Data Parallel Dialect of Scheme

Outline of the Formal Model, Implementation, and Performance

Petr Krajca
SUNY Binghamton, Watson School
Binghamton, NY 13902, U. S. A.
+1 (607) 777 5690
petr.krajca@binghamton.edu

Vilem Vychodil
SUNY Binghamton, Watson School
Binghamton, NY 13902, U. S. A.
+1 (607) 777 5690
vychodil@binghamton.edu

ABSTRACT

We outline new functional programming language and stack-based model of implicit parallel execution of programs. The language, called Schemik, is high-level lexically-scoped implicitly-parallel dialect of Scheme. Schemik is designed as an implicitly parallel language, meaning that the parallel execution of programs is done independently of the programmer and each program written in Schemik always produces the same results no matter which parts of the program are executed simultaneously. The execution of programs is formally described by transitions of a particular pushdown automaton working with two stacks. Because of the limited scope of this paper, we just outline key ideas and postpone detailed description to a full version of this paper.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms

Design, Experimentation, Languages, Performance, Theory

Keywords

Data parallelism, implicit parallelism, functional programming, Scheme, pushdown automata.

1. PROBLEM SETTING

This paper contributes to the design and implementation of implicit parallel programming languages. There are differences in how programming languages support parallel computing. The common approach is to equip a well-established programming language with additional constructs or libraries which allow for threading and provide essential synchronization means. This approach can be called an *explicit parallelism.* Another way to approach parallel execution is to provide a language which is used the same way as any serial language but its execution can be implicitly parallelized by a compiler or an interpreter of the language. We will call this approach the *implicit parallelism* or the *data parallelism.* The implicit approach to parallelism is quite rare, several Fortran compilers or a recent paper [1] are some of the few exceptions.

When considering implicit parallel languages, we must pay attention to the synchronization issues at the level of compilation or interpretation of programs which should be taken care of for the programmer. In an ideal case, the serial execution of any program should give the same result as any possible parallel execution of it. An interest in implicit parallelism is anticipated as the major processor manufacturers have switched their focus from increasing performance by scaling clock speed to developing processors with multiple cores. This shift is, in fact, a hardware paradigm shift which will likely affect software engineering as well. Naturally, software developers will look for methods simplifying the development of parallel programs taking advantage of multiple processor cores. Wen Mei Hwu et al. [4] even argue that the explicit parallelism will be counterproductive for the vast majority of programmers in the long run.

We introduce a new implicit-parallel programming language called Schemik. The language has higher-order functions, macros, continuations as first-class objects, and other features. Our approach to parallel evaluation is unique in way in which we express the execution (evaluation) of programs. It is expressed by means of particular pushdown automata working with two stacks. During the parallel execution, multiple independent evaluators with their own pairs of stacks may appear to evaluate different subexpressions simultaneously. New evaluators appear as parallel branches of current computations. When consistency is ensured, different evaluators may be merged together. This branching and merging of evaluators represent concurrent threads of execution and synchronization. We argue that Schemik is both a new formal model and a practical programming language for which we have an implementation. For a programmer, Schemik is a language with the ideal type of implicit parallelism where programs always return the same values as if they were executed the usual serial way. Let us note at this point that Schemik is not a pure functional language but it allows for side effects.

The approaches to parallel evaluation proposed so far focused almost exclusively on the explicit parallelism. An automated parallelism in Scheme is proposed in [2]. Another paper which deals with implicit parallelization is [5] which is close to our approach. However, our approach is more concrete in that we deal with much more precisely specified language and deal with issues which are not shown in [5]. Recently, the implicit parallelism and data-driven parallelism seem to be the subject of increasing interest. For instance, [1] discusses a data-parallel Haskell.

2. FORMAL MODEL

Our parallel evaluator can be seen as a particular pushdown automaton working with two stacks: (1) *execution stack* containing stack operations controlling the evaluation, and (2) *result stack* containing objects which play the roles of operands and intermediate results of operations. The parallelization of computing is done by creating multiple independent evaluators computing particular subexpressions. The evaluators form a tree structure analogous to the structure of processes in UNIX-like operating systems. A single (main) evaluator is always at the top of the hierarchy. The input (first expression to be evaluated) is encoded on execution stack of the main evaluator and the output (result of evaluation) is pushed on its result stack at the end of the computation. Setting of the stacks represents a configuration of the evaluator. The stack operations which may appear on the execution stack are represented by tuples containing a name of the operation and additional arguments needed to perform the operation (e.g., a link to lexical environment in which the operation is performed). During the computation, the automaton makes transitions from one configuration to another based on the stack operation which resides on the top of the execution stack. For instance, we consider the following stack operations: EVAL (initiates evaluation of given subexpression), INSPECT (controls the order of evaluation of arguments), and FUNCALL (performs function application).

Parallel Evaluation. A new independent evaluator is created as a result of a special stack operation called FEVAL. This operation prepares an expression for evaluation in an independent evaluator, i.e., using a new pair of stacks. Operation FEVAL is specific in that it is not a result of any transition caused by another stack operation. The appearance of FEVAL on any execution stack is caused by external entity called *scheduler* that converts an expression found on an execution stack to the FEVAL operation and creates a new evaluator containing the original expression on the top of its execution stack. This way a parallel branch of the evaluation is created. Invocation of the FEVAL represents merging of two branches of the evaluation. If FEVAL appears on the top of the execution stack, the evaluation in the referred independent evaluator is stopped and its stacks are appended to the corresponding stacks of the current evaluator processing the FEVAL operation. This is the basic idea we use to create and merge together independent branches of computation.

Dealing with Side Effects. Schemik solves inconsistencies related to operations with side effects by proper ordering of critical operations. In order to ensure unique results, we define a specific order in which all operations with side effects must be performed. We use the "left-to-right order" which postulates that any operation with side effects to the left must be performed before all operations with side effects to the right. To ensure a proper order of operations, we introduce a stack operation WAIT which supersedes evaluation of expressions until they are merged with the main evaluator. Special operations of assignment and mutation must use further means of synchronization to ensure correct results when reading data from modified memory. An additional stack operation DROP is used to drop out all parallel branches that may be affected by any assignment or mutation of data structures. Let us note that these two synchronization operations are sufficient to implement sound escape functions and continuations in our model.

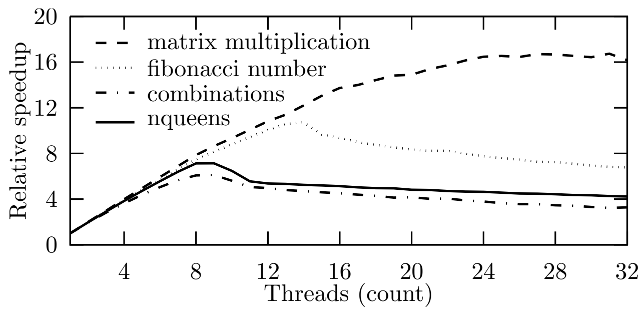

Figure 1: Scalability of various functions

3. IMPLEMENTATION

We have implemented our evaluation model as a small and portable interpreter of Schemik to inspect properties of the evaluation model in the real environment of contemporary computers. To express the benefits of parallel evaluation we are using relative speedup: a ratio of time the program is running in a sequential mode and the time the program is running using multiple evaluators. Figure 1 shows results of some of our experiments which were done on an eight-core UltraSPARC T1 Niagara where each of the eight cores is capable of processing four threads. Even if our implementation is in an early stage of development, it is competitive with production-quality Scheme interpreters like Guile Scheme.

Acknowledgments. The authors' second affiliation is Dept. Computer Science, Palacky University, Olomouc, Czech Republic. Supported by research plan MSM 6198959214.

4. REFERENCES

[1] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *DAMP*, 10–18, 2007.

[2] Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(2):179–396, 1989.

[3] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[4] Wen Mei Hwu, et al. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, 754–759, New York, NY, USA, 2007. ACM.

[5] Jenn-Jong Yee and Chung-Kwong Yuen. Transputer-based emulation of a data-driven LISP machine: BIDDLE. *Journal of Systems and Software*, 23(1):51–63, 1993.