UNIVERSITÄT TRIER
Mathematik / Informatik

Forschungsbericht Nr. 98–13

ForkLight: A Control–Synchronous Parallel Programming Language

Christoph W. Keßler Helmut Seidl

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

Electronic copies of technical reports are available:

Via FTP: Host ftp.informatik.uni-trier.de, directory /pub/reports

Via gopher: Host gopher.informatik.uni-trier.de, port 70, type 1, pathes 1/TechReports/Abstracts and 1/TechReports/FullText

Via WWW: URL http://www.informatik.uni-trier.de/Reports/List

Via email: Send a mail to ftpmail@ftp.informatik.uni-trier.de, subject 'MAIL ME CLEAR', body 'TechReports.HowTo' followed by an empty line, for detailed instructions

Printed copies:

Trierer Forschungsberichte
Fachbereich IV -
Mathematik / Informatik
Universität Trier
D-54286 Trier

ISSN 0944-0488

ForkLight: A Control–Synchronous Parallel Programming Language

Christoph W. Keßler Helmut Seidl
Fachbereich IV - Informatik, Universität Trier
D-54286 Trier, Germany
e-mail: {kessler,seidl}@psi.uni-trier.de

Abstract: We present ForkLight, an imperative parallel programming language for massively parallel asynchronous shared memory machines. It is based on ANSI C and offers additional constructs to hierarchically divide thread groups into subgroups and manage shared and private address subspaces.

While no assumptions are made on uniformity of memory access time or instruction–level synchronicity of the underlying hardware, ForkLight aims to provide the programmer with a powerful mechanism for coordination of parallel processes in the tradition and notation of PRAM algorithms. Beyond the usual asynchronous mode, ForkLight offers a mode for control–synchronous execution of a program which relates the program's block structure to parallel control flow, and which directly maps to parallel divide&conquer implementations. Also, other important parallel algorithmic paradigms can be elegantly expressed, e.g. pipelining.

We give a scheme for compiling ForkLight to C with calls to a small set of basic shared memory access operations such as atomic *fetch&add*. With this interface we can achieve portability across parallel architectures and exploit the local optimizations of their native C compilers.

Keywords: parallel programming language design, shared memory parallel computer, control–synchronous program execution, SPMD model, nested parallelism, source–to–source compiler, ForkLight

1 Introduction

Parallel processing offers an attractive way to increase computer performance. MIMD[1] architectures seem to offer a high degree of flexibility in programming. Nevertheless, devising parallel algorithms, implementing them on a parallel machine, and reasoning about existing parallel programs (e.g., debugging) is difficult and error–prone. Even more difficulties arise if the programmer has to care about explicit data distribution in order to achieve reasonable performance. Unfortunately, automatic parallelization and automatic data distribution is still in a stage that can handle only 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 simulating or emulating a shared memory. Most of these machines have non–uniform memory access time (NUMA). Some of these still require program tuning for locality in order to perform efficiently, e.g. Stanford DASH [28], while others use multithreading to hide the memory access latency 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 which 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, 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 which is commercially available today is UMA or synchronous in this sense. Rather, the common features are the following:

- the user sees a large amount of threads (due to scalable architecture and multithreading)
- the user sees a monolithic shared memory (due to a hidden network)
- there is no common clock
- the memory access time is non–uniform, but more

[1] In a MIMD (multiple instruction streams and multiple data streams) architecture each thread has its own program counter, thus the program control is individual.

or less independent of locality (due to multithreading)

- program execution is asynchronous (due to the previous two items, and because of system features like virtual memory, caching, and I/O)
- 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 hardware 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. Typically, also a standard programming interface is defined for such models, e.g. MPI for the message passing model, HPF for data–parallel execution on distributed memory machines, BSPlib for the BSP model, or Fork95 for the PRAM model.

In our case, the programming model is the Asynchronous PRAM introduced in the parallel theory community in 1989 [18, 12, 13]. An *Asynchronous PRAM* (see Fig. 1). is a MIMD parallel computer with a sequentially consistent shared memory. Each processor runs with its own private 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_incr`, which are required for basic synchronization mechanisms and typically supported by the parallel machines in consideration. Note however that this is not an inadmissible modification of the original Asynchronous PRAM model, since software realizations for these primitives are also possible (at the expense of significant overhead [35]). 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.

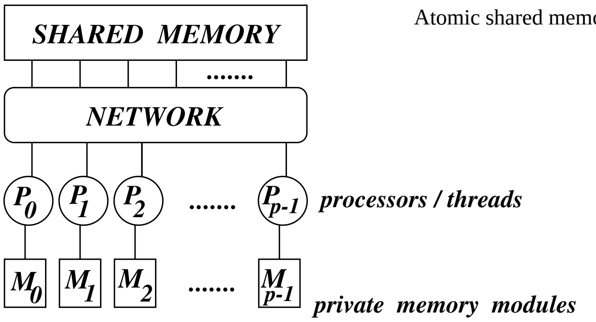
Programming languages especially designed for "true" PRAMs such as Fork95 [24] cannot directly be used for Asynchronous PRAMs, as their efficient implementation relies on unit memory access time and instruction–level synchronous program execution.

In Section 2 we propose ForkLight, a task–parallel programming language for the Asynchronous PRAM model which retains a considerable part of the programming comfort known from Fork95 while dropping the requirement for exactly synchronous execution on the expression operator level, which would inevitably lead to poor performance on any existing asynchronous machine. Rather, synchronicity is relaxed to the basic block level.

In contrast to other task–parallel programming languages such as ParC [4], Tera-C [9], or Cilk [27], ForkLight follows the SPMD model of parallel execution, i.e. all p available processors (or threads) are running just from the beginning of `main` execution. Thus, ForkLight does not need a **spawn** resp. **forall** statement to spawn new parallel threads from a current one. Coordination is provided e.g. by composing the threads into groups. The threads of a group can allocate group–local shared variables and objects. They work *control–synchronously*, i.e. all branches of control flow are synchronized. 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 in [5], [11], [17], [20], [21], as well as data parallelism, task farming, pipelining, and many other parallel algorithmic paradigms. Control–synchronous execution can locally be relaxed towards totally asynchronous computation where this is desired by the programmer, e.g. for efficiency reasons. Also, a clever compiler will aim at removing as many synchronization points as possible without courting the risk of race conditions.

For the compilation of ForkLight discussed in Section 3, we only assume a shared memory and efficient support of atomic increment operations and atomic *fetch&increment* instructions (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 [34] 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 P4 library (shared memory part; [8]); which provides platform independence. The generated code currently runs successfully on SOLARIS workstations with P4 support and on the SB-PRAM. As soon as the future OpenMP standard [30] becomes available for C, we will also provide an OpenMP–based backend.



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-1 threads)
`endparallelsection` (kill spawned threads)

Figure 1: **Asynchronous PRAM model with shared memory access operations.** Assuming that the processors' native load/store operations are used for accessing private memory, the seven operations listed on the right hand side are sufficient to handle shared memory parallelism as generated by the ForkLight compiler. Other auxiliary functions for thread handling, like inspecting the thread ID or the total number of threads, could also be implemented by the run–time library of the compiler.

2 ForkLight language description

The *fork-join model* of parallel execution 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 high overhead.

In contrast, ForkLight follows the *SPMD model* of parallel execution, i.e. there is a constant set of threads that are to be distributed over the available tasks, e.g. parallel loop iterations. Given fixed sized machines, SPMD seems better suited to exploit the processor resources economically. Nevertheless, ForkLight supports nested parallelism as well, as we will see in this section.

ForkLight is based on ANSI C [3]. The new constructs handle the group organization, shared and private address subspaces, and various levels of relaxed control–synchronicity.

2.1 Shared and private variables

The entire shared memory of the Asynchronous PRAM is partitioned — according to the programmer's wishes — into private address subspaces (one for each thread) and a shared address subspace which may be again dynamically subdivided among the different thread groups. Accordingly, variables are classified either as *private* (this is the default) or as *shared* (to be declared with storage class qualifier `sh`), where "shared" always relates to the thread group[2] that executes that variable's declaration. Private objects exist once in each thread's private address subspace, whereas shared objects 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* of each thread is accessible through the function

```
_PhysThreadId()
```

The special private constant variable `$` is intended to hold the current *group–relative thread ID*. It is initially set to the physical thread ID, automatically redefined when the thread enters a subgroup, and automatically restored when the thread leaves it.

2.2 Control–synchronous and asynchronous program regions

ForkLight offers 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*

[2] Groups will be defined in more detail in Section 2.3.

(active) group work on the same basic block[3]. Subgroup creation and implicit barriers occur only in control–synchronous mode. The group concept will be explained in detail in section 2.3.

In *asynchronous mode*, control–synchronicity is not enforced. The group structure is read–only; 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 call to the `barrier()` function, or, to optically emphasize it in the program code, by a barrier statement like

```
===========
```

i.e., a sequence of at least three ='s.

Initially, one thread on which the program has been started by the user executes the startup code and initializes the shared memory. Then it spawns the other threads requested by the user program. All these threads start execution of the program in asynchronous mode by calling function `main()`.

Functions are classified as either 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 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()`.

The `async` statement

```
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 (see above).

The `seq` statement

```
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>`.

[3] A basic block is a sequence of instructions which contains no joinings (except the first instruction) and no branches (except the last instruction) of control flow. Thus, a basic block is entered by control flow only at the first and left only with the last instruction.

Asynchronous functions are executed in asynchronous mode, except for blocks starting with the `start` statement

```
start <stmt>
```

The `start` statement, only permitted in asynchronous mode, switches to control–synchronous mode for its body `<stmt>`. It causes all available threads to barrier–synchronize, to form one large group, and execute `<stmt>` simultaneously and in control–synchronous mode, with unique thread IDs `$` numbered consecutively from 0 to `__P__`-1.

A generalization of `start`, the `join` statement [25], 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:

```
join(<delaystmt>;<springoffcond>;<SMsize>)
   <stmt>
   else <other_work>
```

The first thread arriving at that program point initializes a new (root) group and then waits according to the delay statement specified in `<delaystmt>`. During that waiting time, other threads arriving at this `join` statement may also join the new group. Thereafter the threads of the new group will execute the control–synchronous region `<stmt>` in control–synchronous mode. An upper limit of the new group's size can be installed by specifying a private condition `<springoffcond>` that, where evaluating to a non–zero value, enforces the thread to skip the execution of `<stmt>` immediately before the new group starts. These threads, as well as the "late" threads arriving at the `join` statement after the group started execution of `<stmt>`, execute asynchronously the optional `else` branch. If they run across a `retry` statement inside the `<other_work>`, they jump back to the entry point of the `join` statement (very similar to `continue` in C loops). When the new group finishes its execution of `<stmt>`, it is destroyed, its threads proceed with the next statement. At the same time, this `join` is reset to its original state, i.e. the first thread now arriving there initializes ... etc.

To maintain the static classification of code into control–synchronous and asynchronous regions, within an asynchronous region, 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

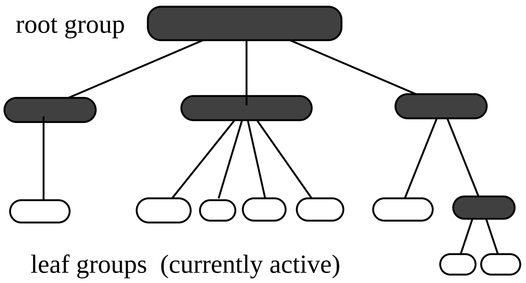


Figure 2: The group hierarchy in ForkLight forms a logical tree.

local variables. In contrast to Fork95 there are no shared formal parameters in ForkLight.

Control–synchronous functions contain an implicit group–wide barrier synchronization point at entry and another one after return, in order to re–install control–synchronous execution also when the function is left via different `return` statements. Note also that the return value of a non–`void` function is always private.

2.3 The group concept

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 (see Figure 2).

Subgroups of a group can be distinguished by their *group ID*. The group ID of the leaf group a thread is member of can be accessed through the shared variable `@`. Differently from Fork95, `@` can be explicitly redefined by the programmer only at the `fork` statement. `join` and `start` initialize `@` to 0. Moreover, in ForkLight, the group–relative thread ID `$` is read–only and automatically computed whenever a new subgroup is created, by renumbering the subgroup's p threads consecutively from 0 to $p-1$. Both the group–relative thread ID `$` and the group ID `@` are automatically saved when splitting the current group, and restored when reactivating it. Thus, the value of `@` and that of `$` remain constant throughout the lifetime of a group.

Also, all threads within the same group have access to a common shared address subspace. Thus, newly allocated shared objects exist once for each group allocating them. A thread can inspect the number of threads belonging to its current group using the routine

```
int groupsize();
```

At the entry into 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 prevent the programmer from errors based on this scenario, ForkLight guarantees control–synchronicity by suitable 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. The only shared variable which can always be assumed to be stable is `@` (see later), thus expressions that only involve `@` and constants are stable.

Technically, the compiler defines a conservative, statically computable subset of the *stable expressions* as follows:

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

Conditional branches with a stable condition expression do not affect control–synchronicity. Otherwise control–synchronicity can generally be lost; this could result in unforeseen race conditions or deadlock. For this reason, unstable branches in control–synchronous mode lead to a splitting of the current group into subgroups — one for each possible branch target. Control synchronicity is then only be maintained within the subgroups. 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. [4]

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.

Unstable branch conditions in control–synchronous mode may nevertheless be useful for performance tuning. Thus, there is the possibility of a pseudocast

```
stable (<expr>)
```

which causes the compiler to treat expression `<expr>` as stable. In this case the compiler 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. [5]

Splitting a group into subgroups can also be done explicitly, using the `fork` statement, which is only admissible in control–synchronous regions. 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>`[6]. 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.

2.3.1 Calling control–synchronous functions

At entry to a control–synchronous function there is an implicit barrier synchronization of the current group, in order to guarantee that local shared variables of the callee can be properly allocated.

2.3.2 `break` and `goto`

For `break` jumps, the target group is statically known; it is an ancestor of the current leaf group in the group hierarchy tree. In this case, the compiler will provide a safe implementation even for the control–synchronous mode.

By a `goto` jump, control–synchronicity is, in principle, not lost. However, the target group may not yet have been created at the time of executing the jump. Even worse, the target group may not be known at compile time. Nevertheless, as long as source and destination of a `goto` are known to be within the activity scope of the same (leaf) group, there is no danger of deadlock. For this reason, we have renounced to forbid `goto` in control–synchronous regions, but all other cases cause a warning to be emitted.

2.4 Pointers and heaps

The usage of pointers in ForkLight is slightly more restricted than in Fork95 since we no longer assume that the private address subspaces are embedded into the global shared memory. Thus, shared pointer variables must not point to private objects. As it is generally 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 it is legal to make a private pointer point to a shared object which is visible for all threads.

ForkLight supplies three kinds of heaps: one permanent, private heap for each thread, one automatic shared heap for each group, and a global, permanent shared heap. Space on the private heaps can be allocated and freed by the asynchronous functions `malloc()` and `free()` known from C. Space on the permanent shared heap is allocated and freed accordingly using the asynchronous functions `shmalloc()` and `shfree()`. The automatic shared heap is intended to provide fast temporary storage blocks which are local to a group. Consequently, the life range of objects

[4] As a rule of thumb: Implicit barriers are, in control–synchronous mode, generated only at branches of control flow, not at reunifications.

[5] Alternatively, there is the option to declare such control constructs as asynchronous by `async`.

[6] Note that empty subgroups (with no threads) are possible; an empty subgroup's work is immediately finished, though.

allocated on the automatic shared heap by the control–synchronous `shalloc()` function is limited to the life range of the group by which that `shalloc()` was executed. Thus, such objects are automatically removed if the group allocating them is released.

Pointers to functions are also supported in ForkLight. Dereferencing a pointer to a control–synchronous function is only legal in control–synchronous mode if it is stable.

2.5 Standard atomic operations

Atomic *fetch&op* operations, also known as *multiprefix* computations when applied in a fully–synchronous context with priority resolution of concurrent write accesses to the same memory location, have been integrated as standard functions called `fetch_add`, `fetch_max`, `fetch_and` and `fetch_or`, in order 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, determining the size p of the current group and consecutive renumbering $0,1,...,p-1$ of the group–relative thread ID `$` could also be done by

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

where the function–local integer variable `p` is shared by all threads of the current group.

The *fetch&add* machine instruction executed in the course of the evaluation of a `fetch_add()` expression is assumed to work atomically. This is very useful to access semaphores in asynchronous mode, e.g., simple locks that sequentialize access to some shared resource where necessary. Like Fork95, ForkLight offers several types of locks in its standard library: simple locks, fair locks, and reader–writer locks.

The *atomic_op* operators are available as library routines. They can e.g. be used for integer global sum, bitwise OR, bitwise AND, maximum, and minimum computations.

2.6 Enforcing exact synchronicity

There are some cases where exact synchronicity would allow more elegant expression of parallel operations. For instance, a parallel pointer doubling operation would, in synchronous mode of Fork95, just look like

```
a[$] = a[a[$]];
```

under the assumption that the shared integer array `a` has as many elements as there are threads available. In order to guarantee correct execution in ForkLight, one has to introduce temporary private variables and a barrier statement:

```
int temp = a[a[$]];
==============
a[$] = temp;
```

Nevertheless, we claim that this is not a serious restriction for the programmer, because (s)he must use the second variant also in Fork95 if not enough threads are available to perform the operation in one step (i.e., if the number `N` of array elements exceeds the number of threads):

```
int i, *temp, prsize;
prsize = (int)ceil(N/groupsize()));
temp = (int *)malloc(prsize);
for (i=0; i<prsize && $*prsize+i<N, i++)
   temp[i] = a[a[$*prsize+i]];
============ // only required in ForkLight
for (i=0; i<prsize && $*prsize+i<N, i++)
   a[$*prsize+i] = temp[i];
free(temp);
```

Currently under discussion is an extension of the current ForkLight design by a statement

```
strict <stmt>
```

that causes fully–synchronous execution for a statement. A drawback of `strict` is that the compiler must perform data dependence analysis and generate code that protects the correct order of accesses to the same shared memory location by locks or barriers. Worst–case assumptions may even lead to complete sequentialization. `strict` would be admissible only in control–synchronous regions. Functions called from `<stmt>` would be executed in control–synchronous rather than fully–synchronous mode.

2.7 Example: Parallel Mergesort

The parallel mergesort algorithm sorts an array of n items (here: integers) with p threads by splitting the

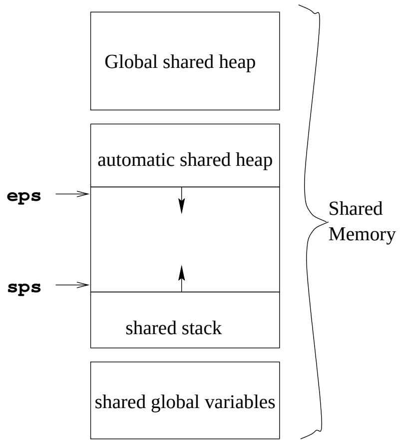


Figure 3: Organization of the shared memory.

array into two halves, sorting these recursively in parallel by $p/2$ threads each, and then merging the two sorted subarrays in parallel into one sorted array. The recursion stops if either the size of the array to be sorted falls below a certain threshold, or if the number of threads available for sorting becomes 1. In both cases a sequential sorting algorithm (e.g. the native `qsort()` routine) may be applied. The run time is $O(n/p \cdot \log p \log n)$, assuming that an optimal sequential sorting algorithm is used for the sequential parts. A ForkLight implementation of the mergesort algorithm is given in appendix A.

3 Compilation of ForkLight

To compile ForkLight programs, we first allocate 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. Group splitting operations also cause a splitting of the remaining shared stack space where we maintain an own shared stack for each subgroup; thus the shared stack has the form of a cactus stack. The automatic shared heap is handled in the same way (a cactus heap); it is allocated at the other end of this shared memory. The shared stack pointer `sps` and the automatic shared heap pointer `eps` may be permanently kept in registers on each thread. A shared stack or automatic heap overflow thus occurs if the `sps` crosses the `eps` or vice versa (see Figure 3).

Another shared memory slice is allocated to install the global shared heap accessed by `shmalloc()` and `shfree()`.

Clearly also a private stack is arranged in each thread's private memory (maintained by the native C compiler) and each thread's private heap can be accessed by the native functions `malloc()` and `free()`. As usual, at each function call a (private) procedure frame is allocated by the native C compiler on each thread's private stack, holding the procedure arguments, saved registers, the return address, and private local variables.

When calling a control–synchronous function which has shared local variables declared at top level, a shared procedure frame is allocated on the group's shared stack. An asynchronous function never has a shared procedure frame.

Shared block–local (and thus, group–local) variables are allocated in a shared group frame on the shared stack, pointed to by the shared group frame pointer `gps`. Each shared group frame is complemented by a private group frame on the private stack, containing the private group information.

Intermixing procedure frames and group frames on the same stack is not harmful, since occurrences of the subgroup-creating language construct `fork` are always properly nested within a function. Thus, separate stacks for group frames and procedure frames are not required, which preserves scarce memory resources from additional fragmentation.

In this section we will explain creation and usage of these frames and the compilation of the most important language constructs into C source code with calls to the basic shared memory access operations.

3.1 Group frames and group–wide barrier synchronization

To keep everything consistent, the compiler builds shared and private group frames at each group–forming statement.

A *shared group frame* (see Figure 4) is allocated on each group's shared stack (in its shared memory subspace). For each thread belonging to a group, the register `gps` points to the group's shared group frame. The shared group frame contains the shared variables local to this group. Moreover, it contains *three synchronization cells* $sc[0]$, $sc[1]$, $sc[2]$. Each thread holds a private counter `csc` pointing to the *current synchronization cell*, so it holds one of the indices[7] 0, 1, or 2. If no thread of the group is currently at a barrier synchronization point, the current synchronization cell

[7]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.

		group–local shared variables
	$sc[2]$	synchronization cells
	$sc[1]$	
gps→	$sc[0]$	

Figure 4: Shared group frame

	`$`	thread ID local to this group
	`@`	this group's ID
	`csc`	current synchronization cell index
	`ignoredframes`	counts non–allocated group frames
	old `gps`	parent group's `gps`
	old `eps`	parent group's `eps`
	old `sps`	parent group's `sps`
gpp→	old `gpp`	parent group's `gpp`

Figure 5: Private group frame

contains just the exact number of threads belonging to this group.

At a (group–wide) barrier synchronization point, each thread first atomically increments the next synchronization cell by 1 (in order to initialize the following barrier synchronization) and then atomically decrements the current synchronization cell by 1 and waits until it sees a zero in the current synchronization cell.

```
_barrier:
R_csc ← csc      // private load
R_next = R_csc + 1;
if  (R_next > 2)   R_next = 0     // (wrap–around)
atomic_add sc[R_next], 1
atomic_add sc[R_csc], -1
while (SMread(sc[R_csc] ≠ 0) ; (wait)    // load_sh
csc← R_next      // private store
```

The algorithm guarantees that all threads have reached the barrier when threads see a zero 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 point. Once the current synchronization cell $sc[R_{csc}]$ reaches value 0, all threads of the group are guaranteed to see this, as this value remains unchanged at least until after the following synchronization point. [8]

The run time is, for most shared memory systems, dominated by the group–wide atomic increment and the `SMread` accesses to shared memory, while all other operations are performed locally.

The corresponding *private group frame* (see Figure 5), pointed to by register `gpp`, is allocated on each thread's private memory subspace. It mainly contains the current values of the group ID `@`[9] , the group–relative thread ID `$`, and the current synchronization cell index `csc`. The `ignoredframes` entry will be explained later. 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 private group frame pointer `gpp`, thus the parent group can easily be restored when being reactivated.

For each group, its *automatic shared heap* is installed at the end of its shared memory subspace. The pointer `eps` to the current lower boundary of the heap is saved in the private group frame at each subgroup–forming operation that splits the shared memory subspace further, and restored after returning to that group. Testing for shared stack or heap overflow thus just means to compare `sps` and `eps`.

3.2 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 allocates a shared procedure frame on the shared stack. These locations should be accessed only when all threads have entered the function. For this reason, there is an implicit barrier synchronization (local to the current group) at the entry point of a control–synchronous function.

3.3 Translation of the `fork` statement

A straightforward implementation of the `fork` statement assumes that all k subgroups will exist and thus distributes shared stack space equally among these. For

`fork ( k; @=e ) <stmt>`

the following code is generated:

(1) $R_k \leftarrow eval(k)$; $R_@ \leftarrow eval(e)$;
(2) **if** $R_@ \geq 0$ **and** $R_@ < R_k$
  set private pointer to future `sc[0]`:
  `sc ← sps+`$R_@$`*(eps-sps)/`R_k;
  `SMwrite( sc, 0 );`

[8] On many parallel machines there is hardware support for *global* barrier synchronization, which is usually more efficient than this explicit handling of semaphores. Thus, where the group is statically known to contain all started processors, the global barrier routine can be called instead.

[9] Note that `@` needs not be stored on the shared group frame since it is read–only.

(3) `barrier` local to the parent group
(necessary to guarantee the zero value in `sc[0]`)
(4) **if** $R_@ \geq 0$ **and** $R_@ < R_k$
$R_\$ \leftarrow$ `fetch_add(sc,1);`
(5) **if** ($R_@ \geq 0$ **and** $R_@ < R_k$) {
if ($R_\$ == 0$) (leader of subgroup $R_@$)
allocate a shared group frame starting at `sc`;
allocate a private group frame
(including storing $R_\$$ and $R_@$); }
(6) `barrier` local to the parent group (necessary to guarantee the final subgroup sizes in `sc[0]`)
(7) **if** ($R_@ \geq 0$ **and** $R_@ < R_k$)
enter subgroup $R_@$
by setting `gps` and `gpp` to the new frames
else goto (10)
(8) `<stmt>`
(9) atomically decrement `sc[csc]` by 1 to cancel membership in the subgroup, and leave the subgroup by restoring `gps`, `sps`, `eps`, `gpp`, `spp` from the entries in the private group frame
(10) (next statement)

Allocation of a shared group frame: The current group's free shared memory space (located between shared stack pointer `sps` and shared heap pointer `eps`) is split into k equally–sized blocks $B_0, ..., B_{k-1}$ (see step (2)). The leader ($R_\$ = 0$) of each subgroup j initializes the remaining cells of the shared group frame at the beginning of block B_j, $0 \leq j \leq k-1$, pointed to by `sc` (see step 5); the other processors in the subgroup just set their `gps`, `sps` and `eps` pointers accordingly. [10] Then `sc[1]` and `sc[2]` are initialized to zero, and the `sps` is advanced by the frame size, i.e. three plus the number of shared variables declared locally to the subgroup.

Allocation of a private group frame (see step 5): Each thread allocates 8 cells of private memory and enters the `gpp`, `sps`, `eps` and `gps` pointer of the parent group. The synchronization cell index `csc` is initialized to zero.

Altogether, the overhead of the above implementation mainly consists of the parallel time for three group–wide barriers, one subgroup–wide concurrent `SMwrite`, and one subgroup–wide `fetch_add` operation. Furthermore, 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.

[10] The shared stack memory may be distributed among the groups also in the ratio of subgroup sizes. While this proportional splitting method seems, on the average, to be the better choice with respect to memory fragmentation compared to the uniform splitting of the shared stack, it may nevertheless be disadvantageous if the subgroups turn out to have equal memory requirements independent of the number of threads executing them. A possible solution may be to install a compiler option that causes the compiler to generate code for proportional splitting if desired by the user.

3.3.1 Optimizations

A first optimization may address the fact that subgroup splitting may be completely skipped if the current group consists of only one thread. Then, also group–wide barrier synchronizations can be skipped. A private status flag `ONE` keeps track of this property; it is set to 1 when at entering a subgroup the group size reaches 1, and verified (and reset to 0 if necessary) whenever the thread leaves a subgroup. The number of skipped subgroup constructions may just be handled in a counter `ignoredframes` stored in the private group frame. Initially set to zero when allocating that frame, the counter is incremented when a new subgroup should have been entered, and decremented when it is left. In that case, however, care must be taken if `@` is redefined to a value different from that in the parent group — then the former value of `@` is pushed on the private stack and restored when the redefining subgroup ceases to exist.

A second optimization exploits the observation that 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 $R_@$. The additional cost is an `atomic_incr` operation to a zero–initialized scratch cell executed by the leader ($R_\$ == 0$) of each new subgroup, another barrier on the parent group, and a `SMread` operation, Clearly, clever static program analysis may avoid this dynamic test in some situations. For instance, if the exact space requirements of one subgroup were statically known, all remaining space could be left to the other subgroups. Unfortunately, the presence of non–exact synchronicity, pointers and weak typing makes such analysis quite difficult.

3.4 Translation of `break`

Threads that leave the current group on the "unusual" way via `break`,have to cancel their membership in all groups on the path in the group hierarchy tree from the current leaf group to the group corresponding to the target of that jump statement. The number of these groups is a compile–time constant. For each of these groups (including the current leaf group) g, the threads must cancel their membership in g. This is done by executing step 9 (*exit_subgroup()*) of Section 3.3 for each such group g when `ignoredframes`$= 0$, and decrementing `ignoredframes` by 1 otherwise. Finally, the jumping threads wait at the synchronization point located (in the case of `continue`), at the end of the surrounding loop resp. `switch` for the other threads of the target group to arrive at that point and to re–synchronize with them.

3.5 Accessing local shared variables

Block–local declarations of shared variables enforce a more complicated addressing scheme for these when the block is local to a subgroup–creating construct, e.g. in

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

Clearly, each subgroup created by the `fork` statement needs its own copy of `x`. For this reason, shared local variables are allocated on the shared group frame and addressed relative to `gps`.

Subgroup creation within the same function can be statically nested. The compiler can determine the group nesting depth of the declaration of such a variable `x`, call it $gd(\mathtt{x})$, as well as that of each use of `x`, call it $gu(\mathtt{x})$. For each use of `x`, the compiler computes the difference $d = gu(\mathtt{x}) - gd(\mathtt{x})$ and, where $d > 0$, inserts code to follow the chain[11] 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.

3.6 Optimization of barriers

Implicit barriers are generated by the compiler before shared data is allocated, e.g. when entering a `csync` function with shared local variables. This guarantees that space for them on the shared stack can be safely reused.

In addition, the programmer adds explicit barrier statements where (s)he considers them necessary.

In order to incur minimum overhead due to synchronization, the total number of `barrier`s generated must be minimized. This can e.g. be done by a post-processing of the generated program. First, a program flow graph is constructed for it. Then, elimination of barriers may follow the following rules:

- if between two subsequent barriers for the same group there are no shared memory accesses in between, then one of these barriers can be eliminated.
- if between three subsequent barriers there are only reading accesses to shared memory, or only accesses to different shared memory locations (this requires data dependence analysis), then the middle barrier can be eliminated.

3.7 Virtual processing

Up to now we required the number of threads p (or processes) executing the program to be a run–time constant limited to the hardware resources, i.e. each of the P hardware processors executes exactly one process. Now we describe what has to be done if $p > P$ threads should be generated. In this case, each of the p physical processors could simulate $k = \lceil p/P \rceil$ threads, in a round-robin fashion.

It is important to note that this is impossible to realize for the fully–synchronous language Fork95 without incurring prohibitive overhead in time and space [24]. But the relaxed synchronicity of ForkLight permits a straightforward implementation.

Context switching is often supported by the native language resp. the processor hardware. The remaining question is where to insert context switches in the generated C program.

If no detailed program dependence analysis is made, it is conservatively safe if a context switch is inserted

- before each reading access to shared memory (i.e., `SMread` and `fetch_add`).
- after each writing access to shared memory (i.e., `SMwrite`, `atomic_add` and `fetch_add`).

This implies that for a barrier statement (see subsection 3.1) at least three context switches would be executed. An optimization is also possible here, similar to the optimizations of barriers discussed in the previous subsection; e.g., if between two subsequent context switches no access to shared memory takes place, one of these switches can be eliminated, etc. This means e.g. for the barrier implementation that the context switch inserted immediately after the `atomic_incr` can be eliminated.

More optimizations are possible if data dependencies are computed. Then context switches between subsequent read accesses to the same shared memory location are not necessary if no write access may occur

[11] As an optimization for deeply nested accesses, fast access to the shared local variables declared at the top level of the function (i.e., where $gd(\mathtt{x}) = 0$) can be provided by an additional frame pointer.

in between, and subsequent accesses that are guaranteed to access different shared memory locations need no context switch in between.

4 Performance Results

We have implemented the compiler for the two parallel platforms that are currently accessible to us and that are still supported[12] by P4: multiprocessor SOLARIS workstations and the SBPRAM. As we shall see, 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 three–processor [13] SOLARIS 2.5.1 workstation, where atomic memory access is sequentialized, we observed modest or no speedup:

parallel mergesort for 48000 integers using the host's optimized `qsort()` routine for sequential sorting:

#threads	1	2	3	4
time [ms]	1020	984	1559	3299

parallel mergesort for 48000 integers using a handwritten routine for sequential sorting, thus a more fair comparison:

#threads	1	2	3	4
time [ms]	4390	3581	3619	4650

parallel quicksort for 120000 integers using the host's optimized `qsort()` routine for sequential sorting:

#threads	1	2	3	4
time [ms]	4447	4656	8087	8834

parallel quicksort for 120000 integers using a handwritten quicksort routine for sequential sorting (without synchronization, of course), thus a more fair comparison:

#threads	1	2	3	4
time [ms]	14498	9834	13433	12700

On the SBPRAM we obtained an important improvement by exploiting its native `fetch_add` and `atomic_add` operators which do not lead to sequentialization, in comparison to standard P4 which does not efficiently support atomic fetch-add or atomic-add.

For the 128 PE prototype [14] of the SBPRAM at Saarbrücken running the SBPRAM operating system PRAMOS, we obtained the following performance results:

parallel mergesort on 1000 integers using the optimized sequential `qsort()` function:

#threads	1	2	4	8	16	32	64
time [ms]	573	373	232	142	88	57	39

parallel mergesort on 10000 integers using a hand-written sequential quicksort routine:

#threads	1	2	4	8	16	32	64
time [ms]	2892	4693	3865	1571	896	509	290

parallel quicksort on 1000 integers using a hand-written sequential quicksort routine:

#threads	1	2	4	8	16	32	64
time [ms]	1519	807	454	259	172	145	130

These figures allow the following interpretations:

- Efficient support for non-sequentializing atomic `fetch_add` and `atomic_add`, as in the SBPRAM or Tera MTA, is essential when running ForkLight programs with large numbers of threads. (ForkLight) executables relying only on pure P4 suffer from serialization and locking/unlocking overhead and are thus not scalable to large numbers of threads.
- On a non–dedicated (multiuser/multitasking) machine like the SOLARIS multiprocessor workstation, performance would be much better for $p > 1$ if the ForkLight run time system had complete control over context switching for its 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.
- Explicit load balancing in an SPMD application may be problematic in particular for small machines (quicksort), or when the machine size does not fit the requirements of the algorithm, e.g. when the number of threads is not a power of 2 (mergesort, $p = 3$).
- Where these requirements are met, our prototype implementation achieves acceptable speedups and

[12] Unfortunately P4 has not been updated since 1993/94 and is thus not available on recent parallel supercomputers. We will thus have to switch to a different portable platform, namely Open-MP, as soon as it becomes available for C. For latest developments, please check the ForkLight WWW page `http://www.informatik.uni-trier.de/~kessler/forklight.html`

[13] For $p >= 4$ at least one processor has to simulate several threads, which dramatically hurts parallel performance because there is no means in P4 programs to release a processor e.g. when waiting at a barrier.

[14] Due to some operating system restrictions, only up to 124 PE's can be used for the application program, the other ones (one PE per processor board) are reserved as I/O servers by PRAMOS.

performance scales quite well even for rather small problem sizes ($N = 1000$).

5 Related Work

For a more general survey on parallel programming languages, we refer e.g. to [32]. Here we focus on imperative parallel programming languages that support the shared memory programming model. Our classification scheme is sketched in Figure 6.

(Control–)Synchronous execution is natural for the data–parallel programming paradigm, which denotes the parallel application of simple operations to the elements of large arrays. Dataparallel programming languages maintain, in general, a sequential flow of program control. This is either naturally enforced by SIMD hardware (e.g., vector processors or array processors), or installed by the compiler when the target is a MIMD machine. Examples of such languages are MODULA-2*, C^* DATAPARALLEL C. The data–parallel Fortran dialects partially allow to exploit also MIMD parallelism in a very restricted way; proper task parallelism is not provided. Typically, all program objects are shared, and parallelism is specified by parallel loops or special dataparallel operators. Nested parallelism, if supported at all, is either specified by nested `forall` loops or induced by nested data structures (like nested parallel lists) that are flattened by the compiler to fit the SIMD hardware, as e.g. in the functional dataparallel language NESL [6].

So-called *task–parallel* or *MIMD* programming languages offer separate, independent control flow for each processor or thread. The language must provide explicit means like barriers, locks, futures, etc., for generation and coordination of parallel activities. Almost all MIMD languages support sequential consistency of shared memory accesses from the programmer's view.

Languages following a recursive fork–join execution model such as *Cilk* [27], *ParC* [4], or Tera-C [9] support nested parallelism, but repeated spawning of threads at run time is an expensive operation. Under some circumstances, fork-join parallel programs can be automatically converted to SPMD programs [15].

In contrast, SPMD languages like Denelcor HEP Fortran [23], EPEX/Fortran [16], PCP [7], Split-C [14], AC [10] start program execution with all available threads and keep their number constant during program execution. The program is often partitioned into parallel and serial sections separated by implicit barriers, i.e. execution is control–synchronous at the top level of program control. Typically, recursively nested parallelism with correspondingly nested levels of control–synchronicity as in ForkLight is not supported; mostly there is only one global name space supported; thus, a parallel recursive divide–and–conquer style as suggested in [5], [11], [17], [20], [21] is not supported. Only PCP has a hierarchical group concept similar to that of ForkLight; nevertheless ForkLight offers more features, like static distinction between control–synchronous and asynchronous code regions, automatic group splitting for unstable branches, more flexible explicit group splitting, shared local variables, group IDs, or the `join` statement.

PRAM languages are those MIMD languages which support exact synchronicity: *Modula-2** [31], FORK [19], Fork95 [24], and *ll* [29]. Although there are still some syntactical similarities between ForkLight and Fork95, there are fundamental differences in semantics: The PRAM programming model, as supported by Fork95, allows the programmer to exploit 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. In contrast, ForkLight seems to constitute an acceptable compromise between automatic thread coordination for ease of programming and efficiency of the generated code.

Valiant's BSP model [33] is an alternative to the sequential consistency of shared memory accesses maintained by most parallel programming languages. Instead, sequential consistency is only valid between so-called supersteps that may contain asynchronous computation with relaxed shared memory consistency. The BSP model does not support nested parallelism; in particular, there is no group concept provided in BSPlib. Clearly, this enforces a less comfortable programming style, such that BSP programming for non–regular problems becomes quite hard. A library extension for C called BSPlib [22] and a programming language [26] based on C++ supporting BSP have been developed at Oxford.

6 Conclusion

We have presented the ForkLight language. ForkLight is suitable for irregular, mainly non–numerical computations. It is intended for compilation to asynchronous MIMD shared memory machines with efficient support for *atomic_incr* and atomic *fetch&incr* operators.

We have shown at the mergesort example that

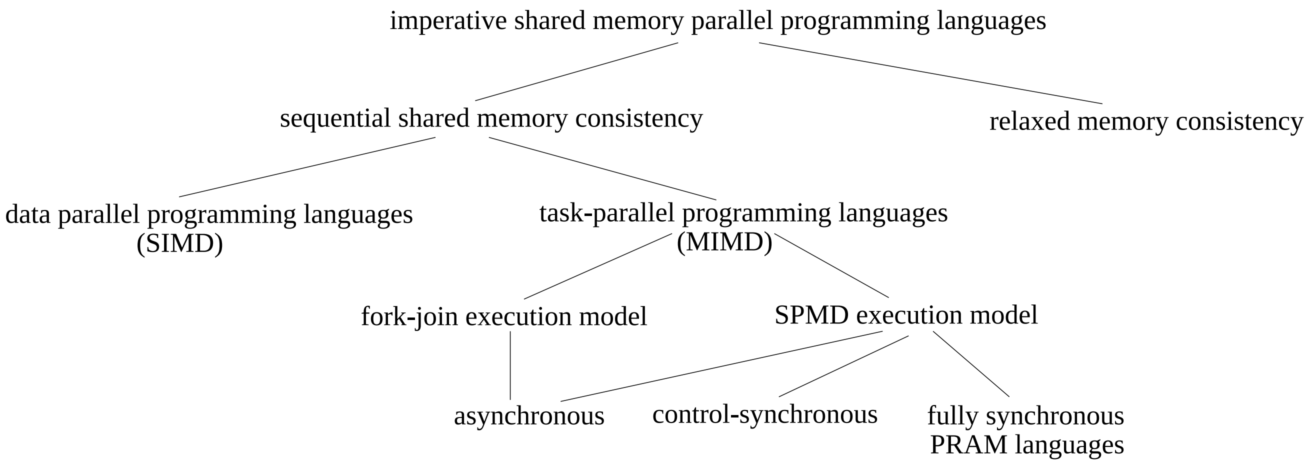


Figure 6: Classification of imperative parallel programming languages for the shared memory model

ForkLight offers similar expressibility as its fully–synchronous predecessor Fork95 and permits writing highly readable code. 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.

The SPMD model of execution that limits the number of threads to a run–time constant forces the programmer to write scalable parallel code and to think about economic usage of the given parallel machine partition, in contrast to fork-join languages that incur a higher overhead for scheduling and hide the scheduling policy from the user by the system. Nevertheless, by the support of nested parallelism in ForkLight, programming is as comfortable as fork-join languages.

We have furthermore given a source–to–source compilation scheme for ForkLight. Optimizations address avoidance of group splitting for one–thread groups, minimization of barrier synchronizations, and minimization of context switches.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and 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, and B. Smith. The Tera Computer System. In *Proc. 4th ACM Int. Conf. on Supercomputing*, pages 1–6. ACM Press, 1990.

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

[4] Y. Ben-Asher, D. Feitelson, and L. Rudolph. ParC — An Extension of C for Shared Memory Parallel Processing. *Software – Practice and Experience*, 26(5):581–612, May 1996.

[5] P. Bhatt, K. Diks, T. Hagerup, V. Prasad, T. Radzik, and S. Saxena. Improved Deterministic Parallel Integer Sorting. *Information and Computation*, 94, 1991.

[6] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[7] E. D. Brooks III, B. C. Gorda, and K. H. Warren. The Parallel C Preprocessor. *Scientific Programming*, 1(1):79–89, 1992.

[8] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20(4):547–564, April 1994.

[9] D. Callahan and B. Smith. A Future–based Parallel Language for a General–Purpose Highly–parallel Computer. Report, Tera Computer Company, Seattle, WA, `http://www.tera.com`, 1990.

[10] W. W. Carlson and 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.

[11] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.

[12] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM model. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[13] R. Cole and O. Zajicek. The Expected Advantage of Asynchrony. *Journal of Computer and System Sciences*, 51:286–300, 1995.

[14] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing'93*, Nov. 1993.

[15] R. Cytron, J. Lipkis, and E. Schonberg. A Compiler–Assisted Approach to SPMD execution. In *Supercomputing '90*, pages 398–406. Los Alamitos: IEEE Computer Society Press, 1990.

[16] F. Darema, D. George, V. Norton, and G. Pfister. A single-program-multiple-data computational model

for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[17] P. de la Torre and C. Kruskal. Towards a Single Model of Efficient Computation in Real Parallel Machines. *Future Generation Computer Systems*, 8:395–408, 1992.

[18] P. B. Gibbons. A More Practical PRAM model. In *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

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

[20] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. Part I: The Model. *Journal of Parallel and Distributed Computing*, 16:212–232, 1992.

[21] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. Part II: Binary Tree and FFT Algorithms. *Journal of Parallel and Distributed Computing*, 16:233–249, 1992.

[22] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib, the BSP Programming Library. `http://www.bsp-worldwide.org/`, May 1997.

[23] H. F. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3:93–110, 1986.

[24] C. W. Keßler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of Parallel Programming*, 25(1):17–50, Feb. 1997.

[25] C. W. Keßler and H. Seidl. Language Support for Synchronous Parallel Critical Sections. In *Proc. APDC'97 Int. Conf. on Advances in Parallel and Distributed Computing, Shanghai, China*. Los Alamitos: IEEE Computer Society Press, March 1997.

[26] D. Lecomber. An object-oriented programming model for BSP computations. In *Proc. PPECC Workshop on Parallel and Distributed Computing*, March 1995.

[27] C. E. Leiserson. Programming Irregular Parallel Applications in Cilk. In *Proc. IRREGULAR'97*, pages 61–71. Springer LNCS 1253, June 1997.

[28] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennesy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[29] C. León, F. Sande, C. Rodríguez, and F. García. A PRAM Oriented Language. In *EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing*, pages 182–191. Los Alamitos: IEEE Computer Society Press, Jan. 1995.

[30] OpenMP Architecture Review Board. OpenMP: a Proposed Industry Standard API for Shared Memory Programming. White Paper, `http://www.openmp.org/`, Oct. 1997.

[31] M. Philippsen and W. F. Tichy. Compiling for Massively Parallel Machines. In R. Giegerich and S. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 92–111. Springer Workshops in Computing Series, 1991.

[32] D. B. Skillicorn and D. T. (Eds.). *Programming Languages for Parallel Processing*. Los Alamitos: IEEE Computer Society Press, 1995.

[33] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), Aug. 1990.

[34] J. M. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch–and–Add.* PhD thesis, New York University, 1988.

[35] X. Zhang, Y. Yan, and R. Castaneda. Evaluating and Designing Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors. *IEEE Parallel & Distributed Technology*, 4(1):25–42, 1996.

A Example: Parallel Mergesort

```
/* Parallel Mergesort in ForkLight.    C.W. Kessler 8/98
 * Sorts N elements using p processors.
 * Assumes that all elements are different;
 * otherwise the result will be wrong.
 */

#include <ForkLight.h>   // required for ForkLight programs
#include <stdio.h>       // host stdio.h
#include <stdlib.h>      // host stdlib.h

#define THRESHOLD 1

/* print an array a of size n sequentially
 */
void print_array( int *a, int n )
{
 int i;
 printf("Array %p of size %d:\n", a, n);
 for (i=0; i<n; i++) printf(" %d", a[i]);
 printf("\n");
}


/* compare function used by the sequential qsort() routine
 */
int cmp( const void *a, const void *b )
{
  if      (*(int *)a < *(int *)b) return -1;
  else if (*(int *)a > *(int *)b) return 1;
       else                       return 0;
}


/* in sequential compute the rank of key within
 * array of size n, i.e. # array-elements < key
 */
int get_rank( int key, int *array, int n )
{
 int left = 0;
 int right = n-1;
 int mid;
 if (key >= array[n-1]) return n;
 if (key == array[n-1]) return n-1;
 if (key <= array[0]) return 0;
 while (left < right-1) {   /*binary search*/
    // always maintain array[left] <= key < array[right]
    mid = (right+left)/2;
    if (key < array[mid]) right = mid;
    else                  left = mid;
 }
 if (key==array[left]) return left;
 else                  return left+1;
}


/* merge array src1 of size n1 and src2 of size n2
 * into one array dest of size n1+n2.
 * Assertions: p>1, n1*n2>=1, dest array is allocated.
 */
csync void merge( int *src1, n1, *src2, n2, *dest)
{
 sh int iter, iter2;
 sh int *rank12, *rank21;  /*temp. rank arrays*/
 int p = groupsize();
 int i;
 rank12 = (int *)shalloc( n1 * sizeof(int) );
 rank21 = (int *)shalloc( n2 * sizeof(int) );
 seq iter = 0;
 seq iter2 = 0;
 =================
 async
   /* self-scheduling par. loop over rank computations: */
   for (i=fetch_add(&iter,1); i<n1; i=fetch_add(&iter,1))
     rank12[i] = get_rank( src1[i], src2, n2 );
 =================
 async
   for (i=fetch_add(&iter2,1); i<n2; i=fetch_add(&iter2,1))
     rank21[i] = get_rank( src2[i], src1, n1 );
 =================
 /* copy elements to dest using the rank information */
 async for (i=$; i<n1; i+=p)  dest[i+rank12[i]] = src1[i];
 async for (i=$; i<n2; i+=p)  dest[i+rank21[i]] = src2[i];
}


/* mergesort for an array of size n.
 * The sorted array is to be stored in
 * sortedarray which is assumed to be allocated.
 */
csync void mergesort( int *array, n, *sortedarray )
{
 int p = groupsize();
 int i;
 if (stable(p==1)) {
     qsort( array, n, sizeof(int), cmp );
     async for (i=0; i<n; i++)  sortedarray[i] = array[i];
 }
 else
 if (stable(n<=THRESHOLD)) { // parallelism doesn't pay off
     seq qsort( array, n, sizeof(int), cmp );
     ================
     async for (i=$; i<n; i+=p)  sortedarray[i] = array[i];
 }
 else {
    sh int *temp = (int *)shalloc( n * sizeof(int) );
    fork (2; @=$%2)
       mergesort( array + @*(n/2),
                  (1-@)*(n/2) + @*(n-n/2), temp + @*(n/2));
    //============= implicit barrier at merge() call:
    merge( temp, n/2, temp+n/2, n-n/2, sortedarray );
 }
}


void main( void )
{
 start {
   sh int *a, *b;
   int N, j;
   N = 100;
   a = (int *) shalloc( N * sizeof(int) );
   b = (int *) shalloc( N * sizeof(int) );
   =================== // arrays allocated
   async for (j=$; j<N; j+=__P__)  a[j] = N - j;
   =================== // array a initialized
   seq print_array( a, N );
   =================== // array a printed
   mergesort( a, N, b );
   =================== // mergesort completed
   seq print_array( b, N );
 }
}
```

ForkLight special program variables				
name	meaning	sharity	type	remark
`__P__`	total number of available threads	shared	`int`	read–only
`@`	current group ID	shared	`int`	redefinable by `fork`
`$`	current thread ID	private	`int`	automatically renumbered in `fork`

ForkLight function (pointer) type qualifiers			
name	meaning	remark	example for usage
`csync`	declare (pointer to) control-synchr. function		`csync int foo( int k ) {....}`
(none)	declare (pointer to) asynchronous function	default	`sh void (*task[Pmax])(void);`
ForkLight storage class qualifiers			
`sh`	declare a shared variable		`sh int k=1, *pk, a[32];`
(none)	declare a private variable	default	`int i,*pi, b[100];`

ForkLight language constructs: statements				
keyword	meaning	example for usage	region	body
`start`	all threads enter control-synchr. mode	`start { a[$]=$; }`	asynchr.	csync.
`join`	some threads enter control-synchr. mode	`join(;0;99) bustour();`	asynchr.	csync.
`retry`	jump back to the corresponding `join`	`else retry;`	asynchr.	—
`async`	enter asynchronous mode	`async puts("Hello");`	csync.	asynchr.
`seq`	one thread per group enters asyn. mode	`seq puts("Hello");`	csync.	asynchr.
`fork`	split current group in subgroups	`fork(3;@=$%3) fn(@);`	csync.	csync.
`===`	group–wide barrier synchronization	`f1(); ======= f2();`	both	—

ForkLight standard functions for atomic operations					
name	meaning	example for usage	mode	operands	result
`fetch_add(ps,ex)`	multiprefix sum	`$=fetch_add(&p,1);`	both	`int *`, `int`	`int`
`fetch_max(ps,ex)`	multiprefix maximum	`fetch_max(&m,a[$]);`	both	`int *`, `int`	`int`
`fetch_and(ps,ex)`	multiprefix bitwise AND	`k=fetch_and(&m,0);`	both	`int *`, `int`	`int`
`fetch_or(ps,ex)`	multiprefix bitwise OR	`k=fetch_or(&m,1);`	both	`int *`, `int`	`int`
`atomic_add(ps,ex)`	atomic increment	`atomic_add(&p,1);`	both	`int *`, `int`	none
`atomic_max(ps,ex)`	atomic maximization	`atomic_max(&p,1);`	both	`int *`, `int`	none
`atomic_and(ps,ex)`	atomic bitwise AND	`atomic_and(&p,1);`	both	`int *`, `int`	none
`atomic_or(ps,ex)`	atomic bitwise OR	`atomic_or(&p,1);`	both	`int *`, `int`	none

ForkLight memory allocation routines				
name	mode	type	parameters	meaning
`malloc`	async.	`char *`	`unsigned int`	allocate block on private heap
`free`	async.	`void`	`char *`	free block on private heap
`shmalloc`	async.	`char *`	`unsigned int`	allocate block on global shared heap
`shfree`	async.	`void`	`char *`	free block on global shared heap
`shalloc`	csync	`char *`	`unsigned int`	allocate block on automatic shared heap
`shavail`	async.	`unsigned int`	`void`	get size of free automatic shared heap space

ForkLight group structure inspection				
name	mode	type	parameters	meaning
`groupsize`	both	`int`	`void`	get the number of threads in my current group

Table 1: ForkLight at a glance.

B Technical remarks on the ForkLight compiler package

The technical requirements to run ForkLight on your local system are an ANSI-C compiler like `gcc` and an installation of the P4 library.

B.1 Installation

The ForkLight compiler package can be obtained from the WWW page
`http://www.informatik.uni-trier.de/~kessler/forklight.html`

After unpacking the ForkLight compiler distribution in a directory, the environment variable `FLDIR` has to be set to this directory. Also, `$(FLDIR)/bin` must be added to the search `PATH`.

Furthermore the `Makefile`s in `$(FLDIR)/lib` and `$(FLDIR)/ex` have to be edited in order to set the paths to the P4 installation and to the target compiler appropriately.

Installation is then done just by `make all` in `$(FLDIR)`.

By default, ForkLight uses `gcc -E` as preprocessor. If your system does not support `gcc`, you must change the string `thepreprocessor` in `$(FLDIR)/src/flcc.h` appropriately.

B.2 Compiler options

The ForkLight compiler `flcc` is built in the `$(FLDIR)/bin` subdirectory. To compile ForkLight sources with `flcc` the following flags should be set:

- `-ForkLight` to tell the compiler that the source to be compiled is a ForkLight program rather than ordinary C code.
- `-nprocs`*num* to produce an executable that expects *num* processors, where *num* is a nonnegative integer. If `num` is zero, the run–time system will infer the total number of available processors at program startup.

  If the `-nprocs` option is omitted, the compiler will choose a default value (4).
- `-I$(FLDIR)/include` to find the header files of ForkLight.

As an example, consider the `Makefile` in `$(FLDIR)/ex`.

In addition, some other flags known from standard C compilers, as `-c` (no linking), `-v` (print version), or `-S` (keep assembler file, i.e. here the generated C code) can be used.

The flags for the target compiler and target linker are architecture dependent. They can be looked up in `$(FLDIR)/ex/Makefile` resp. `$(FLDIR)/lib/Makefile`. The target compiler must specify `-DP4` to activate the P4 backend, `-I$(P4_INCLUDE_DIR)` to include the P4 header files, and, if compiling for the SBPRAM, additionally `-DSBPRAM` to activate the SBPRAM–specific optimizations (non-sequentializing atomic operations, fast spawning of parallel processors). The target linker must link the P4 library; for the SBPRAM, additionally the `sbp` library must be linked.

Currently, debugging is only possible on the C level (target compiler flag `-g`) but not on the ForkLight level.