`fcc` Fork95 Compiler Reference Manual

Christoph W. Keßler
Fachbereich 4 – Informatik
Universität Trier
D-54286 Trier, Germany
e-mail: `kessler@psi.uni-trier.de`

Current version: 2.0

released: Aug 20, 1999

The SB-PRAM is a lock–step–synchronous, massively parallel computer with a (from the programmer's view) physically shared memory [1, 10, 19].

Fork95 [11, 12, 13] is a C–based redesign of the PRAM language `FORK` proposed in [8]. For an in–depth introduction to programming in Fork95 we recommend the Fork95 tutorial [18]. A library of basic PRAM algorithms and data structures implemented in Fork95 is described in [14, 15].

The most recent version of the language, **Fork95 version 2.0** (Aug. 1999), is just denoted as Fork. It is described in the forthcoming textbook *Practical PRAM Programming* by J. Keller, C. Keßler, and J. Träff, to appear in spring 2000 at Wiley, New York.

`fcc` is the `Fork95` compiler for the SB-PRAM. The `fcc` implementation is partially based on the `lcc` compiler [5, 6, 7] for ANSI-C [3]. Parts of the standard library routines have been taken from the GNU C library, while others have been completely rewritten from scratch. I have also completely rewritten the compiler driver. Finally, version 1.9 features a graphical trace file visualization tool, `trv`.

This report describes installation, use, and implementation principles of `fcc`.

1 Installing fcc

1.1 Hardware platforms

`fcc` and the SB-PRAM system tools run on SUN workstations under SunOS and Solaris.

In principle `fcc` should also compile for LINUX machines, but the SB-PRAM assembler `prass` and linker `ld` which are called by the compiler were, up to now, not LINUX–proof. A port of the system tools for LINUX is currently in a testing stage by the SB-PRAM group at Saarbrücken.

Prof. E. Mayordomo from the University of Zaragoza, Spain, reports on a successful installation of both compiler and system tools on an HP K250/3 with 3 PA8000 processors (512 MB memory and 20 Gb disk) under HPUX 10.20.

Please notify us if you manage to install `fcc` and/or the system tools on a platform different from these listed above.

1.2 Compatibility problems

For the installation of `fcc` and the SB-PRAM system tools you need an ANSI C compiler like the GNU C compiler.

Make sure that you are using the version of the SB-PRAM system tools which is distributed at the Fork web page. Recent versions of e.g. the linker and the simulator distributed at Saarbrücken are no longer compatible with Fork, since these contain bug fixes for the current SB-PRAM prototype and require the existence of the SB-PRAM operating system PRAMOS, which is neither required nor compatible with Fork95 programs.

1.3 Installation procedure

Extract the distribution into its own directory. All paths below are relative to this directory. It includes the following top-level directories;

`src`	source files of proper compiler and driver
`man`	man page
`doc`	documentation
`include`	include files
`examples`	example programs, example makefile
`bin`	driver
`lib`	proper compiler, libraries, compiler-specific files
`util`	useful Fork routines (parallel data structures, MPI)

Before installing, you should specify two shell variables in your `.cshrc` file that are required to set up the paths properly: Set `PRAMDIR` to the directory where you have installed the SB-PRAM system software, and set `FORKDIR` to the directory where you have located the Fork95 distribution, e.g.,

```
 setenv PRAMDIR /myhomedirectory/pram
 setenv FORKDIR /myhomedirectory/fork
```

Add the `$FORKDIR/bin` and the `$PRAMDIR/bin` directory to the search path in your local environment specification.

To recompile the compiler for your system, go to the `fork` directory. Run `make all`. This installs `rcc`, the proper Fork95 compiler, in directory `lib`, and `fcc`, the driver, in directory `bin`.[1] Furthermore, several library object files are created in the `lib` directory by the `fcc` compiler just built, which are required at the link phase of `fcc`. In addition, the example programs in the `examples` directory are built using `fcc` (there should be warnings, but no errors), and this documentation is also created in the `doc` directory if not yet available.

The resulting executables (see the section on the example programs) should run properly on the SB-PRAM simulator `pramsim`. Call it by `pramsim` *executable_file*.

To run `pramsim` properly, you need the resource files `.ldrc` and `.pramsimrc` in your current working directory. The `examples` directory contains suitable versions. Nevertheless you may need to adapt `.ldrc` to your purposes. Set up the resources for number of processors and sizes of private and shared memory subspaces in the file `.ldrc` depending on your local workstation's performance and memory resources. Note that the

[1]Note that compilation of `rcc` requires an ANSI C compiler like the GNU C compiler. You must specify this compiler in the `Makefile` first. If you want to change the internal directory structure, you will have to edit `fcc.h` and maybe also the `Makefile` to adjust the paths according to your wishes. You also will have to edit `fcc.h` if you prefer, for compiling Fork95 programs, a C preprocessor other than the GNU preprocessor from your current `gcc` installation.

memory requirements are to be indicated in PRAM words, where a word is 4 bytes. The number p of simulated processors can be changed interactively in the simulator by typing the command `init` u, v with $1 \leq u \leq 128$, $1 \leq v \leq 32$, where $p = u \cdot v$. A forthcoming version of `pramsim` will allow to change memory sizes interactively. A window–based user interface is in preparation.

Finally, install the man page by adding `$FORKDIR/man/man1` to your `$MANPATH`.

2 Running fcc

You can use `fcc` in a way similar to your standard C compiler `cc`. `Fork95` sources can be compiled only if the `-FORK` compiler flag is set (see the options below). For instance, you may compile one of the demo example programs by typing

```
fcc -I$FORKDIR/include -FORK -m parprefix.c
```

which generates an executable `a.out`. (To simplify this command, use `make` or write a shell alias.)

After starting `pramsim`, type the command

```
g
```

to execute the entire program. You can stop the simulator with Ctrl-C. The simulator automatically stops if a processor reaches the end of the program, or if the C function `exit()` is called by a processor. — Alternatively, you can step through the program execution (see the simulator `help` menu for details).

Example programs

The `example` directory contains several example programs:

- `bus.c` is a test for the `join` statement implementing a very simple parallel shared heap memory allocator.
- `conjgrad.c` is an implementation of a conjugate gradient solver. It uses the header file `bpl.h`. No output is provided.
- `crcwqs.c` is an implementation of the parallel CRCW quicksort algorithm by Chlebus/Vrto (JPDC 1991).
- `eratosthenes.c` is an implementation of the prime number sieve by Eratosthenes. It uses synchronous mode for the sieve itself and then computes prime twins using a self–scheduled parallel loop in asynchronous mode. See also the files `primes.sync.c`, `primes.async.c` and `primes.balanc.c` for computation of prime twins using the standard algorithm in various modes.
- `fft.c` Fast Fourier Transform for complex arrays
- `gauss.c` Gaussian Elimination
- `hello.c` prints “Hello world” by each processor.
- `mpadd.c` is a demonstration of `Fork95`’s built–in `mpadd()`–operator.

  Note: Be careful with `mpadd(&shvar,somevar)`: It includes one side–effect for each of its two parameters: `shvar` (first parameter) is set to the global sum of all `somevars`, and `somevar`, if a variable, is set to the result value. These side effects are part of the SB-PRAM’s multiprefix operators and cannot be changed.

- `parprefix.c` shows, at the example of parallel prefix sum computation, how parallel programs should be written to become independent of the number of processors.
- `philo.c` is an implementation of the dining philosophers problem. See also `philo.naiv.c` for a version that is not deadlock–free.
- `pipe.c` demonstrates how pipelining could be programmed in Fork95 at the problem of computing the global sum of n elements.
- `quicksort.c` is a parallel quicksort implementation for n array elements on p processors which allows arbitrary values for n and p.
- `mergesort.c` is a parallel divide–and–conquer implementation of mergesort. This implementation assumes for simplicity that all array elements are different.
- `quickhull.c` is a parallel divide–and–conquer algorithm that computes the convex hull of n points in the 2D Euclidean plane. It also demonstrates the usage of the simple graphics interface: it plots the points, the edges of the resulting hull, and all bisection lines at recursive calls, visualizing the quality of pivoting. You may view `HULL.xpm` e.g. by `xv`.
- `koch.c` is a parallel implementation of a recursive plotting algorithm for fractals known as Koch curves. View the resulting picture `KOCH.xpm` using `xv`.
- `qsort.c` demonstrates the random number generator (seed it, if at all, only by a large, odd, processor–individual number, please!) and the (sequential) `qsort()` routine from `<stdlib.h>`.
- `strassen.c` implements Strassen's well–known recursive algorithm for matrix–matrix–multiplication. The divide–and–conquer step is parallelized using the `fork` statement.
- `pqueue.c` implements a parallel queue with `put` and `get` operations. It can be used e.g. for programming task queues which can be accessed at low overhead and with practically no sequentialization.
- `mandel.c` computes the Mandelbrot set and generates a graphical output in `X-pixmap` format. Julia sets are computed and drawn by `julia.c`.
- `queens.c` is a synchronous implementation of the N-Queens problem. The `join` statement is used to aggregate a certain number (up to 8) of solutions for synchronous line–wise graphical output.
- `aqueens.c` is an asynchronous implementation of the N-queens problem, based on the parallel FIFO queue as parallel task queue implementation. Parallel output is as in `queens.c`.
- `divconcmerge.c` is a mergesort implementation based on the `divide_conquer` skeleton routine.
- `dcsum.c` is a parallel sum implementation based on the `divide_conquer` skeleton routine.
- `recdoub.c` implements the recursive doubling method for solving first–order linear recurrence equations.

Possible extensions to multiprefix operators are discussed in the files `mpmax.c` and `incl_mpadd.c`. Thanks go to Marcus Marr from Edinburgh for providing these.

More complex Fork95 software (library routines for sorting, searching, etc.) is available in the PAD library [14, 15]; see the PAD web page

`http://www.mpi-sb.mpg.de/guide/activities/alcom-it/PAD/`

header file	type of library functions
`fork.h`	(mandatory) group heap; locks; parallel loop macros
`assert.h`	the `assert()` macro
`stdlib.h`	as in C (`rand()`, `qsort()`, etc.
`string.h`	string handling functions
`math.h`	mathematical functions (only single precision)
`ctype.h`	character classification (as in C)
`stdarg.h`	macros for handling a variable number of function arguments
`io.h`	input/output routines
`graphic.h`	graphics routines
`myfcntl.h`	file I/O flags from PRAM-OS
`syscall.h`	routines for interaction with PRAM-OS

Table 1: The library header files for the Fork95 compiler. In the present stage not all functions known from usual ANSI C libraries are available yet.

or ask Jesper L. Träff at MPI Saarbrücken (email: `traff@mpi-sb.mpg.de`).

The asynchronous complement to PAD is my APPEND library of asynchronous data structures like parallel hashtable, parallel FIFO queue, parallel skip list and (soon) parallel random search tree. It is distributed with this Fork95 compiler package in the `util` directory. It features a pseudo–object–oriented programmer interface for the implemented operations. A thorough description will be given in our forthcoming textbook.

3 Options

- `-g0` drops debugging directives; `-g`, `-g2`...`-g9` emit several levels of dbx–type debugger directives.
- `-G`*path* tells the compiler the path of the source program which is required when `-g` is set.
- `-K` generates code that checks for stack overflow.
- `-m` generates code for alignment according to the MODULO flag of the SB-PRAM. This should be switched on if you use concurrent write facilities or multiprefix operators.
- `-P` prints full type headers for each function declaration.
- `-T` causes instrumentation of the code to collect profiling information about shared memory access statistics. This information can either be printed to the screen by `printAccStat()` (now deprecated). The other variant, which looks much nicer, is to use the tracing commands and `trv`/`trvc` to see the information in the graphical visualization of the trace file.
- `-A` emit more warnings (e.g., unused variables, potential programming errors etc.), `-A -A` emit even more warnings.

Further options taken from `lcc` can be looked up in the `fcc` or `lcc` manpage.

4 Library Functions

The file `forklib2.asm` contains the PRAM assembler source of the startup code and several functions closely related to the Fork95 language: The other C sources, in particular `async.c`, contain C library functions and wrapper functions for routines in `forklib2.asm`.

- `malloc` (private malloc on permanent private heap, asynchronous),
  `shalloc` (shared malloc on automatic group–local shared heap),
  `shmalloc` (allocate on global permanent shared heap),
  `realloc` (realloc on permanent private heap, dummy),
  `free` (free malloc objects, dummy),
  `shallfree()` (free all shalloc'ed objects in current function),
  `shfree` (free shmalloc'ed objects).

  The `shalloc()`ed objects generally live until the group defining them terminates, although there is also a possibility for explicit release: `shallfree()` releases all objects `shalloc()`ed so far in the current function (applicable to synchronous functions only).

  `pravail` returns the number of free private memory words for this processor, and `shavail` returns the number of free shared memory words for the current group.

- `exit` to leave the program, `atexit()` to register cleanup functions that should be called when exiting by `exit`

- `barrier` (straight, internally called `forklib_sync`) synchronizes the current group.

- `groupsize` (number of processors in current group, straight, by inspecting the synchronization cell).

- `forklib_divi` (integer division), `forklib_divu` (unsigned integer division), and `forklib_fdiv` (floatingpoint division) are for compiler–internal use only; `itof`, `ftoi` (conversions int–float, straight),

- `memcpy` (internally: `forklib_movb`) for block copy, `strcpy`, `strcmp`, `strncmp`, `memcmp`, `strlen`.

- `srand`, `rand`, `random`; the latter routine takes the address of the seed as parameter and stores the new random number there as a side effect.

- `read`, `write`, `open`, `close` etc.: the syscalls.

- `reltoabs` computes an absolute address from a base–relative (private) address. This may be required if `asm()` is used to read private global addresses. The compiler generates only absolute addresses for private variables and heap objects.

- `syncadd()`, ..., `syncor()` provide the corresponding SB-PRAM operators as library functions. `syncadd_m0` denotes a syncadd that is performed only when the modulo bit is cleared, and `syncadd_m1` is performed only when it is set. Corresponding modulo–sensitive functions exist for syncmax, syncand, and syncor.

- `getct` returns the current value of the (global) round counter of the SB-PRAM, giving 256 times the parallel execution time (consumed since start of the program) in milliseconds.

- `printAccStat` in `io.h` prints an ASCII-text summary of the executing processor's shared memory accesses and barriers to the screen. This requires that the user program to be examined has been compiled with the option `-T`. This information may be used to estimate the performance on non-PRAM architectures. The function is now deprecated, use the tracing routines instead.

Internally defined library functions that are not accessible to the Fork programmer take their parameters from parameter registers, not from the (private) stack. If necessary, there are wrapper Fork routines that provide a proper call interface to them.

Four different kinds of locks have been implemented: simple locks, fair locks, reader-writer-locks, and reader-writer-deletor locks, the first three of which were suggested by Jochen Röhrig at Saarbrücken University. The locks are documented in <fork.h>. To use them for mutual exclusion in asynchronous mode, we recommend the macros given in <fork.h>.

I/O routines are declared in the io.h header file. Input/Output to/from screen can be done using the usual scanf() and printf() functions. File–I/O has been implemented from scratch, partially by reusing sources from other C libraries, also fscanf() works but sscanf() still does not work properly. Buffering is not provided. sprintf() and vsprintf() are still missing. All I/O routines are asynchronous. Make sure to place I/O functions in critical sections to keep your screen display readable. A unified locking mechanism for FILE I/O is planned. For faster printing of strings and floatingpoint numbers you can use the routines puts() resp. prF(), see io.h. In addition, there is the routine pprintf() which works like printf() (no floatingpoint output) but emits the physical processor ID at the beginning of the printed line. This may be useful for debugging.

The mathematical library still needs much work to be done, see the math.h header file. The implemented functions work for single precision floatingpoint numbers (IEEE 754 format); the double keyword is automatically converted to float in order to make things work at least for single precision. The available functions are: fabs(), ceil(), floor(), modf(), ftoi() and itof(). The trigonometric functions sin(), cos(), asin(), acos(), atan() are available as well as the base 10 logarithm log10() and the base e logarithm ln(). Most domain errors are checked for. The routines have been tested; though no guarantee is given for the accuracy.

Profiling and tracing is now also available. Tracing is initialized by initTracing(bufsiz), taking the trace buffer size as parameter (10000 to 100000 should be a good choice). A piece of program to be traced should be preceded by a startTracing() and ended by a stopTracing() call. The trace buffer and the collected profiling information is written to a file with the call writeTraceFile(*filename*, *title*), which takes as parameters the file name and an optional title string. The trace file can be converted to a FIG file by calling the trv tool with the file name *filename* as argument. The resulting fig file *filename*.fig can be edited and converted to other formats by xfig. The version trvc produces a more colored image which is better suitable for screen display, while the trv output is better suitable for greyscale printers.

We have also provided some simple graphics routines. A two–dimensional array of 32-bit pixels with extents x and y is allocated by init_pict(x,y). One may switch between several pixel arrays with switch_to_pict(*cp*) to *cp* as the current pixel array. The picture can be cleared by clear_pixels() in parallel. A single pixel (u, v) may be set by set_pixel(u,v,1) and its color be inspected by get_pixel(u,v). The parallel routine line() plots lines. The parallel routine write_pixmap(*filename*), writes the pixel array to a file in monochromatic X-Bitmap format (which may be viewed e.g. using *xv*). The origin (0,0) of the pixel array is mapped to the upper left corner of the picture. For more details please see the tutorial [18]. An advanced graphics library for Fork95 is currently in preparation.

5 Reserved key words

Additionally to the ANSI C key words, the words sh, pr, start, join, fork, farm, sync, async, straight, ilog2, mpadd, mpmax, mpor and mpand are reserved key words, and this holds also for the library function names.

The sign $ denotes the (relative) process ID, $$ the group rank, and @ the group ID of the processor executing this expression. $ is a private integer variable that can be assigned any integer value. $$ is a constant private integer variable (read–only) that is automatically set by the system to a unique integer between 0 and the current group size minus one. `@` is a constant shared integer variable (read–only) that is automatically set by the system and allows to distinguish the subgroups of a split group with respect to each other. `#` is a constant shared integer variable that holds the current group size.

The number of processors started in the simulator can be accessed by the programmer in the variable `__STARTED_PROCS__`. The variable `__PROC_NR__` contains a unique (absolute) processor number for each processor, ranging from 0 to `__STARTED_PROCS` – 1. The group–relative process ID `$` is initialized to `__PROC_NR__` at the beginning of the program execution. Although `__PROC_NR__` may later be assigned a (different) value by the programmer, he should not change it; there is `$` for this purpose.

The names of standard library functions cannot be overloaded by user routine names. In this sense, they are also reserved key words.

The header file `<fork.h>` which should be included into each Fork95 source program contains also useful macros like

```
forall(i,lb,ub) <statement>
```

that executes `<statement>` within a parallel loop with loop variable `i`, ranging from `lb` to `ub`, using all `__STARTED_PROCS__` started processors. This is especially useful for dataparallel applications.

6 The bus tour concept: `join`

The `join` statement, first introduced in [17], is now fully integrated into the compiler. See the Fork95 Tutorial for an in–depth description of its semantics and the excursion bus analogy.

```
join ( SMsize; delayCond; stayInsideCond) <busTour> else <otherWork>
```

where

- `SMsize` is an expression evaluating to an integer that determines the shared stack size (in memory words) for the group executing the bus tour. It should be at least 100 for simple "bus tours".
- `delayCond` is an integer expression evaluated by the "driver" (i.e., the first processor arriving at the `join` when the bus is waiting). The driver waits until this condition becomes nonzero.
- `stayInsideCond` is an integer expression evaluated by all processors in the "excursion bus". Those processors for which it evaluates to zero branch to the `else` part. The others barrier–synchronize, form a group, and execute `busTour` in synchronous mode.
- `delayCond` and `stayInsideCond` can access the number of processors that meanwhile arrived at this `join` statement by the macro `__NUM_PR__` and their own rank in this set of processors (their "ticket code") by the macro `__RANK__`.
- `busTour` is a synchronous statement.
- the `else` branch is executed by the processors missing the "excursion bus" or springing off when testing the `stayInsideCond` condition. `otherWork` is an asynchronous statement.
- Inside the `else` branch, processors may return to the entry point of this `join` statement by `continue`, and leave the `join` statement by `break`. This syntax corresponding to loops in C is motivated by the fact that the `else` clause indeed codes a loop that is executed until the processor can participate in the following bus excursion or leaves the loop by `break`.

- The `else` clause is optional; a missing `else` clause is equivalent to `else break;`

Example:

```
join( 100; __NUM_PR__>=1; __RANK__<2)
   farm pprintf("JOIN-BODY! $=%d\n",$);
else {
   pprintf("ELSE-PART!\n");
   continue;  /* go back to join head */
              /* break would leave it */
}
```

The `join` statement is used e.g. in the example programs `jointest.c` and `queens.c`.

7 Reserved registers

`fcc` generates symbolic names for the special-purpose-registers. Registers `r18`,...,`r29` are general–purpose registers that can be used for expression evaluation.

number	symbolic	name	comment
0	`sr`,`R0`	status register, Zero	depending on the instruction
1	`pc`	program counter	ignored as target of compute
2	`sp`	system stack pointer	do not change it!
3	`fpp`	frame pointer, private	
4	`app`	argument pointer, private	
5	`Ret`	value/address of function return value	
6	`spp`	stack pointer, private	
7	`par1`	first parameter register	
8	`par2`	second parameter register	
9	`par3`	third parameter register	
10	`par4`	fourth parameter register	
11	`epp`	heap pointer, private	
12	`sps`	stack pointer, shared	
13	`fps`	frame pointer, shared	
14	`eps`	heap pointer, shared	
15	`gpp`	group pointer, private	
16	`gps`	group pointer, shared	
17	`aps`	argument pointer, shared	
18...29	`r18`,..., `r29`	general purpose registers	saved at function calls
30,31	`r30`, `r31`	scratch registers	not saved at function calls

The parameter registers are used only to call internal library functions and operating system functions. They can be used as scratch registers elsewhere.

8 Frame layout

8.1 Procedure frames

8.1.1 Parameter passing at function calls

Functions listed in the file `libfnames` pass their first four private parameters in registers `par1`,...,`par4` (this is mainly to optimize calls to library routines). The `fcc` user can take advantage of this scheme by adding own function's names to this file. However this does not work when these functions should be called by function pointer dereferencing. The first line in `libfnames` contains the total number of its entries, with each line containing one function name.

Functions with a variable number of arguments pass their private parameters located on variable argument positions via the private stack. Shared parameters must not be located on variable positions.

A function using shared parameters must have an ANSI–style prototype.

`fpp` and `fps` are set even if the callee has no (shared) local variables. This is to restore the group pointers properly after a `return` statement in the callee. There is an exception of this, however, if the callee is an asynchronous function.

8.1.2 Private procedure frame for synchronous functions

`spp`⟶		
	last private local variable	
	⋮	not initialized
	first private local variable	
`fpp`⟶	`$$` (rank)	
	old `eps`	
	old `gps`	
	old `gpp`	callee–saved
	old `fps`	
	old `fpp`	
	old `pc`	call–saved
	last saved register	
	⋮	caller–saved
	first saved register	
	last private argument	only if the callee
	⋮	has private arguments
	first private argument	not passed in registers;
`app`⟶	old `app`	caller–saved

8.1.3 Private procedure frame for asynchronous functions

`spp`⟶		
	last private local variable	
	⋮	not initialized
	first private local variable	
`fpp`⟶	(reserved for returning a struct)	
	old `fpp`	callee–saved
	old `pc`	call–saved
	last saved register	
	⋮	caller–saved
	first saved register	
	last private argument	only if the callee
	⋮	has private arguments
	first private argument	not passed in registers;
`app`⟶	old `app`	caller–saved

The compiler uses the most significant bit of the pointer stored in `fpp,-1` to allow for run–time inspection whether this is a synchronous (`eps`, i.e. an absolute address) or asynchronous (old `fpp`, i.e. a private address) function.

8.1.4 Shared procedure frame

`sps`⟶		
	last shared local variable	the shared locals
	⋮	defined at top level
	first shared local variable	of the function
	synchronization cell	this function's
`fps`, `gps`⟶	old gps	group frame
	last shared argument	only if the callee
	⋮	has shared arguments;
	first shared argument	
`aps`⟶	old `aps`	caller–saved

For each synchronous function an extra group frame (see next subsection) is allocated. This is technically required in order to re-synchronize at the end of function execution when some processors `return`ed earlier.

An asynchronous function does not have a shared procedure frame. Note that it can nevertheless allocate shared variables inside a `start` body, since these are local to the group frame built by `start`.

8.2 Group frames

Group–forming operations are `start` and `fork`, and the constructs `if`, `for`, `while`, `do` with (maybe) private branch condition.

8.2.1 Private group frame built by `if` and `fork`

The loops (`for`, `while`, `do`) build only shared group frames.

	(new) process ID `$`	copied from parent group
	(new) group ID `@`	copied from parent group
`gpp`⟶	old `gpp`	points to parent priv. group frame
	`if`: split cell; `fork`: group's shared memory size	debug info
	`if`: 0; `fork`: # new groups opened	debug info
	old sps	
	old eps	

8.2.2 Private group frame built by `start`

	(new) process ID `$`	numbered 0...`__STARTED_PROCS__-1`
	`@` = 0	the only group's ID
`gpp`⟶	old `gpp`	points to parent priv. group frame
	old sps	
	old eps	
	old fps	

8.2.3 Shared group frame

`sps`⟶		
	last group-local shared variable ⋮ first group-local shared variable	the shared variables defined locally to the construct building this group
	synchronization cell	initialized by # processors in this group
`fps`*, `gps`⟶	old gps	points to parent shared group frame

* The `fps` is set only by `start` and `join`.

Note that the cost of adressing a local shared variable depends on the site of definition as well as of the site of the access.

```
sync void foo( ... )
{  sh int i;  /*exists once for each group entering this function*/
   fork (...)  {
      sh int j;  /*exists once for each subgroup*/
      fork (...) {
         sh int k;  /*exists once for each subgroup*/
         k = i * j ... ;
      }
   }
}
```

In synchronous functions, the shared variables defined at top level of the function (e.g., variable `i` in the example code above) are addressed via the `fps` and thus one instruction is sufficient.

In the other cases, the chain of `gps` pointers has to be traversed backwards from the current group frame to the group frame of the group that defined that variable. The cost of addressing is $2x + 1$ where x denotes the number of group frames between the current and the defining group frame (including the defining frame). Thus, in the example above, calculation of the address of `j` costs 3 instructions while the calculation of the address of `k` can be done in one instruction.

9 Optimizations

`fcc` performs a limited set of optimizations automatically.

- Integer division and modulo computation are very expensive. Divisions and Modulo–computations are realized by shifts if the value of the divisor (or modulus) at run-time is a power of two.
- Synchronization after synchronous divisions is suppressed if the divisor is shared or constant.
- Alignment to the MODULO flag is suppressed for accesses to the private stack.
- Small integer (char, pointer, short, unsigned int) constants are loaded by a `getlo` instead of two instructions.
- The group synchronization immediately after a synchronous function call is suppressed if the callee does not contain `return` statements nested in subgroup–creating constructs (thanks go to Adrian Perez Jorge for pointing me to this simple solution).
- Profiling/tracing code is only executed if explicitly desired by the `-T` option. This has been made possible by using a profiling and a non–profiling version of the responsible library files.

The following further optimizations may be useful, but cannot be implemented due to the one-pass-nature of the `lcc`:

- creating a loop group also for a shared loop exit condition is not necessary if the loop body contains no `break` or `return` statement inside a one-sided `if` statement.
- static analysis for keeping track of the MODULO flag of the SB-PRAM. This optimization may speed up shared-memory-intensive user code by up to 33%. The assembler library routines are already hand–optimized.
- and some more ...

10 Limitations

The maximal number of processors used is a compile-time constant and must be at most 4096 for the SB-PRAM prototype at Saarbrücken.

Conditional expressions $e?l : r$ do — unlike the private `if` statement — not build group frames. As a private condition e may introduce a deadlock, a warning is emitted in this case. This is due to the strange construction of expression DAGs in the `lcc`. Nevertheless, this does not prohibit the programmer — he can use the `if` statement instead.

The same holds for `switch` statements: Currently, there are no group frames provided. Thus, a private selector may introduce a deadlock (warning is given). If a synchronous switch over a private selector cannot be avoided, the programmer should replace it by a (if possible, balanced) `if` cascade.

Attention must be paid if `continue` is used in a loop. If between the `continue` and the end of the loop body some synchronization will take place (e.g., at the end of a private `if`, of a private loop, of a `sync` function call or of a `farm`), a deadlock may be introduced. This problem will disappear in a future release of `fcc` by enabling the user to indicate such situations a priori by specifying a compiler option that introduces an extra shared group frame for each loop. A general implementation of `continue` is not sensible for a one–pass–compiler like `fcc`.

It is illegal to access within a `start` or `join` body a shared local variable declared outside the `start` (the effects are undetermined, the compiler does not warn). We generally discourage nesting of more than one `start` or `join` within the same function.

11 Known Bugs

- Make sure to have all needed header files included. There is no complaint from the compiler and linker if one is omitted, but parameter passing may be done the wrong way in such cases. An example is `memcpy()` declared in `string.h`: it passes its arguments in registers instead of pushing them on the private stack. A similar problem holds if a function has shared formal parameters: They are passed on the private stack if there is no prototype available. The compiler does not complain as long as the parameter types match (i.e., they are implicitly assumed to be integer—this is standard C semantics).

- Never use old–style function definitions such as

  ```
  void foo( x, y, z ) char x; int y, z; { .... }
  ```

  this will often cause the compiler to crash with a segmentation fault. Rather use the more convenient new–style declaration (which is the only possibility if shared parameters occur!):

  ```
  void foo( char x, int y, int z ) { .... }
  ```

  Note that also for equal types of subsequent parameters the type identifier (here: `int`) has to be replicated.

- Structures cannot be returned by functions. This may be possible in a future release of `fcc`; for now, you can work around by returning a pointer to the structure, which is, by the way, more efficient since calling the copy routine is not required.

- Something may be wrong with switches (maybe a `lcc 1.9` bug). Moreover, `switch` statements do not lead to group splitting in synchronous mode, i.e. may cause asynchrony. If you have problems with a switch, try to avoid it and use `if` cascades instead.

- `pprintf()` and `fpprintf()` are mainly intended for debug output and do not work for floatingpoint numbers. Use instead either usual `printf()` or the fast and simple `prF` routine (see `io.h`).

- Passing `structs` as arguments or returning a `struct` from a function doesn't work. This is why the function `ldiv()` is defined in a different way to ANSI C.

- Parameter sharity declarations for doubly nested function pointer declarations are ignored:

  ```
  sync int f(
    sh float x,                                 // sh x recognized
    sh sync void (*g)( sh int k ),              // sh k recognized
    sh sync void (*h)( sh sync int (*h2)( sh int k2 ) )
                                                // sh k2 not recognized
  ) {
   ... H2 = ...
   ... h( H2 ) ...   // ! no check for sharities of H2 parameters
  }
  ```

12 Reporting Bugs

Bugs can be reported by sending mail with the shortest program that exposes them and the details reported by `fcc`'s `-v` option to `kessler@psi.uni-trier.de`. Other questions, comments, and requests can be sent to the same address. If you wish that we notify you on future releases of `fcc`, please send a short email message.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On Physical Realizations of the Theoretical PRAM Model. Technical Report 21/1990, Sonderforschungsbereich 124 VLSI Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1990.

[2] P. Bach, M. Braun, A. Formella, J. Friedrich, Th. Grün, H. Leister, C. Lichtenau, and Th. Walle. Building the 4 Processor SB-PRAM Prototype. Technical Report 05/1996, Sonderforschungsbereich 124 VLSI–Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken (Germany), 1996.

[3] American National Standard Institute, Inc., New York. *American National Standards for Information Systems, Programming Language C ANSI X3.159–1989*, 1990.

[4] A. Formella and T. Grün and C.W. Keßler, The SB-PRAM: concept, design, construction, Proc. 3rd Int. IEEE Conference on Massively Parallel Programming Models, Nov. 1997, London, IEEE CS Press.

[5] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software—Practice & Experience*, 21(9):963–988, Sept. 1991.

[6] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

[7] C. W. Fraser and D. R. Hanson. A retargetable C compiler. Benjamin Cummings, 1995.

[8] T. Hagerup, A. Schmitt and H. Seidl. `FORK`: A High–Level Language for PRAMs. Future Generation Computer Systems 8 (1992), 379–393.

[9] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming.* Textbook, approx. 550 pages, to appear in Spring 2000 at Wiley, New York.

[10] J. Keller, W. J. Paul and D. Scheerer. Realization of PRAMs: Processor Design. Proc. WDAG'94, 8th Int. Workshop on Distributed Algorithms, Springer LNCS vol. 857, 17–27, 1994

[11] C. W. Kessler and H. Seidl. Making `FORK` Practical. Technical Report 95-01, SFB 124, Universität Saarbrücken.

[12] C. W. Kessler and H. Seidl. Fork95 language and compiler for the SB-PRAM. Proceedings of 5th Workshop on Compilers for Parallel Computers, Malaga, Spain, June 28–30, 1995. Technical Report UMA-DAC-95/09, University of Malaga, Department of Computer Architecture.

[13] C. W. Kessler and H. Seidl. Integrating synchronous and asynchronous paradigms: The Fork95 programming language. Technical Report 95-05, Universität Trier. See also: Proceedings of MPPM-95 Int. IEEE Conference on Massively Parallel Programming Models, Berlin, Germany, Oct. 9–12, 1995.

[14] C. W. Kessler and J. L. Träff. A Library of Basic PRAM Algorithms and its Implementation in FORK. 8th Annual ACM Symposium on Parallel Algorithms and Architectures. pp. 193–195, ACM press, 1996.

[15] C. W. Kessler and J. L. Träff. Language and Library Support for Practical PRAM Programming. 5th EUROMICRO Workshop on Parallel and Distributed Processing, London, IEEE CS Press, Jan. 1997.

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

[17] C. W. Kessler and H. Seidl. Language Support for Synchronous Parallel Critical Sections. Technical Report 95-23, Universität Trier, Nov. 1995. Proc. Int. Conf. on Advances in Parallel and Distributed Computing (APDC'97), Shanghai, China, 1997. IEEE CS Press.

[18] C. W. Kessler. Practical PRAM Programming with Fork95 — A Tutorial. Technical Report 97-12, Universität Trier, 1997. Contained in the `doc` directory of the `fcc` distribution.

[19] D. Scheerer. Der SB-PRAM Prozessor. PhD dissertation, Universität Saarbrücken, 1995.