L20: Putting it together: Tree Search (Ch. 6)

November 29, 2011

Outline

- Next homework
- Poster information
- SC Followup: Student Cluster Competition
- Chapter 6 shows two algorithms (N-body and Tree Search) written in the three programming models (OpenMP, Pthreads, MPI)
  - How to approach parallelization of an entire algorithm (Foster’s methodology is used in the book)
  - What do you have to worry about that is different for each programming model?

Homework 4, due Friday, December 1 at 11:59PM

Instructions: We’ll go over these in class on November 30. Handin on CADE machines:

```
"handin cs4961 hw4 <problem>
```

1. Given the following sequential code, sketch out two CUDA implementations for just the kernel computation (not the host code). Determine the memory access patterns and whether you will need synchronization. Your two versions should (a) use only global memory and (b) use both global and shared memory. Keep in mind capacity limits for shared memory and limits on the number of threads per block.

```
int a[1024][1024], b[1024];
for (i=0; i<1024; i++)
    for (j=0; j<1024-i; j++)
        b[i+j] += a[j][i] + a[j][i+1] + a[j][i+2] + a[j][i+3];
```

Administrative

- Next homework, CUDA, MPI (Ch. 3) and Apps (Ch. 6)
  - Goal is to prepare you for final
  - We’ll discuss it in class on Thursday
  - Solutions due on Friday, Dec. 1 (should be straightforward if you are in class)
- Poster dry run on Dec. 6, final presentations on Dec. 8
- Optional final report (4-6 pages) due on Dec. 14 can be used to improve your project grade if you need that
Homework 4, cont.

2. Programming Assignment 3.8, p. 148. Parallel merge sort starts with \( n/\text{comm}_sz \) keys assigned to each process. It ends with all the keys stored on process 0 in sorted order. When a process receives another process’ keys, it merges the new keys into its already sorted list of keys. Parallel mergesort then the processes should use tree-structured communication to merge the global list onto process 0, which prints the result.

3. Exercise 6.27, p. 350. If there are many processes and many redistributions of work in the dynamic MPI implementation of the TSP solver, process 0 could become a bottleneck for energy return. Explain how one could use a spanning tree of processes in which a child sends energy to its parent rather than process 0.

4. Exercise 6.30, p. 350 Determine which of the APIs is preferable for the n-body solvers and solving TSP:
   a. How much memory is required... will data fit into the memory?
   b. How much communication is required by each of the parallel algorithms (consider remote memory accesses and coherence as communication)
   c. Can the serial program be easily parallelized by the use of OpenMP directives? Do they need synchronization constructs such as condition variables or read-write locks?

Overview

• Last time we talked about solving n-body problems.
  - “Regular” computation on a grid
  - Parallelization relatively straightforward through standard data and computation partitioning
  - Modest concerns about load imbalance
• This lecture on Tree Search (specifically, single-source shortest path)
  - “Irregular”
  - amount of work associated with a node or path varies
  - Graph may be represented by dense or sparse adjacency matrix (sparse adjacency matrix is also irregular)
  - Impact on implementation
    - Load balancing? Dynamic scheduling?
    - Termination
    - Collecting result plus path, synchronization

Tree search problem (TSP)

• An NP-complete problem.
• No known solution to TSP that is better in all cases than exhaustive search.
• Ex., the travelling salesperson problem, finding a minimum cost tour.
• (Also called single-source shortest path)

A Four-City TSP

0 is root or single source. What is the shortest path?
We’ll need to eliminate recursion

- Non-recursive implementation is more flexible
- Shared data structures
- Freedom to schedule threads/processes flexibly

How to eliminate recursion

- Explicit management of "stack" data structure
- Loop instead of recursive calls

Pseudo-code for a second solution to TSP that doesn’t use recursion (two solutions in text)

```c
Terminate and find best solution
Can this city be added:
on path?
new to path?
shorter than existing paths?

Global variable - stores "best tour"

- How to know whether global "best tour" value is better?
- Guarantee that any writes will be performed atomically
- Textbook suggests reading it without locking
- Value may be slightly out of date, but just increases work slightly since value is monotonically decreasing
- Cost of synchronizing read not worth it
- If the process’s value is not as good, not updated

Pseudo-code for a recursive solution to TSP using depth-first search

```c
```
Making sure we have the “best tour”

In the case where a thread tests and decides it has a better global solution, we need to ensure two things:

1) That the process locks the value with a mutex, preventing a race condition.
2) In the possible event that the first check was against an old value while another process was updating, we do not put a worse value than the new one that was being written.

- We handle this by locking, then testing again.

First scenario

Second scenario

Pseudo-code for a Pthreads implementation of a statically parallelized solution to TSP:

Master thread does static partitioning using breadth-first search

Thread operates on its private stack

Needs mutex (next slides)

Copyright © 2010, Elsevier Inc. All rights Reserved
A Careful Look at Load Imbalance

- Load imbalance very likely
  - The master thread is partitioning the work
  - Some expensive paths will be pruned early
- What to do?
  - Schedule some work statically and some dynamically for under-utilized processors
  - Minimize cost of moving work
  - This could be done a lot smarter than it is
- Challenges:
  - Which processes are available to take on more work?
  - When does execution terminate?

Dynamic Parallelization of Tree Search Using Pthreads

- Termination issues.
- Code executed by a thread before it splits:
  - It checks that there are at least two tours in its stack.
  - It checks that there are threads waiting.
  - It checks whether the new_stack variable is NULL.

Pseudo-Code for Pthreads Terminated Function

```c
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
    new_stack == NULL) { // Do I have extra work, and are there idle threads?
    lock_term_mutex;
    if (threads_in_cond_wait > 0 && new_stack == NULL) {
        Split my_stack creating new_stack;
        pthread_cond_signal(&term_cond_var);
        Wake up an idle thread
    }
    unlock_term_mutex;
    return 0; // Terminated = False; don't quit
} else if (!empty(my_stack)) { // Stack not empty, keep working
    return 0; // Terminated = false; don't quit
} else { // No stack is empty
    lock_term_mutex;
    if (threads_in_cond_wait == thread_count) { // Less thread
        threads_in_cond_wait++;
        pthreadCondBroadcast(&term_cond_var);
        unlock_term_mutex;
        return 1; // Terminated = true; quit
    }
    else if (term_flag) { // Other threads still working, wait for work
        threads_in_cond_wait++;
        pthreadCondWait(&term_cond_var, &term_mutex);
        unlock_term_mutex;
        return 0; // Terminated = false
    } else { // All threads done
        unlock_term_mutex;
        return 1; // Terminated = true; quit
    }
}
```

Pseudo-Code for Pthreads Terminated Function, cont.

```c
Idle thread gets new work
```
**Grouping the termination variables**

```c
typedef struct {
    my_stack_t new_stack;
    int threads_in_cond_wait;
    pthread_cond_t term_cond_var;
    pthread_mutex_t term_mutex;
} term_struct;

term_t term; // global variable
```

**Parallelizing the Tree Search Programs Using OpenMP**

- Some basic issues implementing the static and dynamic parallel tree search programs as Pthreads.
- A few small changes can be noted.

```c
#pragma omp single
```

**Other OpenMP changes**

- Use a critical section instead of a mutex to guard selection of “best tour”, but similar “optimistic” synchronization.
- Need a different construct for conditional wait (not clean)
  - Next slide, but more in textbook to select particular idle thread
  - Requires auxiliary stack of idle threads to select “awakened_thread”

**OpenMP emulated condition wait**

```c
/* Global vars */
int awakened_thread = -1;
work_remains = 1; /* true */
...
cmp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
cmp_set_lock(&term_lock);
```
MPI is similar

- Static and dynamic partitioning schemes
- Maintaining "best_tour" requires global synchronization, could be costly
  - May be relaxed a little to improve efficiency
  - Alternatively, some different communication constructs can be used to make this more asynchronous and less costly
  - MPI_Iprobe checks for available message rather than actually receiving
  - MPI_Ssend and other forms of send allow aggregating results of communication asynchronously

Sending a different number of objects to each process in the communicator to distribute initial stack

```c
int MPI_Scatterv(
    void* sendbuf, /* in */,
    int sendcounts, /* in */,
    int* displacements, /* in */,
    MPI_Datatype sendtype, /* in */,
    void* recvbuf, /* out */,
    int recvcount, /* in */,
    MPI_Datatype recvtype, /* in */,
    int root, /* in */,
    MPI_Comm comm, /* in */)
```

Gathering a different number of objects from each process in the communicator

```c
int MPI_Gatherv(
    void* sendbuf, /* in */,
    int sendcount, /* in */,
    MPI_Datatype sendtype, /* in */,
    void* recvbuf, /* out */,
    int recvcount, /* in */,
    int* displacements, /* in */,
    MPI_Datatype recvtype, /* in */,
    int root, /* in */,
    MPI_Comm comm, /* in */)
```

Checking to see if a message is available

```c
int MPI_Iprobe(
    int source, /* in */,
    int tag, /* in */,
    MPI_Comm comm, /* in */,
    int* msg_avail_p /* out */,
    MPI_Status* status_p /* out */)
```
Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```c
if (my_saw_tour_count >= 2) {
    full_request(my_stack);
    return false; // Still more work /
} else {
    // At most 1 available tour /
    send_rejects(); // Tell everyone who’s requested /
    // work that I have now /
    if (!empty_stack(my_stack)) {
        return false; // Still more work /
    } else {
        // Empty stack /
        if (comm_num == 1) return true;
        cut_of_work();
        while (false) { // Messages unrelated to work, termination /
            clear_stack(); // Messages unrelated to work, termination /
            return true; // No work left. Quit /
        } else if (work_request_sent) {
            // Request work from someone /
            // work_request_sent = true;
            // 
            // Check_for_work(work_request_sent, &work_request); // Request work /
            // Send_work_request();
            // work_request_sent = false;
            return false;
        } else {
            // Empty stack /
            if (comm_num == 1) return true;
            out_of_work();
            work_request_sent = false;
            while (false) { // Messages unrelated to work, termination /
                clear_stack(); // Messages unrelated to work, termination /
                if (no_work_left()) { // No work left. Quit /
                    return true; // No work left. Quit /
                } else {
                    // At most 1 available tour /
                }
            }
        }
    }
}
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Terminated Function for a Dynamically Partitioned TSP solver with MPI (2)

```c
} else if (work_request_sent) {
    send_work_request(); // Request work from someone /
    work_request_sent = true;
} else {
    check_for_work(work_request_sent, &work_request); // Request work /
    if (work_request) {
        receive_work(my_stack);
        return false;
    }
} else {
    // Empty stack /
    if (comm_num == 1) return true;
    out_of_work();
    while (false) { // Messages unrelated to work, termination /
        clear_stack(); // Messages unrelated to work, termination /
        if (no_work_left()) { // No work left. Quit /
            return true; // No work left. Quit /
        } else {
            // At most 1 available tour /
        }
    }
```

Copyright © 2010, Elsevier Inc. All rights Reserved
Packing data into a buffer of contiguous memory

```c
int MPI_Pack(int data_to_be_packed, int to_be_packed_count, MPI_Datatype datatype, int contig_buf, int position_p, MPI_Comm comm)
```

Unpacking data from a buffer of contiguous memory

```c
int MPI_Unpack(int contig_buf, int contig_buf_size, int position_p, int unpack_count, MPI_Datatype datatype, MPI_Comm comm)
```

Summary of Lecture

- This "tree search" is the hardest parallel programming challenge we have studied
  - Load imbalance leads to the need for dynamic scheduling, which in turn leads to termination challenges requiring elaborate synchronization and communication
  - This may be too hard for an efficient OpenMP implementation and is also challenging for MPI
    - We did not even talk about locality!
  - This chapter is dense, but getting through it is worth it
    - It shows REAL parallel programming and the need for elaborate language constructs
    - Other "irregular" algorithms are even more difficult