Tree Search for Travel Salesperson Problem

Pacheco Text Book Chapt 6
T. Yang, UCSB CS140, Spring 2014
Outline

• Tree search for travel salesman problem.
  ▪ Recursive code
  ▪ Nonrecursive code

• Parallelization with threads on a shared memory machine
  ▪ Static partitioning
  ▪ Dynamic partitioning

• Parallelization with MPI
  ▪ Static partitioning
  ▪ Dynamic partitioning

• Data-intensive parallel programming with MapReduce
Tree search for TSP

- The NP-complete travelling salesperson problem: find a minimum cost tour.
  - A tour starts from a home town, visits each city once, and returns the hometown (source)
  - Also known as single-source shortest path problem

- 4-city TSP
  - Node -> city
  - Edge -> cost
  - Hometown = 0
Search Tree for Four-City TSP

Each tree path represents a partial tour from Hometown source
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 */
Recursive vs. nonrecursive design

• Recursion helps understanding of sequential code
  ▪ Not easy for parallelization.

• Non-recursive design
  ▪ Explicit management of stack data structure
  ▪ Loops instead of recursive calls
  ▪ Better for parallelization
    – Expose the traversal of search tree explicitly.
    – Allow scheduling of parallel threads (processes)

• Two solutions with code sample available from the text book.
  ▪ Focus on the second solution
Stack-based nonrecursive code implementation

Pop a partial tour [0] from stack

Push tour [0,1] to stack
- 0→1→2, 3
- 0→1→3, 7

Push tour [0,2]
- 0→2→1, 21
- 0→2→3, 13

Push [0,3]
- 0→3→1, 12
- 0→3→2, 20 x
- 0→3→2→1→0, 43 x

Push tour [0,1,3] to stack
- 0→1→2→3→0, 20
- 0→1→3→2→0, 20
- 0→1→3→2→3→0, 34 x
- 0→2→3→1→0, 22
- 0→2→3→1→2→0, 15
Stack-based nonrecursive code implementation

Stack operations:
- Push \([0,1,2]\)
- Push \([0,1,3]\)
- Push \([0,2,1]\)
- Push \([0,2,3]\)

Node operations:
- Pop \([0,1]\) from stack

Graph transitions:
- 0 \(\rightarrow 3, 8\)
- 0 \(\rightarrow 3 \rightarrow 2, 20 \times\)
- 0 \(\rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 0, 15\)
- 0 \(\rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0, 43 \times\)
- 0 \(\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0, 20\)
- 0 \(\rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0, 34 \times\)
- 0 \(\rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0, 22\)
Non-recursive solution to TSP
(Text book Page 304. Program 6.6)

```c
Push_copy(stack, tour);  // Tour starts from 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);
        Push_copy(stack, curr_tour);
    }
}
Free_tour(curr_tour);
```

- **Fetch a partial tour**
- **Update the best solution if found**
- **Expand the tour with each of feasible cities**
- **Push to stack**
Run-Times of the Three Serial Implementations of Tree Search

<table>
<thead>
<tr>
<th></th>
<th>Recursive</th>
<th>First Iterative</th>
<th>Second Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>(in seconds)</td>
<td>30.5</td>
<td>29.2</td>
<td>32.9</td>
</tr>
</tbody>
</table>

The digraph contains 15 cities. All three versions visited approximately 95,000,000 tree nodes.
Parallel processing with threads

Generate enough partial tours on the stack

Thread 1

0→1, 1

Thread 2

0→2, 3

Thread 3

0→3, 8
Shared global variables

• Stack
  ▪ Every thread fetches partial tours from the stack, expands, and pushes back to the stack.

• Best tour
  ▪ When a thread finishes a tour, it needs to check if it has a better solution than recorded so far.
  ▪ There’s no contention among readers.
  ▪ If another thread is updating while we read, we may see the old value or the new value.
    – The new value is preferable, but to ensure this would be more costly than it is worth.
Handling global variables

• **Stack**
  - Generate enough partial tours for all threads
  - Create private local stack per thread for each to expand locally --- static partitioning

• **Best tour**
  - During checkup, we let readers run without mutex lock.
  - When a thread starts to update the best tour
    - Use a mutex lock to avoid race condition
    - Double check if it is the best before real update.
Pthreads code of statically parallelized TSP

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
    curr_tour = Pop(my_stack);
    if (City_count(curr_tour) > 0) {
        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);
}
```
Pthreads code of statically parallelized TSP

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
    curr_tour = Pop(my_stack);
    if (City_count(curr_tour) == 0) {
        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);
}
```
void Update_best_tour(tour_t tour) {
    pthread_mutex_lock(&best_tour_mutex);
    if (Best_tour(tour)) {
        Copy_tour(tour, best_tour);
        Add_city(best_tour, home_town);
    }
    pthread_mutex_unlock(&best_tour_mutex);
}
First scenario

process x

local tour value

22

3. test
6. lock
7. test again
8. update
9. unlock

global tour value

20

process y

local tour value

27

1. test
2. lock
4. update
5. unlock
Second scenario

process x
local tour value
29
3. test
6. lock
7. test again
8. unlock

process y
local tour value
27
1. test
2. lock
4. update
5. unlock

Copyright © 2010, Elsevier Inc. All rights Reserved
Weakness of static partitioning

- **Load imbalance**
  - Many paths may be dynamically pruned
  - The workload assigned to threads can be uneven.

- **How to improve load balancing?**
  - Schedule computation statically initially.
  - Shift workload dynamically when some threads have nothing to do
    - Also called *work stealing*

- **Challenges/issues**
  - Idle threads wait for assignment. How to coordinate?
  - Which thread shifts its workload to others
  - When to terminate?
Solutions for the raised issues

• When to terminate?
  - All threads are idle and there no more workload to rebalance (all local stacks are empty)

• How can an idle thread get workload?
  - Wait in a Pthread condition variable
  - Wake up if somebody creates a new stack

• How to shift part of workload
  - Workload is represented in the tour stack
  - A busy thread can split part of its tours and create a new stack (pointed by new_stack variable)

• When can a thread split its stack?
  - At least two tours in its stack, there are threads waiting, and the new_stack variable is NULL.
Dynamic work stealing code for thread my_rank

Partition_tree(my_rank, stack);

while (!Terminated(&stack, my_rank)) {
    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, avail);
                Remove_last_city(curr_tour);
            }
    }
}
Code for Terminated()

- **Return 1 (true)**
  - Means no threads are active and the entire program should terminate.

- **Return 0 (false)**
  - Means this thread should work.
    - Either this thread has unfinished workload
      - Check if this thread should split its workload and let others work
        - Namely if it has at least two tours in its stack
        - and there are other threads waiting for some workload.
    - Or this thread has no workload and others have.
      - This thread can wait and fetch some workload from others.
Pseudo-Code for 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 */
    }

All threads are idle. Terminate
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) {
  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 */
Data structure for termination-related 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;

typedef term_struct* term_t;

term_t term;  // global variable
```
Run-times of Pthreads tree search programs

Two 15-city problems.
~95 million tree nodes visited

<table>
<thead>
<tr>
<th>Threads</th>
<th>First Problem</th>
<th>Second Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Static</td>
</tr>
<tr>
<td>1</td>
<td>32.9</td>
<td>32.7</td>
</tr>
<tr>
<td>2</td>
<td>27.9</td>
<td>28.9</td>
</tr>
<tr>
<td>4</td>
<td>25.7</td>
<td>25.9</td>
</tr>
<tr>
<td>8</td>
<td>23.8</td>
<td>22.4</td>
</tr>
</tbody>
</table>

(in seconds)

numbers of times stacks were split
Implementation of Tree Search Using MPI and Static Partitioning
From thread code to MPI code with static partitioning: Small code change

Process 0: Generate enough partial tours on the stack

Distribute initial tours to processes

Process 0 collects final best tour

How to check and update the best tour?
From thread code to MPI code

• Distribute initial partial tours to processes
  ▪ Use a loop of MPI_Send()
  ▪ Or use MPI_Scatterv() which supports non-uniform message sizes to different destinations.

• Inform the best tour to all processes
  ▪ A process finds a new best tour if the new cost is lower.
  ▪ Donot use blocking group communication MPI_Bcast()
  ▪ Sender: May use MPI_Send() to inform others
    – Safer to use MPI_Bsend() with its own buffer space.
  ▪ Receiver: Donot use blocking MPI_Recv().
    – Use asynchronous non-blocking receiving with MPI_Iprobe
Sending a different number of objects to each process in the communicator

```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* recvcounts /* in */,
    int* displacements /* in */,
    MPI_Datatype recvtype /* in */,
    int root /* in */,
    MPI_Comm comm /* in */
);
```
Modes and Buffered Sends

- **MPI provides four modes for sends.**
  - **Standard:** MPI_Send()
    - Use system buffer. Block if there is no buffer space
  - **Synchronous:** MPI_Ssend()
    - Block until a matching receive is posted.
  - **Ready:** MPI_Rsend()
    - Error unless a matching receive is posted before sending
  - **Buffered:** MPI_Bsend()
    - Supply your own buffer space.
Asynchronous non-blocking receive

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 */);
```

If a message is available, use standard MPI_Recv() to receive it.
At the end of MPI tree search

- Gather and print the best tour at the end.
  - Use MPI_Allreduce() to find the lowest from all.
  - Process 0 prints the final result
- Clean unreceived messages before shutting down MPI
  - Some messages won’t be received during parallel search.
  - Use MPI_Iprobe to receive outstanding messages before MPI_Finalize()
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;
Implementation of Tree Search Using MPI and Dynamic Partitioning
From static to dynamic partitioning

• Use majority of MPI code for static partitioning
• Special handling of distributed termination detection
  ▪ Emulate in a distributed memory setting
  ▪ Handle a process runs out of work (stack is empty)
    – Request work from MyRank+1 first.
    – Wait for receiving additional work
    – Quit if no more work is available
  ▪ A process with work splits its stack and sends work to an idle process.
    – Use special MPI message packaging
Send stack tour data structure with MPI message packing

Pack data into a buffer of contiguous memory

```c
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 */
    MPI_Comm comm /* in */
)
```
Unpacking data from a buffer of contiguous memory

```c
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 */
)
```
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 */
            }
        }
    }
}

With extra work, split stack and send to another process if needed.

At most 1 tour, reject other requests.

Notify everybody I am out of work.
} 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 I available tour */

No work here. Send a request to others, wait for assigned work. Quit if no more work available.
Distributed Termination Detection: First Solution

- Each process maintains a variable (oow) as # of out-of-work processes.
  - The entire computation quits if oow = n where n is # of processes.
- When a process runs out of work, notify everybody (oow++)
- When a process receives new workload, notify everybody (oow--)

This algorithm fails with out-of-order receiving from different processes.
<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Out of Work</td>
<td>Out of Work</td>
<td>Working</td>
</tr>
<tr>
<td></td>
<td>Notify 1, 2</td>
<td>Notify 0, 2</td>
<td>oow = 0</td>
</tr>
<tr>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Send request to 1</td>
<td>Send Request to 2</td>
<td>Recv notify fr 1</td>
</tr>
<tr>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
<td>oow = 1</td>
</tr>
<tr>
<td>2</td>
<td>oow = 1</td>
<td>Recv notify fr 0</td>
<td>Recv request fr 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 2</td>
<td>oow = 1</td>
</tr>
<tr>
<td>3</td>
<td>oow = 1</td>
<td>Send work to 1</td>
<td>oow = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>oow = 1</td>
<td>Recv work fr 2</td>
<td>Recv notify fr 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
</tr>
<tr>
<td>5</td>
<td>oow = 1</td>
<td>Notify 0</td>
<td>Working</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
</tr>
<tr>
<td>6</td>
<td>oow = 1</td>
<td>Recv request fr 0</td>
<td>Out of work</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
</tr>
<tr>
<td>7</td>
<td>Recv notify fr 2</td>
<td>Send work to 0</td>
<td>Send request to 1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>oow = 2</td>
<td>oow = 2</td>
</tr>
<tr>
<td>8</td>
<td>Recv 1st notify fr 1</td>
<td>Recv notify fr 2</td>
<td>oow = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Quit</td>
<td>Recv request fr 2</td>
<td>oow = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

oow -- # of out-of-work processes.

oow = 3, forcing Proc 0 quit before receiving work from Proc 1
Distributed Termination Detection: Second Solution

- Use energy conservation as a guiding principle
- Each process has 1 unit of energy initially
- When a process runs out of work, send its energy to process 0.
  - Process 0 adds this energy to its energy variable
- When a process splits its workload, divide its energy in half and sending half to the process that receives work.
  - Use precise rational addition to avoid underflow

Total energy in all processes = n during all steps.
The program terminates when process 0 finds its energy = n
Energy-based Termination Detection: Example

Proc 0
Energy=1

Energy=2

Energy=2 ½

Energy=3

Terminate

Proc 1
Energy=1

Out-of-work. Send energy 1 to Proc 0 Energy=0

Send work request to 2

Energy=1/2

Out-of-work. Energy=0

Proc 2
Energy=1

Split workload. Energy=1/2

Out-of-work. Energy=0

Out-of-work. Energy=0

Total energy in all processes =3 during all steps.
Performance of MPI and Pthreads implementations of tree search

<table>
<thead>
<tr>
<th>Th/Pr</th>
<th>First Problem</th>
<th></th>
<th>Second Problem</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>Pth</td>
<td>MPI</td>
<td>Pth</td>
<td>MPI</td>
</tr>
<tr>
<td>1</td>
<td>35.8</td>
<td>40.9</td>
<td>41.9</td>
<td>(0)</td>
</tr>
<tr>
<td>2</td>
<td>29.9</td>
<td>34.9</td>
<td>34.3</td>
<td>(9)</td>
</tr>
<tr>
<td>4</td>
<td>27.2</td>
<td>31.7</td>
<td>30.2</td>
<td>(55)</td>
</tr>
<tr>
<td>8</td>
<td>35.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>20.1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(in seconds)
Source code from the text book

- **Source code under chapter 6 directory:**
  - tsp_rec.c  Recursive sequential code
  - tsp_iter2.c  Nonrecursive sequential code
  - pth_tsp_stat.c  Pthread code with static partitioning
  - pth_tsp_dyn.c  Pthread code with dynamic partitioning
  - mpi_tsp_stat.c  MPI code with static partitioning
  - mpi_tsp_dyn.c  MPI code with dynamic partitioning
Concluding Remarks

• In a distributed memory environment in which processes send each other work, determining when to terminate is a nontrivial problem.

• Review memory requirements and the amount of communication during parallelization
  ▪ If memory required > memory per machine, then a distributed memory program may be faster
  ▪ If there is considerable communication, a shared memory program may be faster.