IC619: System Programming

Lecture 12

2012.11.26

Min-Soo Kim

Daegu Gyeongbuk Institute of Science & Technology

Chap 8.
Designing concurrent code

Techniques for dividing work between threads

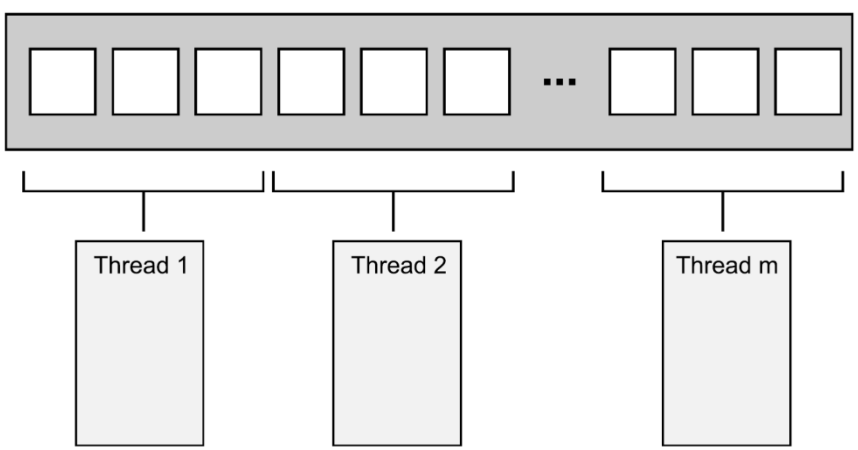
- **Dividing data between threads before processing begins**
  - assumption that the threads are going to be doing essentially the same work on each chunk of data

- **Dividing data recursively**
  - same assumption with above

- **Dividing work by task type**
  - make the threads specialists, where each performs a distinct task

Dividing data between threads before processing begins

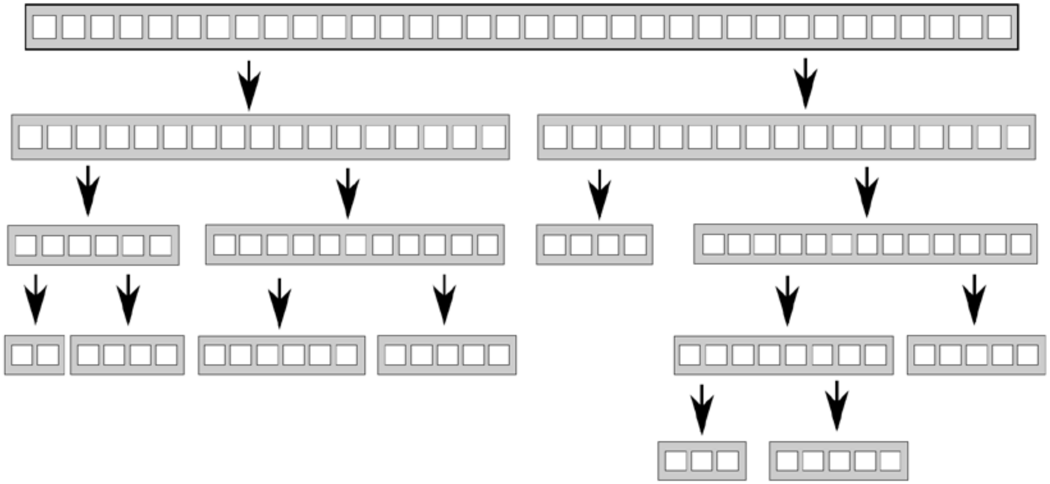
- **Assign each element to one of the processing threads**
  - allocate the first *N* elements to one thread, the next *N* elements to another thread, and so on
- **Used in MPI or OpenMP framework**
  - task is split into a set of parallel tasks
  - worker threads run these tasks independently
  - results are combined in a final *reduction* step



Distributing consecutive chunks of data between threads

Dividing data recursively

- **You can't parallelize QuickSort by simply dividing the data up front**
  - you can know which "half" of the items go in only by processing them
- **There are *more* calls to the quick sort function with each level of recursion**
  - too many threads might actually *slow down* the application
- **You need to keep a tighter rein on the number of threads**
  - rather than starting a new thread for the recursive calls
  - just push the chunk to be sorted onto a thread-safe stack



Parallel Quicksort using a stack of pending chunks to sort

```
template<typename T>
struct sorter
{
    struct chunk_to_sort {
        std::list<T> data;
        std::promise<std::list<T> > promise;
    };
    thread_safe_stack<chunk_to_sort> chunks;
    std::vector<std::thread> threads;
    unsigned const max_thread_count;
    std::atomic<bool> end_of_data;

    sorter(): max_thread_count(std::thread::hardware_concurrency()-1), end_of_data(false) {}
    ~sorter()
    {
        end_of_data=true;
        for(unsigned i=0;i<threads.size();++i)
        {
            threads[i].join();
        }
    }
    void try_sort_chunk()
    {
        boost::shared_ptr<chunk_to_sort > chunk=chunks.pop();
        if(chunk)
        {
            sort_chunk(chunk);
        }
    }
```

```
std::list<T> do_sort(std::list<T>& chunk_data)
{
    if(chunk_data.empty()) {
        return chunk_data;
    }
    std::list<T> result;
    result.splice(result.begin(),chunk_data,chunk_data.begin());
    T const& partition_val=*result.begin();

    typename std::list<T>::iterator divide_point=
        std::partition(chunk_data.begin(),chunk_data.end(),
                       [&](T const& val){return val<partition_val;});
    chunk_to_sort new_lower_chunk;
    new_lower_chunk.data.splice(new_lower_chunk.data.end(), chunk_data,chunk_data.begin(),
                                divide_point);

    std::future<std::list<T> > new_lower=new_lower_chunk.promise.get_future();
    chunks.push(std::move(new_lower_chunk));    // lower chunk might be handled by another thread
    if(threads.size()<max_thread_count)
    {
        threads.push_back(std::thread(&sorter<T>::sort_thread,this));
    }

    std::list<T> new_higher(do_sort(chunk_data));
    result.splice(result.end(),new_higher);
    while(new_lower.wait_for(std::chrono::seconds(0)) != std::future_status::ready)
    {
        try_sort_chunk();                       // try to proces chunks from the stack on this thread
    }
    result.splice(result.begin(),new_lower.get());
    return result;
}
```

```
    void sort_chunk(boost::shared_ptr<chunk_to_sort > const& chunk)
    {
        chunk->promise.set_value(do_sort(chunk->data));
    }

    void sort_thread()
    {
        while(!end_of_data)
        {
            try_sort_chunk();
            std::this_thread::yield();             // hint to reschedule to the next thread
        }
    }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);
}
```

Dividing work by task type

- **Dividing work by task type to separate concerns**
  - run each of the tasks in a separate thread
  - tasks are dependent, so need to communicate with each other
    - e.g, user interface thread handles the user interface, but it might have to update it when asked to do so by other threads
    - e.g., thread running the background task focuses on the operations required for that task; it just happens that one of them is “allow task to be stopped by another thread”
- **Dividing a sequence of tasks between threads**
  - if the task consists of applying the same sequence of operations to many independent data items
  - you can use a *pipeline* to exploit the available concurrency of your system
    - create a separate thread for each stage in the pipeline—one thread for each of the operations in the sequence

Factors affecting the performance of concurrent code

① **How many processors?**

② **Data contention and cache ping-pong**

③ **False sharing**

④ **How close is your data?**

⑤ **Oversubscription and excessive task switching**

(1) How many processors?

- **Big factor that affects the performance of a multithreaded application**
  - you don't know exactly what the target hardware is
    - e.g., you might develop on a dual- or quad-core system, but your customers' systems may have
      - one multicore processor (with any number of cores), or
      - multiple single-core processors, or
      - even multiple multicore processors
- **Oversubscription**
  - more than the number of threads actually ready to run
  - waste processor time switching between the threads

(2) Data contention and cache ping-pong

- **Two threads are executing concurrently on different processors and both reading the same data**
  - data will be copied into their respective caches (no problem)
- **One of the threads modifies the data**
  - this change has to propagate to the cache on the other core
  - this can be *phenomenally* *slow* in terms of CPU instructions
    - equivalent to many hundreds of individual instructions
  - e.g., `counter` is global
    - `fetch_add` is a read-modify-write operation
    - it needs to retrieve the most recent value of the variable

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
    while(counter.fetch_add(1,std::memory_order_relaxed)<100000000)
    {
        do_something();
    }
}
```

High contention

- If another thread on another processor is running the same code
  - the data for counter must be passed back and forth between the two processors and their corresponding caches
- If `do_something()` is short enough, or if there are too many processors running this code
  - the processors are waiting for each other

Cache ping-pong

- data are passed back and forth between the caches many times
  - seriously impact the performance of the application
- acquiring a mutex in a loop is similar to the previous code from the point of view of data accesses

```
std::mutex m;
my_data data;
void processing_loop_with_mutex()
{
    while(true)
    {
        std::lock_guard<std::mutex> lk(m);
        if(done_processing(data)) break;
    }
}
```

(3) False sharing

Cache line

- processor caches don't generally deal in individual memory locations
- instead, they deal in blocks of memory called cache lines
  - typically 32 or 64 bytes in size

False sharing

- situation : the data items in a cache line are unrelated and accessed by different threads
- problem : even though each thread only accesses its own array entry, the cache hardware still has to play cache ping-pong
- solution : structure the data so that
  - data items to be accessed by the same thread are close together in memory
  - data items to be accessed by separate threads are far apart in memory and thus more likely to be in separate cache lines

(4) How close is your data?

- **Data proximity issue (in a single thread)**
  - situation : the data accessed by a single thread is spread out in memory
  - problem
    - more cache lines must be loaded from memory onto the processor cache
    - it can increase memory access latency and reduce performance
- **Task switching issue**
  - situation
    - there are more threads than cores in the system
    - each core is going to be running multiple threads (task switching)
  - problem
    - you try to ensure that different threads are accessing different cache lines in order to avoid false sharing
    - when the processor switches threads, it's more likely to have to reload the cache lines

(5) Oversubscription and excessive task switching

- **It is typical to have more threads than processors**
  - unless you're running on *massively parallel* hardware
- **Extra threads enables the application to perform useful work**
  - rather than having processors sitting idle while the threads wait for
    - external I/O to complete
    - blocked on mutexes
    - condition variables
- **Too many additional threads**
  - there are more threads *ready to run* than available processors
  - operating system starts task switching quite heavily in order to ensure they all get a fair time slice
  - it increases the overhead of the task switching as well as compound any cache problems

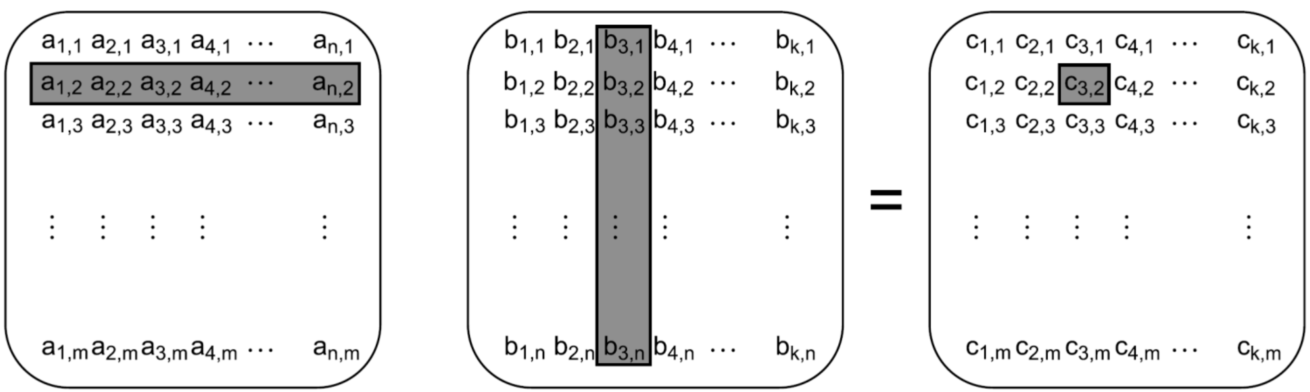
Designing data structures for multithreaded performance

- **Key things to bear in mind when designing data structures for multithreaded performance**
  - contention
  - false sharing
  - data proximity

- **You can often improve the performance**
  - by altering the data layout
  - by changing which data elements are assigned to which thread

Dividing array elements for complex operations

- **For reducing cache usage and chance of false sharing**
- **Example : multiplication of large matrices**
  - each thread calculate the results for a number of columns
    - need to read every value from the first matrix
  - each thread calculate the results for a number of rows
    - need to read every value from the second matrix
  - each thread calculate the results for a rectangular subset
    - only need to read the values corresponding to the rows and columns
    - the same false-sharing potential as division by columns



Data access patterns in other data structures

- **Considerations for optimizing the data access patterns**
  - Try to adjust the data distribution between threads so that data that's close together is worked on by the same thread
  - Try to minimize the size of data required by any given thread
  - Try to ensure that data accessed by separate threads is sufficiently far apart to avoid false sharing
- **Some issues**
  - If the mutex and the data items are close together in memory, this is ideal for a thread that acquires the mutex, but …

test the mutex contention issue

```
struct protected_data
{
    std::mutex m;
    char padding[65536];
    my_data data_to_protect;
};
```

test for false sharing of array data

```
struct my_data
{
    data_item1 d1;
    data_item2 d2;
    char padding[65536];
};
my_data some_array[256];
```

Additional considerations when designing for concurrency

- **Code is said to be *scalable***
  - if the performance increases as more processing cores are added to the system
    - in terms of reduced speed of execution or increased throughput
  - Ideally, the performance increase is linear
    - system with 100 processors performs 100 times better than a system with one processor

- **Exception safety is a matter of correctness**
  - If the code isn't exception safe, it can end up with broken invariants or race conditions

Exception safety in parallel algorithms

- **Sequential algorithm**
  - it only has to worry about ensuring that it tidies up after itself when throwing an exception
    - to avoid resource leaks and broken invariants
  - it can allow the exception to propagate to the caller for them to handle
- **Parallel algorithm**
  - it often requires taking more care with regard to exceptions than a normal sequential algorithm
  - many of the operations will be running on separate threads
  - the exception can't be allowed to propagate because it's on the wrong call stack
  - if a function spawned on a new thread exits with an exception, the application is terminated

A naive parallel version of std::accumulate

```cpp
template<typename Iterator,typename T>
struct accumulate_block
{
    void operator()(Iterator first, Iterator last, T& result)
    {
        // can potentially throw
        result=std::accumulate(first, last, result);  // result is passed by value
    }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first, Iterator last, T init)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=(length+min_per_thread-1)/min_per_thread;

    unsigned long const hardware_threads=std::thread::hardware_concurrency();

    unsigned long const num_threads=
        std::min(hardware_threads!=0 ? hardware_threads : 2, max_threads);
```

```
    unsigned long const block_size= length / num_threads;

    std::vector<T> results(num_threads);
    std::vector<std::thread> threads(num_threads-1);

    Iterator block_start=first;
    for(unsigned long i=0; i<(num_threads-1); ++i)
    {
        Iterator block_end = block_start;
        std::advance(block_end, block_size);

        threads[i]=std::thread(
            accumulate_block<Iterator,T>(),                         // can potentially throw
            block_start, block_end, std::ref(results[i]));

        block_start=block_end;
    }
    accumulate_block()(block_start, last, results[num_threads-1]); // can potentially throw

    std::for_each(threads.begin(), threads.end(), std::mem_fn(&std::thread::join));

    return std::accumulate(results.begin(),results.end(),init);
}
```

Adding exception safety

- **Trying to calculate a result to return while allowing for the possibility that the code might throw an exception**
  - *precisely* what the combination of `std::packaged_task` and `std::future` is designed for

- **Removing one of the potential problems**
  - exceptions thrown in the worker threads are rethrown in the main thread

A parallel version of std::accumulate using std::packaged_task

```
template<typename Iterator,typename T>
struct accumulate_block
{
    T operator()(Iterator first, Iterator last)
    {
        return std::accumulate(first, last, T());         // now return the result directly
    }
};
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=(length+min_per_thread-1)/min_per_thread;

    unsigned long const hardware_threads=std::thread::hardware_concurrency();

    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);

    unsigned long const block_size=length/num_threads;
```

```
    std::vector<std::future<T> > futures(num_threads-1); // rather than a vector of results
    std::vector<std::thread> threads(num_threads-1);

    Iterator block_start=first;
    for(unsigned long i=0; i<(num_threads-1); ++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);

        std::packaged_task<T(Iterator,Iterator)> task(accumulate_block<Iterator,T>());
        futures[i]=task.get_future();
        // result will be captured in the future, as will any exception thrown
        threads[i]=std::thread(std::move(task), block_start, block_end);

        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);

    std::for_each(threads.begin(), threads.end(), std::mem_fn(&std::thread::join));

    T result=init;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();
    }
    result += last_result;
    return result;
}
```

- **Remaining problem**
  - the leaking threads if an exception is thrown between when you spawn the first thread and when you've joined with them all
- **Simplest solution**
  - catch any exceptions
  - join with the threads that are still joinable()
  - rethrow the exception

```
try
{
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        // ... as before
    }
    T last_result=accumulate_block()(block_start,last);

    std::for_each(threads.begin(),threads.end(),
        std::mem_fn(&std::thread::join));
}
catch(...)
{
    for(unsigned long i=0;i<(num_thread-1);++i)
    {
        if(threads[i].joinable())
            thread[i].join();
    }
    throw;
}
```

- **Good thing**
  - all the threads will be joined, no matter how the code leaves the block
- **Bad things**
  - try-catch blocks are ugly
    - joining the threads both in the “normal” control flow and in the catch block
  - you have duplicate code
- **Solution**
  - extracting the joining part out into the destructor of an object
    - the idiomatic way of cleaning up resources in C++

```
class join_threads
{
    std::vector<std::thread>& threads;
public:
    explicit join_threads(std::vector<std::thread>& threads_):
        threads(threads_)
    {}
    ~join_threads()
    {
        for(unsigned long i=0;i<threads.size();++i)
        {
            if(threads[i].joinable())
                threads[i].join();
        }
    }
};
```

An exception-safe parallel version of std::accumulate

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first, Iterator last, T init)
{
    unsigned long const length=std::distance(first,last);

    ...

    std::vector<std::future<T> > futures(num_threads-1);
    std::vector<std::thread> threads(num_threads-1);
    // create an instance of your new class to join with all the threads on exit
    join_threads joiner(threads);

    ...
    // no explicit join loop

    T result=init;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get(); // will block until the results are ready
        // you don’t need to have explicitly joined with the threads at this point
    }
    result += last_result;
    return result;
}
```

Exception safety with `std::async()`

- **The same thing can be done with `std::async()`**
  - The library ensures that the `std::async` calls make use of the hardware threads that are available
    - without creating an overwhelming number of threads
  - It takes advantage of the hardware concurrency

- **It's still exception safe**
  - If an exception is thrown by the recursive call (*Line 20*)
    - the future created from the call to `std::async` (*Line 17*) will be destroyed as the exception propagates
    - this will in turn wait for the asynchronous task to finish (thus avoiding a dangling thread)
  - On the other hand, if the asynchronous call throws
    - this is captured by the future
    - the call to get() (*Line 22*) will rethrow the exception

An exception-safe parallel version of std::accumulate using std::async

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first, Iterator last, T init)
{
    unsigned long const length=std::distance(first,last);
    unsigned long const max_chunk_size=25;

    if(length<=max_chunk_size)
    {
        return std::accumulate(first,last,init);
    }
    else
    {
        Iterator mid_point=first;
        std::advance(mid_point,length/2);

        std::future<T> first_half_result= // spawn an asynchronous task to handle that half
            std::async(parallel_accumulate<Iterator,T>, first, mid_point, init);

        // the second half of the range is handled with a direct recursive call
        T second_half_result = parallel_accumulate(mid_point, last, T());

        return first_half_result.get()+second_half_result;
    }
}
```

Thank you!