# Language and Library Support for Practical PRAM Programming*

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** *We investigate the well-known PRAM model of parallel computation as a* practical parallel programming model*. The two components of this project are a general-purpose PRAM programming language called Fork95, and a library, called PAD, of efficient, basic parallel algorithms and data structures. We outline the primary features of Fork95 as they apply to the implementation of PAD. We give a brief overview of PAD and sketch the implementation of library routines for prefix-sums and bucket sorting. Both language and library can be used with the SB-PRAM, an emulation of the PRAM in hardware.*

## 1 Introduction

We describe a project investigating the PRAM (see e.g. [8]) as a *practical programming model*. The components of the project are a general-purpose programming language for PRAMs, called *Fork95*, and a library of basic *PRAM Algorithms and Data structures*, called *PAD*. The central theme of the project is efficiency: The programming language must be adequate for the implementation of parallel algorithms as found in the literature, and efficiently compilable to the target machine; the library should include the most efficient algorithms (in terms of parallel work, as well as constants), and support easy implementation of more involved algorithms. Although not dependent on a particular physical realization of the PRAM model, this work is immediately applicable to the SB-PRAM [1], a physical realization of a Priority CRCW PRAM built by W. J. Paul's group at the University of Saarbrücken. A 128 PE prototype is operational; the full 4096 PE version is under construction.

This paper reports on work in progress. A very brief outline was given in [12]. Fork95 is described in detail in[1] [10, 11], PAD[2] in [18, 19].

*This work was partially supported by ESPRIT LTR Project no. 20244 - ALCOM-IT

†This author was supported by DFG, SFB 124-D6, VLSI Entwurfsmethoden und Parallelität.

[1] These reports are available at the Fork95 homepage at http://www.informatik.uni-trier.de/~kessler/fork95.html or ftp://ftp.informatik.uni-trier.de/pub/users/Kessler. The Fork95 kit contains the compiler, documentation, a simulator for the SB-PRAM, and small example programs.

[2] Documentation and a first release of PAD is available at http://www.mpi-sb.mpg.de/guide/activities/alcom-it/PAD.

## 2 Fork95 features used in PAD

Fork95 is a language for explicit programming of synchronous shared memory computers (PRAMs) in the SPMD (Single Program, Multiple Data) paradigm. Hence concepts like processors, processor ID's, shared memory, and synchronicity are explicit to the programmer. In contrast to the unlimited number of processors often assumed in the PRAM literature, Fork95 supports only a fixed number of processors, as determined by the underlying hardware. Fork95 offers means for deviating from strictly synchronous execution, in particular asynchronous program regions where exact statement-level synchrony is not enforced. Fork95 grew out of a proposal [5] for a strictly synchronous PRAM programming language, but has been based on the C programming language, from which it inherits features like pointers, dynamic arrays, etc. Carefully chosen defaults permit inclusion of existing C sources without any syntactical change. The asynchronous mode of computation (the default mode) allows to save synchronization points and enables more freedom of choice for the programming model.

For efficiency reasons the significant run time overhead of virtual processor emulation has been abandoned by restricting the number of processes to the hardware resources, resulting in a very lean code generation and run time system. To compensate for this conceptual drawback, the library routines of PAD can be called by any number of processors and with data instances of arbitrary size. Additionally, "parallel iterators" are provided for common data types such as arrays, lists, trees etc., which helps the programmer to schedule the processors evenly over data instances larger than the number of processors.

Fork95 supports many parallel programming paradigms, such as data parallelism, strictly synchronous execution, farming, asynchronous sequential processes including parallel critical sections, pipelining, parallel divide-and-conquer, and parallel prefix computation [10].

### 2.1 Shared and private variables

The entire shared memory of the PRAM is partitioned into private address subspaces (one for each processor) and a shared address subspace, as configured by the programmer. Accordingly, variables are classified either as private (declared with the storage class qualifier `pr`, this is the default) or as shared (`sh`), where "shared" always relates to the group of processors (see Subsection 2.5) that defined that variable. Private variables and objects exist once for each processor.

The usage of pointers in Fork95 is as flexible as in C, since all private address subspaces have been embedded into the global shared memory. In particular, one does not have to distinguish between pointers to shared and pointers to private objects, in contrast to some other parallel programming languages, for example $C^*$ (see eg. [17]). The following program fragment

```
pr int prvar, *prptr;
sh int shvar, *shptr;
prptr = &shvar;
```

causes a private pointer variable `prptr` of each processor to point to the shared variable `shvar`.

Fork95 maintains three kinds of heaps: a global, permanent shared heap, an automatic shared heap for each group of processors, and a private heap for each processor. Space on the private heaps is allocated and freed by `malloc()` and `free()` as in C. Objects allocated on the automatic shared heap by `shalloc()` exist as long as the group of processors that executed `shalloc()`. Permanent shared objects allocated on the permanent shared heap by `shmalloc()` exist until they are explicitly freed by `shfree()`.

Fork95 makes no assumption about concurrent read/write operations, but inherits the conflict resolution mechanism of the target machine. In case of the SB-PRAM both simultaneous read and write operations are allowed. In case of simultaneous writing to the same memory cell, the lowest numbered processor wins, and its value gets stored. Assume all processors execute the following statement simultaneously:

```
shptr = &prvar;  /* concurrent write */
```

This makes the shared pointer variable `shptr` point to the private variable `prvar` of the processor with the lowest processor ID. In this deterministic way, private objects can be made globally accessible.

Processors in Fork95 are identified by their processor ID, which is accessible as a special private variable `$`, initially set to the physical processor ID. At any point during execution of a Fork95 program the processors form groups (see Section 2.5); a special variable `@` shared by all processors belonging to the same group holds the current group ID. `$` and `@` are automatically saved and restored at group–splitting operations.

## 2.2 Synchronous and asynchronous mode

Fork95 offers two different programming modes that are statically associated with source code regions: synchronous and asynchronous mode. In synchronous mode, processors work strictly synchronous:

> **Synchronicity Invariant (SI):** All processors belonging to the same (active) group have their program counters equal at each time step.

In asynchronous mode, the SI is not enforced.

Functions are classified as either synchronous (declared with type qualifier `sync`) or asynchronous (`async`, this is the default). A synchronous function is executed in synchronous mode, except from blocks starting with a `farm` statement

```
farm  <stmt>
```

which enters asynchronous mode and re-installs synchronous mode after execution of `<stmt>` by an exact barrier synchronization.

An asynchronous function is executed in asynchronous mode, except from the body of a `join` statement which enters synchronous mode, beginning with an exact barrier synchronization. The `join` statement

```
join( <lower>, <upper>, <groupspace> )
    <stmt>  [ else <otherstmt> ]
```

(cf. [11]) offers a means for several asynchronously operating processors to meet at this program point, form a new group, and execute `<stmt>` in synchronous mode. The first two parameters may be used to install a lower resp. upper bound on the number of collected processors. Statement `<stmt>` is executed by at most one group of processors at any time, and can thus be viewed as a *synchronous parallel critical section*. A shorthand for a common special case of `join` is

```
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 asynchronous and synchronous regions, where a synchronous caller causes the compiler to insert an implicit `farm` around the call, maintaining asynchronous execution mode for the callee.

**The importance of being synchronous.** In synchronous mode, shared variables need not be protected by locks because they are accessed in a deterministic way: the programmer is guaranteed a fixed execution time for each statement which is the same for all processors. Thus no special care has to be taken to avoid race conditions. This is due to the absence of virtual processing in Fork95. For instance, in

```
sync void foo( sh int x, a )
{... if ($<10)  a = x;  else  y = a; ...}
```

all processors of the `else` part must, by the semantics of synchronous mode, read the *same* value of `a`. To guarantee this in the presence of virtual processing would require a group lock for each shared variable.

Due to the presence of pointers and weak typing in C, a lock would be required for each shared memory cell!

**The importance of being asynchronous.** In asynchronous program regions there are no implicit synchronization points. Maintaining the SI requires a significant overhead also for the cases where each group consists of only one processor, or when the SI is not required for consistency because of the absence of data dependencies. Marking such regions as asynchronous can lead to substantial savings. Considerate usage of `farm` and asynchronous functions can result in significant performance improvements (see Sect. 3.2).

## 2.3 Explicit Parallelism

A Fork95 program is executed by all started PRAM processors in SPMD mode. Thus, parallelism is statically present from the beginning of program execution, rather than being spawned dynamically from a sequential thread of control.

As mentioned Fork95 does not support virtual processing. The PAD library compensates for this inconvenience by having its routines be implicitly parametrized by the number of executing processors. A *work-time* framework [2, 8] is thus convenient for describing the performance of the routines in the library. A library routine implementing an algorithm with running time $t$ (assuming an unbounded number of processors) while performing $w$ operations in total to do its job, runs in time $O(t + w/p)$ when called by a group of $p$ synchronously operating processors. We will say that a routine which runs in $O(n/p)$ time on a data instance of size $n$ takes *(pseudo)constant time* (this is indeed so when $p \geq n$). — In order to make a program independent of the number of processors, PAD provides parallel iterators on common data structures. The number of calling processors is determined by the routines themselves by the Fork95 routine `groupsize()`, and needs not be supplied by the caller.

For instance, a dataparallel loop over $n$ array elements using $p$ processors with ID's $\$ = 0, 1, \ldots, p-1$

```
sh int p=groupsize(); /*get # procs in my group*/
pr int i;
for (i=$; i<n; i+=p)  a[i] = b[i] + c[i];
```

could also be written using a PAD macro

```
par_array(i,n)  a[i] = b[i] + c[i];
```

which causes the private variable `i` of the processor with ID `$` to step through the values `$`, `$`$+p$, `$`$+2p$, ..., up to `n`. This takes (pseudo)constant time and works both in synchronous and asynchronous mode.

## 2.4 Multiprefix operations

Multiprefix operations are essential parallel programming primitives, which can be provided at practically no extra cost in massively parallel shared memory emulations (theoretically, ie. with no asymptotic loss in efficiency, as well as in the number of extra gates needed). The SB-PRAM offers highly efficient multiprefix instructions which execute in only two CPU cycles, independent of the number of processors.

The atomic multiprefix operators of Fork95 are inherited from the SB-PRAM, but should be thought of as part of the language. The expression

```
prvar = mpadd( &shvar, <exp> )
```

atomically adds the (private) integer value of `<exp>` to the shared integer variable `shvar` and returns the old value of `shvar` (as it was immediately before the addition). The operators `mpmax` (multiprefix maximum), `mpand` and `mpor` (multiprefix bitwise AND resp. OR) work analogously. The Fork95 run-time library offers routines for various kinds of locks, semaphores, barriers, self-balancing parallel loops, and parallel queues. All these are based on multiprefix operators.

The multiprefix operators also work if executed by several processors at the same time and even with different shared integer variables `shvar`. Among the processors participating in an `mpadd` instruction on the same shared integer variable `shvar` at the same time, the processor with the $i$th-largest physical processor ID (contributing an expression `<exp>` evaluating to a private integer value $e_i$) receives the (private) value $s_0 + e_0 + e_1 + \ldots + e_{i-1}$, where $s_0$ denotes the previous value of `shvar`. Immediately after the execution of this `mpadd` instruction, `shvar` contains, as a side effect, the global sum $s_0 + \sum_j e_j$ of all participating expressions. Thus, the multiprefix operators provide fast reduction operators for integer arrays which can be used in synchronous as well as in asynchronous mode.

An immediate application of the `mpadd` operator is a PAD library routine to compute, in constant time, the prefix sums over an array of $n$ integers:

```
sync void prefix_add( sh int x[], sh int n,
                      sh int y[], sh int offset)
{ sh int sum = offset;
  pr int i, s;
  par_array(i,n) {  /* n/p segments of size p */
    s = x[i];
    y[i] = mpadd( &sum, x[i]);
    y[i] += s;                              } }
```

For an input array `x` of size `n`, `prefix_add()` computes in array `y` the prefix sums of the input array, offset by `offset`. Ie., `y[i]` is assigned `x[0]+...+x[i]`. The actual number of calling processors is implicitly determined by the parallel iterator `par_array()`.

The fact that multiprefix operators can be executed with different shared variables `shvar`, provides for easy implementation of even more powerful primitives. For instance, "colored prefix summation" in which each element of the input array has an associated color (an integer between 0 and $c - 1 \leq n$) and prefix sums are to be computed colorwise, can also be done in constant, ie. $O(n/p)$ time:

```
sync void colorprefix( sh int x[], sh int n,
       sh int color[], sh int c, sh int ccount[])
{ pr int i, s;
  par_array(i,c) ccount[i] = 0;
  /* ccount[i] will count elements of color[i] */
  par_array(i,n) {
    s = x[i];
    x[i] = mpadd(&ccount[color[i]],x[i]);
    x[i] += s;                                   } }
```

Upon return from this routine, `x[i]` holds the sum of all elements up to and including `i` of the same color as `x[i]` (ie. `color[i]`), and `ccount[i]` holds the sum of all elements with color `color[i]`. We give a surprising application of this primitive in Section 3.2.

### 2.5 The group concept

In synchronous mode, the processors are partitioned into independent, synchronous groups. Shared variables and objects exist once for the group that created them; global shared variables are visible to all processors. Processors within an (active) group maintain the SI, but there is no enforced synchrony among processors belonging to different groups. Such processors, although synchronous at the instruction level (as guaranteed by the hardware), may be executing different instructions of the program at the same time.

Splitting the current group of processors into subgroups can be done explicitly by the `fork` statement

```
fork (<exp1>; @=<exp2>; $=<exp3>)  <stmt>
```

which evaluates the shared expression `<exp1>` to an integer $g$ and splits the current leaf group in $g$ subgroups, with local group ID's `@` numbered 0,1,...,$g-1$. Each processor decides by the value of `<exp2>` which of the new subgroups it will join, i.e. all processors for which the private expression `<exp2>` evaluates to the same value $i$ join group $i$ with `@` $= i$. The assignment to the group-relative processor ID `$` permits local renumbering of `$` inside each new subgroup. The SI is only maintained within subgroups. The parent group is restored (by exact barrier synchronization) when all subgroups have finished their execution of `<stmt>`.

As long as control flow depends only on "shared" conditions that evaluate to the same value on each processor, the SI is preserved without changes. If control flow diverges, due to "private" conditions in `if` or loop statements, the active group is deactivated and split into new subgroups, in order to preserve the SI within the subgroups. These are active until control flow reunifies again. Then, after exact barrier synchronization, the SI holds again for the parent group which is reactivated. Thus, at any point during program execution, the groups form a tree-like hierarchy, with the root group consisting of all started processors, and the leaf groups being the currently active ones, for which the SI currently holds. Subgroup creation can be directly used for parallel Divide&Conquer implementations, as eg. in PAD's `quicksort` routine.

In asynchronous mode, there is no (implicit) splitting of groups required as no statement-level synchrony is to maintain.

## 3 The PAD library structure

The PAD library of parallel algorithms provides support for implementation of parallel algorithms as found in the current theoretical literature by making basic PRAM algorithms and computational paradigms, like prefix sums, sorting, list ranking, tree computations etc. available. PAD provides a set of abstract parallel data types like arrays, lists, trees, graphs, dictionaries. However, the user of the library is responsible for ensuring correct use of operations on data objects, since Fork95 does not support abstract data types as such (see Section 4). PAD contains type definitions for some common parallel data types, and is organized as a set of routines which operate on objects of these types. Computational paradigms, e.g. prefix sums over arrays of arbitrary base type with a given associative function are provided for by library routines with function parameters. The standard operations have often certain "canonical" instances, e.g. prefix sums for integer arrays. Library routines for both general and special cases are normally available.

### 3.1 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 available. For instance, "prefix sums" of an array `x` of `n` objects of type `struct x_object`, with a function `f` implementing an associative function $f$ on the set of `struct x_object`, can be computed by

```
Prefix( x, n, sizeof(struct x_object), PR3 f);
```

Here `f` is a (user-defined) Fork95 function which implements the assignment $x = f(y, z)$ when called with pointers to objects $x$, $y$ and $z$. The macro `PR3` performs the necessary type casting. The recursive routine takes $O(\log n + n/p)$ time, in contrast to the constant time routine `prefix_add()` for integer arrays.

### 3.2 The merge library

The merge library contains operations for parallel searching and merging of ordered arrays, and sorting of arrays. The *merge* routine implements the CREW algorithm in [4], which runs work-optimally in $O((m + n)/p + \log(m + n))$ time, $m$ and $n$ being the lengths of the input arrays. The implementation is very efficient when compared to a "reasonable" sequential merge routine. The running time of the parallel algorithm with one processor (almost) equals the running time of the sequential implementation, and the parallel algorithm gives very close to perfect speed-up [19]. This can be partly ascribed to the use of asynchronous mode. The algorithm partitions the input arrays into $p$ pairs of subsequences of size at most $\lceil (m+n)/p \rceil$, which are then merged together, one pair for each processor. Obviously the concurrent mergings are independent and therefore executed in asynchronous mode, which reduces running time considerably, up to a factor 2. A trick in [4] makes it easy to implement a work-optimal parallel *merge sort* algorithm, which runs in $O(\log^2 n + n \log n/p)$ time, using the abovementioned general merge routine. Speed-up of up to 41 with 128 processors has been achieved for arrays of only 8K integers, see Tab. 1.

With `colorprefix()` we can implement a parallel *bucket sort* which sorts $n$ integers in the interval $0, \ldots n-1$ in (pseudo)constant time. The routine is surprisingly short and simple: `colorprefix()` computes for each input element `x[i]` the number of elements before `x[i]` that are equal to `x[i]`, if we simply use, as the color of element `x[i]`, `x[i]` itself. This yields the *rank* of each element `x[i]`. A prefix sums computation over the `rank` array suffices to determine, for each element `x[i]`, how many elements in array `x` are strictly smaller than `x[i]`. The final position `rank[i]` of `x[i]` in the output is the rank of `x[i]` plus the number of strictly smaller elements:

```
sync void smallperm( sh int keys[], n, rank[])
{ sh int *count = (int *) shalloc(n*sizeof(int));
  pr int i;
  par_array(i,n)  rank[i] = 1;
  colorprefix( rank, n, keys, n, count);
  preprefix_add( count, n, count, 0);
  par_array(i,n)
    rank[i] = rank[i] - 1 + count[keys[i]];     }
```

`preprefix_add( count, ...)` stores in `count[i]` the prefix sum `count[0]+...+count[i-1]`; it is implemented similar to `prefix_add()` in Section 2.4. Bucket sorting just calls `smallperm()` and permutes the input according to the computed `rank` array.

**Proposition 1** *On the SB-PRAM (or other parallel shared memory machine with constant time multiprefix addition)* prefix sums, colored prefix sums *with $n$ colors, and* bucket sorting *of integer arrays with $n$ elements in the range $0, \ldots, n-1$ can be done in (pseudo)constant, i.e. $O(n/p)$, time with p processors. The constant hidden in the $O$-notation is very modest.*

**Some results.** Tab. 1 gives the results of sorting integer arrays using PAD's quicksort, mergesort and smallsort routines as described above. Timings are given in SB-PRAM clock cycles (using the simulator for the machine, which explains the small input sizes). The parallel routines are compared to a reasonable corresponding sequential implementation. Bucket sorting is very efficient and gives speed-up very close to $p$. Small deviations (eg. 113 instead of 128) are due to allocation of the auxiliary arrays. Mergesort is somewhat more efficient than quicksort, both in absolute terms and wrt. speed-up.

| n=2737 | mergesort | SU | quicksort | SU | smallsort | SU |
|---|---|---|---|---|---|---|
| Seq | 2807273 | | 3106589 | | 1068453 | |
| 8 | 567288 | 5 | 1002876 | 3 | 134639 | 8 |
| 16 | 317314 | 9 | 678262 | 5 | 67949 | 16 |
| 32 | 207650 | 14 | 484662 | 6 | 34409 | 31 |
| 64 | 146534 | 19 | 292538 | 11 | 17639 | 61 |
| 128 | 113508 | 25 | 140578 | 22 | 9449 | 113 |
| n=8211 | | | | | | |
| Seq. | 9829269 | | 10829273 | | 3203313 | |
| 8 | 1828718 | 5 | 4741482 | 2 | 401399 | 8 |
| 16 | 972582 | 11 | 2411150 | 4 | 201329 | 16 |
| 32 | 604776 | 16 | 1223450 | 9 | 101099 | 32 |
| 64 | 364248 | 27 | 696608 | 16 | 51179 | 63 |
| 128 | 241906 | 41 | 433704 | 25 | 26219 | 122 |

Table 1: Sorting results. SU = absolute speed-up.

### 3.3 Parallel lists

A *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*, i.e. the operation of determining for each element of the list its distance from the end of the list [8]. A parallel list contains in addition to element pointers the necessary fields for the list ranking operations. Currently a simple, non-optimal list ranking operation based on pointer jumping is implemented in PAD, as well as two work-optimal algorithms from the literature [3, 13]. At present, pointer jumping is at least twice as fast as the best of the work-optimal algorithms [13] for lists of up to 50K elements. Other operations on lists include reversal, catenation, permutation into rank order, and others. A parallel iterator `par_list(elt,i,n,list)` is available for processing the elements of a parallel list in "virtual" parallel.

### 3.4 Trees and Graphs

Trees are represented in PAD by an array of edges and an array of nodes, where edges directed from the same node form a segment of the edge array. An edge is represented by pointers to its tail and head nodes, and to its reverse edge. A "next edge" pointer is used to represent Euler tours. PAD offers routines 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 postorder traversal number, level numbers etc.

PAD also supports least common ancestor preprocessing and querying. The currently implemented routines are based on the reduction to the range query problem, which is part of the prefix library.

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 routines for generic tree contraction

A data type for directed graphs similar to the tree data type is defined. Parallel operations include finding the connected components of a graph, and computing a (minimum) spanning tree

### 3.5 Parallel dictionaries

Currently PAD contains one non-trivial parallel dictionary data structure based on 2-3 trees [15]. 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. [18] gives the full implementation.

## 4 Status, conclusion, and future work

A compiler for Fork95, together with the system software required, is available for the SB-PRAM. Due to the powerful multiprefix instructions of the SB-PRAM, the overheads for locking/unlocking, barrier synchronization and group splitting are very modest.

PAD complements some inconveniences in Fork95. On the other hand, PAD's implementation exemplifies the usability and expressivity of Fork95 for larger scale parallel programming. A first version of PAD, as outlined in Section 3, is available from October 1996. It will be extended with more advanced graph and combinatorial algorithms, e.g. 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, 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.

## 5 Related work

**Other PRAM–oriented languages** 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 (by *flattening*) and executed on SIMD machines. In contrast, the MIMD-oriented Fork95 also allows for asynchronous and task parallelism, low-level PRAM programming and direct access to shared memory locations. — Dataparallel variants of Modula [9, 16] support a subset of Fork95's functionality. The main constructs to express parallelism are synchronous and asynchronous parallel loops. However no strict synchronicity is supported, and there is no group concept. [9] compiles to a PRAM simulator while [16] offers back-ends for several existing machines. `ll` [14], a similar approach, uses Pascal as base language. Further dataparallel languages in this tradition (see e.g. [17]) are $C^*$, *Dataparallel C* [6], and dataparallel Fortran dialects such as *HPF*. The latter ones are mainly targeted towards distributed memory machines and require data layout directives to perform efficiently. Exact synchronicity is not supported as it is not available on the target architectures considered.

**Other PRAM-oriented parallel libraries** A large number of PRAM-inspired algorithms has already been implemented in NESL [2]. [7] reports on concrete implementations on a MasPar for many of the same algorithms as those in PAD.

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

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

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

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

[7] 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*, pp 106–112, 1995.

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

[9] S. Juvaste. The Programming Language pm2 for PRAM. Technical Report B-1992-1, Dept. of Computer Science, University of Joensuu, Finland, 1992.

[10] C.W. Keßler and H. Seidl. Integrating Synchronous and Asynchronous Paradigms: The Fork95 Parallel Programming Language. Tech. Report 95-05, FB 4 Informatik, Univ. Trier, 1995.

[11] C.W. Keßler and H. Seidl. Language Support for Synchronous Parallel Critical Sections. Tech. Report 95-23, FB 4 Informatik, Univ. Trier, 1995.

[12] C.W. Kessler and J.L. Träff. A library of basic PRAM algorithms and its implementation in FORK. In *8th Annual ACM Symposium on Parallel Algorithms and Architechtures (SPAA)*, pp 193–195, 1996. Research Summary.

[13] C.P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Trans. on Computers*, C-34(10):965–968, 1985.

[14] C. León, F. Sande, C. Rodriguez, and F. Garcia. A PRAM Oriented Language. In *EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing*, pp 182–191, 1995.

[15] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proc. 10th ICALP*, Springer LNCS 154, 1983.

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

[17] M.J. Quinn. *Parallel Computing, Theory and Practice, 2nd edition.* McGraw-Hill, 1994.

[18] J.L. Träff. Explicit implementation of a parallel dictionary. Tech. Report SFB 124-D6 10/95, Univ. Saarbrücken, SFB 124 VLSI Entwurfsmethoden und Parallelität, 1995. 53 pp.

[19] J.L. Träff. Parallel searching, merging and sorting. Tech. Report SFB 124-D6 1/96, Univ. Saarbrücken, SFB 124 VLSI Entwurfsmethoden und Parallelität, 1996. 45 pp.