Parallel Programming Concepts

From Threads to Tasks

Peter Tröger

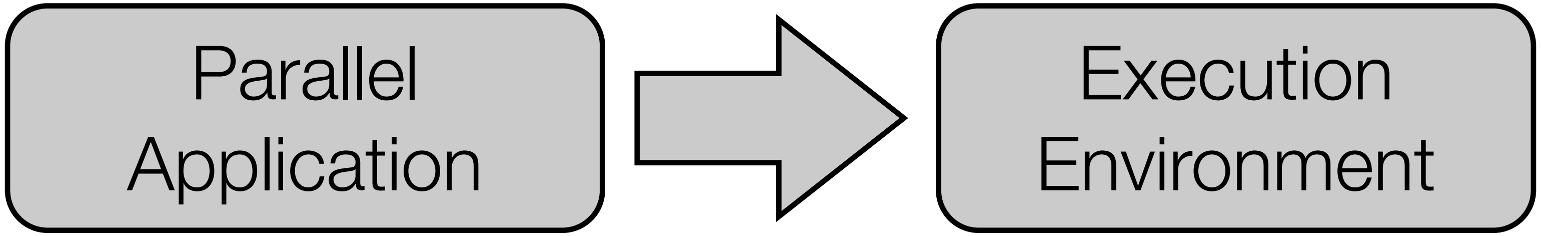
Sources:

Clay Breshears: The Art of Concurrency
Blaise Barney: Introduction to Parallel Computing
OpenMP 3.0 Specification
MPI2 Specification
Anthony Williams: C++11 Concurrency Tutorial
Blaise Barney: OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
http://preshing.com/20120612/an-introduction-to-lock-free-programming

Parallel Programming

Multi-Tasking	PThreads, OpenMP, OpenCL, Linda, Cilk, ...
Message Passing	MPI, PVM, CSP Channels, Actors, ...
Implicit Parallelism	Map/Reduce, PLINQ, HPF, Lisp, Fortress, ...
Mixed Approaches	Ada, Scala, Clojure, Erlang, X10, ...

	Data Parallel / SIMD	Task Parallel / MIMD
Shared Memory (SM)	GPU, Cell, SSE, Vector processor ...	ManyCore/ SMP system ...
Shared Nothing / Distributed Memory (DM)	processor-array systems systolic arrays Hadoop ...	cluster systems MPP systems ...



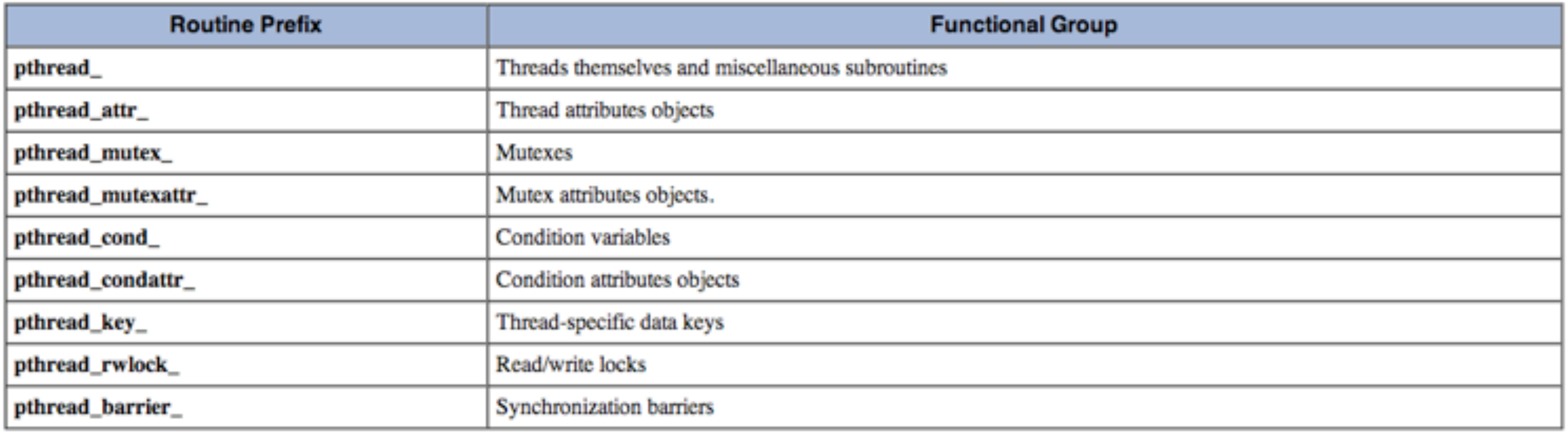
Multi-Tasking

Multi-Tasking	PThreads, OpenMP, OpenCL, Linda, Cilk, ...
Message Passing	MPI, PVM, CSP Channels, Actors, ...
Implicit Parallelism	Map/Reduce, PLINQ, HPF, Lisp, Fortress, ...
Mixed Approaches	Ada, Scala, Clojure, Erlang, X10, ...

POSIX 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, ...
  - **Mutex**-based synchronization
  - Synchronization based on **condition variables**
  - Synchronization based on **read/write locks** and **barriers**
- Semaphore API is a separate POSIX specification (*sem_* prefix)

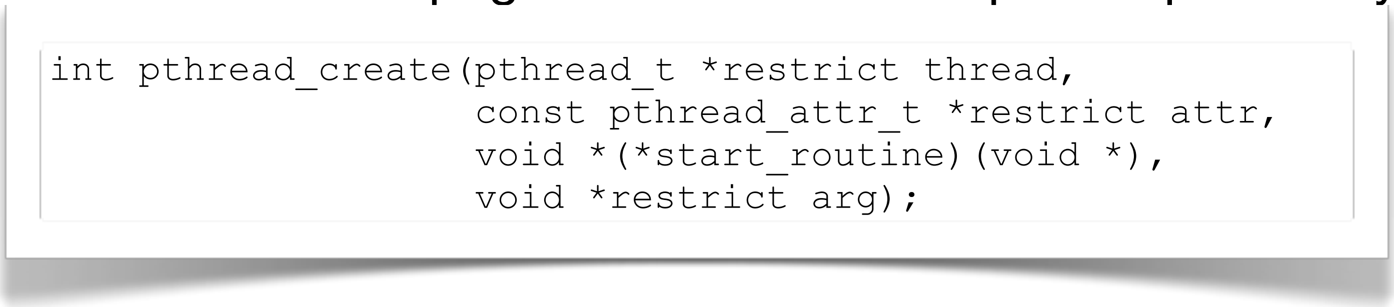
POSIX Pthreads



Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

POSIX Pthreads

- *pthread_create()*
  - Create new thread in the process, with given routine and argument
- *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 (f.e. stack size limit)
  - See discussion in man page about how this improves portability

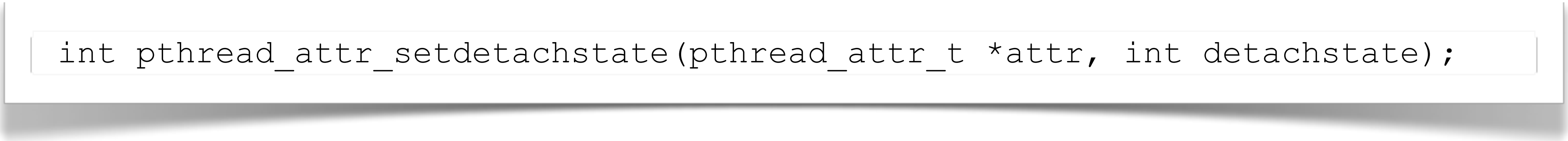


```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

```
/******************************************************************************
* 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){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

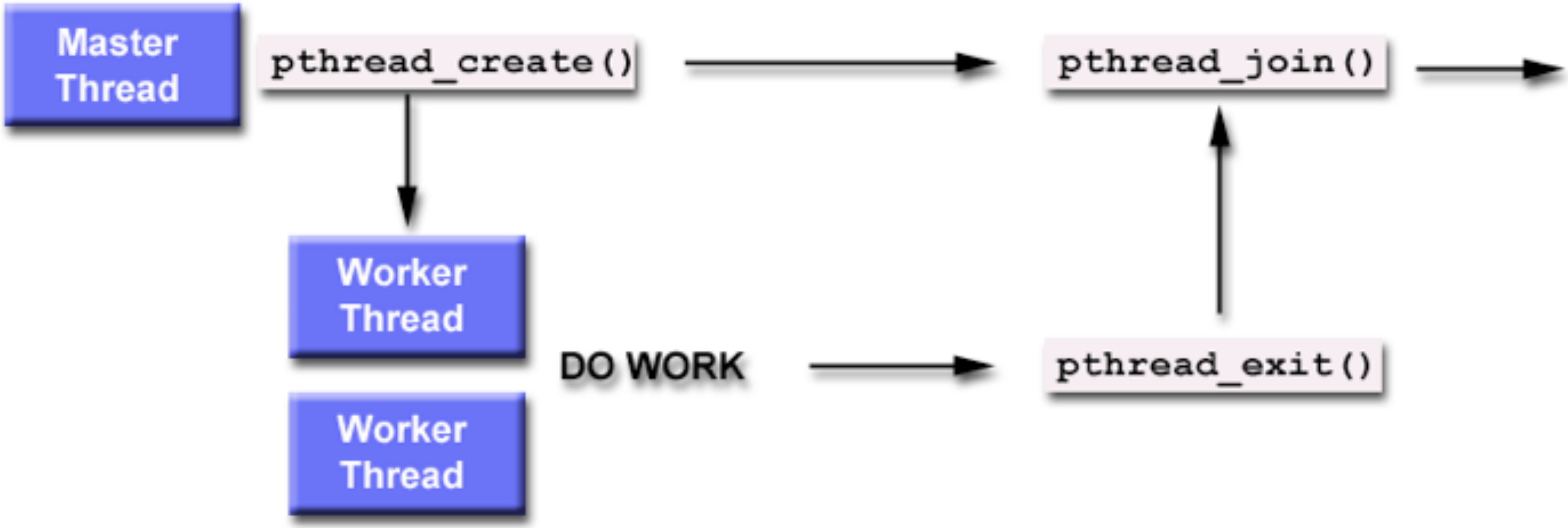
POSIX Pthreads

- *pthread_join()*
  - 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()*
  - Mark thread as not-joinable (*detached*) - may free some system resources
- *pthread_attr_setdetachstate()*
  - Prepare *attr* block so that a thread can be created in some detach state



```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

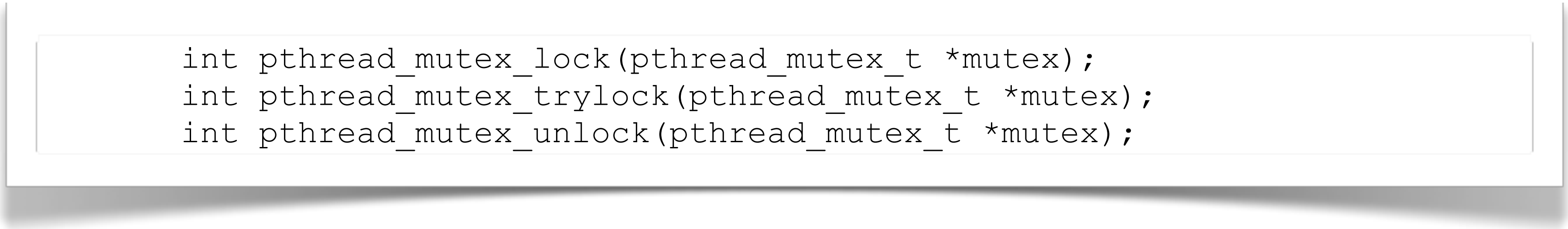
POSIX Pthreads



```
/*****************************************************************************
* FILE: join.c
* AUTHOR: 8/98 Blaise Barney
* LAST REVISED:  01/30/09
******************************************************************************/
#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 Pthreads

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



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

POSIX Pthreads

- Condition variables are always used in conjunction with a mutex
- Allow to wait on a variable change without polling it in a critical section
- *pthread_cond_init()*
  - Initializes a condition variable
- *pthread_cond_wait()*
  - Called with a locked mutex
  - Releases the mutex and blocks on the condition in one atomic step
  - One 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

```
/* FILE: condvar.c
 * LAST REVISED: 10/14/10  Blaise Barney
 */

#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);
}
```

```
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[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  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);

  /* Wait for all threads to complete */
  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);

}
```

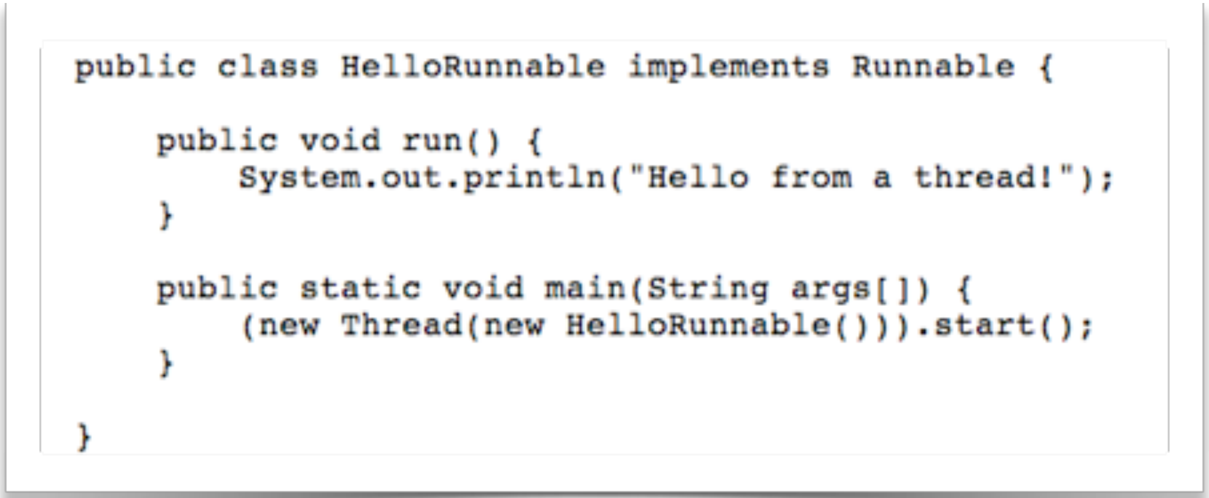
Windows vs. POSIX Synchronization

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

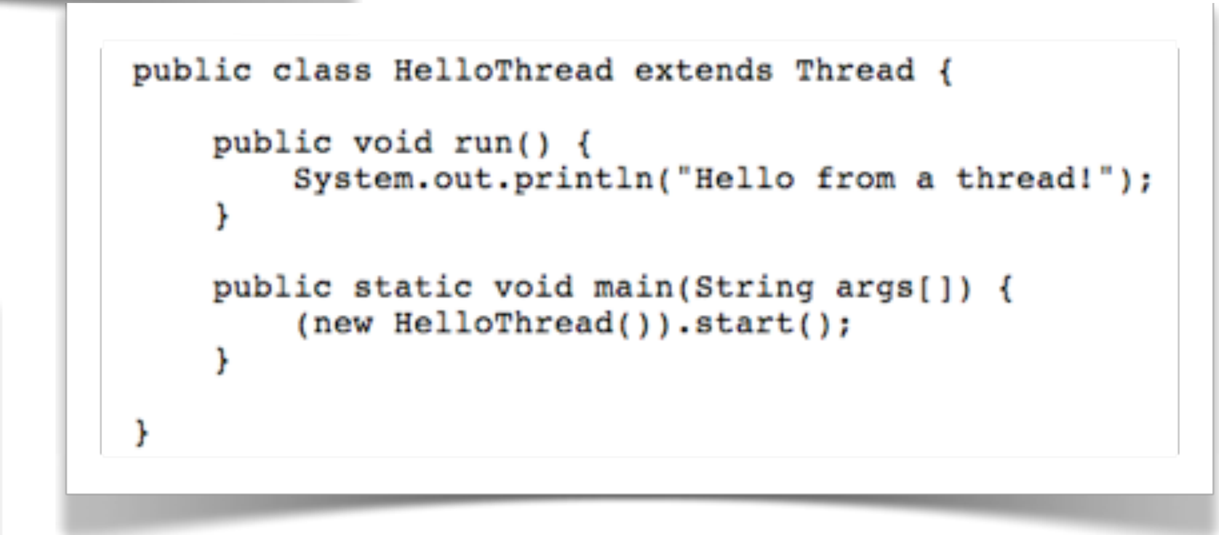
Java

- Java supports concurrency with Java / operating system threads
- Functions bundled in `java.util.concurrent`
- Classical concurrency support
  - `synchronized` methods: Allow only one thread in an objects‘ synchronized methods, based on intrinsic object lock
    - For static methods, locking based on class object
  - `synchronized` statements: Synchronize execution by intrinsic lock of the given object
  - `volatile` keyword: Indicate shared nature of variable - ensures atomic synchronized access, no thread-local caching
  - `wait` / `notify` semantics in `Object`

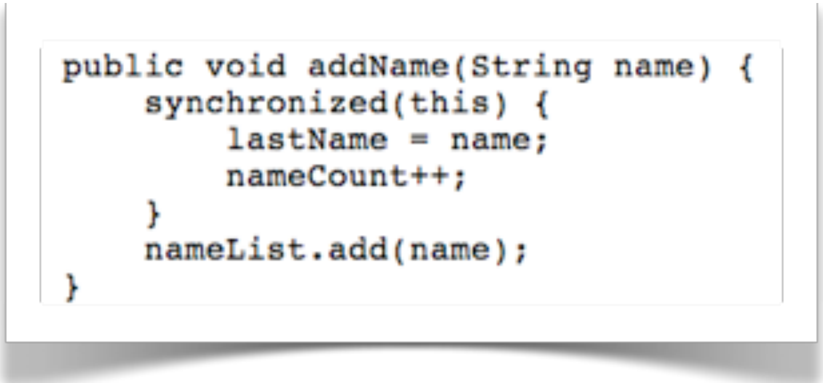
Java Examples



```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```



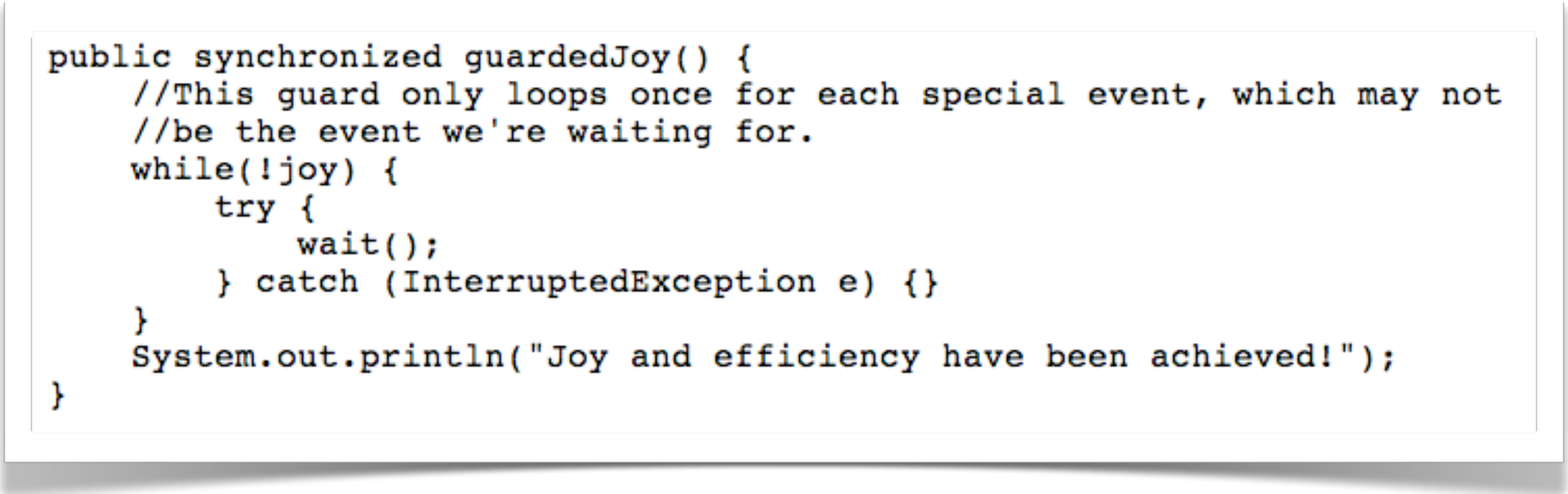
```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```



```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Java wait / notify

- Each object can act as guard with `wait()` / `notify()` functions
    - Guard waiting must always be surrounded by explicit condition check



```
public synchronized guardedJoy() {
    //This guard only loops once for each special event, which may not
    //be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

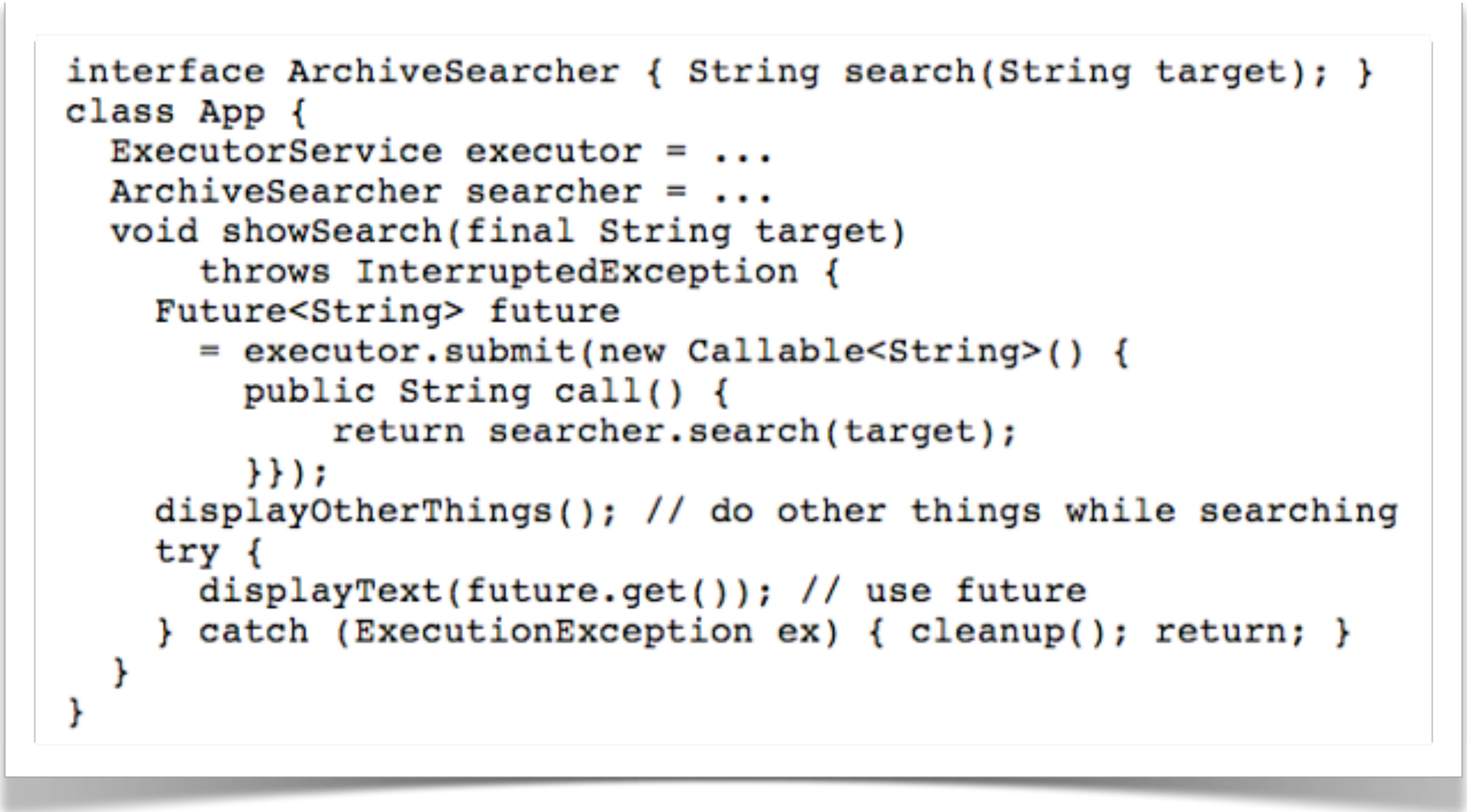
Java High-Level Concurrency

- Introduced with Java 5
- `java.util.concurrent.locks`
- Separation of thread management and parallel activities - *Executors*
  - `java.util.concurrent.Executor`
    - Implementing object provides `execute()` method, is able to execute submitted `Runnable` tasks
    - No assumption on where the task runs, might be even in the callers context, but typically in managed thread pool
    - `ThreadPoolExecutor` implementation provided by class library

Java High-Level Concurrency

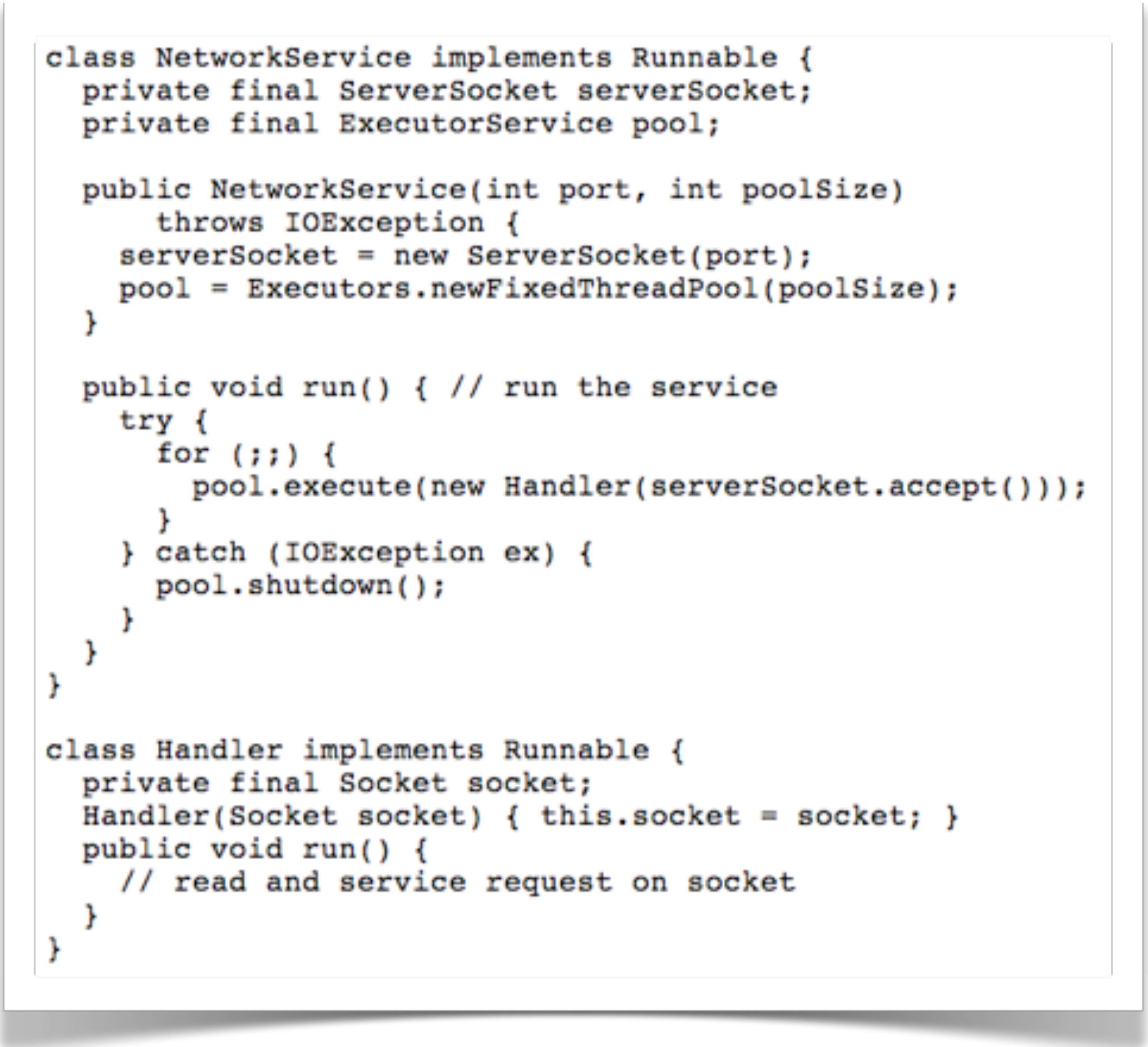
- `java.util.concurrent.ExecutorService`
  - Supports also `Callable` objects as input, which can return a value
  - Additional `submit()` function, which returns a `Future` object on the result
    - `Future` object allows to wait on the result, or cancel execution
  - Methods for submitting large collections of `Callable`‘s
  - Methods for managing executor shutdown
- `java.util.concurrent.ScheduledExecutorService`
  - Additional methods to schedule tasks repeatedly
- Available thread pools from executor implementations:
  Single background thread, fixed size, unbound with automated reclamation

Java High-Level Concurrency



```
interface ArchiveSearcher { String search(String target); }
class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...
  void showSearch(final String target)
      throws InterruptedException {
    Future<String> future
      = executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }});
    displayOtherThings(); // do other things while searching
    try {
      displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
  }
}
```

Java High-Level Concurrency



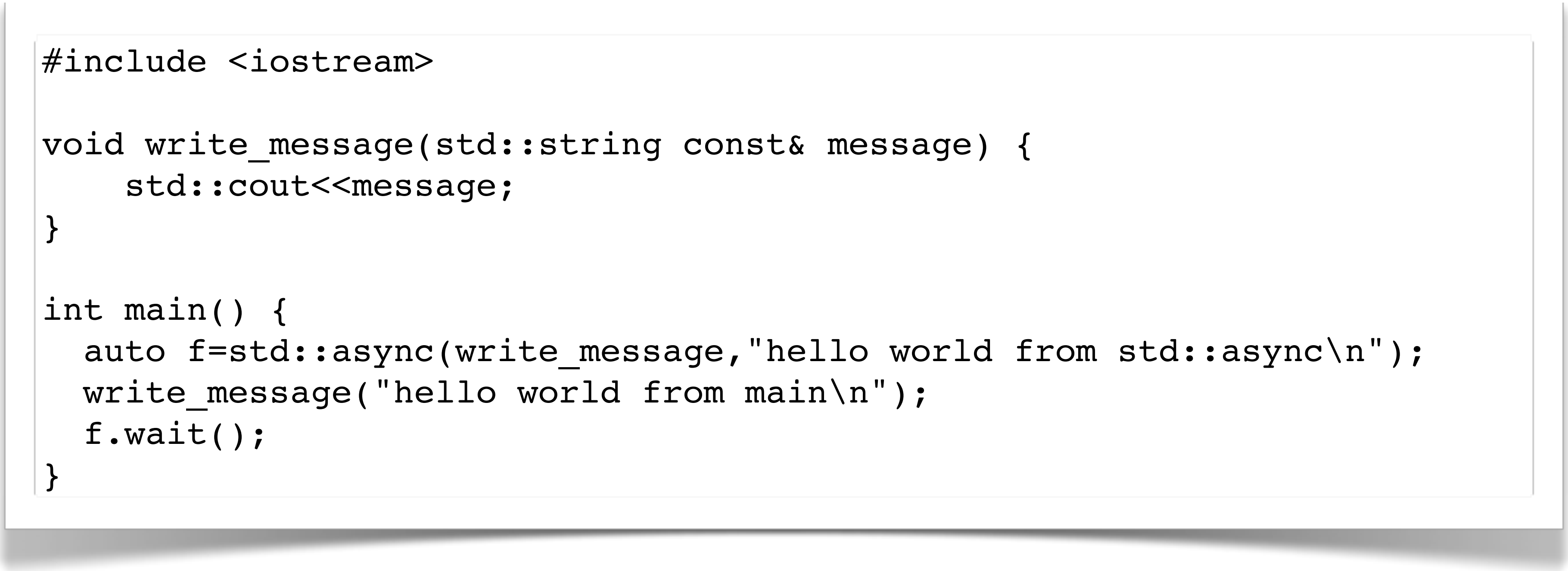
```
class NetworkService implements Runnable {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;

  public NetworkService(int port, int poolSize)
      throws IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void run() { // run the service
    try {
      for (;;) {
        pool.execute(new Handler(serverSocket.accept()));
      }
    } catch (IOException ex) {
      pool.shutdown();
    }
  }
}

class Handler implements Runnable {
  private final Socket socket;
  Handler(Socket socket) { this.socket = socket; }
  public void run() {
    // read and service request on socket
  }
}
```

.NET

- As Java, .NET CLR relies on native thread model
  - Synchronization and scheduling mapped to operating system concepts
- .NET 4 has variety of support libraries
  - *Task Parallel Library (TPL)* - Loop parallelization, task concept
  - Task factories, task schedulers
  - *Parallel LINQ (PLINQ)* - Implicit data parallelism through query language
  - Collection classes, synchronization support
  - Debugging and visualization support

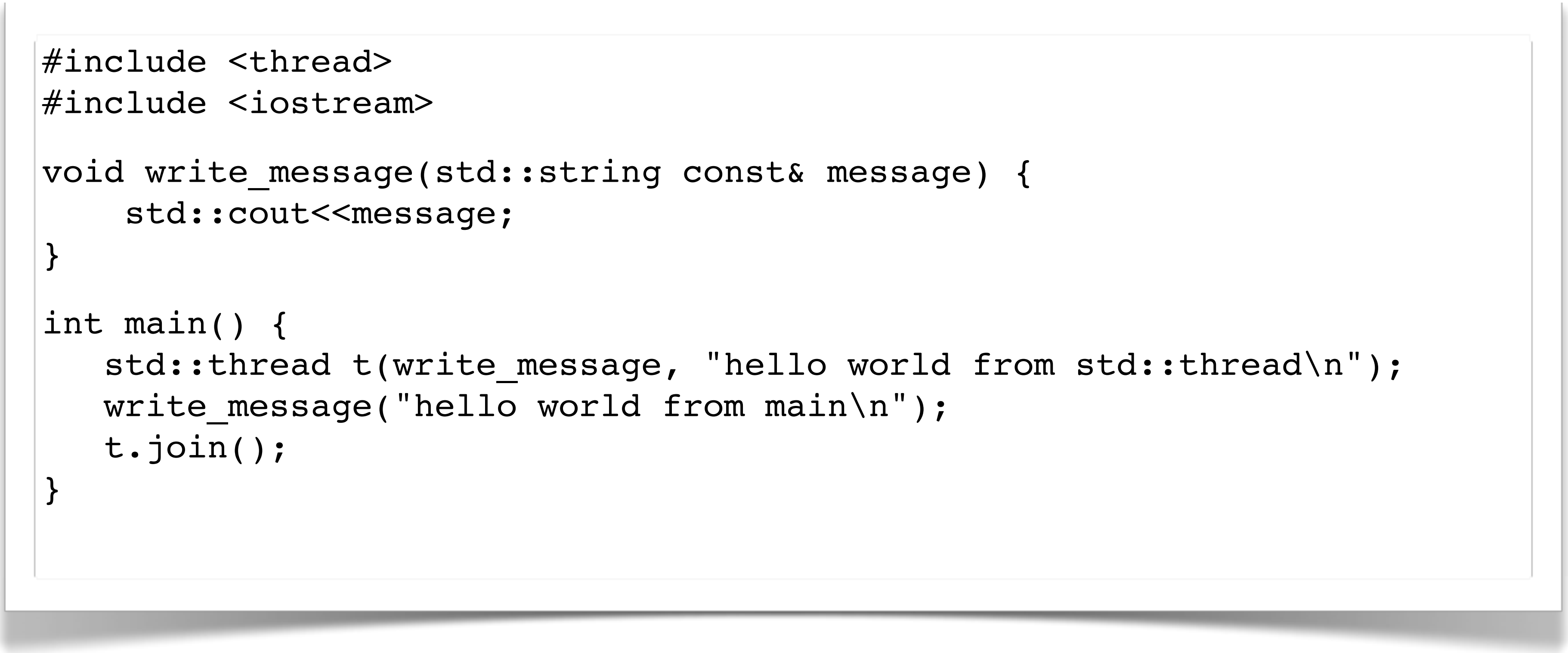
C++

- C++11 specification added support for **threads** and **mutexes**
- Spanning asynchronous tasks with *std::async* or *std::thread*
  - Works with *Callable* instance (functions, member functions, ...)



```
#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();
}
```

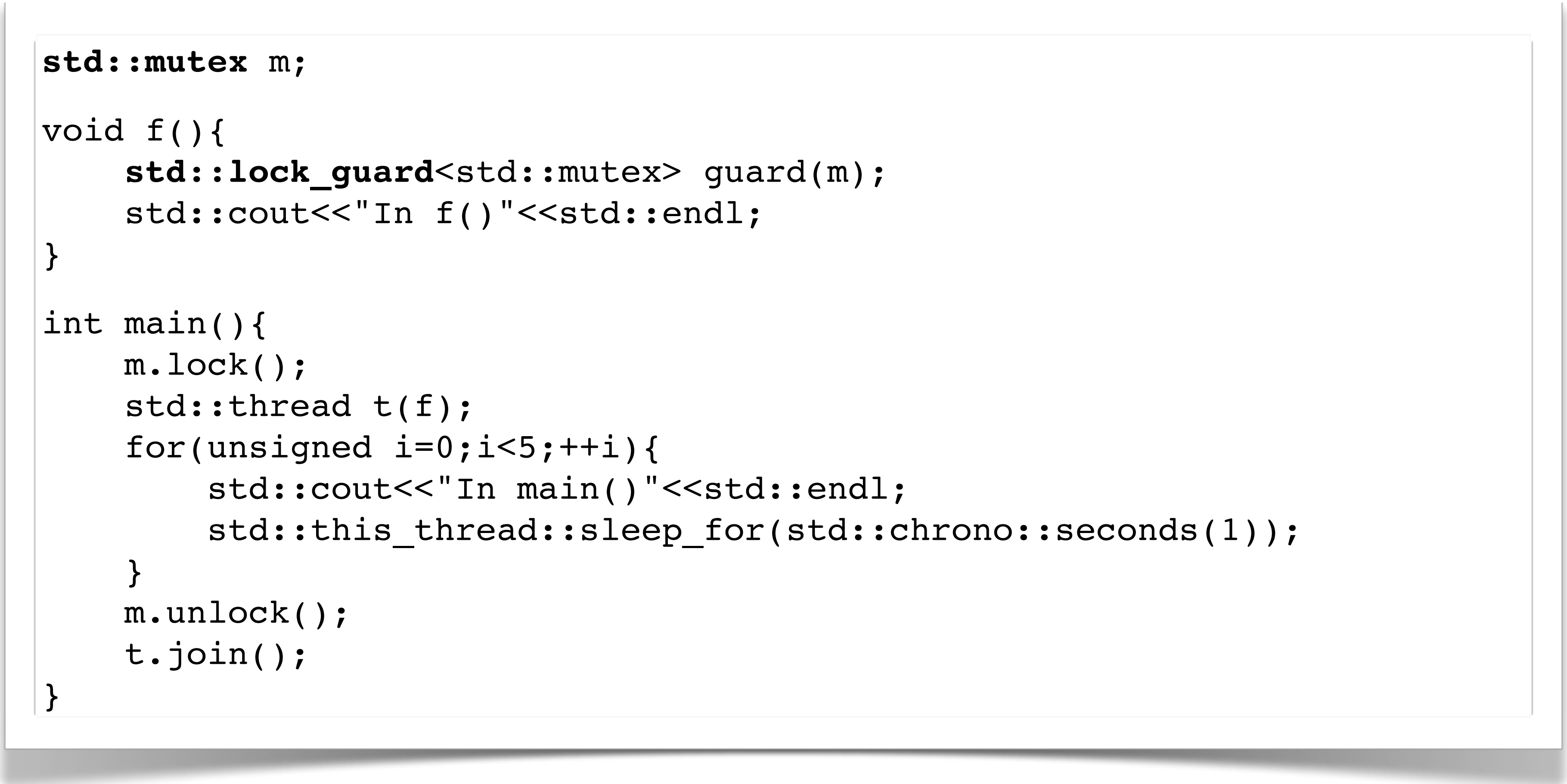
Concurrent Programming in C++



```
#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();
}
```

- Launch policy can be specified
- *get()* method can be used to get the async function call result („*future*“)

Concurrent Programming in C++



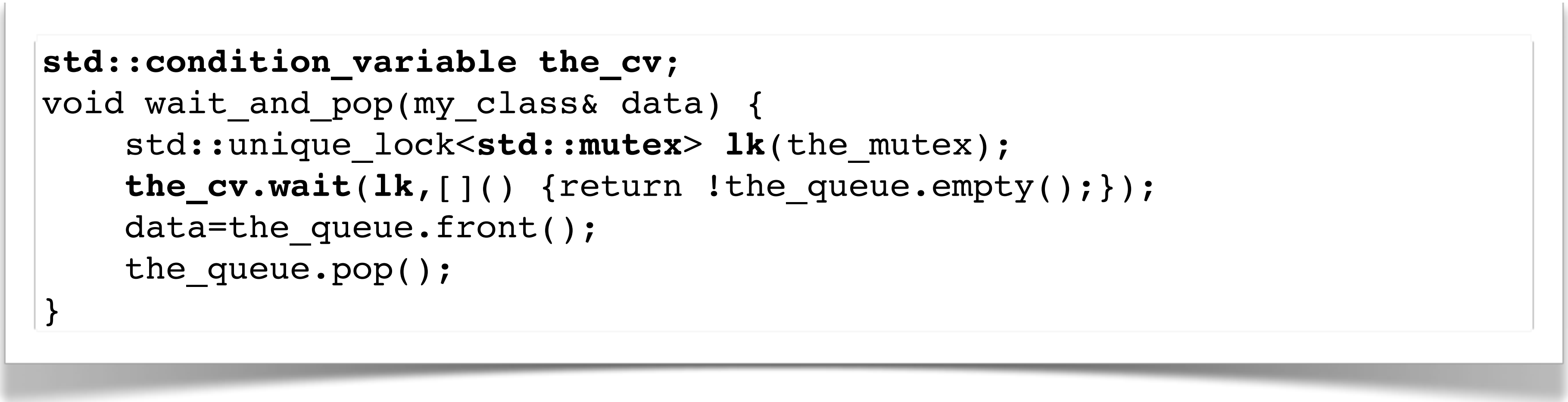
```
std::mutex m;

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();
}
```

- 4 mutex classes, basic operations in the *Lockable* concept: *m.lock(), m.try_lock(), m.unlock()*

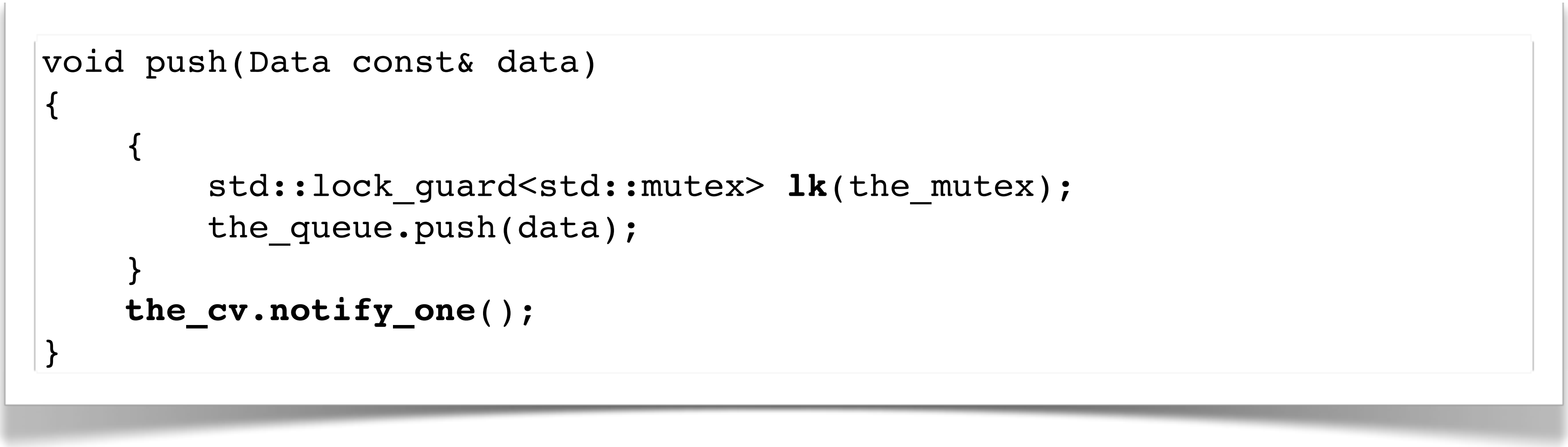
- Locking is tricky with exceptions, so C++ offers some high-level templates

Concurrent Programming in C++

- Waiting for events with condition variables avoids polling



```
std::condition_variable the_cv;
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();
}
```

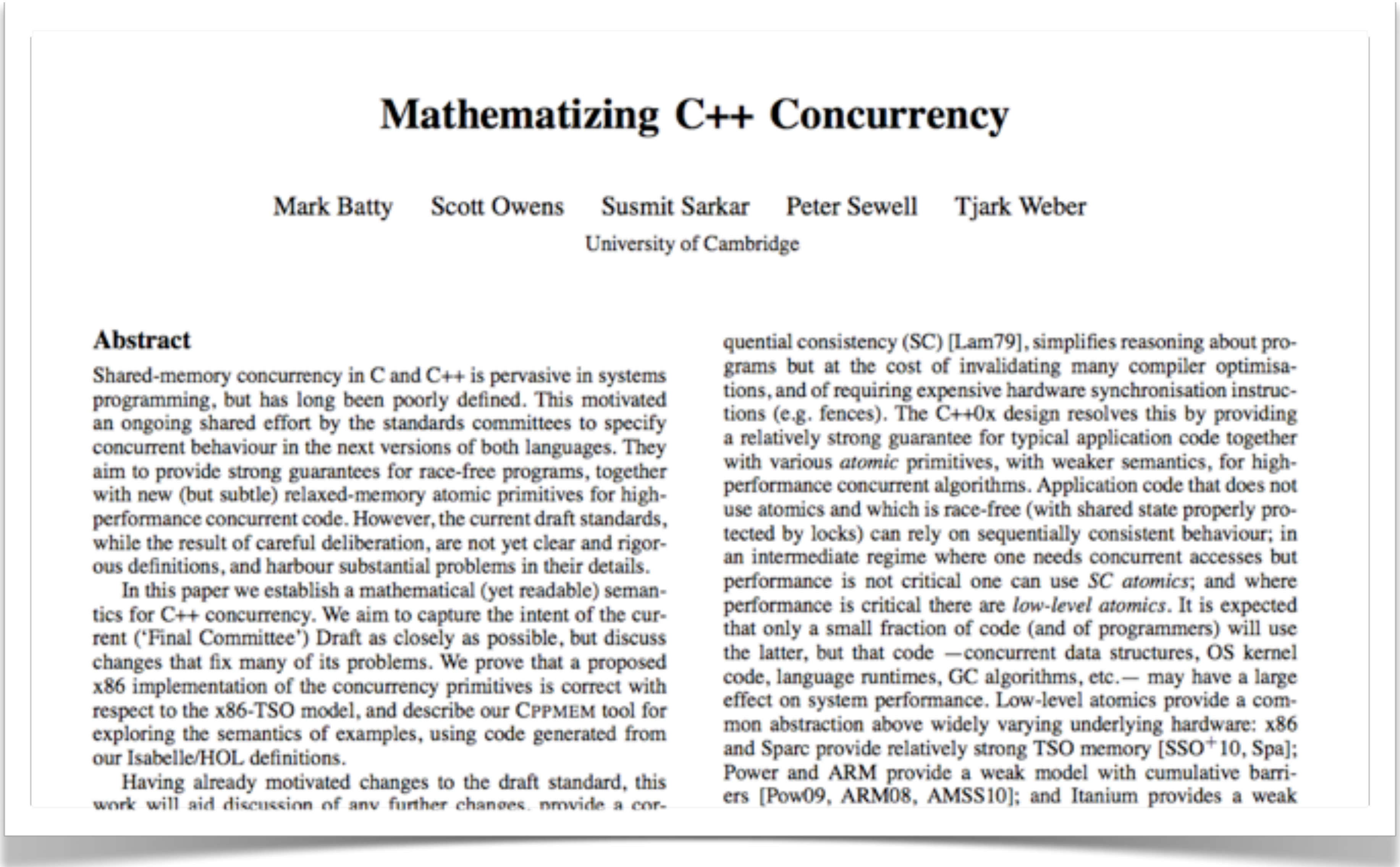


```
void push(Data const& data)
{
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    the_cv.notify_one();
}
```

Concurrent Programming in C++

- Lock-free atomic types that are three from data races
  - *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+=, ...*

Concurrent Programming in C++



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 cor-

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

Threads vs. Tasks

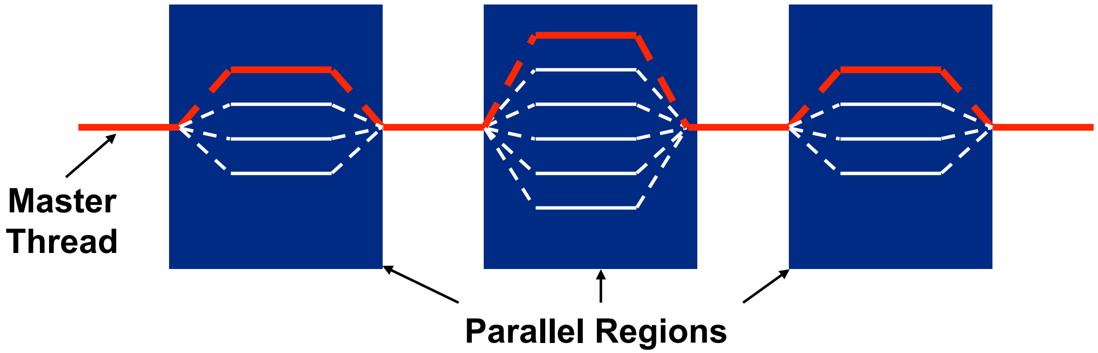
- **Process:** Address space, handles, code, set of threads
- **Thread:** control flow
  - Preemptive scheduling by the operating system
  - Can migrate between cores
- **Task:** control flow
  - Typically modeled as object (TBB, Java) or statement / lambda expression / anonymous function (OpenMP, MS TPL)
  - Cooperative scheduling by a user-mode library, mapping to thread pool
  - Task model replaces context switch with **yielding** approach
  - Typical scheduling policy for tasks is **central queue** or **work stealing**

Multi-Tasking

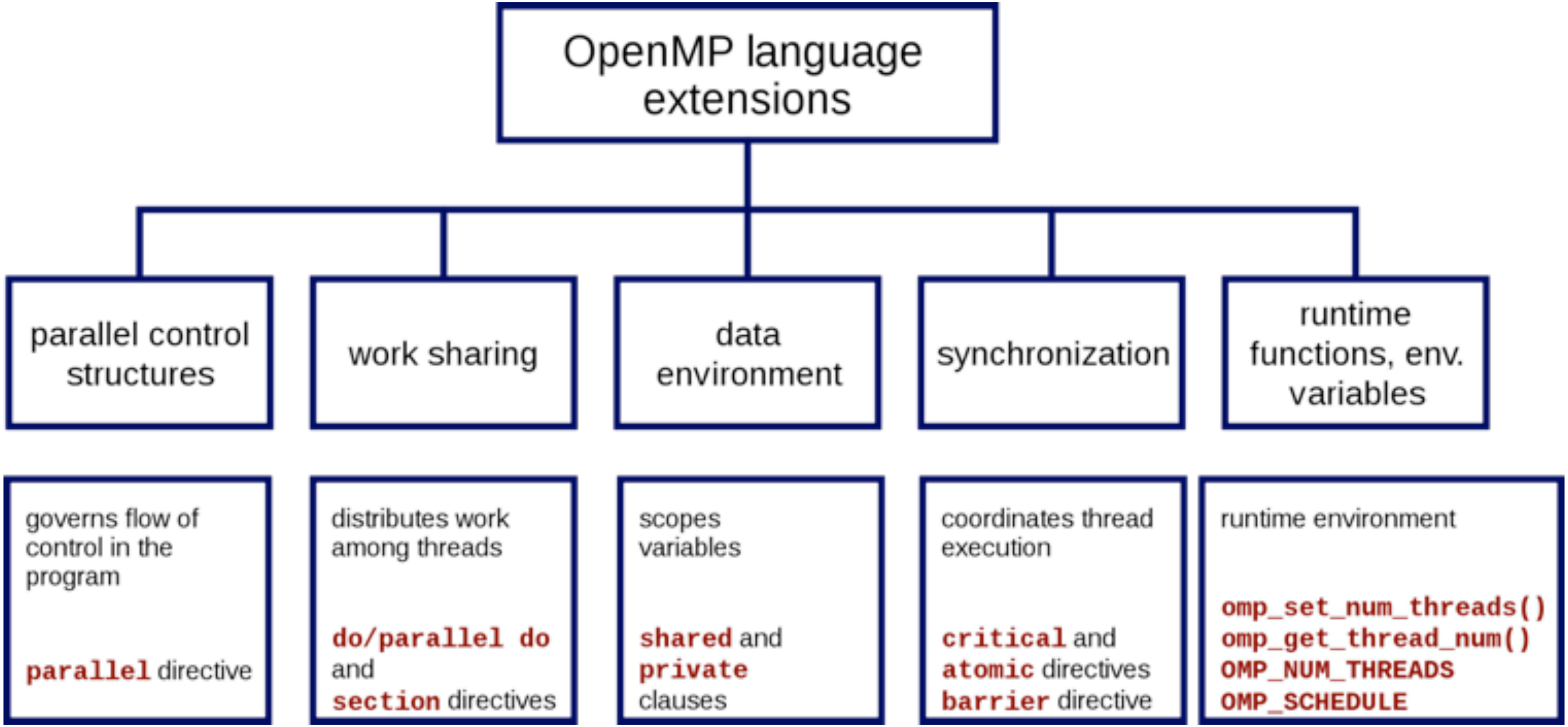
- Relevant issues: Task generation, execution synchronization, data access
  - Manual coordination in a sequential language
    (operating system threads, Java / .NET threads, ...)
    -> *„explicit“ threading*
  - Using a framework for parallel tasks
    (OpenMP, OpenCL, Intel TBB, MS TPL, ...)
    -> *„implicit“ threading*
- Concurrency problems remain
  - Critical section problem with shared variables in different tasks
  - Low-level synchronization primitives wrapped by „concurrent data structures“ in task framework
- Already covered: OpenCL

OpenMP

- Specification for C/C++ and Fortran language extension (currently v3.1)
  - 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 source code
- Programming model: Fork-Join-Parallelism
  - Master thread spawns group of threads for limited code region
    - PARALLEL directive
    - Barrier concept



OpenMP



OpenMP language extensions

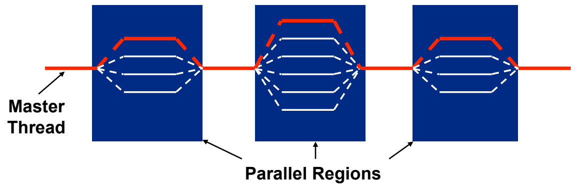
parallel control structures	work sharing	data environment	synchronization	runtime functions, env. variables
governs flow of control in the program	distributes work among threads	scopes variables	coordinates thread execution	runtime environment

`parallel` directive	`do/parallel do` and `section` directives	`shared` and `private` clauses	`critical` and `atomic` directives
			`barrier` directive
				`omp_set_num_threads()`
				`omp_get_thread_num()`
				`OMP_NUM_THREADS`
				`OMP_SCHEDULE`

(from Wikipedia)

OpenMP Pragmas

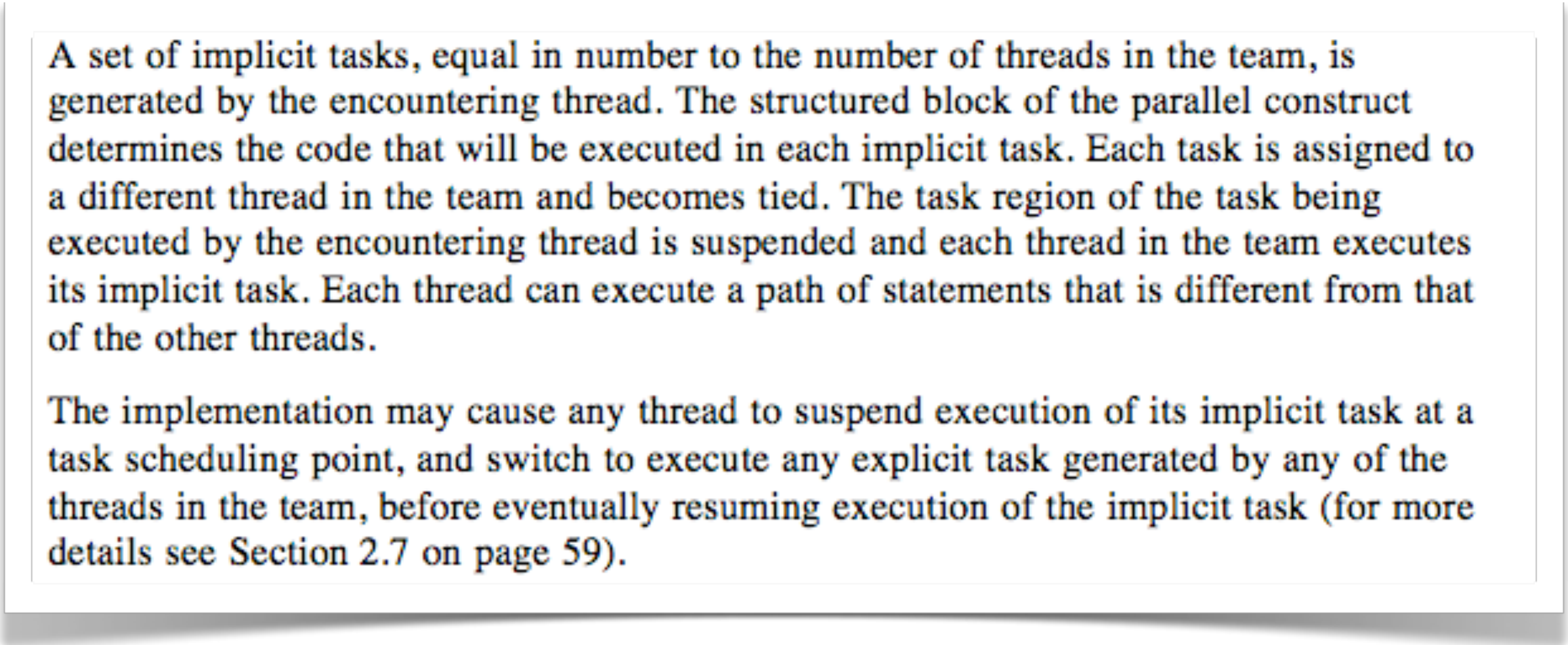
- `#pragma omp` *`construct`* `...` (include `omp.h`)
- OpenMP runtime library: query functions, runtime functions, lock functions
- Parallel region
  - OpenMP constructs are applied to dedicated code blocks, marked by `#pragma omp parallel`
  - Parallel region should have only one entry and one exit point
  - Implicit barrier at beginning and end of the block
- Thread pool for execution of parallel activities



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

OpenMP Parallel Construct

- Encountering **thread** for the parallel region generates a set of implicit **tasks**
- Each resulting implicit **task** is assigned to a different **thread**
- Implementation may suspend **task execution** at a scheduling point



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 details see Section 2.7 on page 59).

OpenMP Configuration / 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 Work Sharing

- Possibilities for distribution of tasks across threads (‚work sharing')
  - `omp sections` - Define code blocks usable as tasks
  - `omp for` - Automatically divide a loop's iterations into tasks
    - Implicit barrier at the end
  - `omp task` - Explicitly define a task
  - `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)
  - Scheduling of tasks defined is handled by the OpenMP implementation
- Clause combinations possible: `#pragma omp parallel for`

OpenMP Work Sharing with Sections

- Explicit definition of code blocks as parallel tasks with **section** directive (function partitioning)
- Executed in the context of the implicit task
- One task may execute more than one section - runtime decision

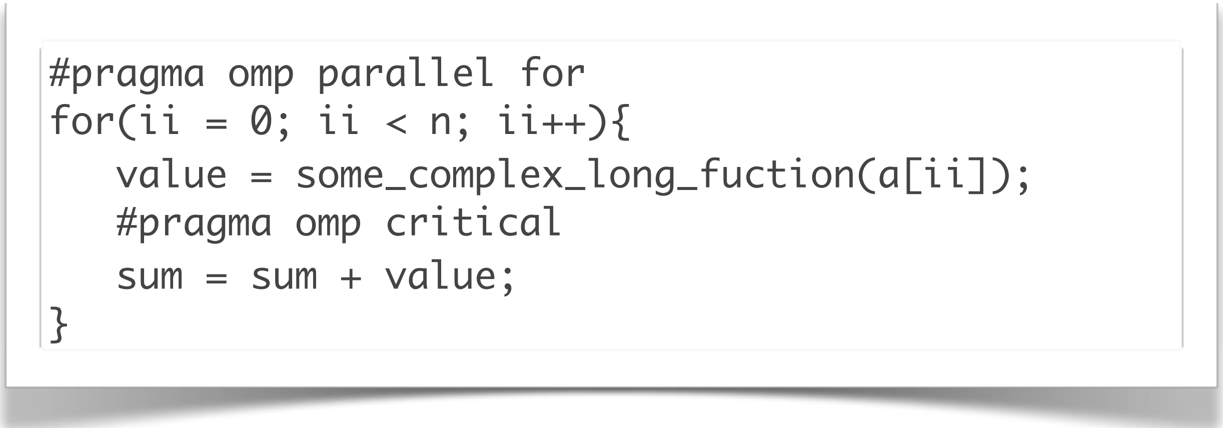
```
#pragma omp parallel
{
  #pragma omp sections [ clause [ clause ] ... ]
  {
    [#pragma omp section ]

      structured-block1

    [#pragma omp section ]

      structured-block2
}}
```

OpenMP Data Scoping

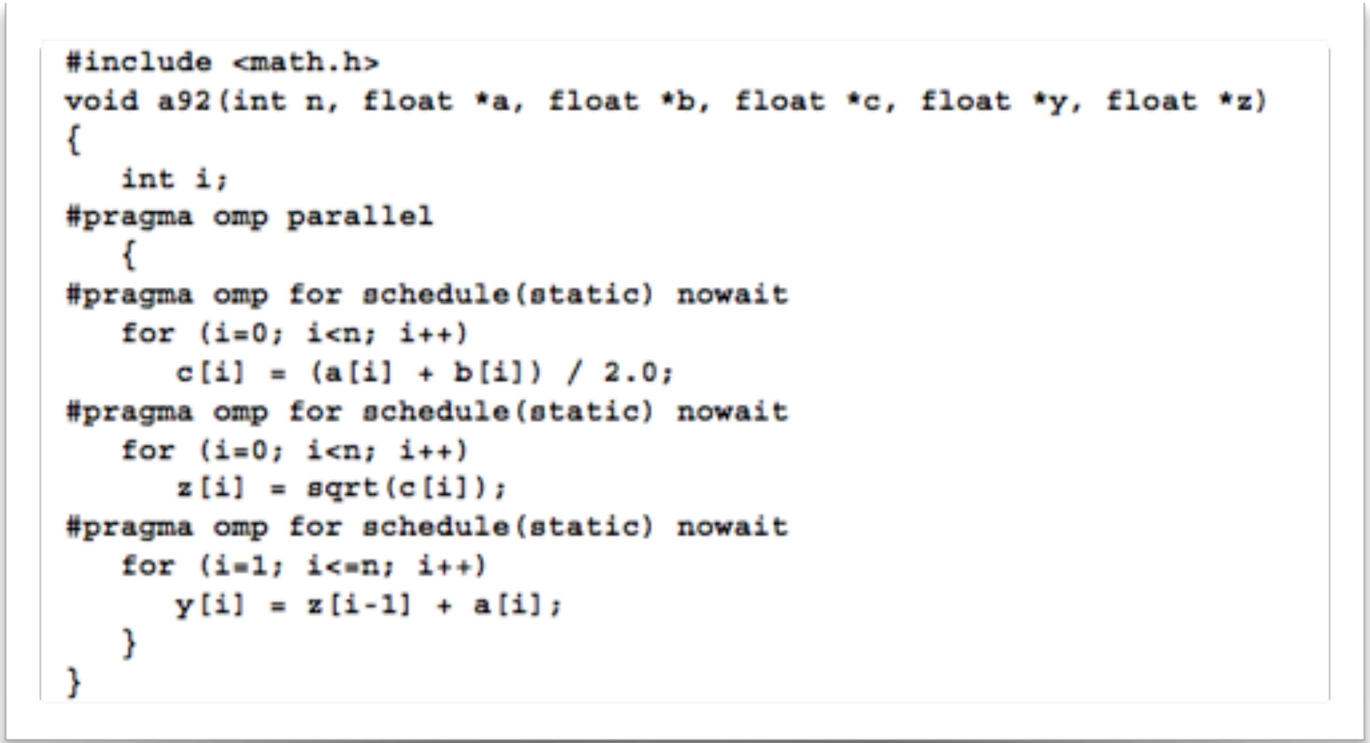
- Shared memory programming model - communication through variables
- **Shared variable**: Name provides access to same memory in all tasks
  - Shared by default: global variables, static variables,
    variables with namespace scope, variables with file scope
  - `shared` clause can be added to any `omp` construct, defines a list of additionally shared variables
  - Provides no automatic protection, just marking of variables for handling by runtime environment
- **Private variable:** Clone variable in each task, by default no initialization
  - Private by default: Local variables in functions called from parallel regions, loop iteration variables, automatic variables
  - Initialization with last value before region (`firstprivate`) possible

OpenMP Work Sharing with Loop Parallelization

- Loop construct: Parallel execution of iterations
- Iteration variable must be integer
- Mapping of threads to iterations is controlled by `schedule` clause
- Implications on exception handling, break-out calls and *continue* primitive



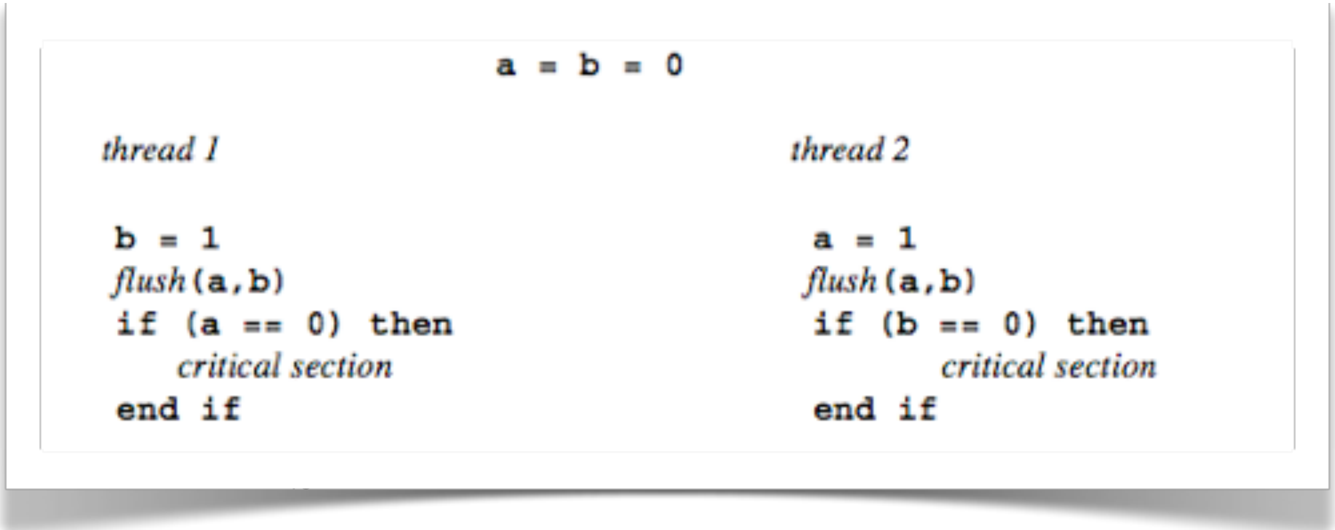
```
#pragma omp parallel for
for(ii = 0; ii < n; ii++){
   value = some_complex_long_fuction(a[ii]);
   #pragma omp critical
   sum = sum + value;
}
```



```
#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 Consistency Model

- 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 reasons
  - Compiler needs to be informed when consistent view is demanded
  - Implicit flush on different occasions, such as barrier region
  - In all other cases, shared variables must be flushed before reading
- Directive:
  `#pragma omp flush`



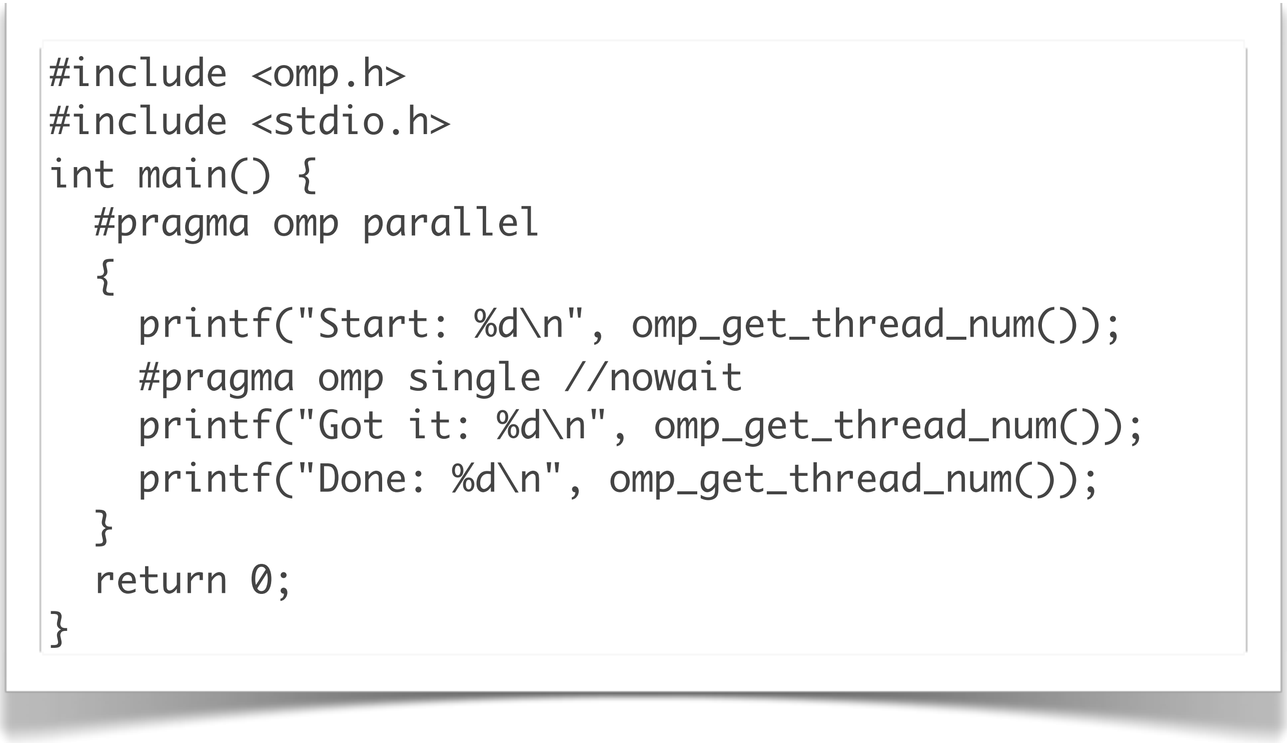
```
                         a = b = 0

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 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
- `schedule (dynamic, [chunk])` - Threads grab iteration resp. chunk
    - Higher overhead, but good for unbalanced iteration work load
- `schedule (guided, [chunk])` - Dynamic schedule, shrinking ranges per step, starting with large block, until minimum chunk size is reached
    - Computations with increasing iteration length (e.g. prime sieve test)

OpenMP Synchronization

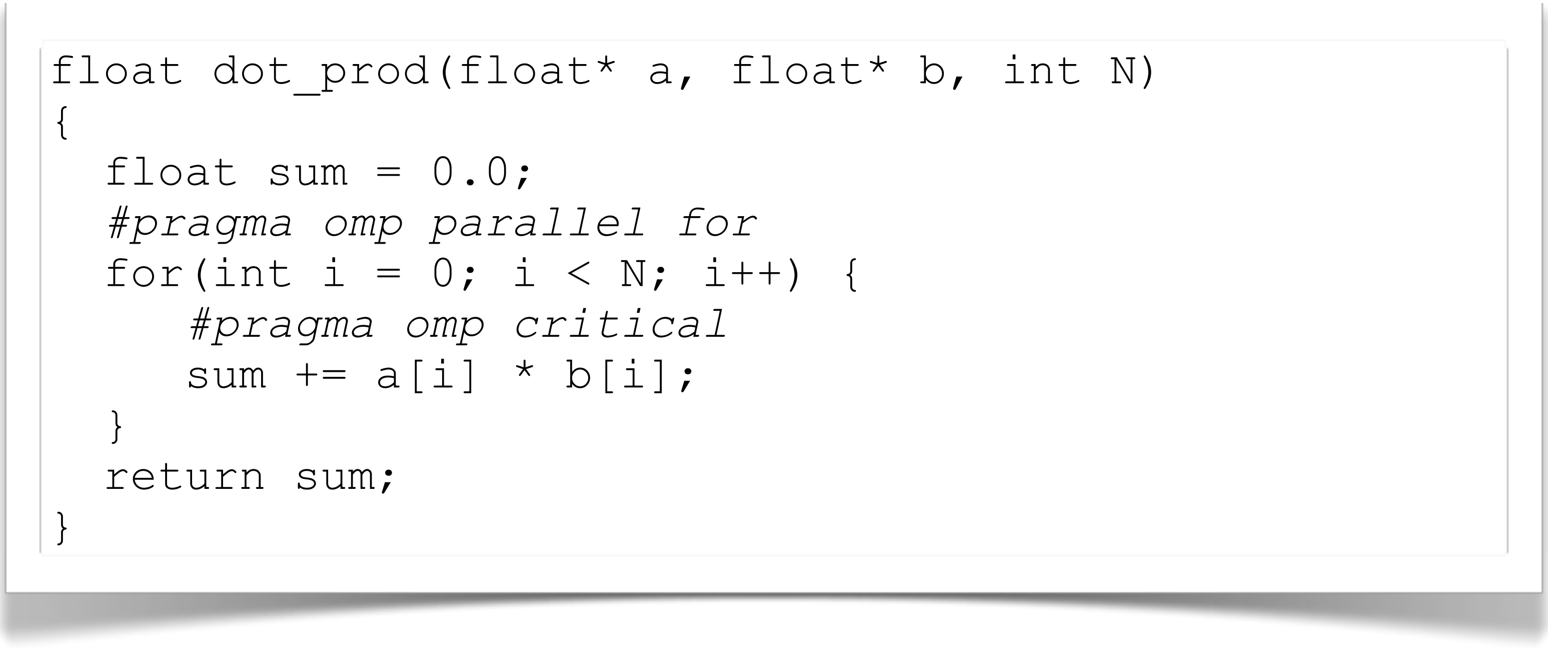
- Synchronizing with task completion
  - Implicit barrier at the end of `single` block, removable by `nowait` clause
  - `#pragma omp barrier` (wait for all other threads in the team)
  - `#pragma omp taskwait` (wait for completion of created child tasks)



```
#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

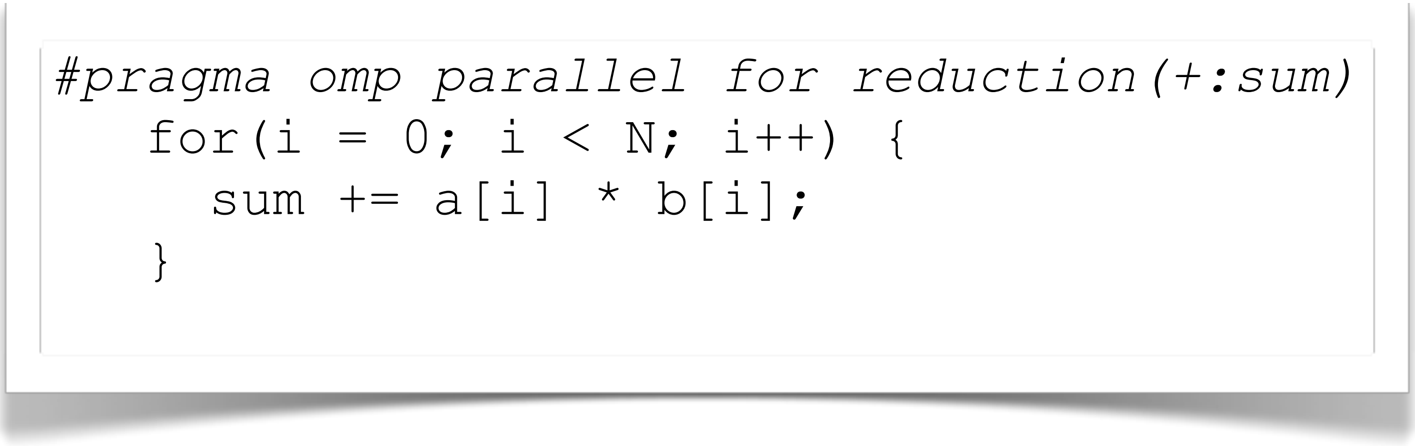
- Synchronizing variable access
  - `#pragma omp critical [name]`
    - Enclosed block executed by all threads, but restricted to one at a time
    - All unnamed directives map to the same unspecified name



```
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



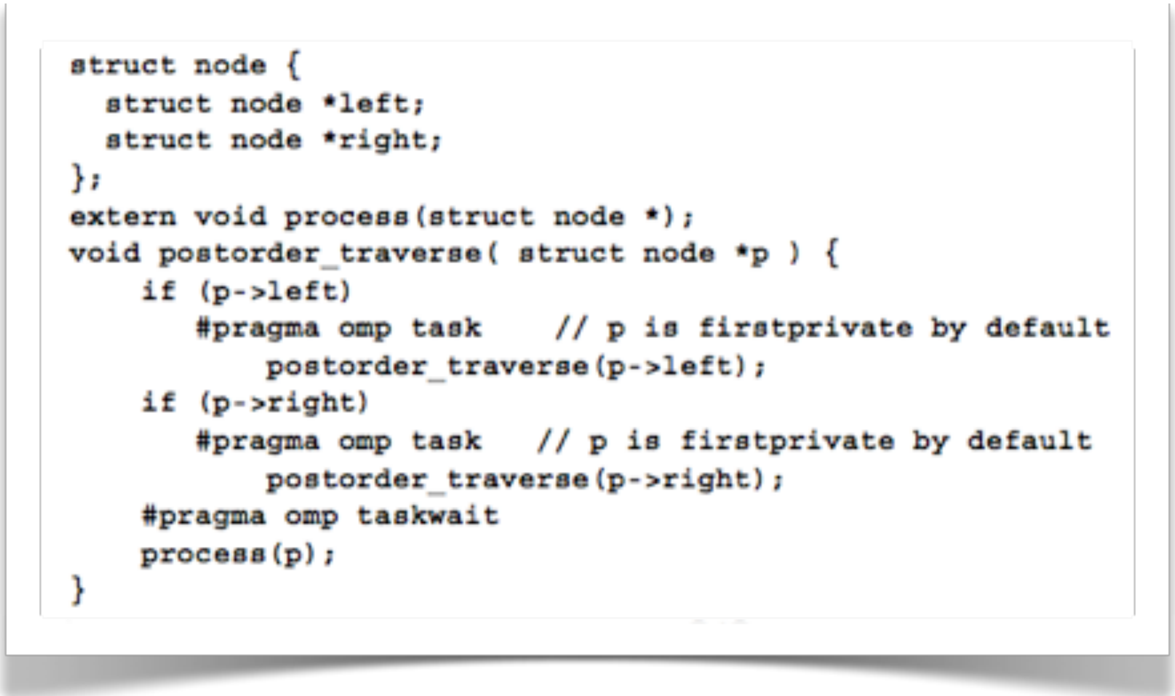
```
#pragma omp parallel for reduction(+:sum)
   for(i = 0; i < N; i++) {
     sum += a[i] * b[i];
   }
```

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`

OpenMP Tasks

- Main change with OpenMP v3, allows description of non-data driven parallelization strategy
  - Farmer / worker algorithms
  - Recursive algorithms
  - Unbounded loops
    (e.g. while loops)

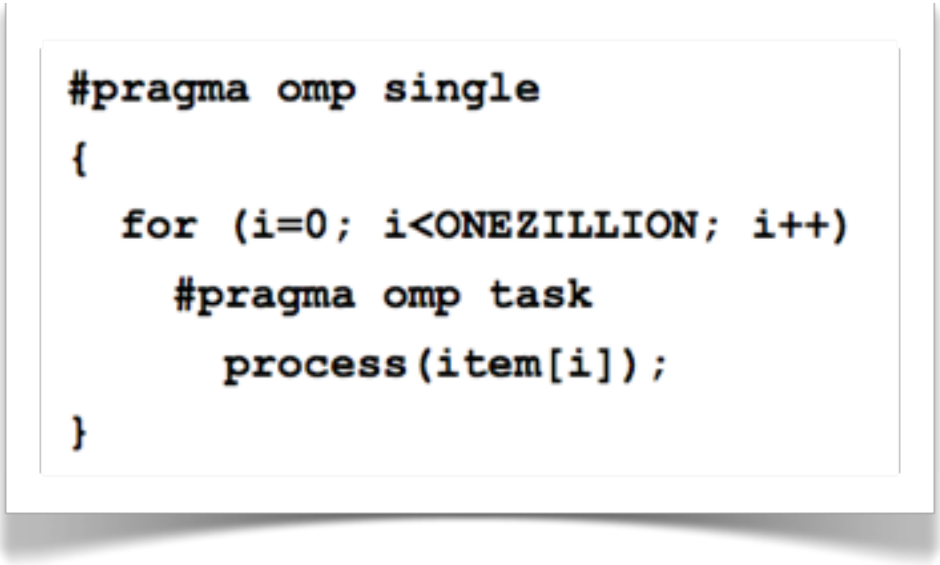


```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
       #pragma omp task    // p is firstprivate by default
           postorder_traverse(p->left);
    if (p->right)
       #pragma omp task   // p is firstprivate by default
           postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

- Definition of tasks as composition of code to execute, data environment, and control variables
- Implicit task generation with `parallel` and `for` constructs
- Explicit task generation with `sections` and `task` constructs

OpenMP Tasks

- Certain construct act as task scheduling point
  - When thread encounters this construct, it can switch to another task



```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Example: Generating task will suspend on exit barrier
  - Executing thread can help processing the task pool, or run the spawned task directly (cache-friendly)

OpenMP 4

- SIMD extensions
    - Allows to tell the compiler that it should vectorize a loop
- 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

Intel TBB

- *Task concept* - define what to run in parallel, instead of managing threads
- Portable C++ library, toolkits for different operating systems
- Complements basic OpenMP features
  - Loop parallelization, parallel reduction, synchronization, explicit tasks
- High-level concurrent containers (hash map, queue, vector)
- High-level parallel operations (prefix scan, sorting, data-flow pipelining)
- Unfair scheduling approach, to favor threads having data in cache
- Supported for cache-aware memory allocation
- Comparable: Microsoft C++ Concurrency Runtime

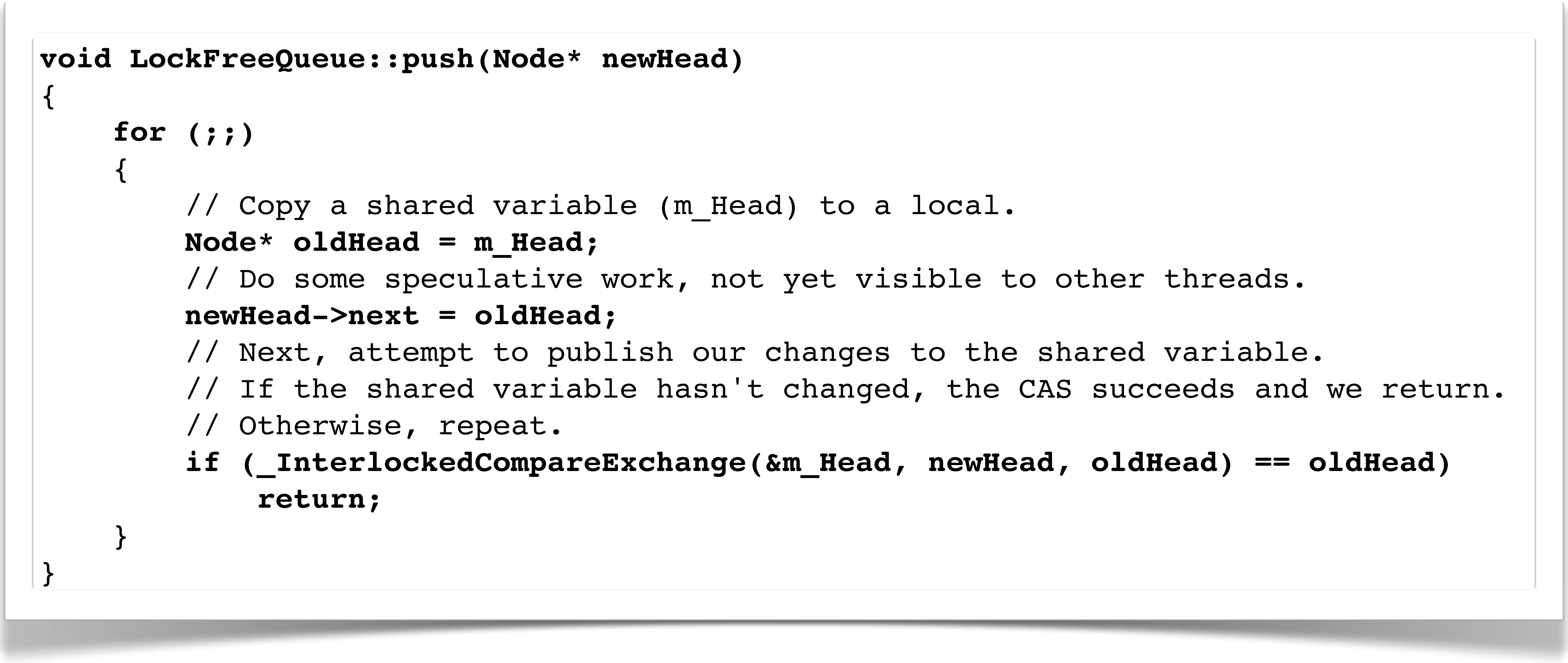
Easy Mappings [Dig]

	Java	TBB	TPL
Parallel For	ParallelArray	parallel_for	Parallel.For
Concurrent Collections	ConcurrentHashMap, ...	concurrent_hash_map, ...	
Atomic Classes	AtomicInteger, ...	atomic<T>	Interlocked
ForkJoin Task Parallelism	ForkJoinTask framework	task	Task, ReplicableTask

Lock-Free Programming

- Lock-free programming as a way of sharing data without maintaining locks
  - Prevents deadlock and live-lock conditions
  - Goal:
    Suspension of one thread never prevents another thread from making progress (e.g. synchronized shared queue)
  - Blocking by design does not disqualify the lock-free realization
- Algorithms rely on well-known hardware support for atomic operations
  - *Read-Modify-Write (RMW)* operations
  - *Compare-And-Swap (CAS)* operations
- These operations are typically mapped in operating system API

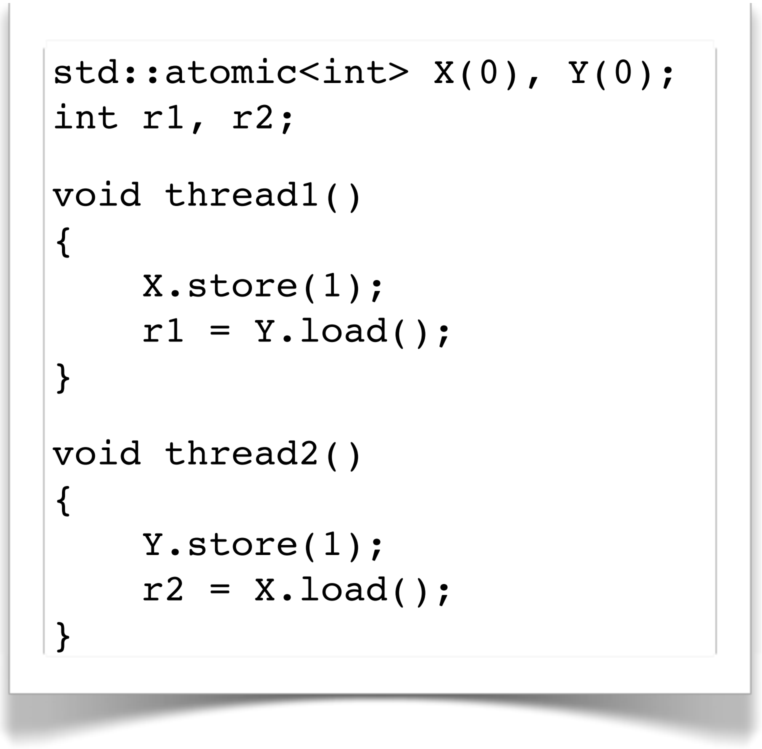
Lock-Free Programming



```
void LockFreeQueue::push(Node* newHead)
{
    for (;;)
    {
        // Copy a shared variable (m_Head) to a local.
        Node* oldHead = m_Head;
        // Do some speculative work, not yet visible to other threads.
        newHead->next = oldHead;
        // Next, attempt to publish our changes to the shared variable.
        // If the shared variable hasn't changed, the CAS succeeds and we return.
        // Otherwise, repeat.
        if (_InterlockedCompareExchange(&m_Head, newHead, oldHead) == oldHead)
            return;
    }
}
```

Sequential Consistency

- Consistency model where the order of memory operations is consistent with the source code
  - Important for lock-free algorithm semantic
  - Not guaranteed by some processor architectures (e.g. PowerPC)
- Java and C++ support the enforcement of sequential consistency
  - Compiler generates additional *memory fences* and *RMW* operations
  - Still does not prevent from memory re-ordering due to instruction re-ordering by the compiler itself



```
std::atomic<int> X(0), Y(0);
int r1, r2;

void thread1()
{
    X.store(1);
    r1 = Y.load();
}

void thread2()
{
    Y.store(1);
    r2 = X.load();
}
```

Work Stealing

- *Blumofe, Robert D.; Leiserson, Charles E.: Scheduling Multithreaded Computations by Work Stealing. In: In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS. 1994), S. 356-368*
- Problem of scheduling scalable multithreading problems on processors
- Work sharing: When processors create new work, the scheduler migrates threads for balanced utilization
- Work stealing: Underutilized processor takes work from other processor
  - Intuitively, less thread migrations
  - Goes back to work stealing research in Multilisp (1984)
  - Communication reaches lower bound for parallel divide-and-conquer
  - Approach by Blumofe et al. relies on „fully strict multithreaded computation“

Randomized Work Stealing

- Lock-free ready dequeue per processor
  - Task are inserted on the bottom, and can be taken from both sides
  - Processor obtains local work by taking one task from the bottom
  - Migrated tasks are taken from the top
  - If no ready task is available on task stall / dead, the processor steals the top-most one from a randomly chosen processor
    - If no victim is available, processor continues to randomly search for one
  - Stalled tasks which are enabled again by other tasks are placed on the bottom of the ready dequeue of their enabling processor
- Algorithm maintains the busy-leaves property - ready tasks are either executed or wait for a processor becoming free
- Supported in Microsoft TPL, Intel TBB, Java, Cilk, ...

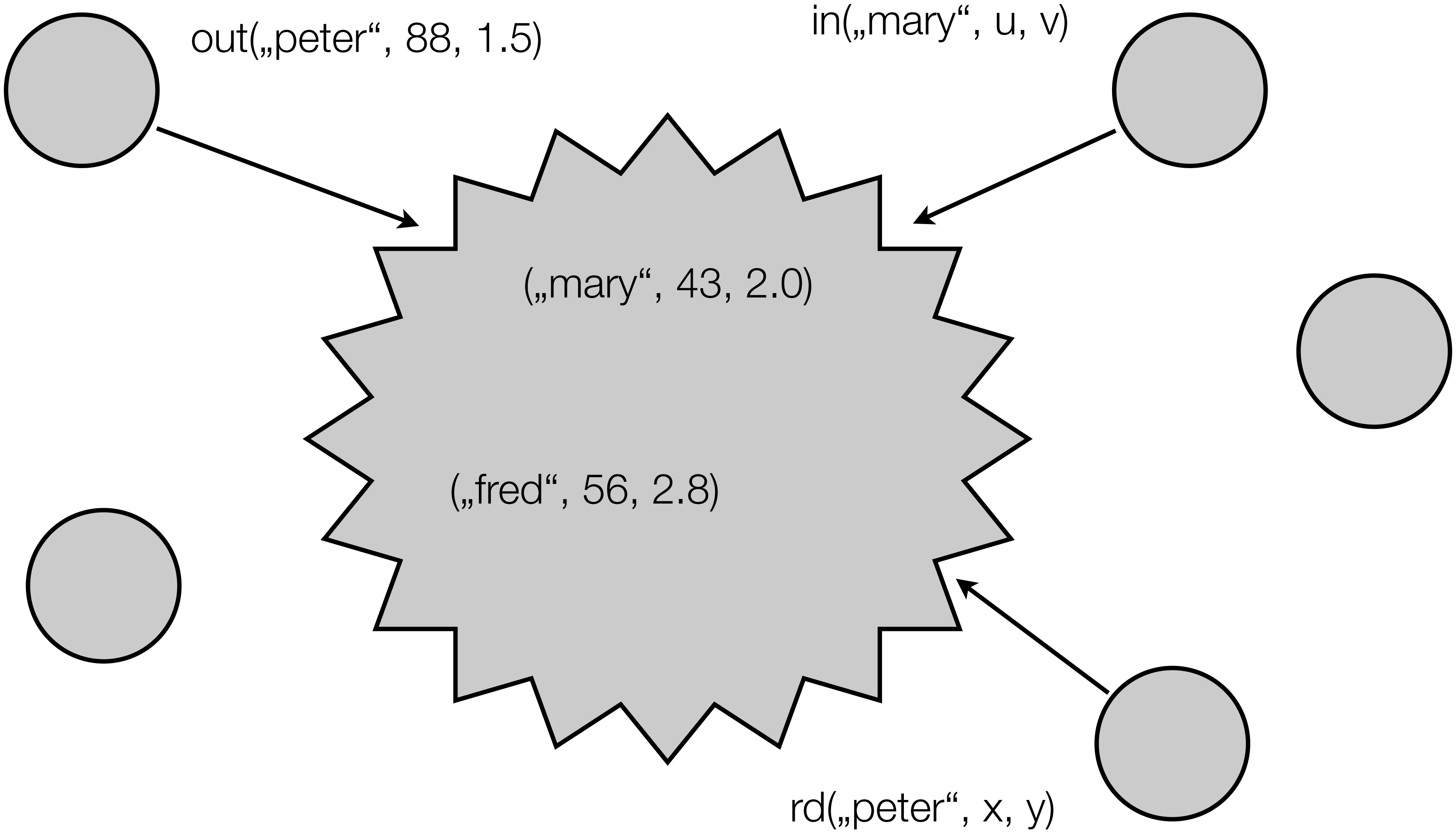
Apple Grand Central Dispatch

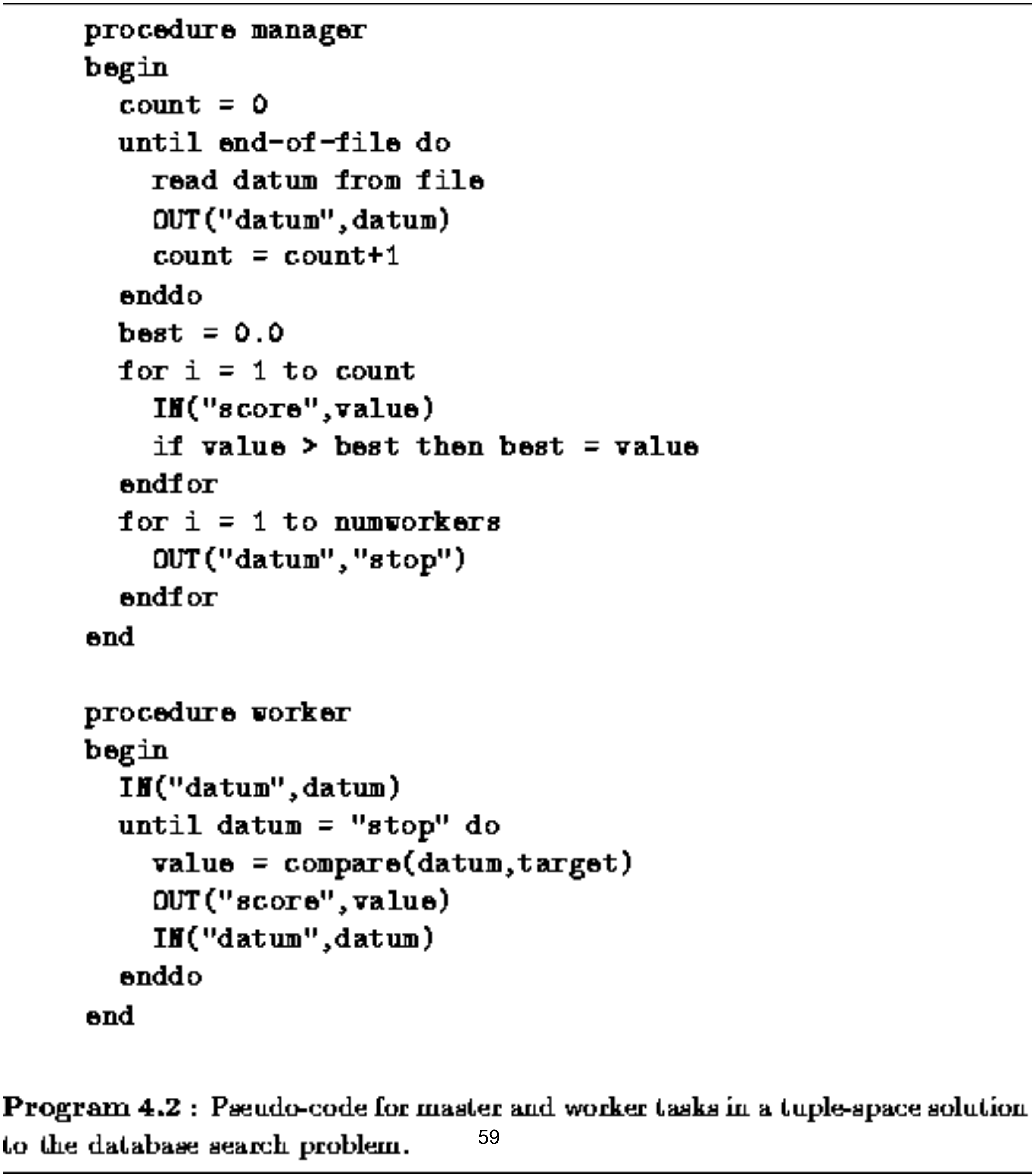
- Part of MacOS X operating system since 10.6
- Task parallelism concept for developer, execution in thread pools
    - Tasks can be functions or **blocks** (C / C++ / ObjectiveC extension)
    - Submitted to dispatch queues, executed in thread pool under control of the Mac OS X operating system
        - Main queue: Tasks execute serially on application‘s main thread
        - Concurrent queue: Tasks start executing in FIFO order, but might run concurrently
        - Serial queue: Tasks execute serially in FIFO order
- Dispatch groups for aggregate synchronization
- On events, dispatch sources can submit tasks to dispatch queues automatically

Linda Model

- Concurrent programming model, developed in Yale University research project
- Tuple-space concept
  - Abstraction of distributed shared memory
  - Set of programming language extensions for facilitating parallel programming
  - Tuple: Fixed fixed-length list containing elements of different type
  - Associative memory - tuples are accessed not by their address but rather by their content and type
  - Destructive (*in*) and nondestructive (*rd*) reads
  - Sequential programs embed insert/retrieve tuple operations
- Multiple implementations (LindaSpaces, GigaSpaces, IBM TSpaces, …)

Tuple Spaces





```
procedure manager
begin
  count = 0
  until end-of-file do
    read datum from file
    OUT("datum",datum)
    count = count+1
  enddo
  best = 0.0
  for i = 1 to count
    IN("score",value)
    if value > best then best = value
  endfor
  for i = 1 to numworkers
    OUT("datum","stop")
  endfor
end

procedure worker
begin
  IN("datum",datum)
  until datum = "stop" do
    value = compare(datum,target)
    OUT("score",value)
    IN("datum",datum)
  enddo
end
```

Program 4.2 : Pseudo-code for master and worker tasks in a tuple-space solution to the database search problem.