaries in a key-value store, and binaries in a replicated store.

- Not every line of code has to be written in PetaBrick, just the lines that require high performance – e.g. a simple microservice that has no complex logic wouldn’t benefit from PetaBrick optimisation.

SWE commits changes

Changes are built on a remote machine identical to production in a container, auto tuner enabled

Produced binaries makes it way through the testing pipeline until production
Ideas for Future Work – GPUs?

- Extract the idea of choice and analysis per hardware architecture.
- Analyse the benefit of running algorithms on GPUs
- GPUs have different memory constraint than what traditional algorithms were designed for, auto tuning helps!
- Other work in this area already exist using ML for choosing GPU/CPU, but no (afaik) work exist to choose algorithm and tune it
Questions?

I HAVE NO IDEA WHAT I'M DOING