# Fork

## Quick Reference Card

Christoph W. Kessler

Universität Trier
FB IV – Informatik, 54286 Trier, Germany
kessler@psi.uni-trier.de
http://www.informatik.uni-trier.de/~kessler

November 1999

**Fork** is the short name for **Fork95** version **2.0** [1]. **Fork95** [2] is an imperative parallel programming language based on the Arbitrary CRCW PRAM model. It is an extension of ANSI-C and supports a rich variety of parallel algorithmic paradigms and programming techniques. A compiler is available for the **SB-PRAM** [1], a scalable hardware realization of a Multiprefix Priority CRCW PRAM with 2048 processors at the University of Saarbrücken, Germany; a pre-prototype with 512 processors is already operational. A software simulator and other system software of the **SB-PRAM** is publically available.

This reference card gives a short overview of Fork. For further informationplease look at the Fork homepage: `www.informatik.uni-trier.de/~kessler/fork95`

## What Fork adds to C

- **SPMD Execution** Each processor has a copy of the program. The number of processors remains fixed during execution.
- `__STARTED_PROCS__` Global run–time constant holding the number of available PRAM processors.
- `__PROC_NR__` Global processor ID variable, read–only, consecutively ranging from 0 to `__STARTED_PROCS__`$-1$.
- **Shared and Private Address Subspaces** The PRAM's shared memory is partitioned into a shared and `__STARTED_PROCS__` private address subspaces.
- **Shared Variables** declared (only in →synchronous program regions) by storage class qualifier `sh`. The scope of sharing is the group active at the declaration.
- **Private Variables** declared by storage class qualifier `pr` (optional), reside once in each processor's private memory subspace.
- **Concurrent Write** conflict resolution scheme (Arbitrary CRCW) is inherited from the underlying hardware. With the SB-PRAM, the processor with highest hardware ID succeeds (Priority CW).
- **Asynchronous Function** declared by type qualifier `async` (default, i.e. optional)
- **Synchronous Function** declared by type qualifier `sync`. Only synchronous functions can have shared formal parameters.
- **Straight Function** declared by qualifier `straight`.
- **Asynchronous Program Regions** Asynchronous functions or `farm` bodies, excluding `start` and `join` bodies inside. An asynchronous or straight call to a synchronous function is forbidden.
- **Synchronous Program Regions** Synchronous functions and `start` or `join` bodies, excluding `farm` statements inside. A synchronous or straight call to an asynchronous function is automatically casted to an asynchronous region.
- **Straight Program Regions** Straight functions, excluding `farm`, `start`, or `join` statements inside. No private branch conditions. Callable from any region.
- **Asynchronous Mode of Execution** applicable to asynchronous and straight program regions. The current group is inactive (no shared variables or group heap objects declarable, no implicit synchronization points). All processors of the current group are continuously available, thus `barrier` and locks can be used as usual.
- **Synchronous Mode of Execution** applicable to synchronous and straight program regions. Maintains the **Synchronicity Invariant (SI)**: All processors in the same (active) group work synchronously, i.e. their program counters are equal at any time (→Group concept).
- **The `farm` statement** switches from synchronous to asynchronous mode for execution of its body. Implicit group–wide (exact) barrier at the end of the body.
- **The `join` statement** collects asynchronously operating processors and switches to synchronous mode for the execution of its synchronous body.

```
join ( SMsize; delayCond; stayInsideCond )
       bodyStatement;
  else missedStatement;
```

  Only one group can operate on the body of a `join` statement at any time [3]. The first processor arriving at the `join` allocates `SMsize` words of shared memory for the new group's shared stack and group heap. While `delayCond` evaluates to a nonzero value for this first processor, other processors are allowed to join the new group. After the wait phase, processors that evaluate `stayInsideCond` to zero leave the new group and execute `missedStatement`, while the others synchronize and execute `body` in synchronous mode. While this group operates on the body, other arriving processors execute `missedStatement`. A `continue` in `missedStatement` jumps back to the `join` entry, a `break` leaves the `join` statement.
- **The `start` statement** switches from asynchronous to synchronous mode for *all* processors of a group.
- **Group Concept** At program start, all processors belong to the same group. In →synchronous mode, the →SI is maintained automatically for each active group. Active groups may allocate shared variables and objects.
- **Group Hierarchy Tree** In →synchronous mode, active groups can be →split into subgroups. Hence the groups form a tree–like hierarchy, where the active groups are the leaves.
- **Group size** accessible in synchronous regions as the shared run–time constant `#`.
- **Group rank** of a processor, ranging from 0 to `#-1`, accessible in the private run–time constant `$$`. Automatic re-ranking at each group activation or reactivation.
- **Group ID** denoted by the shared variable `@`. Set at group splitting constructs, saved/restored automatically.
- **Group–relative Processor ID** denoted by the private variable `$`. Set by `start`, `join`, and by the programmer (`$=...;`), saved/restored automatically.
- **Automatic Group Splitting** Potential divergence of control flow at `if` statements or loops with a *private* branch condition causes the current group to become inactive and split into child groups, one for each branch target. To these the SI applies only internally. The former group is reactivated when control flow reunifies again (implicit group–wide exact barrier).
- **The `fork` statement** (only in synchronous regions)

```
fork(exp1; @=exp2; $=exp3) body;
```

  with $g$ as value of shared expression `exp1`: deactivates the current group and splits it into $g$ subgroups numbered 0,...,$g-1$. Each processor evaluates the private expression `exp2` to a value $j$ and joins the subgroup with ID $j$ to execute the `body`. If $j < 0$ or $j \geq g$, the processor skips the `body`. `$=exp3` (optional) may renumber the processor ID's within the subgroups. The SI (see synchronous mode) applies to each subgroup internally. The subgroups execute `body` concurrently. At the end of `body`, the former group is reactivated (implicit group–wide barrier).
- **The `barrier` statement** performs an explicit group–wide (exact) barrier synchronization.

- **Multiprefix Operators** are atomic integer expression operators (not functions). Relative order of execution within the same PRAM step according to machine–specific (SB-PRAM) multiprefix rank.
  `k=mpadd(&shvar,expr)` multiprefix addition,
  `k=mpmax(&shvar,expr)` multiprefix maximum,
  `k=mpand(&shvar,expr)` multiprefix bitwise and,
  `k=mpor(&shvar,expr)` multiprefix bitwise or.
- **Atomic Update Functions** similar, no return value:
  `void syncadd(&shvar,expr)` atomic increment, etc.
- **Group Heap** A shared memory block of size `k` words shared by the current group is allocated by
  `sync char *shalloc(sh unsigned int k)`
  and lives as long as the group that allocated it.
  `sync void shallfree()` frees all objects `shalloced` so far in the current function.
- **Private Heap** As in C, private memory is dynamically allocated by `malloc()` and freed explicitly by `free()`.
- **Global Shared Heap** A shared memory block of size $k$ allocated by a processor executing
  `async void *shmalloc( pr unsigned int k )`
  and lives until freed explicitly by `shfree()`.

## Programming Techniques and Libraries

- **Statically scheduled parallel loop**

```
int i;
forall ( i, lb, ub, # )  stmt(i);
```

  where the **forall macro** expands to

```
for (i=lb+$$; i<ub; i+=#)  stmt(i);
```

  Similar: `Forall` (for stride $> 1$), `forall2` (two-dimensional flattened loop), `Forall2` (for strides $> 1$).
- **Dynamically scheduled parallel loop**

```
int i;
sh int ct = 0;
for (i=mpadd(&ct,1);i<N;i=mpadd(&ct,1))
   stmt(i);
```

  or, using the **FORALL macro**,

```
FORALL( i, &ct, 0, N, 1);
```

- **Parallel Divide-and-Conquer**

```
sync sometype DC( sh int n; ...)
{ sh int d;  pr int i;
  if (trivial(n)) return conquer( n, ...);
  if (#==1) return seqDC( n, ... );
  d = divide( n, ...);
  if (#<d)
     farm forall(i,0,d,#) seqDC( n/d,...[i]);
  else
     fork( d; @=$$ % d; )
        DC( n/d, ...[@]);
  return combine( n, d, ...);
}
```

- **Synchronous and Asynchronous Pipelining** along a graph, **MPI Message passing** (see `util` directory), asynchronous **Task Queue**, ... see [1].
- **Skeleton functions** for all these paradigms see [1].
- **APPEND Library** of asynchronous parallel data structures like parallel hashtables, parallel randomized search tree, parallel skip list, parallel priority queue: contained in the `util` directory of the Fork package, see [1].
- **PAD library** of synchronous PRAM algorithms and data structures by J. Träff [1,4] covers searching, merging, sorting etc., parallel dictionaries, lists, trees, graphs.

## Important Standard Library Routines

- `straight int groupsize()` size of my current group
- `sh SimpleLock` $l$ `= new_SimpleLock();` create and initialize a simple lock object $l$
- `void simple_lock_init(SimpleLock` $l$`);` re–initialize an allocated shared `SimpleLock` object `l`
- `void simple_lockup(SimpleLock` $l$`);` lock $l$
- `int simple_unlock(SimpleLock` $l$`);` unlock $l$
- `sh FairLock` $l$ `= new_FairLock();` create and initialize a fair lock object $l$
- `void fair_lock_init(FairLock` $l$`);` initializes $l$
- `void fair_lockup(FairLock` $l$`);` lock $l$
- `int fair_unlock(FairLock` $l$`);` unlock $l$
- `sh RWLock` $l$ `= new_RWLock();` create and initialize a readers–writers lock object $l$
- `void rw_lock_init(RWLock` $l$`)` re–initialize $l$
- `void rw_lockup(RWLock` $l$`, int` $m$`)` $m$-lock $l$
- `void rw_unlock(RWLock` $l$`, int` $m$`, int` $w$`);`
- `sh RWDLock` $l$ `= new_RWLock();` create and initialize a readers–writers–deletors lock object $l$
- `void rwd_lock_init(RWDLock` $l$`);` re–initialize $l$
- `int rwd_lockup(RWDLock` $l$`, int` $m$`);` $m$-lock $l$
- `void rwd_unlock(RWDLock` $l$`, int` $m$`, int` $w$`);` $m$-unlock $l$. $m \in$ {`RW_READ, RW_WRITE, RW_DELETE`}
- `int getct()` returns the current value of the clock cycle counter (1 cc $= 4\ \mu$s on the SB-PRAM).
- `initTracing(` $k$ `);` initialize trace buffer of size $k$
- `startTracing();` start logging events (`-T`)
- `stopTracing();` stop logging events (`-T`)
- `writeTraceFile(`*filename, comment*`);` for `trv`

## Special Include Files

- `fork.h` (group heaps, locks, parallel loop macros)
- `stdlib.h` (shmalloc/shfree, qsort, ...)
- `io.h` input/output routines (on SB-PRAM host)
- `syscall.h` interface to PRAMOS / simulator OS

## Important Compiler Options

- `-A` emits more warnings, `-A -A` even more
- `-c` suppresses linking
- `-g`, `-g1`, `-g2` generate various levels of debug code
- `-I`*path* specifies path for include files
- `-m` align shared memory accesses with modulo flag to avoid simultaneous reading and writing to same cell (usually necessary)
- `-o` *name* renames the output file (default: `a.out`)
- `-S` suppresses deletion of the assembler file
- `-T` (also for linking) generates tracing code (for `trv`)

## Graphical Trace File Visualizer

`trv` *filename*`.trv` creates *filename*`.fig`, a graphical visualization in FIG format. Gantt chart, statistics for shared memory accesses, idle times at barriers and locks.

`fig2dev -Lps` *filename*`.fig` > *filename*`.ps` can be used to generate postscript images, `xfig` for editing.

`trvc` is a variant of `trv` for color graphics devices.

## Online Software and Documentation

by anonymous ftp, either via the web homepage or directly at `ftp.informatik.uni-trier.de` in directory `/pub/users/Kessler`. Includes documentation, all sources, and example programs. There is also a distribution of the SB-PRAM system software tools including assembler, linker, loader and simulator.

## Introductory Literature on Fork


[1] J. Keller, C. W. Kessler, J. L. Träff. Practical PRAM Programming. Textbook, 550 p., Wiley, to appear in 2000.

[2] C. W. Kessler, H. Seidl. The Fork95 programming language: Design, Implementation, Application. Int. Journal on Parallel Programming, **25**(1), pp. 17–50, Plenum Press, 1997.

[3] C. W. Kessler, H. Seidl. Language Support for Synchronous Parallel Critical Sections. Proc. APDC'97 Int. Conf. on Advances in Par. and Distr. Computing, Shanghai, March 19–21, IEEE CS press, 1997.

[4] C. W. Kessler, J. L. Träff. Language and Library Support for Practical PRAM Programming. *Parallel Computing* **25**(2) pp. 105–135, Elsevier, 1999.