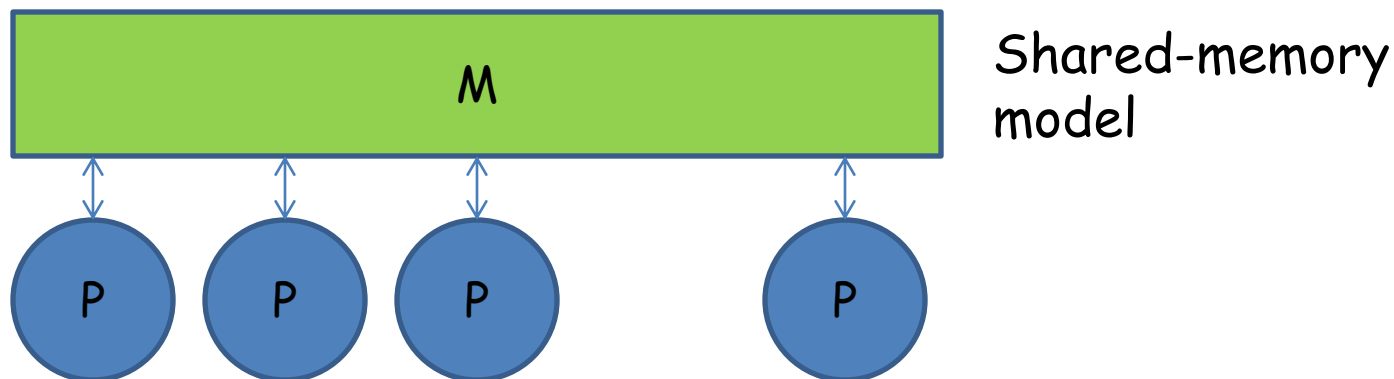


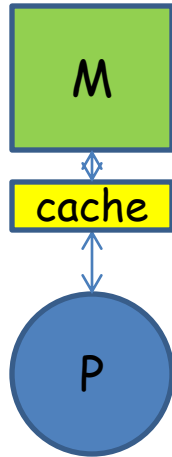
Introduction to Parallel Computing

Jesper Larsson Träff
Technical University of Vienna
Parallel Computing

Shared-memory architectures & machines



Naive, shared memory (programming) model: processors execute processes, processes are not synchronized, special methods for sharing memory between processes, NUMA



Cache: small, fast memory, close to processor, accessed main memory locations are stored temporarily in cache, reused when possible

Caches may help to alleviate/hide memory („von Neumann“) bottleneck

- Main memory: Gbytes, access times > 100 cycles
- Cache: Kbytes \rightarrow Mbytes, access times, 1-20 cycles

Typically 2-3 levels of caches in modern processors, and several special caches, TLB, victim cache, instruction cache, ...

Caches, recap.

Cache consists of a number of **lines** that stores **blocks** of memory. A **cache line** holds a block and additional status information (dirty/valid bit, tag)

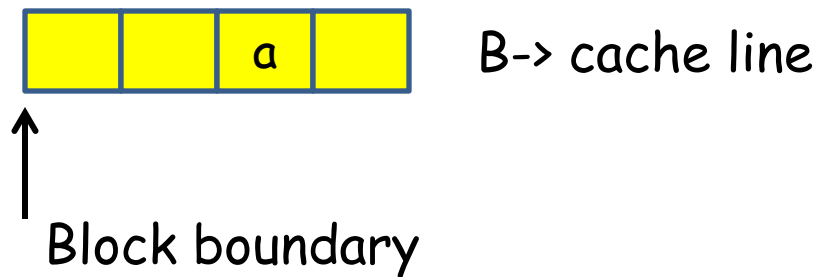
Typical block size: 64Bytes

Caches exploit and makes sense because of:

- **Temporal locality:** locations are typically used several times in close succession, several operations on same operand
- **Spatial locality:** when a location is addressed, typically locations close to it ($a+1$, $a+2$, ...) will be also be used

Properties of algorithms/programs, and **not always so**

Access to main memory in block size units B , aligned to block boundary



Memory **read** a :

if address a already in cache, reuse from there, if not read from memory through cache, evict previous line

Memory **write a**:

different possibilities. If a is already in cache, write overwrites;
if a is not in cache

- **Write allocate**: if a is not in cache, read a
- **Write non-allocate**: write directly to memory

- **Write-through cache**: each write is immediately passed on to memory (typically non-allocate)
- **Write back**: cache line block is written back when line is evicted (typically write allocate)

Address a :

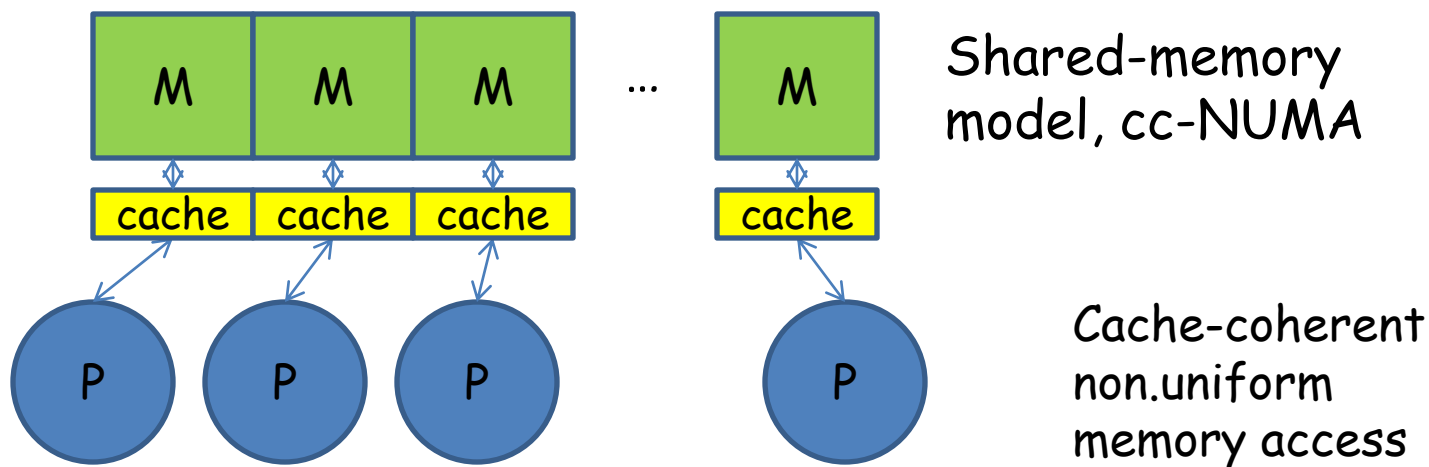
- If a can go into only one specific line of the cache: **directly mapped**
- If a can go into any line of the cache: **fully associative**
- If a can go into any of a small set of lines: **set-associative** (typically 2-way, 4-way)

Replacement policies for associative caches

- LRU: least recently used
- LFU : least frequently used

Typically, all maintained in hardware

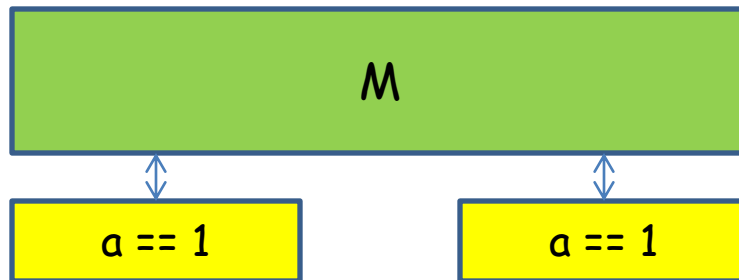
Multiprocessor/multi-core caches



Typically, several cores shares caches at some levels

Cache coherence

Processor/core 0 and 1 with private caches, both have read location a into cache. Processor 0 writes to a ?



$a = 7;$

$b = a; // ??$

Read by 1 occurs „after“ write by 0. If b is still 1, cache system is **not** coherent

Let the order of memory accesses to a specific **location a** be given by the program order

Cache is coherent if

1. If processor P writes to **a** at time t_1 and reads **a** at $t_2 > t_1$, and there are no other writes (by P or other) to **a** between t_1 and t_2 , then P reads the value written at t_1
2. If P1 writes to **a** at t_1 and another P2 reads **a** at $t_2 > t_1$ and no other P writes to **a** between t_1 and t_2 , then P2 reads the value written by P1 at t_1
3. If P1 and P2 writes to **a** at the same time, then either the value of P1 or the value of P2 is stored at **a**

Ad 1. Program order is preserved for each processor for locations that are not written by other processors

Let the order of memory accesses to a specific **location a** be given by the program order

Cache is coherent if

1. If processor P writes to **a** at time t_1 and reads **a** at $t_2 > t_1$, and there are no other writes (by P or other) to **a** between t_1 and t_2 , then P reads the value written at t_1
2. If P1 writes to **a** at t_1 and another P2 reads **a** at $t_2 > t_1$ and no other P writes to **a** between t_1 and t_2 , then P2 reads the value written by P1 at t_1
3. If P1 and P2 writes to **a** at the same time, then either the value of P1 or the value of P2 is stored at **a**

Ad 2. Here, t_1 and t_2 have to be „sufficiently“ separated in time. Updates by P1 must eventually become visible to the other processors

Let the order of memory accesses to a specific **location a** be given by the program order

Cache is coherent if

1. If processor P writes to **a** at time t_1 and reads **a** at $t_2 > t_1$, and there are no other writes (by P or other) to **a** between t_1 and t_2 , then P reads the value written at t_1
2. If P1 writes to **a** at t_1 and another P2 reads **a** at $t_2 > t_1$ and no other P writes to **a** between t_1 and t_2 , then P2 reads the value written by P1 at t_1
3. If P1 and P2 writes to **a** at the same time, then either the value of P1 or the value of P2 is stored at **a**

Ad 3. Writes are required to „**serialize**“. Either of the values simultaneously written will be stored. „Same time“ means „sufficiently close“ in time.

cc-NUMA systems (most multi-core and SMP nodes): cache coherent, non-uniform memory access

Cache coherence maintained by hardware at the **cache line level**.
Standard approaches and protocols:

- Update based
- Invalidation based

- Snooping/bus based
- Directory based

All: **expensive in hardware** („transistors“, „power“); can affect performance negatively

Sharing/false sharing

Cache coherence is maintained at the cache line level. Processor 0 updates y , processor 1 updates x (with e.g. $\&x == \&z[1]$, $\&y = \&z[2]$)



```
for (i=0; i<n; i++) y += i-1;
```

```
for (i=0; i<n; i++) x += 2*i;
```

Although x and y are different memory locations, each update will cause cache coherency traffic!! Because cache coherency is at the cache line level, x and y are **falsely shared**

Memory consistency

In what order do writes to different locations not necessarily in cache become visible in memory and to other processors?

Core 0:

```
x = 0;  
// ... some code  
x = 1;  
if (y==0) {  
    // body  
}
```

Core 1:

```
y = 0;  
// ... some code  
y = 1;  
if (x==0) {  
    // body  
}
```

x not in cache
of core 1, y not
in cache of
core 0

Can core 0 and core 1 both execute body of if-statement?

Core 0:

```
x = 0;
// ... some code
x = 1;
if (y==0) {
    // body
}
```

Core 1:

```
y = 0;
// ... some code
y = 1;
if (x==0) {
    // body
}
```

If $x=1$; $y=1$; appears at the same time, no cores execute body

If core 0 in body, then core 1 has executed $y=0$; but not $y=1$;
thus core 1 cannot enter body

Correct?

Only under sequential
consistency (or similar)

Sequential consistency: memory accesses of each processor are performed in program order; program result is as for some interleaving of the memory accesses of all processors

Sequential consistency is typically **not** guaranteed by modern multiprocessors:

- Caches, may delay writes
- Write buffers, may delay and/or reorder writes
- Memory network: may reorder writes
- Compiler: may reorder updates

Relaxed consistency models (see other lecture...) pose weaker constraints on hardware, may still allow correctness reasoning

In short:

To guarantee intended effect/correctness of a shared-memory multiprocessor program, special instructions that enforce memory updates to take effect may have to be used

Example:

memory fence(**f**) : completes all writes before the instruction and sets flag **f**

Another processor waiting for **f** will „know“ that all writes of the other processor before **f** was set will have been completed

Other approaches to alleviating memory bottleneck

- Prefetching: start loading operands well before use
- Multi-threading: when a thread („virtual processor“) issues a load, switch to another thread

Note: multi-threading requires explicitly parallel programs

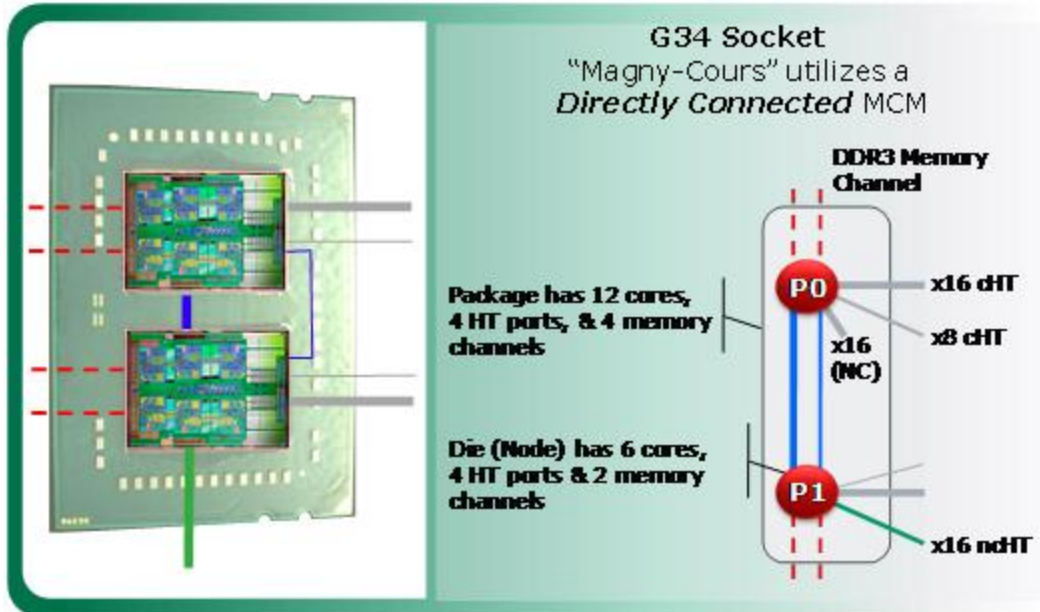
Note: both prefetching and multi-threading are **latency hiding** techniques. Memory bandwidth is still required for the number of outstanding memory requests

TU Wien parallel computing shared-memory node

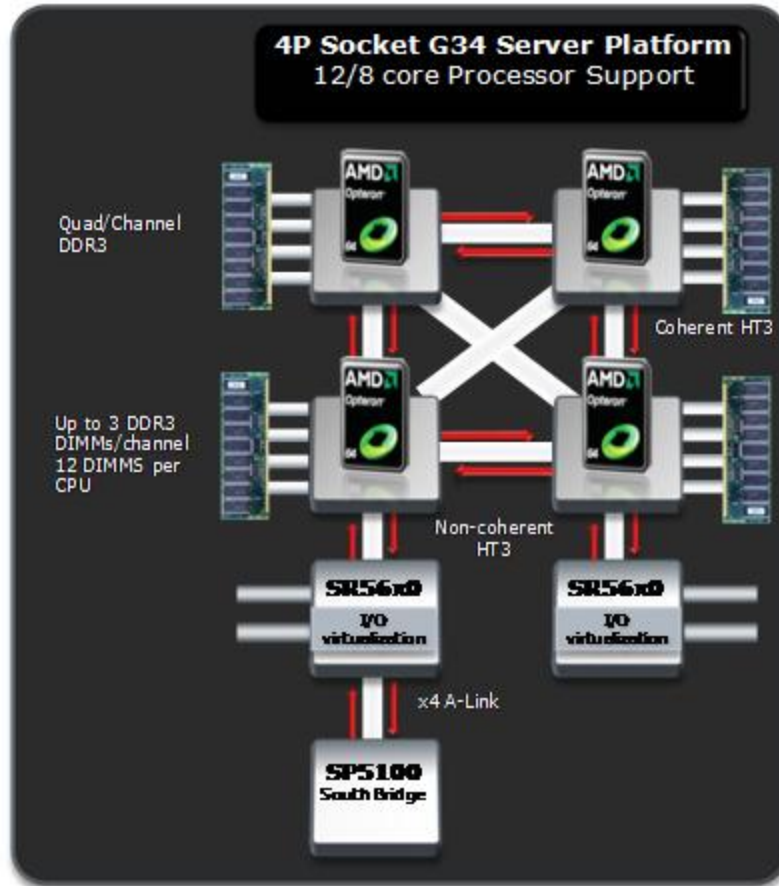
4xAMD „**magny cours**“ 12-core Opteron 6168 processors
128GByte main memory, 1.9GHz

- Per core L1 cache: 128KB
- Per core L2 cache 512KB
- Shared L3 cache 12288KB

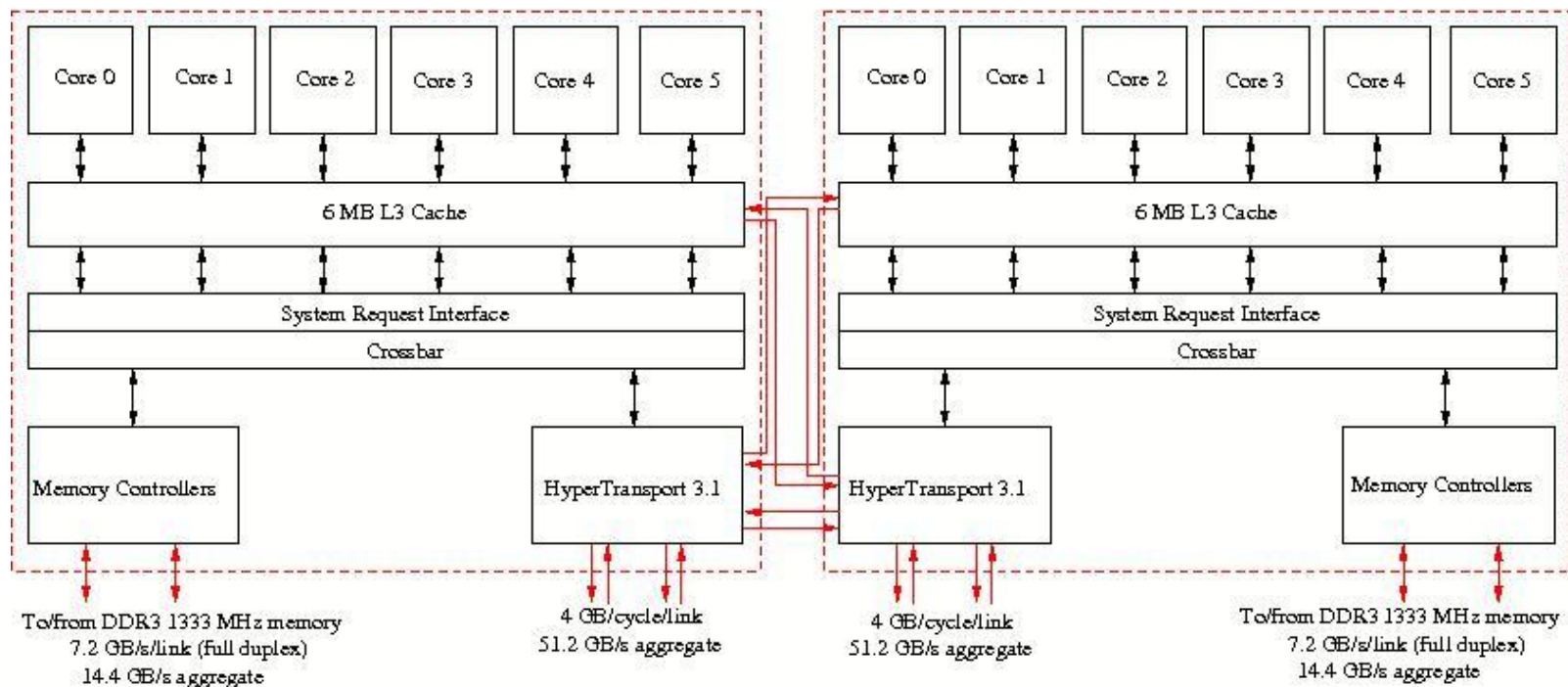
12 core = 2x6 cores, 2 dies on chip?



HT: HyperTransport - standardized processor-processor interconnect

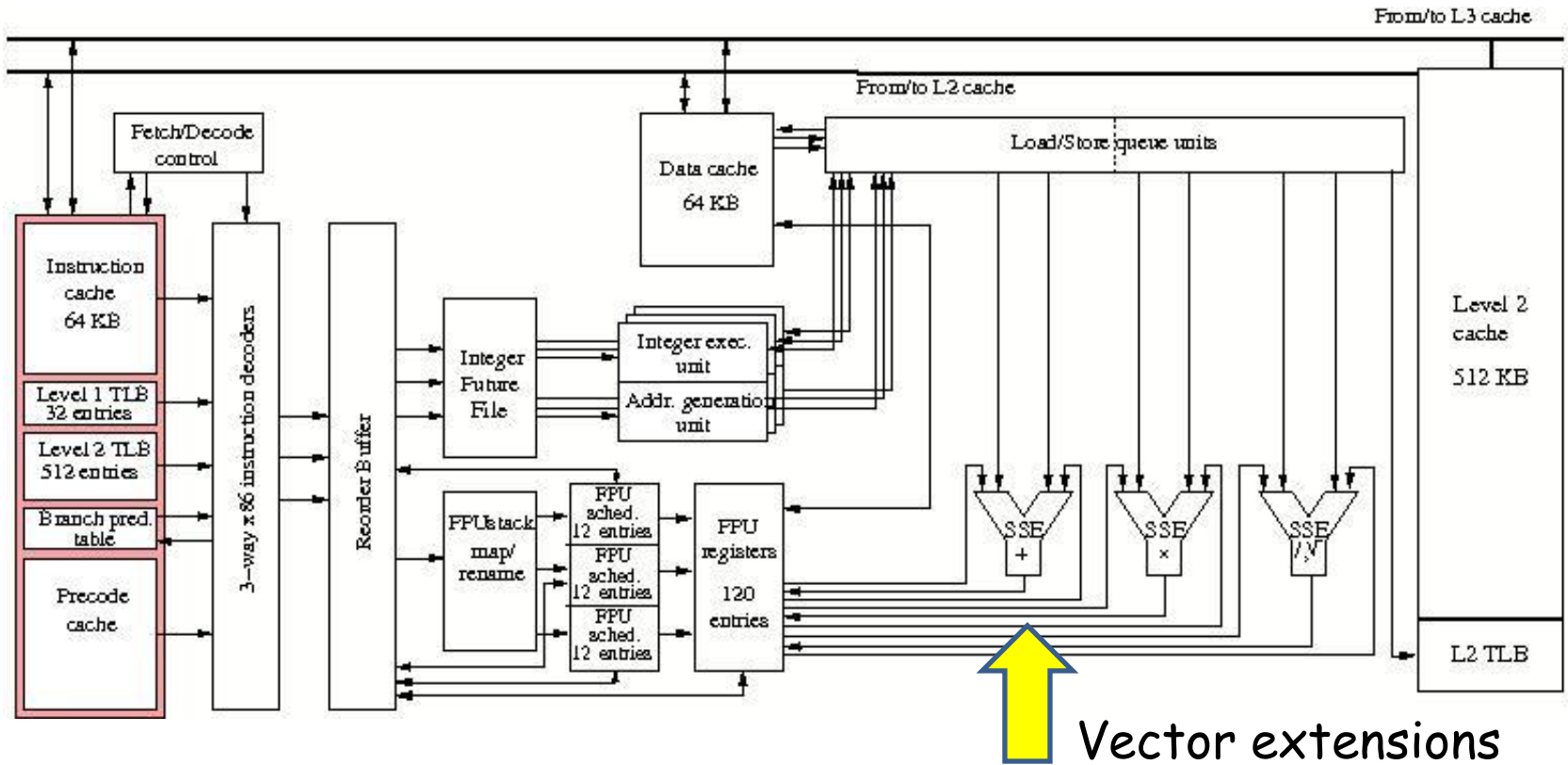


48-core shared-memory system from 4x12-core



Check-exercise: try to find the (superscalar) issue width? Peak performance? of the Opteron/Magny Cours processor

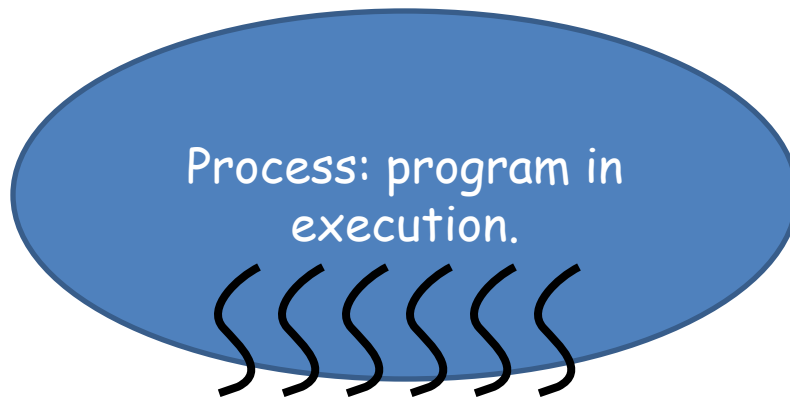
From University of Utrecht, EuroBen homepage: www.phys.uu.nl/eurben



L1 cache: 64KB data, 64KB instruction

Thread model

Thread: independent stream of instructions that **can be** scheduled by the OS. Typically, threads live inside an OS „process“, and shares all global information of the process (Thread: „smallest unit that can be independently scheduled“)



- UNIX process global information:
- File pointers
 - **Global variables**
 - Static variables
 - **Heap storage**

Per thread: local variables (stack), registers, „thread local storage“

POSIX threads, pthreads

POSIX: Portable Operating Systems Interface for uniX

Standard thread library API for UNIX (Linux etc.) since 1995:
IEEE/ANSI 1003.1c-1995

Official standard documents cost money; standard available as
man pages, internet, several tutorials and books

Low-level interface for C/UNIX thread programming

More modern thread model, including memory model: Java threads

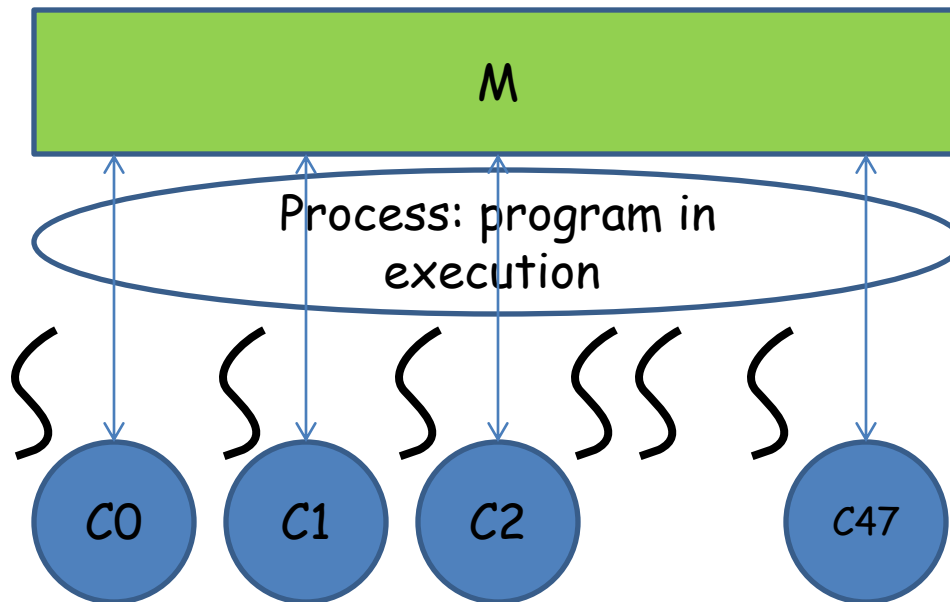
(p)threads „Programming model“

1. **Fork-join type parallelism**: a thread can „**spawn**“ (start) any number of new threads (up to system limitations), wait for threads to terminate
2. Initially one main („master“) thread is active. Code for thread is a procedure/function
3. Spawned threads are peers, any thread can wait for termination of any other thread
4. Threads are scheduled by the underlying system, **may** or **may not** run simultaneously, may or may not be scheduled to available processors/cores

5. No implicit synchronization between threads, threads progress independently, and asynchronously
6. Threads share process global data
7. Coordination mechanisms for protecting access to shared variables (locks, condition variables). All concurrent updates must be protected, otherwise program illegal, outcome undefined
8. ...

Pthreads: **no cost model, no memory model, ...**

Pragmatics (for **parallel computing**): runnable threads are expected to be scheduled to free cores. Scheduling and binding (mapping to specific core) can be influenced



pthread for C:

Main program is main thread, spawns off and waits for termination of additional threads. Thread code: C function

- Include header `<pthread.h>`
- All pthread types and functions prefixed by `pthread_`
- pthread functions return **error code**, or status information, **good practice to check!!** (not done here...)

Compile with

```
gcc -Wall -o pthreadshello pthreadshello.c -pthread
```

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

pthread_t: type of thread object (**opaque**), thread id returned here (pointer), must be allocated globally by spawning thread

```
static pthread_t newthread
```

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

`void *(start_routine) (void *)`: type template for the function to run as thread. Takes arguments via generic pointer, returns generic pointer, standard C convention

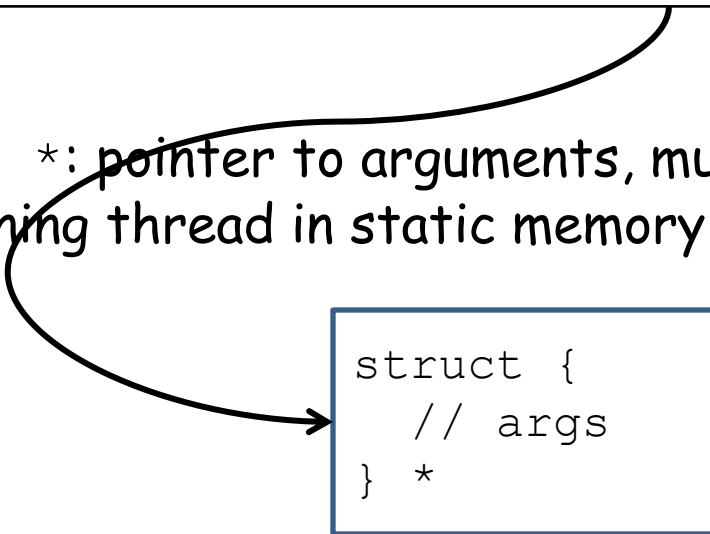
```
void *newcode(void *genericargs) {
    myarg_t realargs = (myarg_t*)genericargs;
    // work to be done by this thread
}
```


Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void *),
                  void *arg);
```

`void *`: pointer to arguments, must have been allocated by spawning thread in static memory (heap)



```
struct {
    // args
} *
```

Starting/spawning a thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void *),
                  void *arg);
```

Execution of thread can be influenced by attributes:
stacksize, scheduling properties, ... NULL, or

Not this lecture

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Finalizing/terminating thread

```
#include <pthread.h>

void pthread_exit(void *status);
```

Terminates thread, pointer to return status can be supplied;
return status can be caught by joining thread

Joining threads

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **status);
```

Main thread

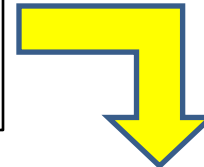
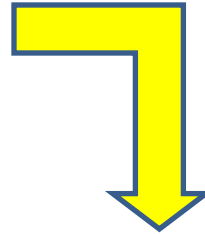
```
int main () {  
  pthread_t t;  
  pthread_create(&t,...);  
  ... // main continues  
}
```

New thread

```
threadcode() {  
  // ...  
  pthread_exit(NULL);  
}
```

Some other thread

```
pthread_join(t,NULL);
```



A small example

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

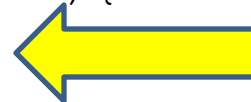
// pthreads header
#include <pthread.h>

// global state; here read-only - don't do this..
int threads_glob;

void *something(void *argument) {
    int rank = (int)argument;

    printf("Thread rank %d of %d responding\n",
           rank, threads_glob);
    pthread_exit(NULL);
}
```

C style: cast void *
argument back to
intended type



A small example

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

// pthreads header
#include <pthread.h>

// global state; here read-only - don't do this..
int threads_glob;

void *something(void *argument) {
    int rank = (int)argument;

    printf("Thread rank %d of %d responding\n",
           rank, threads_glob);
    pthread_exit(NULL);
}
```

Here misuse of
pointer to store rank

```
int main(int argc, char *argv[]){
    int threads, rank;
    int i; pthread_t *handle;

    threads = 1;
    for (i=1; i<argc&&argv[i][0]!='-'; i++) {
        if (argv[i][1]=='t')
            i++,sscanf(argv[i],"%d",&threads);
    }
    threads_glob = threads;
    // number of threads read and stored globally
    handle = (pthread_t*)malloc(threads*sizeof(pthread_t));
    // fork the threads
    for (i=0; i<threads; i++) {
        pthread_create(&handle[i],NULL,
            something,(void*)i);
    }
}
```

Getting
command line
arguments

Local scalar variable cast into generic void
pointer, correct, but dangerous misuse

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

// pthreads header
#include <pthread.h>

// global state; here read-only - don't do this..
int threads_glob;

void *something(void *argument) {
    int rank = *(int*)argument;

    printf("Thread rank %d of %d responding\n",
           rank, threads_glob);
    pthread_exit(NULL);
}
```

Better: cast and
deref




```
int main(int argc, char *argv[]){
    int threads, rank;
    int i;  pthread_t *handle;

    threads = 1;
    for (i=1; i<argc&&argv[i][0]=='-'; i++) {
        if (argv[i][1]=='t')
            i++,sscanf(argv[i],"%d",&threads);
    }
    threads_glob = threads;
    // number of threads read and stored globally

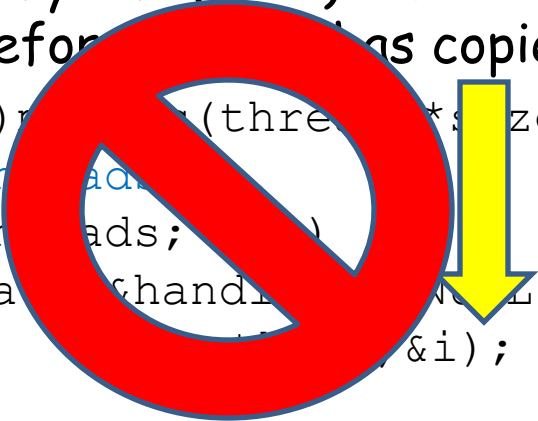
    handle =
        (pthread_t*)malloc(threads*sizeof(pthread_t));
    // fork the threads
    for (i=0; i<threads; i++) {
        pthread_create(&handle[i],NULL,
            something,&i);
    }
}
```

Problem?

```
int main(int argc, char *argv[]){
    int threads, rank;
    int i; pthread_t *handle;

    threads = 1;
    for (i=1; i<argc&&argv[i][0]!='-'; i++) {
        if (argv[i][1]=='t')
            i++,sscanf(argv[i],"%d",&threads);
    }
    threads_glob = threads;
    // number of threads read and stored globally
    handle =
        (pthread_t*)calloc(threads,sizeof(pthread_t));
    // fork the threads
    for (i=0; i<threads; i++)
        pthread_create(&handle[i],NULL,
            &i);
}
```

Only one (local) variable, may be overwritten
before it has copied into local



Problem?

Example:

a value (storage of i) is overwritten by one thread, **may** (or **may not**) happen before the other threads have read intended value. Program outcome dependent on relative timing of threads. **Bad, unintended non-determinism...**

Race condition:

Outcome of parallel program execution is dependent on the relative timing of the updates by processors/threads

```
int main(int argc, char *argv[]){
    int threads, *rank;
    int i; pthread_t *handle;

    // ... get the number of threads
    handle =
        (pthread_t*)malloc(threads*sizeof(pthread_t));
    rank = (int*)malloc(threads*sizeof(int));
    // fork the threads
    for (i=0; i<threads; i++) {
        rank[i] = i;
        pthread_create(&handle[i],NULL,
                      something,&rank[i]);
    }
    // join the threads again
    for (i=0; i<threads; i++) pthread_join(handle[i],NULL);
    free(rank); free(handle);
    return 0;
}
```

Own location for each thread, no overwrite

Free storage nicely

Wait for threads to terminate

```
#define NDEBUG  
// assertion checking disabled
```

Checking return codes with assertions

Enables assertion
checking, macro
`assert(expr);`

```
#include <assert.h>  
  
int main(int argc, char *argv[]) {  
    int threads, *rank;  
    int i; pthread_t *handle;  
  
    // ... get the number of threads, allocate  
  
    // fork the threads  
    for (i=0; i<threads; i++) {  
        rank[i] = i;  
        errcode = pthread_create(&handle[i], NULL,  
                                something, &rank[i]);  
        assert(errcode==0);  
    }  
    // ...  
}
```

Assertion `errcode==0`
expected to evaluate to
true ($\neq 0$), otherwise abort

Potential problem: sequential spawning of threads can limit scalability (Amdahl).

In general: thread creation can be expensive

```
for (i=0; i<threads; i++) {  
    rank[i] = i;  
    pthread_create(&handle[i], NULL,  
                  something, &rank[i]);  
}  
// join the threads again  
for (i=0; i<threads; i++) pthread_join(handle[i], NULL);
```

Fix: spawn recursively

`pthread_t` thread identifiers are opaque; normally user gives thread „identity“ (as in example), a thread can inquire its own `pthread_t` id; `pthread_t` id's can be compared

```
#include <pthread.h>

pthread_t pthread_self(void);
```

```
#include <pthread.h>

int pthread_equal(pthread_t thread_1,
                  pthread_t thread_2);
```

Explicit parallelization of data parallel loop

```
for (i=0; i<n; i++) {  
    a[i] = f(i);  
}
```

Thread i (on core i) performs

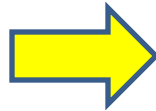
```
for (i=start; i<end; i++) {  
    a[i] = f(i);  
}
```

$start = i * n / threads$
 $end = (i+1) * n / threads$

Explicit parallelization of data parallel loop

```
for (i=0; i<n; i++) {  
    a[i] = f(i);  
}
```

Arguments struct



```
typedef struct {  
    int *array;  
    // pointer shared, global data  
    int start, end;  
    int rank; // threads rank  
} rankindex_t;
```

```
loopblock(void *what)  
{  
    rankindex_t *ix = (rankindex_t*)what;  
    int *a = ix->array;  
    int i, start=ix->start, end=ix->end ;  
  
    for (i=start; i<end; i++) a[i] = f(i);  
}
```

Function for
loop block

Example: matrix-vector product

$y = x^*A$, $n \times m$ matrix x , n vector A

```
for (i=0; i<n; i++) {  
    y[i] = 0;  
    for (j=0; j<m; j++) {  
        y[i] += x[i][j]*A[j];  
    }  
}
```

Nested loop

Parallelized by tiling outer loop

```
for (i=rank; i<n; i+=threads) {  
    y[i] = 0;  
    ...  
}
```

Each thread rank
handles every
threads'th index

Thread rank:

```
for (i=rank; i<n; i+=threads) {  
    y[i] = 0;  
    for (j=0; j<m; j++) {  
        y[i] += x[i][j]*A[j];  
    }  
}
```

Problem?

y[0]	= 0;
y[1]	= 0;
y[2]	= 0;
y[3]	= 0;

y values go into (local) caches

Thread rank:

```
for (i=rank; i<n; i+=threads) {  
    y[i] = 0;  
    for (j=0; j<m; j++) {  
        y[i] += x[i][j]*A[j];  
    }  
}
```

y[0]	+= x[i][j]...;
y[1]	+= x[i][j]...;
y[2]	+= x[i][j]...;
y[3]	+= x[i][j]...;

False sharing: updates on y causes either cache update traffic or invalidates/memory reads

Thread rank:

```
for (i=rank*n/p; i<(rank+1)*n/p; i++) {  
    y[i] = 0;  
    for (j=0; j<m; j++) {  
        y[i] += x[i][j]*a[j];  
    }  
}
```

Solution?

Exercise: test effects of false sharing (best and worst cases) on TU Wien parallel computing shared-memory node, with explicit thread affinity

Binding threads to cores

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

int pthread_setaffinity_np(pthread_t thread,
                           size_t cpusetsize,
                           const cpu_set_t *cpuset);

int pthread_getaffinity_np(pthread_t thread,
                           size_t cpusetsize,
                           cpu_set_t *cpuset);
```

`_np`: non-portable, non-standard extension to pthreads (but commonly supported in some form)

Thread will be migrated to one of the cores in `cpuset`

Coordination constructs for avoiding race conditions

- Locks/mutex'es - for ensuring mutual exclusion
- Condition variables
- Advanced, non-standard features: semaphores, barriers, spin locks

Note: these are all classical **concurrent computing** constructs. Some classical algorithms to solve the problems under weak architecture assumptions: Dekker's algorithm, Lamport's bakery, ...

Caution: the constructs were developed for few resources, **not** necessarily sufficient **for highly parallel, scalable programming**

Critical section:

Code manipulating shared resources, that must **not** be concurrently manipulated by other active entities (threads, processes, ...)

Shared resources: simple variables, data structures, devices

Mutual exclusion property/algorithm: at most one thread in given critical section

Pthread „model“: it is not allowed to update shared variables outside of critical sections. The lock constructs shall ensure a consistent view of memory.

Locks

Lock: shared object between any number of threads.

Lock state: **locked** (acquired), or **unlocked** (released)

At most one thread can acquire the lock, must release after use.
When a thread attempts to acquire a lock that is already acquired by another thread it is blocked, and waits until the lock is released

If any thread that is waiting to acquire a lock is eventually granted the lock, the lock is called **fair!!**

Pthread lock is called **mutex**, type `pthread_mutex_t`

Static allocation and initialization with

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Dynamically allocated mutexes

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutex_attr *attr);
```

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Locking and unlocking

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Unsafe program, what is the intended value of x for thread 0 and 1?

$x = 0;$

Thread 0:

$a = x;$

Thread 1:

$b = x;$

Thread 2:

$x = c;$

Race condition: depends on relative timing of threads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
lock(&lock);  
a = x;  
unlock(&lock);
```

Thread 1:

```
lock(&lock);  
b = x;  
unlock(&lock);
```

Thread 2:

```
lock(&lock);  
x = c;  
unlock(&lock);
```

Mutual exclusion ensured - enforced by locking

Both read and write accesses to x need to be protected by the lock mutex

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
lock(&lock);  
a = x;  
unlock(&lock);
```

Thread 1:

```
lock(&lock);  
b = x;  
unlock(&lock);
```

Thread 2:

```
lock(&lock);  
x = c;  
unlock(&lock);
```

Mutual exclusion ensured - enforced by locking

Note: pthread locks are **not fair**, **no guarantee** that a thread trying to acquire a lock will **eventually** acquire it

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
lock(&lock);  
lock(&lock);  
a = x;  
unlock(&lock);
```

Thread 1:

```
lock(&lock);  
b = x;  
unlock(&lock);
```

Thread 2:

```
lock(&lock);  
x = c;  
unlock(&lock);
```

Deadlock!

Deadlock: two or more threads are in a situation where they dependently on each other cannot progress. **Deadlock will eventually proliferate to all threads**

What about this?

Thread 0:

`a = f(x);`

Thread 1:

`b = f(x);`

Thread 2:

`c = f(y);`

No apparent races, independent evaluation of some function f

OK?

Depends on f , must be such that it can indeed be executed concurrently: „**tread safe**“

Thread safety

Tautological definition: a function is thread-safe if it can be **executed concurrently** by any number of threads and will always produce **correct results**

Non-thread safe functions are

1. Functions that do not protect (write access) to shared variables
2. Functions that keep state over successive invocations (`static` variables).
3. Functions that return pointers to `static` variables
4. Functions that call thread-unsafe functions

Careful with functions supplied by other party, e.g. system functions

Example: `rand()` keeps state internally in static variables, notoriously **not** thread safe

Some system functions are made thread safe by locking. Can have undesirable effects - serialization slowdown, deadlock

More on locks

Testing and getting lock/non-blocking lock

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

If `mutex` is not held by other thread, lock acquired; if already held by other thread `EBUSY` is returned, calling thread is not blocked

Dead-locks:

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
```

Thread 0:

```
...  
pthread_mutex_lock(&lock1);  
pthread_mutex_lock(&lock2);  
...
```

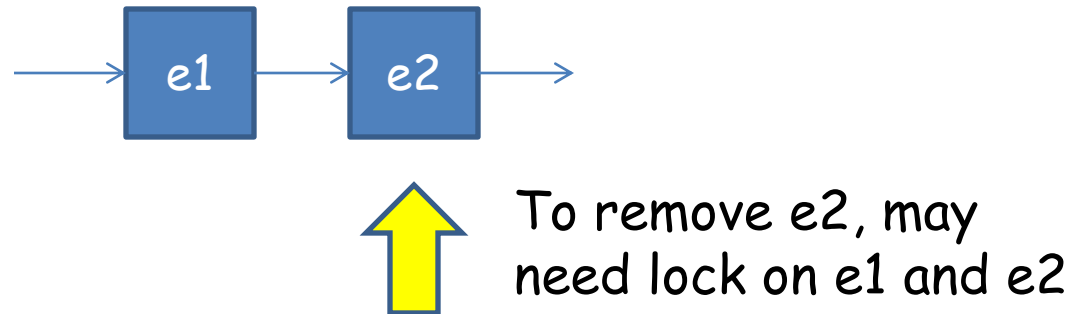
Thread 1:

```
...  
pthread_mutex_lock(&lock2);  
pthread_mutex_lock(&lock1);  
...
```

Can - and will - lead to deadlock!!

Beware: even the most „unlikely“ deadlock situation will eventually happen! **Design correct programs...**

Multiple locks, example: list processing



Problem with locks: code for different threads may have been written with different locking conventions, by different people, at different times...

More flexible locks: reader/writer locks

Allow several threads to acquire lock for reading shared variables, single thread to acquire for writing

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
```

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

```
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
```

Thread 0:

```
rdlock(&lock);  
a = x;  
unlock(&lock);
```

Thread 1:

```
rdlock(&lock);  
b = x;  
unlock(&lock);
```

Thread 2:

```
wrlock(&lock);  
x = c;  
unlock(&lock);
```

Thread 0 and 1 can both enter their critical section simultaneously, thread 2 can only alone be in its critical section


```
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
```

Thread 0:

```
rdlock(&lock);  
a = x;  
unlock(&lock);
```

Thread 1:

```
rdlock(&lock);  
b = x;  
unlock(&lock);
```

Thread 2:

```
wrlock(&lock);  
x = c;  
unlock(&lock);
```

Note: pthread locks are **not fair**, **no guarantee** that a thread trying to acquire a lock will **eventually** acquire it

More lock flexibility: condition variables

Thread may temporarily relinquish lock, and wait (suspend) for condition-signal

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
                    const pthread_cond_attr *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Wait for signal on condition variable inside critical section

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mutex);
```

Thread suspended (waits), lock is temporarily relinquished. When thread is later resumed (woken up) by a signal from some other thread, it has again acquired lock

Good practice: recheck whether wait-condition is fulfilled

Deadlock: threads mutually wait on some condition, no thread signals

Wait for signal on condition variable inside critical section

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mutex);
```

Thread suspended (waits), lock is temporarily relinquished. When thread it later resumed (woken up) by a signal from some other thread, it has again acquired lock

Good practice: recheck wheter wait-condition is fulfilled.

There can be **spurious wakeups** - threads signaled wrongly or getting a signal spuriously from pthreads

Signal some waiting thread

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

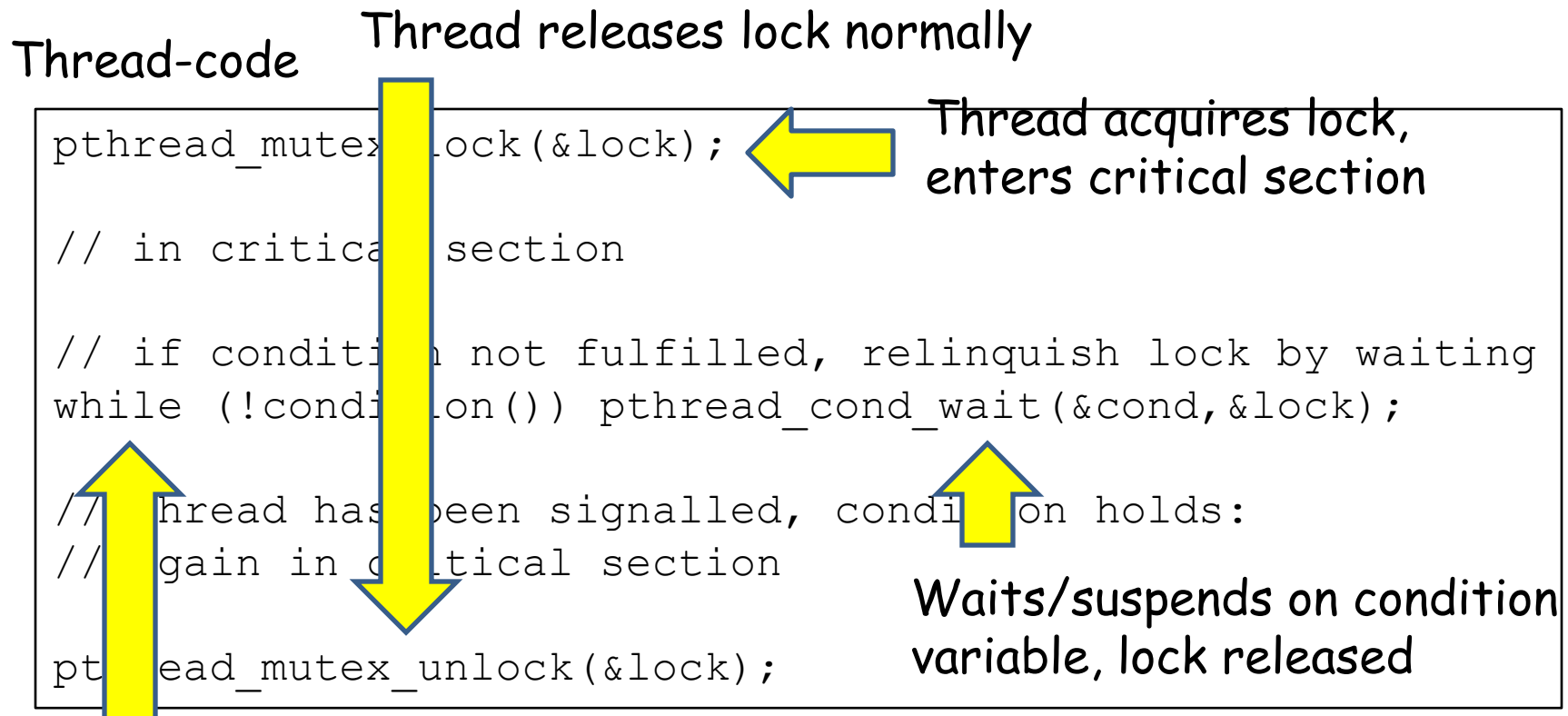
Signal all waiting threads

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

If more than one thread is waiting, which gets signal is undetermined (can be influenced by attributes); broadcast signals one after another

Standard condition variable pattern



After signal, lock is again acquired (mutual exclusion!),
condition can be rechecked

Example: readers-writers lock with condition variables

Idea:

Keep track of number of readers, pending writers, whether there is a writer, condition variables to suspend readers and writers trying to acquire lock, standard mutex for ensuring mutual exclusion to the shared data structure

```
typedef struct {
    int readers;
    int waiting, writer;
    pthread_cond_t read_ok, write_ok;
    pthread_mutex_t gateway;
} rwlock_t;
```

Init function: no readers, no writer, no pending; initialize mutex and condition variables

Acquire reading lock

```
void rwlock_rlock(rwlock_t *rwlock)
{
    pthread_mutex_lock(&rwlock->gateway);
    while (rwlock->waiting>0 || rwlock->writer) {
        pthread_cond_wait(&rwlock->read_ok,
                        &rwlock->gateway);
    }
    rwlock->readers++;
    pthread_mutex_unlock(&rwlock->gateway);
}
```


Acquire single writing lock

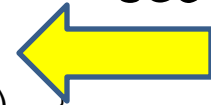
```
void rwlock_wlock(rwlock_t *rwlock)
{
    pthread_mutex_lock(&rwlock->gateway);
    rwlock->waiting++;
    while (rwlock->writer || rwlock->readers > 0) {
        pthread_cond_wait(&rwlock->writer_ok,
                        &rwlock->gateway);
    }
    rwlock->waiting--;
    rwlock->writer = 1;
    pthread_mutex_unlock(&rwlock->gateway);
}
```

Unlock: wake up threads waiting to acquire lock

```
void rwlock_unlock(rwlock_t *rwlock)
{
    pthread_mutex_lock(&rwlock->gateway);
    if (rwlock->writer) rwlock->writer = 0;
    else rwlock->readers--;
    pthread_mutex_unlock(&rwlock->gateway);

    // resume threads waiting to acquire
    if (rwlock->readers==0&&rwlock->waiting>0) {
        pthread_cond_signal(&rwlock->writer_ok);
    } else pthread_cond_broadcast(&rwlock->reader_ok);
}
```

Signal can
be sent
outside
critical
section



But actually **race**: readers/waiting can be changed by other threads after unlock

Correctness:

Establish (prove) invariants: readers counts the number of threads having acquired read lock, writer is true if and only if a process has acquired write lock, etc.

Note: the original implementation from <book?> was
not correct at all

(Un)Fairness properties:

Threads acquiring write lock can starve threads wanting to acquire read lock (??)

- Newer writer can starve older writer
- Newer reader can acquire lock before older reader - or writer

Unlock: wake up threads waiting to acquire lock

```
void rwlock_unlock(rwlock_t *rwlock)
{
    pthread_mutex_lock(&rwlock->gateway);
    if (rwlock->writer) rwlock->writer = 0;
    else rwlock->readers--;

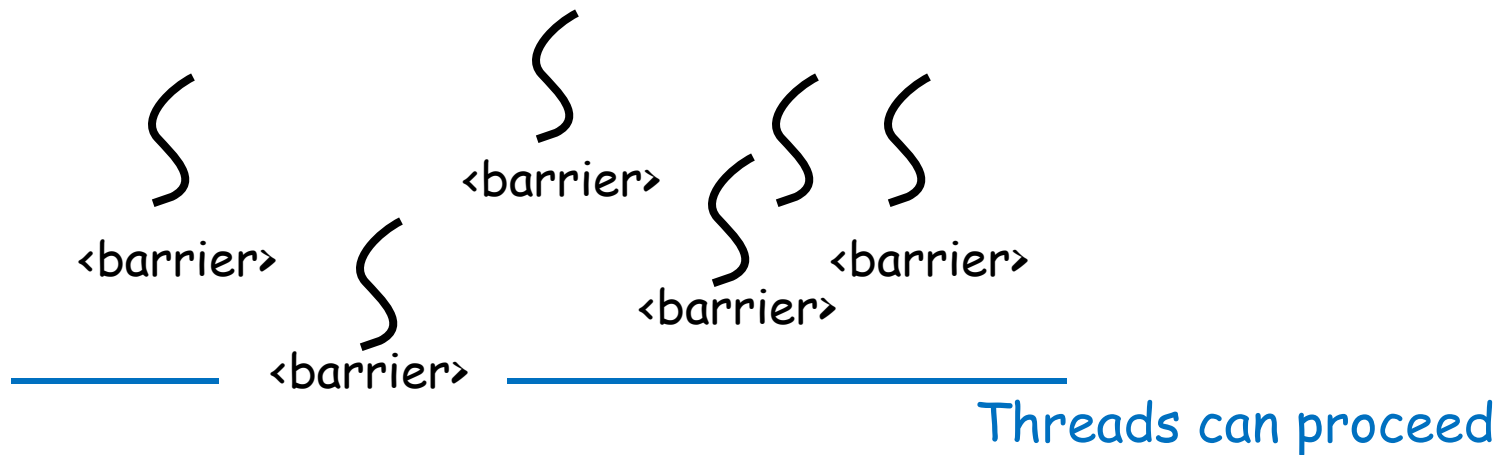
    // resume threads waiting to acquire
    if (rwlock->readers==0&&rwlock->waiting>0) {
        pthread_cond_signal(&rwlock->writer_ok);
    } else pthread_cond_broadcast(&rwlock->reader_ok);

    pthread_mutex_unlock(&rwlock->gateway);
}
```

Thread signals but keeps lock; signals sent after lock release

Example: Barrier synchronization with condition variables

Each thread execution `<barrier>` shall wait until all/some number of threads have executed `<barrier>`



```
typedef struct {
    int tc; // thread count
    pthread_cond_t barrier_ok;
    pthread_mutex_t barwait;
} barrier_t;
```

Naive barrier

Also from <book?>

```
void barrier(barrier_t *b, int tc)
{
    pthread_mutex_lock(&b->barwait);
    b->tc++;
    if (b->tc==tc) {
        b->tc =0;
        pthread_cond_broadcast(&b->barrier_ok);
    } else pthread_cond_wait(&b->barrier_ok, &b->barwait);
    pthread_mutex_unlock(&b->barwait);
}
```

Note:

1. This barrier implementation is not scalable, $O(p)$
2. Probably not safe on spurious wake ups
3. Other problems

Fixes:

1. Tree structured barrier
2. Extra flag

[Mellor-Crummey, Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM TOPLAS* (1): 21-65 (1991)]

Spin locks - specific implementation for performance

```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock,
                      int pshared);
```

Mutex semantics, but different
pragmatics/implementation/performance


```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Pragmatics/implementation:

thread waiting to acquire lock does not suspend, waits for lock release by „spinning“ on flag

Contrast: mutex locks, thread blocking on lock may be suspended (put to sleep) by OS, and resumed when lock is released

```
#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Hint:

Hand-implemented locks, or other data structure requiring waiting - useful to suspend thread and yield processor to some other thread

```
#include <sched.h>

int sched_yield(void);
```

Spinlocks: possibly faster on dedicated (parallel) applications on dedicated systems, expensive OS suspension not required.

On overloaded systems - more threads than cores/processors - spinlocks can behave very badly

Portability caveat:

to enforce „spinning“ behavior, explicit use of spinlocks needed, program needs rewrite/recompilation/conditional compilation.

Why not controlled by mutex-attributes?

Example: coming to terms without locks

Task: find all primes between 2 and 10^9

Idea: first independently, and in parallel, check all 10^9-1 candidates

Observation: check very fast for some numbers - those with a small prime factor; also, the number of primes in different intervals differ, by prime number theorem

Note: for illustration purposes only, for better ideas see [Crandall, Pomerance: Prime numbers. Springer, 2002]

Statically scheduled data parallel loop will likely lead to load imbalance

```
for (i=2; i<1000000000; i++) {  
    if (isPrime(i)) printf(„Found %d\n“, i);  
}
```

Static schedule: each thread executes block of $1000000000/p$ successive iterations

If a few of the processors execute only the expensive `isPrime` checks, T_{par} will be close to T_{seq} , no Speedup

Better solution: use a shared, global counter

```
int i = 0; // shared global

// Thread i code
while (i < 1000000000) {
    int j = i; i++; // thread gets next value of i
    if (isPrime(j)) printf(„Found %d\n“, j);
}
```

Problem?

`i++;` translates into

```
tmp = i;  
tmp = tmp+1;  
i = tmp;
```

Thread 0:

```
tmp = i;  
  
tmp = tmp+1;  
i = tmp;
```

Thread 1:

```
tmp = i;  
  
tmp = tmp+1;  
i = tmp;
```

Both threads
reads same
value for `i`

`i` incremented by 1 only - race condition!!

Better solution: use a shared, global counter

```
int i = 0; // shared global

// Thread i code
while (i < 1000000000) {
    int j;
    pthread_mutex_lock(&counter);
    j = i; i++;
    pthread_mutex_unlock(&counter);
    if (isPrime(j)) printf("Found %d\n", j);
}
```

Problem?

Better solution: use a shared, global counter

Thread 0

```
int i = 0; // shared global
```

```
// Thread i code
```

```
while (i < 1000000000) {
```

```
    int j;
```

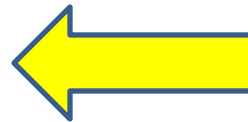
```
    pthread_mutex_lock(&counter);
```

```
    j = i; i++;
```

```
    pthread_mutex_unlock(&counter);
```

```
    if (isPrime(j)) printf(„Found %d\n“, j);
```

```
}
```



Thread 0 acquired lock,
may be interrupted for
arbitrarily long time; no
progress

Better solution: use a shared, global counter with atomic increment

```
int i = 0; // shared global

// Thread i code
while (i < 1000000000) {
    int j = fetch_and_inc(&i); // return value of i, inc
    if (isPrime(j)) printf(„Found %d\n“, j);
}
```

Correct. Threads can
always progress

Example of lock-free algorithm: each thread will always be able
to progress - no matter what other threads are doing

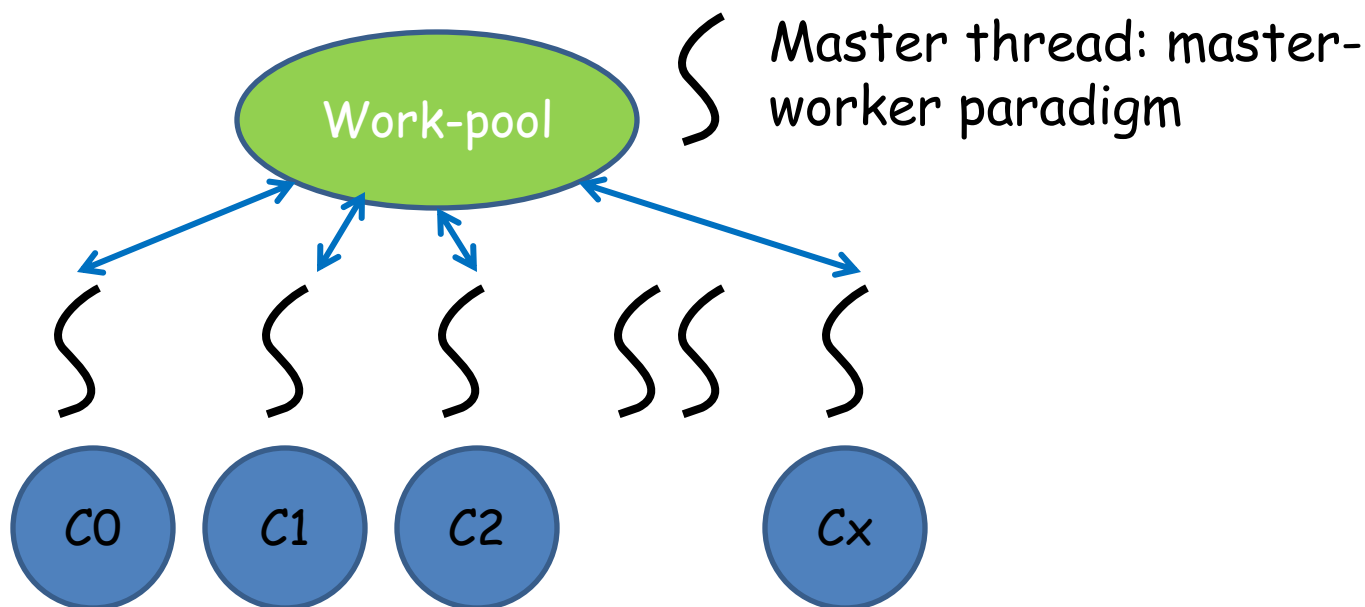
Atomic instructions in modern multi-core processors

- `fetch-and-inc(a)`: atomically return old value of `a`, increment
- `fetch-and-dec(a)`: atomically return old value of `a`, decrement
- `fetch-and-add(a,x)`: atomically return old value of `a`, add `x` to `a`
- `test-and-set/compare-and-swap (e,u,a)`: if content of `a` is equal to `e`, replace content of `a` with `u`, atomically
- `LL/SC`

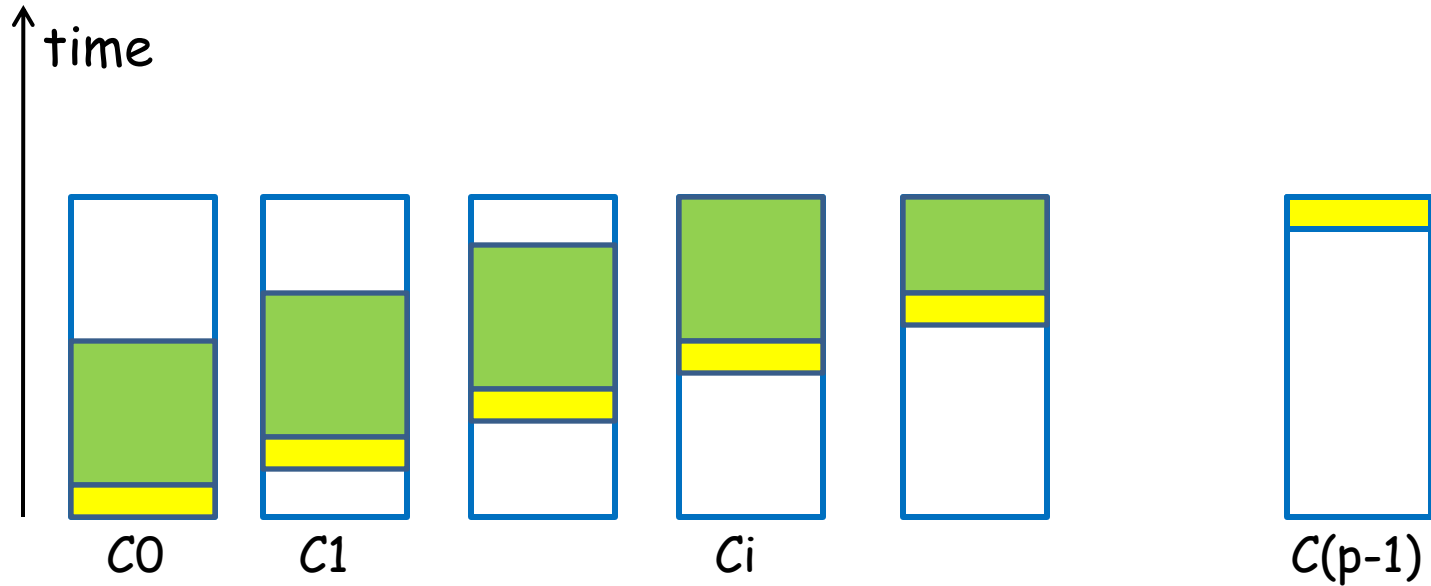
See: [Herlihy, Shavit: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008]


Work-pools, master-worker paradigm

„Master maintains pool of work, workers ask for work, execute, return new work/results to master, until all done“



Master/Work-pool possible scalability bottleneck



 Getting work: explicitly asking master, or accessing shared data structure

Implementation sketch, work executed in generated order

Use deque data structure as work-pool

Threads:

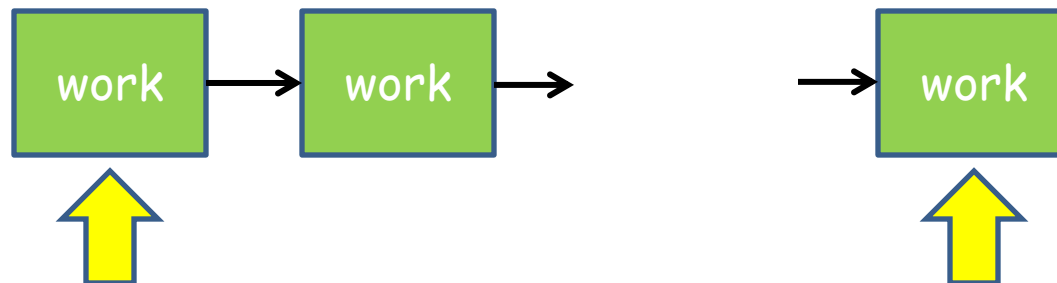
1. Acquire mutex, check list, if non-empty take from front, otherwise wait on condition variable.
2. Execute work.
3. New work: acquire mutex, insert at end of deque, wake up waiting threads

Until termination

General work-task structure

```
typedef struct work {  
    void (*routine) (void *args);  
    void *args;  
    struct work *next;  
} work_t;
```

Work pool: linked list, first and last element



Task parallel algorithms use work-pool-like implementation to keep threads busy executing tasks

```
void QuickSort(int x[],n) {
    if (n<=1) return;

    pivot = choosepivot(x,n); // x[pivot] is pivot element
    ix = partition(x,pivot); // ix is index of pivot after
    spawn QuickSort(x,ix); // recurse
    spawn QuickSort(x+ix+1,n-1-ix);
}
```

← Spawned task may execute in parallel on other core

With linear partition and optimal pivot parallel time is $O(n+n/2+n/4+\dots) = O(n)$ - with p $O(\log n)$ cannot do better

Problems:

1. **Centralized resource**, bad for scalability
2. **Locks**: thread updating shared resource can **lock out** all other threads indefinitely

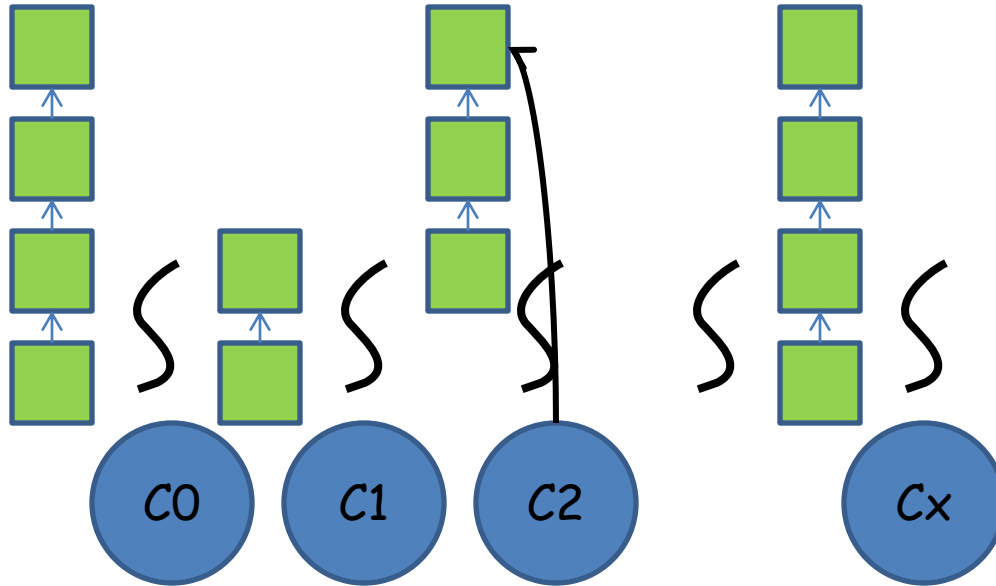
Solutions:

Work-stealing: Cilk, TBB, ...

1. **Local task queues**, a thread primarily uses local queue, when empty steals some work from some other thread's queue
2. **Lock-free data structures** enabling a thread always to either make progress by itself, or ensure that some other thread is making progress

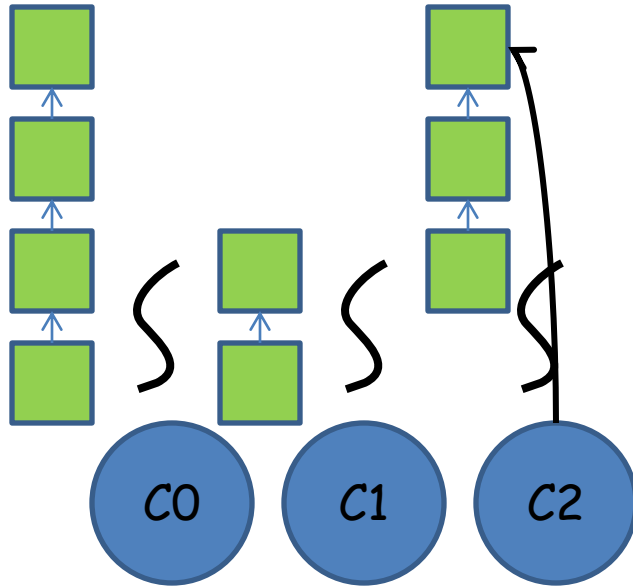
[Robert D. Blumofe, Charles E. Leiserson: Scheduling Multithreaded Computations by Work Stealing. J. ACM 46(5): 720-748 (1999)]

Local
dequeues

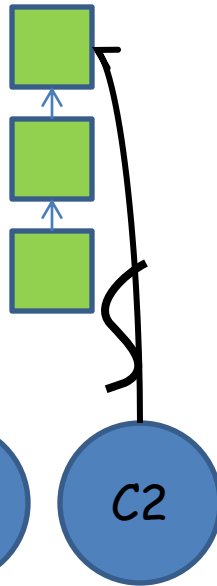


Put new work in
local queue

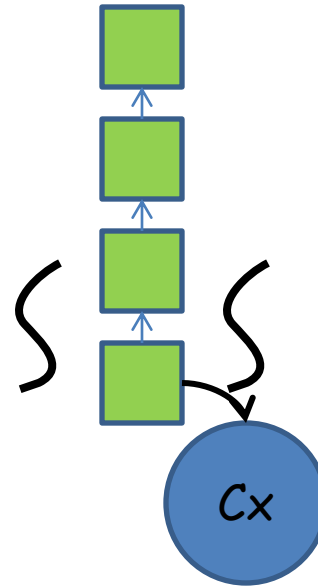
Local
dequeues



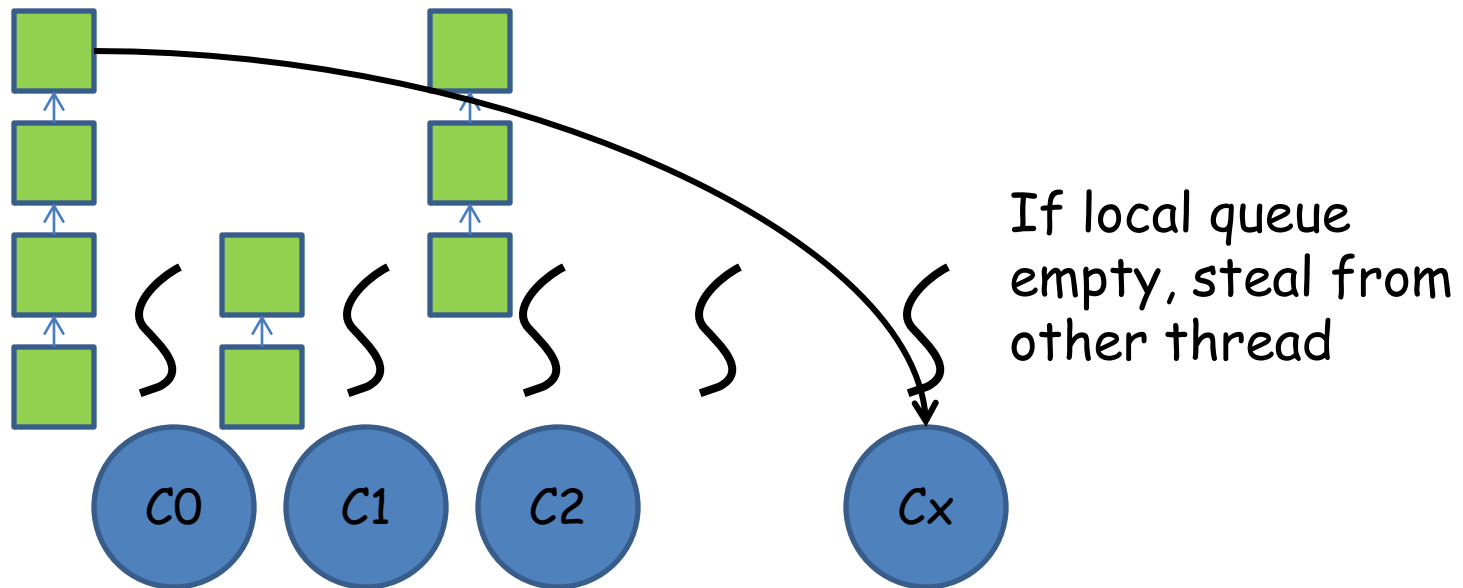
Put new work in
local queue



Get work from
local queue



Randomized stealing: good theoretical properties, $O(d+W/p)$ with **high probability** under certain conditions, d : depth, W : work



Deterministic stealing: can provide good locality properties

Solutions:

Lock-free data structures (deques, stacks, ...) make extensive use of **strong atomic operations**: CAS (compare-and-swap)

Caution:

To lock or not to lock for performance: difficult, practical issue, application and system dependent

Lock-free data structures: active research area, practical and theoretical issues and challenges

See: master lecture on advanced multiprocessor programming

OpenMP

Standard for (mostly) **data parallel** shared-memory programming in C/C++/Fortran, „Open Multi-Processing“

Developed by group of vendors/compiler companies, universities, users. Official standard since 1997, maintained by Architecture Review Board, non-profit organization owning the OpenMP trademark

Latest release of standard: OpenMP 3.1, July 2011



See www.openmp.org

Also www.compunity.org

ARB Permanent Members:

- AMD
- Cray
- Fujitsu
- HP
- IBM
- Intel
- NEC
- The Portland Group, Inc.
- Oracle Corporation
- ORNL
- Microsoft
- Texas Instruments
- CAPS-Entreprise
- NVIDIA

Auxiliary Members:

- ANL
- ASC/LLNL
- cOMPunity
- EPCC
- LANL
- NASA
- RWTH Aachen University
- Texas Advanced Computing Center



Chair of Language committee: Bronis de Supinski, LLNL

Basic idea:

- Provide for gradual parallelization of C/Fortran programs by identifying constructs - **loops** - where parallelism can easily be exploited
- Constructs and type of parallelism identified by language-pragmas (and a few library operations)



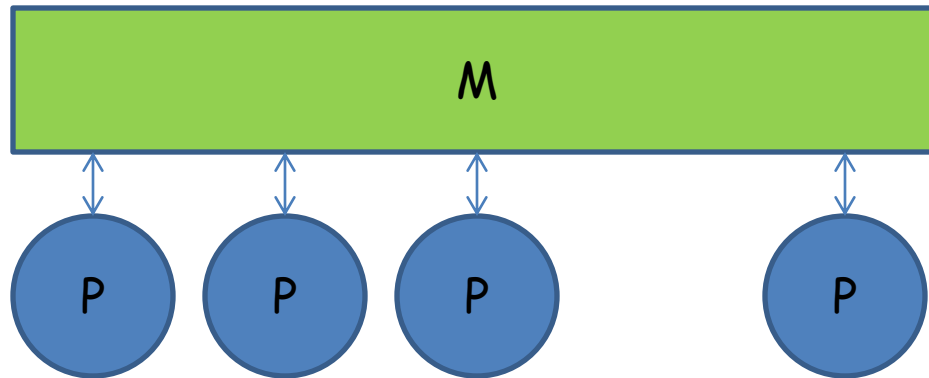
- Requires compiler support
- Idea: a correct OpenMP program is always a correct sequential program (library calls may have to be replaced)

Most C/Fortran compilers now support OpenMP

- GNU gcc
- Intel (one of the first to fully support OpenMP)
- IBM
- Portland Group
- Microsoft
- HP
- Cray
- ...

Lack of/bad compiler support did for some years limit use of OpenMP. Efficient support of OpenMP probably not trivial

OpenMP architecture model



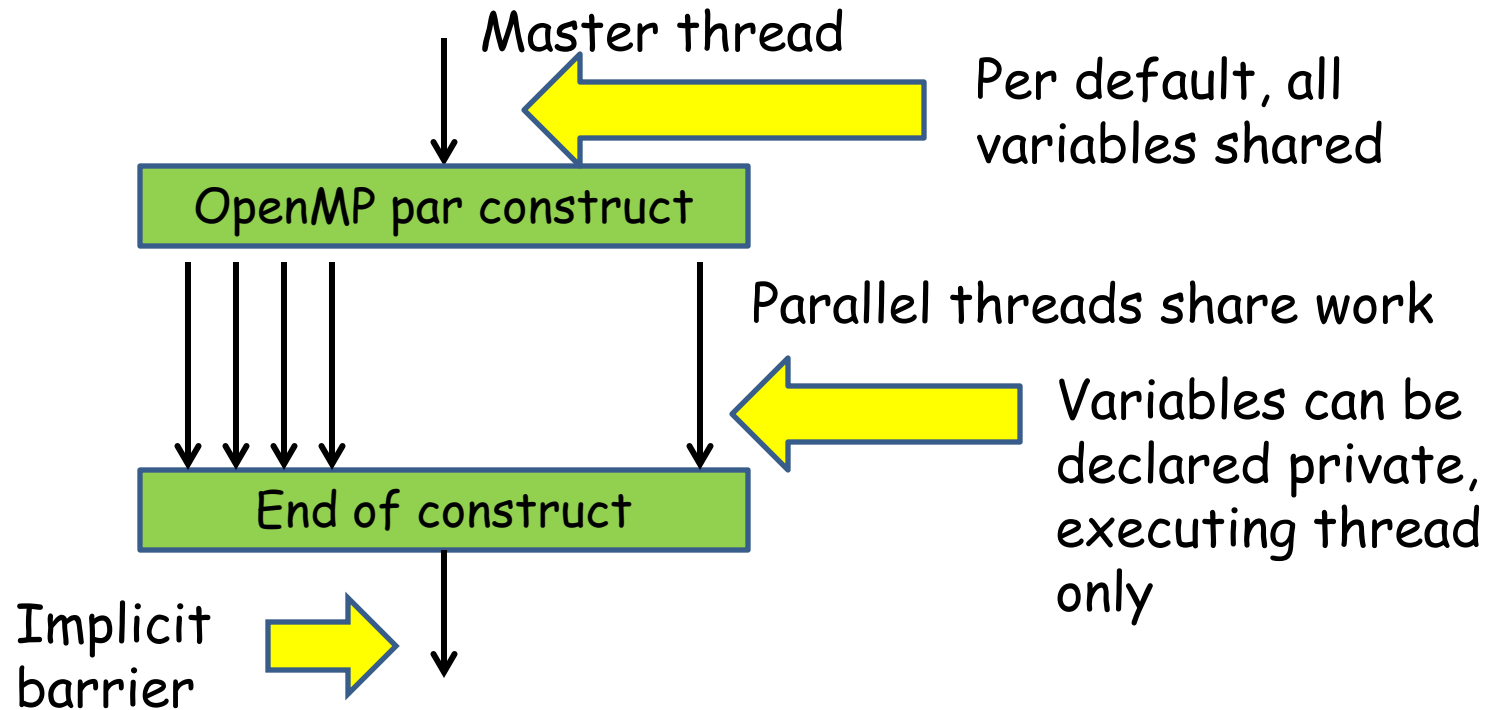
Naive, flat, shared-memory model, processors-memory, no explicit cost-model, UMA

OpenMP programming model

1. Parallelism is (mostly) implicit
2. Fork-join parallelism: **master thread** implicitly spawns threads through OpenMP construct (pragma), threads **join** at end of construct
3. Number of threads limited by number of processors/cores
4. Threads intended to be executed in parallel by available cores/processors
5. Work of OpenMP construct divided across threads

6. Threads can share variables; **shared variables are shared among all threads**
7. Threads can have private variables
8. Unintended updates of shared variables can lead to **race conditions**
9. **Synchronization constructs** for preventing race conditions
10. OpenMP 3.0: task model

Not this lecture



Data transfer between shared and private (copies) variables is transparent, implicit

OpenMP for C:

- Include header `<omp.h>`
- OpenMP constructs identified by `#pragma <directive>`
[clauses]
- Some library routines for getting number of threads,
synchronization mechanisms, ...
- Library routines prefixed by `omp_`
- Macro `_OPENMP` defined (to version date) for conditional
compilation

Compile with

```
gcc -Wall -fopenmp -o openmphello -O3 openmphello.c
```

OpenMP for Fortran:

- OpenMP constructs surrounded by `!$OMP <directive>`
[clauses]

Not this lecture

1st example

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

#include <omp.h> // OpenMP header

int main(int argc, char *argv[]) {
    int threads, myid;
    int i;  threads = 1;

    for (i=1; i<argc&&argv[i][0]=='-'; i++) {
        if (argv[i][1]=='t') sscanf(argv[++i], "%d", &threads);
    }

    printf("Maximum number of threads possible is %d\n",
           omp_get_max_threads());
    // ...
}
```

OpenMP library call

Normally some small multiple of number of physical processors/cores

```
int main(int argc, char *argv[]){
    int threads, myid;
    int i;  threads = 1;

    // ...

    if (threads < omp_get_max_threads()) {
        if (threads < 1) threads = 1;
        omp_set_num_threads(threads);
    } else {
        threads = omp_get_max_threads();
    }

    // ...
}
```

Just setting shared
variable threads to at
most max_threads



```
int main(int argc, char *argv[]){
    int threads, myid;
    int i;  threads = 1;

// ...

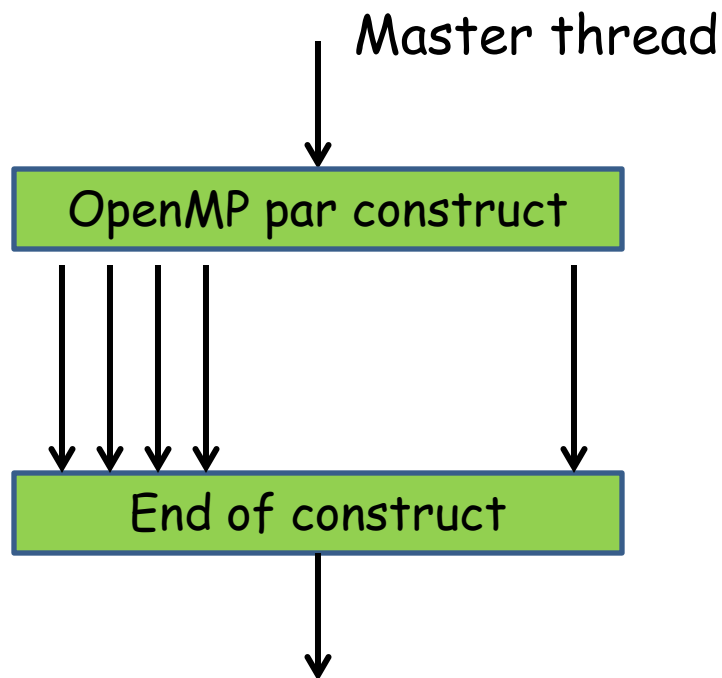
#pragma omp parallel num_threads(threads)
{
    myid = omp_get_thread_num();
    printf("Thread id of %d active\n",myid,threads);
}

return 0;
}
```

OpenMP directive: parallel region executed by num_threads cores

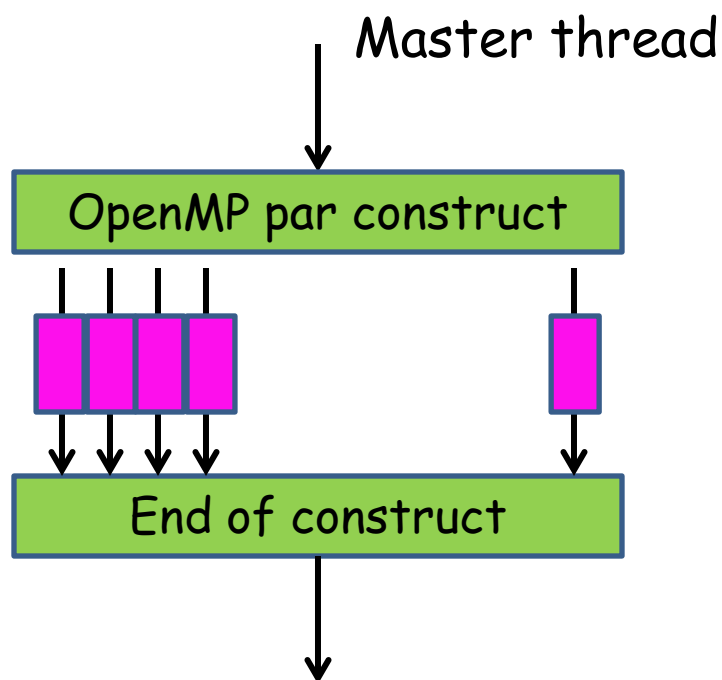
Library call: get thread id - should rarely be needed

Basic work sharing constructs



```
#pragma omp parallel  
{  
    // threads  
}
```

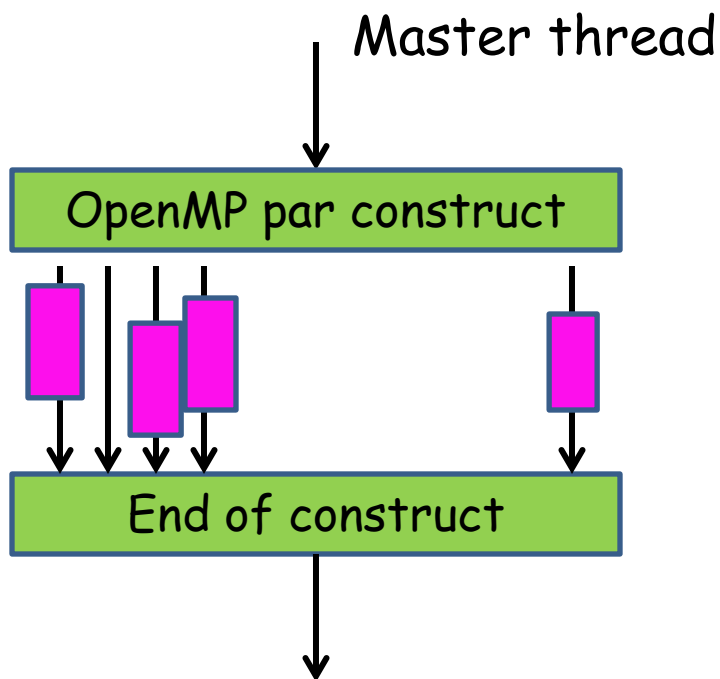
Basic work sharing constructs



```
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0; i<n; i++) {  
        // iterations shared  
    }  
}
```

Data parallel loop scheduled over available threads

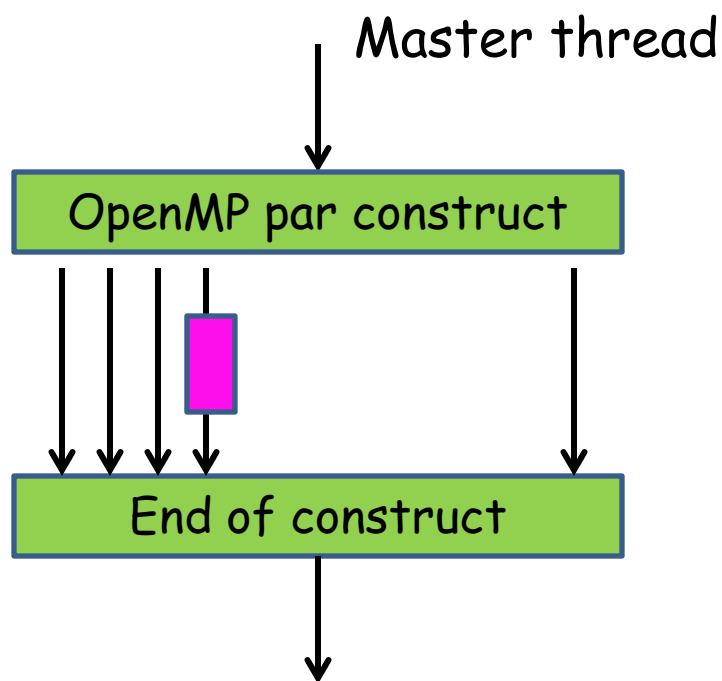
Basic work sharing constructs



Static, finite task parallelism

```
#pragma omp parallel
{
    #pragma omp sections
    #pragma omp section
    {
        // A
    }
    #pragma omp section
    {
        // B
    }
    // ...
}
```

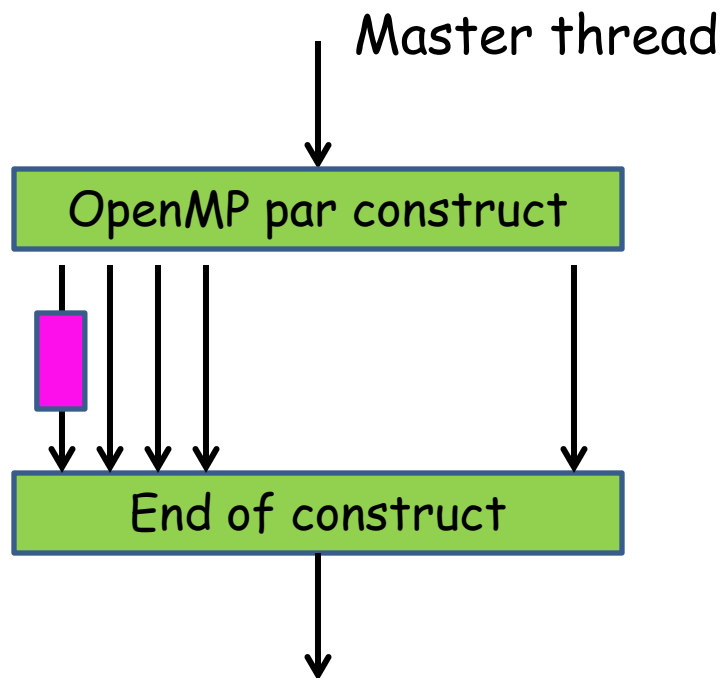
Basic work sharing constructs



```
#pragma omp parallel
{
    #pragma omp single
    {
        // some thread
    }
}
```

Sequential code in parallel construct, no mutual exclusion

Basic work sharing constructs



```
#pragma omp parallel  
{  
    #pragma omp master  
    {  
        // master thread 0  
    }  
}
```

Sequential code by master in parallel construct, no mutual exclusion