A Library of basic PRAM Algorithms and its Implementation in FORK

Christoph W. Keßler
FB 4 Informatik, Universität Trier
D-54286 Trier, Germany
kessler@psi.uni-trier.de

Jesper Larsson Träff*
Max–Planck–Institut für Informatik
D-66123 Saarbrücken, Germany
traff@mpi-sb.mpg.de

Abstract

A library, called PAD, of basic parallel algorithms and data structures for the PRAM is currently being implemented using the PRAM programming language Fork95. Main motivations of the PAD project is to study the PRAM as a practical programming model, and to provide an organized collection of basic PRAM algorithms for the SB-PRAM under completion at the University of Saarbrücken. We give a brief survey of Fork95, and describe the main components of PAD. Finally we report on the status of the language and library and discuss further developments.

1 Introduction

For the past 15 years the PRAM has been the primary theoretical model for the design of parallel algorithms. By now, a very considerable collection of concrete algorithms for a variety of problems is available, complemented by broadly applicable design paradigms and techniques [9]. In principle a significant part of this knowledge can be made useful in practice by concrete implementations of basic algorithms and language support for relevant design paradigms. In order to establish the PRAM (or closely related model) as a *practical* parallel programming model, such an undertaking is inevitable. Although it is not dependent on a particular physical realization of the PRAM model, the effort reported here is immediately applicable to the SB-PRAM [1], an emulation in hardware of a Priority CRCW PRAM being built by W. J. Paul's group at the University of Saarbrücken.

We discuss a project investigating the PRAM as a practical programming model. The components of the project are a general-purpose PRAM programming language, called *Fork95*, and a library, called *PAD*, of basic *PRAM Algorithms and Data structures*. Both efforts focus on efficiency: The programming language must be adequate for the implementation of parallel algorithms as found in the literature and efficiently compilable; the library should support easy implementation of more involved algorithms, and include the most efficient algorithms in terms of parallel work, as well as constants. For a more comprehensive treatment of Fork95, see [11, 12, 13]; for PAD, see [16, 17, 18]. For current status, see `http://www-wjp.cs.uni-sb.de/fork95` and `http://www.mpi-sb.mpg.de/guide/activities/alcom-it/PAD`.

*This author was supported by DFG, SFB 124-D6, VLSI Entwurfsmethoden und Parallelität. This work was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT).

2 Fork95: A general purpose programming language for PRAMs

Fork95 is an explicit PRAM programming language, and hence concepts like processors, processor ID's, shared memory, and synchronicity are explicit to the programmer. Fork95 grew out of a proposal [6] for a strictly synchronous language for PRAM programming, but has instead been based on C, from which it inherits features like pointers, dynamic arrays, and structured data types. The choice of C together with carefully chosen defaults makes it possible to reuse existing sequential code. We introduced an asynchronous mode of computation (as the default mode), which allows to save synchronization points and enables more freedom of choice for the programming model. For efficiency reasons we have abandoned virtual processor emulation by limiting the number of processors to the hardware resources, resulting in efficient code generation and small run time overhead. Fork95 is intended for medium-level PRAM programming, and can immediately be used with the SB-PRAM [1]. However, the language should not be thought of as limited to this machine. In the following we give an overview of the essential features of Fork95.

Shared and private variables The shared memory of the PRAM is statically partitioned into a shared address subspace, and private address subspaces for each processor. Accordingly, variables are classified as either private (`pr`, the default) or as shared (`sh`), where "shared" is relative to the group of processors that declared that variable. For each physical processor there is a special private variable `$` which is initially set to its processor ID, and a special variable `@` shared by all processors belonging to the same group. `@` holds the current group ID, and `$` the current group-relative processor ID. These variables are automatically saved and restored at group forming operations; however, the user is responsible for assigning reasonable values to them. For handling concurrent read/write operations, Fork95 inherits the conflict resolution mechanism of the target machine. The use of pointers in Fork95 is as flexible as in C, since the private address subspaces are embedded into the global shared memory of the PRAM. In particular, one does not have to distinguish between pointers to shared and pointers to private objects, in contrast to e.g. C^* [7].

Synchronous and asynchronous regions Fork95 provides synchronous and asynchronous programming modes, statically associated with program regions and functions. The *farm* statement `farm <stmt>` designates asynchronous mode and reinstalls synchronous mode at the end by barrier synchronization. The `join` statement [13] designates synchronous mode, initiated by barrier synchronization. A common special case of the `join` construct is the *start* statement, `start <stmt>` which collects *all* available processors to enter the synchronous region `<stmt>`. Synchronous functions can only be called from synchronous regions. Calling asynchronous functions is possible from both asynchronous and synchronous regions; in a synchronous region the call is implicitly embedded in a `farm` statement.

The group concept In synchronous mode, the processors are partitioned into independent groups. Shared variables and objects exist once for the group that created them; global shared variables are accessible to all processors. The processors within an (active) group operate strictly synchronously (at the statement level). As long as control flow depends only on "shared" conditions evaluating to the same value on each processor, synchronicity within the group is preserved. If control flow diverges due to e.g. an `if` condition depending on private variables the group is deactivated and split into new subgroups in order to preserve synchronizity within the subgroups. The subgroups are active until control flow reunifies again. After barrier synchronization synchronicity is reestablished in the reactivated parent group. Thus, at any point of program execution the processor groups form a tree structure with the root group consisting of all started processors, and the leaf groups being the currently active groups.

An active group can also be split explicitly by the *fork* statement `fork(<exp1>; @=<exp2>; $=<exp3>) <stmt>` which evaluates the shared expression `<exp1>` and splits the current leaf group into that many subgroups. Each processor evaluates `<exp2>` to determine which of the new subgroups to join. The assignment to the group relative processor ID `$` allows to locally renumber `$` inside each new subgroup. Synchronous execution in the parent group is restored by barrier synchronization when all subgroups have finished their execution of `<stmt>`.

In asynchronous mode the group hierarchy is still visible, but no splitting of groups at conditional statements is done because statement level synchronicity is not enforced. The `join` statement [13] allows a processor to store and leave its old group hierarchy and join a new root group.

Multiprefix operations Fork95 has powerful atomic multiprefix operators, inherited from the SB-PRAM. For instance, the expression `mpadd(&shvar,<exp>)` atomically adds the (private) integer value of `<exp>` to the shared integer variable `shvar` and returns the old value of `shvar`. In synchronous mode the processor with the ith–largest physical processor ID participating in the execution of `mpadd(&shvar, <exp>)` receives the (private) value $s_0 + e_0 + e_1 + \ldots + e_{i-1}$, where s_0 is the value of `shvar` prior to the execution, and e_i the value of `<exp>` for the processor with `$` $= i$. After execution `shvar` contains the global sum $s_0 + \sum_j e_j$.

Related work NESL [2] is a functional dataparallel language partly based on ML. Its main data structure is the (multidimensional) list. Elementwise operations on lists are converted to vector instructions for execution on SIMD machines. In contrast, the MIMD-oriented Fork95 also allows for asynchronous and task parallelism, low-level PRAM programming and direct shared memory access. Dataparallel variants of Modula, e.g. [15], support a subset of Fork95's functionality. The main parallel constructs are synchronous and asynchronous parallel loops; there is no group concept. Other PRAM oriented dataparallel languages are *Dataparallel C* and C^* [7].

3 PAD: A library of basic PRAM algorithms

PAD is an attempt to provide support for the implementation of parallel algorithms as found in the current theoretical literature by providing access to some of the ubiquitous basic PRAM algorithms and computational paradigms like prefix sums, list ranking, tree contraction, sorting etc. PAD provides a set of abstract parallel data types like arrays, lists, trees, graphs, dictionaries, and is organized as a set of procedures which operate on objects of these types. However, the user of the library is responsible for ensuring correct use of operations on data objects, since the C based Fork95 does not support abstract data types directly. Computational paradigms, e.g. prefix sums over an array of arbitrary base type using a given associative function are provided for by procedures with type information and procedure parameters. The standard operations in many cases have certain "canonical" instances, e.g. prefix sums for integer arrays. For efficiency reasons both general and specialized versions are supplied in such cases. Fork95 does not support virtual processing. PAD compensates for this by having its procedures implicitly parametrized by the number of executing processors. The PAD implementation of an algorithm running in time t performing work w runs in $O(t+w/p)$ time when called within a synchronous group of p processors. Furthermore a "poor man's virtual processing" is provided for by *parallel iterators* for the supported data types. In the following we briefly discuss the abstract data types contained in the first version of PAD.

The prefix library The prefix library contains basic operations for the array data type, mainly of the "prefix-sums" kind. Operations like computation of all prefix sums, total sum, prefix sums by groups etc. for arrays over arbitrary base types with a user specified associative function are provided. Using the built-in multiprefix instructions of the SB-PRAM, prefix sums for integer arrays of length n can be computed in $O(n/p)$ time. For arrays over an arbitrary base type with some given associative function the running time is $O(\log n + n/p)$ with a somewhat larger constant.

The merge library The merge library provides the important array operations of parallel searching and merging on ordered arrays, and sorting of arrays. The PAD merge procedure is an implementation of the CREW algorithm given in [5], which runs cost-optimally in $O((\log n + n/p)$ time, where n is the total length of the input arrays. Experiments show that the implementation is very efficient when compared to a "reasonable" sequential merge procedure, i.e. the running time of the parallel algorithm with one processor roughly equals the running time of the sequential implementation, and the speed-up is close to perfect [18]. A clever trick in [5] makes it easy to implement a work-optimal parallel merge sort algorithm, running in $O(\log^2 n + n \log n/p)$ time, by using the general PAD merge procedure and the possibility to define a new, special (2-component lexicographic) ordering. The merge library also provides a Quicksort implementation, and a very efficient sorting procedure for small integers [4].

Parallel lists The *parallel list* data type gives direct access to the elements of an ordinary linked list, and is represented as an array of pointers to the elements of the list. The primary operation on parallel lists is *ranking*, that is, determining for each list element its distance from the end of the list. The parallel list type contains the necessary fields for the list ranking operations. Currently only a simple, non-optimal list ranking operation based on pointer jumping is implemented [9]. Other operations on lists include catenation, permutation into rank order and others.

Trees Trees in PAD are represented by an array of edges and an array of nodes, with edges directed from the same node forming a segment of the edge array. An edge is represented by pointers to its tail and head nodes, a pointer to its reverse edge, and has a "next edge" pointer used for representing Euler tours. Nodes have parent and sibling pointers for representing rooted trees. The library provides operations which allow a single processor to access and manipulate nodes and edges, as well as collective, parallel operations on trees. Parallel operations on trees include computing an Euler tour, rooting a tree, computing pre- and post-order traversal number, level numbers etc. PAD also supports least common ancestor preprocessing and querying. The currently implemented procedures are based on the reduction to the range query problem, see [9]. In the current implementation preprocessing takes $O(\log n+n\log n/p)$ time (non-optimal), and processors can then answer least common ancestor queries in constant time. Other parallel operations on trees include procedures for generic tree contraction, see also [9].

Graphs A data type for directed graphs similar to the tree data type is defined. Parallel operations (not yet implemented) will include finding the connected components, and extracting a (minimum) spanning tree [3, 9, 10].

Parallel dictionaries Currently PAD includes one non-trivial parallel data structure, namely a parallel dictionary based on 2-3 trees [14]. Dictionaries can be defined over base types ordered by an integer key. A parallel dictionary constructor makes it possible to build a dictionary from an ordered array of dictionary items. Dictionary operations include parallel searching and parallel (pipelined) insertion and deletion. Dictionaries can also maintain the value of some associative function, and provide a generic search function, which makes it possible to define the extra dictionary operations used in the parallel 2-3 partial sums tree data structure [19]. [16] presents a full implementation with initial measurements. The experiments indicate that parallel incremental operations on 2-3 trees are expensive in comparison to building/destroying a dictionary as a whole.

Related work Most of the basic algorithms considered for PAD has previously been implemented in the much more mature NESL project [2]. A main difference is NESLs use of lists, where PAD offers a broader selection of more traditional data structures. NESL is targeted towards existing platforms, where PAD presupposes a (virtual) PRAM, and can probably in this context be more efficient. Concrete implementations of many PRAM graph algorithms on the MasPar, were discussed in [8].

4 Status and future work

A compiler for Fork95 together with system software and a simulator for the SB-PRAM is already available, while a first version of PAD will be released in the summer of 1996. The next phase of PAD will extend the basic library in order to implement more advanced graph and combinatorial algorithms, like graph decompositions and maximum flow algorithms. An important test for both language and library design will be the ease with which such more involved algorithms can be implemented.

Further developments of Fork95 are foreseen, possibly by including new language constructs (e.g. explicit pipelining), possibly in the direction of making (parts of) the language useful for other machine models or PRAM variants. A Fork95++ based on C++ would make a safer and more elegant library interface possible, and is highly desirable.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the physical design of PRAMs. *The Computer Journal*, 36(8):756–762, 1993.

[2] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[3] K. W. Chong and T. W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18:378–402, 1995.

[4] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[5] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

[6] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A high-level language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[7] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming.* MIT Press, 1991.

[8] T.-S. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. In *9th International Parallel Processing Symposium (IPPS)*, pages 106–112, 1995.

[9] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[10] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19:383–401, 1995.

[11] C. W. Keßler and H. Seidl. Fork95 Language and Compiler for the SB-PRAM. 5th Int. Workshop on Compilers for Parallel Computers, 1995.

[12] C. W. Keßler and H. Seidl. Integrating Synchronous and Asynchronous Paradigms: The Fork95 Parallel Programming Language. Proc. MPPM-95 Int. Conf. on Massively Parallel Programming Models, 1995.

[13] C. W. Keßler and H. Seidl. Language Support for Synchronous Parallel Critical Sections. Technical Report 95-23, FB IV Informatik der Universität Trier, 1995.

[14] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings of the 10th ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 597–609, 1983.

[15] M. Philippsen and W. F. Tichy. Compiling for Massively Parallel Machines. In *Code Generation – Concepts, Tools, Techniques*, pages 92–111. Springer, 1991.

[16] J. L. Träff. Explicit implementation of a parallel dictionary. Technical Report SFB 124-D6 10/95, Universität des Saarlandes, Sonderforschungsbereich 124, 1995.

[17] J. L. Träff. PAD: A library of basic PRAM algorithms. Submitted, 1995.

[18] J. L. Träff. Parallel searching, merging and sorting. Technical Report SFB 124-D6 1/96, Universität des Saarlandes, Sonderforschungsbereich 124, 1996.

[19] U. Vishkin. A parallel blocking flow algorithm for acyclic networks. *Journal of Algorithms*, 13:489–501, 1992.