C/C++ Programming Abstractions for Parallelism & Concurrency

Torvald Riegel
Senior Software Engineer
Red Hat
06.26.12
Parallelism and Concurrency

- Parallelism vs. concurrency = vs.

- Programming abstractions:
  - P: Express and allow parallel execution
  - C: Handle concurrency: sharing, interleavings, conflicts, ...

- P&C are related
  - Parallel programs have at least some concurrent parts
  - When handling concurrency (synchronization), try to execute in parallel as much as possible

- Amdahl’s Law:
  \[
  speedup = \frac{1}{(1 - P) + \frac{P}{N}}
  \]
  \(P\) ... Parallel part
  \((1 - P)\) ... Sequential part
  \(N\) ... #CPUs
Multi-threading support in C++11

- Parallelism
  - Threads
  - Futures

- Concurrency
  - Memory model
  - Atomic data types
  - Locks
  - Condition variables
This talk

- Parallelism
  - Threads
  - Futures

- Advanced abstractions for task-based parallelism
  - Example: Intel Threading Building Blocks (TBB)

- Concurrency
  - Memory model
  - Atomic data types
  - Locks
  - Condition variables

- Transactional Memory
Task-based parallelism

- Split program into tasks (units of work)
  - Tasks can depend on other tasks
  - Creates a directed acyclic graph (DAG)
  - Tasks not ordered in DAG are parallel
    - But can still synchronize w/ each other
- Can express data parallelism, message passing, ...

- Critical path
  - Longest path in task DAG
  - Limits speedup (Amdahl’s Law)
    - Dependencies force sequential execution of tasks on a path
How to express tasks?

- Explicit threading:
  - Create one thread for each task, join at task dependencies
    ```cpp
    try {
      auto task = std::thread(work);
      task.join();
    } catch (std::system_error e) { error_fallback(); }
    ```

- Task abstractions:
  - Create tasks that may execute in another or a new thread
    ```cpp
    auto task = std::async(work);
    task.get();
    ```
  - Task scheduler provided by the abstraction (e.g., TBB) manages threads and executes task
Explicit threading problems

- How many threads?
  - Too few: lack of parallelism
  - Too many: less locality, larger memory-system footprint
  - Called from an already parallel context?

- Even with a custom thread pool:
  - Execute which tasks on which thread?
  - How to coordinate with other thread pools?

- Further (future) challenges:
  - Resource usage is system-wide optimization problem
  - NUMA, hardware heterogeneity, ...
Scheduling: Paying attention to the critical path

- Example (#CPUs = 2):
  - Critical path: 3 tasks
    - Min. exec time = 3

- Explicit threading:
  - 1 task = 1 thread:
    Exec time = 1 + 3/2 + 1 = 3.5
  - OS scheduler isn’t aware of critical path
    - Unless program explicitly sets thread priorities

- Task scheduler
  - #Threads = #CPUs, w/ work stealing
  - Critical path heuristics:
    Exec time = 1 + 1 + 1 = 3
  - But heuristics aren’t perfect (critical path needs to be close to master thread (left))
Parallelism abstractions in TBB: Examples

- Parallel iteration, reductions, sorting, ...
- Message passing with tasks (flow graph)

Sum of all elements in an array:

```cpp
int sum = tbb::parallel_reduce(
    tbb::blocked_range<int*>(array, array + size),
    0,
    [](const tbb::blocked_range<int*>& r, int v) {
      for (auto i=r.begin(); i!=r.end(); ++i) v += *i;
      return v;
    },
    [](int x, int y) { return x+y; }
);
```

Input array, automatically partitioned

Identity value

Combines results of partitions

Task that returns the sum of elements in a partition
Task-based parallelism: Summary

- **Explicit threading:**
  - Lots of control, but more difficult to get right
  - Makes modular programming harder

- **Task-based parallelism abstractions:**
  - Library contains most of the implementation
    - Simpler to start programming with
    - Integration with other libraries / services becomes easier
    - Adjusting to new HW becomes easier (library update)
  - Modular programming model is possible
  - Less control (e.g., task/thread placement) or pay for control with similar problems as with explicit threading
## Concurrency and atomicity

<table>
<thead>
<tr>
<th>C++11 atomic types</th>
<th>Transactional Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide atomicity for concurrent accesses by different threads</td>
<td>Both based on C++11 memory model</td>
</tr>
<tr>
<td>Single memory location</td>
<td>Any number of memory locations</td>
</tr>
<tr>
<td>Low-level abstraction, exposes HW primitives</td>
<td>High-level abstraction, mixed SW/HW runtime support</td>
</tr>
</tbody>
</table>

- C11 has (very) similar support
Atomic types and accesses

- Making a type T atomic: atomic<T>
- Load, store:
  - atomic<int> a; a = a + 1; a.store(a.load() + 1);
- CAS and other atomic read-modify-write:
  - int exp = 0; a.compare_exchange_strong(exp, 1);
  - int previous = a.fetch_add(23);
- Sequential consistency is default
  - All s-c ops in total order that is consistent with per-thread program orders
- Other weaker memory orders can be specified
  - locked_flag.store(false, memory_order_release);
  - Important orders: acquire, acq_rel, release, relaxed, seq_cst
Why a memory model?

- Defines multi-threaded executions (undefined pre C++11)
  - Normal, nonatomic memory accesses
  - Ordering of all operations enforced by atomic/synchronizing memory accesses

- Common ground for programmers and compilers
  - Formalizations of the model exist

- Unified abstraction for HW memory models
  - Portable concurrent code (across HW and compilers)
  - Simpler than several HW memory models
Happens-before (HB)

- Order of operations in a particular execution of a program
- Derived from / related to other relations:
  - Sequenced-before (SB): single-thread program order
  - Reads-from: which store op’s value a load op reads
  - Synchronizes with (SW)
    - Example: acquire-load reads from release-store (both atomic)
  - Total orders for seq_cst operations, lock acquisition/release
  - Simplified: HB = transitive closure of SB U SW

- Compiler generates code that ensures some valid HB:
  - Must be acyclic and consistent with all other relations/rules
  - Generated code ensures HB on top of HW memory model
- Simpler: Use just SC by not specifying memory orders
Data-race freedom (DRF)

- Data race: Nonatomic accesses, same location, at least one a store, not ordered by HB
- Any valid execution has a data race? => Undefined behavior

- Programs must be DRF
  - Allows compiler to optimize
- Compiler preserves DRF
  - Access granularity
  - Speculative stores, reordering, hoisting, ...
Examples

- Simple statistics counter:
  
  ```java
  counter.fetch_add(1, memory_order_relaxed);
  counter.store(counter.load(memory_order_relaxed) + 1, memory_order_relaxed);
  ```

- Publication:
  
  ```java
  init(data);
  data_public.store(true, memory_order_release);
  if (data_public.load(memory_order_acquire))
    use(data);
  ```

- Beware of data races:
  
  ```java
  temp = data;
  if (data_public.load(memory_order_acquire))
    use(temp);
  ```

Program behavior is undefined

Races with `init`
Transactional Memory (TM): What is it?

- TM is a programming abstraction
  - Declare that several actions are atomic
  - But don’t have to implement how this is achieved

- TM implementations
  - Are generic, not application-specific
  - Several implementation possibilities:
    - STM: Pure SW TM algorithms
      - Blocking or nonblocking, fine- or coarse-granular locking, ...
    - HTM/HyTM: using additional HW support for TM
      - E.g., Intel TSX, AMD ASF
Transactional language constructs for C/C++

- Declare that compound statements, expressions, or functions must execute atomically
  - `__transaction_atomic { if (x < 10) y++; }`
- No data annotations or special data types required
- Existing (sequential) code can be used in transactions
- Language-level txns are a portable interface for HTM/STM
  - HTM support can be delivered by a TM runtime library update

- Draft specification for C++
  - HP, IBM, Intel, Oracle, Red Hat
  - C++ standard study group on TM (SG5)
  - C will be similar (GCC supports txns in C and C++)
How to synchronize with transactions?

- TM extends the C++11 memory model
  - All transactions totally ordered
  - Order contributes to Happens-Before (HB)
  - TM implementation ensures some valid order that is consistent with HB
  - Does not imply sequential execution!

- Data-race freedom still required

  init(data); __transaction_atomic { data_public = true; }

  Correct: __transaction_atomic {
            if (data_public) use(data); }

  Incorrect: __transaction_atomic { temp = data; // Data race
            if (data_public) use(temp); }
## Atomic vs. relaxed transactions

<table>
<thead>
<tr>
<th>Atomic</th>
<th>Relaxed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic wrt.:</td>
<td>All other code</td>
</tr>
<tr>
<td>Restrictions on txnal code:</td>
<td>No other synchronization (conservative, WIP)</td>
</tr>
<tr>
<td>Keyword:</td>
<td>__transaction_atomic</td>
</tr>
</tbody>
</table>

- Restrictions/safety of atomic checked at compile time
  - Compiler analyzes code
  - Additional function attributes to deal with multiple CUs
  - WIP: dynamic checking at runtime instead of static (optional)
TM supports a modular programming model

- Programmers don’t need to manage association between shared data and synchronization metadata (e.g., locks)
  - TM implementation takes care of that
- Functions containing only txnal sync compose w/o deadlock, nesting order does not matter
- Early university user studies suggest that txns lead to simpler programs with fewer errors compared to locking
- Example:
  ```c
  void move(list& l1, list& l2, element e)
  { if (l1.remove(e)) l2.insert(e); }
  • TM: __transaction_atomic { move(A, B, 23); }
  • Locks: ?
  ```
Summary & other programming abstractions

- **Parallelism**
  - Task-based parallelism

- **Concurrency**
  - Memory model, atomics
  - Transactional Memory

- **Others:**
  - Accelerators/GPGPU: OpenCL, OpenACC, ...
  - OpenMP
  - Data/vector parallelism (e.g., Cilk+ array notations)

- **Others:**
  - Concurrent data structures
    - Containers: hash maps, queues, ...
Conclusion: Choice and integration are key

- No one programming abstraction that rules them all
- Our approach:
  - Providing the useful choices
    - Supporting proven and promising new abstractions
  - Integration
    - Interoperability between abstractions
    - Standardization (e.g., C++ standard study groups 1+5)

- If you haven’t yet, please
  - Get familiar with P&C abstractions
  - Think about how they would fit into your applications
  - Give us feedback