Thread Building Blocks and OpenMP

COMP 522 MULTICORE COMPUTING

Deyu Fu
Intel® Threading Building Blocks

- Intel TBB is a C++ template library
- Abstracts away implementation details of task parallelism
  - number of threads
  - mapping of task to threads
  - mapping of threads to processors
  - memory and locality
Threading Building Blocks

Components

- Parallel algorithmic templates
  - parallel_for
  - parallel_reduce
  - Pipeline
  - parallel_sort
  - parallel_while
  - parallel_scan

- Synchronization Primitives
  - atomic ops on integer types
  - Mutex
    - spin_mutex
    - queuing_mutex
    - spin_rw_mutex
  - queuing_rw_mutex

- Concurrent containers
  - concurrent hash map
  - concurrent queue
  - concurrent vector

- Work-stealing Task scheduler

- Scalable Memory allocators
  - cache_aligned_allocator
  - scalable_allocator

- Timing
  - tick_count
Strengths of TBB

- Reduces source code complexity over Pthreads
- Determines runtime parameters automatically
- Automatically scales parallelism to utilize all available cores
- Schedules tasks using work-stealing inspired by the early Cilk scheduler
Open Specifications for Multi Processing

• An API for explicit multi-threaded, shared memory parallelism
• OpenMP is a directive-based programming model
  • pragmas (C/C++); significant comments in Fortran
  • if a compiler does not recognize a directive, it ignores it
  • If OpenMP is disabled when compiling a program, the program will execute sequentially
OpenMP Components

- #pragma omp parallel
- #pragma omp for
  - #pragma omp ordered
- #pragma omp section
- #pragma omp critical
- #pragma omp barrier
- #pragma omp single
- #pragma omp task
- #pragma omp target
TBB, Cilk, OpenMP and PThread

• Task-based

• Implementation model
  • Library-based: Pthread, TBB
  • Directive-based: OpenMP
  • Language: Cilk

• This talk compares TBB, Cilk and OpenMP
The TBB Task Scheduler

- Work-stealing task scheduler
- Pool of threads
- Local pools
  - Contains tasks that are ready to run
TBB Scheduling Strategies

• Randomized work-stealing algorithm
  • If a thread’s pool becomes empty, it attempts to steal a task from another random thread’s pool
  • Keep doing it until steal from a non-empty pool

• Alternative
  • static scheduling methods - threads are assigned work up-front – OpenMP schedule(static)
  • dynamic scheduling methods - a central pool of tasks
Optimal Bounds

\[ E(T|P) = O\left(\frac{T\downarrow 1}{P} + T\downarrow \infty \right) \]

- \( T\downarrow 1 \): sequential time of the application
- \( P \): number of threads
- \( T\downarrow \infty \): Critical Path
OpenMP: Fork-Join Parallelism

- Parallel region and master thread

Diagram showing the sequence of operations for parallel and sequential work, including:
1. Find 3 available threads
2. Assign thread IDs
3. Assign work
4. Signal ready
5. Initialize thread state
6. Barrier
7. Cleanup
Nearly-Constant-Time Thread Allocation
OpenMP Scheduling Strategies

• Four scheduling classes mapping iterations to threads
  • Static: work partitioned at compile time
  • Dynamic: work evenly partitioned at run time
  • Guided: guided self-scheduling
  • Runtime

• Task scheduling
  • Tasks are executed by threads of a team
  • A task can be tied to a thread (i.e. migration/stealing not allowed)
    • By default: a task is tied to the first thread that executes it
  • Untied tasks
    • implementation may schedule for locality and/or load balance
Data Scoping is Tricky

- Static and global variables are shared
- Automatic (local) variables are private
- Variables for orphaned tasks are firstprivate(c) by default
  - each private copy of c is initialized with the value of c in the “initial thread”, which is the one that encounters the parallel directive
Example: Fibonacci number

Cilk++

```cpp
class FibTask : public task {
  // describe the task
  public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_) : n(n_), sum(sum_) {}
  task* execute() {
    // do the work
    if (n <= cutoff) *sum = SerialFib(n);
    else {
      // generate more tasks
      long x, y;
      FibTask& a = *new(allocate_child());
      FibTask(n-1, &x);
      FibTask& b = *new(allocate_child());
      FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return n1 + n2;
  }
}
```

TBB

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

OpenMP

```cpp
int main(int argc, char **argv)
{
  int n, result;
  n = atoi(argv[1]);
  #pragma omp task shared(x)
  x = fib(n - 1);
  #pragma omp task shared(y)
  y = fib(n - 2);
  #pragma omp taskwait
  result = x + y;
  printf("fib(%d) = %d\n", n, result);
}
```

Need `shared` for x and y; default would be `firstprivate`
Example: Fibonacci number

- **Serial Code:**
  ```c
  long SerialFib( long n ) {
    if( n<2 )
      return n;
    else
      return SerialFib(n-1)+SerialFib(n-2);
  }
  ```

- **Parallel Code for TBB:**
  ```c
  long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(task::allocate_root())
      FibTask(n,&sum);
    task::spawn_root_and_wait(a);
    return sum;
  }
  ```
Task Scheduler Design

- **Task graphs**
  - a directed graph where nodes are tasks
  - Each node points to its *parent* or *NULL*
  - *Parent*: another task that is waiting on it to complete
  - *RefCount*: counts the number of tasks that have it as parent
  - *Depth*: one more than the depth of its parent

- Work of the task is performed by a user-defined function *execute*
• TBB requires users to set task refcounts explicitly with the \texttt{set\_ref\_count}, instead of atomically incrementing it in \texttt{allocate\_child}.

• An additional guard reference is required for this.
Explain of Example

- Each task that spawns children waits at `spawn_and_wait_for_all` until all children complete.
- A thread that enters a `spawn_and_wait_for_all` is free to execute other ready tasks while it waits.

```
class FibTask: public task {  // describe the task
  public:
    const long n;
    long* const sum;
    FibTask(long n_, long* sum_): n(n_), sum(sum_) {}  
  task* execute() {  // do the work
    if(n<=cutoff) *sum=SerialFib(n);  
    else {  // generate more tasks
      long x, y;
      FibTask& a = *new(allocate_child());
      FibTask(n-1, &x);
      FibTask& b = *new(allocate_child());
      FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum=x+y;
    }
};
```
A task goes into a pool only when it is deemed ready to run, i.e., it has been spawned and has a refcount of 0.
Managing Task Pools

Each per-thread task pool is implemented as an array of lists of tasks.
Breadth-First Theft and Depth-First Work

- A task
  - performs work
  - may spawn additional tasks into the local pool

<table>
<thead>
<tr>
<th>Pros</th>
<th>Depth-first work</th>
<th>Breadth-first work</th>
</tr>
</thead>
<tbody>
<tr>
<td>provides better temporal locality</td>
<td>limits the space</td>
<td>breadth-first unfolding</td>
</tr>
<tr>
<td>limits the space</td>
<td></td>
<td>exponential number of nodes exist simultaneously</td>
</tr>
</tbody>
</table>

| Cons                          | Limits parallelism                      | excessively consuming memory                          |
Breadth-First Theft and Depth-First Work

- Solution to balance efficient execution and parallelism
  - Breadth-first theft and Depth-first work

- Breadth-first theft
  - Raises parallelism sufficiently to keep threads busy

- Depth-first work
  - Keeps each thread operating efficiently once it has sufficient work to do
Scheduling Trade-offs and Optimizations

- One difference
  - In Cilk, stolen tasks is freed from victim processor stack
  - In TBB, stolen tasks remain on victim processor on wait calls

- Another difference
  - Cilk relies on the compiler to perform Cilk-specific optimizations
  - TBB cannot rely on compiler for optimizations

- TBB task API has been designed to allow users do scheduling optimizations manually
Continuation Tasks

- A stealing rule
  - Only steal tasks deeper than any waiting task
- Processor stack may overflow without this rule
  - Stolen tasks remain on victim processor on wait calls
  - Shallow tasks may unfold on top
- Restrict stealing choice thus reduce parallelism
Continuation Tasks

Solution: *continuation tasks*

- A task replace itself in the graph with a continuation task
- Return and free up its stack space
- Continuation task is not in stack but in heap
- Children complete
- Continuation task is spawned to finish the work

Result

- Only active tasks are on the processor stack
Code of Continuation Tasks

• Note
  • Task a and task b are children of c
  • Overhead introduced

```cpp
FibContinuation& c =
  *new( allocate_continuation() )
  FibContinuation(sum);
FibTask& a = *new( c.allocate_child() )
  FibTask(n-2,&c.x);
FibTask& b = *new( c.allocate_child() )
  FibTask(n-1,&c.y);
c.set_ref_count(2);
c.spawn( b );
c.spawn( a );
return NULL;
```
Scheduler Bypass

- Scheduler Bypass
  - Task’s `execute` function explicitly returns the next task to execute

- Note that this can only be used when using continuation tasks

```javascript
  e.spawn(a);
  return &a;
```

- Benefit
  - Avoid complex scheduler logic to select next task
Task Recycling

• Task Recycling
  • Instead of deallocate task object, recycle task objects to do another task

• Note that this can only be used when using continuation tasks

• Benefit
  • Avoiding the repeated allocation and deallocation

```
FibContinuation& c =
    *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
recycle_as_child_of(c);
n -= 2;
sum = &c.x;
c.set_ref_count(2);
c.spawn( b );
return this;
```
Scalable Memory Allocation

• Two principles in general-purpose memory allocators
  • Efficient use of memory space
  • Minimization of CPU overhead

• More important in threaded
  • Scalable
  • Cache locality and prevention of false sharing

• Goal
  • Minimize use of single global lock
The TBB Scalable Allocator

- Handle “large objects”
  - Border between large and “regular” - slightly below 8K
  - Explicitly managing 8K – 64K is future work
- Note that large objects do not show up frequently

Figure 3: High-level design of the scalable allocator
The TBB Scalable Allocator

- Minimize requests and maximize reuse
  - Request memory from the OS in 1MB chunks and divides each chunk into 16K-byte aligned *blocks*
  - Requested memory never returned to OS
- Request new blocks only when no available in both local and global heap

**Figure 3: High-level design of the scalable allocator**
The TBB Scalable Allocator

- Thread-private heaps
  - cut down synchronization and reduce false sharing
- Heaps are *segregated* into *bins*
- Bins
  - Double-linked list of blocks
  - Each bin only store objects with similar size
The TBB Scalable Allocator

- Benefits of bins
  - Per-object header not needed
  - Block header contains all information
  - Objects are tightly packed in the block

- Separate free lists
  - Use space in private free list to allocate when possible – no synchronization
  - Fetch everything from public free list when needed – synchronization also needed
Experimental Results

• Presenting performance data to evaluate the performance

• All results were collected on a server system with two Quad-Core Intel® Xeon® processors X53555 running Red Hat Enterprise Linux 4

• Using 1 through 8 threads
Performance of the Task Scheduler

- Show results for applications using TBB
  - Without scheduling optimization (TBB)
  - Using only continuation passing (TBB+C)
  - Using continuation passing and scheduler bypass (TBB+CB)
  - Using continuation passing, scheduler bypass, and task recycling (TBB+CBR)

- For each benchmark we show the speedups relative to an optimized serial implementation
Results

- Fibonacci
  - Calculate the 50th Fibonacci number, with serial cutoffs of 12 and 20
  - Higher is better
Results

- **Parallel_for**
  - Uses Intel TBB parallel_for algorithm to iterate over a range of 100 million integers applying an empty loop body to each element
  - Higher is better
Results

• **Sub_string_finder**
  • For each position in a string, calculates the location of the largest substring found elsewhere in the string that matches a string starting at the current position
  • Higher is better
Results

- Tacheon
  - A 3D ray tracer that is distributed with TBB library
  - Higher is better
Results

• Linear speedups for several small benchmarks
• The overhead of the TBB scheduler is seen
• Continuation passing alone often leads to a slowdown
• Enable scheduler bypass and task recycling can save some performance
Performance of Memory Allocation

- Other allocator being compared
  - The default memory allocator of GNU C runtime library (glibc) v2.3.4.
  - Google’s TCMalloc (google-perftools v0.92)
  - Hoard v3.6.2
  - Memory Tuning System* (MTS)
  - SmartHeap* for SMP
Results

- False-sharing
  - check for the performance penalty due to false sharing
  - Lower is better
Results

• Speed-cross
  • Stress-test of the multi-threaded behavior
  • Lower is better
Results

- MTS demo test
  - Attempts to mimic typical allocation behavior
  - Lower is better
Results

- Larson
  - Model the allocation behavior of a multi-threaded server
  - Higher is better
Combined Performance

• For this analysis, we use the tree sum application provided in the TBB library
  • First generates a binary tree that contains nodes each holding a float value
  • Then performs a summation of the values in the tree
Result

- Allocation phase
  - Allocate nodes of the tree
  - Higher is better
Result

- Sum phase
  - Sum the values contained in the allocated tree
  - Higher is better
Conclusion

• Task scheduler and scalable memory allocator both have good performance
• The scheduling optimizations were shown to have a small performance impact
• TBB reduce the need of users to understand the many complex issues
• Though model is important, sometime a tricky implementation optimization can lead to a significant performance improvement and become very important
  • OpenMP thread allocator on IBM Blue Gene/Q
  • TBB scalable memory allocator