Parallel and Concurrent Programming Algorithms And Higher Level Concepts

Marwan Burelle

marwan.burelle@lse.epita.fr
http://wiki-prog.infoprepa.epita.fr

Outline

Data Structures

Overview

1. Data Structures
   - Global Consideration
   - Finer Locking ?
   - Non-Blocking

Sharing Data

- First, always try to apply the following mantra:

Don't share data !

- When non-scalar data are shared among several threads, you must take care of the concurrent access.
- The first concern is structural coherency, but there's a lot of other issues to be observed.
- As usual there's questions about the observed relative order of operations: when and how does threads view updates from each others ?
- One must also consider the resistance of the structures to heavy contention and its ability to scale.

Is Locking Enough ?

- Locking helps ensure data structural coherency and eventually forces a sequencial ordering.
- Locks won't protect you against deadlocks, contention and priority inversion.
- Depending on usage, one must also provides more complex synchronization (producers/consumers model, reader/writers model . . .)
- Locks scale badly against high number of threads and heterogenous loads (lots of running applications.)

Structures Kinds And Locking

- Some data structures are more suited than some others.
- In the worst case, a data structure always requires a global lock, while some can be used with a finer locking schema.
- Arrays and vectors (array based lists) belongs to the worst cases: any translate or swap operations (often needed in vectors or heap) requires global locking. Concurrent modifications of adjacent cells often induce false sharing (unneeded cache synchronizations.)
- On the other hand, linked data structures offer better possibilities of finer locking techniques.

The Lesser Is Better

- Modern data structures tries to avoid locks.
- Non-blocking structures completely avoid locking.
- Some plateforms provide *copy-on-write* mechanism.
- Some even try (the fools) to use functional approach to avoid mutable shared states.
- At least, you can try fine grain locking or optimistic data structures, most of the time they have good performance and scale better than usual locked structures.

Overview

Globally locked list

globally locked list

```
template<typename T>
struct locked_list {
  typedef std::lock_guard<std::mutex> synchronized;
  struct cell {
    T            value;
    cell        *next;
    cell(cell *n, T v) : value(v), next(n) {}
  };
  void add(T x) { synchronized _lock(lock);
    head = new cell(head, x);
  }
  unsigned size() { synchronized _lock(lock);
    unsigned                   l = 0;
    for (auto cur = head; cur != 0; cur = cur->next) ++l;
    return l;
  }
  std::mutex             lock;
  cell                  *head;
};
```

Globally Locked List

insert method

```
void locked_list::insert(T x, unsigned pos) {
  synchronized         _lock(lock);
  cell                *cur = head, *pred = 0;
  for (unsigned i = 0; i < pos && cur != 0; ++i) {
    pred = cur;
    cur = cur->next;
  }
  cur = new cell(cur, x);
  if (pred == 0) head = cur;
  else pred->next = cur;
}
```

Finer locking ?

- In our previous example, we lock the whole list for each operations done.
- But that's not needed ! We can only lock the cell we need.
- We add a lock on each cell and then:
    - For read traversal, we only need to lock the currently readed cell (and only if we can remove cells.)
    - For `add` we need a lock on the head pointers for other modifiers but not for readers.
    - For `insert` we need to lock the previous cell for other modifiers.
- Since traversal requires locking, threads traverse the list in FIFO order.

Issues And Enhancement

- The main issue is to correctly acquire and release locks.
- For a read-traversal, we need to be sure that once we get the pointer to the cell this one will remain valid.
- When inserting and removing you need to hold the lock on the cell before the position of the insertion (or before the cell to be removed.)
- Holding a lock on cell must enforce two main properties: the content of the cell won't be update by another thread and the cell will remain valid.
- We can try to enhance this locking schema: if no concurrent removing occurs, we can avoid locking on the read-traversal.

Example With Fine Grained Lock

Using one lock per cell

```
void insert(T x, unsigned pos) {
  head->mutex.lock();
  if (head->next) {
    cell              *pred = head, *cur;
    cur = pred->next;
    cur->mutex.lock();
    for (unsigned i=0; i < pos && cur; ++i) {
      pred->mutex.unlock();
      pred = cur;
      cur = cur->next;
      cur->mutex.lock();
    }
    pred->next = new cell(cur, x);
    cur->mutex.unlock();
    pred->mutex.unlock();
  } else {
    head->next = new cell(head->next, x);
    head->next.unlock();
  }
}
```

Optimistic Approache

- In the version with one lock per cell we spent a lot of time on locking even if no concurrent event hapens durint our operation.
- Optimistic approaches consider that trying to perform operation without locking, and if needed retry using locks.
- Even if the retry operation is more expensive than the one-lock-per-cell corresponding version, most of the time the unlocked try will succeed directly and the culmulated gain out weights the time spent retrying.

Example Of Optimistic List

insert in an optimistic list

```
bool validate(cell *pred, cell * cur) {
  for (auto c = head; c; c = c->next) {
    if (c == pred)
      return pred->next == cur;
  }
  return false;
}

void insert(T x, unsigned pos) {
  do {
    auto      pred = head;
    auto      cur  = pred->next;
    for (unsigned i=0; i < pos && cur; ++i, cur = cur->next)
      pred = cur;
    pred->mutex.lock();
    bool      need_unlock = false;
    if (cur) {
      need_unlock = true;
      cur->mutex.lock();
    }
    if (validate(pred,cur)) {
      pred->next = new cell(cur,x);
      pred->mutex.unlock();
      if (need_unlock) cur->mutex.unlock();
      return;
    }
    pred->mutex.unlock();
    if (need_unlock) cur->mutex.unlock();
  } while (true);
}
```

Overview

1. Data Structures
   - Global Consideration
   - Finer Locking ?
   - Non-Blocking

Living Without Any Lock ?

- Can we do better ? Can we completely avoid locking ?
- This the goal of non-blocking operations.
- Motivation: when a threading holding a lock get schedule, it blocks all other concurrent operations without *using* the locked data.
- Can we maintain progression in the system ?

Lock Free And Wait Free

- **lock-free:** in a given set of processes, there's always at least one process capable of progression. It's a global progression property, the whole set of processes is capable of progression.
- **wait-free:** in a given set of processes, each process can perform its action in a finite (bounded) number of steps. The progression is then local.
- The lock-free property is an important requirement for contention resistant algorithms. It's also an important property for multi-threaded programs running in a highly multi-programmed environment. When using locks, the owner of the lock may be inactive, due to scheduling constraints, bocking the progression of the whole program, while other threads maybe active and waiting for the release of the lock.

What Do We Need ?

- An atomic way to conditionally update pointers: **compare and swap** (CAS)
- CAS is the only operation permitting lock-free/wait-free algorithm (one can use ll/sc model, but most of implementation are broken.)

Syntax: Compare And Swap

```
bool CAS(int *A, int newval, int cmpval) {
  if (*A == cmpval) {
    *A = newval;
    return true;
  } else {
    return false;
  }
}
```

Example: Lock-Free Queue

- Like optimistic list we use a fail/retry model.
- The algorithm tries (and succeeds) to keep the queue in a coherent state.
- The only intermediary possible state can be completed by any other thread.
- We follow [1] algorithm.

Example: Lock-Free Queue

Lock-Free Queue

```
struct List {
  struct node {
    int                  value;
    std::atomic<node*>   next;
    node() { next = 0; }
    node(int x) : value(x) {}
  };

  std::atomic<node*>    Head, Tail;

  List() {
    Head = new node();
    Tail = Head;
  }
  bool pop(int *lvalue);
  void push(int x);
};
```

Example: Lock-Free Queue

Lock-Free Queue

```
bool List::pop(int *lvalue) {
  node        *head, *tail, *next;
  do {
    // acquire pointers
    head = Head.load(); tail = Tail.load();
    next = head.next.load();
    if (Head != head) continue;  // are we coherent
    if (next == 0) return false; // empty queue
    if (head == tail) {
      // missing completion on tail
      // do the job of an unfinished push
      Tail.compare_exchange_weak(tail, next);
      continue;
    }
    lvalue = next->value;        // copy value
    // done ? try to cut the head off
    if (Head.compare_exchange_weak(head, next)) break;
    // fails ? got to retry
  } while (true);
}
```

Example: Lock-Free Queue

Lock-Free Queue

```
void List::push(int x) {
  node           *node = new node(x);
  node           *head;
  do {
    tail = Tail.load();
    if (Tail != tail) continue;
    if (tail->next != 0) {
      // missing completion on tail
      Tail.compare_exchange_weak(tail,tail->next);
      continue;
    }
    // update the next of the tail
    if (tail.next.compare_exchange_weak(next,node)) break;
  } while (true);
  // finally update the tail (may fail, but we don't care)
  Tail.compare_exchange_weak(tail, node);
}
```

Tasks Systems

Direct Manipulation of Physical Threads

- Physical (*system*) threads are not portable
- Most of the time, physical threads are almost independant process
- Creating, joining and cancelling threads is almost as expensive as process manipulations
- Synchronisation often implies kernel/user context switching
- Scheduling is under system control and doesn't take care of synchronisation and memory issues
- Data segmentation for parallel computing is problem **and** hardware driven:
  - Data must be split in order to respect memory and algorithm constraints
  - Number of physical threads needs to be dependant of the number of processors/cores to maximize performances
- ...

Light/Logical Threads

- One can implement threads in full user-space (*light threads*) but we loose physical parallelism.
- A good choice would be to implement *logical threads* with scheduling exploiting physical threads.
- Using logical threads introduces loose coupling between problem segmentation and hardware segmentation.
- *Local* scheduling increase code complexity and may introduce overhead.

Tasks based approach

- A good model for logical threads is a tasks system.
- A task is a sequential unit in a parallel algorithm.
- Tasks perform (*sequential*) computations and may spawn new tasks.
- The tasks system manage scheduling between *open* tasks and available physical threads.
- Tasks systems often use a *threads pool*: the system start a bunch of physical threads and schedule tasks on available threads dynamically.

Simple tasks system: waiting queue.

- *Producer* schedule new *tasks* by pushing it to the queue.
- *Consumer* take new *tasks* from the queue.
- *Producer* and *Consumer* are physical threads, we call them **worker**.
- Each worker may play both role (or not.)
- Tasks can be input values or data ranges for a fixed task's code.
- It is also possible to implement tasks description so producer can push any kinds of task.
- For most cases, we need to handle a kind of *join*: special task pushed when computation's results are ready, in order to closed unfinished tasks (think of a parallel reduce or parallel Fibonacci numbers computation.)

Tasks Sytems For Real

- Java `Executor` provides a task-based threading approach
- Intel's TBB (Threading Building Blocks) is completely based on this paradigm:
  - High level tools (such as parallel for) are based on a task and the librairy provides a scheduling mechanism to efficiently executes task.
  - You can also directly use the task system and build you're own partitionning.
  - TBB provides also pipeline mechanism

Higher Level Tools

Overview

3. Higher Level Tools
   - OpenMP
   - Tell Me More (Go, OpenCL, . . .)

OpenMP

- OpenMP is an extension to the C/C++ language
- It provides concurrent primitives for parallel loops and other things.
- Support must be included at compiler level.
- It's actually one of the most efficient support.

An Example

A Parallel For

```c
#include <omp.h>
#define CHUNKSIZE        100
#define N                1000

main() {
  int           i, chunk;
  float         a[N], b[N], c[N];

  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
      #pragma omp for schedule(dynamic,chunk) nowait
      for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
    }  /* end of parallel section */
}
```

Overview

3 Higher Level Tools
- OpenMP
- Tell Me More (Go, OpenCL, . . .)

Go

- go is a new language from Google (designed by former Bell lab's Ken Thompson and Rob Pike)
- go is a kind of *modern* C with non-intrusive OO features (without classes)
- go uses a notion of co-routines: one can launch functions in a separate thread.
- *go-routines* are managed and executed upon an independant scheduling scheme using separated physical threads.
- Rather than using shared memory, go prefers communication channel (a kind of typeded pipes inspired from Limbo concept.)

OpenCL/Cuda

- OpenCL (and Cuda) are essentially dedicated to exploiting computing power provided by devices (like GPGPU)
- The model is simple: almost no shared memory, execution organized in *groups of groups* on a grid.
- The code is written in a dedicated language (not far from C) and compiled on the fly by the support library and then uploaded on the device for execution.
- The classical approach is to execute the same kernel (name for functions runs on the device) on each node of the grid.

Bibliography

Michael and Scott.
Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.
In *PODC: 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1996.