Lesson: An introduction to PRAM programming in Fork

Expressions: Atomic Multiprefix Operators (for integers only)

Let \( \psi_i \) be the location pointed to by the \( \psi \) expression of processor \( i \). Let \( Q_{\psi_i} / P \) denote the set of processors \( i \) with \( \psi_{\psi_i} \neq \psi \). Each processor \( i \) evaluates \( \psi \) expression to a value \( \epsilon_i \). Then the result returned by \( \text{mpadd} \) to processor \( i \) is the prefix sum

\[
\sum_{j=0}^{i} \epsilon_j
\]

and memory location \( \psi \) is assigned the sum

\[
\sum_{j=0}^{i} \epsilon_j \rightarrow \psi
\]

\( \epsilon \) and memory location \( \psi \) is assigned the sum

\[
\sum_{j=0}^{i} \epsilon_j \rightarrow \psi
\]

Then the result returned by \( \text{mpadd} \) to processor \( i \) is the prefix sum.

- \( \psi \) is evaluated as a value.
- Let \( Q_{\psi_i} / P \) be the set of processors with \( \psi_{\psi_i} \neq \psi \).
- Let \( \psi_{\psi_i} \) be the location pointed to by the \( \psi \) expression of processor \( i \).
- Let \( \psi \) be the location pointed to by the \( \psi \) expression of processor \( i \).

\[
\sum_{j=0}^{i} \epsilon_j \rightarrow \psi
\]

Synchronous execution simultaneously

Example: Multiprefix addition

[Diagram showing shared memory access and race conditions]

\[
\text{mpadd}(\text{mpadd}(3); 2); \\
\text{mpadd}(4);
\]

returns

\[
4 \rightarrow \text{shared memory}
\]

Shared and private variables

- Pointer: no specification of pointer’s sharity required
- subclass and superclass definitions
- is relative to defining group of variables
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)
- `mpor` (multiprefix bitwise or)
- `mpmax` may use as atomic `test&set` operator.

Example:

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

Atomic Update Operators

- `ilog2`:
  ```c
  syncadd(ps, e) atomically add value `e` to contents of location `ps`
  ```
  ```c
  syncmax atomically update with maximum
  ```
  ```c
  syncand atomically update with bitwise and
  ```
  ```c
  syncor atomically update with bitwise or
  ```

Synchronous and asynchronous program regions

A = read_array( &n );
A = sort( A, n );
{
  if ( n > 0 ) {
    return a;
    pr int myrank = compute_rank( a, n );
  }
  else
    return NULL;
  printf("Error: n=%d\n", n);
  a[myrank] = a[__PROC_NR__];
}
```

Switching from synchronous to asynchronous mode and vice versa

```c
if (n<100) print_array( A, n );
```

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

```c
start 
```
Groups of processors are explicit:

- Group concept
- Group ID: @
- Group size: # or groupsize()
- Group rank: $$ (automatically ranked from 0 to #-1)

+ 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.
- Group ranks $$ are (automatically) ranked from 0 to #-1.

Groups of processors are explicit:

- Explicit group splitting: The fork statement
- Implicit group splitting: The if statement
- Implicit subgroup creation: Loop with private condition

Examples of group splitting:
- Implicit:
  - With a conditional block
  - With a loop block
- Explicit:
  - With the fork statement
Koch(x[i], y[i], x[i+1], y[i+1], level + 1);

//  parallel divide-and-conquer step
fork( 4; i = $$ % 4; )

//  partially parallel divide-and-conquer step
// not enough processors in the group?
if (# < 4)

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;    y[2] = starty + dy/2;
    x[3] = startx + (2*dx/3); y[3] = starty + (2*dy/3);

//  terminate recursion:
if (level >= DEGREE) {
    return;
}

//  compute x and y coordinates of interpolation points P0, P1, P2, P3, P4:
for (i = 0; i < 4; i++)

for (i = 0; i < 4; i++)
Asynchronous regions: Critical sections and locks (1)

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

Example:

```c
float var = 0.0;
var = var + 1.0;
```

Access to var must be atomic. Atomic execution can be achieved by sequentialization (mutual exclusion).

Asynchronous regions: Critical sections and locks (2)

```c
int lock = 0;
float var = 0.0;
```

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

Access to var must be atomic. Atomic execution can be achieved by sequentialization (mutual exclusion).

Asynchronous regions: Critical sections and locks (3)

```c
int lock = 0;
float var = 0.0;
while (mpmax(&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 (4)

```c
int lock = 0;
float var = 0.0;
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.

```c
simple_lockup( sl );
```

Access to var must be atomic. Atomic execution can be achieved by sequentialization (mutual exclusion).
Asynchronous regions: Predefined lock data types and routines

(a) Simple lock
(b) Fair lock (FIFO order of access guaranteed)
(c) Readers/Writers lock (multiple readers OR single writer)
(d) Readers/Writers/Deletors lock (lockup fails if lock is being deleted)

```
mode in { RW_READ, RW_WRITE, RW_DELETE }
```

Sequential vs. synchronous parallel critical sections (1)
- deterministic parallel access by executing a synchronous parallel algorithm
- at most one group of processors inside at any point of time
- sequentialization of concurrent access to a shared object/resource

Sequential vs. synchronous parallel critical sections (2)
- allow simultaneous entry of more than one processor
- entry conditions?
- when to terminate the entry procedure?
- what happens with processors not allowed to enter?

Asynchronous Regions: Implementation of the Fair Lock

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

```c
get your ticket
/*wait*/
{  int myticket = mpadd( &(fl->ticket), 1);
while (myticket > fl->active) ;}
```

Sequential parallel critical section
- sequential critical section

Entry conditions?
- when to terminate the entry procedure?
- what happens with processors not allowed to enter?
The join statement: excursion bus analogy (1)

- execute else part:
- continue in else part: jump back to bus stop (join entry point)
- break in else part: continue with next activity (join exit point)

missedStatement;

Bus gone?

missedStatement;

join( SMsize; delayCond; stayInsideCond )

else

Ticket

join the Fork95 bus tours!

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:
driver then switches off the ticket automation
- ticket number is 0? -> driver
- otherwise: form a group, execute synchronously:
  - if not execute(spring off and continue with else part
  - else:

stayInsideCond

synchronously
Experiment:

First variant: sequential critical section, using a simple lock

Second variant: parallel critical section, using join

Question: Does this really pay off in practice?

Simple block-oriented parallel shared heap memory allocator

N-Queens program uses join for parallel output of solutions

asynchronous N-Queens program

traced time period: 439 msecs

trv

trv

- at return: leave the bus, re-open ticket automation and process them as a whole in parallel!
- collect multiple queries to shmalloc() / shfree() with join()
- if not ticket number is 0? -> driver!
- otherwise: form a group, execute the next activity
- otherwise: form a group, execute

simple block-oriented parallel shared heap memory allocator
Available software packages

- N-body simulation [ppp 7.8]
- FView fish-eye viewer for layered graphs [ppp 9]
- Asynchronous parallel data structures
  - Skeleton functions [ppp 7]
  - MPI core implementation in Fork [ppp 7.6]
- PRAM algorithms and data structures
  - APFEND library [ppp 7.4]
  - FViewfish-eye viewer for layouted graphs [ppp 9]