Lesson: An introduction to Fork

Programming model
Hello World
Shared and private variables
Expression-level synchronous execution
Multiprefix operators
Synchronicity declaration
Synchronous regions: Group concept
Asynchronous regions: Critical sections and locks
Sequential vs. synchronous parallel critical sections
join statement
Software packages for Fork

SPMD style of parallel program execution

- fixed set of processors
- no spawn() command
- main() executed by all started processors as one group

The PRAM programming language Fork

language design: [Hagerup/Seidl/Schmitt'89] [K./Seidl'95, '97] [Keller,K., Träff'

Arbitrary CRCW PRAM with atomic multiprefix operators

synchronicity of the PRAM transparent at expression level

variables to be declared
either private or shared

private address subspaces embedded in shared memory

implementation for SB-PRAM

Hello World

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

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

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=00000001fc
EXIT: vp=#1, pc=00000011fc
EXIT: vp=#2, pc=00000001fc
EXIT: vp=#3, pc=00000011fc
Stop nach 11242 Runden, 642.400 kIps
01fc 18137FFF POPNG R6, ffffffff, R
## Shared and private variables

- each variable is classified as either shared or private "sharity"

- sh relates to defining group of processors

- pointers: no specification of pointee’s sharity required

---

### Synchronous execution at the expression level

**shared array**

```plaintext
s:  a[++] = a[++] + a[++];
```

// $ in {0..p-1} is processor rank

**synchronous execution**

result is deterministic

**asynchronous execution**

race conditions!

---

### Expressions: Atomic Multiprefix Operators (for integers only)

Set $P$ of processors executes simultaneously

$$k = \text{mpadd}( \text{ps}, \text{expression} );$$

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

$$k \leftarrow s_i + \sum_{j \in Q_{ps}, j < i} e_j$$

and memory location $ps_i$ is assigned the sum

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

---

### Example: Multiprefix addition

**mpadd** may be used as atomic fetch&add operator.
Expressions: Atomic Multiprefix Operators (cont.)

Example: User-defined consecutive numbering of processors

```c
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:

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

Atomic Update Operators / `ilog2`

- `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 \( \lfloor \log_2 k \rfloor \) for integer `k`

Synchronous and asynchronous program regions

Fork program code regions statically classified as either synchronous, straight, or asynchronous.

```
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;
    }
}
```

```
async void main( void )
{ sh int *A, n;
  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

```
G (inactive) current group (active)
```

```
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 (...) (see later)
```

```
```
```
```
```
Group concept

Groups of processors are explicit:

<table>
<thead>
<tr>
<th>Group ID: @</th>
<th>Group size: #</th>
<th>Group rank: $$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀ P₁ P₂ P₃</td>
<td>#</td>
<td>$$(automatically ranked from 0 to -1)</td>
</tr>
</tbody>
</table>

- Scope of sharing for function-local variables and formal parameters
- Scope of barrier-synchronization
- Scope of synchronous execution

Synchronicity invariant: (in synchronous regions): All processors in the same active group operate synchronously.

Implicit group splitting: IF statement with private condition

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

private condition expression
→ current group $G$ of processors must be split into 2 subgroups to maintain synchronicity invariant.

(parent) group $G$ is reactivated after subgroups have terminated → $G$-wide barrier synchronization

Implicit subgroup creation: Loop with private condition

```
while (cond) do
    statement;
```

Implicit subgroup creation: Loop with private condition

Explicit group splitting: The fork statement

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

body statement is executed in parallel by all subgroups in parallel

(program point 1)

(program point 2)
Dynamic / recursive splitting of groups into disjoint subgroups
→ at any time the group hierarchy is a logical tree.
supports nested (multi-level) parallelism
Asynchronous regions: Critical sections and locks (1)

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;
}
```

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 regions: Critical sections and locks (2)

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 {
    lock = 0;
    while (lock > 0) ; /* wait */
    lock = 1;
    var = var + 1.0;
    lock = 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 regions: Critical sections and locks (3)

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;
}
```

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 regions: Critical sections and locks (4)

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 );
}
```

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
Asynchronous regions: Predefined lock data types and routines

(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 }

Sequential vs. synchronous parallel critical sections (1)

- Sequential critical section
  - e.g., [Dijkstra'68]

  - sequentialization of concurrent accesses to a shared object / resource

- Synchronous parallel critical section
  - allow 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

Sequential vs. synchronous parallel critical sections (2)

Entry conditions?

When to terminate the entry procedure?

What happens with processors not allowed to enter?
### The join statement: excursion bus analogy (1)

```java
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: excursion bus analogy (2)

```java
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: excursion bus analogy (3)

```java
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: excursion bus analogy (4)

```java
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

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

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

```c
otherwise: form a group, execute busTourStatement synchronously
```
The join statement: excursion bus analogy (5)

```java
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

The join statement, example (1): parallel shared heap memory allocation

<table>
<thead>
<tr>
<th>time</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>……</th>
<th>P2047</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shmalloc(400)</td>
<td>shfree(10)</td>
<td>shmalloc(50)</td>
<td>shfree(56)</td>
<td>shmalloc(17)</td>
<td>shfree(500)</td>
</tr>
<tr>
<td></td>
<td>shmalloc(20)</td>
<td>shmalloc(40)</td>
<td>shmalloc(17)</td>
<td>shmalloc(300)</td>
<td>shmalloc(300)</td>
<td>shmalloc(17)</td>
</tr>
<tr>
<td></td>
<td>shmalloc(100)</td>
<td>shfree(4)</td>
<td>shmalloc(40)</td>
<td>shmalloc(12)</td>
<td>shmalloc(4)</td>
<td>shmalloc(4)</td>
</tr>
</tbody>
</table>

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?

The join statement, example (2)

**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`

<table>
<thead>
<tr>
<th>p</th>
<th>asynchronous</th>
<th>using join</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5390 cc (21 ms)</td>
<td>5508 cc (25 ms)</td>
</tr>
<tr>
<td>2</td>
<td>5390 cc (21 ms)</td>
<td>7076 cc (27 ms)</td>
</tr>
<tr>
<td>4</td>
<td>5420 cc (21 ms)</td>
<td>8764 cc (34 ms)</td>
</tr>
<tr>
<td>8</td>
<td>5666 cc (22 ms)</td>
<td>9522 cc (37 ms)</td>
</tr>
<tr>
<td>16</td>
<td>5698 cc (22 ms)</td>
<td>10034 cc (39 ms)</td>
</tr>
<tr>
<td>32</td>
<td>7368 cc (28 ms)</td>
<td>11538 cc (45 ms)</td>
</tr>
<tr>
<td>64</td>
<td>7712 cc (30 ms)</td>
<td>11678 cc (45 ms)</td>
</tr>
<tr>
<td>128</td>
<td>11216 cc (43 ms)</td>
<td>11482 cc (44 ms)</td>
</tr>
<tr>
<td>256</td>
<td>20332 cc (79 ms)</td>
<td>11432 cc (44 ms)</td>
</tr>
<tr>
<td>512</td>
<td>38406 cc (150 ms)</td>
<td>11556 cc (45 ms)</td>
</tr>
<tr>
<td>1024</td>
<td>75410 cc (294 ms)</td>
<td>11636 cc (45 ms)</td>
</tr>
<tr>
<td>2048</td>
<td>149300 cc (583 ms)</td>
<td>11736 cc (45 ms)</td>
</tr>
<tr>
<td>4096</td>
<td>300500 cc (1173 ms)</td>
<td>13380 cc (52 ms)</td>
</tr>
</tbody>
</table>

The join statement, example (3)

asynchronous parallel N-queens program uses join for parallel output of solutions

<table>
<thead>
<tr>
<th>time</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>……</th>
<th>P2047</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shmalloc(400)</td>
<td>shfree(10)</td>
<td>shmalloc(50)</td>
<td>shfree(56)</td>
<td>shmalloc(17)</td>
<td>shfree(500)</td>
</tr>
<tr>
<td></td>
<td>shmalloc(20)</td>
<td>shmalloc(40)</td>
<td>shmalloc(17)</td>
<td>shmalloc(300)</td>
<td>shmalloc(300)</td>
<td>shmalloc(17)</td>
</tr>
<tr>
<td></td>
<td>shmalloc(100)</td>
<td>shfree(4)</td>
<td>shmalloc(40)</td>
<td>shmalloc(12)</td>
<td>shmalloc(4)</td>
<td>shmalloc(4)</td>
</tr>
</tbody>
</table>

Experiment:
- 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?
Available software packages

PAD library [Träff'95–98], [PPP 8]
  PRAM algorithms and data structures

APPEND library [PPP 7.4]
  asynchronous parallel data structures

MPI core implementation in Fork [PPP 7.6]

Skeleton functions [PPP 7]
  generic map, reduce, prefix, divide-and-conquer, pipe, ...

FView fish-eye viewer for layouted graphs [PPP 9]

N-body simulation [PPP 7.8]