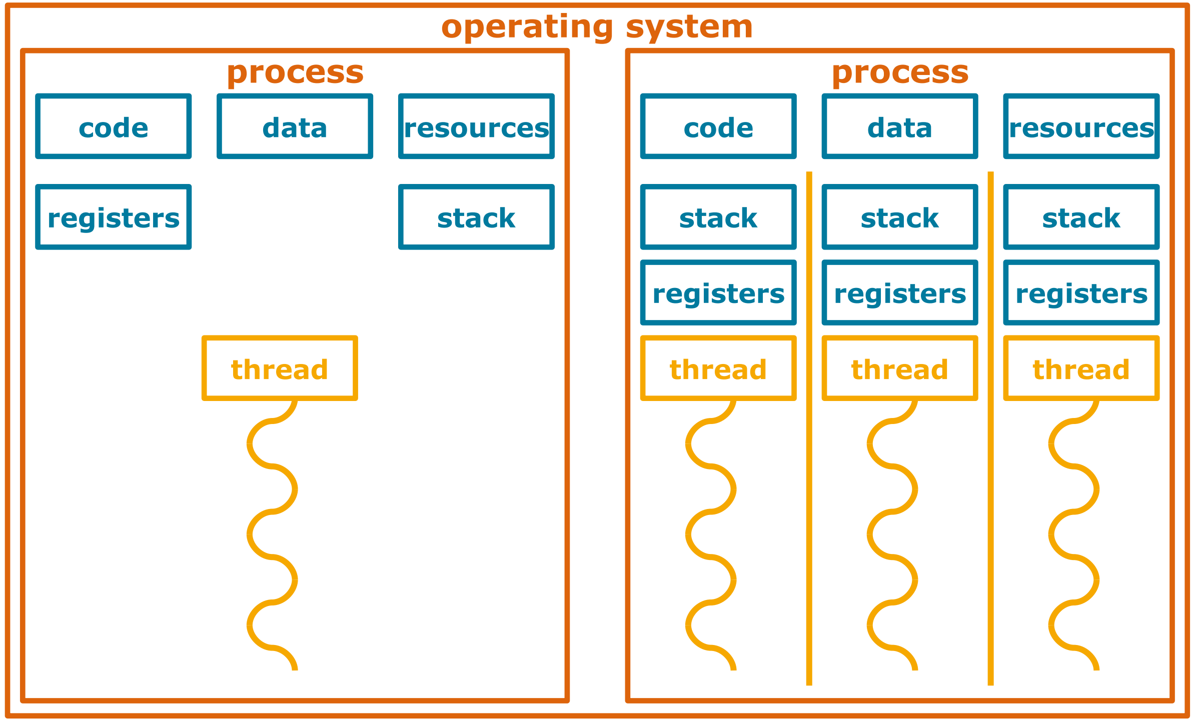
Parallel Programming and Heterogeneous Computing

B2 - Shared-Memory: Programming Models

Max Plauth, *Sven Köhler*, Felix Eberhardt, Lukas Wenzel, and Andreas Polze

Operating Systems and Middleware Group

Recap: Processes and Threads



traditional UNIX approach

Kernel scheduling late to the game

POSIX Threads (Pthreads)

```
pthread {
_create
_self
_cancel
_exit
_join
_kill
_attr_setstacksize
_attr_setstackaddr
_mutex_lock
_mutex_trylock
_mutex_unlock
_cond_signal
_cond_timedwait
_cond_wait
_rwlock_rdlock
_rwlock_unlock
_rwlock_wrlock
_barrier_wait
_key_create
_setspecific
[...]
```

POSIX Threads (Pthreads)

- Part of the POSIX specification collection, defining an API for thread creation and management (*pthread.h*)
- Implemented by all (!) Unix-alike operating systems available
  - Utilization of kernel- or user-mode threads depends on implementation
- Groups of functionality (*pthread_* function prefix)
  - Thread management - Start, wait for termination, …
  - Synchronization based on **mutexes**
  - Synchronization based on **condition variables**
  - Synchronization based on **read/write locks** and **barriers**
- Semaphore API is a separate POSIX specification (*sem_* prefix)

POSIX Threads

pthread_create()

- Create new thread in the process, with given routine and argument

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

pthread_exit(), pthread_cancel()

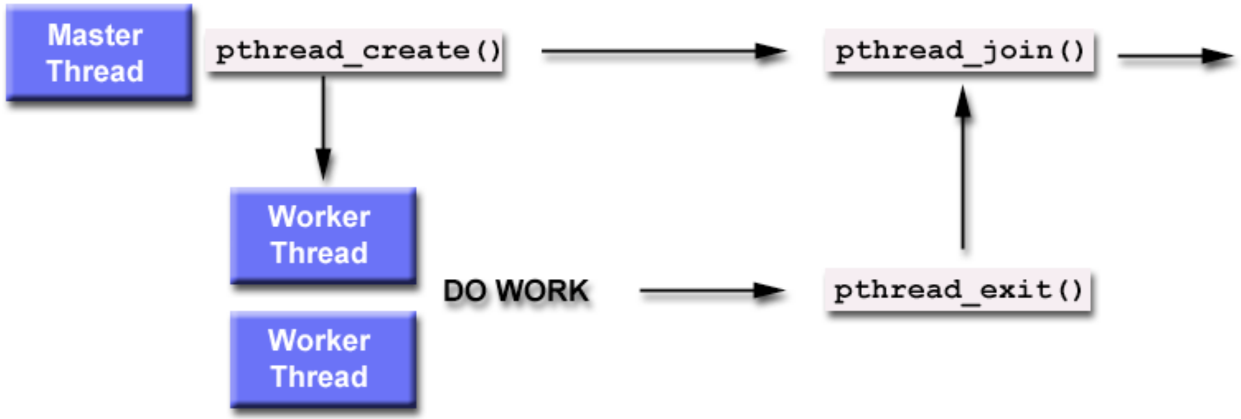
- Terminate thread from inside our outside of the thread

pthread_attr_init() , pthread_attr_destroy()

- Abstract functions to deal with implementation-specific attributes (e.g. stack size limit)
- See discussion in man page about how this improves portability

```
/******************************************************************************
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* AUTHOR: Blaise Barney
* LAST REVISED: 08/09/11
******************************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS        5

void *PrintHello(void *threadid)
{
   long tid; tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for (t = 0; t < NUM_THREADS; t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc != 0) {
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
     }
   }
   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

POSIX Threads



POSIX Threads: Synchronization

- `pthread_join(pthread_t thread, void **code)`
  - Blocks the caller until the specific thread terminates
  - If thread gave exit code to *pthread_exit()*, it can be determined here
  - Only one joining thread per target is thread is allowed

- `pthread_detach(pthread_t thread)`
  - Mark thread as not-joinable (*detached*) - may free some system resources

- `pthread_attr_setdetachstate(pthread_attr_t *attr, int dstate)`
  - Prepare *attr* block so that a thread can be created in some detach state

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS        4

void *BusyWork(void *t) {
   int i; long tid; double result = 0.0; tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i < 1000000; i++) {
      result = result + sin(i) * tan(i); }
   printf("Thread %ld done. Result = %e\n", tid, result);
   pthread_exit((void*) t); }

int main (int argc, char *argv[]) {
   pthread_t thread[NUM_THREADS]; pthread_attr_t attr; int rc; long t; void *status;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for (t=0; t < NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *) t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);}}

   pthread_attr_destroy(&attr);
   for (t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1); }
      printf("Main: completed join with thread %ld having a status of %ld\n",t, (long) status);}

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL); }
```

POSIX Threads

- *int pthread_mutex_init(pthread_mutex_t *mutex,*
  *const pthread_mutexattr_t *attr)*
  - Initialize new mutex, which is unlocked by default
- *int pthread_mutex_lock(pthread_mutex_t *mutex),*
  *int pthread_mutex_trylock(pthread_mutex_t *mutex)*
  - Blocking / non-blocking wait for a mutex lock
- *int pthread_mutex_unlock(pthread_mutex_t *mutex)*
  - Operating system decides about wake-up preference
  - Focus on speed of operation, no deadlock or starvation protection mechanism

- Also support for normal, recursive, and error-check mutex that reports double locking (see *pthread_mutexattr*)

POSIX Threads

- Condition variables are always used in conjunction with a mutex
- Allow to wait on a variable change without polling it in a critical section

- *int pthread_cond_init(pthread_cond_t *cond,*
  *const pthread_condattr_t *attr)*
  - Initializes a condition variable

- *int pthread_cond_wait(pthread_cond_t *cond,*
  *pthread_mutex_t *mutex)*
  - Called with a **locked** mutex
  - Releases the mutex and blocks on the condition in one atomic step
  - On return, the mutex is again locked and owned by the caller
- *pthread_cond_signal(), pthread_cond_broadcast()*
  - Unblock thread waiting on the given condition variable

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */

void main() {
  /* declarations and initializations */
  task_available = 0;
  pthread_init();
  pthread_cond_init(&cond_queue_empty, NULL);
  pthread_cond_init(&cond_queue_full, NULL);
  pthread_mutex_init(&task_queue_cond_lock, NULL);
  /* create and join producer and consumer threads */
  ...
}

void *producer(void *producer_thread_data) {
  int inserted;
  while (!done()) {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 1)
      pthread_cond_wait(&cond_queue_empty, &task_queue_cond_lock);
    insert_into_queue();
    task_available = 1;
    pthread_cond_signal(&cond_queue_full);
    pthread_mutex_unlock(&task_queue_cond_lock);
}

void *consumer(void *consumer_thread_data) {…}
void *watch_count(void *t)
{
  long my_id = (long)t;
  printf("Starting watch_count(): thread %ld\n", my_id);
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("Thread %ld Count= %d. Going into wait...\n”, my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("Thread %ld Signal received. Count= %d\n", my_id,count);
    printf("Thread %ld Updating count...\n", my_id,count);
    count += 125;
    printf("Thread %ld count = %d.\n", my_id, count);
  }
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  pthread_t threads[3]; pthread_attr_t attr; int i, rc; long t1=1, t2=2, t3=3;

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Count = %d. Done.\n", NUM_THREADS, count);
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

Int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t) {
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    if (count == COUNT_LIMIT) {
      printf("Thread %ld, count = %d  Threshold reached. ",
              my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
    }
    printf("Thread %ld, count = %d, unlocking mutex\n",
                    my_id, count);
    pthread_mutex_unlock(&count_mutex);
    /* Do some work so threads can alternate on mutex lock */
    sleep(1); }
  pthread_exit(NULL);
}
```

Windows vs. POSIX Synchronization

Windows	POSIX
WaitForSingleObject	pthread_mutex_lock()
WaitForSingleObject(timeout==0)	pthread_mutex_trylock()
Auto-reset events	Condition variables

Further PThreads Functionality

- *int pthread_setconcurrency(int new_level)*
  - Only meaningful for **m:n** user-to-kernel threading environments
- *int pthread_setaffinity_np(pthread_t thread,*
  *size_t cpusetsize, const cpu_set_t *set)*
  - Modify processor affinity mask of a thread
  - Forked children inherit this mask
  - Useful for pinning threads explicitely
    - Better load balancing, avoid cache pollution
- *int pthread_sigmask(int how, const sigset *set, sigset *oset)*
  - Individual threads can mask out signals for explicit responsibilites
- *Int pthread_barrier_wait(pthread_barrier_t *barrier)*
  - Barrier implementation, optional part of POSIX standard (check for *_POSIX_BARRIERS* macro)

2

std::async std::thread

C++11

- C++11 specification added support concurrency constructs
- Allows asynchronous tasks with *std::async* or *std::thread*
- Relies on *Callable* instance (functions, member functions, lambdas, ...)

```cpp
#include <future>
#include <iostream>

void write_message(std::string const& message) {
  std::cout<<message;
}

int main() {
  auto f = std::async(write_message,
    "hello world from std::async\n");
  write_message("hello world from main\n");
  f.wait();
}
```

```cpp
#include <thread>
#include <iostream>

void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
   std::thread t(write_message,
     "hello world from std::thread\n");
   write_message("hello world from main\n");
   t.join();
}
```

https://en.cppreference.com/w/cpp/thread

C++11: Futures & Promises

- Launch policy for the *async call* can be specified
  - Deferred or immediate launch of the activity
- As for all asynchronous task types, a **future** is returned
  - Object representing the (future) result of an asynchronous operation, allows to block on the result reading
  - Original concept by Baker and Hewitt [1977]
- A **promise** object can store a value that is later acquired via a future object
  - Separate concept since futures are only readable
  - Can provide a dummy barrier implementation
- Future == Handle, Promise == Value
- Promise and future as concept also available in Java 5, Smalltalk, Scheme, CORBA, …

```cpp
#include <iostream>
#include <future>
#include <thread>

int main()
{
    // future from a packaged_task
    std::packaged_task<int()> task([](){ return 7; }); // wrap the function
    std::future<int> f1 = task.get_future();  // get a future
    std::thread(std::move(task)).detach(); // launch on a thread

    // future from an async()
    std::future<int> f2 = std::async(std::launch::async, [](){ return 8; });

    // future from a promise
    std::promise<int> p;
    std::future<int> f3 = p.get_future();
    std::thread( [](std::promise<int>& p){ p.set_value(9); },
                 std::ref(p) ).detach();

    std::cout << "Waiting..." << std::flush;
    f1.wait();
    f2.wait();
    f3.wait();
    std::cout << "Done!\nResults are: "
              << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';
}
```

ParProg20 B2 Programming Models

Sven Köhler

C++11: Locks and RAII

- Four mutex classes, basic operations in the *Lockable* concept: *m.lock(), m.try_lock(), m.unlock()*
- Locking is tricky with exceptions, so C++ offers some high-level templates

```
std::mutex m;
void f(){
    std::lock_guard<std::mutex> guard(m);
    std::cout << "In f()" << std::endl;
}

int main(){
    m.lock();
    std::thread t(f);
    for(unsigned i=0;i<5;++i){
        std::cout<<"In main()"<<std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    m.unlock();
    t.join();
}
```

C++11: Condition Variables

Waiting for events with condition variables avoids polling

```
std::condition_variable the_cv;
std::mutex the_mutex;

void wait_and_pop(my_class& data) {
    std::unique_lock<std::mutex> lk(the_mutex);
    the_cv.wait(lk,[]() {return !the_queue.empty();});
    data = the_queue.front();
    the_queue.pop();
}

void push(Data const& data)
{
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    the_cv.notify_one();
}
```

```
while (the_queue.empty()) {
    the_cv.wait(lk);
}
```

C++11 std::atomic

- Lock-free std::atomic<T> types that are free from data races for T =
  - *char, schar, uchar, short, ushort, int, uint, long, ulong, char16_t, wchar_t, intptr_t, size_t, …*
- Common member functions
  - *is_lock_free()*
  - *store(), load()*
  - *exchange()*
- Specialized member functions
  - *fetch_add(), fetch_sub(), fetch_and(), fetch_or(), operator++, operator+=, …*

C++11 Memory Model

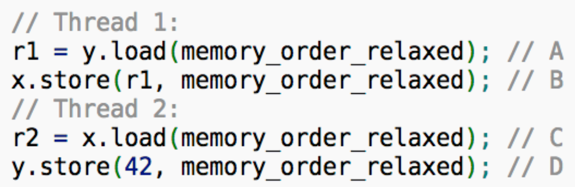
- C++11 makes concurrency a first-class language citizen
  - Similar to Java, .NET, and other runtime-based languages
  - (Side note: Fixed Java >=5 memory model with JSR-133)
  - Unlike any C++ or C version before
- Demands a memory model of the language
  - What means atomicity? When is a written value visible?
  - Relationship between variables and registers / memory
  - Only chance for the compiler to apply optimizations such as re-ordering of instructions
  - Irrelevant without a concurrency concept in the language
  - Proper definition leads to **portable concurrency behavior**
- C++11 needs to define that for **native code** !!!

https://web.archive.org/web/20131111103613/http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/threadsintro.html

C++11 Memory Model

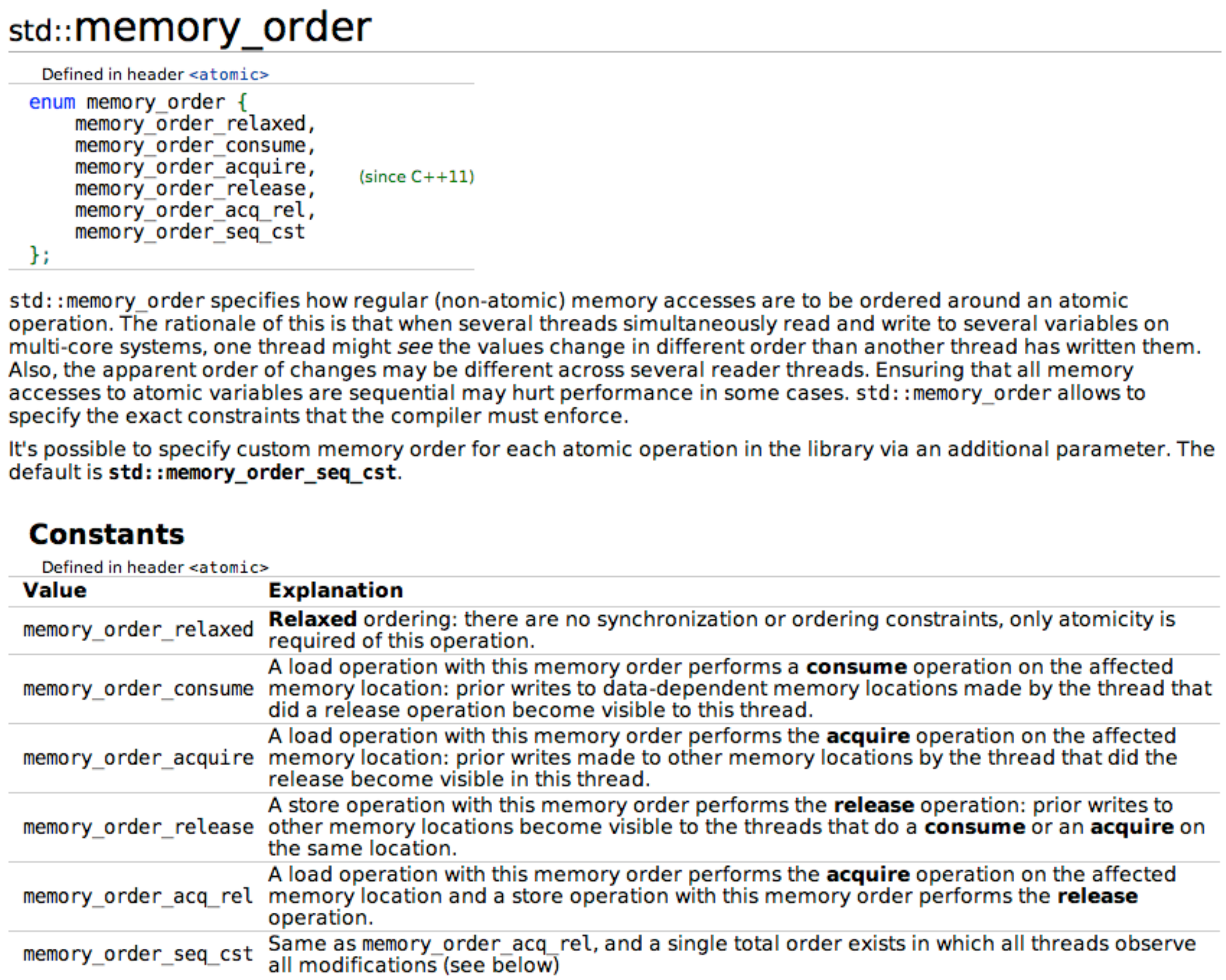
Example: Atomic objects have *store()* and *load()* methods that ensure sequential consistency

- Comparable to Java *volatile*
- Leads to x86 instructions for *memory fencing*
- Fine-grained options to influence access order from threads, which may allow fence removal by the compiler
- http://en.cppreference.com/w/cpp/atomic/memory_order



```
// Thread 1:
r1 = y.load(memory_order_relaxed); // A
x.store(r1, memory_order_relaxed); // B
// Thread 2:
r2 = x.load(memory_order_relaxed); // C
y.store(42, memory_order_relaxed); // D
```

- A sequenced-before B
- C sequenced-before D
- r1 == r2 == 42 may happen



std::memory_order

Defined in header `<atomic>`

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

(since C++11)

`std::memory_order` specifies how regular (non-atomic) memory accesses are to be ordered around an atomic operation. The rationale of this is that when several threads simultaneously read and write to several variables on multi-core systems, one thread might *see* the values change in different order than another thread has written them. Also, the apparent order of changes may be different across several reader threads. Ensuring that all memory accesses to atomic variables are sequential may hurt performance in some cases. `std::memory_order` allows to specify the exact constraints that the compiler must enforce.

It's possible to specify custom memory order for each atomic operation in the library via an additional parameter. The default is **`std::memory_order_seq_cst`**.

Constants

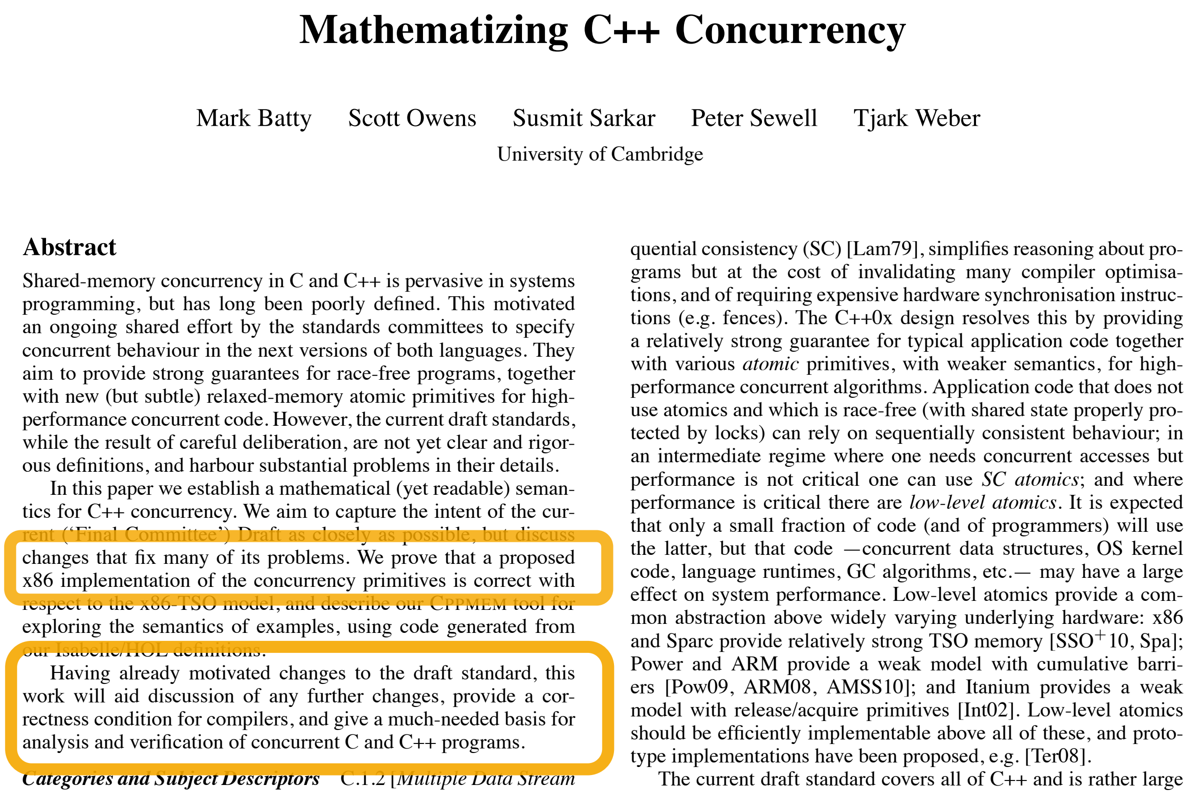
Defined in header `<atomic>`

Value	Explanation
`memory_order_relaxed`	**Relaxed** ordering: there are no synchronization or ordering constraints, only atomicity is required of this operation.
`memory_order_consume`	A load operation with this memory order performs a **consume** operation on the affected memory location: prior writes to data-dependent memory locations made by the thread that did a release operation become visible to this thread.
`memory_order_acquire`	A load operation with this memory order performs the **acquire** operation on the affected memory location: prior writes made to other memory locations by the thread that did the release become visible in this thread.
`memory_order_release`	A store operation with this memory order performs the **release** operation: prior writes to other memory locations become visible to the threads that do a **consume** or an **acquire** on the same location.
`memory_order_acq_rel`	A load operation with this memory order performs the **acquire** operation on the affected memory location and a store operation with this memory order performs the **release** operation.
`memory_order_seq_cst`	Same as `memory_order_acq_rel`, and a single total order exists in which all threads observe all modifications (see below)

ParProg20 B2 Programming Models

Sven Köhler

Proving C++ Can Be Implemented



Mathematizing C++ Concurrency

Mark Batty Scott Owens Susmit Sarkar Peter Sewell Tjark Weber
University of Cambridge

Abstract

Shared-memory concurrency in C and C++ is pervasive in systems programming, but has long been poorly defined. This motivated an ongoing shared effort by the standards committees to specify concurrent behaviour in the next versions of both languages. They aim to provide strong guarantees for race-free programs, together with new (but subtle) relaxed-memory atomic primitives for high-performance concurrent code. However, the current draft standards, while the result of careful deliberation, are not yet clear and rigorous definitions, and harbour substantial problems in their details.

In this paper we establish a mathematical (yet readable) semantics for C++ concurrency. We aim to capture the intent of the current ('Final Committee') Draft as closely as possible, but discuss changes that fix many of its problems. We prove that a proposed x86 implementation of the concurrency primitives is correct with respect to the x86-TSO model, and describe our CPPMEM tool for exploring the semantics of examples, using code generated from our Isabelle/HOL definitions.

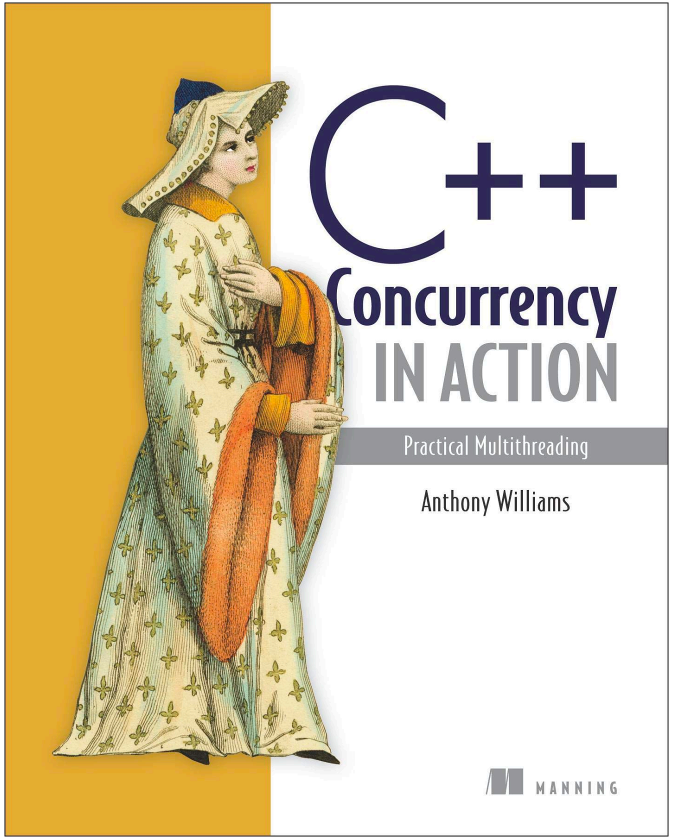
Having already motivated changes to the draft standard, this work will aid discussion of any further changes, provide a correctness condition for compilers, and give a much-needed basis for analysis and verification of concurrent C and C++ programs.

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream*

quential consistency (SC) [Lam79], simplifies reasoning about programs but at the cost of invalidating many compiler optimisations, and of requiring expensive hardware synchronisation instructions (e.g. fences). The C++0x design resolves this by providing a relatively strong guarantee for typical application code together with various *atomic* primitives, with weaker semantics, for high-performance concurrent algorithms. Application code that does not use atomics and which is race-free (with shared state properly protected by locks) can rely on sequentially consistent behaviour; in an intermediate regime where one needs concurrent accesses but performance is not critical one can use *SC atomics*; and where performance is critical there are *low-level atomics*. It is expected that only a small fraction of code (and of programmers) will use the latter, but that code —concurrent data structures, OS kernel code, language runtimes, GC algorithms, etc.— may have a large effect on system performance. Low-level atomics provide a common abstraction above widely varying underlying hardware: x86 and Sparc provide relatively strong TSO memory [SSO$^+$10, Spa]; Power and ARM provide a weak model with cumulative barriers [Pow09, ARM08, AMSS10]; and Itanium provides a weak model with release/acquire primitives [Int02]. Low-level atomics should be efficiently implementable above all of these, and prototype implementations have been proposed, e.g. [Ter08].

The current draft standard covers all of C++ and is rather large

Further Reading



3

#pragma omp

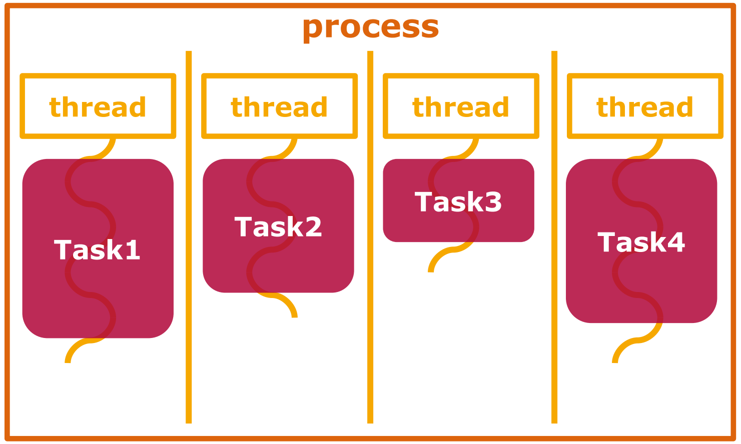
Explicit vs Implicit Threading

Thread generation, synchronization, data access:

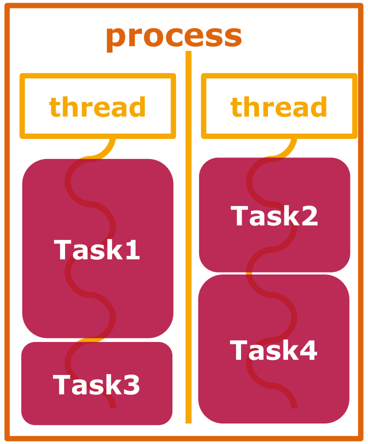
Explicit, as part of some sequential code (OS API, C++/Java/Python Threads)

Implicit, based on a framework (OpenMP, OpenCL, Intel TBB, ...)

Explicit Threading



Implicit Threading



Threads vs. Tasks

: Task
(OpenMP context)

- **Process:** Address space, resource handles, code, set of threads
- **Thread:** Control flow
  - Preemptive scheduling by the operating system
  - Can migrate between cores
- **Task:** Control flow
  - Modeled as object, statement, lambda expression,
    or anonymous function
  - Cooperative scheduling, typically by a user-mode library
  - Dynamically mapped to threads from a pool
  - Task model replaces context switch with **yielding** approach
  - Typical scheduling policy is **central queue** or **work stealing**

Work Stealing

Blumofe, Leiserson, Charles:
Scheduling Multithreaded Computations by Work Stealing (FOCS 1994)

Problem of scheduling scalable multithreading problems on SMP

Work sharing: When processors create new work,
the scheduler migrates threads for balanced utilization

Work stealing: Underutilized core takes work from other processor,
leads to less thread migrations

- Goes back to work stealing research in Multilisp (1984)
- Supported in OpenMP implementations, TPL, TBB, Java, Cilk, …

Randomized work stealing: Lock-free ready dequeue per processor

- Task are inserted at the bottom, local work is taken from the bottom
- If no ready task is available, the core steals the top-most one from another randomly chosen core; added at the bottom

- Ready tasks are executed, or wait for a processor becoming free

Large body of research about other work stealing variations

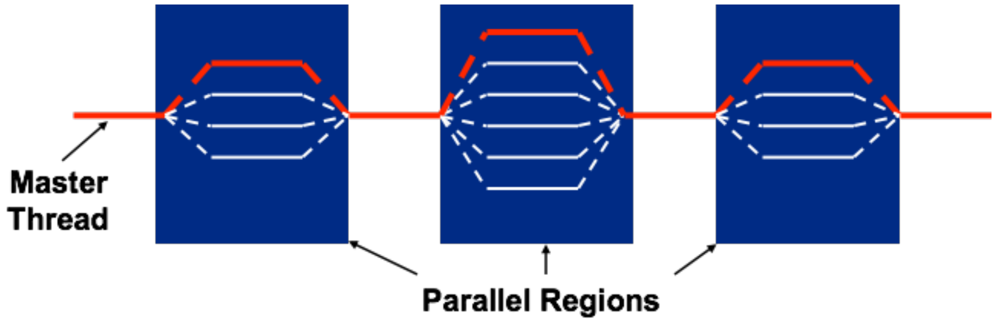
OpenMP

Specification for C/C++ and Fortran language extension

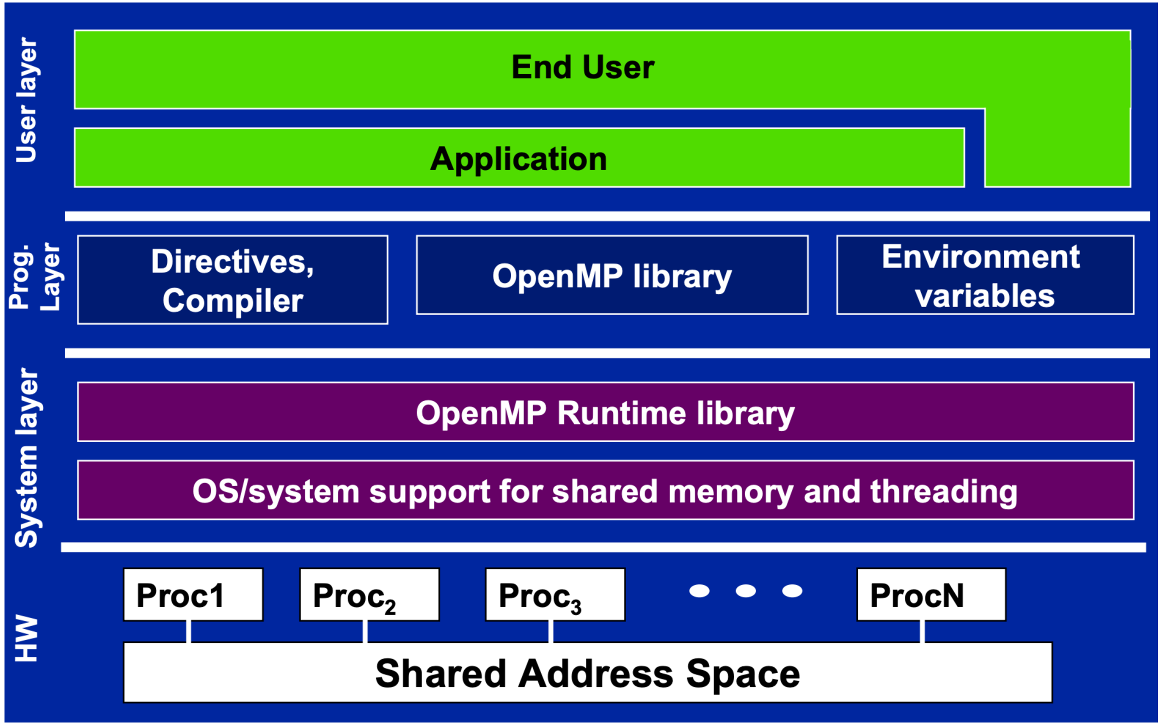
- Portable shared memory thread programming
- High-level abstraction of task- and loop parallelism
- Derived from compiler-directed parallelization of serial language code (HPF), with support for incremental change of legacy code
- **Multiple implementations** exist

Programming model: Fork-Join-Parallelism

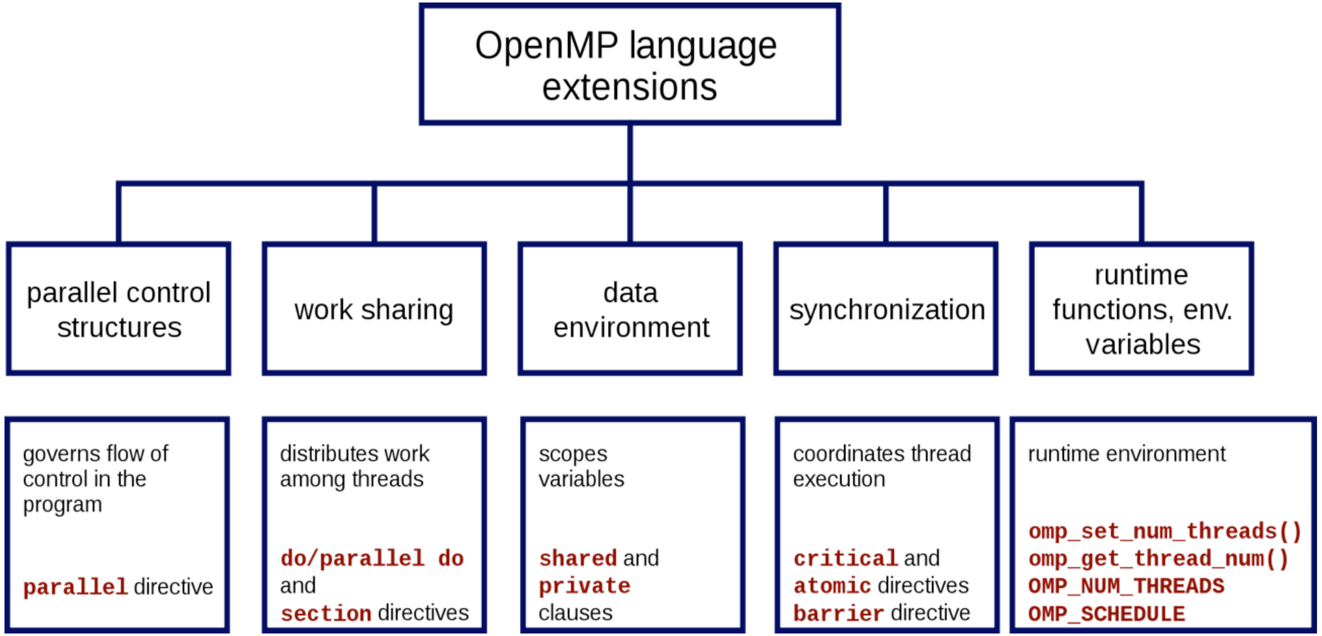
- Master thread spawns group of threads for limited code region



OpenMP Stack



OpenMP C/C++ Language Extensions


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

(public domain, Wikipedia)

OpenMP Basic Constructs

```
#pragma omp contstruct ...
statement; / { block }
```

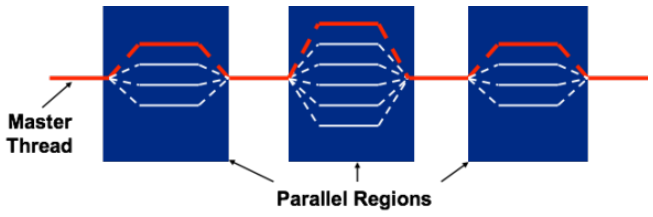
OpenMP Parallel Region: #pragma omp parallel

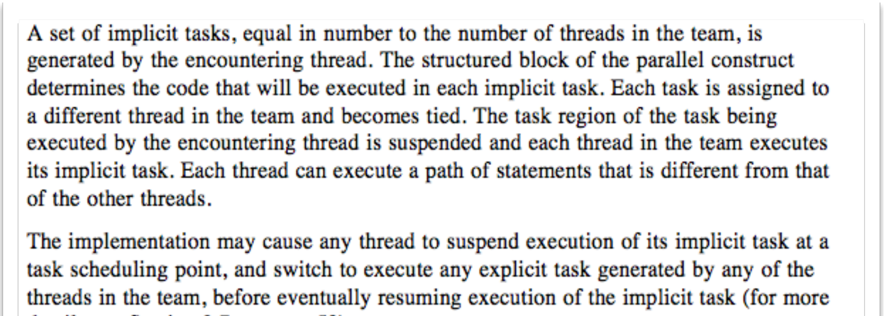
Encountering **thread** for the parallel region generates a set of implicit **tasks,** each with possibly different instructions, assigned to a **thread** from **pool**

Task execution may **suspend** at some **scheduling point**

- **Implicit barrier** regions, encountered barrier primitives
- Encountered task / taskwait constructs
- At the end of a task region (with memflush)

Idle worker threads may sleep or spin, depending on library configuration (performance issue in serial parts)




A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the parallel construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and switch to execute any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more

OpenMP Configuration and Query Functions

Environment variables

- `OMP_NUM_THREADS`: number of threads during execution, upper limit for dynamic adjustment of threads
- `OMP_SCHEDULE`: set schedule type and chunk size for parallelized loops of scheduling type `runtime`

Query functions

- `omp_get_num_threads`: Number of threads in the current parallel region
- `omp_get_thread_num`: Current thread number in the team, master=0
- `omp_get_num_procs`: Available number of processors
- …

OpenMP hello world

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

int main (int argc, char * const argv[]) {
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n",
           omp_get_thread_num(),
           omp_get_num_threads());
    return 0;
}
```

```
>> gcc -fopenmp -o hello_omp hello_omp.c
```

ParProg20 B2 Programming Models

Sven Köhler

OpenMP Sections

- Explicit definition of code blocks being distributable amongst threads with `section` directive
- Executed in the context of the implicit task
- Intended for non-iterative parallel work in the code
- One thread may execute more than one section - runtime decision
- Implicit barrier at the end of the `sections` block
  - Can be overriden with the `nowait` clause

```
#pragma omp parallel
{
  #pragma omp sections [ clause [ clause ] ... ]
  {
    [#pragma omp section ]

       structured-block1

    [#pragma omp section ]

       structured-block2
}}
```

OpenMP Work Sharing Overview

Possibilities for distribution of tasks across threads (‚work sharing‘)

- `omp sections` - Define code blocks dividable among threads
  - Implicit barrier at the end
- `omp for` - Automatically divide a loop‘s iterations into tasks
  - Implicit barrier at the end
- `omp single / master` - Denotes a task to be executed **only** by first arriving thread resp. the master thread
  - Implicit barrier at the end,
    intended for non-thread-safe activities (I/O)
- `omp task` - Explicitly enqueue task (may start immediately, no barrier)

Task scheduling is handled by the OpenMP implementation

Clause combinations possible: `#pragma omp parallel for`

OpenMP Data Sharing

- **Shared variable**: Name provides access to memory in all tasks
  - Shared by default: global extern variables, static variables, variables with namespace scope, variables with file scope
  - `shared` clause can be added to any `omp` construct, defines a list of additionally shared variables
  - Provides **no automatic protection**, just marks variables for handling by runtime environment
- **Private variable:** Clone variable in each task, no initialization
  - Use `private` clause for having one copy per thread
  - Private by default: Local variables in functions called from parallel regions, loop iteration variables, section scope variables
  - `firstprivate:` Initialization with last value before region
  - `lastprivate:` Result value after region from last loop iteration or lexically last `section` directive

OpenMP Data Sharing: Example

```
#pragma omp parallel for shared(n, a) private(b)
for (int i = 0; i < n; i++)
{
    b = a + i;
    // ...
}
```

OpenMP Data Sharing: default clause

```
#pragma omp parallel for default(shared)
```

```
#pragma omp parallel for default(none) shared(n)
```

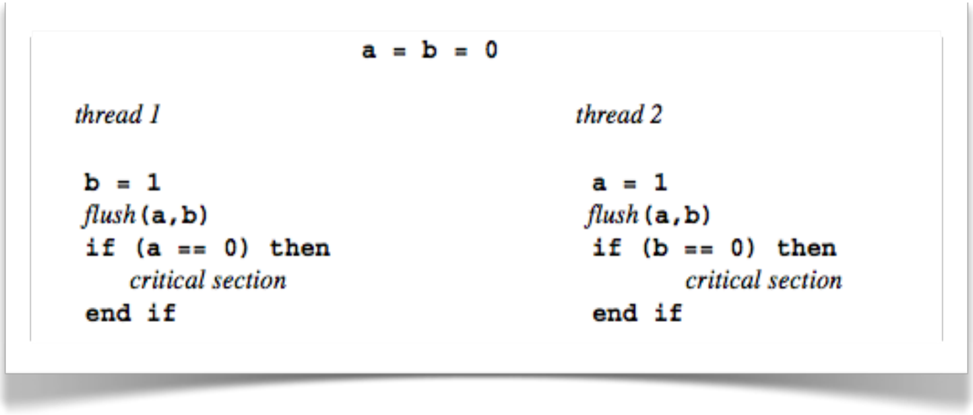
Forces programmer to explicitly state sharing
(compile time error otherwise)

OpenMP Consistency Model

A thread's temporary view of memory is not required to be consistent with memory at all times (weak-ordering consistency)

- Example: Keeping loop variable in a register for efficiency
- Compiler needs information when consistent view is demanded
- Implicit flush on different occasions, such as barrier region
- In all other cases, read shared variables must be flushed before

`#pragma omp flush`

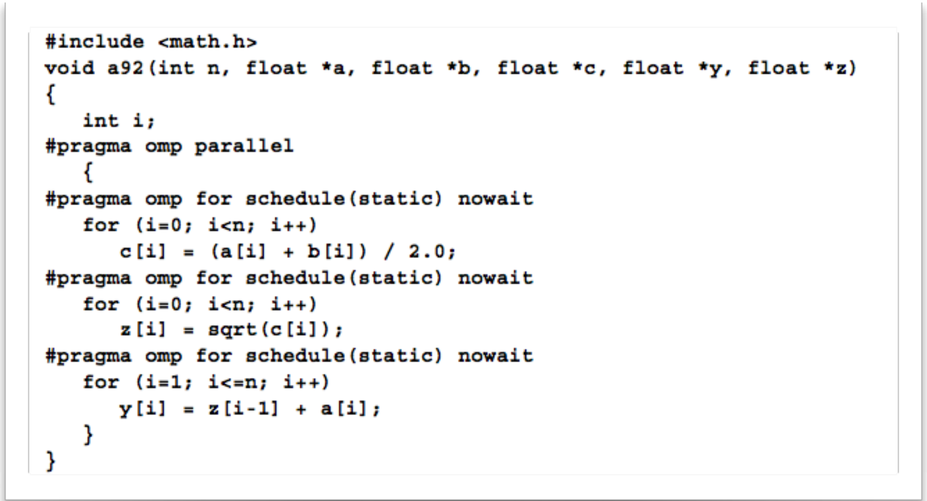


```
                    a = b = 0

thread 1                          thread 2

 b = 1                             a = 1
 flush(a,b)                        flush(a,b)
 if (a == 0) then                  if (b == 0) then
     critical section                  critical section
 end if                            end if
```

OpenMP Loop Parallelization

- `for` construct: Parallel execution of iterations
- Iteration variable must be integer
- Mapping of threads to iterations is controlled by `schedule` clause
- Has implications on exception handling, `break` and `continue` primitives

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
    result[i] = some_complex_function(i);
}
```



```
#include <math.h>
void a92(int n, float *a, float *b, float *c, float *y, float *z)
{
   int i;
#pragma omp parallel
   {
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      z[i] = sqrt(c[i]);
#pragma omp for schedule(static) nowait
   for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
   }
}
```

OpenMP Loop Parallelization Scheduling

- `schedule (static, [chunk]):`
  - Contiguous ranges of iterations (chunks) are assigned to the threads
  - Low overhead, round robin assignment to free threads
  - Static scheduling for predictable and similar work per iteration
  - Increasing chunk size reduces overhead, improves cache hit rate
  - Decreasing chunk size allows finer balancing of work load
  - Default is one chunk per thread
- `schedule (guided, [chunk])`
  - Dynamic schedule, shrinking ranges per step
  - Starts with large block, until minimum chunk size is reached
  - Good for computations with increasing iteration length (e.g. prime sieves)
- `schedule (dynamic, [chunk])`
  - Idling threads grab iteration (or chunk) as available (work-stealing)
  - Higher overhead, but good for unbalanced/unpredicable iteration work load

OpenMP Synchronization

Synchronizing with task completion

- Implicit barrier at the end of `single` block, removable by `nowait` clause
- `#pragma omp barrier` (wait for all other threads in the team)
- `#pragma omp taskwait` (wait for completion of child tasks)

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

int main() {
  #pragma omp parallel
  {
    printf("Start: %d\n", omp_get_thread_num());
    #pragma omp single //nowait
    printf("Got it: %d\n", omp_get_thread_num());
    printf("Done: %d\n", omp_get_thread_num());
  }
  return 0;
}
```

OpenMP Synchronization

Synchronizing variable access with `#pragma omp critical`

- Enclosed block is executed by all threads,
  but restricted to one at a time

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for
  for(int i = 0; i < N; i++) {
     #pragma omp critical
     sum += a[i] * b[i];
  }
  return sum;
}
```

OpenMP Synchronization

Alternative: `#pragma omp reduction (op: list)`

- Execute parallel tasks based on private copies of `list`
- Perform reduction on results with `op` afterwards
- Without race conditions

Supported associative operands:
+, *, -, ^, bitwise AND, bitwise OR, logical AND, logical OR, min, max

```
#pragma omp parallel for reduction(+:sum)
   for(i = 0; i < N; i++) {
     sum += a[i] * b[i];
   }
```

OpenMP Tasks

- Major change with OpenMP 3, allows description of irregular parallelization problems
  - Farmer / worker algorithms, recursive algorithms, while loops
- Definition of tasks as composition of code to execute, data environment, and control variables
  - Unit of work that may be deferred
  - Can be nested inside parallel regions and other tasks, so recursion becomes possible
  - Implicit task generation with `parallel` and `for` constructs
- Tasks run at **task scheduling points**
- Runtime may move tasks between threads, or delay them
- `sections` are similar, but mainly work for static partitioning
- **Tied tasks** always keep the same thread and follow the scheduling point concept, developer may untie tasks

OpenMP Tasks Example: List Traversal

[Duran, BSC]

```
void traverse_list ( List l )
{
Element e;

#pragma omp parallel private(e)
    for ( e = l->first; e ; e = e->next )
       #pragma omp single nowait
          process(e);
}
```

OpenMP 2

```
void traverse_list ( List l )
{
Element e;
for ( e = l->first; e ; e = e->next )
    #pragma omp task
       process(e);

#pragma omp taskwait
```

OpenMP 3

- Parallelize operations on list items
- Traversal of dynamic structure, so sections do not help
- Without tasks
  - Poor performance due to abuse of single construct
- Barrier with `taskwait`
  - Thread suspends until all direct child tasks are done

OpenMP Tasks Example: Post-order Tree Traversal

```
void traverse( struct node *p ) {
    if (p->left)
        #pragma omp task
        traverse(p->left);
    if (p->right)
        #pragma omp task
         traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

```
int main (void) {
    ...
    #pragma omp parallel
    {
        #pragma omp single
        traverse (p);
    }
    ...
}
```

p is `firstprivate` by default

OpenMP Best Practices [Süß & Leopold]

Typical correctness mistakes

- Access to shared variables not protected
- Use of locks / shared variables without `flush`
- Declaring parallel loop variable as `shared`

Typical performance mistakes

- Use of `critical` when `atomic` would be sufficient
- Too much work inside a `critical` section
- Unnecessary `flush / critical`

Süß, M., & Leopold, C. (2005) Common mistakes in OpenMP and how to avoid them. In: *International Workshop on OpenMP* (pp. 312-323). Springer, Berlin, Heidelberg.

OpenMP 4[.5] (2013-2015)

- Portable primitives to describe SIMD parallelization
  - Loop vectorization with *simd* construct
  - Several arguments for guiding the compiler (e.g. alignment)
- Offloading/Targeting extensions
  - Thread with the OpenMP program executes on the *host device*, an implementation may support other *target devices*
  - Control off-loading of loops and code regions on such devices
- New API for using a *device data environment*
  - OpenMP - managed data items can be moved to the device
  - Threads cannot migrate between devices
- New primitives for better cancellation / exception handling
- User-defined reduction operations
- Allows to model task dependencies (task groups, graphs)

OpenMP 5 (November 2018)

- Memory allocation models (represent different memory regions)
- Task reductions
- Better accelerator support (unification with OpenACC)
- Improved portability (`declare variant`, `metadirective`)
- Improved C++ support (e.g. iterators)
- New interfaces for debugging and performance analysis

4

#include

Intel Threading Building Blocks (TBB)

- Portable C++ library, toolkits for different operating systems
- Also available as open source version
- Complements basic OpenMP
  - Loop parallelization, parallel reduction, synchronization, explicit tasks
- High-level concurrent containers
  - hash map, queue, vector, set
- High-level parallel operations
  - prefix scan, sorting, data-flow pipelining, deterministic reduce
- Unfair scheduling approach, to favor threads having data in cache
- Supported for cache-aware memory allocation
- Comparable: Microsoft C++ Concurrency Runtime

Intel Math Kernel Library (MKL)

- Intel library with hand-optimized functions for ...
  - Highly vectorized and threaded linear algebra
    - Basic Linear Algebra Subprograms (BLAS) API, confirms to de-facto standards in high-performance computing
    - Vector-vector, matrix-vector, matrix-matrix operations
  - Fast fourier transforms (FFT)
    - Single precision, double precision, complex, real, ...
  - Vector math and statistics functions
    - Random number generators and probability distributions
    - Spline-based data fitting
- C or Fortran API calls
- Beats any automated compiler optimization

And now for a break and a cup of herbal tea*.

*or beverage of your choice