L21: Putting it together: Tree Search (Ch. 6)

November 20, 2012

Administrative

- CUDA project due Wednesday, Nov. 28
- Poster dry run on Dec. 4, final presentations on Dec. 6
- Optional final report (4-6 pages) due on Dec. 14 can be used to improve your project grade if you need that

Outline

- 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?

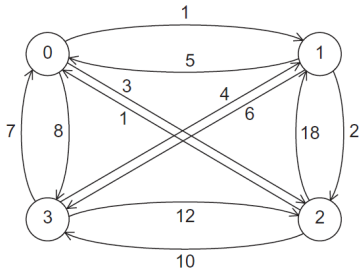
Overview

- Previously 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

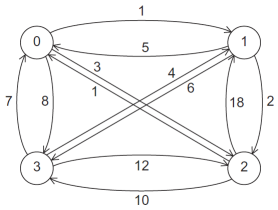
- An NP-complete problem.
- No known solution to TSP that is better in all cases than exhaustive search.
- Ex., the travelling salesperson problem, finding a minimum cost tour.
- (Also called single-source shortest path)

A Four-City TSP



0 is root or single source. What is the shortest path?

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




```
void Depth_first_search(tour_t tour) {
   city_t city;

   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
      for each neighboring city
         if (Feasible(tour, city)) {
            Add_city(tour, city);
            Depth_first_search(tour);
            Remove_last_city(tour);
         }
   }
} /* Depth_first_search */
```


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)

```
Push_copy(stack, tour);  // Tour that visits only the hometown
while (!Empty(stack)) {
   curr_tour = Pop(stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour))
         Update_best_tour(curr_tour);
   } else {
      for (nbr = n-1; nbr >= 1; nbr--)
         if (Feasible(curr_tour, nbr)) {
            Add_city(curr_tour, nbr);
            Push_copy(stack, curr_tour);
            Remove_last_city(curr_tour);
         }
   }
   Free_tour(curr_tour);
}
```

Termination and find best solution

Can this city be added:
on path?
new to path?
shorter than existing paths?

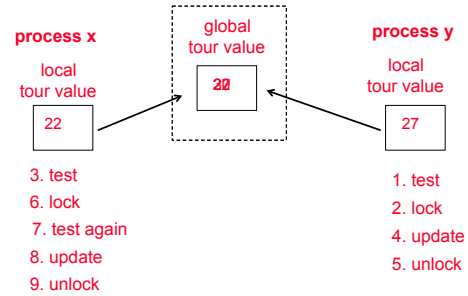
Global variable – stores "best_tour"

- 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

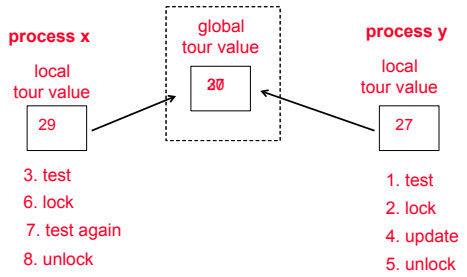
Making sure we have the "best tour"

- 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

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Master thread does static partitioning using breadth-first search

Thread operates on its private stack

Needs mutex (next slides)

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 it has 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

```
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
      new_stack == NULL) {
   lock term_mutex;
   if (threads_in_cond_wait > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      pthread_cond_signal(&term_cond_var);
   }
   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 { /* My stack is empty */
   lock term_mutex;
   if (threads_in_cond_wait == thread_count-1) { /* Last thread  */
                                                 /* running      */
      threads_in_cond_wait++;
      pthread_cond_broadcast(&term_cond_var);
      unlock term_mutex;
      return 1;  /* Terminated = true; quit */
```

Do I have extra work, and are there idle threads?

Wake up an idle thread

Termination test

Pseudo-Code for Pthreads Terminated Function, cont.

Idle thread gets new work

```
} else { /* Other threads still working, wait for work */
   threads_in_cond_wait++;
   while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
   /* We've been awakened */
   if (threads_in_cond_wait < thread_count) { /* We got work */
      my_stack = new_stack;
      new_stack = NULL;
      threads_in_cond_wait--;
      unlock term_mutex;
      return 0;  /* Terminated = false */
   } else {  /* All threads done */
      unlock term_mutex;
      return 1;  /* Terminated = true; quit */
   }
}  /* else wait for work */
}  /* else my_stack is empty */
```


Grouping the termination variables

```
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;
typedef term_struct* term_t;

term_t term;  // global variable
```


Parallelizing the Tree Search Programs Using OpenMP

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

Pthreads

```
if (my_rank == whatever)
```

OpenMP

```
# 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

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_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_Bsend 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

```
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

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


Checking to see if a message is available

```
int MPI_Iprobe(
      int            source           /* in  */,
      int            tag              /* in  */,
      MPI_Comm       comm             /* in  */,
      int*           msg_avail_p      /* out */,
      MPI_Status*    status_p         /* out */);
```


```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_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 none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```

Terminated Function for a Dynamically Partitioned TSP solver that Uses MPI.

Printing the best tour

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;

MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);
if (global_data.rank == 0) return;  /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_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 none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
```


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

```
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```


Packing data into a buffer of contiguous memory

```
int MPI_Pack(
      void*          data_to_be_packed    /* in     */,
      int            to_be_packed_count   /* in     */,
      MPI_Datatype   datatype             /* in     */,
      void*          contig_buf           /* out    */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      MPI_Comm       comm                 /* in     */)
```

Unpacking data from a buffer of contiguous memory

```
int MPI_Unpack(
      void*          contig_buf           /* in     */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      void*          unpacked_data        /* out    */,
      int            unpack_count         /* in     */,
      MPI_Datatype   datatype             /* in     */,
      MPI_Comm       comm                 /* in     */)
```

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

THE UNIVERSITY OF UTAH