Practical PRAM Programming with Fork

Tutorial

Seminar on parallel programming
Universidad de la Laguna, Tenerife, Spain
December 1999

Christoph Kessler
University of Trier, Germany

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

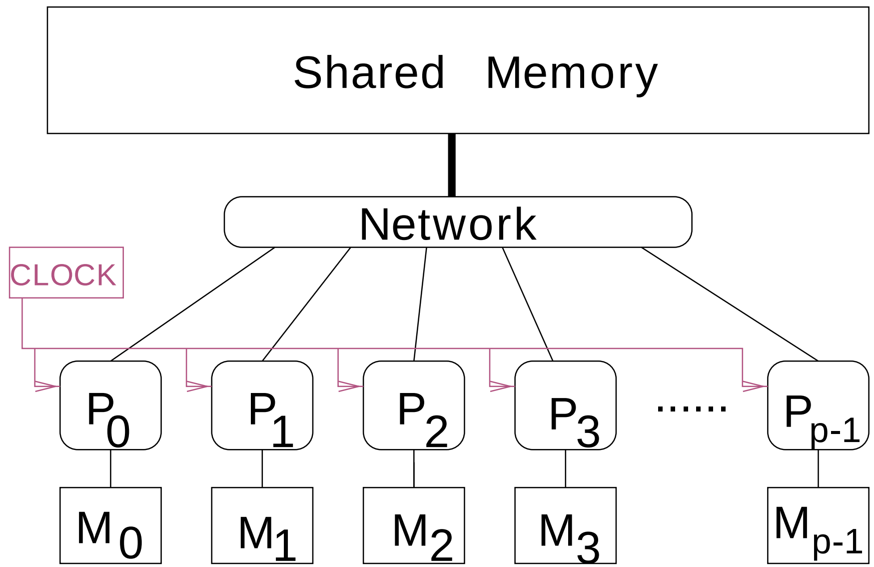
Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

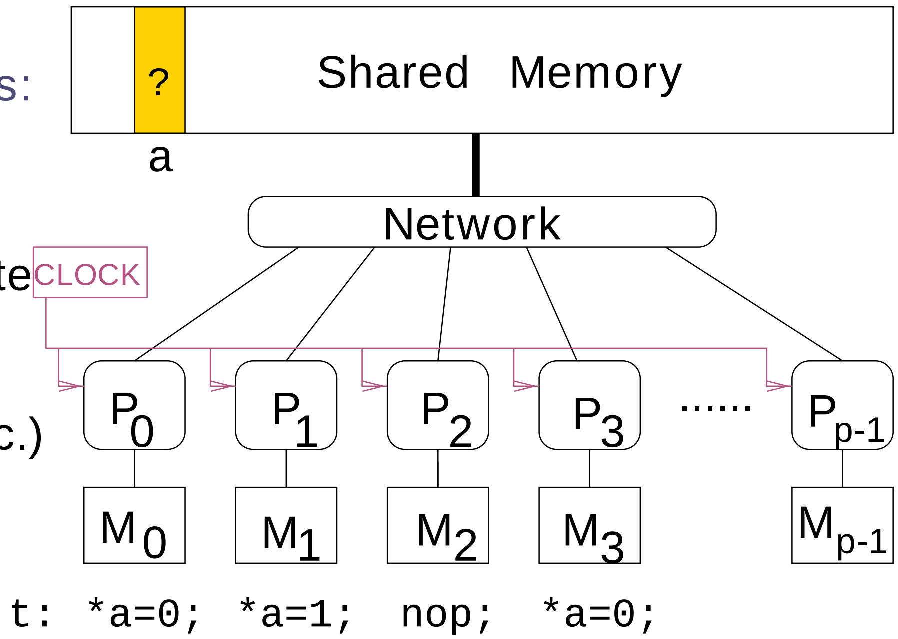
PRAM model

- Parallel Random Access Machine [Fortune / Wyllie '78]
- p processors, individual program control, but common clock signal
- connected by a shared memory with uniform memory access time
- sequential memory consistency (no caches)



PRAM model

- Parallel Random Access Machine
- p processors, individual program control, but common clock signal
- connected by a shared memory with uniform memory access time
- sequential memory consistency (no caches)



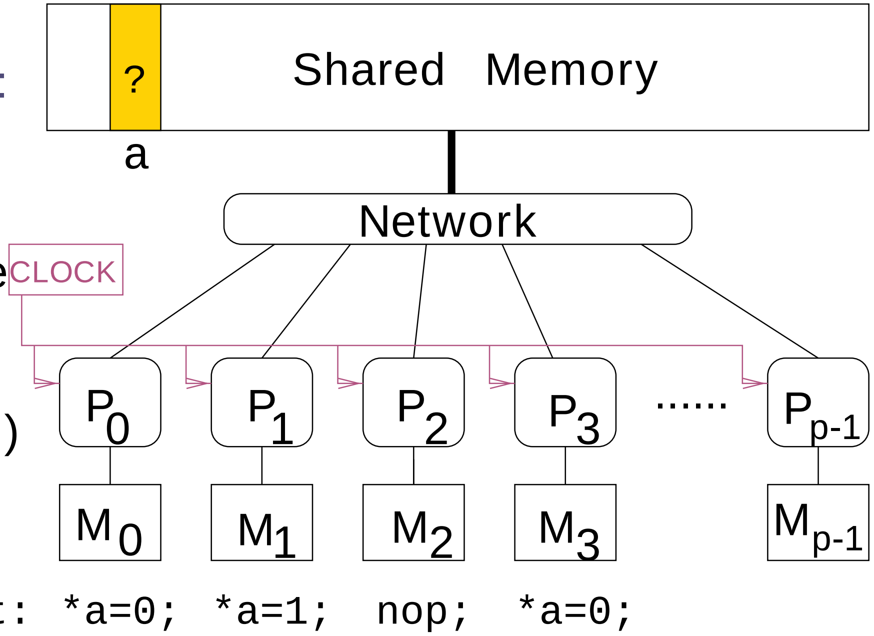
Memory access conflict resolution variants:

EREW = exclusive read, exclusive write
CREW = concurrent read, exclusive write
CRCW = concurrent read, concurrent write

Arbitrary CRCW
Priority CRCW
Combining CRCW (global sum, max, etc.)
......

PRAM model

- Parallel Random Access Machine
- p processors, individual program control, but common clock signal
- connected by a shared memory with uniform memory access time
- sequential memory consistency (no caches)



Memory access conflict resolution variants:

EREW = exclusive read, exclusive write
CREW = concurrent read, exclusive write
CRCW = concurrent read, concurrent write

Arbitrary CRCW
Priority CRCW
Combining CRCW (global sum, max, etc.)
......

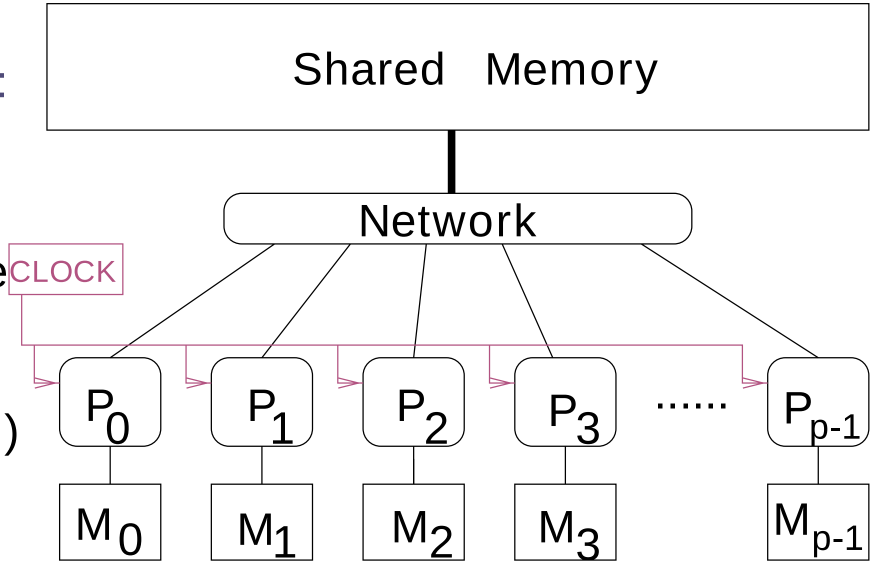
CRCW is stronger than CREW.

Example: Computing logical OR of p bits on a CRCW PRAM in constant time:

```
sh int a = 0;
if (mybit == 1)   a = 1;
```
(else do nothing)

PRAM model

- Parallel Random Access Machine
- p processors, individual program control, but common clock signal
- connected by a shared memory with uniform memory access time
- sequential memory consistency (no caches)



Memory access conflict resolution variants:

EREW = exclusive read, exclusive write
CREW = concurrent read, exclusive write
CRCW = concurrent read, concurrent write

Arbitrary CRCW
Priority CRCW
Combining CRCW (global sum, max, etc.)
......

- easy to understand, popular in theory
- easy to write programs for
- easy to compile for

- but unrealistic ???

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

The PRAM programming language Fork

Fork = Fork95 v.2.0 (1999)

language design started in 1994 [Kessler / Seidl '95]

programming model:

Arbitrary CRCW PRAM with atomic multiprefix operators

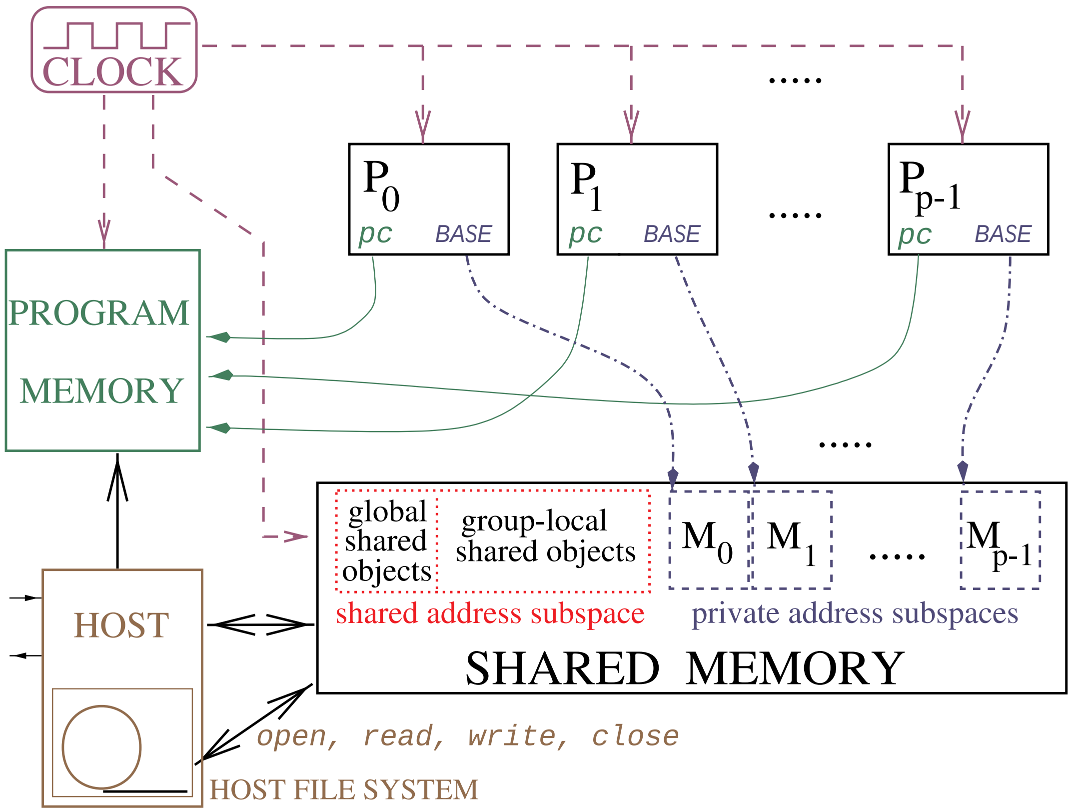
extension of C

MIMD, SPMD execution style
(fixed number of processors)

private address subspaces
embedded in shared memory

synchronicity of the PRAM
transparent at expression level

implementation for SB-PRAM

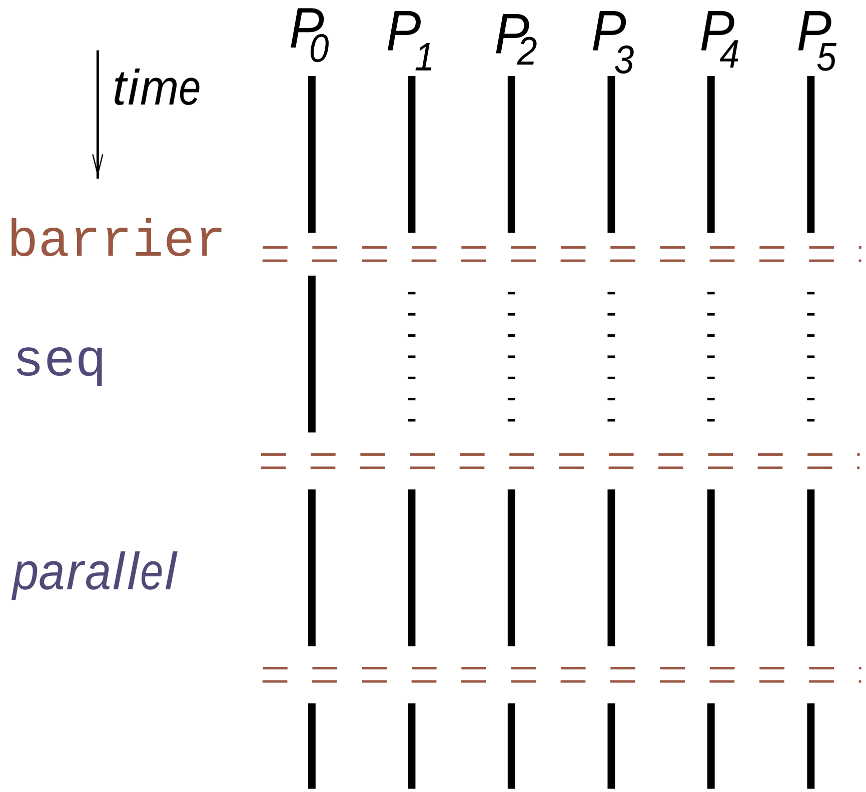


SPMD style of parallel execution

- fixed set of processors

- no spawn() command

- main() executed by all started
  processors as one group



TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

Shared and private variables

- each variable is classified as either shared or private

"sharity"

- sh relates to defining group of processors

- special shared run-time constant variable: `__STARTED_PROCS__`

- special private run-time constant variable: `__PROC_NR__`

```
sh int npr = __STARTED_PROCS__;
pr int myreverse = npr - __PROC_NR__ - 1;
pprintf("Hello world from P%d\n", __PROC_NR__ );
```

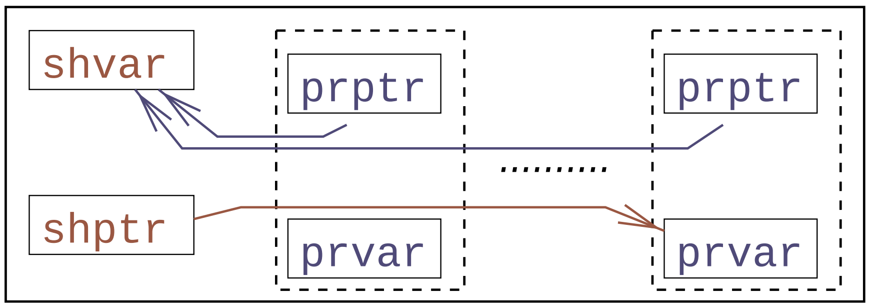
- pointers: no specification of pointee's sharity required

```
pr int prvar, *prptr;
sh int shvar, *shptr;

prptr = &shvar;

shptr = &prvar; // concurrent write!
```

SHARED MEMORY



shared subspace · *private subspace of P0* · .... · *private subspace of P2047*

First steps in Fork: "Hello World"

```
#include <fork.h>
#include <io.h>

void main( void )
{
 if (__PROC_NR__ == 0)
    printf("Program executed by %d processors\n",
           __STARTED_PROCS__ );
 barrier;
 pprintf("Hello world from P%d\n",
           __PROC_NR__ );
}
```

First steps in Fork: "Hello World"

```
PRAM P0 = (p0, v0)> g

Program executed by 4 processors

#0000# Hello world from P0
#0001# Hello world from P1
#0002# Hello world from P2
#0003# Hello world from P3
EXIT: vp=#0, pc=$000001fc
EXIT: vp=#1, pc=$000001fc
EXIT: vp=#2, pc=$000001fc
EXIT: vp=#3, pc=$000001fc
Stop nach 11242 Runden, 642.400 kIps
01fc 18137FFF  POPNG   R6, ffffffff, R1

PRAM P0 = (p0, v0)>
```

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

Atomic Multiprefix Operators (for integers only)

Assume a set P of processors executes simultaneously

```
k = mpadd( ps, expression );
```

Let ps_i be the location pointed to by the ps expression of processor $i \in P$.
Let s_i be the old contents of ps_i.
Let $Q_{ps} \subseteq P$ denote the set of processors i with $ps_i = ps$.
Each processor $i \in P$ evaluates *expression* to a value e_i.

Then the result returned by `mpadd` to processor $i \in P$ is the prefix sum

$$k \leftarrow s_i \quad \sum_{j \in Q_{s_i},\; j<i} e_j$$

and memory location ps_i is assigned the sum

$$*ps_i \leftarrow s_i + \sum_{j \in Q_{s_i}} e_j$$

`mpadd` Example

SHARED MEMORY

`0xe40:`	`0`
`0xf3c:`	`4`

P_0	P_1	P_2	P_3	P_4
`mpadd(0xf3c, 1 );`	`mpadd(0xe40, 2 );`	`mpadd(0xe40, 3 );`	`mpadd(0xf3c, 4 );`	`mpadd(0xe40, 5 );`
returns 4	returns 0	returns 2	returns 5	returns 5

SHARED MEMORY

`0xe40:`	`10`
`0xf3c:`	`9`

`mpadd` may be used as atomic *fetch&add* operator.

Example: User-defined consecutive numbering of processors

```
sh int counter = 0;
pr int me = mpadd( &counter, 1 );
```

Similarly:
`mpmax` (multiprefix maximum)
`mpand` (multiprefix bitwise and)
`mpand` (multiprefix bitwise or)

`mpmax` may be used as atomic *test&set* operator.

Example: `pr int oldval = mpmax( &shmloc, 1 );`

Atomic Update Operators:

`syncadd(`*ps*`, `*e*`)` atomically add value *e* to contents of location *ps*
`syncmax` atomically update with maximum
`syncand` atomically update with bitwise and
`syncor` atomically update with bitwise or

`ilog2(`*k*`)` returns floor of base–2 logarithm of integer *k*

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

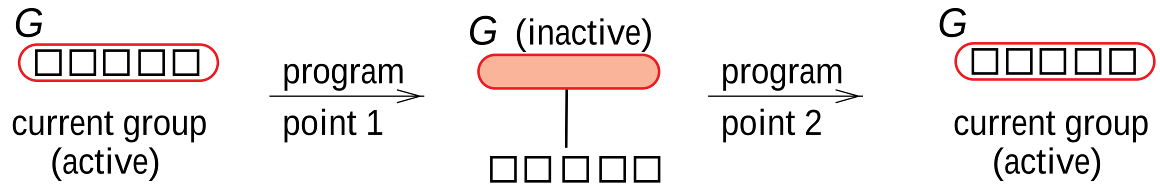
NestStep language concepts

Synchronous, straight, and asynchronous regions in a Fork program

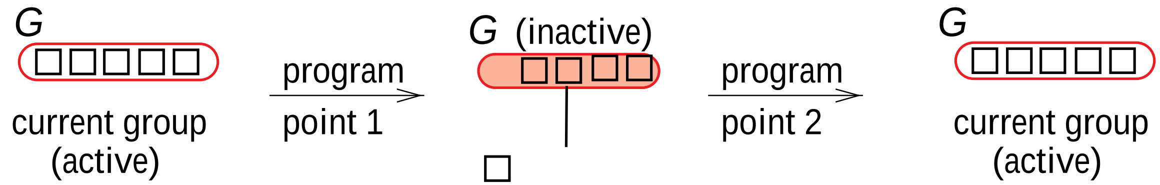
```
sync int *sort( sh int *a, sh int n )
{ extern straight int compute_rank( int *, int);
  if ( n>0 ) {
      pr int myrank = compute_rank( a, n );
      a[myrank] = a[__PROC_NR__];
      return a;
  }
  else
      farm  {
          printf("Error: n=%d\n", n);
          return NULL;
      }
}
extern async int *read_array( int * );
extern async int *print_array( int *, int );
sh int *A, n;
async void main( void )
{
   A = read_array( &n );
   start  {
       A = sort( A, n );
       seq  if (n<100) print_array( A, n );
   }
}
```

Switching from synchronous to asynchronous mode and vice versa

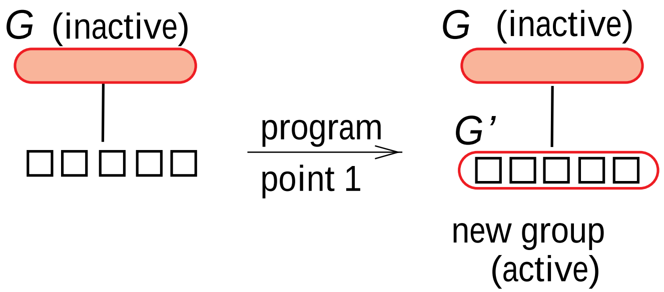
```
farm
    ←  program point 1
  statement;
    ←  program point 2
```

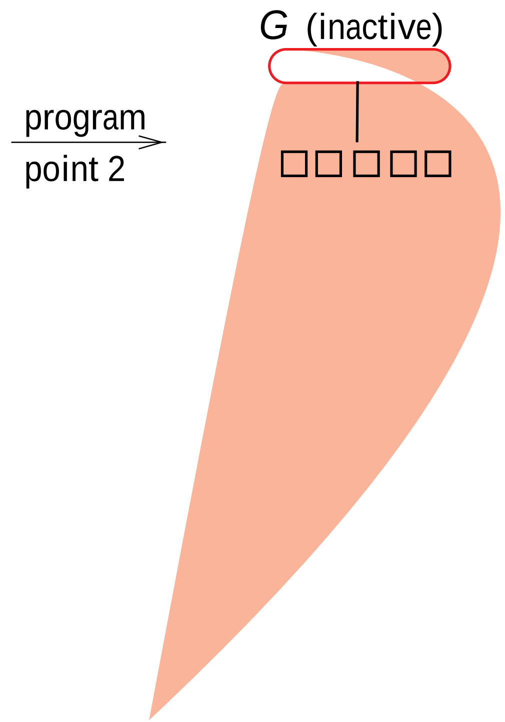



```
seq
    ←  program point 1
  statement;
    ←  program point 2
```




```
start
    ←  program point 1
  statement;
    ←  program point 2
```






```
join (...)

  statement;
```

(see later)

Group concept

P_0 P_1 P_2 P_3

Group ID: @ (set automatically)

Group size: # or `groupsize()`

Group rank: $$ (automatically ranked from 0 to #-1)

Group-relative processor ID: $ (saved/restored, set by programmer)

Scope of sharing for function-local variables and formal parameters

Scope of barrier-synchronization (barrier statement)

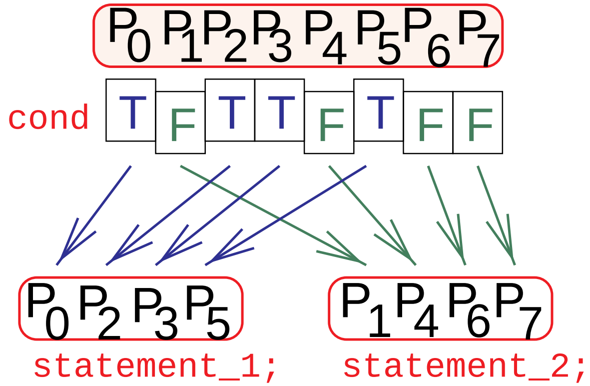
Scope of synchronous execution:

Synchronicity invariant (holds in synchronous regions) :

All processors in the same active group operate synchronously.

Implicit group splitting: private IF statement

```
if (cond)
          statement_1;
else
          statement_2;
```



Private condition may evaluate to different values on different processors
--> current group of processors must be split into 2 subgroups

(parent) group is deactivated while the subgroups are active

new subgroups get group IDs @ = 0 and @ = 1

group-relative processor IDs $ may be locally redefined in a subgroup

group ranks $$ are renumbered subgroup-wide automatically

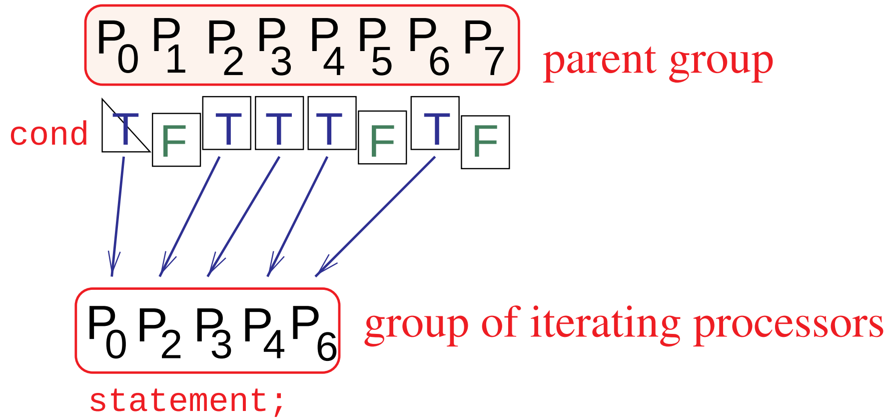
shared stack and heap of parent group is divided among child groups

(parent) group is reactivated (implicit barrier) after subgroups have terminated

Loops with private exit condition

```
while ( cond ) do

      statement;
```



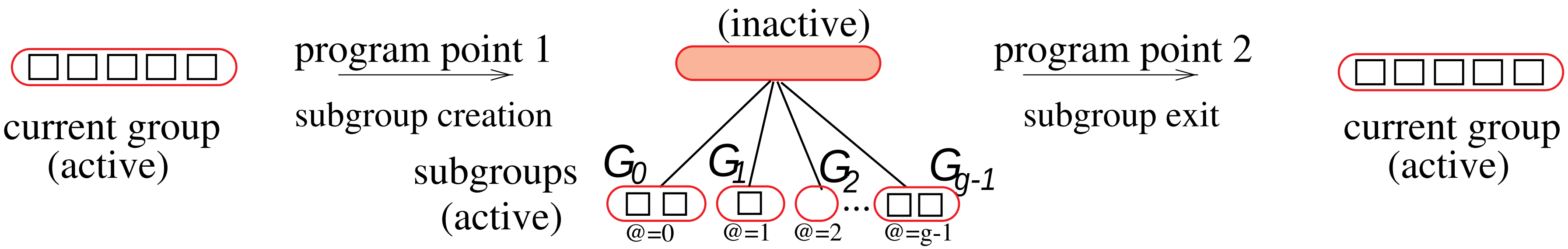
- Processors that evaluate cond to TRUE join the subgroup of iterating processors and remain therein until cond becomes FALSE.

- statement is executed synchronously by the processors of the iterating group.

- As soon as cond becomes FALSE, the processors wait at the end of statement for the others (implicit barrier).

Explicit group splitting: The fork() statement

```
fork ( g; @ = fn($$); $=$$)
                                                   <------ program point 1
     statement;
                                                   <------ program point 2
```

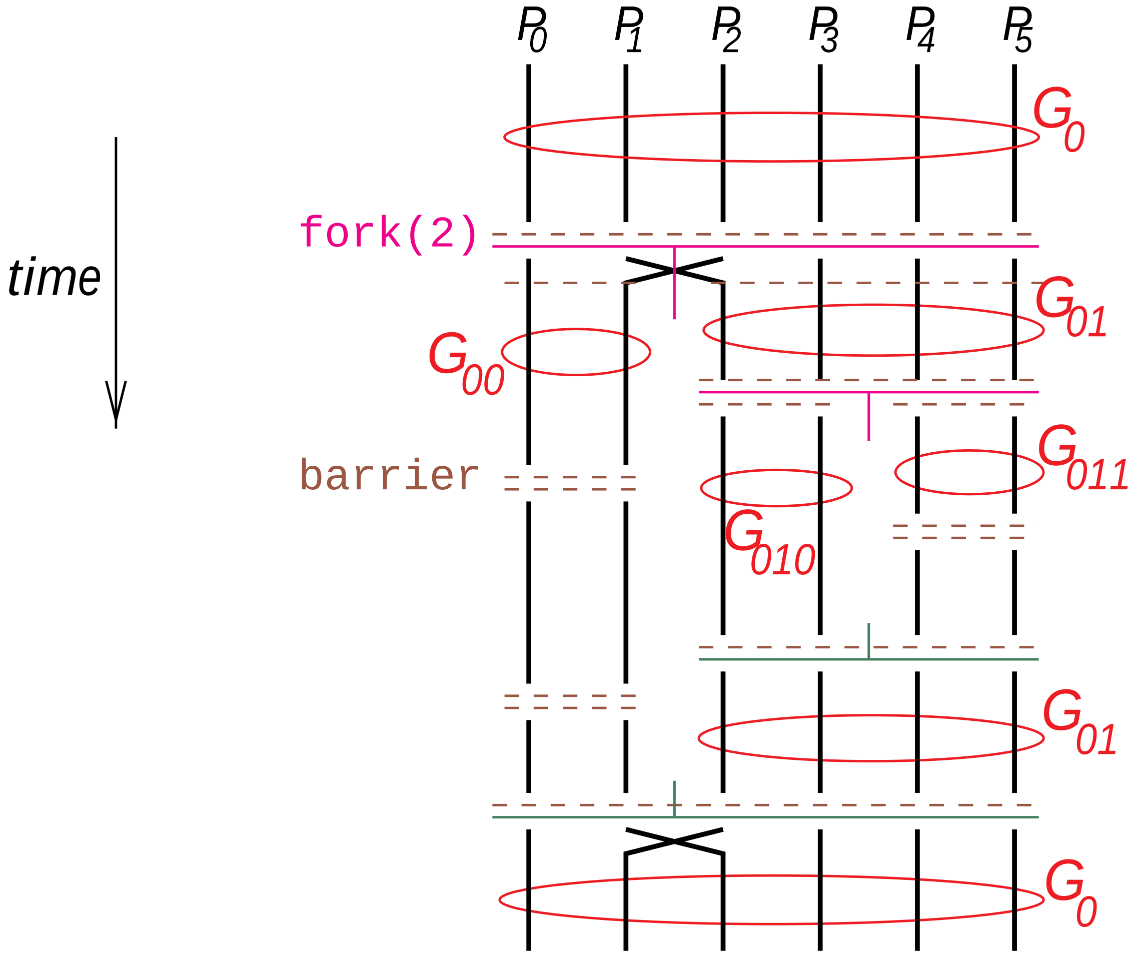


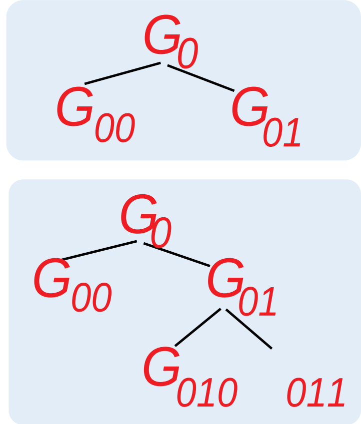
first parameter: current group is split into g subgroups
second parameter: assignment to @, decides about subgroup to join
third parameter (optional): possible renumbering of $ within the subgroups

body statement is executed by each subgroup in parallel

parent group is deactivated while subgroups are active
parent group is reactivated when all subgroups have terminated
(implicit barrier)

Hierarchical processor group concept





Group hierarchy tree

- dynamic group splitting into disjoint subgroups
- groups control degree of synchronicity (also barrier scope) and sharing
- group hierarchy forms a logical tree at any time

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

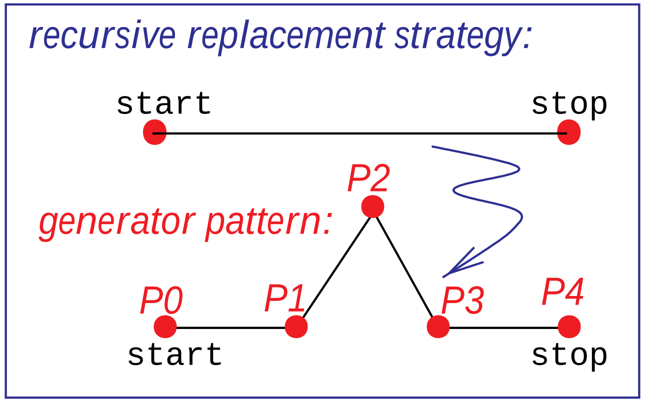
ForkLight language design and implementation

NestStep language concepts

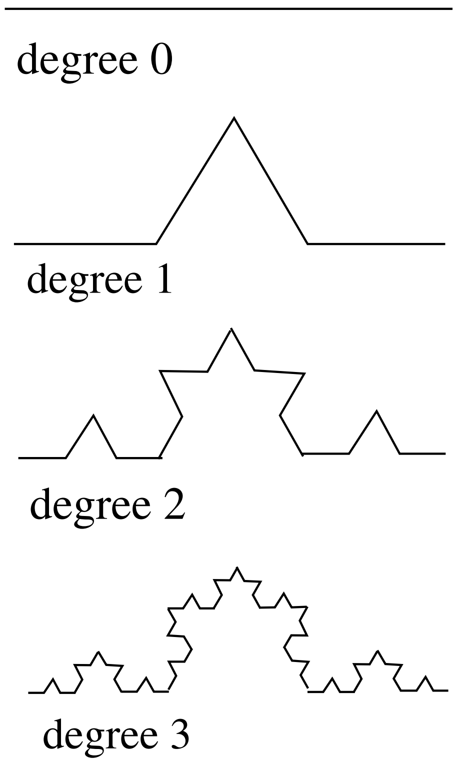
Example: Drawing Koch curves

```
void seq_Koch ( int startx, int starty,
                int stopx, int stopy, int level )
{
 int x[5], y[5], dx, dy;
 int i;
 if (level >= DEGREE) {    // reach limit of recursion:
    seq_line( startx, starty,
              stopx, stopy, color, width );
    return;
 }
 // compute x and y coordinates of interpolation points  P0, P1, P2, P3, P4:
 dx = stopx - startx;       dy = stopy - starty;
 x[0] = startx;             y[0] = starty;
 x[1] = startx + (dx/3);    y[1] = starty + (dy/3);
 x[2] = startx + dx/2 - (int)(factor * (float)dy);
 y[2] = starty + dy/2 + (int)(factor * (float)dx);
 x[3] = startx + (2*dx/3); y[3] = starty + (2*dy/3);
 x[4] = stopx;              y[4] = stopy;

 for ( i=0; i<4; i++ )    //  4  recursive calls
    seq_Koch( x[i], y[i], x[i+1], y[i+1], level + 1 );
}
```



initiator pattern:



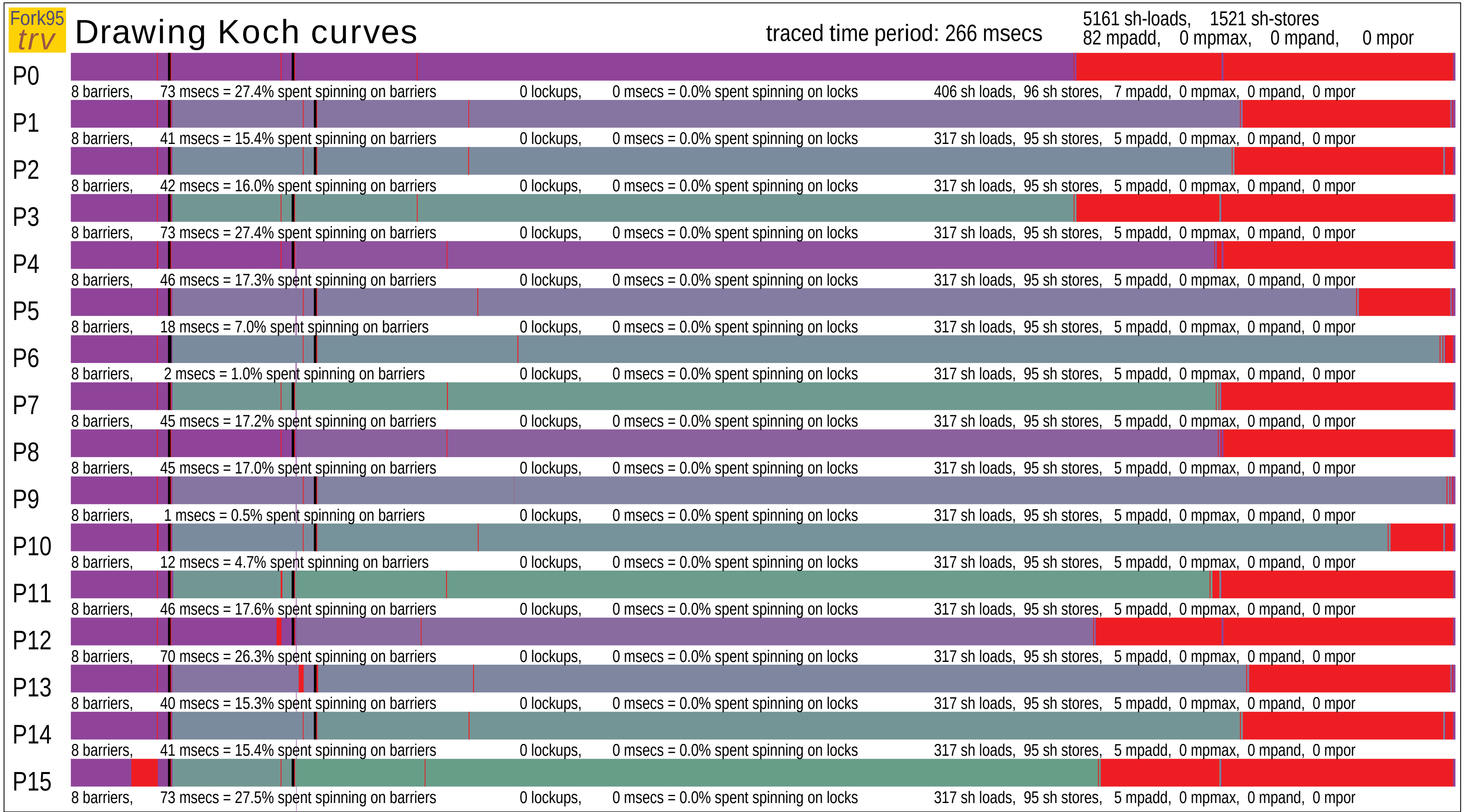
Example: Drawing Koch curves in parallel

```
sync void Koch ( sh int startx, sh int starty,
                 sh int stopx,  sh int stopy,  sh int level )
{
 sh int x[5], y[5], dx, dy;
 pr int i;

 if (level >= DEGREE) {   // terminate recursion:
    line( startx, starty, stopx, stopy, color, width );
    return;
 }
 seq {    // linear interpolation:
    dx = stopx - startx;       dy = stopy - starty;
    x[0] = startx;             y[0] = starty;
    x[1] = startx + (dx/3);    y[1] = starty + (dy/3);
    x[2] = startx + dx/2 - (int)(factor * (float)dy);
    y[2] = starty + dy/2 + (int)(factor * (float)dx);
    x[3] = startx + (2*dx/3); y[3] = starty + (2*dy/3);
    x[4] = stopx;              y[4] = stopy;
 }
 if (# < 4)    // not enough processors in the group?
    for ( i=$$; i<4; i+=# )     // partially parallel divide-and-conquer step
       farm seq_Koch( x[i], y[i], x[i+1], y[i+1], level + 1 );
 else
    fork ( 4; @ = $$ % 4; )     // parallel divide-and-conquer step
       Koch( x[@], y[@], x[@+1], y[@+1], level + 1 );
}
```

Program trace visualization with the trv tool

-T: instrument the target code to write events to a trace file. Can be processed with trv to FIG image



TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

Asynchronous mode: Critical sections and locks

Asynchronous concurrent read + write access to shared data objects constitutes a critical section
(danger of race conditions, visibility of inconsistent states, nondeterminism)

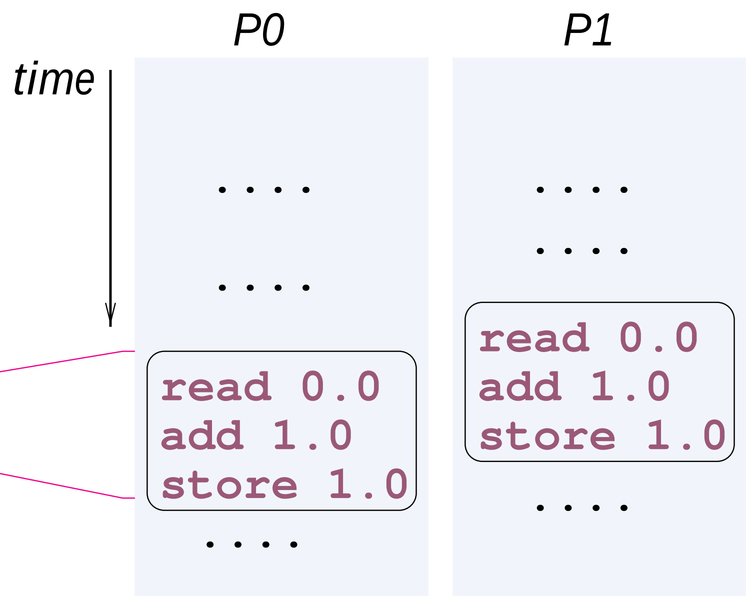
Example:

```
sh float var = 0.0;

....
farm {
   ....


   var = var + 1.0;

   ....
}
```



Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Asynchronous mode: Critical sections and locks

Asynchronous concurrent read + write access to shared data objects constitutes a critical section

(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;
sh int lock = 0; /* mutex var. */
....
farm {
   ....
   while (lock > 0) ;  /* wait */
   lock = 1;
   var = var + 1.0;
   lock = 0;
   ....
}
```

!!NOT ATOMIC!!

time ↓

P0	P1
var = 0.0	var = 0.0
lock = 0	lock = 0
....	....
....	....
lock == 0	lock == 0
lock = 1	lock = 1
read 0.0	read 0.0
add 1.0	add 1.0
store 1.0	store 1.0
....	....

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Access to the lock variable must be atomic as well: fetch&add or test&set

Asynchronous mode: Critical sections and locks

Asynchronous concurrent read + write access to shared data objects constitutes a critical section

(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;
sh int lock = 0;
....
farm {
   ....
   while (mpmax(&lock, 1)) ;
                      /* wait */
   var = var + 1.0;
   lock = 0;
   ....
}
```

time ↓	P0	P1
	var = 0.0	var = 0.0
	lock = 0	lock = 0
	....	....
	....	....
	mpmax: 1	mpmax: 0
	/* wait */	read 0.0
		add 1.0
		store 1.0
		lock = 0
	mpmax: 0	
	read 1.0	....
	add 1.0	
	store 2.0	
	....	

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Access to the lock variable must be atomic as well: fetch&add or test&set

in Fork: use the mpadd / mpmax / mpand / mpor operators

Asynchronous mode: Critical sections and locks

Asynchronous concurrent read + write access to shared data objects constitutes a critical section
(danger of race conditions, visibility of inconsistent states, nondeterminism)

Example:

```
sh float var = 0.0;
sh SimpleLock sl;
seq sl = new_SimpleLock();
....
farm {
   ....
   simple_lockup( sl );
                              /* wait */
   var = var + 1.0;
   simple_unlock( sl );
   ....
}
```

time ↓

P0	P1
var = 0.0	var = 0.0
lock = 0	lock = 0
....	....
....	....
mpmax: 1	mpmax: 0
	read 0.0
/* wait */	add 1.0
	store 1.0
	lock = 0
mpmax: 0	
read 1.0	....
add 1.0	
store 2.0	
....	

Access to var must be atomic.

Atomic execution can be achieved by sequentialization (mutual exclusion).

Access to the lock variable must be atomic as well: fetch&add or test&set

in Fork: alternatively: use predefined lock data types and routines

Predefined lock data types and routines in Fork

(a) Simple lock

```
SimpleLock new_SimpleLock ( void );
void simple_lock_init ( SimpleLock s );
void simple_lockup ( SimpleLock s );
void simple_unlock ( SimpleLock s );
```

(b) Fair lock (FIFO order of access guaranteed)

```
FairLock new_FairLock ( void );
void fair_lock_init ( FairLock f );
void fair_lockup ( FairLock f );
void fair_unlock ( FairLock f );
```

(c) Readers/Writers lock (multiple readers OR single writer)

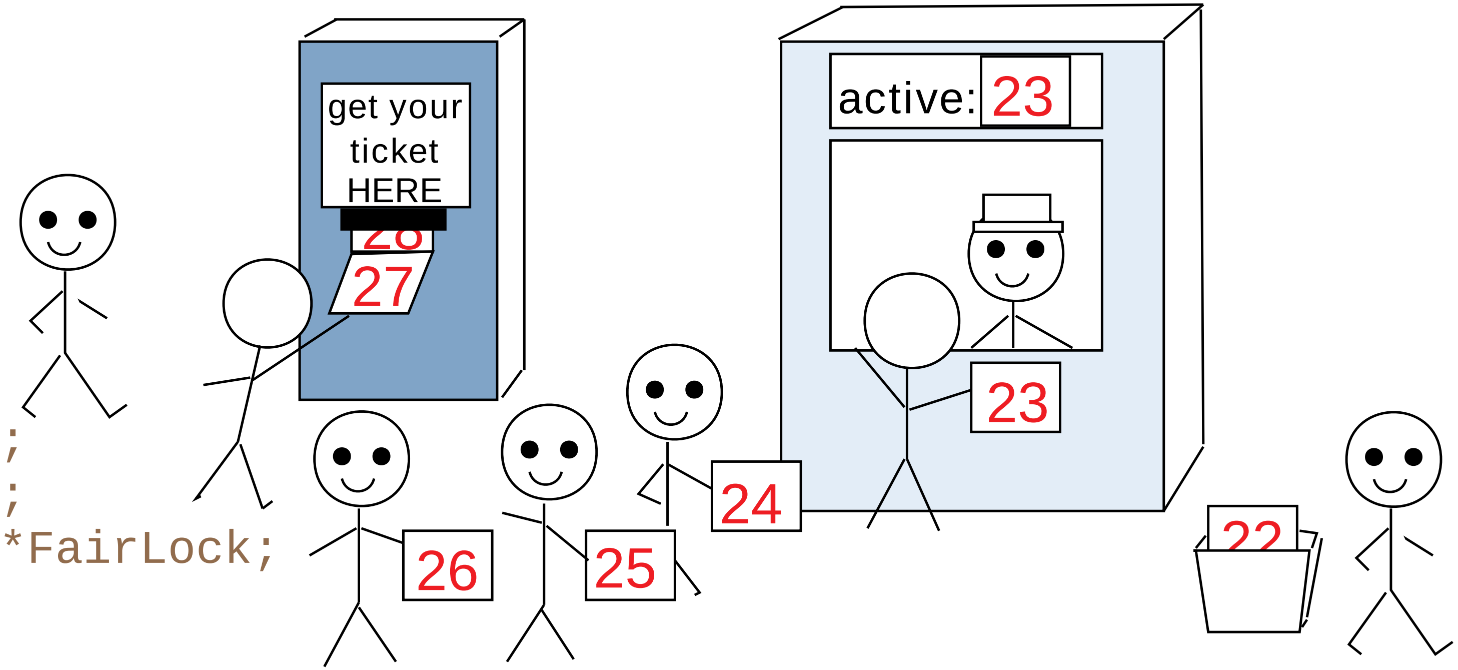
```
RWLock new_RWLock ( void );
void rw_lock_init ( RWLock r );
void rw_lockup ( RWLock r, int mode );
void rw_unlock ( RWLock r, int mode, int wait );
                         mode in { RW_READ, RW_WRITE }
```

(d) Readers/Writers/Deletors lock (lockup fails if lock is being deleted)

```
RWDLock new_RWDLock ( void );
void rwd_lock_init ( RWDLock d );
int  rwd_lockup ( RWDLock d, int mode );
void rwd_unlock ( RWDLock d, int mode, int wait );
                         mode in { RW_READ, RW_WRITE, RW_DELETE }
```

Implementation of the fair lock in Fork

Analogy: booking office management system



2 counters:

```
struct {
   int ticket;
   int active;
} fair_lock, *FairLock;
```

```
void fair_lockup ( FairLock fl )
{
  int myticket = mpadd( &(fl->ticket), 1);/*atomic fetch&add*/
  while (myticket > fl->active) ;   /*wait*/
}
```

```
void fair_unlock ( FairLock fl )
{
  syncadd( &(fl->active), 1 ); /*atomic increment*/
}
```

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

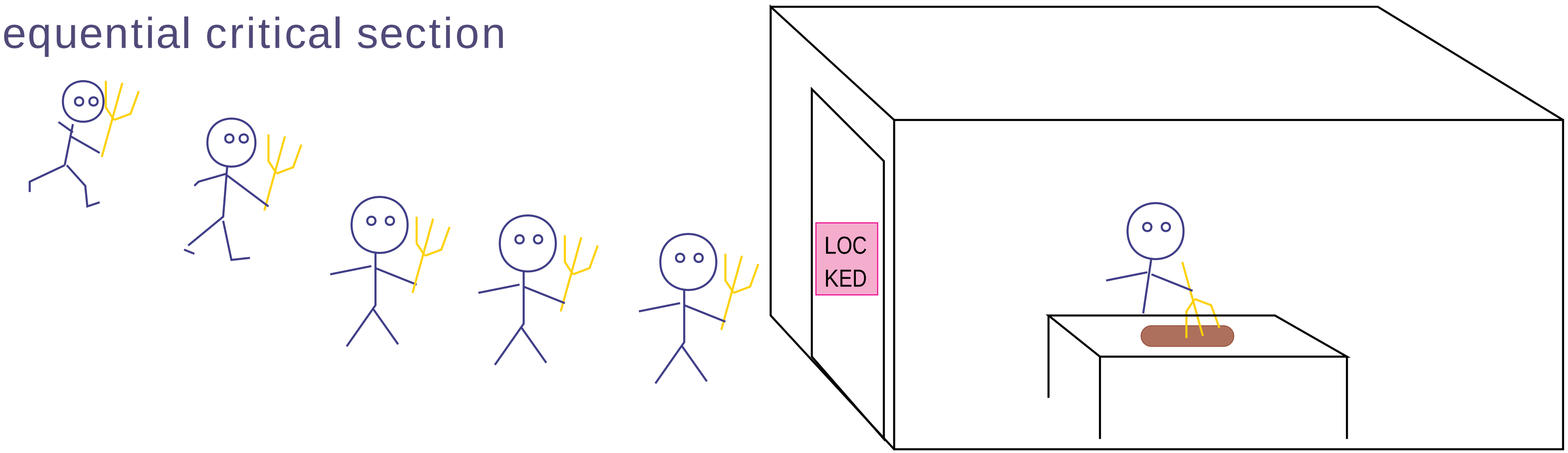
- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

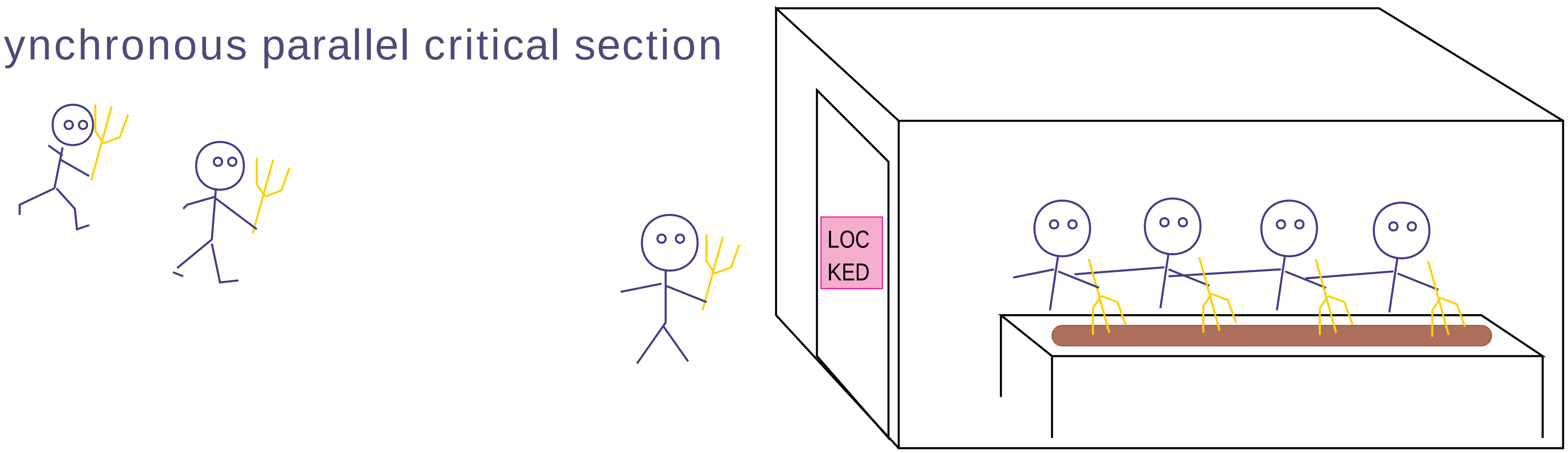
NestStep language concepts

sequential critical section



-> sequentialization of concurrent accesses to a shared object / resource

synchronous parallel critical section

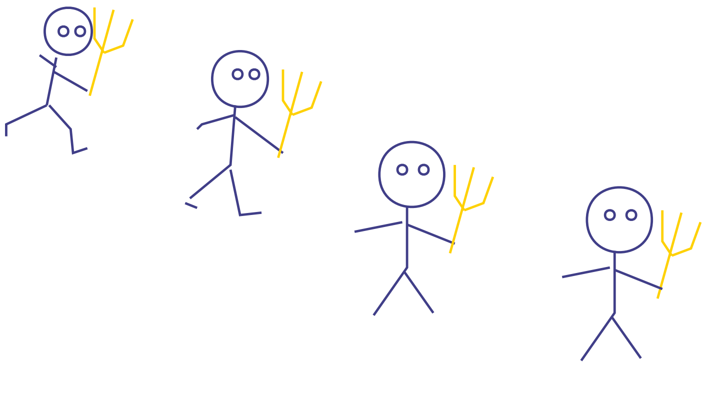


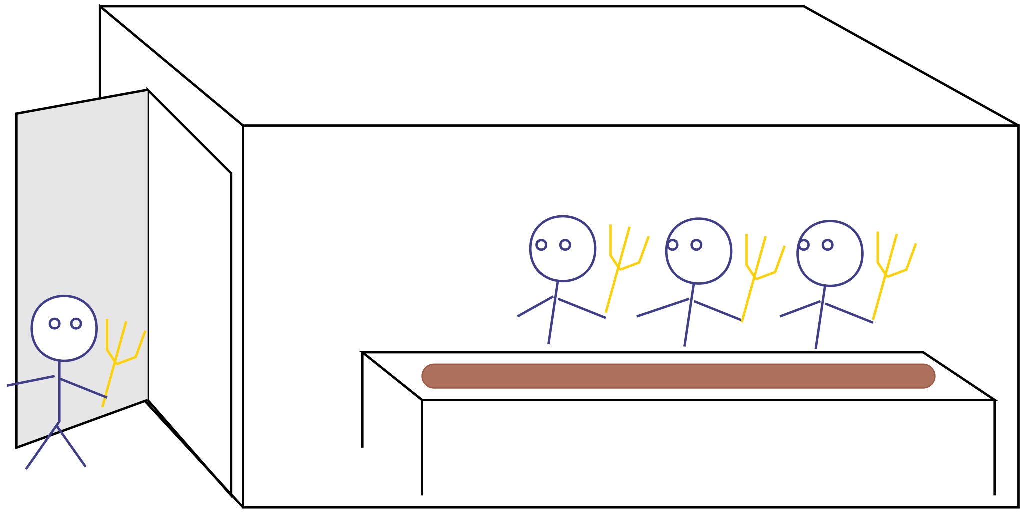
-> simultaneous entry of more than one processor

-> deterministic parallel access by executing a synchronous parallel algorithm

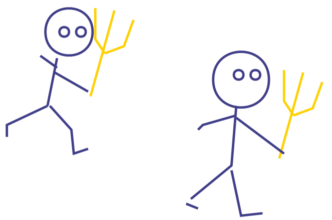
-> at most one group of processors inside at any point of time

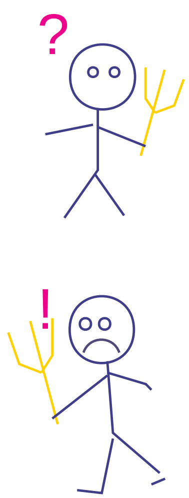
Entry conditions?

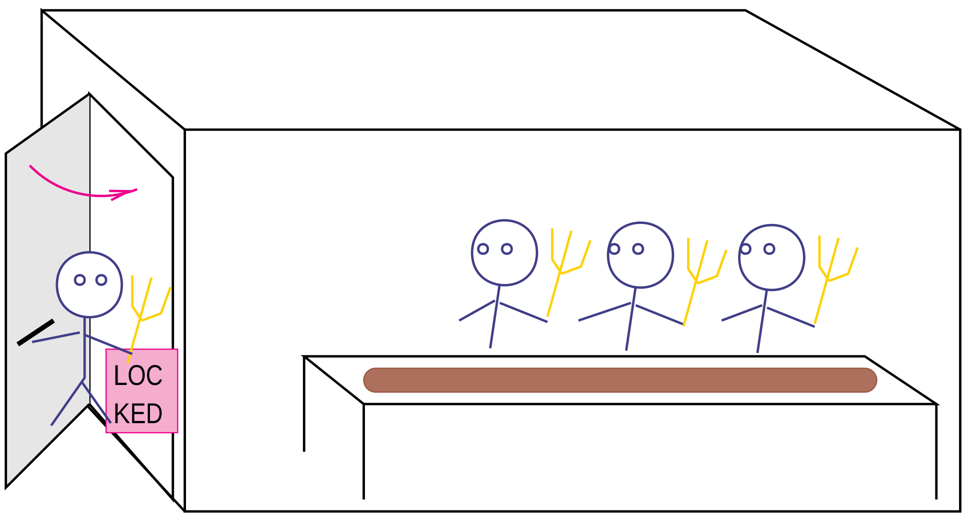




When to terminate the entry procedure?

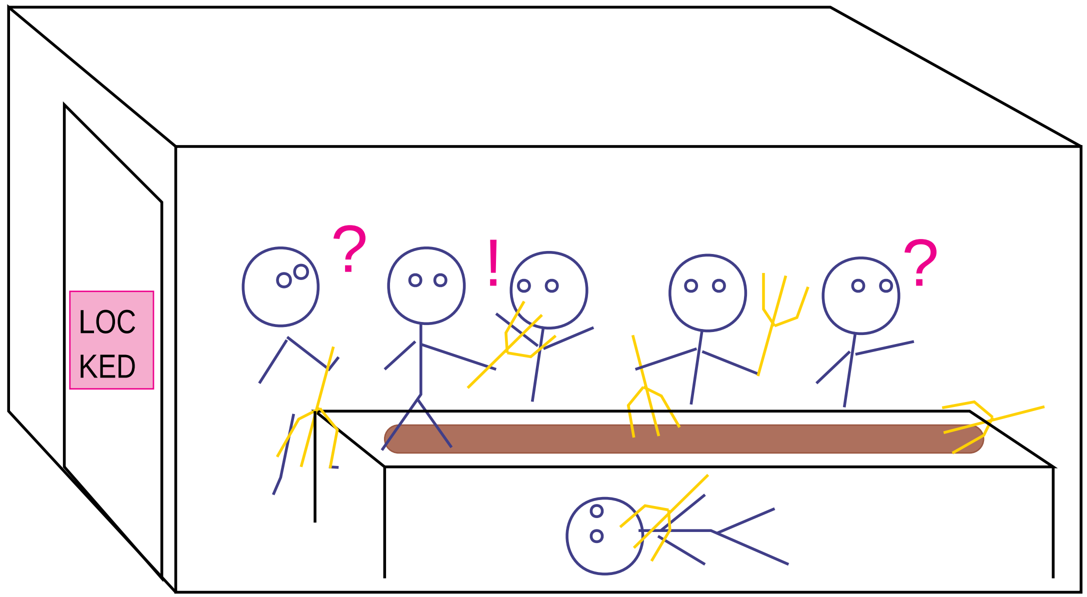






What happens with processors

not allowed to enter?



Need a synchronous parallel algorithm

in order to guarantee deterministic execution!

SYNCHRONOUS PARALLEL CRITICAL SECTIONS

sequential critical sections (e.g. Dijkstra'68):

access to shared object / resource by asynchr. processes
must be mutually exclusive:
guarded by a semaphore (lock)

=> sequentialization of accesses

Idea: synchronous parallel critical section

- allow simultaneous entry of more than one processor

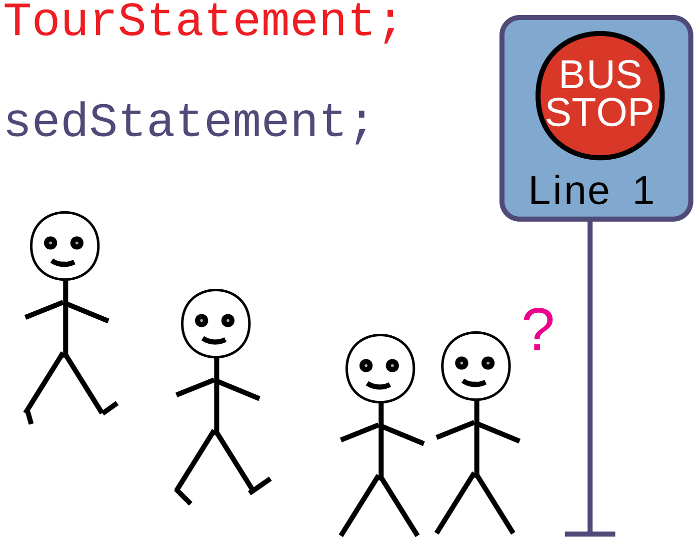
- deterministic parallel execution of the critical section
  by applying a suitable synchronous (PRAM) algorithm

- after termination of the parallel critical section,
  a new bunch of processors is allowed to enter

=> sequential critical section = special case of parallel critical section

The join() statement: The excursion bus analogy

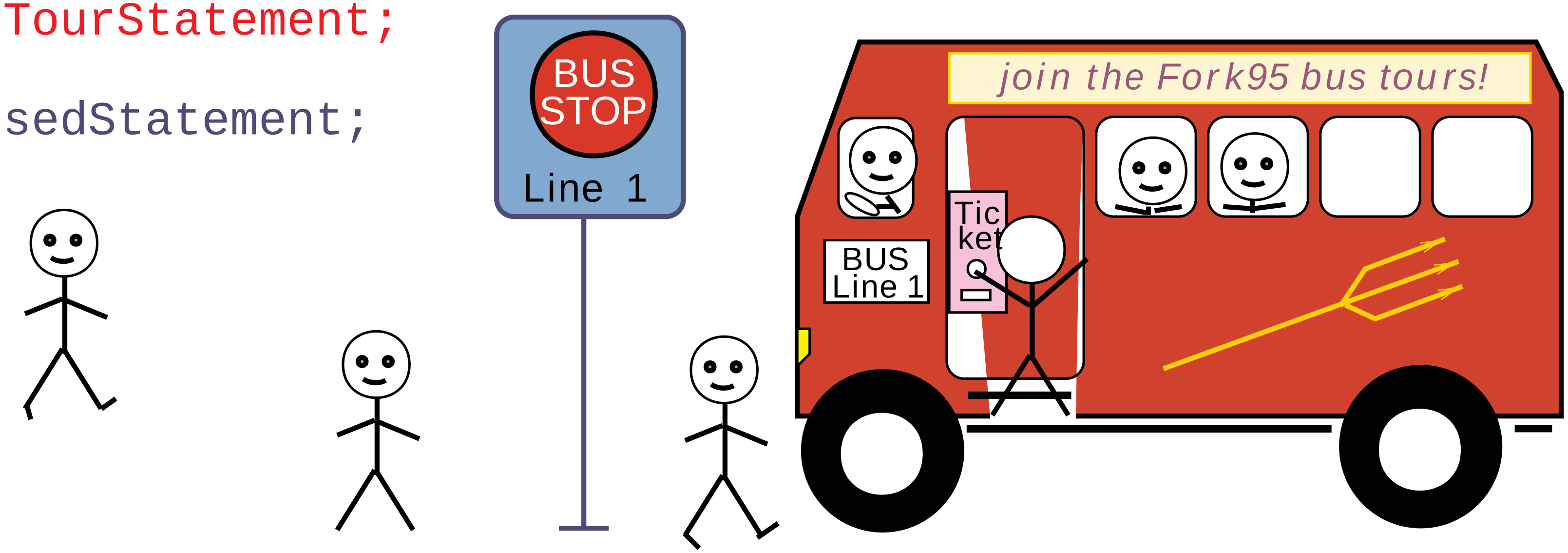
```
join ( SMsize; delayCond; stayInsideCond )
    busTourStatement;
else
    missedStatement;
```



Bus gone?
- execute else part: `missedStatement;`
- continue in else part: jump back to bus stop (join entry point)
- break in else part: continue with next activity (join exit point)

The join() statement: The excursion bus analogy

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

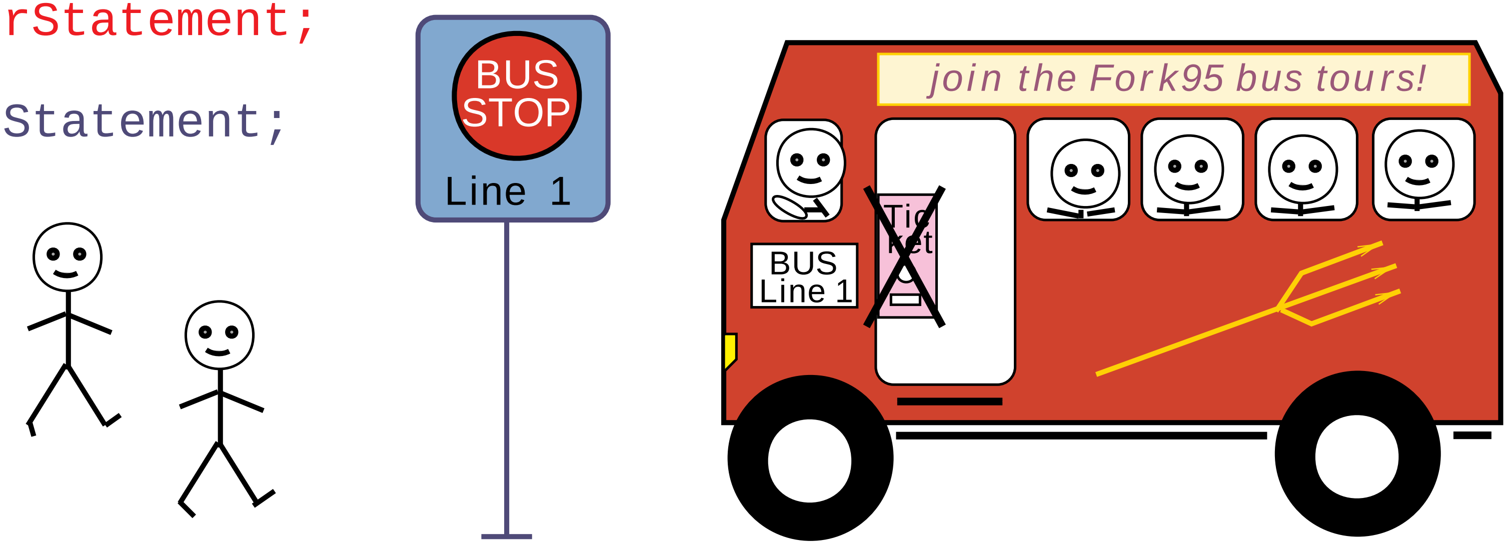


Bus waiting: - get a ticket and enter

- ticket number is 0? -> driver!
  - driver initializes shared memory (SMsize) for the bus group
  - driver then waits for some event: `delayCond`
  - driver then switches off the ticket automaton

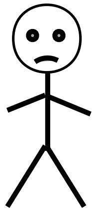
The join() statement: The excursion bus analogy

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



Bus waiting: - get a ticket and enter

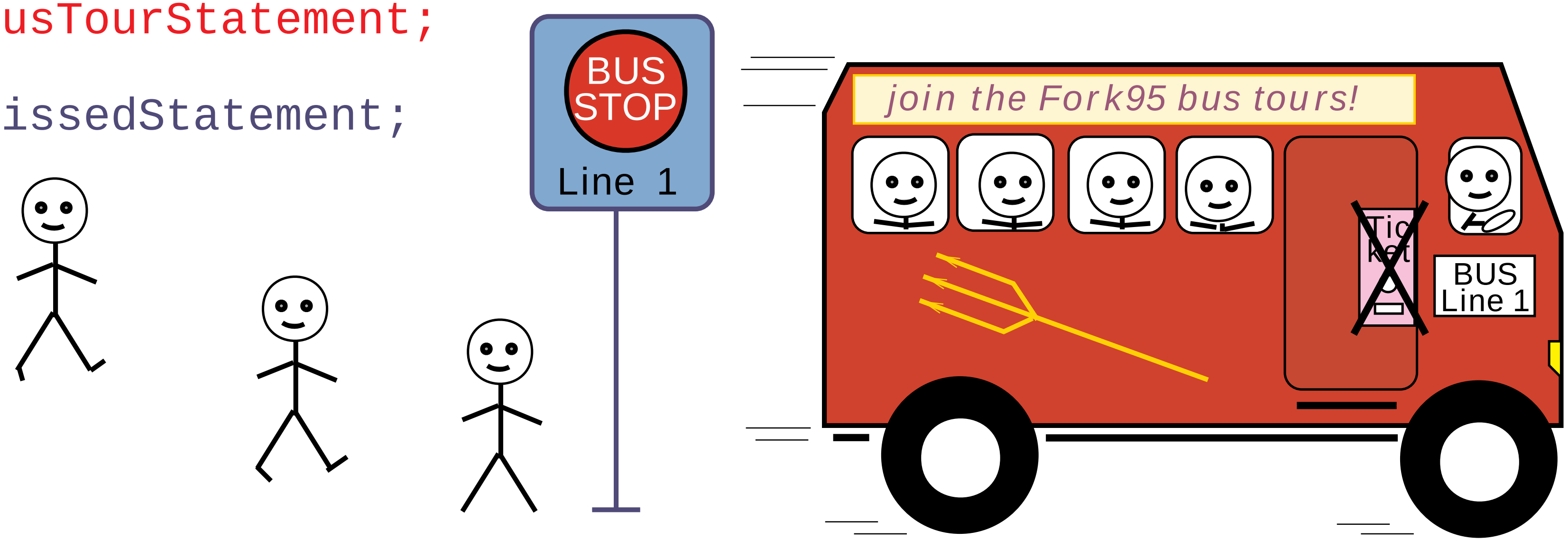
- ticket number is 0? -> driver!



- if not `stayInsideCond` spring off and continue with else part

The join() statement: The excursion bus analogy

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



Bus waiting: - get a ticket and enter

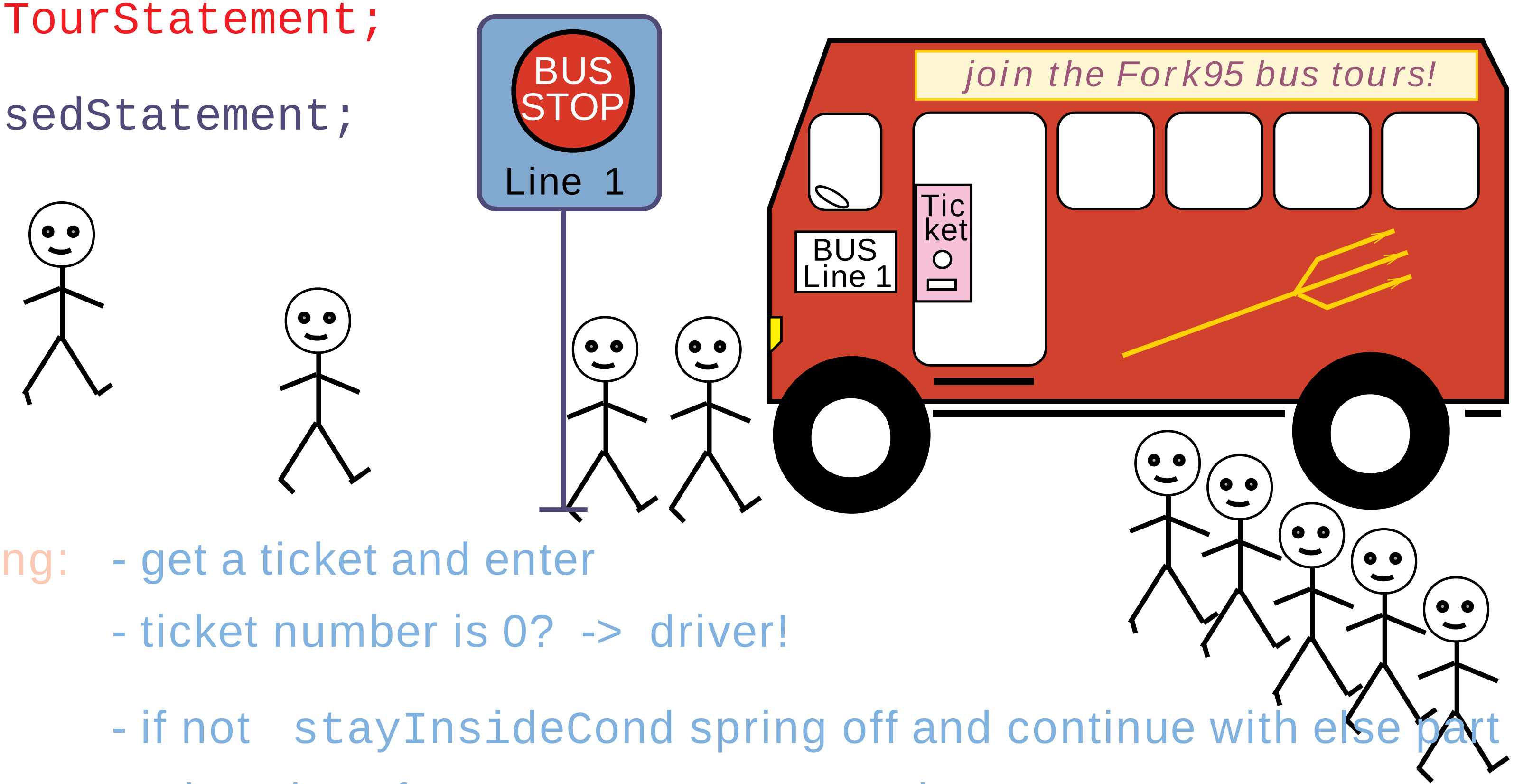
- ticket number is 0? -> driver!

- if not `stayInsideCond` spring off and continue with else part

- otherwise: form a group, execute `busTourStatement` synchronously

The join() statement: The excursion bus analogy

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



Bus waiting: - get a ticket and enter

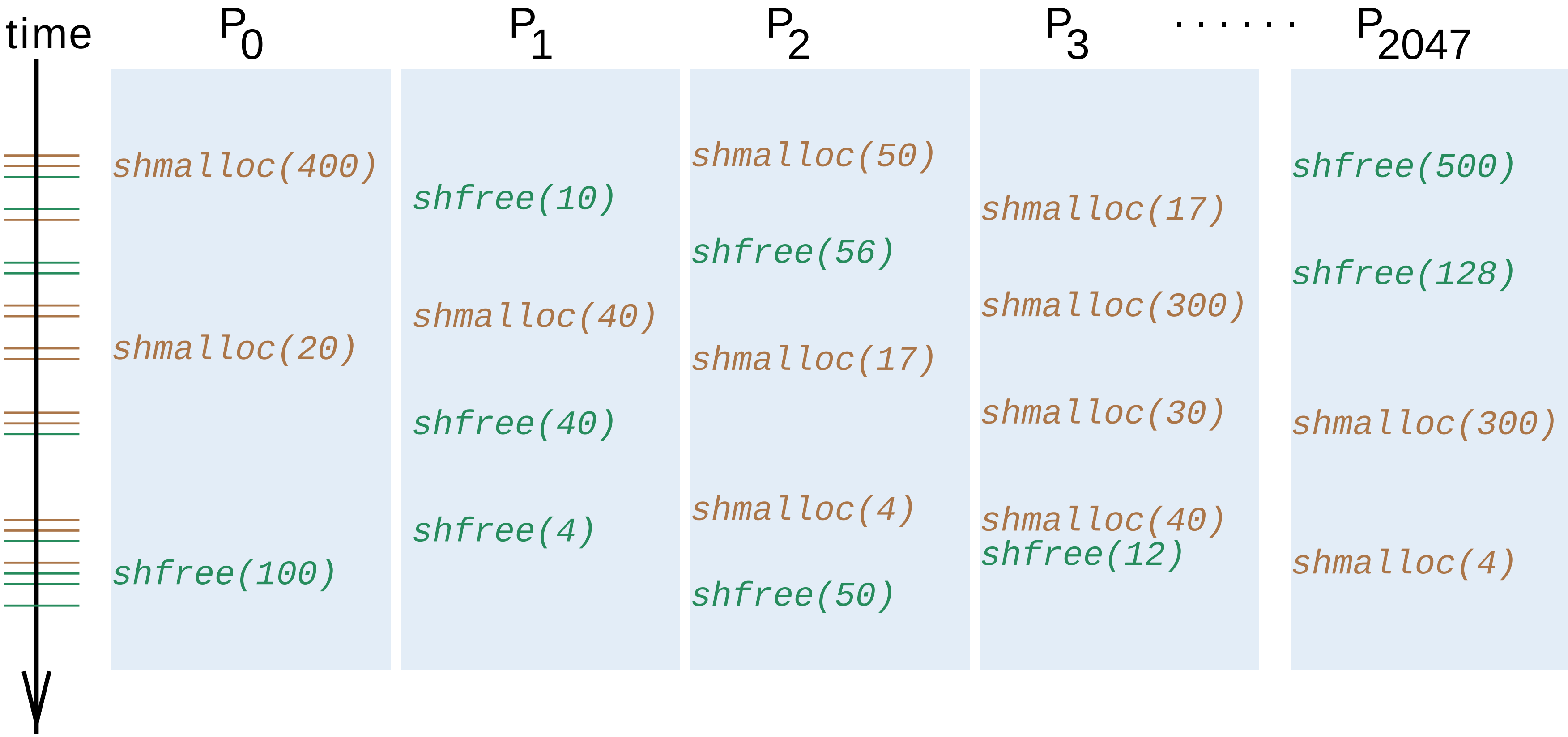
- ticket number is 0? -> driver!

- if not `stayInsideCond` spring off and continue with else part

- otherwise: form a group, execute `busTourStatement`

- at return: leave the bus, re-open ticket automaton and continue with next activity

Example: parallel shared heap memory allocation



Idea: - use a synchronous parallel algorithm for shared heap administration

- collect multiple queries to shmalloc() / shfree() with join() and process them as a whole in parallel!

Question: Does this really pay off in practice?

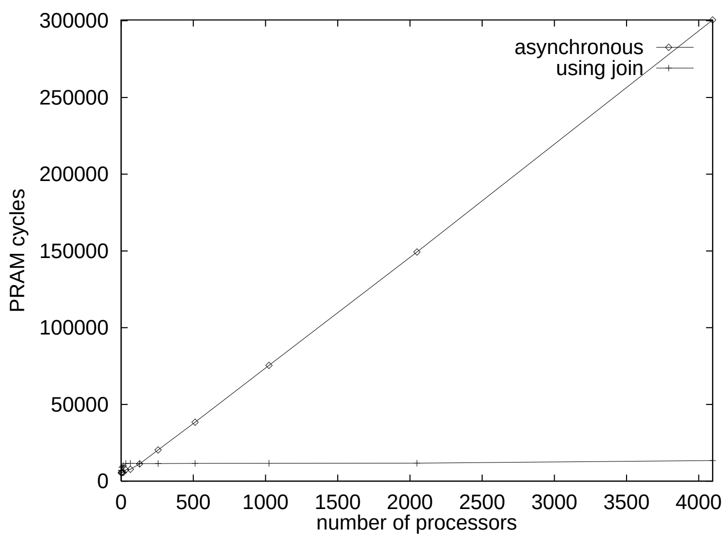
EXPERIMENT

Simple block–oriented parallel shared heap memory allocator

First variant: sequential critical section, using a simple lock

Second variant: parallel critical section, using `join`

p	asynchronous		using `join`	
1	5390 cc	(21 ms)	6608 cc	(25 ms)
2	5390 cc	(21 ms)	7076 cc	(27 ms)
4	5420 cc	(21 ms)	8764 cc	(34 ms)
8	5666 cc	(22 ms)	9522 cc	(37 ms)
16	5698 cc	(22 ms)	10034 cc	(39 ms)
32	7368 cc	(28 ms)	11538 cc	(45 ms)
64	7712 cc	(30 ms)	11678 cc	(45 ms)
128	11216 cc	(43 ms)	11462 cc	(44 ms)
256	20332 cc	(79 ms)	11432 cc	(44 ms)
512	38406 cc	(150 ms)	11556 cc	(45 ms)
1024	75410 cc	(294 ms)	11636 cc	(45 ms)
2048	149300 cc	(583 ms)	11736 cc	(45 ms)
4096	300500 cc	(1173 ms)	13380 cc	(52 ms)



The join construct for synchronous parallel critical sections

- semantics: see excursion bus analogy

- flexible way to switch from asynchronous to
  synchronous mode of execution

- allows to embed existing synchronous Fork95 routines
  into large parallel software packages

- allows to run different hierarchies of relaxed synchronicity concurrently
  (group hierarchy tree becomes a forest)

- use for synchronous parallel critical sections:
  - pays off for high access rates
    (e.g., due to large number of processors, bursts of accesses)
  - requires a synchronous parallel (PRAM) algorithm

- examples for use: shared heap memory allocator, parallel block merging,
  synchronous parallel output (e.g., N-Queens boards)

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

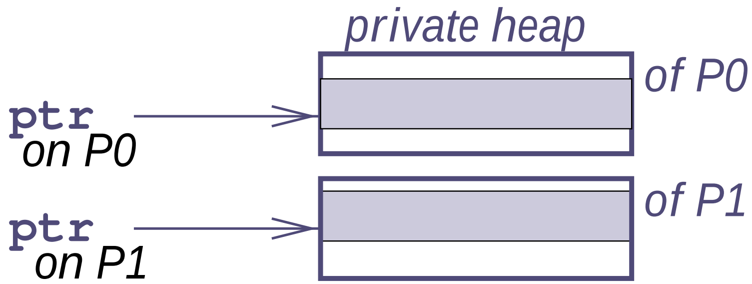
ForkLight language design and implementation

NestStep language concepts

Heaps for dynamic memory allocation

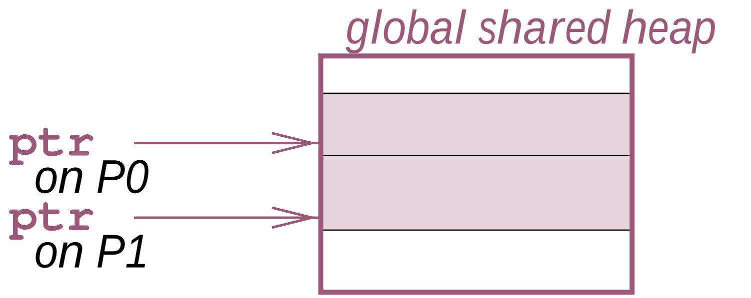
Private heap:

```
void *malloc( unsigned int k );
void free( void *ptr );
```



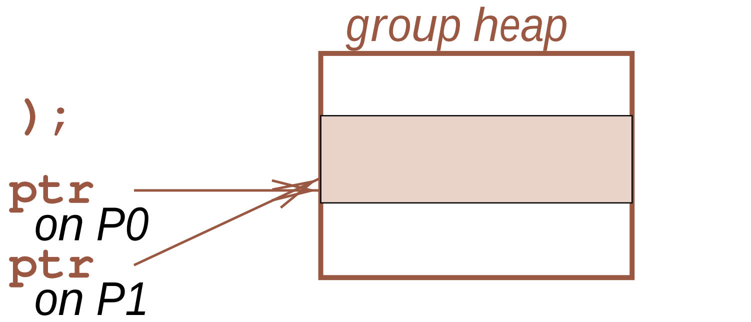
Global, shared heap

```
void *shmalloc( unsigned int k );
void shfree( void *ptr );
```



Group heap

```
sync void *shalloc( unsigned int k );
sync void shallfree();
```



TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

Statically scheduled parallel loop

(applicable in synchronous or asynchronous regions)

```
int i;
for ( i=$$; i<N; i+=# )
     statement( i );
```

Example: N=10, #=4

P0	P1	P2	P3
i=0	i=1	i=2	i=3
i=4	i=5	i=6	i=7
i=8	i=9	---	---

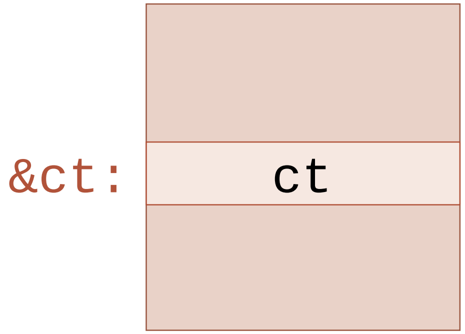
predefined macros in <fork.h>

```
int i;
forall( i, 0, N, # )
     statement( i );
```

Dynamically scheduled parallel loop

(useful in asynchronous regions with varying execution time of iterations)

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



on SB-PRAM: low overhead

predefined macros in <fork.h>

```
int i;
sh int ct;
......
FORALL( i, &ct, 0, N, 1 )
         statement( i );
```

TALK OUTLINE

PRAM model

SB-PRAM

Fork language

- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical sections; the join construct
- heaps
- programming parallel loops
- applicability, projects, history
- related work

Fork compilation issues

ForkLight language design and implementation

NestStep language concepts

Applicability of Fork

Teaching Parallel Programming

- ideal as a first parallel language
- play with parallelism
- simulator

Parallel Algorithm Design

- to test (new) PRAM algorithms for practical relevance
- supports many (all?) parallel algorithmic paradigms

Parallel Software Engineering with Fork

- common programming language as basis (upgrade to C++ would be desirable)
- existing sequential C sources can be reused nearly without any syntactical change
- allows to build large parallel software packages
  -> PAD library (synchronous parallel algorithms and data structures, by J. Träff)
  -> APPEND library (asynchronous parallel data structures)
  -> MPI core implementation
  -> Skeleton functions (also nestable)
- major applications already implemented: FView (by J. Keller), N-body simulation

Visit the Fork WWW homepage!

Fork compiler package (version 2.0 from Nov. 1999)
with all sources, example programs and documentation

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

Also: SB-PRAM simulator and system tools

System requirements: SunOS / Solaris or HP-UX (not Linux, sorry)

The slides of this presentation are available at

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