Summary:

The paper proposes an approach called *PetaBricks* providing a new implicitly parallel language including a compiler and run-time framework, which makes it easy to implement multiple algorithms for a given problem to compute a solution very efficiently.
PetaBricks is especially useful in case of algorithmic compositions such as common sort algorithms. As example the `std::sort` implementation of the C++ STL is given, which defaults to a merge sort unless the list is smaller than a fixed threshold, in which case insertion sort is performed. Although this threshold has been hardcoded into the library, in practice it varies depending on certain hardware and input characteristics. Therefore, the value has to be adapted ("auto-tuned") to these characteristics. In general, the authors call this a cutoff point, which describes the point at which the execution of one algorithm should be continued by a better performing algorithm for the given situation. Assuming that PetaBricks is given a sufficient detailed description of the algorithm, these cutoff points can be determined automatically.
Additionally, PetaBricks supports even the generation of parallel output code from the given algorithmic description, since the PetaBricks language only models a detailed description of what the algorithm does and leaves the choice how to the compiler and run-time system.
The PetaBricks language mainly consists out of two major constructs: *transforms* and *rules*. The transform is introduced with a header consisting out of *to*, *from* and *through* arguments. These represent inputs, outputs and intermediate data used within the transform. The transform itself defines the algorithm, which can be invoked from other compatible transforms. The transform construct therefore consists out of rules, which encode different algorithmic choices for the algorithm. Thereby defines each rule, how the given state can processed in order to archive a solution. The compiler now tries to find the optimal sequence of applications of rules for the given transforms, that finally compute the output of the program. As mentioned earlier the explicit rule dependencies allow automatic parallelization and automatic detection of auto-tunables such as cut-off points.

Summary A4

Like the approach presented in last week's paper the one we discuss this week is about PetaBricks which includes a language and a compiler for automatic algorithm choice. Furthermore it is able to detect parallelism in these algorithms and exploit it.
Since algorithms mostly work fine for certain input ranges and loose performance for others it is convenient to use different algorithms for different input ranges. The problem with connecting several versions of an algorithm by hand is on the one hand that the programmer has to find out the points where to change the currently used one and on the other hand that these values depend on the architecture the program is executed on so that they cannot be optimal when defined once in the code.
PetaBricks solves this problem by a language providing constructs to enable an autotuning mechanism that optimizes these parameters. There is the "transform"- construct that defines algorithms which provide several implementations and can be optimized. Its header consists of a "to", "from" and a "through" argument representing the input, output and intermediate data. Within a transform several "rules" can be defined to encode the algorithm choice. These rules also have a "to" and a "from" field in their header and contain dependencies. Additionally they have a "where" field to limit where the rule can be applied. So the compiler has to find a sequence of rules which are able provide the complete program output.
Besides the compiler PetaBricks includes the already mentioned autotuning system. This system uses the encoded information about choices and tunable parameters the compiler added to the generated code and produces a so called application configuration file which includes the choices the system made.
As a benchmark the authors ran the poisson function, the eigenproblem, sorting algorithms and matrix multiplication algortihms on the PetaBricks. The system always chose the fastest implementation for every configuration and the autotuning could even improve the speedup.

Questions / Opinion:

1. In paragraph 3.2 the authors mention that code generated for dynamic scheduling incurs overhead. Why is that the case and why can it be solved by generating a second version which is a sequential one for the leaves of the execution tree?

1 Summary

Finding an optimal solution to a problem is generally not possible. The performance depends on parameters such as the architecture, input data, parallelism, etc. Furthermore, the complex nature of code often constraints the compiler such that algorithm composition or selection is not possible anymore.
In this paper, the authors present a language and a source-to-source compiler to automate algorithm selection and tuning of cutoffs, called PetaBricks. PetaBricks is an implicitly parallel programming language, that allows natural encoding of multiple algorithms into a program, and takes the burden of algorithm selection from the programmer and hoists it to the compiler. By providing multiple algorithms to solve (parts of) a problem, the developer gives the choice of algorithm selection and composition to the compiler. PetaBricks is the first language where choice is provided at language level. Since algorithm composition relies on the compiler, the compiler can also automatically detect the optimal switch over points between the algorithms.
PetaBricks introduces *transforms* and *rules*. A transform is similar to a function in C++: it has a header consisting of *from*, *to*, and *through* arguments, and a body containing rules. A rule describes how to compute (part of) the output given the corresponding inputs. The body of a rule is written in C++-like code. The more rules a transform has, the more choices are exposed to the compiler. PetaBricks can generate two types of code: 1) choices and autotuning are embedded in the binary, such that the program can be tuned, generating a configuration file and 2) a configuration file is used to statically select choices and cutoffs.
PetaBricks attaches an autotuning library to the binary if necessary. The PetaBricks runtime library contains a scheduler to manage parallelism, and is responsible for data and the configuration.
In their evaluation, the authors use various benchmarks they implemented and that are relevant in scientific and computing kernels. On all benchmarks their approaches score the best performance overall.

2 Questions & Opinions

If the PetaBricks compiler automatically parallelizes a program, it has to make sure concurrent tasks don't interfere. When thinking of dynamic programming, parallelizing becomes difficult. As the importance of the *through* part in a transform header is not elaborated, might this be used to tell the compiler where tasks may share memory and interfere? If not, what is the *through* keyword used for?

Their approach seems to fit well to recursive algorithms, but is this also applicable for more complicated algorithms, performing on complex data structures, and having more complicated control flow?

PetaBricks: A Language and Compiler for Algorithmic Choice

1. Summary

This paper discusses a new compiler and language called petabricks that relieves programmer from having to decide which algorithm should be plugged in for which kind of input and platform. Instead, programmer specifies different algorithms to compute output, and petabricks compiler and runtime auto-tunes the program for best parallel performance by deciding switch points between algorithms.

Algorithmic choices are provided by the programmer using constructs called rules inside transforms. Transforms are equivalent to functions. User can also embed normal C++ code inside the rules. User can also explicitly specify some rules using where clause. There is also a construct named tunable which can be used to provide parameters for auto-tuning.

Petabricks compiler calculates dependencies, removes redundant rules from the transforms. It also computes switch points between different algorithms. Runtime includes a scheduler that takes the dependency graph, and schedules task in depth first manner in order to provide for maximum parallelism. Light weight thread migration is also supported using continuation points at which thread data is spilled into heap. Auto-tuner is used to decide thresholds and switch points between rules for different platforms. Auto-tuner uses a data structure called choice dependency graph to do this.

A nice side-effect of being able to specify multiple algorithms is to check correctness of them by checking output consistency for different set of inputs. Finally, in order to avoid deadlocks between tasks, system analyses the dependency graph for cycles.

Benchmarks of different kinds are used for evaluation. For each of these benchmarks, petabricks system performs better than any implementation of single algorithm.

2. Open Questions

1. Is depth first search scheduling of tasks in the dependency graph is better than any other kind of scheduling of tasks?
2. Is application region finding is contained in choice grid analysis phase? What is the exact distinction between them?

Summary PetaBricks

The paper introduces PetaBricks, a language and runtime which allows the programmer to express algorithmical choices and which automatically finds the best fitting algorithm for a specific architecture.

The authors motivate their language by stating that composed algorithms are already implemented by hand. However, this is deemed suboptimal by the authors of the paper, as the optimal point where algorithms should be switched varies from one system to another. Furthermore, even finding one optimal toggle point can be far from trivial. To migitiate this issue, PetaBricks has language features to express algorithmic choice: rules and transforms. Transforms are quite similiar to functions, except that they have to, from and through keywords which are used to explicitly state inputs, outputs and "intermediate data". Rules are used to express how some subset of data is computed (the actual compute part is done in a C++ subset). They also feature to and from keywords. Thanks to this explicit stating of dependencies, the compiler is able to automatically parallelize rules when it tries to find a sequence of rules to compute the acutal output. The actual compilation works as follows: after the AST is built, dependencies are normalized. Then, for each rule, the data regions to which they are applicable are computed (using linear algebra).Those regions are then split into a rectilienear grid, where for each cell a uniform set of rules is applicable. Using those, a data dependency graph between them is computed. This graph is then used for the actual code generation. The runtime system features a scheduler for automatic parallelization; for more efficient parallelism it uses work stealing. Furthermore, the runtime has an auto-tuning system; it works bottom-up, first tuning the small subproblems and then going up one level at a time. It seems to use genetic programming to find an optimal solution.

The authors evaluate on 3 different problems: Solving Poisson's equation, symmetric eigenproblem and sorting. They can achieve speedups compared to using only one algorithm, or a mix of algorithms with a hard coded switch point.

Open Questions

- Why exactly have PetaBrick's transforms an optional "through"? Could one not use to instead of it?
- What is a hpyer-quadrant in the context of graphs?
- What is the symbolic center to which dependencies are relative to?
- How useful is the system for non-recursive problems?

PetaBricks: A Language and Compiler for Algorithmic Choice

1 Summary

The paper presents the language PetaBricks which is tailored to allow automatic parallelization and adaptive choice between algorithms.
The language introduces several new constructs, described of these are mainly transforms and rules:

Transforms are basically problem definitions, they declare the input and output data of the function that the program computes. A transform may contain several rules to implement choice.

Rules are algorithm specifications that perform the computation for a specified region of data. Every rule can specify which part of the output it generates and which inputs it needs, additionally rules can specify on which data regions they are applicable.

The compiler will use this information to build a choice grid, specifying when a rule is applicable. Using this choice grid the compiler will then build a rule dependency graph, defining possible choices based on data region and ordering information for rule application.

The output of the compiler is a autotuning program that will perform a calibration to determine the specific rule choices based on the rule dependency graph. This calibration is implemented using a genetic algorithm which will feed the transform with input data to train the population of rule choices. Besides choices between rules the tuning is also able to identify rules that can be executed in parallel using the rule dependence graph.

The output of the calibration is a choice configuration file which encodes the best choices between rules obtained by the genetic algorithm. This file can either be input to the runtime application to execute the program with the obtained configuration or it can be feedback into the compiler to produce a final binary with all choices hardcoded.

2 Questions

- How do they generate training input for their genetic tuning?
- Is parallelization included in their genetic tuning or is it performed afterwards?