PetaBricks

Building adaptable and more efficient programs for the multi-core era is now within reach.

By Jason Ansel and Cy Chan

DOI: 10.1145/1836543.1836554

With the dawn of the multi-core era, programmers are being challenged to write code that performs well on an increasingly diverse array of architectures. A single program or library may be used on systems ranging in power from large servers with dozens or hundreds of cores to small single-core netbooks or mobile phones. A program may need to run efficiently both on architectures with many simple cores and on those with fewer monolithic cores. Some of the systems a program encounters might have GPU coprocessors, while others might not. Looking forward, processor designs such as asymmetric multi-core [3], with different types of cores on a single chip, will present an even greater challenge for programmers to utilize effectively.

Programmers often find they must make algorithmic changes to their program in order to get performance when moving between these different types of architectures. For example, when moving from a sequential to a parallel architecture, a programmer may have to change his or her algorithm to expose more parallelism.

A similar need for algorithmic change arises when switching between different types of parallel architectures. In a recent paper [1], we showed that even when changing between similar architectures with the same number of cores, dramatic algorithmic changes can be required to optimize performance.

To obtain portable performance in this new world of more diverse architectures, we must build programs that can adapt to whatever hardware platform they are currently running on. Our team of researchers at the Massachusetts Institute of Technology has developed PetaBricks [1], a new, implicitly parallel language and compiler that allows the user to specify algorithmic choices at the language level. Using this mechanism, PetaBricks programs define not a single algorithmic path, but a search space of possible paths. This flexibility allows our compiler to build programs that can automatically adapt, with empirical autotuning, to every architecture they encounter.

"PetaBricks programs define not a single algorithmic path, but a search space of possible paths. This flexibility allows our compiler to build programs that can automatically adapt, with empirical autotuning, to every architecture they encounter."

PETABRICKS LANGUAGE

The PetaBricks language provides a framework for the programmer to describe multiple ways of solving a problem while allowing the autotuner to determine which of those ways is best for the user's situation. It provides both algorithmic flexibility (multiple algorithmic choices) as well as coarse-grained code generation flexibility (synthesized outer control flow).

At the highest level, the programmer can specify a transform, which takes some number of inputs and produces some number of outputs. In this respect, the PetaBricks transform is like a function call in any common procedural language. The major difference with PetaBricks is that we allow the programmer to specify multiple pathways to convert the inputs to the outputs for each transform. Pathways are specified in a dataflow manner using a number of smaller building blocks called rules, which encode both the data dependencies of the rule and C++-like code that converts the rule's inputs to outputs. Dependencies are specified

Figure 1: Pseudocode for kmeans, an example of the PetaBricks program, is shown.

```
1  transform kmeans
2  from Points[n, 2]      // Array of points (each column
3          // stores x and y coordinates)
4  through Centroids[sqrt(n), 2]
5  to Assignments[n]
7  {
7    // Rule 1:
8    // One possible initial condition: Random
9    // set of points
10   to (Centroids . column ( i ) c) from ( Points p) {
11       c=p . column (rand ( 0, n ) )
12   }
13
14   //Rule 2:
15   //Another initial condition; Centerplus initial
16   //centers (kmeans++)
17   to ( Centroids c ) from ( Points p ) {
18   CenterPlus ( c , p) ;
19   }
20
21   //Rule 3:
22   //The kmeans iterative algorithm
23   to ( Assignments a ) from ( Points p , Centroids c ) {
24      while ( true ) {
25         int change ;
26         AssignClusters ( a , change , p , c , a ) ;
27         if ( change==0 ) return ; // Reached fixed point
28         NewClusterLocations ( c , p , a ) ;
29      }
30   }
31 }
```

"It is a daunting task to write a program that will perform well on not only today's architectures but also those of the future. We believe that PetaBricks can give programs the portable performance needed to increase their effective lifetimes."

by naming the inputs and outputs of each rule, but unlike in a traditional dataflow programming model, more than one rule can be defined to output the same data. Thus, the input dependences of a rule can be satisfied by the output of one or more rules.

It is up to the PetaBricks compiler and autotuner to decide (for a given architecture and input) which rule combination will most efficiently produce the transform output while satisfying all intermediate data dependencies. For example, the autotuner may find that it is preferable to use rules that minimize the critical path of the transform on parallel architectures, while rules with the lowest computational complexity may fare better on sequential architectures. The example in the following section will help further illustrate the PetaBricks language.

While it is helpful for a PetaBricks programmer to know beforehand what algorithms perform best under what circumstances, the framework can help those who do not in a different, and possibly more fundamental, way.

An expert may narrow the field of algorithmic choices to those he or she knows will perform well under a variety of circumstances, and then leverage PetaBricks to do the heavy lifting, tailoring their programs to their specific machine architectures. A non-expert with no such domain knowledge can take advantage of PetaBricks by enumerating several algorithms, and then learn from the autotuner what performs well under what circumstances.

We hope both types of users will find PetaBricks an indispensable tool in their programming arsenal.

EXAMPLE PROGRAM

Figure 1 presents an example PetaBricks program, kmeans. The program groups the input Points into a number of clusters and writes each point's cluster to the output Assignments. Internally the program uses the intermediate data Centroids to keep track of the current center of each cluster. The transform header declares each of the input (from), intermediate (through), and output (to) data structures. The rules contained in the body of the transform define the various pathways to construct the Assignments data from the initial Points data. The transform can be depicted using the dependence graph shown in **Figure 2**, which indicates the dependencies of each of the three rules.

The first two rules specify two different ways to initialize the Centroids data needed by the iterative kmeans solver in the third rule. Both of these rules require the Points input data. The third rule specifies how to produce the output Assignments using both the input Points and intermediate Centroids. Note that since the third rule depends on the output of either the first or second rule, the third rule cannot be executed until the intermediate data structure Centroids has been computed by one of the first two rules. The autotuner and compiler will make sure the program will satisfy these dependencies when producing tuned code.

Additionally, the first rule provides an example of how the autotuner can synthesize outer control flow. Instead of using a rule that explicitly loops over every column of Centroids 2D array, the programmer has specified a computation that is done for each column of the output. The order over which these columns are iterated is then synthesized and tuned by the compiler and autotuner. Columns may also be processed in parallel if dependencies are satisfied. Operations can be specified on a per-row or per-cell basis as well, allowing optimizations such as cache-blocking to be automatically discovered.

To summarize, when our transform is executed, the cluster centroids are initialized either by the first rule, which performs random initialization on a per-column basis with synthesized outer control flow, or the second rule, which calls the CenterPlus algorithm. Once Centroids is generated, the iterative algorithm in the third rule is called.

PETABRICKS COMPILER INFRASTRUCTURE

Figure 3 displays the general flow for the compilation of a PetaBricks transform. Compilation is split into two representations. The first representation operates at the rule level, and is similar to a traditional high-level sequential intermediate representation. The second representation operates at the transform level, and is responsible for managing choices and for code synthesis.

The main transform level represen-

Figure 2: In this dependency graph for kmeans example, the rules are the vertices while each edge represents the dependencies of each rule. Each edge color corresponds to each named data dependence in the pseudocode shown in Figure 1.

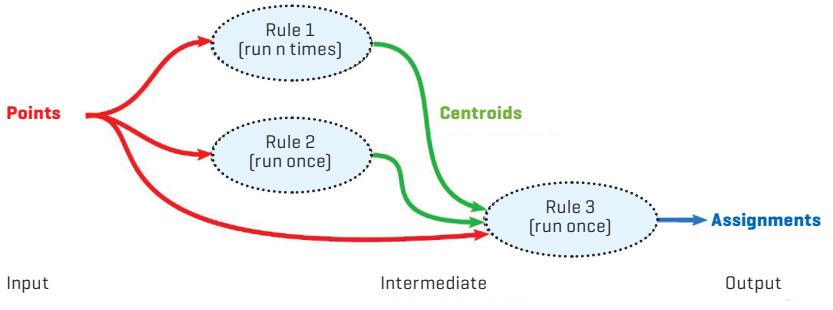
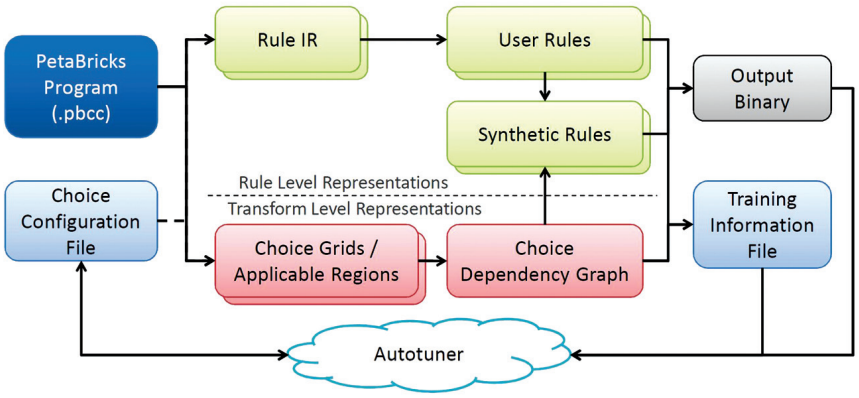


Figure 3: The flow for the compilation of a PetaBricks program with a single transform is shown. Additional transforms would cause the center part of the diagram to be duplicated.



tation is the choice dependency graph, which is the primary way that choices are represented in PetaBricks. At a high level, the information contained in the choice dependency graph is similar to the dependency graph shown for our example program in **Figure 2**. However, the data is represented as an "inverse" of that graph: data dependencies (previously represented by edges) are represented by vertices, while rules (previously represented by vertices) are represented by graph hyperedges. Additionally, data may be split into multiple vertices in the choice dependency graph if the transform contains rules that operate on just subregions of that data. The PetaBricks compiler uses this graph to manage code choices and to synthesize the outer control flow of the rules.

The final phase of compilation generates an output binary and a training information file containing static analysis information. These two outputs are used by the autotuner to search the space of possible algorithmic paths. Autotuning creates a choice configuration file, which can either be used by the output binary to run directly or can be fed back into the compiler to allow additional optimizations.

The autotuner follows a genetic algorithm approach to search through the available choice space. It maintains a population of candidate algorithms which it continually expands using a fixed set of high level mutators, which are generated through static code analysis. The autotuner then prunes the population in order to allow it to evolve more optimal algorithms. The input sizes used for testing during this process grow exponentially, which naturally exploits any optimal substructure inherent to most programs.

For more details on the PetaBricks language and compiler infrastructure please see our prior work **[1]**.

PERFORMANCE

In our prior work **[1]**, we implemented a suite of benchmarks to showcase the benefits of using our compiler and autotuner. First, we illustrate the impact of allowing greater algorithmic flexibility on computational performance. Figures 4-6 show the performance of our autotuned sort, eigenproblem, and 2D Poisson's equation solver algorithms, respectively, compared to implementations that utilize a single algorithmic choice. In all cases the autotuned algorithm obtained a significant speedup. These results were gathered on a 8-way (dual socket, quad core) Intel Xeon E7340 system running at 2.4 GHz using 64-bit CSAIL Debian 4.0 with Linux kernel 2.6.18 and GCC 4.1.2.

Note that in some cases, such as with our sort benchmark, the autotuned algorithm was able to significantly outperform the best available single algorithm. This is because our autotuned algorithm is able to leverage multiple algorithms during a single execution. Our autotuned sort program is able to switch from radix sort to merge sort to insertion sort during its recursive decomposition whenever the best algorithm for the current input size changes.

Second, we illustrate the benefits to program portability. Multi-core architectures have drastically increased the processor design space resulting in a large variety of processor designs currently on the market. Such variance can significantly hinder efforts to port performance critical code. We optimized our sort benchmark on three parallel architectures designed for a variety of purposes: Intel Core 2 Due mobile processor, Intel Xeon E7340 server processor, and the Sun Fire T200 Niagara low power, high throughput server processor.

Table 1 illustrates the benefits of targeted tuning on each architecture. We found that programs trained on architectures different from the ones on which they are run exhibit significant slowdowns. For example, even though they have the same number of cores, the autotuned configuration file from the Niagara machine results in a 2.35x loss of performance when used on the Xeon processor. On average we observed a slowdown of 1.68x across all of the systems we tested.

Table 2 displays the optimal configurations for the sort benchmark after running the same autotuning process on the three architectures. It is interesting to note the dramatic differences between the choice of algorithms, composition switching points, and scalability. The Intel architectures (with

"Multi-core architectures have drastically increased the processor design space resulting in a large variety of processor designs currently on the market."

Figure 4: Given a model of a DNA complex, the various kinds of reactions can be analyzed by a simulation.

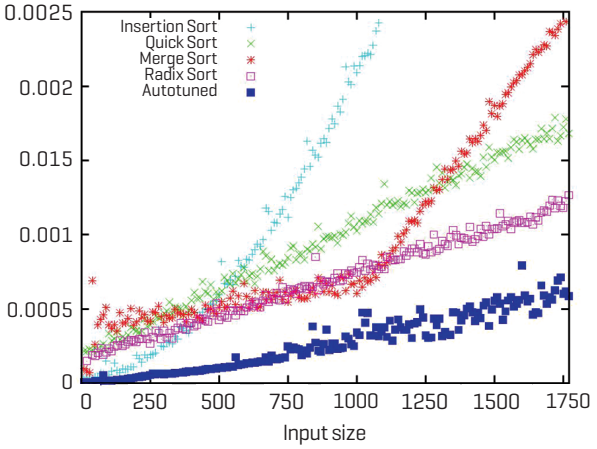


Figure 5: Differences in performance are shown for solving the eigenvalue problem using various algorithms compared to our autotuned PetaBricks algorithm. "Cutoff 25" corresponds to the hard-coded hybrid algorithm found in LAPACK.

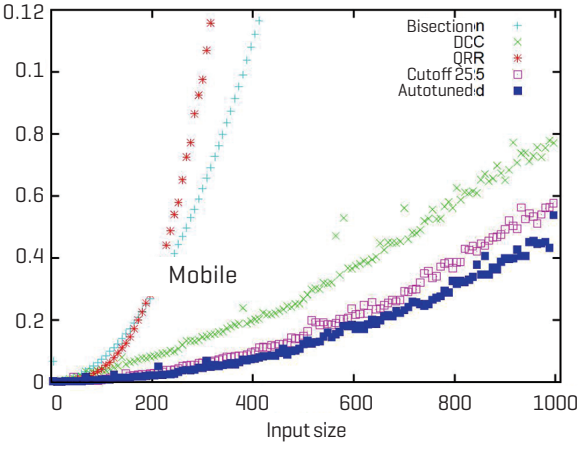
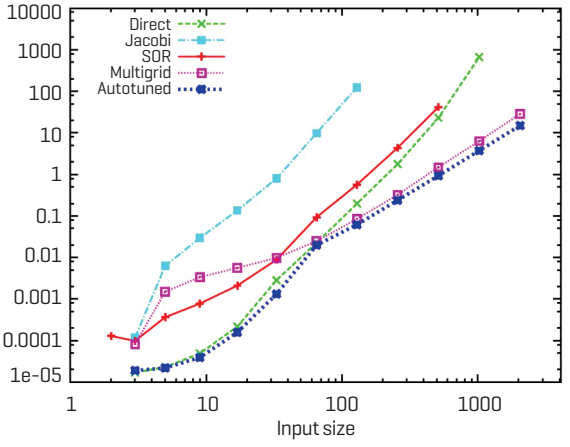


Figure 6: Differences in performance are shown for solving the 2D Poisson's equation up to an accuracy level of 10^9 using different algorithms compared to our autotuned PetaBricks algorithm.



"To obtain portable performance in this new world of more diverse architectures, we must build programs that can adapt to whatever hardware platform they are currently running on."

larger computation to communication ratios) appear to perform better when PetaBricks produces code with less parallelism, suggesting that the cost of communication often outweighs any benefits from running code containing fine-grained parallelism. On the other hand, the Sun Niagara processor performs best when executing code with lots of parallelism as shown by the exclusive use of recursive algorithms.

CURRENT AND FUTURE WORK

Variable accuracy. Different algorithmic choices available to the programmer often produce outputs of varying quality. We must understand the trade-off between computation time and accuracy in order to make the correct choices during autotuning. To expose these trade-offs to the compiler, we added simple language extensions to PetaBricks. With these extensions, our compiler can perform fully automatic compile-time and install-time autotuning and analysis to construct optimized algorithms for any user-specified target accuracy.

Dynamic choices. One facet of optimizing numerical algorithms we wish to explore is the selection of algorithms based on the numerical characteristics of the input, such as the condition number of a matrix, or the "pre-sortedness" of an array. We are exploring the use of run-time analysis for algorithm selection and parameter tuning that may make decisions based on deeper characteristics of the input in addition to size.

Heterogeneous architectures. One of the most difficult types of architecture to utilize is those with different

Table 1: Slowdown for sort benchmark on an input size of 100,000 can be seen when the program is tuned on a machine different from the one it is run on. Slowdowns are relative to training natively. Descriptions of abbreviated system names can be found in Table 2.

		Trained on			
		Mobile	Xeon 1-way	Xeon 8-way	Niagra
Run on	Mobile	–	1.09x	1.67x	1.47x
	Xeon 1-way	1.61x	–	2.08x	2.50x
	Xeon 8-way	1.59x	2.14x	–	2.35x
	Niagra	1.12x	1.51x	1.08x	–

types of cores in a single system. Examples of this include coprocessors such as GPGPUs and asymmetric multi-core chips [3]. We are expanding our runtime and autotuning infrastructure to support these types of processors so PetaBricks programs can effectively utilize heterogeneous architectures.

Distributed memory and clusters. An interesting future direction is support for a distributed memory back-end so that we can run unmodified PetaBricks programs on clusters. Moving to clusters will add even more choices for the compiler to analyze, as it must decide both what algorithms to use and where to run them. A key challenge in this area is autotuning the management of data.

RECOMMENDED FURTHER READING

There is a large variety of work related to PetaBricks' autotuning approach of optimizing programs. Please see our original PetaBricks paper [1] for a more detailed description of our framework and a more comprehensive listing of related work. For readers who are more applications-oriented, please see an in-depth example of our framework applied to a variable-accuracy multigrid solver [2].

PORTABILITY AND PERFORMANCE

Getting consistent, scalable, and portable performance is difficult. The compiler has the daunting task of selecting an effective optimization configuration from possibilities with drastically different impacts on the performance. No single choice of parameters can yield the best possible result as different algorithms may be required under different circumstances. The high performance computing community has always known that in many problem domains, the best sequential algorithm is different from the best parallel algorithm. Varying problem size and data sets will also require different algorithms. Currently there is no viable way for incorporating all these algorithmic choices into a single program to produce portable programs with consistently high performance.

We have introduced the first language that allows programmers to naturally express algorithmic choice explicitly so as to empower the compiler to perform deeper optimization. We have created a compiler and an autotuner that is not only able to compose a complex program using fine-grained algorithmic choices but also find the right choice for many other parameters.

Trends show us that programs have a lifetime running into decades while architectures are much shorter lived. With the advent of multi-core processors, architectures are experiencing drastic changes at an even faster rate. Under these circumstances, it is a daunting task to write a program that will perform well on not only today's architectures but also those of the future. We believe that PetaBricks can give programs the portable performance needed to increase their effective lifetimes.

Biographies

Jason Ansel and Cy Chan are PhD candidates in the Computer Science and Artificial Intelligence Laboratory at MIT. Ansel is advised by Professor Saman Amarasinghe and Chan is advised by Professor Alan Edelman.

References

1. Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. 2009. PetaBricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland.
2. Chan, C., Ansel, J., Wong, Y. L., Amarasinghe, S., and Edelman, A. 2009. Autotuning multigrid with PetaBricks. In *ACM/IEEE Conference on Supercomputing*, Portland, Ore.
3. Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. 2005. Heterogeneous chip multiprocessors. *Computer, 38*:32–38, 2005.

Table 2: The table shows the automatically tuned configuration settings for the sort benchmark on various architectures.

Abbreviation	System	Frequency	Cores Used	Scalability	Algorithm Choices (with switching points)
Mobile	Core 2 Duo Mobile	1.6 GHz	2 of 2	1.92	IS (150) 8MS (600) 4MS (1295) 2MS (38400) QS(∞)
Xeon 1-way	Xeon E7340 (2x4 core)	2.4 GHz	1 of8	–	IS (75) 4MS (98) RS(∞)
Xeon 8-way	Xeon E7340 (2x4 core)	2.4 GHz	8 of 8	5.69	IS (600) QS (1420) 2MS (∞)
Niagara	Sun Fire T200 Niagra	1.2 GHz	8 of 8	7.79	16MS (75) 8MS (1461) 4MS (2400) 2MS (∞)

Abbreviations

IS	insertion sort	4MS	4-way merge sort
QS	quick sort	2MS	2-way merge sort, with recursive merge that can be parallelized.
RS	radix sort		
16MS	16-way merge sort		
8MS	8-way merge sort		

The numbers in parentheses next to the algorithm indicate the upper bound on the range of input sizes handled by that algorithm; for example, for Xeon 1-way, insertion sort is used for sizes up to 75, 4MS for 76 to 98, and RS for 99 and larger. Multiple algorithms are used as the problem is recursively decomposed into smaller and smaller sub-problems.