Practical PRAM Programming with Fork

Research talk

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

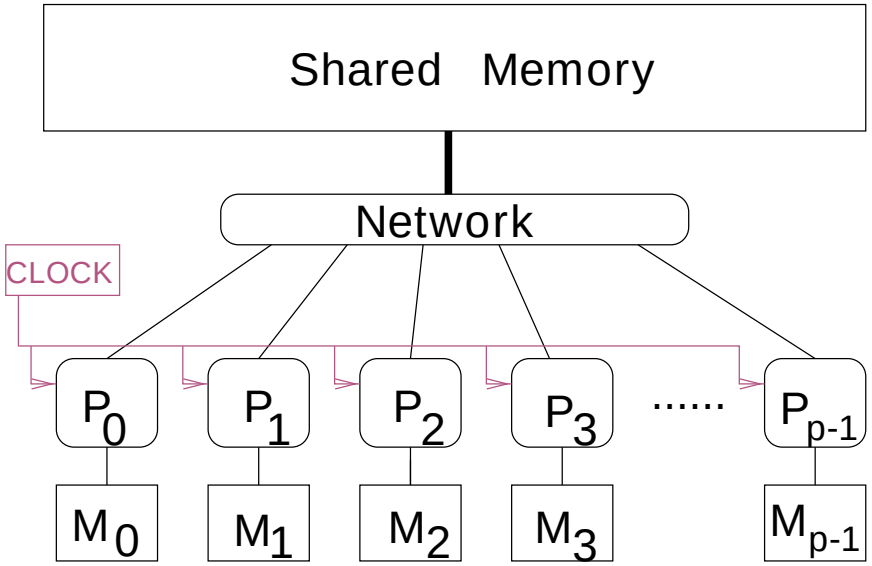
Fork compilation issues

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)

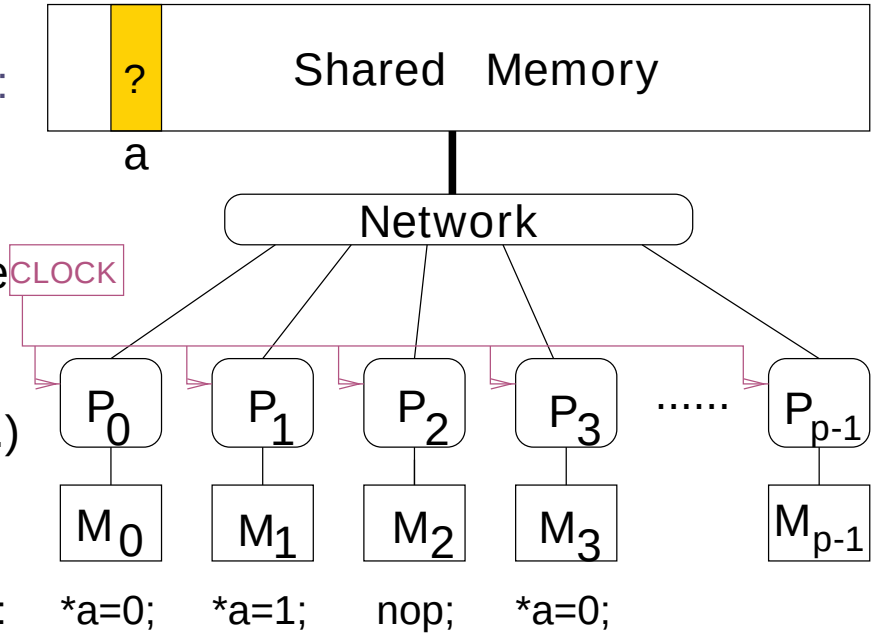


PRAM model

- 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.)
  .....

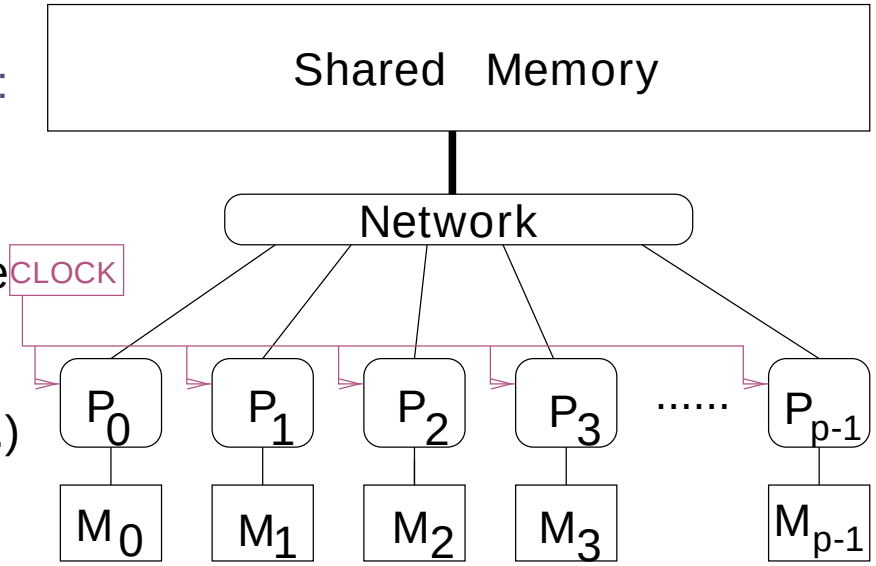


PRAM model

- 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.)
  .....



- 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

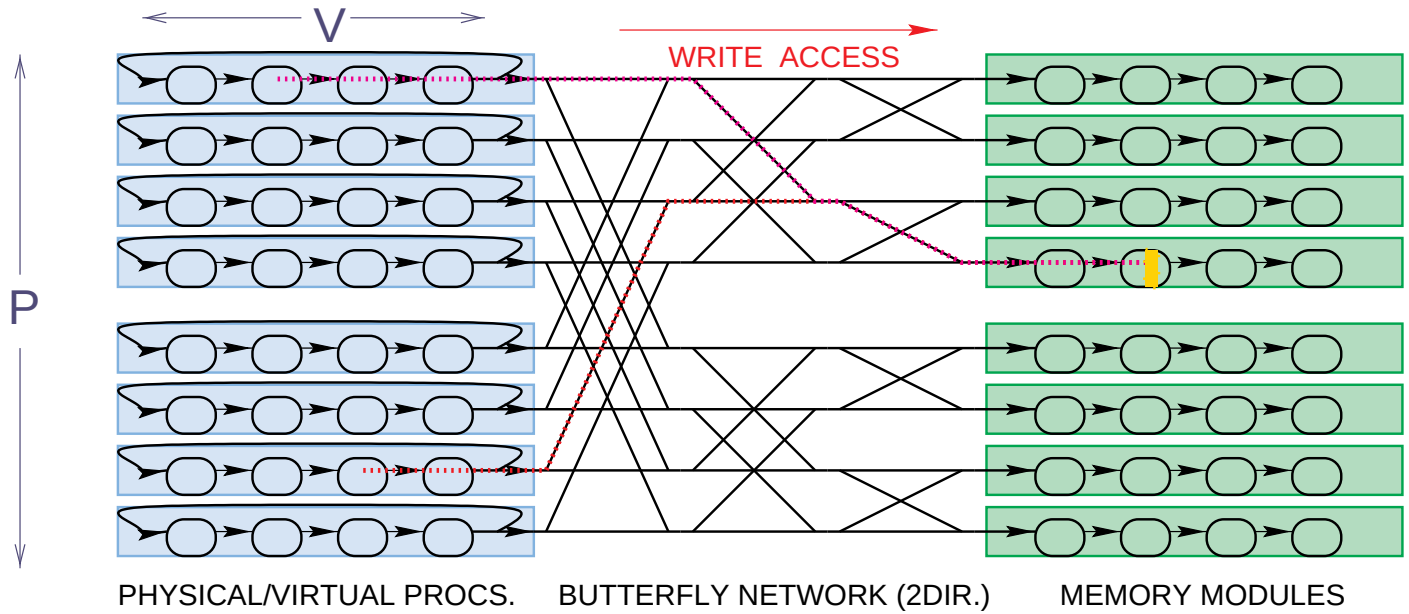
ForkLight language design and implementation

NestStep language concepts

SB-PRAM: A realization of the PRAM model in hardware

From the architectural point of view:

- pipelined butterfly network, switches have a simple ALU and memory
- write conflicts are resolved by combining on the network
- P physical processors (special design), P log P switch processors
- each physical processor simulates V = c log P virtual processors
  in a round-robbin way (instruction by instruction, special design)
- scalable
- university prototypes: (1) P=16, V=32 (1998) (2) P=64, V=32 (2000)
- performance: (1991 design:) 250 kFLOPs (also memory bandwidth) per virt. proc.



SB-PRAM: A realization of the PRAM model in hardware

From the architectural point of view:

- pipelined butterfly network
- write conflicts are resolved by combining
- P physical processors
- each physical processor simulates V = c log P virtual processors
- scalable
- university prototypes: (1) P=16, V=32 (1998) (2) P=64, V=32 (2000)
- performance: (1991 design:) 250 kFLOPs (also memory bandwidth) per virt. proc.

From the programmer' s point of view: Combining/Priority CRCW PRAM

- MIMD, p = P*V processors, synchronous on assembler level
- shared memory, unit memory access time: 1 CPU cycle
  (delayed LOAD: + 1 cycle)
- processor:
  32 bit RISC-style, LOAD/STORE-architecture
  BASE register supports embedding of private memory subspaces
- powerful multiprefix operations: 1 CPU cycle + 1 delay cycle
  (MPADD, MPMAX, MPAND, MPOR)

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

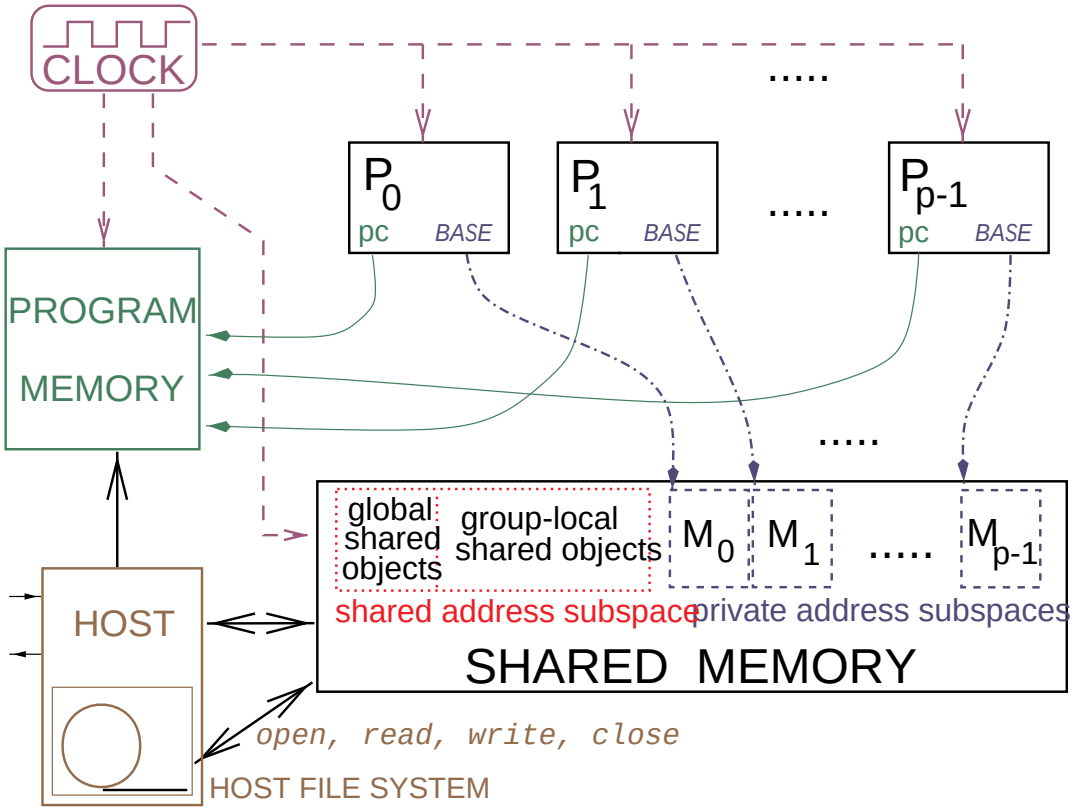
extension of C

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

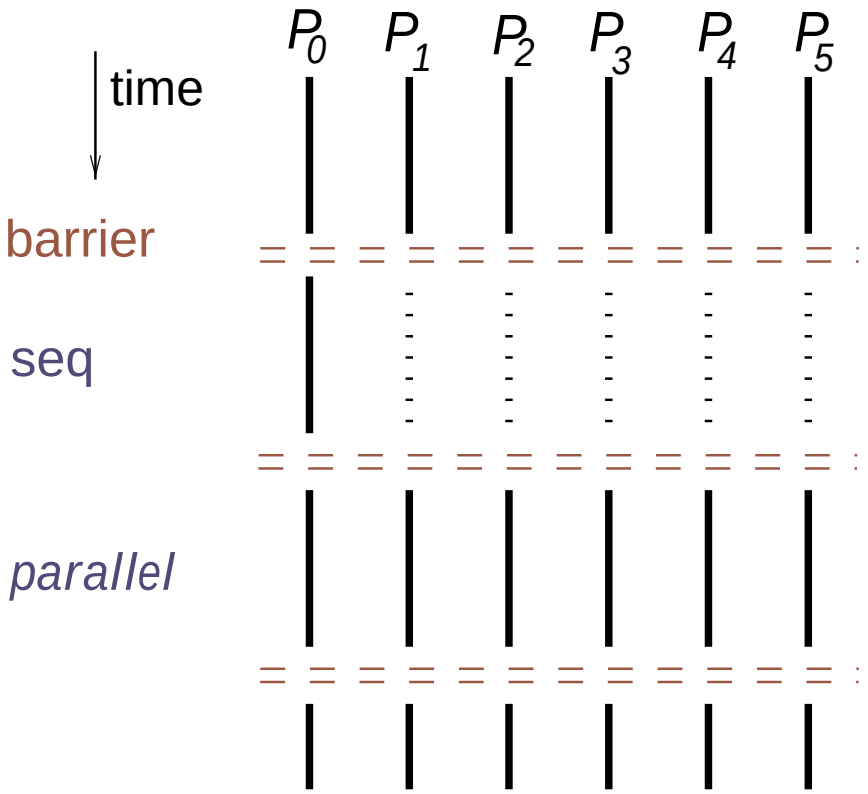


SPMD style of parallel execution

- 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
- sh relates to defining group of processors

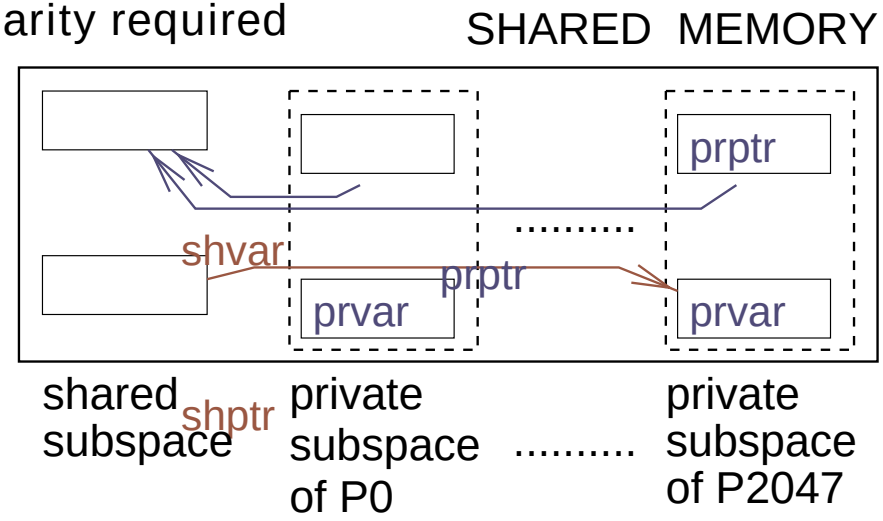
"sharity"

- special shared run-time constant variable: __STARTED_PROCS__
- special private run-time constant variable: __PROC_NR__

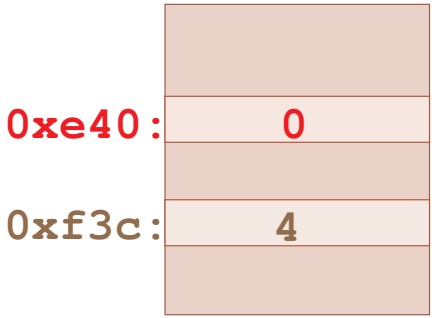
```
sh int npr = __STARTED_PROCS__;
pr int myreverse = npr - __PROC_NR__ - 1;
pprintf("Hello world from P%d\n", __PROC_NR__ );
```

- pointers: no specification of pointee's sharity required

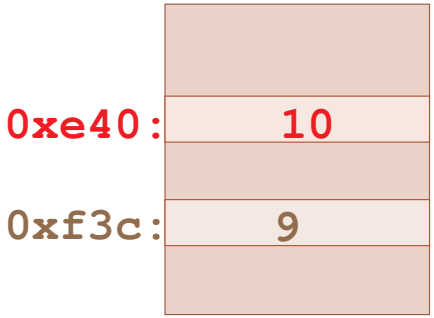
```
pr int prvar, *prptr;
sh int shvar, *shptr;

prptr = &shvar;

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



mpadd Example



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



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

Example: User-defined consecutive numbering of processors

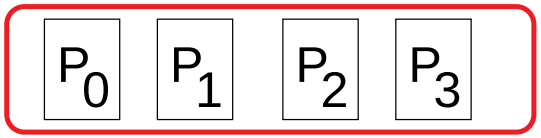
```
sh int counter = 0;
pr int me = mpadd( &counter, 1 );
```

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

`mpmax` may be used as atomic test&setoperator.

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

Group concept



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)

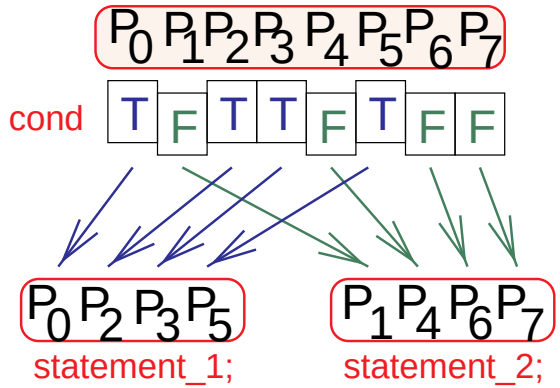
Scope of synchronous execution:

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

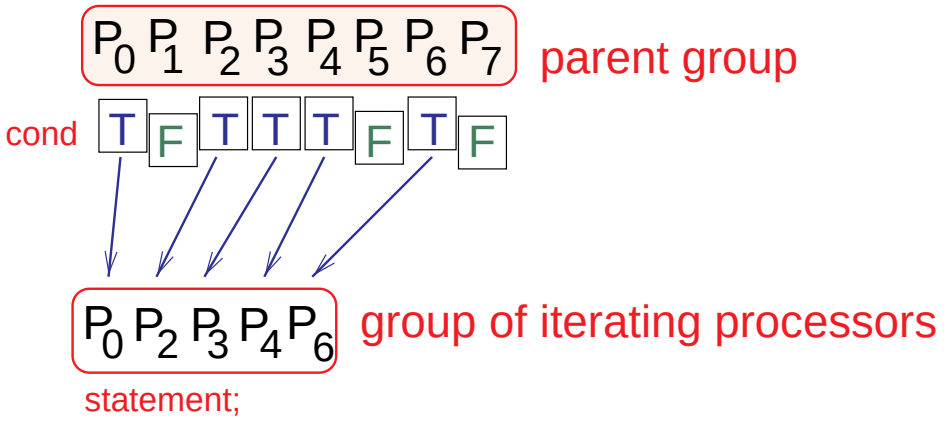
shared stack and heap of parent group is divided among child groups

(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.

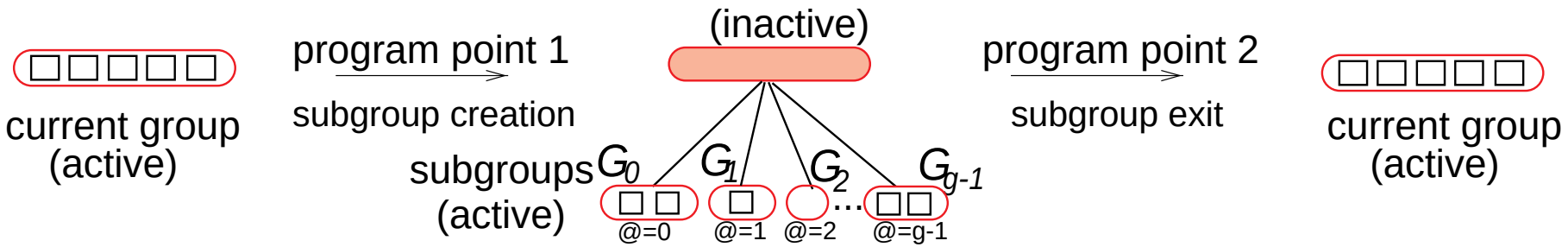
- As soon as cond becomes FALSE, the processors wait at the end of statement for the others (implicit barrier).

Explicit group splitting: The fork() statement

```
fork ( g; @ = fn($$); $=$$)

    statement;
```

program point 1

program point 2

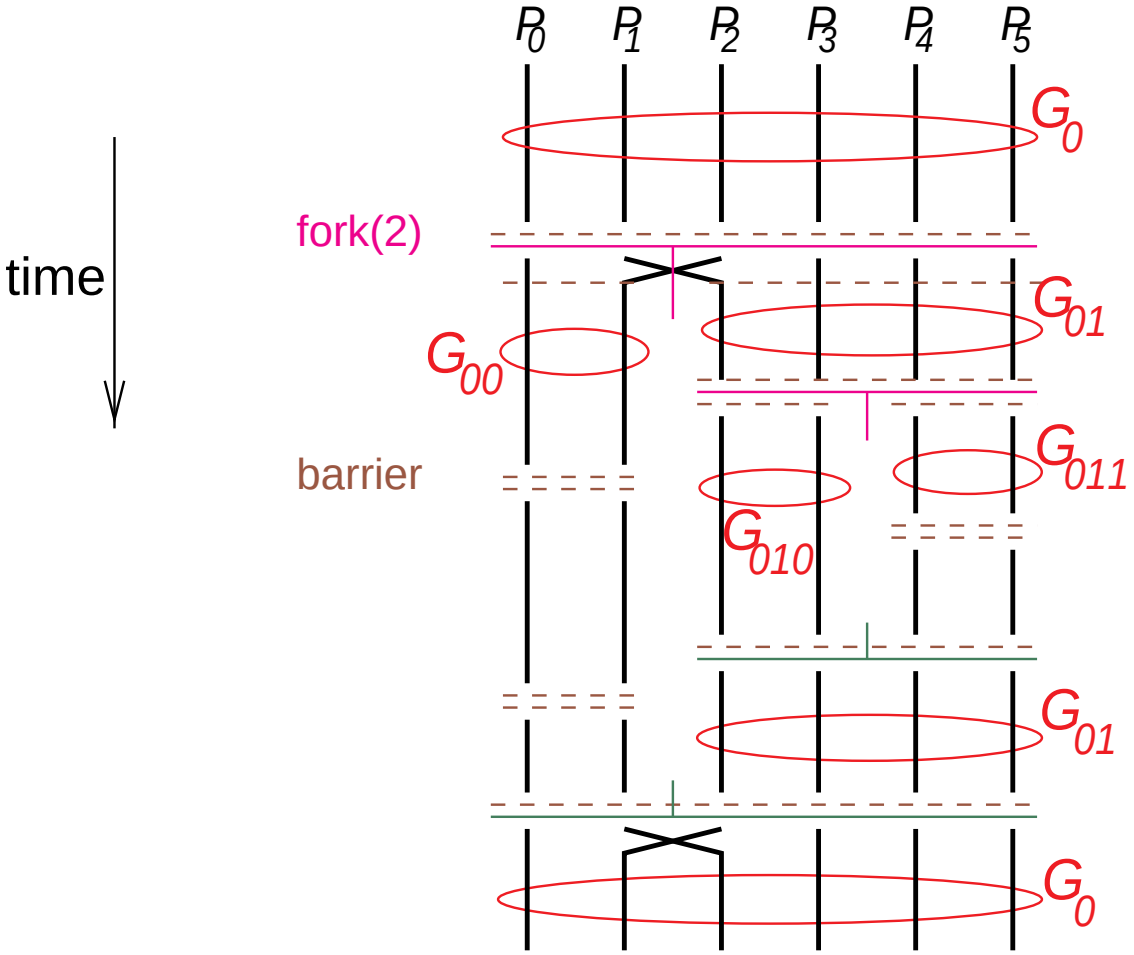


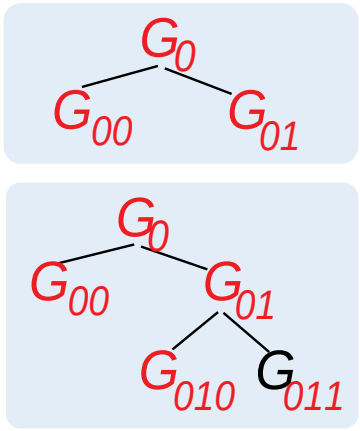
first parameter: current group is split into g subgroups
second parameter: assignment to @, decides about subgroup to join
third parameter (optional): possible renumbering of $ within the subgroups

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

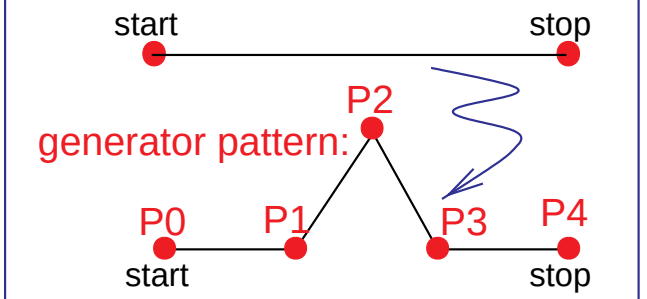
ForkLight language design and implementation

NestStep language concepts

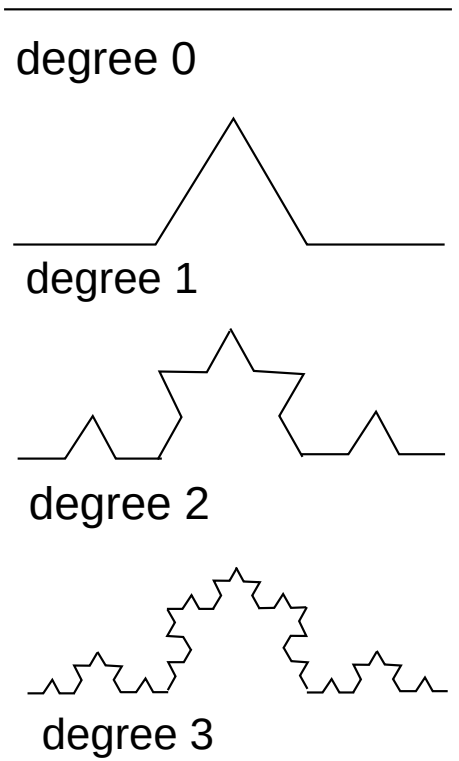
Example: Drawing Koch curves

```
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 );
}
```

recursive replacement strategy:



initiator pattern:



TALK OUTLINE

PRAM model

SB-PRAM

Fork language

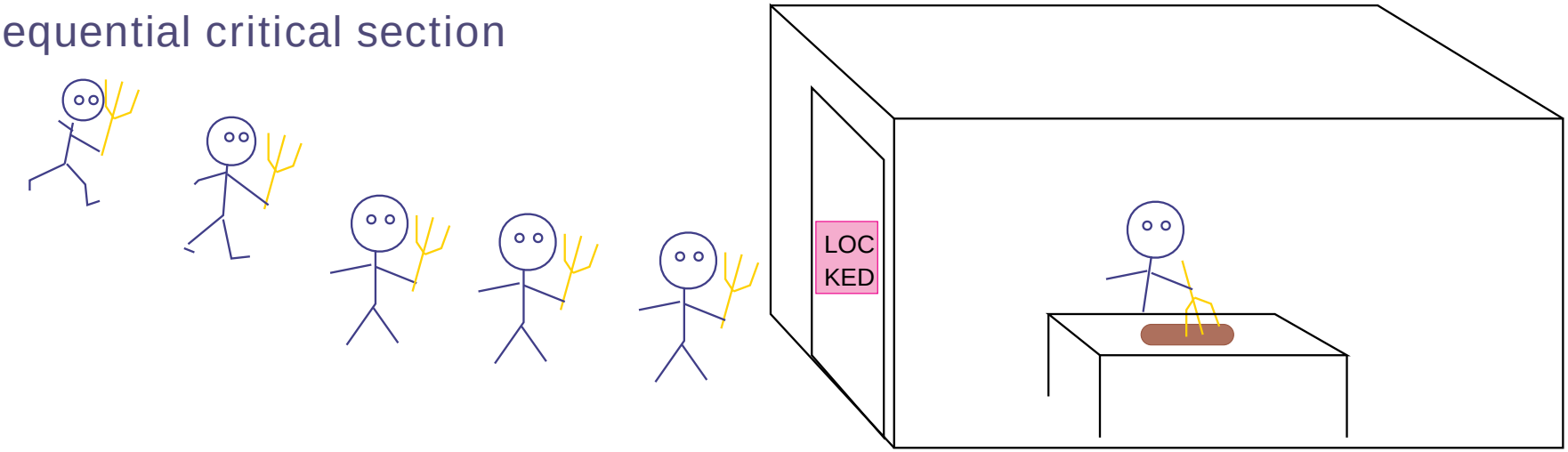
- programming model, SPMD execution
- declaration of sharity, first steps
- expressions (multiprefix operators)
- synchronicity declaration, group concept
- example (Koch curves), graphical trace file visualization
- asynchronous computations: critical sections and locks
- synchronous parallel critical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

Fork compilation issues

ForkLight language design and implementation

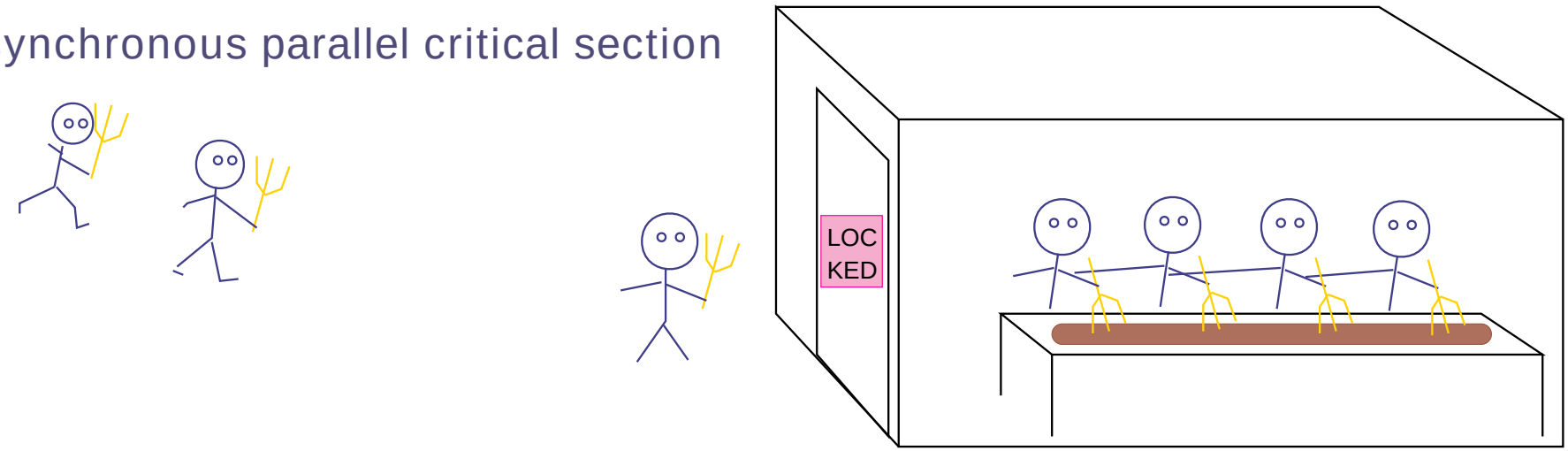
NestStep language concepts

sequential critical section



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

synchronous parallel critical section

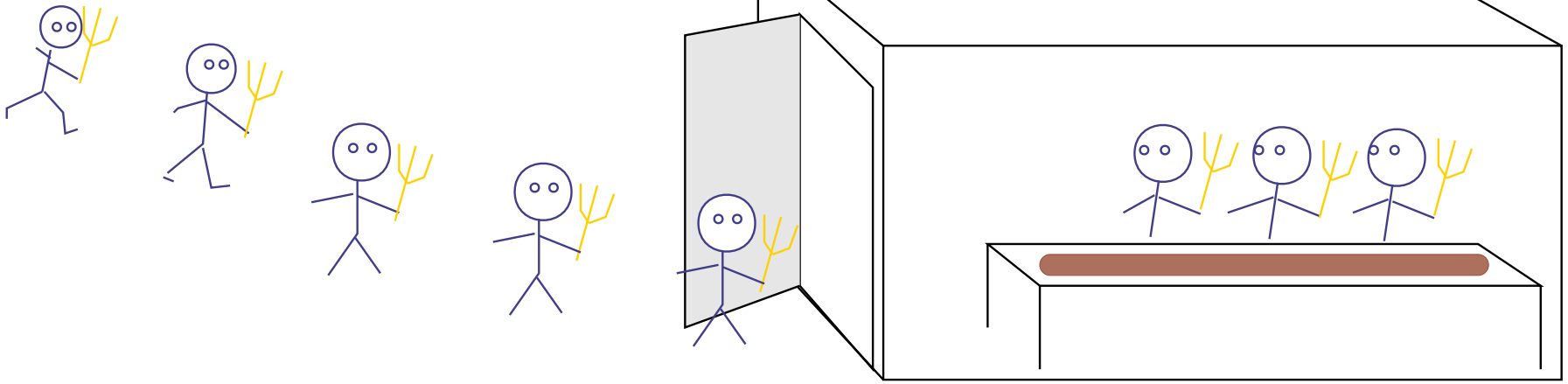


-> simultaneous entry of more than one processor

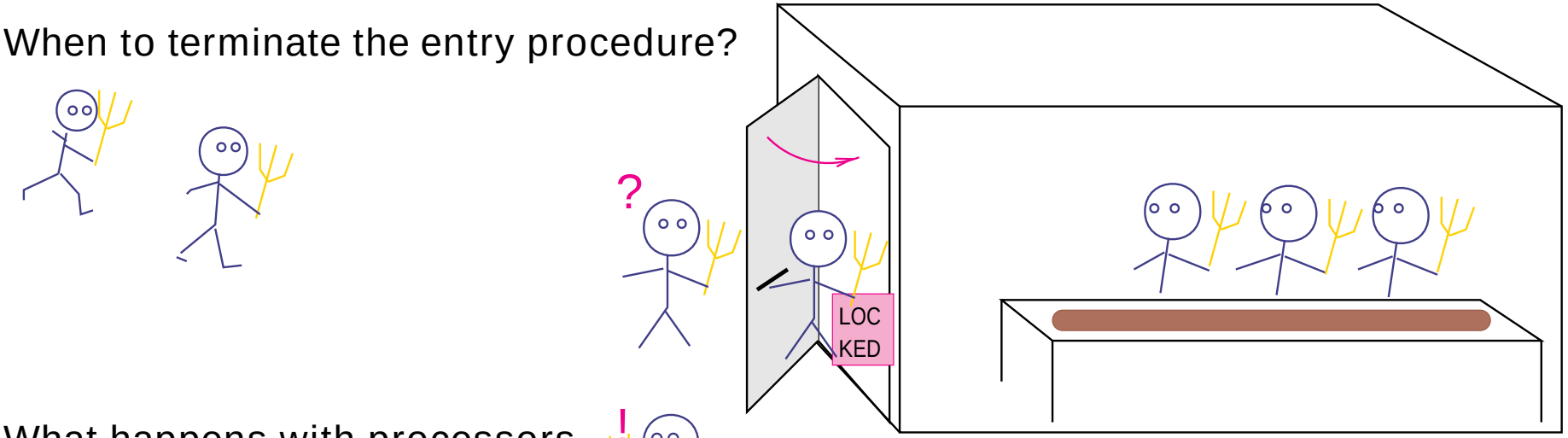
-> deterministic parallel access by executing a synchronous parallel algorithm

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

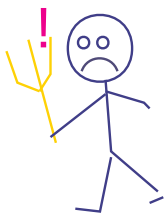
Entry conditions?

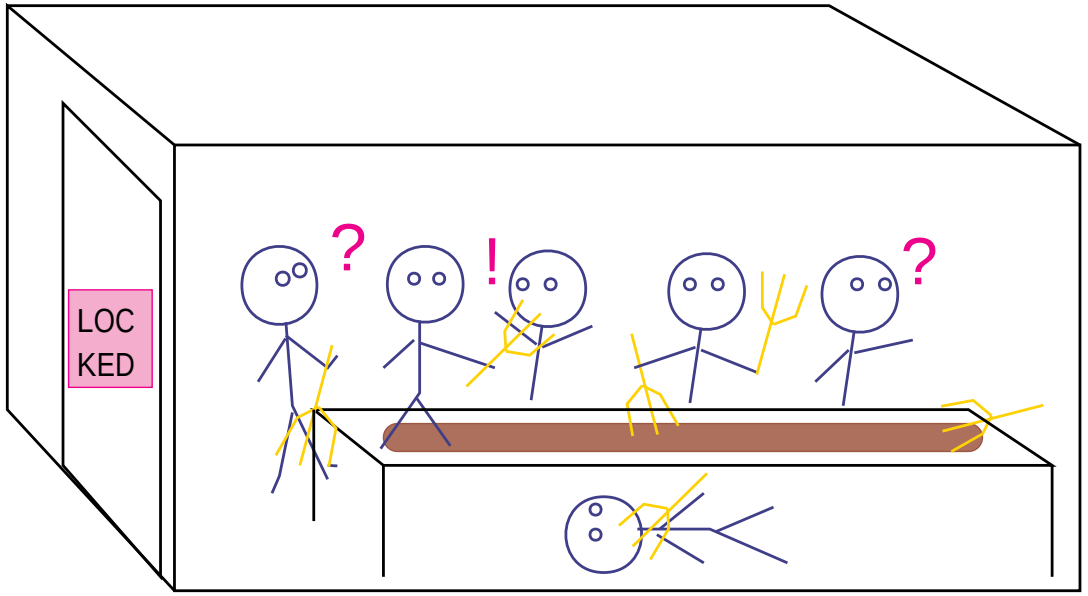


When to terminate the entry procedure?



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

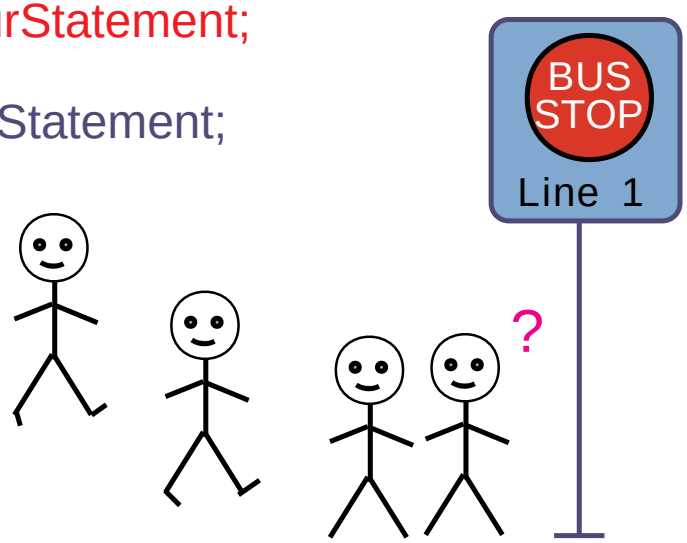
- deterministic parallel execution of the critical section
by applying a suitable synchronous (PRAM) algorithm

- 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;
```

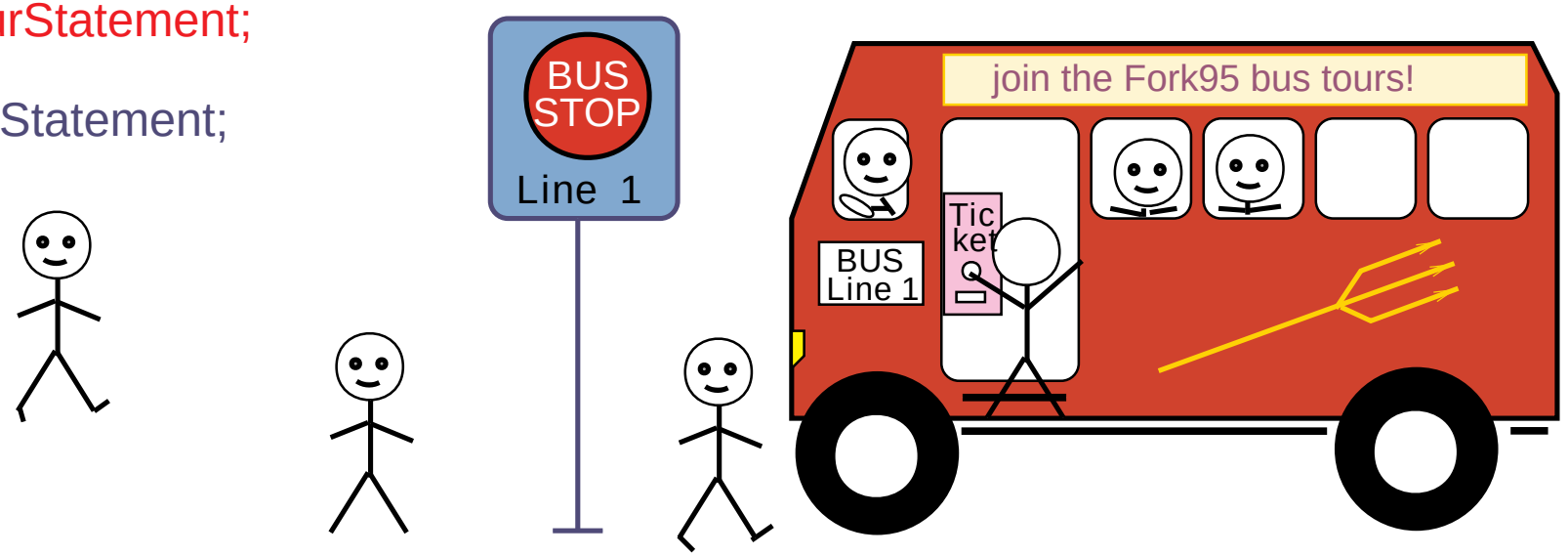


Bus gone? - execute else part: missedStatement;

- 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;
```

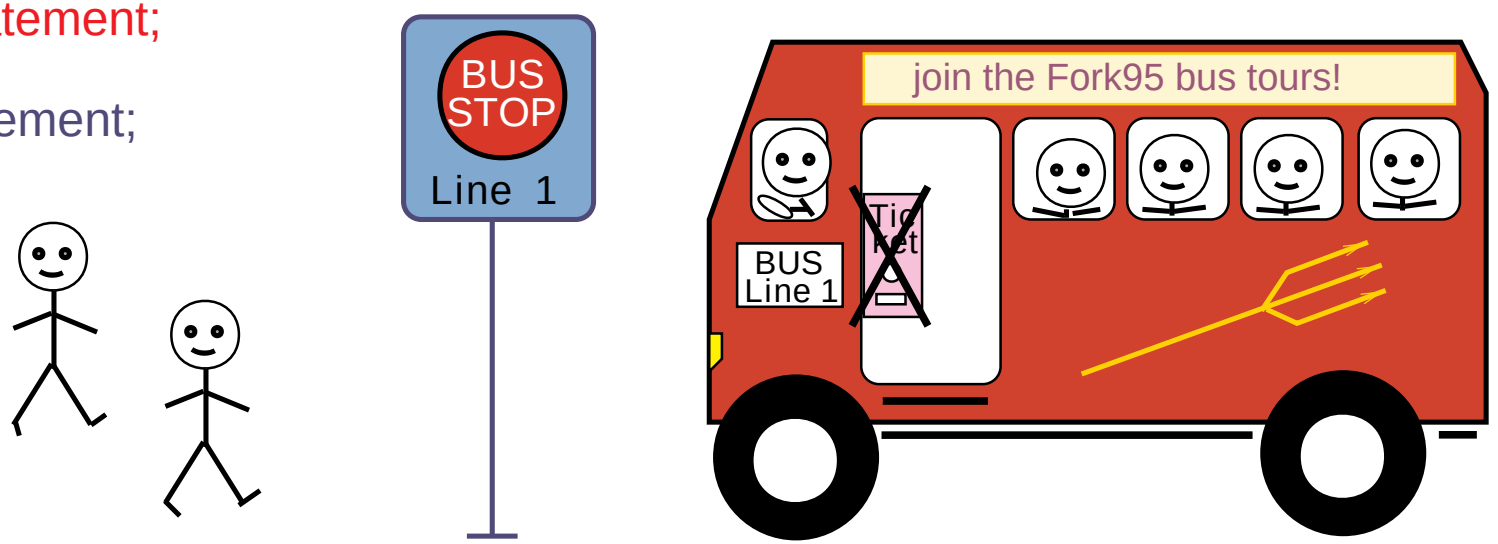


Bus waiting: - get a ticket and enter

- 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;
```



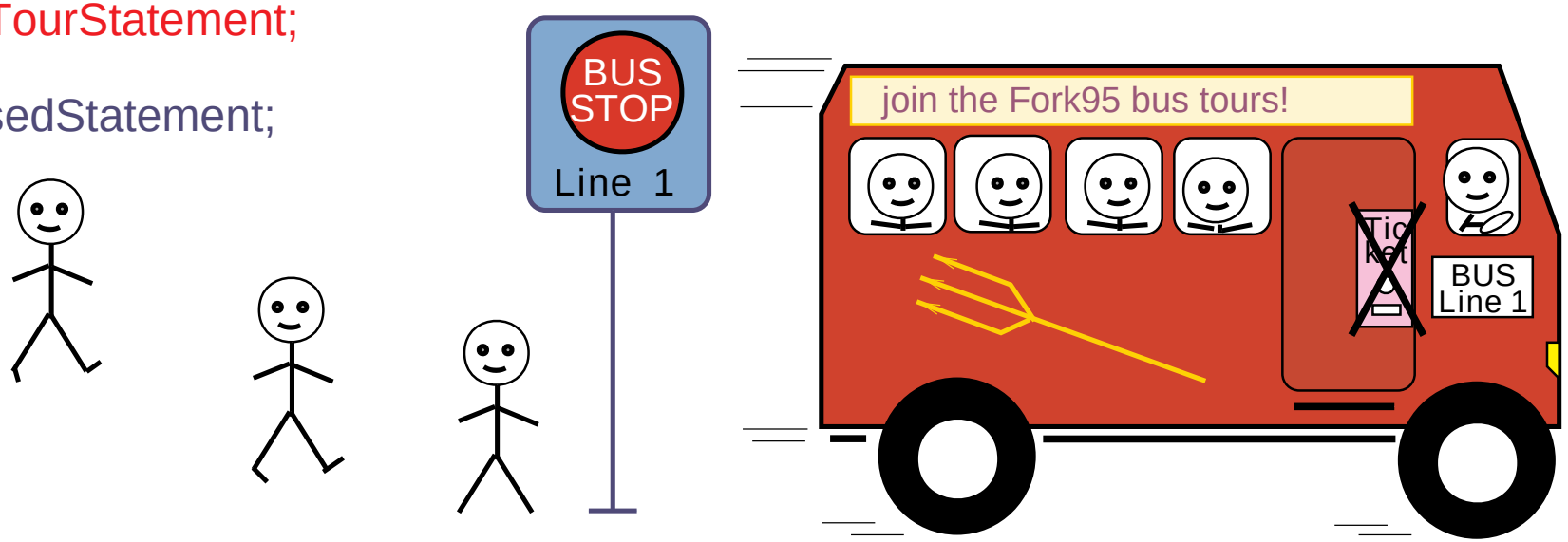
Bus waiting: - get a ticket and enter

- 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

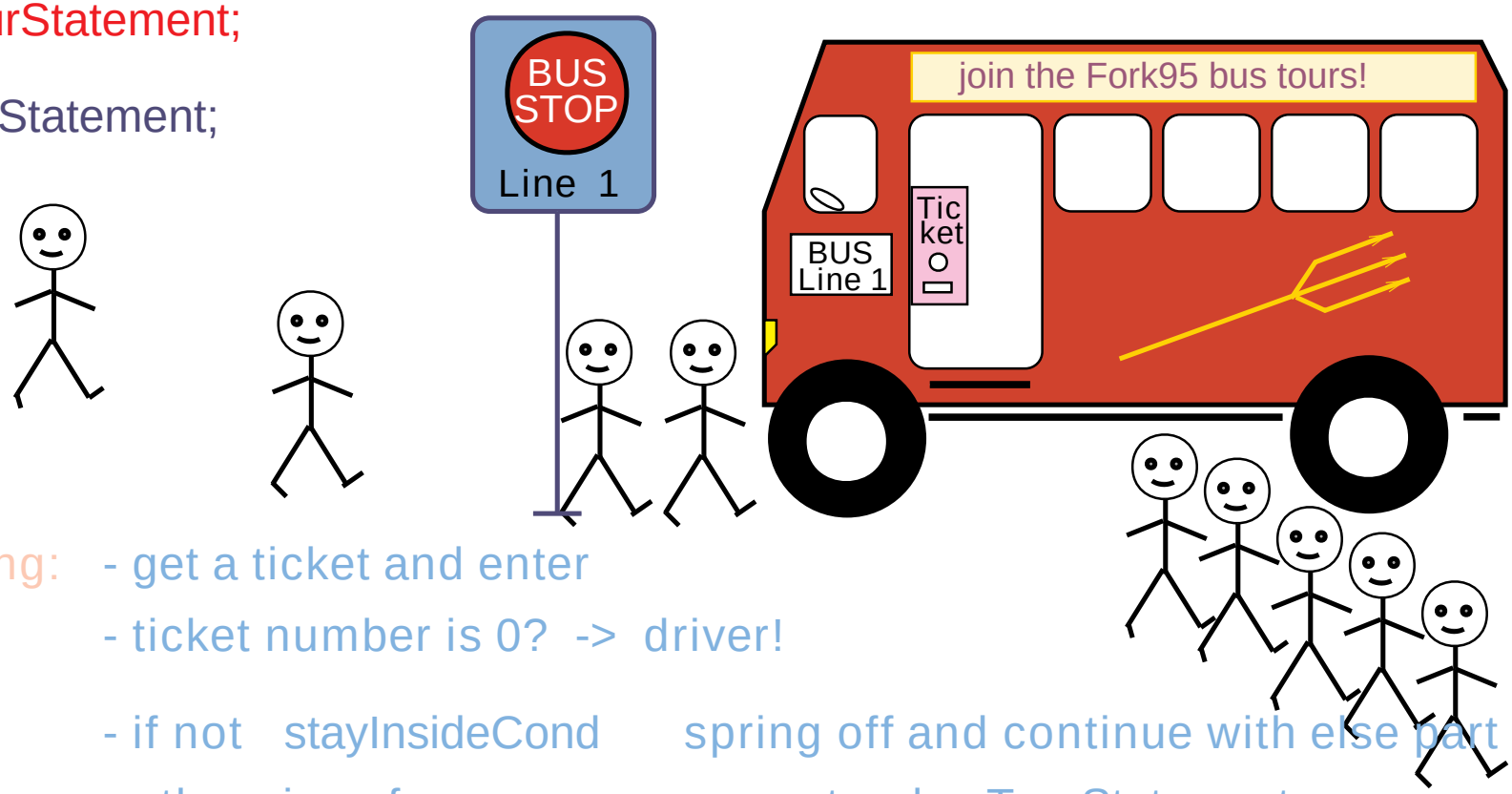
- ticket number is 0? -> driver!

- if not stayInsideCond spring off and continue with else part

- otherwise: form a group, execute busTourStatement synchronously

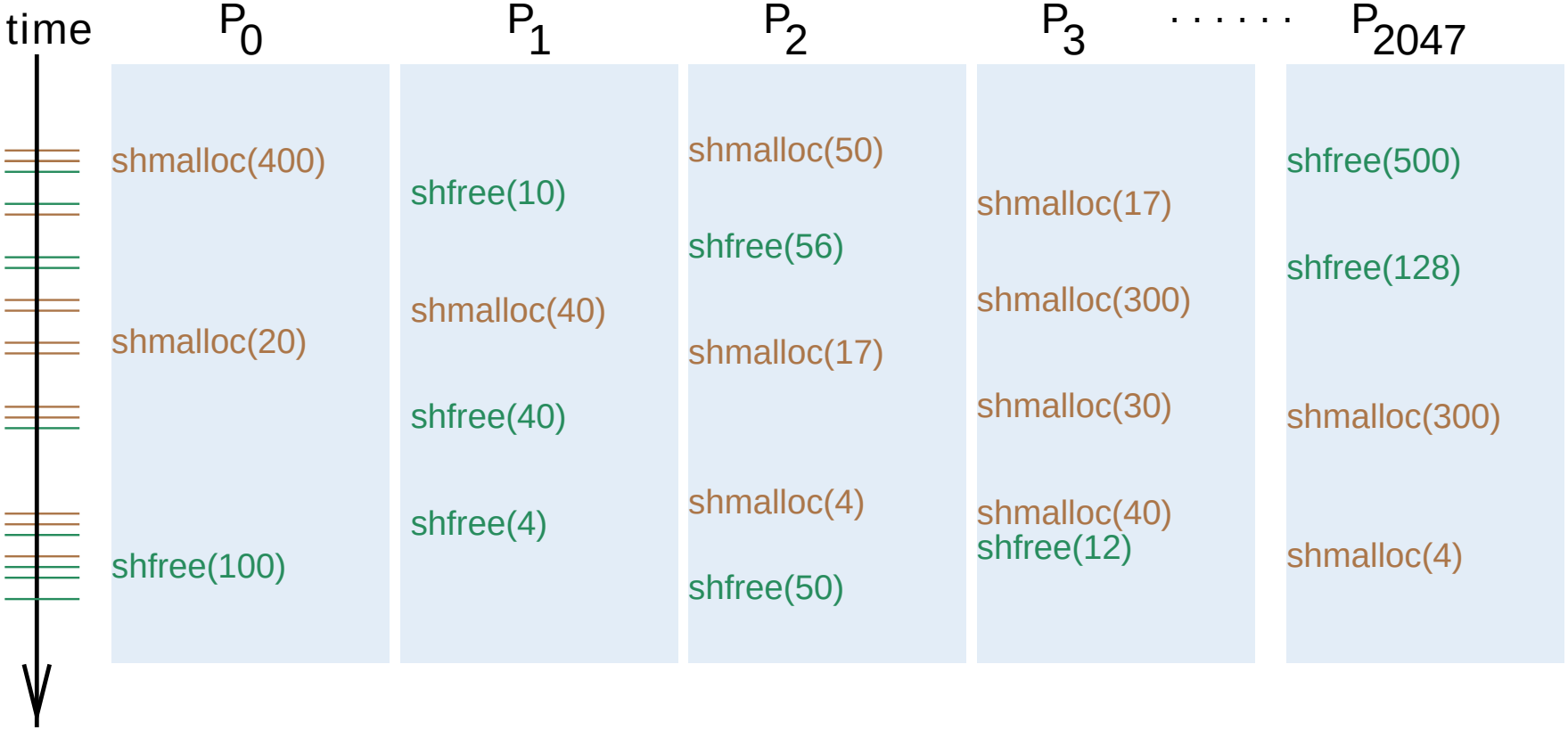
The join() statement: The excursion bus analogy

```
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?

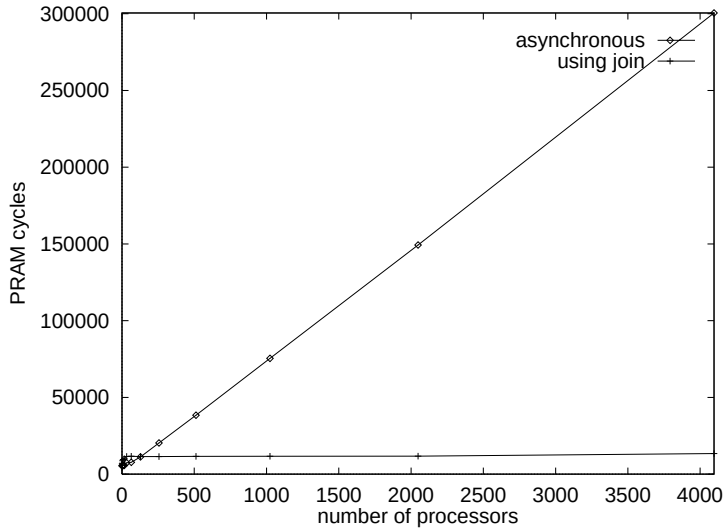
EXPERIMENT

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

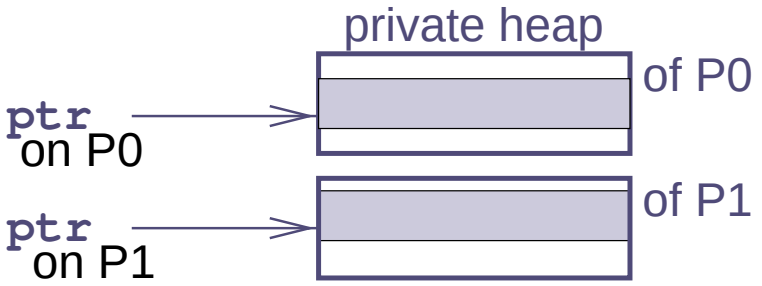
ForkLight language design and implementation

NestStep language concepts

Heaps for dynamic memory allocation

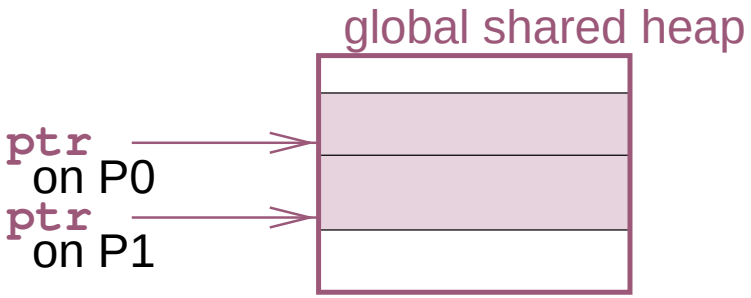
Private heap:

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



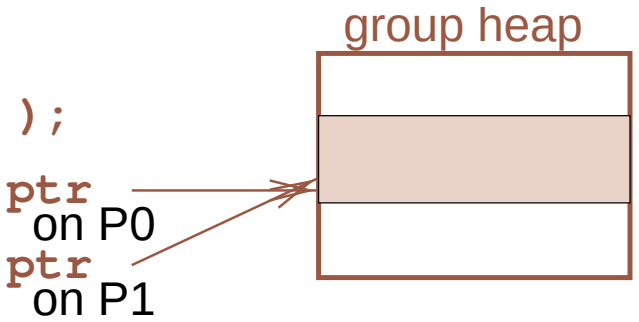
Global, shared heap

```
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

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

Projects based on Fork

finished:

- PAD library of PRAM data structures and algorithms (J. Träff, 1995-99)
- HSM hierarchical skeleton model (M. Marr, Ph.D. thesis 1998, U. Edinburgh)
- MPI subset implementation in Fork (student project, U. Trier, 1999)
- APPEND library of asynchronous concurrent data structures
  (student project / C. Kessler, U. Trier, 1999)

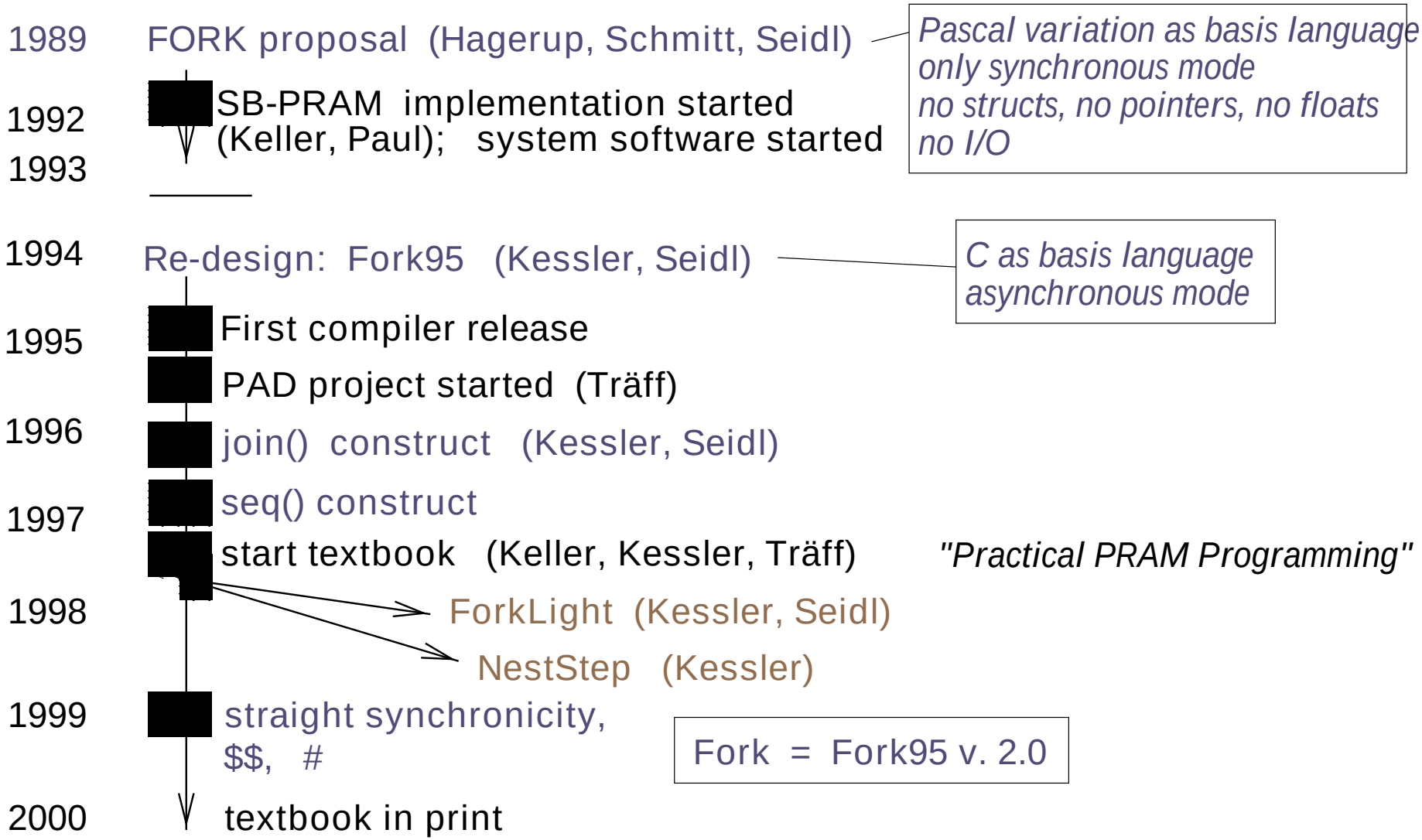
ongoing (status: end of 1999):

- FView graphical visualization tool (J. Keller, FU Hagen)
- improved implementation of shmalloc() / shfree() (student project, U. Trier)
- source-to-source compiler for Fork95 (A. Perez Jorge, U. La Laguna)
- basic linear algebra routines (M.S. project, T. Ashby, U. Edinburgh)

proposals:

- compilation of Fork to asynchronous shared memory machines
- compilation of Fork to distributed shared memory platforms
- optimization of compiler-generated SB-PRAM assembler code
- porting pramsim to LINUX
- parallelization of pramsim for small SMP's
- skeleton layer for Fork: design and implementation .....

Fork: History and future



TALK OUTLINE

- 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

PRAM model
SB-PRAM
Fork language

Fork compilation issues
ForkLight language design and implementation
NestStep language concepts

Classification of some parallel programming languages

	shared memory model			message passing
	SIMD / dataparallel		MIMD	(MIMD)
exactly synchronous execution supported	NESL, V, C , Dataparallel C, Fortran 90, Vector-C, APL, CM-Fortran	HPF, Fortran-D, Vienna-Fortran, MPP-Fortran	FORK, Fork ll, pm2, Modula-2*	—
no exactly synchronous execution	—	—	ParC, Cilk, Sequent-C, Tera-C, P4, LINDA, PCP,pthreads Open-MP	PVM MPI, MPI-2

Why is there no virtual processor emulation in Fork95?

- SB-PRAM processor offers fast context switch (1 cc) for up to 32 contexts

BUT:

- private memory subspaces become inacceptably small (~3K)
- synchronous regions: must preserve cross-processor data dependencies

  semantics of Fork would require a lock for each shared memory cell:

  Example:

```
sync foo(...) { ...
   if ($<10)  a = x;
   else       y = a;
}
```

  all processors of the else part must read the same value of a

  -> access to a must be guarded by a lock
  -> sequentialize groups wrt. accessing a
  -> overhead: locking 16 cc, unlocking 32 cc
  -> memory requirements doubled; more complicated address calculation

- synchronization routine becomes more expensive
- asynchronous regions: context switch after each write, before each read access

Why is there no virtual processor emulation in Fork95? (continued)

Our solution:

+ number of available processors can be inspected at low expense
+ library offers customized parallel iterators over common data structures
  (e.g. arrays, trees, DAGs, lists, ...)

```
pr int i;
for (i=$$; i<N; i+=#)    x[i] = y[i]*z[i];
```

or

```
forall (i,0,N,#)    x[i] = y[i]*z[i];
```

BTW:

+ fast context switch would be abandoned on future SB-PRAM successors

Fork COMPILATION ISSUES

Compilation for asynchronous shared memory machines

- requires static analysis of cross-processor data dependences
- compiler must preserve these
  by inserting barriers or bilateral synchronization

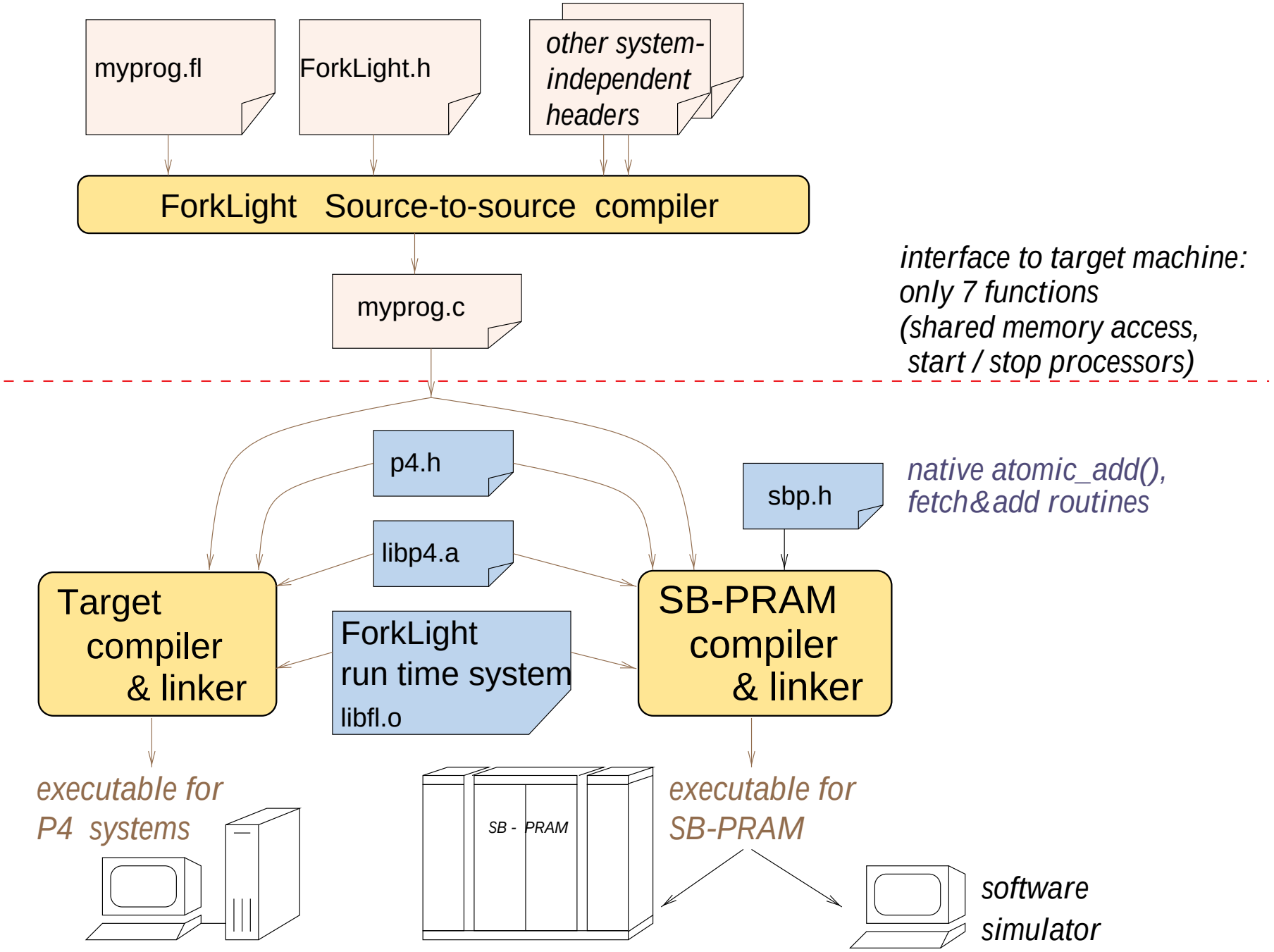
Compilation for distributed-memory architectures

generation of interprocessor communication statements

+ if control flow is synchronous (SIMD) or statically known:
  - by the compiler: if data dependences statically analyzable
  - by the compiler with run-time support:
    if data dependences are dynamically analyzable

+ if control flow is asynchronous:
  - by hardware/OS support or software Virtual Shared Memory system

ForkLight: Implementation



ForkLight

- Language design 1997-98 [Kessler / Seidl HPCN'99]
- C-based MIMD parallel programming language, syntax similar to Fork95
- MIMD, SPMD, hierarchical group concept
- programming model: Asynchronous PRAM
- control-synchronous mode of execution

  All processors belonging to the same active group
  are working within the same (extended) basic block.
- asynchronous mode of execution (as in Fork)
- portable prototype implementation based on P4 (also possible: OpenMP)
- related work: PCP (Parallel C Preprocessor) [Brooks, Gorda, Warren '92]
- compiler source code and example programs:

  http://www.informatik.uni-trier.de/~kessler/forklight.html

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

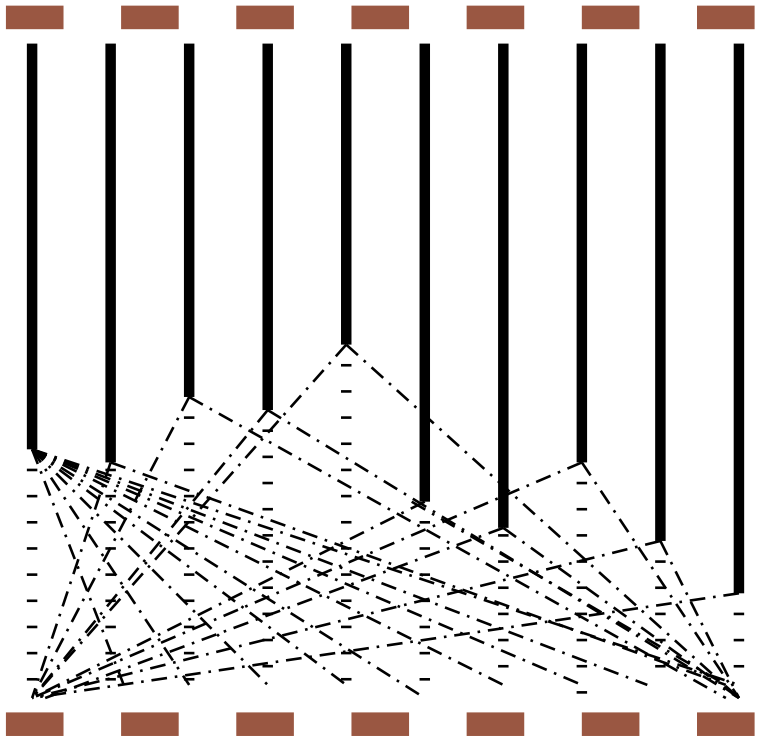
NestStep

- Idea: Fork-like language that relaxes the sequential memory consistency
- BSP model (bulk-synchronous parallelism)

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

NestStep: The step statement

```
sh<+> int sum;
step
    statement;
```



global barrier

local computation
using local copies of shared variables

communication phase

update local copies of shared variables

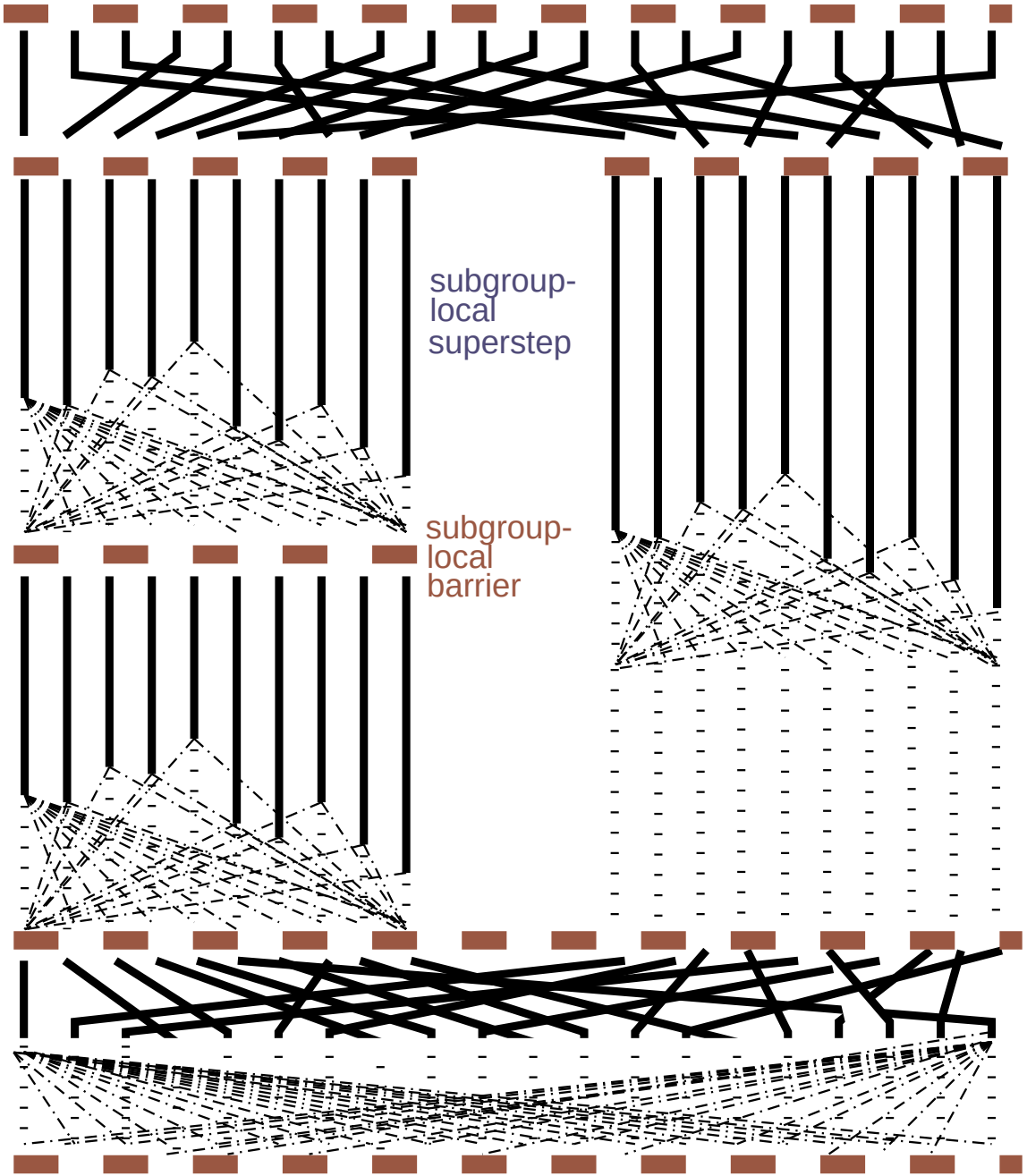
next barrier

NestStep

- Idea: Fork-like language that relaxes the sequential memory consistency
- BSP model (bulk-synchronous parallelism)
- extension of C / C++ / Java
- hierarchical group concept as in Fork
- distributed shared memory emulation
  on top of a distributed memory system
  with BSP-compliant memory consistency

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

Nesting supersteps: The neststep() statement



end of previous superstep

split current group into 2 subgroups
processors join new subgroup
renumber processor ranks
subgroup-wide

sub-supersteps are independently executed by the two subgroups

```
neststep( 2, @=... )
    statement;
```

changes to shared variables are only committed within the current group

restore parent group

final combine phase
commit inter-subgroup changes to shared variables

end of nested superstep

NestStep

- Idea: Fork-like language that relaxes the sequential memory consistency
- BSP model (bulk-synchronous parallelism)
- extension of C / C++ / Java
- hierarchical group concept as in Fork
- distributed shared memory emulation
  on top of a distributed memory system
  with BSP-compliant memory consistency
- (preliminary) language design: PDPTA'99
- implementation of a first prototype (1998)
  + run-time system written in Java
    (exploits Java object serialization and socket communication
  + experiences: works, but Java implementation is extremely inefficient
  + planned: implementation in C / MPI

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

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