PAD – A Library of PRAM Algorithms and Data structures*

User Manual. Version 1. Revision 2.

Jesper Larsson Träff†
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
traff@mpi-sb.mpg.de

December 1997
UNDER REVISION

Contents

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

1 Introduction

This manual summarizes the first version of the PAD library of *PRAM Algorithms and Data structures*. This first version of PAD contains basic *parallel data structures* like arrays, lists, trees, graphs and dictionaries, and provides various fundamental operations on these data structures, eg. prefix sums, searching, merging and sorting, list ranking, rooting of trees, lowest common ancestor computations, generic tree contraction, connected components, and dictionary operations. For motivation and overview of the PAD project, see [10]. An excellent introduction to PRAM algorithms can be found in [2]. PAD is work in progress and will be extended with more data structures (graphs, strings) and algorithms in the immediate future. Comments, corrections, error reports, and suggestions are welcome, and should be sent to the author.

PAD is implemented in the C based PRAM programming language `Fork95`, which is described in [3]. The `Fork95` compiler and a simulator for a "real" PRAM, the SB-PRAM [1], currently under completion by W. J. Paul's group at the University of Saarbrücken, are available via the `Fork95` homepage

`http://www.informatik.uni-trier.de/~kessler/fork95.html`

The source code itself of the PAD library can be found as `CWEB`s [6] in various technical reports [8, 9, 11, 12, 13]. The PAD distribution package is available via the PAD homepage `http://www.mpi-sb.mpg.de/guide/activities/alcom-it/PAD`

Short overviews of `Fork95` and PAD have been given in [4, 5].

PAD is subdivided into a number of sublibraries, each with its own header file.

- The general PAD header file, `pad.h`, which defines some useful macros and iterators, must always be included when using the library.
- The *prefix library* (Section 5) contains basic operations on *parallel arrays*, mainly of the prefix sums type. The prefix library also contains operations for range querying. Definitions for the prefix library are found in `prefix.h`.
- Searching, merging and sorting operations on arrays are found in the *merge library* (Section 6), whose header file is `merge.h`.
- The *list library* (Section 7) defines a *parallel list* data type, and contains basic operations on lists, first and foremost *list ranking*. Definitions for the list library are found in `list.h`.
- A *parallel tree* data type is defined in the *tree library* (Section 8), which contains operations on trees based on the Euler tour technique (rooting, level computations etc.). The library also has operations for lowest common ancestor preprocessing and querying on rooted trees, and for generic tree contraction of binary trees. Definitions for the tree library are found in `tree.h`.
- The *graph library* defines a graph data type and the most basic parallel operations on graphs, like finding the connected components. Definitions for the *graph library* are found in `graph.h`.

  NOTE: The graph library is not part of the first release of PAD.
- PAD contains a *parallel dictionary* (Section 10) data type and basic parallel operations on 2-3 trees. Definitions for parallel dictionaries are found in `dict.h`.

2 Rules and guidelines

The reader is assumed to be familar with the `Fork95` PRAM programming language. Likewise familiarity with basic PRAM terminology is assumed. The algorithms implemented will not be described in this manual; instead the reader is referred to [10] and the references given there.

The PAD library defines a (small) set of *abstract parallel data types*, for which a set of corresponding (parallel) operations is defined. Currently types for lists, trees, graphs and dictionaries are defined, while arrays are just C-style arrays with the normal indexing conventions. Some operations have “canonical” instances (ie. prefix operations on integer arrays), and therefore exist in different variants. Each data type comes with a set of *parallel iterators* which serve as a poor man's substitute for virtual processor emulation; the `Fork95` language does *not* support virtual processing.

In PAD each processor has a private integer variable `ME` (in `Fork95`: `$`) which serves as processor number. In a synchronous context all processors belonging to the same group share an integer variable `MY` (in `Fork95`: `@`), which serves as group number. The macro `proc(i)` can be used when only the processor(s) with `ME` equal to `i` should be active, typically to avoid concurrent writing to a shared variable, or when performing input/output.

Each of the parallel operations of PAD must be called from a synchronous program region. Parallel operations can be called by any number of processors, and when called internally determine the number of calling processors. Unless otherwise noted, the calling processors are assumed to be numbered consecutively starting from 0. There is a library function for renumbering which can be called as desired. A PAD procedure implementing an algorithm that takes t time steps when executed with an unbounded number of processors and performs w operations in total, takes $O(t + w/p)$ time when called by a group of p synchronous processors.

Many operations are generic, ie. implement a computational paradigm (eg. tree contraction [2]) and are specialized to a specific application by procedure parameters for performing the concrete operations (eg. concrete rake and unrake operations). Such parameters implement operations on a small, bounded number of elements, and are intended to be executed by only one processor, but many such operations may take place simultaneously. They must therefore be declared as *asynchronous functions*. These procedures are called with pointers to the elements upon which they are to work. When calling a generic PAD operation type casting of concrete procedure parameters into generic procedures with `void*` arguments must be performed. This type casting can be done with the macros `PR1`, `PR2`, `PR3`, for procedures (no return value) of 1, 2 or 3 parameters, or `FC1(`*type*`)`, `FC2(`*type*`)`, `FC3(`*type*`)`, for functions with a return value of type *type* of 1, 2, or 3 parameters.

PAD strives to adhere to an abstract data type discipline. Ideally instances of data types should be manipulated exclusively through PAD library functions, with no knowledge of the internal structure assumed on behalf of the user. In this first version of the library, this goal has probably not been fully achieved. The library will be improved in this direction. The data types of PAD in many cases come equipped with allocators and deallocators, which, when present, should be used exclusively for memory management.

In order to assess the efficiency of the algorithms implemented in PAD, most operations also have a sequential counterpart in the form of an asynchronous procedure which can be called by one processor. Many of the data types also come with a set of generators, also to be used for test and evaluation purposes. More are planned in future versions of PAD.

In the first version of PAD there is hardly any error handling. Likewise it is fully the users responsibility to ensure correct use of the library operations. PAD also offers no debugging facilities. Hopefully the library operations are correct, and the user is responsible for debugging his own programs by deliberate use of `pprintf()` statements. However, the sublibraries can be compiled with the flag `TESTING` defined, which will generate output from central library operations which may or may not be useful in debugging. If errors are discovered in PAD procedures, it is recommended to contact the author.

3 Installation Guide

The PAD package consists of the source code for the sublibraries described above, for each sublibrary a small example program, a `Makefile` for compiling the library, the PAD manual (this document), and a file with definitions which can be used when writing your own PAD programs as `CWEB`s.

The PAD package comes as a compressed archive, `PAD.`x`.`y`.tar.gz`, x and y being version and revision number respectively. PAD is probably best installed in its own subdirectory, eg. `PAD`, created prior to installation. The `Fork95` package must have been installed prior to compilation of PAD. The archive is uncompressed with the command

```
gunzip PAD.1.0.tar.gz
```

and unpacked with the command

```
tar xvf PAD.1.0.tar
```

PAD is compiled with the command

```
make all
```

which will result in a (rather large) number of warnings, all of which can safely be ignored. No error messages should be produced. The proper installation of the library can be verified by running the small example programs `padexamp`, `preex`, `mergeex`, `listex`, `treeex`, and `dictex` on the SB-PRAM simulator. All programs should terminate normally, and produce half-way sensible output (see the technical reports [8, 9, 11, 12, 13]).

4 An example

The following, complete example program is intended to give a flavor of programming with PAD. It shows how to use the generic prefix operation to implement a segmented prefix operation, and then gives a few examples of parallel operations on trees. The example can be found in the file `padexamp.c` which comes with the PAD package.

```
/* PAD example. October 1996 */

/* FORK libraries */
#include <fork.h>
#include <io.h>

/* PAD sublibraries */
#include "pad.h"
#include "prefix.h"
#include "list.h"
#include "tree.h"

/* define an associative function for segmented prefix sums */
```

```
struct mpair {
  int e; /* element */
  int m; /* marker */
};
async void addmpair(struct mpair *x, struct mpair *y, struct mpair *z)
{
  if (z->m==1) {
    x->e = z->e; /* start of new group */
    x->m = 1;
  } else {
    x->e = y->e+z->e; /* within group: add */
    x->m = y->m;
  }
}

void main()
{
  start { /* collect all processors to execute synchronously */
    sh int p ;
    sh int n, m;
    pr int i;
    sh int *x;
    sh int *marker;

    sh struct mpair *a;

    sh TREE *t;
    pr TREE_NODE *v;
    sh LIST *eulerlist;
    sh RANGE *lcaquery;
    sh int deg, ch;

    p = __STARTED_PROCS__ ; /* get number of available processors */

    proc (0) {
      n = 100; m = 25;
    }
    x = (int *) shalloc(n*sizeof(int));
    marker = (int *) shalloc(n*sizeof(int));
    a = (struct mpair *) shalloc(n*sizeof(struct mpair));

    /* segment, and compute segmented prefix sums */
    aritseq(x,n,1,1);
    prefix_add(x,n,x,0);
    par_array(i,n) marker[i] = 0;
    par_array(i,n) if (x[i]<n) marker[x[i]] = 1;
    par_array(i,n) {
```

```
      a[i].e = 1;
      a[i].m = marker[i];
    }
    Prefix(a,n,sizeof(struct mpair),PR3 addmpair);
    par_array(i,n) pprintf("%d: (%d,%d)\n",i,a[i].e,a[i].m);

    /* simple examples of tree computations */
    proc (0) {
      n = 50; deg = 5; ch = 3;
    }
    proc(0) pprintf("Generating small complete tree\n");
    t = complete_tree(n,deg,ch);
    proc(0) pprintf("Actual treesize n=%d m=%d\n",tree_n(t),tree_e(t));

    eulertour(t); /* make an Euler tour */
    eulerlist = (LIST *) shalloc(tree_e(t)*sizeof(LIST)); /* allocate */
    proc(0) pprintf("Rooting tree\n");
    tree_root(t,NODE(t,0),eulerlist);
    par_nodes(v,t) {
      pprintf("node %d level %d pre %d post %d desc %d\n",
      NODE_NO(t,v),
      node_level(v),node_pre(v),node_post(v),node_desc(v));
      if (LEAF(v)) pprintf("leaf %d\n",NODE_NO(t,v));
    }

    /* LCA stuff */
    lcaquery = lca_preprocess(t,eulerlist);
    par_array(i,tree_n(t)-1)
      pprintf("lca(%d,%d) = %d\n",i,i+1,
      NODE_NO(t,lca(NODE(t,i),NODE(t,i+1),lcaquery)));
    tree_height(t,eulerlist);
    par_nodes(v,t)
      pprintf("height %d: %d\n",NODE_NO(t,v),node_height(v));
  }
}
```

The program can be found in the file `padexamp.c`. The example is compiled by the command

```
fcc -I$FORKDIR/include -c -g0 -m -FORK -S padexamp.c
```

Compilation produces some warnings of the form

```
padexamp.c:65: warning: call by async function pointer implicitly enters
farming mode
padexamp.c:71: warning: call by async function pointer implicitly enters
farming mode
```

These can safely be ignored. The object file `padexamp.o` is linked with prefix, list and tree library by the command

```
fcc -o padexamp padexamp.o trees.o ranking.o prefixes.o
```

Finally, the program can be executed on the SB-PRAM simulator by the command

```
pramsim padexamp
```

5 The Prefix Library

The *prefix library* contains basic operations on arrays, mostly of the prefix sums type. It also contains operations for processor renumbering, and segmenting intervals into roughly even sized pieces. Arrays in PAD are normal C-style arrays, represented by a (shared) pointer to the start of the array, the number of elements, and, for non-integer arrays, the element size in units as returned by the `sizeof` operator. Names of operations on non-integer arrays begin with a capital letter, and most often correspond to an integer array operation, whose name then starts with a small letter.

The header file `prefix.h` must be included. Link with `prefixes.o`.

5.1 Array and interval iterators

Below `i` is assumed to be a private variable, and `n`, `f` and `l` to be shared or private expressions having values n, f, l, respectively. The array iterators are intended to simulate parallel execution of the statement (block) following the iterator in that the variable `i` will attain all values in the interval $0, \ldots, n-1$ or $f, \ldots, f+l-1$. The iterators are intended for use in a synchronous context, but will work in an asynchronous context also. When executed in a context with p processors, the interval from 0 to $n-1$ (or from f to $f+l-1$) is divided into blocks of consecutive indices of size at most p, each of which is executed in parallel by the p processors. The iterators differ in the order in which blocks are executed.

`par_array(i,n)`	blocks in increasing order, ie. in the kth iteration, $k \geq 0$, variable `i` of processor `ME` takes the value $\texttt{ME} + kp$.
`par_arrayseg(i,f,l)`	in the kth iteration, $k \geq 0$, variable `i` of processor `ME` takes the value $\texttt{ME} + f + kp$.
`rap_array(i,n)`	blocks in decreasing order, ie. in the kth iteration, $k \geq 0$, variable `i` of processor `ME` takes the value $n - 1 - \texttt{ME} - kp$.
`rap_arrayseg(i,f,l)`	in the kth iteration, $k \geq 0$, variable `i` of processor `ME` takes the value $f + l - 1 - \texttt{ME} - kp$.

The parallel array iterators cannot be nested.

5.2 Segmenting operations

The prefix library defines a data type `MY_SEG` for representing segments of arrays, ie. blocks of consecutive indices. Segmenting is used for concurrent, sequential processing. Data items of type `MY_SEG` has an `index` and a `size` field with the obvious intention.

```
sync void mysegment(sh const int n, pr MY_SEG *segment)
```

segments the interval from 0 to $n-1$ into roughly equal sized blocks, each of size between $\lfloor n/p \rfloor$ and $\lceil n/p \rceil$ elements, when called by a group of p synchronous processors. The index and size of the block allocated to processor `ME` are stored in the respective fields of the private parameter `segment`.

```
sync void *Mysegment(void *a, const sh int n, sh const int size, int *mysize)
```

segments an array `a` of elements of size `size` into roughly equal sized blocks (as above), and returns a pointer to the block of each of the calling processors. The size of the block is stored in the private parameter `mysize`.

```
sync void segment(sh const int n, sh const int s, sh MY_SEG segments[])
```

segments the interval from 0 to $n-1$ into s roughly equal sized blocks, the indices and sizes of which are stored in array `segments`. The `segments` array must have been allocated prior to the call.

```
sync int renorm()
```

returns a new, consecutive processor number to each calling processors. The function is intended to be called when a new subgroup has been formed, eg. from within the branches of a conditional, when consecutive processor numbering is required for the following library calls. New processor number is assigned to each calling processor by the call `ME = renorm();`.

5.3 Prefix operations

Given an associative operation $\oplus$ over a set S, the kth *prefix sum* with offset o of a finite sequence $x_i \in S, i = 0, \ldots, n-1$ is the sum $o + \oplus_{i=0}^{k} x_i$. The kth *preprefix sum* is the sum $o + \oplus_{i=0}^{k-1} x_i$ for $k > 0$ and o for $k = 0$. The kth *suffix sum* is the sum $o + \oplus_{k=i}^{n-1} x_i$. PAD contains operations for computing all prefix, preprefix and suffix sums in parallel where the input sequence is given as an array.

```
sync void prefix_add(sh int in[], sh const int n, sh int out[], sh int offset)
sync void prefix_max(sh bool in[], sh const int n, sh bool out[], sh bool offset)
sync void prefix_and(sh bool in[], sh const int n, sh bool out[], sh bool offset)
sync void prefix_or(sh bool in[], sh const int n, sh bool out[], sh bool offset)
```

compute prefix sums, maxima, bitwise conjunctions and bitwise disjunctions over integer array `in` of size `n` into array `out`. The sums are offset by offset value `offset`.

```
sync void prefix(sh int inout[], sh const int n, async int (*f)(pr int, pr int))
```

computes the prefix sums over the array `inout` of size `n` for the associative integer function `f`. The original contents of the input array are lost.

```
sync int preprefix_add(sh int in[], sh const int n, sh int out[], sh int offset)
sync int preprefix_and(sh bool in[], sh const int n, sh bool out[], sh bool offset)
sync int preprefix_or(sh bool in[], sh const int n, sh bool out[], sh bool offset)
sync int preprefix_max(sh bool in[], sh const int n, sh bool out[], sh bool offset)
```

compute preprefix sums, maxima, bitwise conjunctions and bitwise disjunctions over integer array `in` of size `n` into array `out`. The sums are offset by offset value `offset`.

```
sync void suffix_add(sh int in[], sh const int n, sh int out[], sh int offset)
sync void suffix_max(sh bool in[], sh const int n, sh bool out[], sh bool offset)
sync void suffix_and(sh bool in[], sh const int n, sh bool out[], sh bool offset)
sync void suffix_or(sh bool in[], sh const int n, sh bool out[], sh bool offset)
```

compute suffix sums, maxima, bitwise conjunctions and bitwise disjunctions over integer array `in` of size `n` into array `out`. The sums are offset by offset value `offset`.

```
sync void suffix(sh int inout[], sh const int n, async int (*f)(pr int, pr int))
```

computes the suffix sums over the array `inout` of size `n` for the associative integer function `f`. The original contents of the input array are lost.

```
sync void Prefix(sh void *inout, sh const int n, sh const int size,
                 async void (*fu)(pr void *, pr void*, pr void *))
sync void Suffix(sh void *inout, sh const int n, sh const int size,
                 async void (*fu)(pr void *, pr void*, pr void *))
```

compute prefix and suffix sums over an array `inout` of elements of size `size` for an associative operation `f`. The asynchronous function `f` must be an *update function* which realizes an assignment $x \leftarrow f(y, z)$ when called with pointers to the arguments x, y and z.

Assume the elements of the sequence $x_i \in S$ have an associated color $c_i \in \{0, \ldots, c-1\}$ where c is the number of colors. The kth colored prefix sum is the sum of all $x_i, i \leq k$, of the same color as x_k. PAD contains operations for computing all colored prefix sums over integer arrays.

```
sync void colorprefix_add(sh int x[], sh const int n,
                          sh int color[], sh const int c, sh int ccount[])
sync void colorprefix_max(sh int x[], sh const int n,
                          sh int color[], sh const int c, sh int ccount[])
sync void colorprefix_and(sh int x[], sh const int n,
                          sh int color[], sh const int c, sh int ccount[])
sync void colorprefix_or(sh int x[], sh const int n,
                         sh int color[], sh const int c, sh int ccount[])
```

compute the colored prefix sum, maximum, bitwise conjunction and bitwise disjunction of an input sequence in array `x` with colors in array `color`. The number of colors must be at most `c`. The sum of all elements of color k is stored in `ccount`$[k]$.

Assume the elements of the sequence x_i have been divided into groups (or segments) of consecutive elements by marking the beginning of each new group. A marker sequence m_i marks the elements, $m_i = 1$ if x_i is the first element of a new group, otherwise $m_i = 0$. The kth grouped prefix sum is the sum of the elements $x_i, i \leq k$, belonging to the same group as x_k, ie. the sum of the elements x_i for $j \leq i \leq k$ where $m_j = 1$ and $m_i = 0$ for $i > j$. PAD contains operations for computing all grouped prefix sums.

```
sync int groups(sh bool marker[], sh const int n)
```

computes the number of groups as defined by the `marker` array.

```
sync int group_sizes(sh int marker[], sh const int n, sh int sizes[])
```

computes the number of elements in each group as defined by the `marker` array. Upon return, if element i is marked, `sizes`$[i]$ is the number of elements in the group to which i belongs; if i is not marked, `sizes`$[i]$ is the number of elements preceeding i in the group to which i belongs.

```
sync void group_prefix_add(sh int in[], sh const int n, sh int out[],
                           sh int marker[], sh int *groups)
sync void group_prefix_max(sh int in[], sh const int n, sh int out[],
                           sh int marker[], sh int *groups)
sync void group_prefix_and(sh int in[], sh const int n, sh int out[],
                           sh int marker[], sh int *groups)
sync void group_prefix_or(sh int in[], sh const int n, sh int out[],
                          sh int marker[], sh int *groups)
```

compute the grouped prefix sums, maxima, bitwise conjunctions and bitwise disjunctions over input array `in` into output array `out`. The number of groups, as defined by the `marker` array, is stored in the variable `groups`.

```
sync void group_prefix(sh int inout[], sh const int n,
                       async int (*f)(pr int, pr int),
                       sh int marker[], sh int *groups)
```

computes the grouped prefix sums over integer array `inout` for the associative integer function `f`. The number of groups, as defined by the `marker` array, is stored in the variable `groups`.

```
sync void Group_prefix(sh void *inout,
                       sh const int n, sh const int size,
                       async void (*fu)(pr void *, pr void *, pr void *),
                       sh int marker[], sh int *groups)
```

computes grouped prefix sums for an array inout of elements of size size for an associative operation given as an *update function* fu. The asynchronous update function must realize an assignment $x \leftarrow f(y, z)$ when called with pointers to elements x, y and z. The number of groups, as defined by the marker array, is stored in the variable groups.

```
sync void group_broadcast(sh int inout[], sh const int n,
                          sh int marker[], sh int *groups)
```

broadcasts each marked integer element of the inout array to the entire group to which this element belongs. The number of groups, as defined by the marker array, is stored in the variable groups.

```
sync void Group_broadcast(sh void *x, sh const int n, sh const int size,
                          sh int marker[], sh int *groups)
```

broadcasts each marked element of the inout array to the entire group to which this element belongs. The number of groups, as defined by the marker array, is stored in the variable groups.

5.4 Other parallel prefix operations

```
sync int locate(sh int y, sh int x[], sh const int n)
sync int Locate(sh void *y, sh void *x, sh int n, sh int size,
                async bool (*eq)(void *, void *))
```

locate the first occurrence of element y in the array x. If x is not an integer array, y is given as a pointer to an element to locate, and equality defined by the asynchronous function eq.

```
sync int sum_array(sh int in[], sh int n, async int (*f)(pr int, pr int))
sync void Sum_array(sh void *in, sh int n, sh const int size, sh void *sum,
                    async void (*fu)(pr void *, pr void *, pr void *))
```

compute the sum of the elements in array in for some associative function. In case of a non-integer array, the sum function must be supplied as an update function realizing an assignment $x \leftarrow f(y, z)$ when called with pointers to elements x, y and z.

```
sync void broadcast(sh int y, sh int x[], sh int n)
sync void Broadcast(sh void *y, sh void *x,
                    sh const int n, sh const int size)
```

broadcast a value y to an n element array x.

5.5 Range preprocessing and querying

Let f be an associative function over some set S, and let $x_i \in S, 0 \leq i < n$ be a sequence of values stored in an array. A range query on the array $f[i,j], 0 \leq i,j < n$ is defined by $f[i,i] = x_i$ and $f[i,j] = f(x_i, f[i+1,j]), j > i$. The prefix library defines a *range query data structure* `RANGE`, which can be used for preprocessing arrays for answering range queries with respect to a given associative function in constant time.

```
sync void range_alloc(sh const int n, sh const int size, sh RANGE *range)
```

allocates a `RANGE` structure for an array of `n` elements of size `size`. Allocation updates the parameter `range`.

```
sync void range_preprocess(sh void *in, sh const int n, sh const int size,
                           sh RANGE *range,
                           async void (*fu)(void *, void *, void *))
```

preprocesses the arrary `in` for range querying with respect to an associative function f. The function f must be supplied as an update function `fu` implementing an assignment $x \leftarrow f(y,z)$ when called with pointers to x, y, and z. The range structure `range` must have been allocated prior to the call.

```
async pr void range_query(int i, int j, RANGE *query, void *answer,
                          async void (*fu)(void *, void *, void *))
```

answers the query $f[i,j]$ where the associative function f is supplied as an update function `fu`. The preprocessed array is given by the range query structure `query`.

```
async int range_min(int i, int j, RANGE *query)
sync void range_min_preprocess(sh int in[], sh int n, sh RANGE *range)
```

Range preprocessing and querying over integer arrays for the minimum function.

5.6 Sequential prefix operations

The sequential prefix operations have the same functionality as their parallel counterparts.

```
void seq_add(int in[], const int n, int out[], int offset)
void seq_prefix(int inout[], const int n, async int (*f)(pr int, pr int))
void seq_group_prefix(int inout[], const int n, async int (*f)(int, int),
                      int marker[], int *groups)
void Seq_prefix(void *inout, const int n, const int size,
                async void (*f)(pr void*, pr void*, pr void*))
void Seq_group_prefix(void *inout, const int n, const int size,
                      async void (*fu)(void *, void *, void *),
                      int marker[], int *groups)
```

5.7 Elementwise array operations

```
sync void copy_array(sh int in[], sh int out[], sh const int n)
sync void Copy_array(sh void *in, sh void *out, sh const int n, sh int size)
```

copy array `in` into array `out`.

```
sync void copy_subarray(sh int in[], sh int out[], sh const int n,
                        sh bool condition[])
sync void Copy_subarray(sh void *in, sh void *out,
                        sh const int n, sh const int size, sh bool condition[])
```

copy the elements of array `in` for which the corresponding entry in the `condition` array is true into array `out`. No compaction takes place.

```
sync void extract_subarray(sh int in[], sh int out[], sh int index[], sh const n)
sync void Extract_subarray(sh void *in, sh void* out, sh const int size,
                           sh int index[], sh const n)
```

the n elements given by the array `index` of increasing indices of array `in` are copied into consecutive locations of array `out`. Input and output array need not be distinct.

```
sync void permute_array(sh int in[], sh int out[],
                        sh const int n, sh int index[])
sync void Permute_array(sh void *in, sh void *out,
                        sh const int n, sh const int size, sh int index[])
```

Element i of input array `in` is permuted into position `index`$[i]$ of output array `out`. The `index` array should be a permutation of the indices from 0 to $n-1$, and input and output arrays must be distinct.

```
sync void compact_array(sh int in[], sh int out[],
                        sh const int n, sh int index[], sh bool condition[])
sync void Compact_array(sh void *inout,
                        sh const int n, sh const int size,
                        sh int index[], sh bool condition[])
```

If `condition`$[i]$ is true for index i, element i of input array `in` is copied into position `index`$[i]$ of output array `out`. It is assumed that `index`$[i] \leq i$.

```
sync void apply_func(sh int x[], sh const int n,
                     async void (*f)(pr int[], pr int))
sync void Apply_func(sh void *x, sh const int n, sh const int size,
                     async void (*f)(pr void *, sh int size, pr int))
```

apply function `f` in parallel for all indices of array `x`. The function is called with a pointer to the array, an index, and, for non-integer arrays, the element size.

```
sync void apply_binfunc(sh int x[], sh int y[], sh int z[], sh const int n,
                        async int (*f)(pr int, pr int))
sync void Apply_binfunc(sh void *x, sh const int sizex,
                        sh void *y, sh const int sizey,
                        sh void* z, sh const int sizez,
                        sh const int n,
                        async void (*fu)(pr void *, pr void *, pr void *))
```

The assignment $x \leftarrow f(y, z)$ is performed elementwise for the arrays `x`, `y` and `z`. For integer arrays, f is implemented by the function `f` which is called with elements from `y` and `z`. For arrays of arbitrary base type, f is implemented by an update function which is called with pointers to elements of `x`, `y` and `z`.

5.8 Array generators

```
sync void aritseq(sh int x[], sh const int n, sh int a, sh int b)
```

generates an arithmetical sequence $x_i = a + bi$

```
sync void recurrence(sh int a[], sh int b[], sh const int n)
```

initializes the array `b` to the solution of the first-order linear recurrence $x_i = a_i x_{i-1} + b_i, i > 0$ and $x_0 = b_0$, where a_i and b_i are given in the arrays `a` and `b`. The content of the input array `a` is also not preserved.

```
sync void random_array(sh int x[], sh const int n,
                       sh int seed[], sh const int s)
```

A primitive parallel random number generator. Given an array `seed` of `s` user supplied seeds, a sequence `x` of `n` random numbers is generated by segmenting the the output array into `s` segments, and generating random numbers for each segment sequentially. This way the generated random sequence depends only on the seeds supplied, not on the number of processors in the calling processor group.

6 The Merge Library

The operations of searching and merging of ordered arrays, and sorting of arrays are found in the merge library. To use this sublibrary the header file `merge.h` must be included. Link with `merging.o`.

6.1 Sequential integer search, merge and sort operations

```
async int bin_search_low(pr int y, pr int *x, pr int n)
async int bin_search_high(pr int y, pr int *x, pr int n)
```

searches the ordered integer array x of size n for the integer y. The search low function returns an index i such that $x_{i-1} < y \leq x_i$ (n, if $y > x_{n-1}$). The search high function returns an index i such that $x_i \leq y < x_{i+1}$ (-1, if $y < x_0$).

```
async void seq_merge(pr int x[], pr int n, pr int y[], pr int m, pr int z[])
```

merges two ordered integer arrays x and y of size n and m, respectively, into an array z. The output array of size $n + m$ must have been allocated prior to the call.

```
async void seq_mergesort(pr int x[], pr int n)
```

mergesorts the integer array x.

```
async void seq_quicksort(pr int x[], pr int n)
```

quicksorts the integer array x.

6.2 Parallel integer search, merge and sort operations

```
sync int ordered_search_low(sh int y, sh int x[], sh int n)
sync int ordered_search_high(sh int y, sh int x[], sh int n)
```

searches the ordered integer array x for the integer y in parallel. Functionality is as for the sequential binary search functions.

```
sync void merge(sh int *x, sh int n, sh int *y, sh int m, sh int *z)
```

merges ordered integer arrays x and y into an array z.

```
sync void mergesort(sh int x[], sh int n)
```

Parallel mergesort.

```
sync void quicksort(sh int x[], sh int n)
```

Parallel quicksort.

```
sync void smallsort(sh int x[], sh int n, sh int c)
```

sorts an array of integers in the range $0, \ldots, c-1$.

```
sync void smallperm(sh int keys[], sh int n, sh int c, sh int rank[])
```

computes the rank (number of elements in the array strictly smaller, or equal but preceeding) of each element of an array of integers in the range $0, \ldots, c-1$.

6.3 Parallel search, merge and sort operations

These operations are equivalent to the integer array operations just described. For arrays of arbitrary base type, order is defined by an asynchronous comparison function `cmp`, which must be supplied as parameter.

```
async int Bin_search_low(pr void *y, pr void *x, pr int n, pr int size,
                         async bool (*cmp)(pr void *, pr void *))
async int Bin_search_high(pr void *y, pr void *x, pr int n, pr int size,
                          async bool (*cmp)(pr void *, pr void *))
async void Seq_merge(pr void *x, pr int n, pr void *y, pr int m, pr void *z,
                     pr int size, async bool (*cmp)(pr void *, pr void *))
sync void Merge(sh void *x, sh int n, sh void *y, sh int m, sh void *z,
                sh int size, async bool (*cmp)(pr void*, pr void*))
sync void Mergesort(sh void *x, sh int n, sh int size, sh void *xx,
                    async bool (*cmp)(pr void *, pr void *))
```

The sorted output of the `Mergesort` procedure is given in the array `xx` of the same type as the input array. The output array must have been allocated prior to the call.

7 The List Library

The list library defines the parallel list data type `LIST`. A parallel list is essentially an array of pointers to the elements of an ordinary linked list. An ordinary list is converted into a parallel list by first creating pointers from the `LIST` array to the list elements, and then copying the linked structure from the list to the `LIST` array. Conversion between ordinary and parallel list representation is done by library calls, and requires a function for accessing the next element pointer and a function for updating the next element pointer for the elements of the (ordinary) list. After conversion into a parallel list, next pointers of the list elements can only be accessed through library calls. A parallel list is allocated as a shared array of type `LIST` with size equal to the number of elements in the list. The header file `list.h` must be included. Link with `ranking.o`.

In addition to the operations described below, algorithms for coloring and computing independent sets in lists have also been implemented. As need arises, these can be made available.

NOT YET IMPLEMENTED: Procedures for maintaining sets of lists.

7.1 Access macros and functions

Let `list` be a pointer to a list array of type `LIST`, let `i` be a (private) integer expression with value i, and let `e` be a pointer to list item e (element of the `LIST` array).

`list_elt(list,i)`	pointer to ith element of list
`elt_rank(list,i)`	rank of ith element of list
`elt_succ(list,i)`	pointer to successor of ith element of list
`elt_pred(list,i)`	pointer to predecessor of ith element of list

`RANK(e)`	rank of item e
`NEXT(e)`	next item
`PREV(e)`	previous item
`TAIL(e)`	true, if item e is the end (tail) of a list (no successor)
`HEAD(e)`	true, if item e is the head of a list (no predecessor)

When an ordinary list of elements with next pointer `next` has been converted into a parallel list, `e->next` no longer points to the next element, but instead to the parallel list item that points to element `e`. Hence `RANK(e->next)` returns the rank of list element `e`, etc.

7.2 The parallel list iterator

Since parallel lists are arrays of type `LIST`, the parallel array iterators can in most cases also serve as list iterators. There is a list iterator which in "virtual" parallel lets variables `e` and `i` step through a parallel list, `e` pointing to elements of the original list, and `i` pointing to items of the parallel list.

`par_list(e,i,n,list)`	processes elements and items of parallel list `list` in "virtual" parallel

7.3 Parallel list operations

```
sync void link_up(sh LIST list[], pr void *element)
```

The elements pointed to by the private pointers `element` are linked up into the parallel list `list`; `list[ME]` is made to point to the element supplied by processor `ME`.

```
sync void list_make(sh LIST list[], sh int n,
                    sh async void *(*next)(void *),
                    sh async void (*upd)(void *,void *))
```

converts the array of element pointers `list` into a parallel list. Asynchronous functions for accessing and updating the next element field of the list elements must be supplied. After the call, the next element field of each element points to the corresponding item of the parallel list.

```
sync void list_restore(sh LIST list[], sh int n,
                       sh async void (*upd)(void *,void *))
```

restores a parallel list into an ordinary list. A function for updating the next element pointer of the list elements must be supplied.

```
sync void apply_listfunc(sh LIST list[], sh int n, async void (*f)(void *))
```

applies the function `f` to all elements.

```
sync void list_move(sh LIST *list1, sh LIST *list2, sh int n,
                    sh int esize, sh async void (*upd)(void *, void *))
```

copies the parallel list `list1` to the list `list2`, and moves all elements from `list1` to `list2` (ie. instead of pointing to the corresponding item of `list1` each element now points to a corresponding item of `list2`). A function for updating the next pointer of the elements must be supplied.

```
sync void sublist_move(sh LIST list1[],
       sh LIST list2[], sh int index[], sh int n,
       sh async void (*upd)(void *, void *))
```

A sublist of `list1` consisting of the `n` items indexed by the `index` array is moved to `list2`. Each element of the sublist is updated to point to its corresponding item in `list2`. A function for updating the next element pointer must therefore be supplied.

```
sync void list_makecopy(sh LIST *list1, sh LIST *list2, sh int n,
                        sh int esize, sh async void (*upd)(void *, void *))
```

copies `list1` to `list2`, including copies of the elements. `esize` is the element size, and `upd` a function for updating the next element pointer.

```
sync void list_rank(sh LIST list[], sh int n)
```

ranks the parallel list `list` of length `n` using the best ranking algorithm of the library (currently the simple pointer jumping algorithm, `wyllie()`).

```
sync void krs_rank(sh LIST list[], sh int n)
```

An implementation of the list ranking algorithm of Kruskal, Rudolph and Snir [7].

```
sync void cv_rank(sh LIST list[], sh int n)
```

An implementation of the list ranking algorithm of Cole and Vishkin [2].

```
sync void wyllie(sh LIST list[], sh int n)
```

The simple pointer jumping algorithm for list ranking.

```
sync void list_rankorder(sh LIST list[], sh int n,
                         sh async void (*upd)(void *, void *))
```

permutes a ranked list into *rankorder*, that is the element with rank i will be the ith item of the parallel list `list`.

```
list_aorder(list,n,f)
sync void list_revorder(sh LIST list[], sh int n,
                        sh async void (*upd)(void *, void *))
```

permute a ranked list into *reverse rankorder*, also called *array order*, that is the element with rank i will be the $n-1-i$th item of the parallel list `list`, where n is the length of the list. Alternatively, the macro can be used.

```
sync void list_getrank(sh LIST list[], sh int n, sh int permutation[])
```

returns the rank of the elements of the parallel list `list` to the array `permutation`.

```
sync void permute_list(sh int permutation[], sh int n, sh LIST list[],
                       sh async void (*upd)(void *, void *))
```

permutes the parallel list `list` by moving the ith item to position `permutation`$[i]$.

```
sync void force_list(sh int permutation[], sh int n, sh LIST list[], sh int head)
```

forces item i of the list `list` to point to item `permutation`$[i]$. The first item of the list will be item `head`.

```
sync void force_seqlist(sh LIST list[], sh const int n);
```

forces a sequential list, ie. a list which starts at item 0 and where item i points to item $i+1$.

```
sync void list_reverse(sh LIST list[], sh int n)
```

reverses the parallel list `list`.

8 The Tree Library

The tree library defines the parallel tree data type `TREE`, as well as data types `TREE_NODES` and `TREE_EDGES`. A tree is represented by a compact array of nodes, and a compact array of edges. The edges adjacent to a node must for a consecutive segment of the edge array. A node has a pointer into the edge array to its first adjacent edge. Each edge has pointers to tail and head nodes, as well as a pointer to its reverse edge. Both nodes and edges have a pointer to an associated content block. The header file `tree.h`, as well as the list library header `list.h`, must be included. Link with `trees.o`.

NOT YET IMPLEMENTED: Procedures for making a tree binary. Procedures for manipulating forests. Procedures for adding edges and nodes to a tree.

8.1 Access macros and functions

`tree_n(t)`	number of nodes in tree
`tree_e(t)`	number of edges in tree
`node_id(v)`	integer id of node
`node_degree(v)`	degree of node
`node_first(v)`	first edge on the adjacency list of node `v`
`node_last(v)`	last edge on the adjacency list of node `v`
`node_content(v)`	pointer to contents of node
`NODE(t,i)`	the ith node in the node array
`NODE_NO(t,v)`	position of node `v` in node array
`edge_id(e)`	integer id of edge
`edge_content(e)`	pointer to contents of edge
`EDGE(t,i)`	the ith edge in the edge array
`EDGE_NO(t,e)`	position of edge `e` in edge array
	Macros for rooted trees:
`node_level(v)`	level of node
`node_height(v)`	height of node
`node_post(v)`	postorder number of node
`node_pre(v)`	preorder number of node
`node_desc(v)`	number of descendants of node
`LEAF(v)`	true, if node is a leaf
`DOWN(e)`	true, if edge is pointing downwards (from root)
`UP(e)`	true, if edge is pointing upwards (towards root)

8.2 Tree iterators

The parallel tree iterators provides primitive support for processing nodes or edges of trees in parallel. When used in a synchronous context of p processors, the node or edge array is divided into blocks of size p. The statement of the iterator is executed for each block, one block after the other.

`par_edges(e,t)`	processes the edges of tree `t` in "virtual" parallel
`par_nodes(v,t)`	processes the nodes of tree `t` in "virtual" parallel
`par_adjacent(e,v)`	steps through the edges adjacent to node `v` in "virtual" parallel

The tree iterators cannot be nested.

8.3 Parallel tree operations

```
sync TREE *tree_alloc(sh int n, sh int m)
```

allocates space for a tree with `n` nodes and `m` edges. A pointer to the tree is returned.

```
sync void tree_copy(sh TREE *t, sh TREE *nt)
```

copies tree `t` to tree `nt`. Space for `nt` must have been allocated prior to the call. The contents of nodes and edges is *not* copied, but the content pointers of the new tree are made to point to the corresponding contents of the old tree.

```
sync void tree_construct(sh TREE *t, sh int n, sh int degree[])
```

allocates space for node and edge arrays for a tree with n each with degree as given by the `degree` array.

```
sync void tree_nodealloc(sh TREE *t, sh int size)
```

allocates a contents block for each node of tree `t` of size `size`. Contents blocks are alloacted roughly evenly on the private heaps of the calling processors.

```
sync void tree_edgealloc(sh TREE *t, sh int size)
```

allocates a contents block for each edge of tree `t` of size `size`. Contents blocks are alloacted roughly evenly on the private heaps of the calling processors.

```
sync void tree_nodeinit(sh TREE *t,
                        sh sync void (*leafinit)(void *),
                        sh sync void (*nodeinit)(void *))
```

initializes the nodes of a rooted tree `t` by calling functions `leafinit` and `nodeinit`, respectively, each with a pointer to a tree node.

```
sync void tree_edgeinit(sh TREE *t, sh sync void (*edgeinit)(void *))
```

initializes the edges of a tree `t` by calling function `edgeinit` on each tree edge.

```
sync void tree_compact(sh TREE *t)
```

compacts a tree by removing all nodes and edges that are marked as deleted. A node is marked as deleted if its first edge pointer is null. An edge is marked as deleted if either its tail or head node pointer is null.

8.4 Euler tour technique and operations on rooted trees

```
sync void eulertour(sh TREE *t)
```

computes an Eulertour on tree `t` using the reverse edge pointers of the tree (must be defined).

```
sync void tree_root(sh TREE *t, sh TREE_NODE *root, sh LIST *eulerlist)
```

roots tree `t` at node `root`, and computes parent and sibling pointers, level numbers, post- and preorder traversal numbers, number of descendants etc. for all tree nodes. An Euler tour of the tree must have been computed. As a byproduct a ranked Euler list in array order is returned. Space for the Euler list must have been allocated prior to the call.

```
sync void mark_subtree(sh TREE *t, sh TREE_NODE *st, sh LIST euler[])
```

marks all nodes and edges in the subtree rooted at node `st` as deleted (for later removal).

```
sync RANGE *lca_preprocess(sh TREE *t, sh LIST euler[])
```

preprocesses tree `t` for later lowest common ancestor querying. The procedure allocates and returns a range query structure of type `RANGE` (see section 5.5).

```
async TREE_NODE *lca(pr TREE_NODE *u, pr TREE_NODE *v,
                     pr RANGE *lcaquery)
```

returns a pointer to the lowest common ancestor node of input nodes `u` and `v`.

```
sync void tree_height(sh TREE *t, sh LIST eulerlist[])
```

computes the height of tree `t` by reduction to the range query problem. An Euler list in array order must be supplied.

```
sync void bintree_contract(sh TREE *t, sh LIST *eulerlist,
                           async void (*rk)(void *,void *, void*),
                           async void (*rkrt)(void *, void *, void *),
                           async void (*unrk)(void *, void *, void *))
```

contracts and un-contracts binary tree `t` by parallel application of rake and unrake operations [2]. An Euler list in array order must be supplied. The asynchronous function `rk` is called for each rake operation with pointers to (the contents of) the leaf being raked, its parent, and its sibling. The asynchronous function `rkrt` is called when the tree has been contracted to a three node tree, with pointers to (the contents of) left child, root, and right child. The asynchronous function `unrk` is called for each unrake operation with pointers to (the contents of) the leaf being unraked, its parent and its sibling.

```
sync void bintree_height(sh TREE *t, sh LIST *eulerlist)
```

computes the height of all nodes of a complete binary tree (using tree contraction).

8.5 A tree generator

```
sync TREE *complete_tree(sh int n, sh int degree, sh int children)
```

generates a complete tree of at most `n` nodes as follows: the first node has degree `degree`. Each node generated will have `degree` new neighbors, and the `children` first of these will again have `degree` new neighbors, and so forth. Hence each node of the tree has degree either `degree`, `degree` + 1 or 1. The representation of the tree is compact, and reverse edge pointers are also generated.

8.6 Sequential tree operations

```
async void seq_dfs(TREE *t, TREE_NODE *root)
```

performs a depths first search of tree `t` and computes parent pointers, level numbers, post- and preorder traversal numbers etc. The procedure is mainly included for comparison with the parallel operation for tree rooting.

9 Graphs

The header file `graph.h` must be included. It defines the parallel graph data type `GRAPH`. Link with `graphs.o`.

NOT YET IMPLEMENTED: The graph library – connected components, biconnected components, ear decomposition, shortest paths, maximal matching – is not part of the first version of PAD.

10 Parallel Dictionaries

Parallel dictionaries are implemented using 2-3 trees. The library defines a data type `DICT_NODE` for dictionary nodes. A dictionary is a pointer to the root of a 2-3 tree of this type. In order to use the parallel dictionaries, the header file `dict.h` must be included. Programs must be linked with link with the object file `dictionary.o`.

10.1 Dictionary node operations

```
async DICT_NODE *dictitem_new(pr int key, pr void *content)
```

allocates a dictionary node with a given key on the private heap of the allocating processor.

```
async void *dictitem_newcontent(pr DICT_NODE *item, pr void *content)
```

replaces the contents of dictionary node `item` with `content` and returns a pointer to the old contents.

`dictitem_key(item)`

access macro for the key of an item.

`dictitem_content(item)`

access macro for the contents (pointer) of an item.

`dict_size(dictionary)`

access macro returning the number of elements currently in `dictionary`.

`dict_empty(dictionary)`

evaluates to false if `dictionary` is empty.

10.2 Parallel dictionary operations

`sync DICT_NODE *dict_build(sh DICT_NODE *item[], sh int n)`

builds a new dictionary from an ordered list of dictionary items of size `n`.

`sync int dict_flatten(sh DICT_NODE *dictionary, sh DICT_NODE *item[])`

flattens a dictionray 2-3 tree into an ordered array of items.

`async DICT_NODE *dict_closest(pr DICT_NODE *dictionary, pr int key)`

searches a dictionary for elements with keys given by the private expression `key`, and returns a pointer to the first dictionary node whose key is greater than or equal to `key`. Note that the function is asynchronous and can therefore only be called by one processor; parallel searching is performed by simultaneous calls to the search function.

`async DICT_NODE *dict_search(pr DICT_NODE *dictionary, pr int key)`

searches a dictionary for elements with keys given by the private expression `key`, and returns a pointer to the dictionary node whose key is equal to `key`. If no such node exists, a null pointer is returned. Note that the function is asynchronous and can therefore only be called by one processor; parallel searching is performed by simultaneous calls to the search function.

`async DICT_NODE *dict_retrieve(pr DICT_NODE *dictionary, pr int k)`

retrieves in parallel the kth smallest node of the dictionary, where $1 \leq k \leq$ `dict_size` is the value of the private expression `k`. If k is out of range, a null pointer is returned. Note that the function is asynchronous and can therefore only be called by one processor; parallel searching is performed by simultaneous calls to the search function.

```
async DICT_NODE *dict_find(pr DICT_NODE *dictionary, pr DICT_NODE *searchitem,
                           async DICT_NODE *(*sel)(pr DICT_NODE *, pr DICT_NODE *))
```

finds in parallel dictionary nodes (leaves) which match the node pointed to by `searchitem` by repeated application of the `sel` function. Starting from the root, this function is called with the `searchitem` pointer and a pointer to a dictionary node, and must return a child of this dictionary node. Note that the function is asynchronous and can therefore only be called by one processor; parallel searching is performed by simultaneous calls to the search function.

```
sync int dict_sum(pr DICT_NODE *item,
                  sh sync int (*treesum)(pr DICT_NODE *))
```

computes in parallel the value of the integer function `treesum` along the paths from the dictionary leaves `item` to the root. The function `treesum` is called for each node on the path and must return an integer.

```
sync DICT_NODE *dict_insert(sh DICT_NODE *dictionary,
                            sh DICT_NODE *item[], sh int n)
```

inserts an ordered sequence of `n` items into a dictionary, and returns a pointer to the (new) root of the dictionary. The array `item` must contain pointers to the elements to be inserted. These pointers are lost upon return.

```
sync DICT_NODE *dict_delete(sh DICT_NODE *dictionary,
                            sh DICT_NODE *oldnodes[], sh int n)
```

deletes an ordered sequence of `n` items from a dictionary, and returns a pointer to the (new) root of the dictionary.

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] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[3] Christoph W. Keßler and Helmut Seidl. The Fork95 parallel programming language: Design, implementation, application. *International Journal of Parallel Programming*, 1996. Forthcoming.

[4] Christoph W. Kessler and Jesper Larsson Träff. A library of basic PRAM algorithms and its implementation in FORK. In *8th Annual ACM Symposium on Parallel Algorithms and Architechtures (SPAA)*, pages 193–195, 1996. Research Summary.

[5] Christoph W. Keßler and Jesper Larsson Träff. Language and library support for practical PRAM programming. In *Euromicro Parallel and Distributed Processing*, 1997. Forthcoming.

[6] Donald E. Knuth and Silvio Levy. *The* `CWEB` *System of Structured Documentation*. Addison-Wesley, 1993.

[7] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10):965–968, 1985.

[8] Jesper Larsson Träff. Explicit implementation of a parallel dictionary. Technical Report SFB 124-D6 10/95, Universität des Saarlandes, Sonderforschungsbereich 124, VLSI Entwurfsmethoden und Parallelität, 1995. 53 Pages.

[9] Jesper Larsson Träff. List ranking and other operations on parallel lists. Technical report. In preparation, 1996.

[10] Jesper Larsson Träff. PAD: A library of basic PRAM algorithms. Manuscript, 1996.

[11] Jesper Larsson Träff. Parallel searching, merging and sorting. Technical Report SFB 124-D6 1/96, Universität des Saarlandes, Sonderforschungsbereich 124, VLSI Entwurfsmethoden und Parallelität, 1996. 45 Pages.

[12] Jesper Larsson Träff. Parallel tree operations. Technical report. In preparation, 1996.

[13] Jesper Larsson Träff. Prefix sums and other operations on parallel arrays. Technical Report SFB 124-D6 6/96, Universität des Saarlandes, Sonderforschungsbereich 124, VLSI Entwurfsmethoden und Parallelität, 1996. 49 Pages.