

Practical multicore programming

The DRF fragment

Slides and demos available at
<http://cambium.inria.fr/~maranget/MPRI/>

Luc Maranget

Luc.Maranget@inria.fr

1

Multicore programming, why?

Run faster.

Some program are much easier to write that way
(e.g. avoid asynchronous IO).

2

Run faster, our running example

The n queens puzzle: placing n chess queens on an $n \times n$ chessboard so that no two queens threaten each other.

A classic in backtracking: place one queen per row, from top to bottom:

```
int solve(int n, int row, int *cols) {
    if (row == n) return 1 ;
    int r = 0 ;
    for (int i = 0 ; i < n ; i++) {
        if (ok(i,row,cols)) {
            cols[row] = i ;
            r += solve(n,row+1,cols) ;
        }
    }
    return r;
}
```

Demo (in queens):

```
% ./q.out -v -v 8 2>&1 | less
...
```

3

Run faster, sequentially

Simple implementation (demo in queens):

```
% safe ./q.out 15
2279184
status: 0
real: 34.70
user: 34.57
sys: 0.01
```

Optimised implementation (demo in queens):

```
safe ./fast.out 15
2279184
status: 0
real: 0.66
user: 0.66
sys: 0.00
```

Optimisations: symetries, integers as bitsets, avoiding function calls...
The point is: parallelize the fastest program, not the slowest.

4

Faster without threads

A process in three steps

- 1 Place queens in the first d rows.
- 2 Count how many ways there are for placing the remaining $n - d$ queens.
- 3 Sum counts.

Step 2 can run concurrently...

Demo (queens), two C programs:

- `gen.out -d[d] n` : place the first d queens.
- `run.out` : count solutions with the first d queens placed as read on standard input.

The programs `gen.out` and `run.out` communicate through files.

We shall concurrently run the requested invocations of `run.out`, by using a shell script, a Makefile, or the `parallel` utility.

5

Before we parallelize, a quiz

How do we solve the 8-queens puzzle?

- With `./gen.out` only: set $d = 8$.

```
% ./gen.out -d8 8
0 8 8 0 4 7 5 2 6 1 3
1 8 8 0 5 7 2 6 3 1 4
...
45 8 8 5 7 1 3 0 6 4 2
```

With symmetry: $46 \times 2 \rightarrow 92$ solutions.

- With `./run.out` only: set $d = 0$.

```
% ./run.out
0 8 0
92
```

6

Aggressive parallelism

```
% sh shell.sh 17 > A/script.sh
```

Here is `script.sh`:

```
# Fork the computing processes, one per line of "gen.out 17" output
( echo 0 17 2 0 2 | ../run.out > 000.out ) &
( echo 1 17 2 0 3 | ../run.out > 001.out ) &
...
( echo 119 17 2 14 16 | ../run.out > 119.out ) &
# Wait for the computing processes to terminate
wait
# Sum partial results
cat 000.out 001.out ... 119.out > 17.out
( echo 0 && awk '{printf("%s +\n", $1)}' 17.out && echo p ) | dc
```

```
% safe sh script.sh
```

```
95815104
real: 4.74
user: 36.66
sys: 0.07
```

Drawback: Why run more than 8 processes at the same time on a 4 core $\times 2$ hyperthreaded machine?

7

Controlling parallelism with `make -j N`

```
% sh make.sh 17 > B/Makefile
```

The Makefile:

```
all: 17.out
    @( echo 0 && awk '{printf("%s +\n", $$1)}' 17.out && echo 'p' ) | dc

OUT := 000.out ... 119.out

17.out: $(OUT)
    @cat $(OUT) > 17.out

000.out:
    echo 1 20 15 1 1 1 1 1 | ../run.out > $@
...
063.out:
    echo 63 20 15 1 2 3 3 3 | ../run.out > $@

# My machine has 4 X 2 virtual processors
% safe make -C B -j 8
make: Entering directory '/home/maranget/MPRI/01/queens/B'
echo 0 17 2 0 2 | ../run.out > 000.out
...
95815104
make: Leaving directory '/home/maranget/MPRI/01/queens/B'
status: 0
real: 4.79
user: 36.98
sys: 0.05
#make -j 4 worth trying!
```

8

With the parallel utility

Usage:

```
parallel -j N command -- A1 ... An
```

Will run n invocations of *command* on arguments $A_1 \dots A_n$, with at most N invocations running concurrently.

This interface is not ideal, as our program `run.out` reads its arguments from standard input.

Easily corrected:

```
% cat run.sh
#! /bin/sh
echo $1 | ./run.out
```

9

Using the parallel utility

A `parallel.sh` script that combines `gen.out` and the `parallel` utility:

```
...
./gen.out -g$G $N | \
(
  while read arg; do A="'$arg' $A"; done
  echo 0
  eval "parallel $J sh ./run.sh -- $A" | \
    awk '{printf("%s +\n",$1)}'
  echo p
) | dc
```

Demo (in queens):

```
% J="-j 8" safe sh ./parallel.sh 17
95815104
status: 0
real: 4.80
user: 37.51
sys: 0.07
```

10

Processes vs. threads

Up to now we used *processes*.

A process is the running instance of a program:

- A process consists in, register values, memory, file descriptors etc.
- The process own its *memory*.
- Processes communicate (mostly) through the file system.

A *thread* is a lightweight process:

- A process may host several threads.
- A thread consists in register values, file descriptors, etc.
- The threads in a process share the memory (or part of).

11

Multicore programming, from inside programs

Principle

- Manage threads explicitly,
- threads communicate through shared memory.

Advantages

- Efficiency: threads cost less to create than processes. Also consider context switch cost.
- Memory is faster than files, or memory hardware is faster than disk.
- User convenience: only one program to run, no scripts.

Issues

- Very difficult to get right.
- Relaxed memory models...

This class

- Programming with the C POSIX threads library (pthread).
- Well synchronised programs only — programming on top of the Data Race Free model (DRF).

12

Starting a thread, getting its result

A.k.a. “asynchronous function call” — of `f` that accepts an argument of type `void *` and returns a result of type `void *`

```
void *f (void *arg) { ... }
```

```
void run(...) {
```

```
    // Compute f(arg) asynchronously.
```

```
    void *arg = ... ;
```

```
    pthread_t th ;
```

```
    create_thread(&th,f,arg) ;
```

```
    // Some computation performed concurrently with f(arg)
```

```
    ...
```

```
    // Get back f(arg)
```

```
    void *r = join_thread(&th) ;
```

```
    ...
```

```
}
```

13

Thread create and join, helper functions

Handle error checking — this is C!

```
#include <pthread.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
...
```

```
static void exit_error (char *msg, int st) {  
    fprintf(stderr, "%s: %s\n",msg,strerror(st));  
    exit(EXIT_FAILURE);  
}
```

```
}
```

```
void create_thread
```

```
(pthread_t *th,void *(*f)(void *),void *x) {
```

```
    int st = pthread_create(th,NULL,f,x) ;
```

```
    if (st != 0) exit_error("pthread_create",st) ;
```

```
}
```

```
void *join_thread(pthread_t *th) {
```

```
    void *r ;
```

```
    int st = pthread_join (*th,&r) ;
```

```
    if (st != 0) exit_error("pthread_join",st) ;
```

```
    return r ;
```

```
}
```

14

Informal semantics (man pthread_create)

```
int pthread_create
```

```
(pthread_t *th,...,void *(*f)(void *),void *z)
```

- Call `f` with argument `z` on a new thread whose identity is stored in `*th`.
- Returns 0 (success), or error status.
- ... are options, which we ignore for now.

```
int pthread_join(pthread_t th, void **r)
```

- If the thread identified by `th` has returned `v`, store `v` into `*r`.
- If not, suspend and wait for `th` to return.
- Returns 0 (success), or error status.
- It is an error to call `pthread_join` more than once on the same thread.

Notice: Threads can be created “*detached*”. Detached threads cannot join (and spare the needed resources).

15

C11 thread create and join

The “new” C11 standard defines the following functions, with shorter names and (unfortunately) a slightly different interface (in header `threads.h`).

```
typedef int (*thrd_start_t)(void*);
```

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

- The spawned function now returns an `int` (was `void *`)
- The ‘options’ argument is no longer here.

```
int thrd_join(thrd_t thr, int *res);
```

- We still have: if the thread identified by `thr` has returned `v`, store `v` into `*res`.
- But the type of `v` has changed w.r.t. `pthread`!

To keep things gcc simple, we stick to `pthread`.

16

Asynchronous function call, easy example I

Let us compute:

$$\sum_{k=1}^n k^2$$

Sketch

- Fork n threads to compute $1^2, 2^2, \dots, n^2$.
- Sum square as we get thread results.

17

Asynchronous function call, easy example I

A bit of boxing.

```
// ‘‘Boxed’’ int
typedef struct { int v ; } val_t ;
// Or typedef int val_t ;
val_t *alloc_val(int i) ;
void free_val(val_t *p) ;

// Actual computation
int square(int i) { return i*i ; }

// Stub function
void *f(void *p) {
    val_t *_p = (val_t *)p ;
    int i = _p->v ;
    free_val(_p) ;
    return alloc_val(square(i)) ;
}
```

Safer and cleaner than casting “**int**” (or “**intptr_t**”) into “**void ***” and back.

18

Asynchronous function call, easy example II

```
int sum(int n) {
    // Fork
    pthread_t th[n] ;
    for (int k = 0 ; k < n ; k++)
        create_thread(&th[k],f,alloc_val(k+1)) ;

    // Retrieve and sum results
    int r = 0 ;
    for (int k = 0 ; k < n ; k++) {
        val_t *p = (val_t *)join_thread(&th[k]) ;
        r += p->v ;
        free_val(p) ;
    }
    return r ;
}
```

19

Petty optimisation: spare one thread

```
int sum(int n) {
    // Fork
    pthread_t th[n-1] ;
    for (int k = 0 ; k < n-1 ; k++)
        create_thread(&th[k],f,alloc_val(k+1)) ;

    // Retrieve and sum results
    int r = square(n) ;
    for (int k = 0 ; k < n-1 ; k++) {
        val_t *p = (val_t *)thread_join(&th[k]) ;
        r += p->v ;
        free_val(p) ;
    }
    return r ;
}
```

20

Exercise I

Count n -queens solutions using aggressive parallelism. That is write:

```
/* count_t is the type of unsigned 64bits integers */
count_t run(int n, int depth)
```

Useful functions:

```
/* Thread create and join*/
void create_thread(pthread_t *th, void *(*f)(void *), void *x) ;
void *join_thread(pthread_t *th) ;

/* Subtask: concretely a placement of the d first queens */
typedef struct { ... } subtask_t ;

/* Run a subtask */
count_t run_subtask(subtask_t *z) ;

typedef void emit_t(subtask_t *z) ;
/* Subtask generator, calls emit on all subtasks,
   returns number of generated subtasks */
int generate_subtasks(int n, int depth, emit_t emit) ;
```

21

A little help

This is how one uses generator/runner for sequential computation: (exp1/seq.out).

```
static count_t sum ;

void emit_run(subtask_t *z) {
    sum += run_subtask(z) ;
}

count_t run(int n, int depth) {
    sum = 0 ;
    (void)generate_subtasks(n,depth,emit_run) ;
    return sum ;
}
```

Hence, write “emit” that creates threads and join on them later. Simplification: you can assume there will be less then NTASKS subtasks.

22

A little more help

Define the proper type `val_t` for boxed `count_t`:

```
typedef struct { count_t c ; } val_t ;
```

```
void free_val(val_t *p) ;
val_t *alloc_val(count_t c) ;
```

Define an array to store threads identifiers:

```
static pthread_t th[NTASKS] ;
static int th_next ;
```

Then, it's up to you:

```
void *run_stub(void *z) {
    count_t r = run_subtask((subtask_t *)z) ; free_arg(z) ;
    return alloc_val(r) ;
}

void emit_thread(subtask_t *z) {
    if (th_next >= NTASKS) exit EXIT_FAILURE ; // Enough space?
    create_thread(&th[th_next],run_stub,copy_arg(z)) ; //NB:copy
    th_next++ ;
}
```

23

Solution I

```
count_t run(int n, int depth) {
    count_t sum = 0 ;

    // Fork
    th_next = 0 ;
    int ntasks = generate_subtasks(n,depth,emit_thread) ;

    // Join
    for (int k = 0 ; k < ntasks ; k++) {
        val_t *r = (val_t *)join_thread(&th[k]) ;
        sum += r->c ;
        free_val(r) ; // optional
    }
    return sum ;
}
```

Demo: Check performance (queens/tnaive.out).

```
% safe ./tnaive.out 17
...
real: 4.56
user: 36.17
sys: 0.06
```

24

Optimising the sum of squares

Avoid dynamic memory allocation (a frequent C programmer's concern)

```
static volatile int sq ; // Notice "volatile"

void *f2(void *p) {
    int i = ((val_t *)p)->v ;
    sq += i*i ; // ie int x = sq ; int y = i*i ; sq = x + y ;
    return NULL ;
}
```

Savings achieved:

- Update running sum "sq" instead of returning boxed result.

Also notice:

- Argument space reclaimed by caller.

25

Optimised sum of squares

```
int sum2(int n) {
    sq = 0 ; // Be cautious
    // Fork
    pthread_t th[n] ; val_t arg[n] ; // Stack allocation
    for (int k = 0 ; k < n ; k++) {
        val_t *a = &arg[k] ;
        a->v = k+1 ;
        create_thread(&th[k],f2,a) ;
    }

    // Join
    for (int k = 0 ; k < n ; k++)
        (void)join_thread(&th[k]) ;

    return sq ;
}
```

Do you see a problem? Yes, the program is broken...

26

A simpler, broken, program

```
static volatile int sum = 0, start = 0 ;

void *f(void *p) {
    while (!start) ; // Wait partner
    sum++ ;
    return NULL ;
}

void run_loose(int n) {
    int broken = 0 ;
    for (int k = 0 ; k < n ; k++) {
        sum = 0 ; start = 0 ;
        pthread_t th1,th2 ;
        create_thread(&th1,f,NULL) ; create_thread(&th2,f,NULL) ;
        start = 1 ;
        (void)join_thread(&th2) ; (void)join_thread(&th1) ;
        if (sum != 2) broken++ ;
    }
    if (broken > 0) printf("Broken:_%i/_%i\n",broken,n) ;
}
```

Demo (in exp20): ./two.out

27

Where broken?

The instructions `sum++` performs two accesses to memory:

```
int x = sum ; // Access R, read.
int y = x + 1 ; // Compute
sum = y ; // Access W, write.
```

In our programming model (SC) accesses are *atomic* (they don't mix). The effect of a program on memory results of a given interleaving of memory accesses (a.k.a a *schedule*).

Consider the following scheduling for threads 1 and 2:

$$R_1, R_2, W_2, W_1$$

The final value of `sum` is: 1.

28

Enforcing atomicity with locks

If $[R, W]$ is considered a scheduling unit, the remaining schedules are:

$[R_1, W_1], [R_2, W_2] \quad [R_2, W_2], [R_1, W_1]$

And the result is always 2.

In practice, the “scheduling unit” is defined by a *lock* L (or mutual exclusion lock) as:

```
lock(L) ;
sum++ ;
unlock(L) ;
```

The instruction block from lock to unlock is a *critical section*.
And there can be several locks, only the critical sections of the same lock do not mix.

29

Mutexes, or locks, creation

We allocate all such primitive data dynamically (calling malloc/free).

```
pthread_mutex_t *alloc_mutex(void) {
    pthread_mutex_t *r = malloc_check(sizeof(*r)) ;
    // Important, initialize mutex
    int st = pthread_mutex_init(r, NULL) ;
    if (st != 0) exit_error("pthread_mutex_init", st) ;
    return r ;
}

void free_mutex(pthread_mutex_t *p) {
    int st = pthread_mutex_destroy(p) ;
    if (st != 0) error_exit("pthread_mutex_destroy", st) ;
    free(p) ;
}
```

30

Locking and unlocking (pthreads)

We write some wrappers for error-checking — this is C!

```
void lock_mutex(pthread_mutex_t *p) {
    int st = pthread_mutex_lock(p) ;
    if (st != 0) exit_error("pthread_mutex_lock", st) ;
}

void unlock_mutex(pthread_mutex_t *p) {
    int st = pthread_mutex_unlock(p) ;
    if (st != 0) exit_error("pthread_mutex_unlock", st) ;
}
```

31

Bonus: C11 locks (not available everywhere)

```
#include <threads.h>

mtx_t *alloc_mutex(void) {
    mtx_t *r = malloc_check(sizeof(*r)) ;
    int st = mtx_init(r, mtx_plain) ;
    if (st != thrd_success) exit_cmd("mtx_init") ;
    return r ;
}
...

void lock_mutex(mtx_t *p) {
    int st = mtx_lock(p) ;
    if (st != thrd_success) exit_cmd("mtx_lock") ;
}
```

Etc. Notice the slightly cleaner interface... (e.g. `mtx_init` second argument).

32

The informal semantics of locks

A lock holds a bit of information: taken or free.

- **lock**: Acquire the lock.
 - Read status, if free, then set status to taken – *Atomically*. and return.
 - If taken then wait until free.
 - By polling the status, (*busy wait*),
 - or by going to sleep.
- **unlock**: Release the lock.
 - Set lock status to free,
 - then awake one sleeping thread, if any,
 - then return.

Remark: Same functionality as class 00 (Dekker), *very* different implementation.

33

Bonus: C11 atomics

C11 atomics feature “atomic” read-modify-write operations.

```
atomic_int sum = ATOMIC_VAR_INIT(0) ;
atomic_int start = ATOMIC_VAR_INIT(0) ;

void *f(void *p) {
    while (!atomic_load_explicit(&start, memory_order_relaxed)) ;
    // Atomic increment of sum
    (void)atomic_fetch_add_explicit(&sum, 1, memory_order_relaxed) ;
    return NULL ;
}
```

Besides, conflicting accesses on C11 atomics are not racy.

34

The simpler program fixed

```
static volatile int sum = 0 ;
static pthread_mutex_t *mutex ;

void *g(void *p) {
    lock_mutex(mutex) ;
    sum++ ;
    unlock_mutex(mutex) ;
    return NULL ;
}

void run_locked(void) {
    sum = 0 ;
    mutex = alloc_mutex() ;

    pthread_t th1, th2 ;
    create_thread(&th1, f, NULL) ; create_thread(&th2, f, NULL) ;
    (void)join_thread(&th2) ; (void)join_thread(&th1) ;
    free_mutex(mutex) ;
    if (sum != 2) printf("Pas possible") ;
}
```

35

Problems with locks

Performance:

- Critical sections cannot execute simultaneously, and parallelism decreases. Solutions.
 - Write short critical sections, in particular avoid non-termination risks.
 - Use several locks (but see next slide).
- The code for of lock/unlock takes time. Solution: attempt balance between poll/suspend.
- Contention: when a lot of lock are performed simultaneously, performance degrades severely. Solution: hierarchical locks.

36

Problems with locks

Correction: Locks are error prone. One easily reaches *deadlock*:

Assume that f_1 use lock L_1 and f_2 uses L_2 . Then, if f_1 and f_2 are mutually recursive, we may have the following execution trace:

Thread 1	Thread 2
lock(L_1) ;	lock(L_2) ;
...	...
lock(L_2) ;	lock(L_1) ;
...	...
unlock(L_2) ;	unlock(L_1) ;
...	...
unlock(L_1) ;	unlock(L_2) ;

As a consequence, programming with lock is not compositional, saved for a crippling discipline: use only one lock (or *master lock*), or take locks following a defined order (hard to check).

37

Exercise II

Write the n -queens solver without `thread_join`.

Thread creation is simplified, since there is no need to save threads somewhere. Termination gets more involved.

To address termination, let us define a new *component* `wait_t` and two primitives:

```
/* Perform a 'tick' */
void tick(wait_t *p) ;

/* Wait for n ticks ticks */
void wait_done(wait_t *p, int n) ;
```

Synchronisation is as follows, n subtasks are generated:

- ▶ A tick is performed whenever a subtask is completed,
- ▶ Programm terminates having waited for n ticks.

38

Computing a subtask, and ticking

```
static volatile count_t sum ;
static pthread_mutex_t *mutex ;
static wait_t *wait_on ; // New component, to be defined.

void *run_stub(void *z) {
    count_t r = run_subtask((subtask_t *)z) ;
    free_arg(z) ;
    lock_mutex(mutex) ; sum += r ; unlock_mutex(mutex) ;
    tick(wait_on) ; // Signal I am done
    return NULL ;
}

void emit_thread(subtask_t *z) {
    pthread_t th ; // Hum, looks ok for detached thread!
    // Notice detached, ie no provision for join.
    create_thread_detached(&th, run_stub, copy_arg(z)) ;
}
```

39

Coding bonus

For the curious, here is the code of `create_thread_detached`:

```
void create_thread_detached
(pthread_t *th, void *(*f)(void *), void *x) {
    pthread_attr_t tattr ;
    int st ;

    st = pthread_attr_init(&tattr) ;
    if (st != 0) exit_error("pthread_attr_init",st) ;

    st = pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED) ;
    if (st != 0) exit_error("pthread_attr_setdetachstate",st) ;

    st = pthread_create(th, &tattr, f, x) ;
    if (st != 0) exit_error("pthread_create",st) ;

    st = pthread_attr_destroy(&tattr) ;
    if (st != 0) exit_error("pthread_attr_destroy",st) ;
}
```

Well... Just remember that one does not join on a detached thread and thus spare the associated resources.

40

Breaking news: simpler coding...

```
void create_thread_detached
(pthread_t *th, void *(*f)(void *), void *x) {
    int st ;

    st = pthread_create(th, NULL, f, x) ;
    if (st != 0) exit_error("pthread_create", st) ;
    st = pthread_detach(*th);
    if (st != 0) exit_error("pthread_detach", st) ;
}
```

41

Exercise II, run function

```
count_t run(int n, int depth) {
    /* Initialise */
    mutex = alloc_mutex() ;
    wait_on = alloc_wait() ;
    sum = 0 ;

    /* Fork all subtasks */
    int ntasks = generate_subtasks(n, depth, emit_thread) ;

    /* Wait result */
    wait_done(wait_on, ntasks) ;
    return sum ;
}
```

42

Real exercise II

Write the `wait_t` component.

```
/* Component to wait on */
typedef struct {
    pthread_mutex_t *mutex ;
    volatile int nret ;
} wait_t ;

wait_t *alloc_wait(void) {
    wait_t *r = malloc_check(sizeof(*r)) ;
    r->nret = 0 ;
    r->mutex = alloc_mutex() ;
    return r ;
}

void tick(wait_t *p) ; // To be written
void wait_done(wait_t *p, int ntasks) ; // To be written
```

43

Solution II

Function `tick` is easy, we have seen this before:

```
void tick(wait_t *p) {
    lock_mutex(p->mutex) ;
    p->nret++ ;
    unlock_mutex(p->mutex) ;
}
```

Function `wait_on` looks easy:

```
void wait_done(wait_t *p, int ntasks) {
    while (p->nret < ntasks) ;
}
```

But...

- Busy waiting degrades performance (demo `queens/busy.out`, can be worse)
- We are no longer in the DRF fragment of pthreads!

44

Data Race Free guarantee

Race: occurs when two different threads access the same shared location *simultaneously*, and when at least one access is a write.

(Non) simultaneous accesses Accesses “ordered” by pthreads calls:

- Accesses in critical sections that use the same lock.
- Accesses performed before calling `pthread_create`, and accesses by the created thread.
- Accesses by thread `th`, and accesses performed by the caller of `pthread_join(th, ...)`.

DRF execution: An execution with no data races.

DRF guarantee: All executions of a program whose SC executions are DRF (a DRF program) are SC. The behaviour of non-DRF programs is unspecified.

And indeed we have a race on `wait_on->nret`.

45

Data-race old-style

Simultaneous accesses to data may, in some circumstance, yield absurd results. On some 32bits mode, we execute simultaneously:

```
void P0(void *p) {
    uint64_t *x = (uint64_t)p;
    *x = 0x0101010101010101 ;
}

void P1(void *p) {
    uint64_t *x = (uint64_t *)p;
    uint64_t r = *x;
}
```

Demo :

```
# We execute the test (initial value of *x is zero)
10025054:>1:r=0x0;
29906 :>1:r=0x1010101;
9945040:>1:r=0x101010101010101;
```

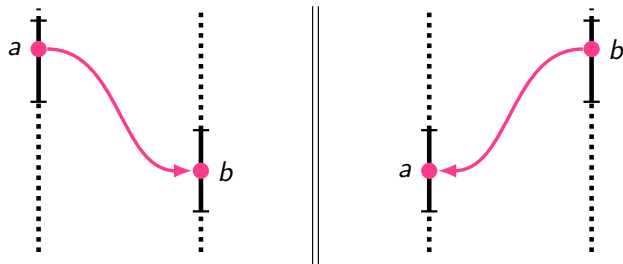
Value `0x1010101` results from accesses to quad words not being atomic.

46

Ordering memory accesses

Critical sections restore atomicity by mutual exclusion. That is, the instructions in critical section never execute simultaneously:

As a consequence of critical sections non-overlapping, critical sections are ordered:



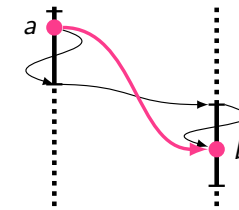
Let *a* and *b* be memory accesses.

A usable (*i.e.* with DRF guarantee) memory model will lift critical section ordering to memory accesses.

47

Ordering accesses, in pthreads

In POSIX threads, two accesses “separated” by synchronisation calls are *not* simultaneous. Hence, they are not *racy*.



Here, accesses *a* and *b* are separated by the sequence unlock-lock.

- UnLock has *release* semantics (before release → order).
- Lock has *acquire* semantics (after acquire → order).

48

Coarse locking

```

void P0
(int *x, int *y, mtx_t *m) {
    mtx_lock(m);
    *x = 1;
    int r0 = *y;
    mtx_unlock(m);
}

void P1
(int *x, int *y, mtx_t *m) {
    mtx_lock(m);
    *y = 1;
    int r1 = *x;
    mtx_unlock(m);
}

```

Demo ? All accesses of P₀ (resp. P₁) are performed before those of P₁ (res. P₀). Thus we get:

```

...
Histogram (2 states)
9999825:>0:r0=1; 1:r1=0;
10000175:>0:r0=0; 1:r1=1;
...

```

Notice : The "order" of accesses inside critical sections is irrelevant.

49

Fine-grain locking

To a memory cell, one associate a mutex ($x \rightarrow mx$ and $y \rightarrow my$).

```

void P0 (int *x, int *y, mtx_t *mx, mtx_t *my) {
    mtx_lock(mx); *x = 1; mtx_unlock(mx);
    mtx_lock(my); int r0 = *y; mtx_unlock(my);
}

```

```

void P1 (int *x, int *y, mtx_t *mx, mtx_t *my) {
    mtx_lock(my); *y = 1; mtx_unlock(my);
    mtx_lock(mx); int r1 = *x; mtx_unlock(mx);
}

```

Demo ? (in litmus/) All potential *data-races* (i.e. all conflicting accesses) are covered. One sees that the non-SC outcome ($r0=0$ et $r1=0$) is not observed :

```

...
Histogram (3 states)
9455682:>0:r0=1; 1:r1=0;
9280838:>0:r0=0; 1:r1=1;
1263480:>0:r0=1; 1:r1=1;
...

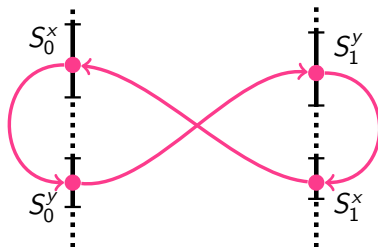
```

50

Some sufficient condition for fine-grain locking

Let us note S_0^x (write x), S_0^y (read y), S_1^x (read x), S_1^y (write y) les sections critiques.

Let us assume $r0=0$ (P₀ reads 0 from y) and $r1=0$ (P₁ reads 0 from x).

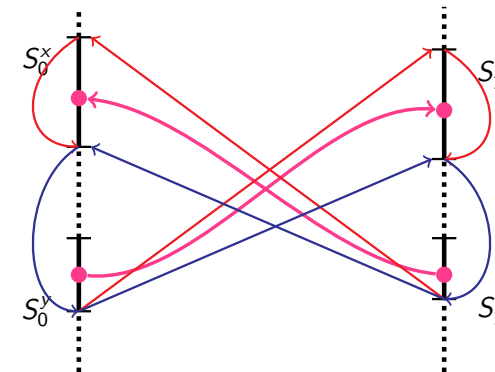


And that critical section of a same thread stay in-order, even when they operate on different mutexes.

Bingo ! (cycle.) That is, the non-SC behaviour is excluded by fine-grain locking.

51

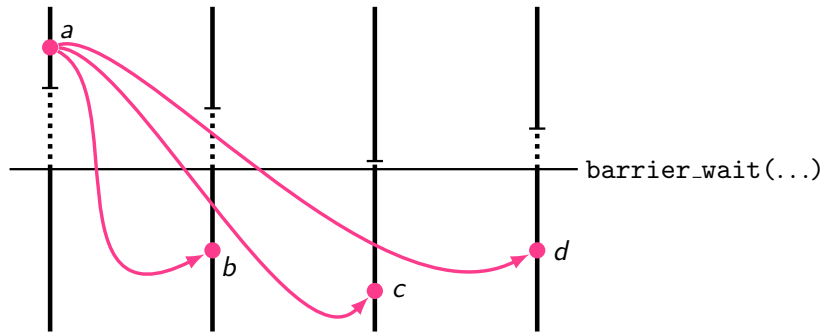
A more realistic account of fine-grain locking



As $r0=0$ and $r1=0$, from unlock to lock, **acquire**, **unlocks** ordered, **release** and **Bingo!**

52

There is more than locks: synchronisation barrier



Accesses before (a) and after (b, c and d) the barrier cannot execute simultaneously. There is no data-race.

53

Back to our race on p->nret, bad solutions

We can...

- Ignore the issue (it works on x86).
- Program (correctly) out of the DRF fragment (hard, non-portable [no longer true with C11], see next classes).
- Avoid the race, as writes to p->nret are protected by p->mutex, we additionally protect reads:

```
void wait_done(wait_t *p, int ntasks) {
    for ( ; ; ) {
        int over ;
        lock_mutex(p->mutex) ;
        over = p->nret >= ntasks ;
        unlock_mutex(p->mutex) ;
        if (over) return ;
    }
}
```

Efficiency penalty may be severe (try, demo)... It can be alleviated by introducing a sleep delay in loop (however, introduces latency).

54

Solving all problems

We aim at:

- The waiting thread sleeps as long as less than ntasks computing threads have ticked.
- The last thread to tick awake the main thread.

In fact, we aim at something similar to the “sleep when someone is in critical section” behavior or mutexes.

A condition variable *C* is a device for doing this.

- wait(*C*,*L*), release the lock *L* and suspend on condition *C* atomically.
- signal(*C*) wake up one thread suspended on *C*, if any.
- broadcast(*C*) wake up all threads suspended on *C*.

Notice that awoken threads will hold the mutex *L* they have released when performing wait(*C*,*L*).

55

A simpler example

A synchronous cell, to be used once (pthread_join may use something similar).

```
typedef struct {
    pthread_mutex_t *mutex;
    pthread_cond_t *cond;
    int v, something ;
} cell_t ;

// NB. alloc_cell initialises mutex and condition
cell_t alloc_cell(void) ;
void free_cell(cell_t *p) ;

// Put v in cell
void put(cell_t *p, int v) ;

// Get value from cell, suspending until something is here.
int get(cell_t *p) ;
```

56

A simpler example, continued

```
void put(cell_t *p, int v) {
    lock_mutex(p->mutex) ;
    if (something) {
        unlock_mutex(p->mutex) ;
        fprintf(stderr, "put more than once!\n") ;
        exit(2)
    }
    p->something = 1 ;
    p->v = v ;
    signal_cond(p->cond) ; // Signal (potential) reader
    unlock_mutex(p->mutex) ;
}

int get(cell_t *p) {
    int r ;
    lock_mutex(p->mutex) ;
    // Correct, when wait_cond returns only when signalled
    if (!something) wait_cond(p->cond, mutex) ;
    r = p->v ;
    unlock_mutex(p->mutex) ;
    return r ;
}
```

57

Spurious wakeups

Notice We wrote `if (!something) wait_cond(...)`. While the preferred idiom is `while (!something) wait_cond(...)`.

Why so?

- **Spurious wakeups?** `wait_cond` may return for any reason.
- POSIX standard allows spurious wakeups.
 - May facilitate implementation:
 - Interruption/signal handling?
 - Or, `signal_cond` may awake more than one waiter...
 - Program logic (see FIFO later) often commands a loop around `wait_cond` anyway.
- I did not observe them,
- but we have to program according to the standard.

58

get in presence of spurious wakeups

The function `get` must be written as follows:

```
int get(cell_t *p) {
    int r ;
    lock_mutex(p->mutex) ;
    // If no spurious wakeup, loop will run at most once
    while (!something) wait_cond(p->cond, mutex) ;
    r = p->v ;
    unlock_mutex(p->mutex) ;
    return r ;
}
```

59

Exercise II with a condition variable, part A

```
typedef struct {
    pthread_mutex_t *mutex ;
    pthread_cond_t *cond ;
    volatile int nret, ntasks ; // Notice ntasks kept inside
} wait_t ;

wait_t *alloc_wait(void) {
    wait_t *r = malloc_check(sizeof(*r)) ;
    r->nret = 0 ;
    r->ntasks = 0 ;
    r->mutex = alloc_mutex() ;
    r->cond = alloc_cond() ;
    return r ;
}

void tick(wait_t *p) ; // To be written
void wait_done(wait_t *p, int ntasks) ; // To be written
```

60

Solution II with a condition variable

```
void wait_done(wait_t *p, int ntasks) {
    lock_mutex(p->mutex) ;
    p->ntasks = ntasks ;
    while (p->nret < p->ntasks)
        wait_cond(p->cond, p->mutex) ;
    unlock_mutex(p->mutex) ;
}

void tick(wait_t *p) {
    lock_mutex(p->mutex) ;
    p->nret++ ;
    if (p->ntasks > 0 && p->nret >= p->ntasks)
        signal_cond(p->cond) ;
    unlock_mutex(p->mutex) ;
}
```

61

Controlled parallelism

Remember, using `make -j N` (or `parallel -j N`) we could limit computing processes to N instances.

We want the same for threads.

Idea:

- Have N computing threads,
- which execute available subtasks one after the other, sequentially.

Sometimes called “a processor farm”, computing threads are “slaves”. A “master” allocates subtasks to slaves.

62

Master and slaves

Assume a (concurrent, blocking, bounded) FIFO component:

Slaves get subtasks from the FIFO:

```
static fifo_t *fifo ;
static count_t sum ;
static pthread_t *mutex ;

void *slave(void *) {
    for ( ; ; ) {
        subtask_t *z = get(fifo) ; // Will block if fifo is empty
        count_t c = run_subtask(z) ;
        free_arg(z) ;
        lock_mutex(mutex) ;
        sum += c ;
        unlock_mutex(mutex) ;
    }
}
```

63

Master and slaves

While the master (main thread) put subtasks into the fifo.

```
void emit_fifo(subtask_t *z) {
    // Will block if fifo is full
    put(fifo, copy_arg(z)) ;
}

int master(int n, int depth) {
    return generate_subtasks(n, depth, emit_fifo) ;
}
```

Code sketch

```
count_t run(int n, int depth, int nprocs, int fsz) {
    sum = 0 ;
    // Initialise
    fifo = create_fifo(fsz) ;
    // Fork slaves
    pthread_t th[nprocs] ;
    for (int k = 0 ; k < nprocs ; k++)
        create_thread(&th[k], slave, NULL) ;
    // Act as master
    int ntasks = master(n, depth) ;
    // Shall see master termination later
    ...
}
```

64

Let us write the fifo

Starting from a non-concurrent, non-blocking, bounded fifo.

```
typedef struct {
    int sz ;
    int fst, lst, nitems ;
    subtask_t **t ; // Array of (subtask_t *)
} fifo_t ;

typedef enum {OK, NO} ret_val ; // Return value for put below

int put(fifo_t *f, subtask_t *z) {
    if (f->nitems == f->sz) return NO ;
    f->t[f->lst] = z ;
    f->lst++ ; f->lst %= f->sz ; f->nitems++ ;
    return OK ;
}

subtask_t *get(fifo_t *f) {
    subtask_t *r ;
    if (f->nitems == 0) return NULL ; // special value
    r = f->t[f->fst] ;
    f->fst++ ; f->fst %= f->sz ; f->nitems-- ;
}
```

65

Concurrent fifo, definition

```
typedef struct {
    pthread_mutex_t *mutex ;
    pthread_cond_t *is_empty, *is_full ;

    int sz ;
    int fst, lst, nitems ;
    subtask_t **t ;
} fifo_t ;
```

66

Concurrent fifo, creation

```
fifo_t *alloc_fifo(int sz) {
    fifo_t *r = malloc_check(sizeof(*r)) ;
    r->fst = r->lst = r->nitems = 0 ;
    r->sz = sz ;
    r->t = calloc(sz, sizeof(*r->t)) ;
    if (!r->t) {
        perror("calloc") ;
        exit(2) ;
    }
    r->mutex = alloc_mutex() ;
    r->is_empty = alloc_cond() ;
    r->is_full = alloc_cond() ;
    return r ;
}
```

67

Concurrent fifo, put

```
void put(fifo_t *f, subtask_t *z) {
    lock_mutex(f->mutex) ;
    // If full ?
    while (f->nitems == f->sz) {
        wait_cond(f->is_full, f->mutex) ;
    }
    // Now store z
    int was_empty = f->nitems == 0 ;
    f->t[f->lst] = z ;
    f->lst++ ; f->lst %= f->sz ; f->nitems++ ;
    // If was empty?
    if (was_empty) {
        broadcast_cond(f->is_empty) ; // Why not signal?
    }
    unlock_mutex(f->mutex) ;
}
```

68

Concurrent fifo, get

Exercise III!

```
subtask_t *get(fifo_t *f) {
    subtask_t *r ;
    lock_mutex(f->mutex) ;
    // If empty?
    while (f->nitems == 0) {
        wait_cond(f->is_empty,f->mutex) ;
    }
    // Retrieve value
    int was_full = f->nitems == f->sz ;
    r = f->t[f->fst] ;
    f->fst++ ; f->fst %= f->sz ; f->nitems-- ;
    // If was full
    if (was_full) {
        broadcast_cond(f->is_full) ;
    }
    unlock_mutex(f->mutex) ;
    return r ;
}
```

69

Why not signal (instead of broadcast) in put and get?

If you do not see, try `ex3/bad.out -v 18` (code with signal in place of broadcast).

We then see no parallelism, why?

We witness the following scenario (or a similar one):

- N slaves suspend on the empty fifo,
- The master fills the fifo, awaking one slave while putting the first task.
- As a result, $N - 1$ tasks are suspended and no one awakes them.

70

How do we detect termination?

Solution (1) the `wait_t` component:

```
void *slave(void *) {
    for ( ; ; ) {
        subtask_t *z = get(fifo) ; // Will block if fifo is empty
        count_t c = run_subtask(z) ;
        ...
        tick(wait_on) ;
    }
}
```

Problem: thread resources are not reclaimed (to that aim slave should return...).

```
// Shall see master termination later
wait_done(wait_on,ntasks) ;
...
return count ;
}
```

(Going further: Add a kill functionality to the fifo.)

71

How do we detect termination?

Solution (2): special value `NULL` in fifo means that computation is over.

Master:

```
// Shall see master termination later
put(fifo,NULL) ;
for (int k = 0 ; k < nprocs ; k++)
    (void)pthread_join(&th[k]) ;
...
```

And slave:

```
void *slave(void *) {
    for ( ; ; ) {
        subtask_t *z = get(fifo) ; // Will block if fifo is empty
        if (z == NULL) {
            put(fifo,NULL) ; // For other slaves..
            return NULL ;
        }
        ...
    }
}
```

Notice: Fifo behaviour is instrumental. **Demo:** Efficiency (`queens/topt.out -j4 -d2 17`).

72

Reclaiming the fifo

```
count_t run(int n, int depth, int nprocs, int fsz) {
    sum = 0 ;
    fifo = alloc_fifo(fsz) ;
    mutex = alloc_mutex() ;
    ...
    // Complete master cleanup and termination
    put(fifo, NULL) ;
    for (int k = 0 ; k < nprocs ; k++)
        (void)pthread_join(&th[k]) ;
    /*
     * At this point, no slave is blocked on fifo,
     * since all slaves returned. We can reclaim the fifo
     */
    free_fifo(fifo) ;
    free_mutex(mutex) ; // Reclaim mutex
    return 2*sum ; // By symmetry
}
```

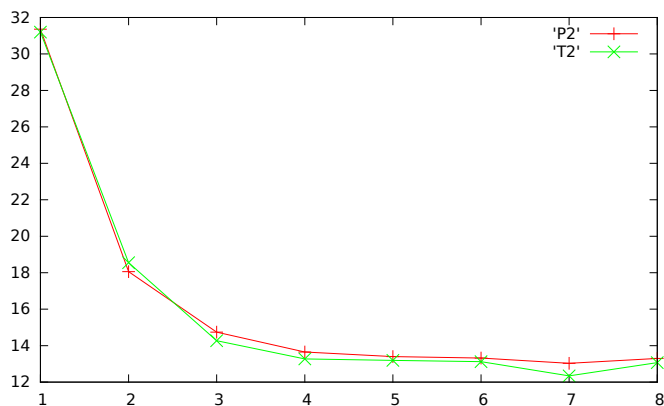
73

Practical multicore programming

Threads vs. Processes, the n -queens experiment

74

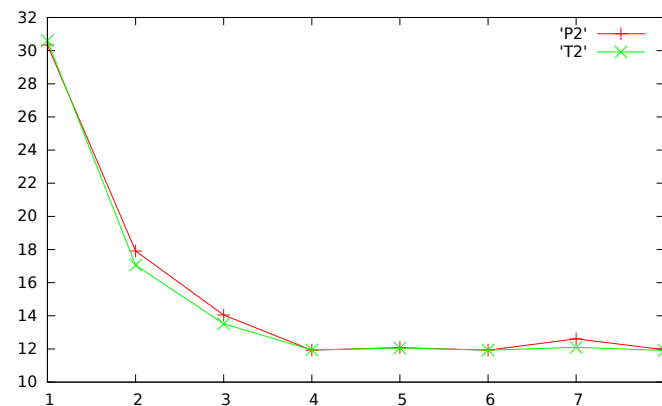
Performance on some 2 cores, $\times 2$ machine



wall clock time in sec. P2 is the “parallel” implementation, T2 is the pthreads implementation.

75

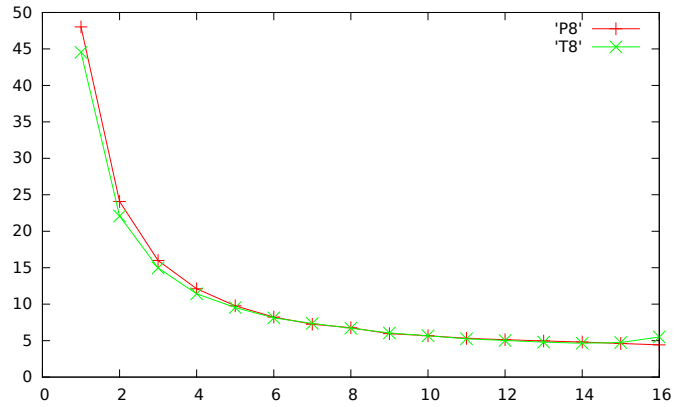
Performance on this 2 cores, $\times 2$ machine



Wall clock time in sec. P2 is the “parallel” implementation, T2 is the pthreads implementation.

76

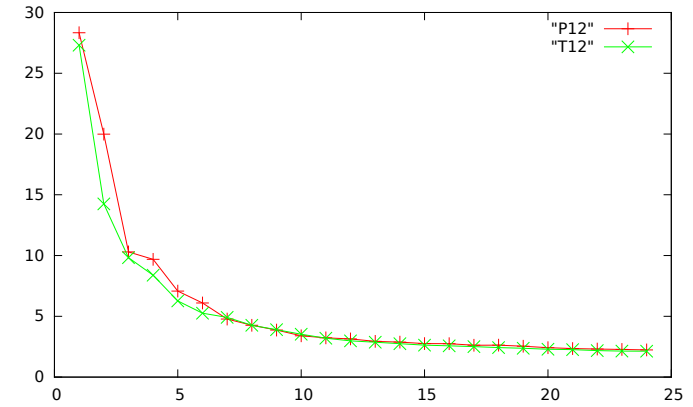
Performance on a 8 cores, ×2 machine



Wall clock time in sec. P8 is the “parallel” implementation, T8 is the pthreads implementation.

77

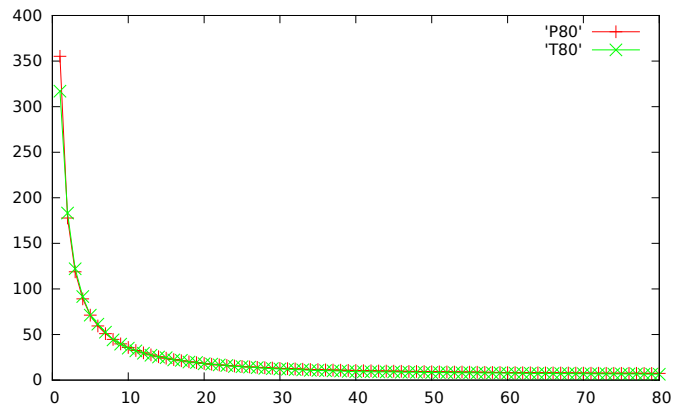
Performance on a 12 cores, ×2 machine



Wall clock time in sec. P12 is the “parallel” implementation, T12 is the pthreads implementation.

78

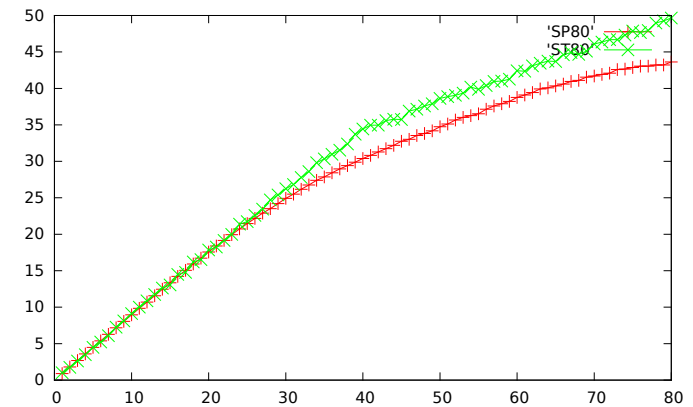
Performance on a 40 cores, ×2 machine



Wall clock time in sec. P80 is the “parallel” implementation, T80 is the pthreads implementation.

79

Speedup on a 40 cores, ×2 machine



Wallclock times are divided by the running time of our faster sequential implementation: `./fast.out -d3 18`. SP80 is the “parallel” implementation, ST80 is the pthreads implementation.

80