Shared-Memory Programming Models

Programmierung Paralleler und Verteilter Systeme (PPV)

Sommer 2015

Frank Feinbube, M.Sc., Felix Eberhardt, M.Sc.,
Prof. Dr. Andreas Polze

Shared-Memory Parallelism

- Process model
  - All memory is local, unless explicitely specified
  - Traditional UNIX approach
- Light-weight process / thread model
  - All memory is global for all execution threads
    - Logical model, remember NUMA !
  - Stack is local
  - Thread scheduling by operating system, manual synchronization
  - POSIX Threads API as industry standard for portability
- Task model
  - Directive / library based concept of tasks
  - Dynamic mapping of tasks to threads from a pool

Threads in classical operating systems

HPI Hasso Plattner Institut

- Windows Threads
- Unix processes / threads / tasks
- Windows fibers

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

POSIX Threads (Pthreads)

HPI Hasso Plattner Institut

- 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 Threads

- *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);
```

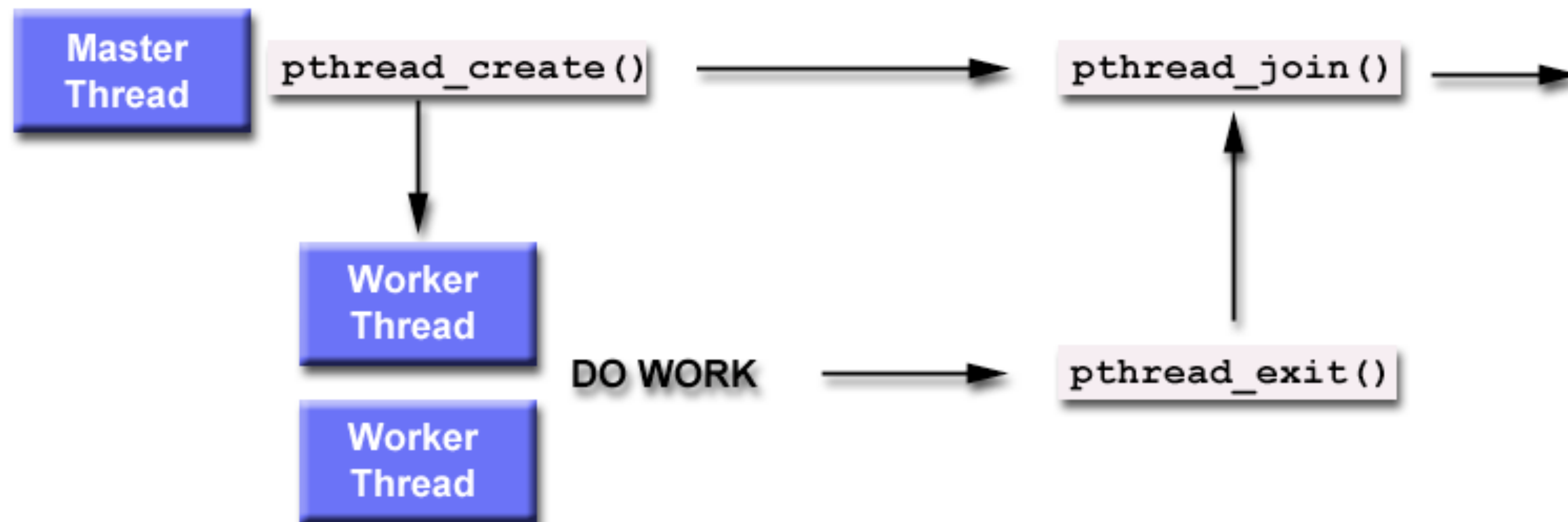
```
/******************************************************************************
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* AUTHOR: Blaise Barney
* LAST REVISED: 08/09/11
******************************************************************************/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

- *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 Threads



```
#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

- *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 decides about wake-up preference
  - Focus on speed of operation, no deadlock or starvation protection mechanism
- Support for normal, recursive, and error-check mutex that reports double locking

```
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 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
- *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
  - 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

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

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

Hasso Plattner Institut

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

Further PThreads Functionality

- *pthread_setconcurrency()*
  - Only meaningful for **m:n** threading environments
- *pthread_setaffinity_np()*
  - Modify processor affinity mask of a thread
  - Forked children inherit this mask
  - Useful for pinning threads explicitely
    - Better load balancing, avoid cache pollution
- *pthread_sigmask()*
  - Individual threads can mask out signals for explicit responsibilites
- *pthread_barrier_wait()*
  - Barrier implementation, optional part of POSIX standard (check for *_POSIX_BARRIERS* macro)

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 Monitors

- 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

HPI Hasso Plattner Institut

- 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

HPI Hasso Plattner Institut


```
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
  }
}
```

Java 6 / 7

- **Lock elision**
  - If the references to a lock have only some ‚local scope‛, it is silently ommited by the JIT compiler
  - Example: Appending strings to a *StringBuffer*
- **Biased locking**
  - Locking consists of lease acquision and lock allocation
  - Looping over a synchronized block optimized by not requiring the thread to release the lease every time
- **Lock coarsening / merging**
  - Combine subsequent synchronized blocks or synchronized method calls
- Java spin locks suspend the thread after a while
  - **Adaptive spin locks** are based on previous attempts on the same lock in the same thread

Hasso Plattner Institut

.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++11

- C++11 specification added support concurrency constructs
- Allows asynchronous tasks with *std::async* or *std::thread*
- Relies on *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(); }
```

```
#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(); }
```

C++11

- Launch policy can be specified for the *async* call
  - 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
- 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';
}
```

C++11

- 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();
}
```

Hasso Plattner Institut

- 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();
}
```

C++11

- Lock-free atomic types that are free 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+=, ...*

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** !!!
- 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)

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

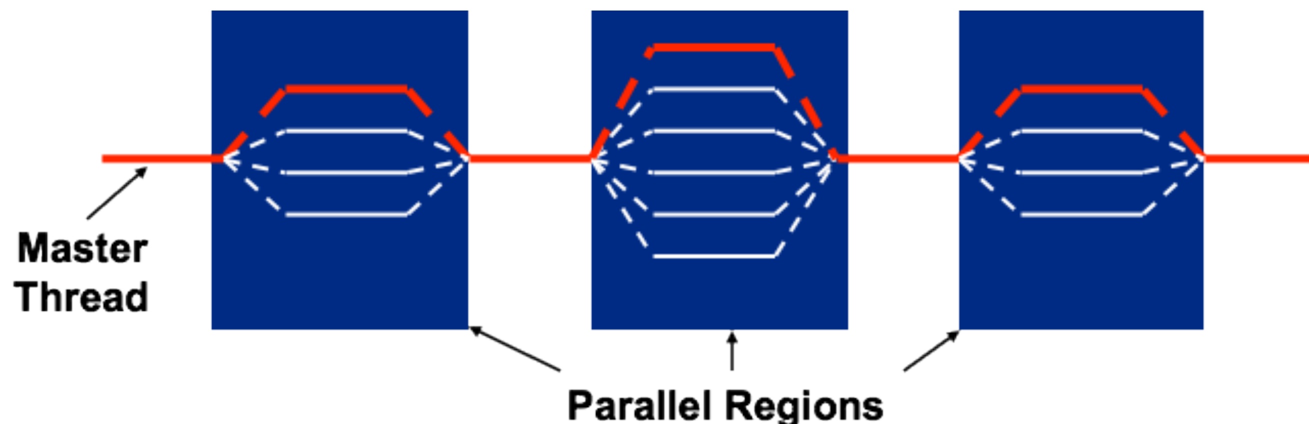
- **Process:** Address space, resource handles, code, set of threads
- **Thread:** Control flow
  - Preemptive scheduling by the operating system
  - Can migrate between cores
- **Task:** Control flow
  - Modeled as object, statement, lambda expression, or anonymous function
  - Cooperative scheduling, 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**

Multi-Tasking

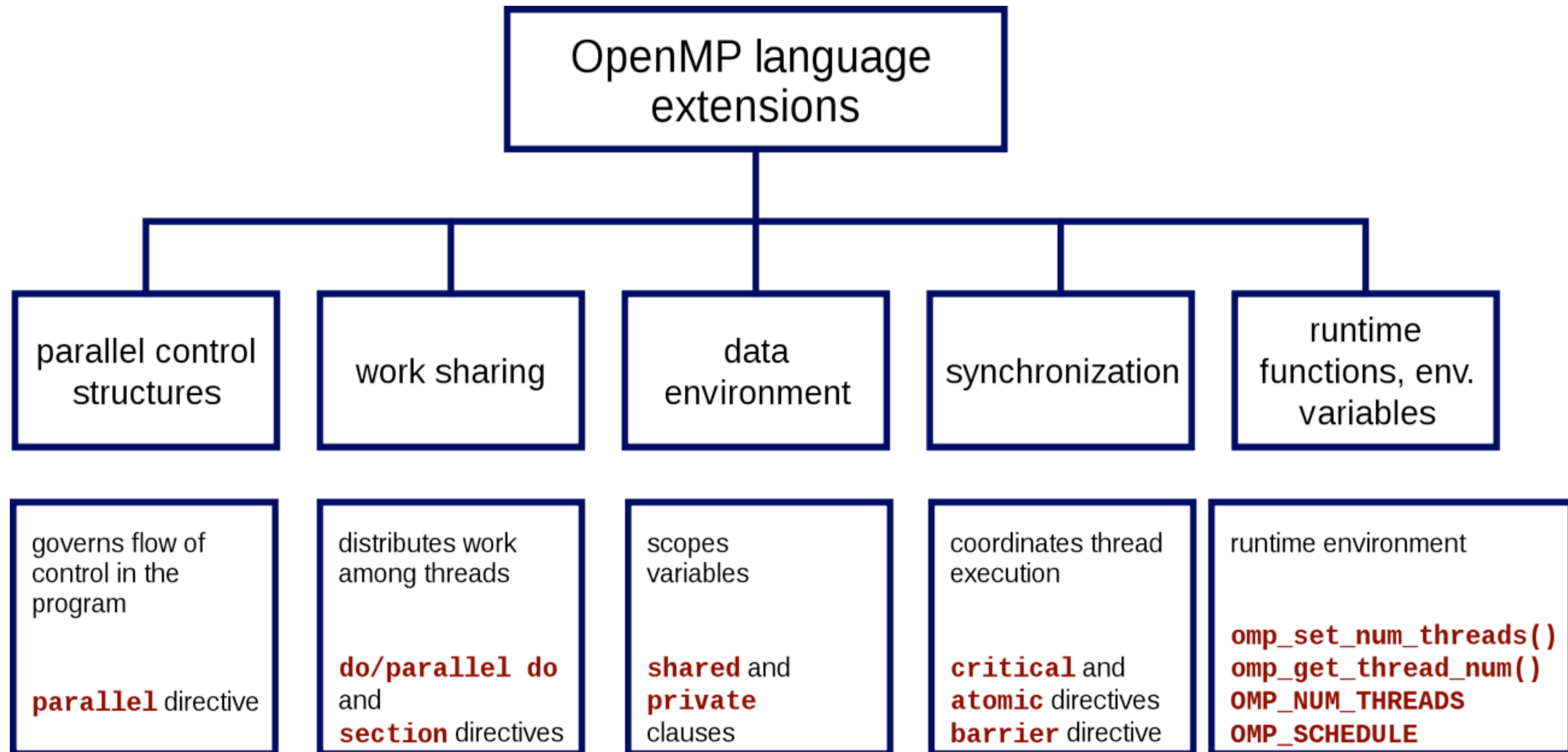
- Relevant issues: Task generation, synchronization, data access
  - Explicit activity as part of some sequential code (operating system thread API, Java / .NET threads, ...) -> *„explicit“ threading*
  - Implicit activity based on a framework (OpenMP, OpenCL, Intel TBB, MS TPL, ...) -> *„implicit“ threading*
- Concurrency problems remain the same
  - Critical section problem with shared variables in different tasks
  - Low-level synchronization primitives typically wrapped by „concurrent data structures“ in the task framework

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
- Programming model: Fork-Join-Parallelism
  - Master thread spawns group of threads for limited code region



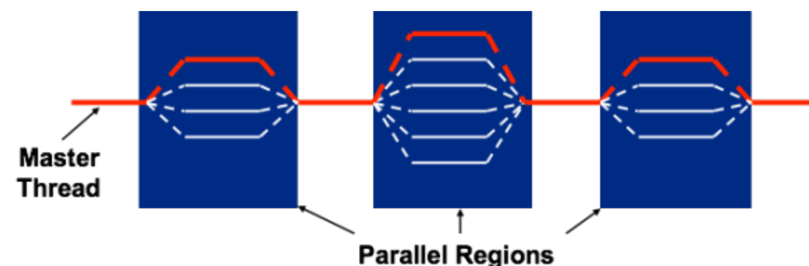
OpenMP



(from Wikipedia)

OpenMP

- 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 Region

Hasso Plattner Institut

- Encountering **thread** for the parallel region generates a set of implicit **tasks,** each with possibly different instructions
- Each resulting implicit **task** is assigned to a different **thread**
- 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

> 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 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
  - ...

```
#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 omp omp.c
```

OpenMP Work Sharing

Hasso Plattner Institut

- 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 define a task
- Task scheduling is handled by the OpenMP implementation
- Clause combinations possible: `#pragma omp parallel for`

Hasso Plattner Institut

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 Data Sharing

Hasso Plattner Institut

- **Shared variable**: Name provides access to 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, 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, automatic variables
  - `firstprivate:` Initialization with last value before region
  - `lastprivate:` Result value after region from last loop iteration or lexically last `section` directive
  -

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
  - Compiler needs information when consistent view is demanded
  - Implicit flush on different occasions, such as barrier region
  - In all other cases, read 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
- 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 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 (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
  - Starts with large block, until minimum chunk size is reached
  - Good for computations with increasing iteration length
    (e.g. prime sieve test)

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;
}
```

HPI Hasso Plattner Institut

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

- 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 Tasks

Hasso Plattner Institut

- 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


```
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);
}
```

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

#pragma omp taskwait
```

[Duran, BSC]

- 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 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 4

- SIMD extensions
  - Portable primitives to describe SIMD parallelization
  - Loop vectorization with *simd* construct
  - Several arguments for guiding the compiler (e.g. alignment)
- 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 support
- User-defined reduction operations

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

Cilk

HPI Hasso Plattner Institut

- C language combined with several new keywords
  - Different approach to OpenMP pragmas
  - Developed at MIT since 1994 (!)
  - Initial commercial version Cilk++ with C / C++ support
- Since 2010, offered by Intel as **Cilk Plus**
  - Official language specification to foster other implementations
  - Meanwhile maintained as GCC branch (similar to OpenMP)
  - Support for Windows, Linux, and MacOS X
- Basic concept of **serialization**
  - Any Cilk program compiled as concurrent code has the same execution semantics as the serial version

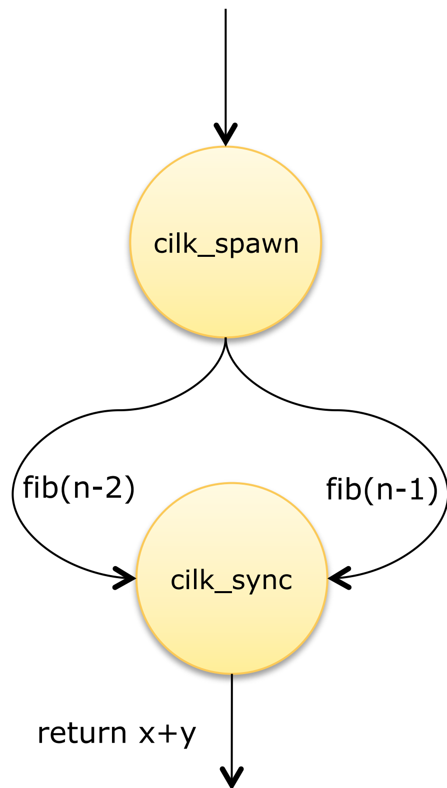
Intel Cilk Plus

- Three keywords to express potential parallelism
  - **cilk_spawn**: Asynchronous function call
    - Runtime decides, spawning is not mandated
  - **cilk_for**: Allows loop iterations to be performed in parallel
    - Runtime decides, parallelization is not mandated
  - **cilk_sync**: Wait until all spawned calls are completed
    - Barrier for *cilk_spawn* activity
- Runtime decided the level of parallelism, performs work stealing
- **Strand:** Instruction sequence in-between a change of parallelism
- **Reducers**: Lock-free private 'views' on variables
- Notation for **SIMD array operations** and **SIMD functions**
- **Serialization**: Cilk keyword become ordinary statements, code semantics are expected to remain the same

Intel Cilk Plus

- *Strand* concept makes it possible to express every program as directed acyclic graph (DAG)

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
}
```




```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```


[cilkplus.org]

Intel Cilk Plus


```
for (int i = 0; i < 8; ++i)
{
    do_work(i);
}
```

```
for (int i = 0; i < 8; ++i)
{
    cilk_spawn do_work(i);
}
cilk_sync;
```

```
cilk_for (int i = 0; i < 8; ++i)
{
    do_work(i);
}
```

[cilkplus.org]

Intel Cilk Plus

- *Accumulator / reduction algorithm*
  - Compute one result value by updating it with every computational step (that may be parallelized)
  - Same reduction concept as with OpenMP and others
  - Problem of avoiding *data races*

```
1 #include <iostream>
2 int main()
3 {
4     unsigned long accum = 0;
5     for (int i = 0; i != 1000; i++) {
6         accum += i*i;
7     }
8     std::cout << accum << "\n";
9 }
```

```
01 #include <iostream>
02 #include <cilk/cilk.h>
03
04 int main()
05 {
06     unsigned long accum = 0;
07     cilk_for (int i = 0; i != 1000; i++) {
08         accum += i*i;
09     }
10     std::cout << accum << "\n";
11 }
```

[software.intel.com]

Intel Cilk Plus

- Express accumulated result as **reducer** pointer variable to get automated locking
- Parallel reducer operations are promised to be in **serial ordering**

```
01 #include <iostream>
02 #include <cilk/cilk.h>
03
04 int main()
05 {
06     unsigned long accum = 0;
07     cilk_for (int i = 0; i != 1000; i++) {
08         accum += i*i;
09     }
10     std::cout << accum << "\n";
11 }
```

```
01 #include <iostream>
02 #include <cilk/cilk.h>
03 #include <cilk/reducer_opadd.h>
04
05 int main()
06 {
07     cilk::reducer_opadd<unsigned long> accum(0);
08     cilk_for (int i = 0; i != 1000; i++) {
09         *accum += i*i;
10     }
11     std::cout << accum.get_value() << "\n";
12 }
```

[software.intel.com]

Intel Cilk Plus

- Express accumulated result as **reducer** pointer variable to get automated locking
- Parallel reducer operations are promised to be in **serial ordering**

```
01 #include <iostream>
02 #include <cilk/cilk.h>
03
04 int main()
05 {
06     unsigned long accum = 0;
07     cilk_for (int i = 0; i != 1000; i++) {
08         accum += i*i;
09     }
10     std::cout << accum << "\n";
11 }
```

```
01 #include <iostream>
02 #include <cilk/cilk.h>
03 #include <cilk/reducer_opadd.h>
04
05 int main()
06 {
07     cilk::reducer_opadd<unsigned long> accum(0);
08     cilk_for (int i = 0; i != 1000; i++) {
09         *accum += i*i;
10     }
11     std::cout << accum.get_value() << "\n";
12 }
```

[software.intel.com]

Intel Cilk Plus


```
01 #include<cilk/cilk.h>
02 #include <cilk/reducer_list.h>
03
04 // The tree node structure.
05 //
06 template <typename Key, typename Value>
07 struct TreeNode {
08     TreeNode* left_subtree;
09     TreeNode* right_subtreee;
10     Key key;
11     Value value;
12 };
```

- Parallel tree search
- Resulting list is always 'in-order'
  - Left subtree
  - Root
  - Right subtree
- Stable semantics regardless of parallelization

```
14 // The worker function. Walk a subtree and add the values
15 // of all nodes that match a key to a list reducer.
16 //
17 template <typename Key, typename Value>
18 void filter_and_collect(const TreeNode<Key, Value>* subtree,
19                         const Key& key,
20                         cilk::reducer_list_append<Value>& list)
21 {
22     if (!subtree) return;
23     cilk_spawn filter_and_collect(subtree->left, key, list);
24     if (subtree->key == key) {
25         list->push_back(subtree->value);
26     }
27     filter_and_collect(subtree->right, key, list);
28 }
```

```
30 // The main function. Compute and return a list of the 'value'
31 // fields of all the nodes in a tree whose 'key' fields match a
32 // specified 'key'.
33 //
34 template <typename Key, typename Value>
35 std::list<Value> filter_tree(const TreeNode<Key, Value>* tree,
36                              const Key& key)
37 {
38     cilk::reducer_list_append<Value> list;
39     filter_and_collect(tree, key, list);
40     return list.get_value();
41 }
```

[software.intel.com]

Intel Cilk Plus

- Predefined reducers for C and C++, custom reducers supported
- Optimized internal operation based on *strands* concept
  - Each *strand* gets a private view on the reducer variable
    - No locking during update
  - When *strands* join again, the reducer merges the operations

Reducer	Description
Lists	
reducer_list_append	Creates a list by adding elements to the back.
reducer_list_prepend	Creates a list by adding elements to the front.
Min/Max	
reducer_max	Calculates the maximum value of a set of values.
reducer_max_index	Calculates the maximum value and index of that value of a set of values.
reducer_min	Calculates the minimum value of a set of values.
reducer_min_index	Calculates the minimum value and index of that value of a set of values.
Math Operators	
reducer_opadd	Calculates the sum of a set of values.
Bitwise Operators	
reducer_opand	Calculates the binary AND of a set of values.
reducer_opor	Calculate the binary OR of a set of values.
reducer_opxor	Calculate the binary XOR of a set of values.
String Operators	
reducer_string	Accumulates a string using append operations.
reducer_wstring	Accumulates a "wide" string using append operations.
Files	
reducer_ostream	An output stream that can be written in parallel.

Intel Cilk Plus

- Cilk support the high-level expression of array operations
  - Gives the runtime a chance to parallelize work
  - Intended for data parallel element operations without any ordering constraints
- New operator **[:]**
  - Specify data parallelism on an array
  - *array-expression[lower-bound : length : stride]*
  - Multi-dimensional sections are supported: *a[:][:]*

- Short-hand description for complex loops
  - **A[:]=5**
    for (i = 0; i < 10; i++)
    A[i] = 5;
  - **A[0:n] = 5;**
  - **A[0:5:2] = 5;**
    for (i = 0; i < 10; i += 2)
    A[i] = 5;
  - **A[:] = B[:];**
  - **A[:] = B[:] + 5;**
  - **D[:] = A[:] + B[:];**
  - **func (A[:]);**

Intel Cilk Plus

- Array notation can be used inside conditions
  if (5 == a[:])
  results[:] = "Matched";
  else
  results[:] = "Not Matched";
- Function mapping is executed in parallel with no specific order
  A[:] = pow(B[:], c);
- In C++, this works with any defined operator
  A[:] = B[:] + C[:]; // A[:] = operator+(B[:], C[:]);
- Several predefined reduction macros applicable to array sections
  - *__sec_reduce_add, __sec_reduce_mul, __sec_reduce_max, __sec_reduce_min, __sec_reduce_all_zero, __sec_reduce_any_zero*
- Array sections can be used as array indices for gather / scatter
  - *C[:] = A[B[:]]* (gather), *A[B[:]] = C[:]* (scatter)

Intel Threading Building Blocks (TBB)

- Portable C++ library, toolkits for different operating systems
- Also available as open source version
- Complements basic OpenMP / Cilk features
  - 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

HPI Hasso Plattner Institut

Easy Mappings [Dig]

Hasso Plattner Institut

	Oracle Java	Intel TBB	MS .Net 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