Proc. HPCN-Europe'99, Amsterdam, April 12–14, 1999, Springer LNCS, pp. 525–534.

ForkLight: A Control–Synchronous Parallel Programming Language

Christoph W. Keßler and Helmut Seidl

FB IV - Informatik, Universität Trier, 54286 Trier, Germany
e-mail: {kessler,seidl}@psi.uni-trier.de

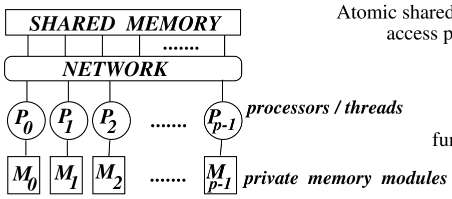
Abstract. ForkLight is an imperative, task-parallel programming language for massively parallel shared memory machines. It is based on ANSI C, follows the SPMD model of parallel program execution, provides a sequentially consistent shared memory, and supports dynamically nested parallelism. While no assumptions are made on uniformity of memory access time or instruction–level synchronicity of the underlying hardware, ForkLight offers a simple but powerful mechanism for coordination of parallel processes in the tradition and notation of PRAM algorithms: Beyond its asynchronous default execution mode, ForkLight offers a mode for control–synchronous execution that relates the program's block structure to parallel control flow.

We give a scheme for compiling ForkLight to C with calls to a very small set of basic shared memory access operations like atomic *fetch&add*. This yields portability across parallel architectures and exploits the local optimizations of their native C compilers. Our implementation is publically available; performance results are reported. We also discuss translation to OpenMP.

1. INTRODUCTION. Parallel processing offers an attractive way to increase computer performance. MIMD architectures, where program control is individual to each processor, offer high flexibility in programming, but devising, implementing and debugging parallel programs is difficult. Even more difficulties arise if the programmer has to care about explicit data distribution to achieve reasonable performance. Automatic parallelization and automatic data distribution techniques are still limited to rather regular programs. Addressing the latter problem, parallel computer manufacturers and research groups recently devised several types of massively parallel (or at least scalable) machines providing a shared memory. Some of these still require program tuning for locality to perform efficiently, e.g. Stanford DASH [18], while others use multithreading to hide the memory access latency and high–bandwidth memory networks, and thus become more or less independent of locality issues, e.g. Tera MTA [2].

So far there is only one massively parallel shared memory architecture that offers uniform memory access time (UMA), the SB-PRAM [1]. Due to a common clock all processors of the SB-PRAM work synchronously, i.e. they start (and complete) execution of an instruction simultaneously. This synchronicity makes parallel programming very easy, as it leads to deterministic parallel program execution; it is particularly well suited for the implementation of synchronous algorithms using e.g. fine–grained pipelines or data–parallel operations. Furthermore, such synchronous MIMD machines with UMA shared memory are very popular in theoretical computer science, where they are known as PRAMs (*P*arallel *R*andom *A*ccess *M*achines).

On the other hand, no massively parallel MIMD machine that is commercially available today is UMA or synchronous in this sense. Rather, the following features are common: The user sees a large amount of threads (due to scalable architecture and multithreading) and a monolithic shared memory (due to a hidden network). There is no common clock. The memory access time is non–uniform, but more or less independent of locality (due to multithreading). Program execution is asynchronous (due to the



Atomic shared memory access primitives:
- **SMwrite** (write a value into a shared memory cell)
- **SMread** (read a value from a shared memory cell)
- **atomic_add** (add an integer to a cell)
- **fetch_add** (add an integer to a cell and return its previous value)

furthermore:
- **shmalloc** (shared memory allocation)
- **beginparallelsection** (spawn p threads)
- **endparallelsection** (kill spawned threads)

Fig. 1. Asynchronous PRAM model with shared memory access operations. The seven operations on the right hand side are sufficient to handle shared memory parallelism as generated by the ForkLight compiler. The processors' native load/store operations are used for accessing private memory. Other thread handling functions like inspecting the thread ID or the number of threads are implemented by the compiler's run–time library.

previous two items, and because of system features like virtual memory, caching, and I/O). Also, there is efficient hardware support for atomic *fetch&op* instructions. — A typical representative of this class of parallel machines is the Tera MTA [2].

In order to abstract from particular features and to enhance portability of parallel algorithms and programs, one often uses a *programming model* that describes the most important architecture properties. Suitable parameterization allows for straightforward estimates of run times; such estimations are the more accurate, the more the particular parallel hardware fits the model used. In our case, the programming model is the Asynchronous PRAM introduced in the parallel theory community in 1989 [13, 9, 10]. An *Asynchronous PRAM* (see Fig. 1) is a MIMD parallel computer with a sequentially consistent shared memory. Each processor runs with its own clock. No assumptions are made on uniformity of shared memory access times. Thus, much more than for a true PRAM, the programmer must explicitly take care of avoiding race conditions (nondeterminism) when accessing shared memory locations or shared resources (screen, shared files) concurrently. We add to this model some atomic *fetch&op* instructions like `fetch_add` and atomic update instructions like `atomic_add`, which are required for basic synchronization mechanisms. This is not an inadmissible modification of the original Asynchronous PRAM model, as software realizations for these primitives do exist (but incur significant overhead [23]). In short, this programming model is closely related to the popular PRAM and BSP models but offers, in our view, a good compromise, as it is closer to real parallel machines than PRAMs and easier to program than BSP.

The PRAM programming model, as supported e.g. by Fork95 [15], *ll* [19], and *Modula-2** [21], offers deterministic write conflict resolution and operator–level synchronous execution: there are no race conditions at all, data dependencies need not be protected by locks or barriers. Unfortunately, this ideal parallel programming model leads to very inefficient code when compiled to asynchronous machines, in particular if the compiler fails to analyze data dependencies in irregular computations and resorts to worst-case assumptions. Hence, programming languages especially designed for "true" PRAMs, cannot directly be used for Asynchronous PRAMs.

In this paper we propose ForkLight, a task–parallel programming language for the Asynchronous PRAM model. It retains a considerable part of the programming comfort known from Fork95 but drops the requirement for exactly synchronous execution. Rather, synchronicity is relaxed to the basic block level (*control–synchronicity*).

The *fork-join model* of parallel execution, adopted e.g. by ParC [3], Tera-C [6], or Cilk [17], corresponds to a tree of processes. Program execution starts with a sequential process, and any process can spawn arbitrary numbers of child processes at any time. While the fork-join model directly supports nested parallelism, the necessary scheduling of processes requires substantial support by a runtime system and incurs overhead. — In contrast, in the *SPMD model* of parallel execution, all p available processors (or threads) are running just from the beginning of `main` execution; they are to be distributed explicitly over the available tasks, e.g. parallel loop iterations. Given fixed sized machines, SPMD seems better suited to exploit the processor resources economically.

ForkLight follows the SPMD model. Coordination is provided e.g. by composing the threads into groups. The threads of a group can allocate group–local shared variables and objects. In order to adapt to finer levels of nested parallelism, a group can be (recursively) subdivided into subgroups. In this way, ForkLight supports a parallel recursive divide–and–conquer style as suggested e.g. in [8], as well as data parallelism, task farming, and other parallel algorithmic paradigms. In contrast, most other languages adopting the SPMD model, like Denelcor HEP Fortran [14], EPEX/Fortran [12], PCP [4], Split-C [11], AC [7], support only one level of parallelism and one global name space; only PCP has a hierarchical group concept similar to that of ForkLight. Moreover, in ForkLight, control–synchronous execution can locally be relaxed towards totally asynchronous computation as desired by the programmer, e.g. for efficiency reasons.

For the compilation of ForkLight discussed in Section 3, we only assume a shared memory and efficient support of atomic increment and atomic *fetch&increment* operations (see Fig. 1). These are powerful enough to serve as the basic component of simple locking/unlocking and barrier mechanisms and to enhance e.g. the management of parallel queues or self–scheduling parallel loops [22] and occur in several routines of the ForkLight standard library. A source–to–source compiler for ForkLight has been implemented based on the methods given in Section 3. It generates C source code plus calls to the routines listed in Fig. 1 that are currently implemented by calls to the shared-memory P4 routines [5], which provides platform independence. We also give an implementation scheme and optimization for OpenMP [20]. We report results on Solaris multiprocessor workstations with P4 support and on the SB-PRAM. Our implementation is available at `http://www.informatik.uni-trier.de/~kessler/forklight`

2. LANGUAGE DESCRIPTION. ForkLight extends C by constructs for group handling, declaration and allocation of shared variables, and relaxing control–synchronicity.

Shared and private variables. Variables are classified either as *private* (this is the default) or as *shared*; the latter are to be declared with the storage class qualifier `sh`. Here "shared" always relates to the thread group (see later) that executes that variable's declaration. Private variables exist once for each thread, whereas shared variables exist only once in the shared memory subspace of the thread group that declared them.

The *total number of started threads* is accessible through the constant shared variable `__P__`, the *physical thread ID* through the function `_PhysThreadId()`.

Execution modes and regions. There are two different program execution modes that are statically associated with source code regions: control–synchronous mode in control–synchronous regions, and asynchronous mode in asynchronous regions.

In *control–synchronous mode* ForkLight maintains the invariant that *all threads belonging to the same (active) group work on the same basic block.* Subgroup creation and implicit barriers occur only in control–synchronous mode.

In *asynchronous mode*, control–synchronicity is not enforced. The group structure is frozen; shared variables and automatic shared heap objects cannot be allocated.

There are no implicit synchronization points. Synchronization of the current group can, though, be explicitly enforced by a barrier() call or by a *barrier statement* denoted by a sequence of at least three ='s to optically emphasize it in the program code.

Functions are classified as control–synchronous (to be declared with type qualifier csync) or asynchronous (this is the default). main() is asynchronous by default. A control–synchronous function is a control–synchronous region, except for (blocks of) statements explicitly marked as asynchronous by async or as sequential by seq. An asynchronous function is an asynchronous region, except for statements explicitly marked as control–synchronous by start or join. async <stmt> causes the processors to execute <stmt> in asynchronous mode. In other words, the entire <stmt> (which may contain loops, conditions, or calls to asynchronous functions) is considered to be part of the "basic" (w.r.t. control–synchronicity) block containing this async. There is no implicit barrier at the end of <stmt>. If the programmer desires one, (s)he may use an explicit barrier. seq <stmt> causes <stmt> to be executed by exactly one thread of the current group; the others skip <stmt>. There is no implicit barrier at the end of <stmt>.

Asynchronous functions and async blocks are executed in asynchronous mode, except for start <stmt> statements that are only permitted in asynchronous mode. start switches to control–synchronous mode for its body <stmt>. It causes all __P__ threads to barrier–synchronize, form a group, and execute <stmt> simultaneously and in control–synchronous mode, with unique thread IDs $ numbered from 0 to __P__ -1. A generalization of start, the join statement, allows to more flexibly collect a variable amount of threads over a specified time or event interval and make them execute a control–synchronous statement [16].

In order to maintain the static classification of code, within asynchronous regions only async functions can be called. In the other way, calling an async function from a control–synchronous region is always possible and results in an implicit entering of the asynchronous mode. Shared local variables can only be declared / allocated within control–synchronous regions. In particular, asynchronous functions must not allocate shared local variables. All formal parameters must be private. There is an implicit group–wide barrier synchronization point at entry to control–synchronous functions.

Nested parallelism and groups. ForkLight programs are executed by *groups* of threads, rather than by individual threads. When program execution starts, there is just one group, the *root group*, that contains all available threads. Groups may be recursively subdivided. Thus, at any point of program execution, all presently existing groups form a tree–like *group hierarchy*. Only the *leaf groups* of the group hierarchy are active. Subgroups of a group can be distinguished by their *group ID*. A thread can access its current group's ID through the constant @. The subgroup ID @ can be preset by the programmer at the fork statement. join and start set @ to 0. The group–relative thread ID $, a private constant variable, is automatically computed whenever a new subgroup is created, by renumbering the subgroup's p threads from 0 to $p-1$. $ and @ are automatically saved when splitting the current group, and restored when reactivating it. All threads of a group have access to a common shared address subspace. allocating them. A thread can inspect the number of threads in its current group by the function groupsize().

At entry to a control–synchronous region (i.e., a join or start body), the threads form one single thread group. However, without special handling control flow could diverge for different threads at conditional branches such as if statements, switch statements, and loops. Only in special cases it can be statically determined that all threads are going to take the same branch of control flow. Otherwise, control–synchronicity could be lost. To avoid this, ForkLight guarantees control–synchronicity by suitable automatic group splitting. Nevertheless, the programmer may know in some situations that such a splitting is unnecessary. For these cases, (s)he can specify this explicitly:

We consider an expression to be *stable* if it is guaranteed to evaluate to the same value on each thread of the group for all possible program executions, and *unstable* otherwise. An expression containing private variables (e.g., `$`) is generally assumed to be unstable. But even an expression e containing only shared variables may be also unstable: Since e is evaluated asynchronously by the threads, it may happen that a shared variable occurring in e is modified (maybe as a side effect in e, or by a thread outside the current group) such that some threads of the group (the "faster" ones) use the old value of that variable while others use the newer one, which may yield different values of e for different threads of the same group. Technically, the compiler defines a conservative, statically computable subset of the *stable expressions* as follows:

(1) A (shared) constant is a stable expression. (This also includes `@` and shared constant pointers, e.g. arrays.)
(2) The pseudocast `stable(`e`)` is stable for any expression e (see below).
(3) If expressions e_1, e_2 and e_3 are stable, then also the expressions $e_1 \oplus e_2$ for $\oplus \in \{+, -, *, /, \%, \&, |, \hat{}, \&\&, ||\}$, $\ominus e_1$ for $\ominus \in \{-, \sim, !\}$, $e_1[e_2]$, $e_1.\mathit{field}$, $e_1 \rightarrow \mathit{field}$, $*e_1$, and $e_1 ? e_2 : e_3$ are stable.
(4) All other expressions are conservatively regarded as unstable.

Conditional branches with a stable condition expression do not affect control–synchronicity. Branches in control–synchronous mode with unstable conditions lead to automatic splitting of the current group into subgroups — one for each possible branch target. Control synchronicity is then only maintained within each subgroup. Where control flow reunifies again, the subgroups cease to exist, and the previous group is restored. There is no implicit barrier at this program point. (A rule of thumb: Implicit barriers are, in control–synchronous mode, generated only at branches of control flow, not at reunifications.) For an unstable two–sided `if` statement, for instance, two subgroups are created. The processors that evaluate their condition to true join the first, the others join the second subgroup. The branches associated with these subgroups are executed concurrently. For a loop with an unstable exit condition, one subgroup is created that contains the iterating processors.

Nevertheless, sometimes unstable branch conditions in control–synchronous mode may be tolerable without automatic group splitting, for performance tuning purposes. The pseudocast `stable (<expr>)` causes the compiler to treat expression `<expr>` as stable; the compiler then assumes that the programmer knows that possible unstability of `<expr>` will not be a problem in this context, for instance because (s)he knows that all processors of the group will take the same branch.

Splitting a group into subgroups can also be done explicitly. Executing

`fork (` e_1`; @=`e_2 `) <stmt>`

means the following: First, each thread of the group evaluates the stable expression e_1 to the number of subgroups to be created, say g. Then the current leaf group is deactivated and g subgroups $g_0, ..., g_{g-1}$ are created. The group ID of g_i is set to i. Evaluating expression e_2 (which is typically unstable), each thread determines the index i of the newly created leaf group g_i it will become member of. If the value of e_2 is outside the range $0, ..., g-1$ of subgroup IDs, the thread does not join a subgroup and skips `<stmt>`. The IDs `$` of the threads are renumbered consecutively within each subgroup from 0 to the subgroup size minus one. Each subgroup gets its own shared memory subspace, thus shared variables and heap objects can be allocated locally to the subgroup. Now, each subgroup g_i executes <stmt>. When a subgroup finishes execution, it ceases to exist, and its parent group is reactivated as the current leaf group. Unless the programmer writes an explicit barrier statement, the processors may immediately continue with the following code. Note that empty subgroups (with no threads) are possible; an empty subgroup's work is immediately finished, though.

Pointers and heaps. Since the private address subspaces are not embedded into the global shared memory but addressed independently, shared pointer variables must not point to private objects. As it is, in general, not possible to statically verify whether the pointee is private or shared, dereferencing a shared pointer containing a private address will lead to a run time error. Nevertheless any pointer may point to a shared object.

ForkLight supplies three kinds of heaps. First, there is the usual, private heap for each thread with the (asynchronous) functions `malloc()` and `free()` known from C. Second, there is a global, permanent shared heap with the asynchronous functions `shmalloc()` and `shfree()`. Finally, there is one automatic shared heap for each group that is intended to provide fast temporary storage blocks local to a group. Consequently, the life range of objects allocated on the automatic shared heap by the control–synchronous `shalloc()` function is limited by the life range of the allocating group.

Pointers to functions are also supported. In control–synchronous mode, dereferencing a pointer to a control–synchronous function is only legal if it is stable.

Standard atomic operations. Atomic *fetch&op* operations, also known as *multiprefix* computations when applied in a synchronous context with priority resolution of concurrent write accesses to the same memory location, are available as standard functions called `fetch_add`, `fetch_max`, `fetch_and` and `fetch_or`, to give the programmer direct access to these powerful operators. They can be used in control–synchronous as well as in asynchronous mode. Note that the order of execution for concurrent execution of several, say, `fetch_add` operations to the same shared memory location is not determined in ForkLight. For instance, computing the size p of the current group and new thread ID variables `myrank` consecutively numbered $0,1,...,p-1$ may be done by this code, where the function–local integer variable `p` is shared by all threads of the current group.

```
csync void foo( void ) {
  int myrank;
  sh int p = 0;
  ===== //guarantees p is init.to 0
  myrank = fetch_add( &p, 1 );
  ===== //guarantees p is groupsize
  ...     }
```

The atomicity of these operators is very useful to access semaphores in asynchronous mode, e.g., simple locks that sequentialize access to some shared resource where necessary. In its standard library, ForkLight offers simple locks, fair locks, and reader–writer locks, and further atomic memory operations `void atomic_`*op*`()`.

3. COMPILATION. The ForkLight implementation uses its own shared memory management using a sufficiently large slice of shared memory. To the bottom of this shared memory slice we map the shared global initialized resp. non–initialized variables. In the remainder of this shared memory part we arrange a shared stack and an automatic shared heap (see figure). Group splitting operations also cause splitting of the remaining shared stack space, creating an own shared stack and automatic heap for each subgroup; (i.e., a "cactus" stack and heap). The shared stack pointer `sps` and the

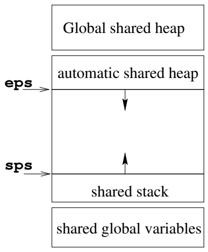


Fig. 2. Barrier synchronization pseudocode and shared group frame

$R_{csc} \leftarrow$`csc`; $R_{next} \leftarrow R_{csc} + 1$;
if $(R_{next} > 2)$ $R_{next} \leftarrow 0$ // wrap–around
`atomic_add(` `gps`$+R_{next}$`, 1);`
`atomic_add(` `gps`$+R_{csc}$`, -1);`
while `(SMread(`$sc[R_{csc}]$`)` $\neq 0$`)` `;` // wait
`csc`$\leftarrow R_{next}$;

	↑
	group–local shared var's
	$sc[2]$
	$sc[1]$
`gps`→	$sc[0]$

automatic shared heap pointer `eps` may be permanently kept in registers on each thread. A shared stack or automatic heap overflow occurs if `sps` crosses `eps`. Another shared memory slice is allocated to install the global shared heap. Initially, the thread on which the user has started the program executes the startup code, initializes the shared memory, and spawns the other threads as requested by the user. All these threads start execution of the program in asynchronous mode by calling `main()`. A private stack and heap are maintained in each thread's private memory by the native C compiler.

Group frames and group–wide barrier synchronization. For each group the compiler keeps a shared and private group frames. The *shared group frame* (see Fig. 2) is allocated on the group's shared stack. It contains the shared variables local to this group and *three synchronization cells* $sc[0]$, $sc[1]$, $sc[2]$. Each thread holds a register `gps` pointing to its current group's shared group frame, and a private counter `csc` indexing the *current synchronization cell*. When a new group is created, `csc` is initialized to 0, $sc[0]$ is initialized to the total number of threads in the new group, and $sc[1]$ and $sc[2]$ are initialized to 0. If no thread of the group is currently at a barrier synchronization point, the current synchronization cell $sc[$`csc`$]$ contains just the number of threads in this group.

At a group–wide barrier synchronization, each thread atomically increments the next synchronization cell by 1, then atomically decrements the current synchronization cell by 1, and waits until it sees a zero in the current synchronization cell, see Fig. 2. The algorithm guarantees that all threads have reached the barrier when a zero appears in the current synchronization cell. Only then they are allowed to proceed. At this point of time, though, the next current synchronization cell, $sc[R_{next}]$, already contains the total number of threads, i.e. is properly initialized for the following barrier synchronization. Once $sc[R_{csc}]$ is 0, all threads of the group are guaranteed to see this, as this value remains unchanged at least until after the following synchronization point. The run time is, for most shared memory systems, dominated by the group–wide `atomic_add` and `SMread` accesses to shared memory, while all other operations are local.

Each shared group frame is complemented by a *private group frame* containing the private group information, pointed to by register `gpp`. It contains the current values of the group ID `@`, the group–relative thread ID `$`, and the current synchronization cell index `csc`. `@` needs not be stored on the shared group frame since it is read–only. Also, the parent group's shared stack pointer `sps`, group–local heap pointer `eps`, and group frame pointer `gps` are stored in the private group frame, together with the parent group's `gpp`, thus the parent group can easily be restored when being reactivated.

Translation of a function call. Asynchronous functions are just compiled as known from sequential programming, as no care has to be taken for synchronicity. A control–synchronous function with shared local variables needs to allocate a shared group frame. As these variables should be accessed only after all threads have entered the function, there is an implicit group–wide barrier at entry to a control–synchronous function.

Translation of the `fork` statement. A straightforward implementation assumes that all k subgroups will exist and distributes shared stack space equally among these. For `fork ( k; @=e ) <stmt>` the following code is generated:

(1) $R_k \leftarrow eval(k)$; $R_@ \leftarrow eval(e)$; $slice \leftarrow \lfloor(\texttt{eps-sps})/R_k\rfloor$;
(2) **if** $(0 \leq R_@ < R_k)$ { `sc` $\leftarrow$ `sps`$+R_@ * slice$; `SMwrite( sc, 0 );` }
(3) `barrier` local to the (parent) group // necessary to guarantee a zero in `sc[0]`
(4) **if** $(0 \leq R_@ < R_k)$ { $R_\$ \leftarrow$ `fetch_add(sc,1);` allocate a private group frame `pfr` and store there `gps`, `eps`, `sps`, `gpp`; init. new `csc` field to 0, `@` field to $R_@$, `$` field to $R_\$$
if $(R_\$ = 0)$ { `sc[1]`$\leftarrow$ 0; `sc[2]`$\leftarrow$ 0; } }
(5) `barrier` local to the (parent) group // guarantees final subgroup sizes in `sc[0]`
(6) **if** $(0 \leq R_@ < R_k)$ // enter subgroup $R_@$ otherwise skip
{ `gps`$\leftarrow$ `sc`; `gpp`$\leftarrow$ `pfr`; `sps`$\leftarrow$ `gps`+3+#sh.locals; `eps`$\leftarrow$ `gps`+*slice*; } **else** goto (9)
(7) code for `<stmt>`
(8) `atomic_add( gps+csc, -1);` // cancel membership in the subgroup
leave the subgroup by restoring `gps`, `sps`, `eps`, `gpp` from the private group frame
(9) (next statement)

The overhead of the above implementation mainly consists of the parallel time for two group–wide barriers, one subgroup–wide concurrent `SMwrite`, and one subgroup–wide `fetch_add` operation. Also, there are two exclusive `SMwrite` accesses to shared memory locations. The few private operations can be ignored, since their cost is usually much lower than shared memory accesses. There are some *optimizations:* (1) group splitting and barriers can be skipped if the current group consists of only one thread. (2) Some of the subgroups may be empty. In that case, space fragmentation can be reduced by splitting the parent group's stack space in only that many parts as there are different values of $R_@$. (3) Not all group splitting operations require the full generality of the `fork` construct. Splitting into equally (or weighted) sized subgroups, as in PCP [4], can be implemented with only one barrier and without the `fetch_add` call, as the new subgroup sizes and ranks can be computed directly from locally available information.

Accessing local shared variables. Subgroup creation within the same function can be statically nested. The compiler determines the group nesting depth of the declaration of each variable `x` (see example on the right), call it $gd(\texttt{x})$, as well as that of each use of `x`, call it $gu(\texttt{x})$. For each use of `x`, the compiler computes the difference $d = gu(\texttt{x}) - gd(\texttt{x})$ and, where $d > 0$, inserts code to follow the chain of `gps` pointers d times upwards the group tree, in order to arrive at the group frame containing `x`. Because d is typically quite small, this loop is completely unrolled. Note that all these read accesses but the last one are private memory accesses.

```
csync void foo()
{fork(...) {
  sh int x=@; //decl
  ...
  fork(...)
    ... = x;  //use
}}
```

Translation to OpenMP. OpenMP [20] is a shared–memory parallel application programming interface for Fortran and C/C++, consisting of a set of compiler directives and several run time library functions. Although OpenMP is, in principle, task-parallel, it is more tailored towards dataparallel programming. Memory consistency must be enforced by the programmer by explicit `flush()` operations. OpenMP follows the fork-join execution model. Nevertheless, nested parallelism is only an optional feature of the implementations. Varying defaults for declarations of shared and private variables make OpenMP programs quite hard to read. Currently some OpenMP implementations are available for Fortran but not yet for C/C++.

A transcription of the existing ForkLight back-end to OpenMP is straightforward: `begin–` and `endparallelsection()` correspond to a `omp parallel` directive at the top level of the program. The shared stack and heap are simulated by two large arrays declared `shared volatile` at this directive, `SMread` and `SMwrite` become accesses of these arrays. Explicit `flush`ing after `SMwrite`s is not necessary for `volatile` shared variables. `omp barrier` and other synchronization primitives of OpenMP cannot be used for ForkLight because they are not applicable to nested SPMD computations; thus we will use our own implementation for the synchronization routines. The `atomic` directive of OpenMP is applicable to increment and decrement operators, but `fetch_add` has to be expressed using the sequentializing `critical` directive. In order to increase scalability, we propose *hashing of critical section addresses*: The `omp critical` directive optionally takes a compile–time constant name (string) as a parameter. `critical` sections with different names may be executed concurrently, while entry to all `critical` sections with the same name is guarded by the same mutual exclusion lock. For our implementation of atomic memory operations like `fetch_add` we use a static finite set of locks and distribute the shared memory accesses among these by a hash function.

Performance Results. We have implemented the compiler for the two parallel platforms that are currently accessible to us and that are still supported by P4: multiprocessor Solaris workstations and the SB-PRAM. These two completely different types of architecture represent two quite extremal points in the spectrum of shared–memory architectures regarding execution of P4 / ForkLight programs.

On a loaded four–processor Solaris 2.5 workstation, where atomic memory access is sequentialized, we observed good speedup for well–parallelizable problems like pi–calculation or matrix product but only modest or no speedup for problems that require frequent synchronization.

problem	size	time [s] with # threads			
		1	2	3	4
Pi-calculation	5000000	22.61	14.55	11.37	7.86
matrix product	200 x 200	41.08	24.16	19.22	14.49
par. mergesort	48000	4.39	3.58	n.a.	2.33
par. quicksort	120000	14.50	9.83	n.a.	6.35

On the SB-PRAM we exploited its native `fetch_add` and `atomic_add` operators which do, differently from standard P4, not lead to sequentialization. For the SB-PRAM prototype at Saarbrücken we obtained these results.

problem	size	time [ms] with #threads						
		1	2	4	8	16	32	64
par. mergesort	1000	573	373	232	142	88	57	39
par. mergesort	10000	2892	4693	3865	1571	896	509	290
par. quicksort	1000	1519	807	454	259	172	145	130

We observe: (1) Efficient support for non-sequentializing atomic `fetch_add` and `atomic_add`, as in SB-PRAM or Tera MTA, is essential when running ForkLight programs with large numbers of threads. Executables relying only on pure P4 suffer from serialization and locking/unlocking overhead and are thus not scalable to large numbers of threads. — (2) On a non–dedicated, loaded multiuser / multitasking machine like our Solaris multiprocessor workstation, parallel speedup suffers from poor load balancing due to stochastic delaying effects: the processors are unsymmetrically delayed by other users' processes, and at barriers these delays accumulate. — (3) Even when running several P4 processes on a single processor, performance could be much better for $p > 1$ if the ForkLight run time system had complete control over context switching for its own threads. Otherwise, much time is lost spinning on barriers to fill the time slice assigned by an OS scheduler that is unaware of the parallel application. This is an obvious weakness of P4. — (4) Explicit load balancing in an SPMD application may be problematic in particular for small machine sizes (quicksort), or when the hardware scheduler does not follow the intentions of the user, e.g. when the scheduler maps several P4 threads to a processor where only one thread was intended for. — (5) Where these requirements are met, our prototype implementation achieves acceptable speedups and performance scales quite well even for rather small problem sizes.

4. CONCLUSION. From a software engineering point of view, control–synchronous execution is an important guidance to the programmer since it transparently relates the program block structure to the parallel control flow. By the support of nested parallelism in ForkLight, programming is as comfortable as in fork-join languages.

References

1. F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, Dec. 1993.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. The Tera Computer System. *Proc. 4th ACM Int. Conf. on Supercomputing*, pp. 1–6, 1990.
3. Y. Ben-Asher, D. Feitelson, L. Rudolph. ParC — An Extension of C for Shared Memory Parallel Processing. *Software – Practice and Experience*, 26(5):581–612, May 1996.
4. E. D. Brooks III, B. C. Gorda, K. H. Warren. The Parallel C Preprocessor. *Scientific Programming*, 1(1):79–89, 1992.
5. R. Butler, E. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20(4):547–564, April 1994.
6. D. Callahan, B. Smith. A Future–based Parallel Language for a General–Purpose Highly–parallel Computer. Tera Computer Company, http://www.tera.com, 1990.
7. W. W. Carlson, J. M. Draper. Distributed Data Access in AC. In *Proc. ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 39–47. ACM Press, 1995.
8. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.
9. R. Cole, O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM model. *Proc. 1st Ann. ACM Symp. on Par. Algorithms and Architectures*, pp. 169–178, 1989.
10. R. Cole, O. Zajicek. The Expected Advantage of Asynchrony. *JCSS* 51:286–300, 1995.
11. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick. Parallel Programming in Split-C. *Proc. Supercomputing'93*, Nov. 1993.
12. F. Darema, D. George, V. Norton, G. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.
13. P. B. Gibbons. A More Practical PRAM model. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.
14. H. F. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3:93–110, 1986.
15. C. W. Keßler, H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of Parallel Programming*, 25(1):17–50, Feb. 1997.
16. C. W. Keßler, H. Seidl. ForkLight: A Control–Synchronous Parallel Programming Language. Tech. Report 98-13, Univ. Trier, FB IV–Informatik, 54286 Trier, Germany, 1998.
17. C. E. Leiserson. Programming Irregular Parallel Appplications in Cilk. *Proc. IRREGULAR'97*, pp. 61–71. Springer LNCS 1253, 1997.
18. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennesy, M. Horowitz, M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.
19. C. León, F. Sande, C. Rodríguez, F. García. A PRAM Oriented Language. *EUROMICRO Wksh. on Par. and Distr. Processing*, pp. 182–191. IEEE CS Press, 1995.
20. OpenMP ARB. OpenMP White Paper, `http://www.openmp.org/`, 1997.
21. M. Philippsen, W. F. Tichy. Compiling for Massively Parallel Machines. In *Code Generation – Concepts, Tools, Techniques*, pp. 92–111. Springer Workshops in Computing, 1991.
22. J. M. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch–and–Add*. PhD thesis, New York University, 1988.
23. X. Zhang, Y. Yan, R. Castaneda. Evaluating and Designing Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors. *IEEE Par. & Distr. Techn.*, 4:25–42, 1996.