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
  - Nonrecusive 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

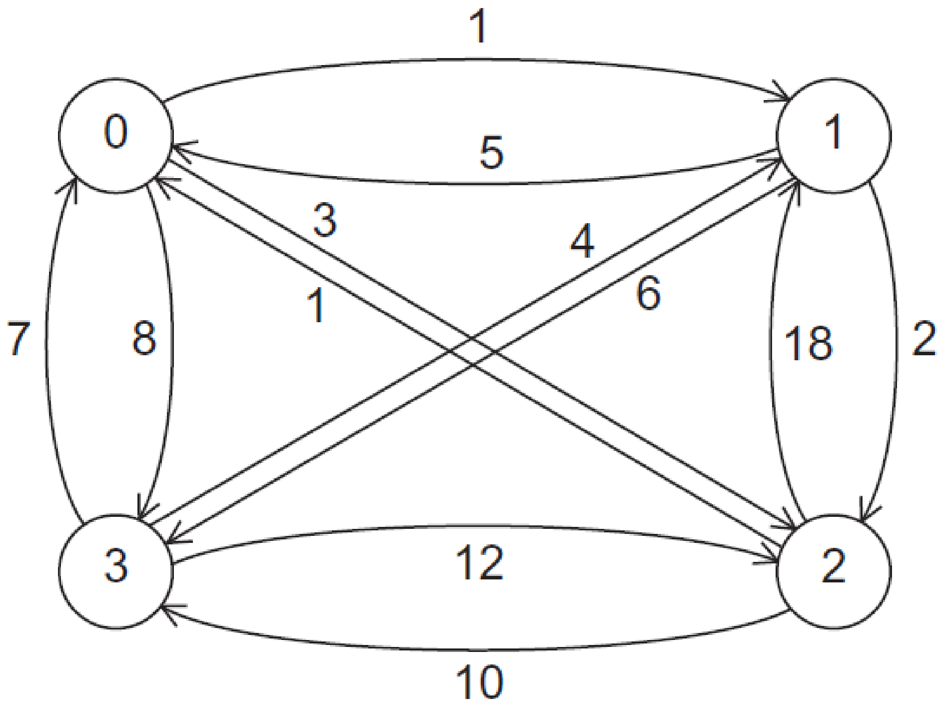
- **The NP-complete travelling salesperson problem: find a minimum cost tour.**
  - A tour starts from a home town, visits each city once, and returns the hometown (source)
  - Also known as single-source shortest path problem

- **4-city TSP**

Node->city

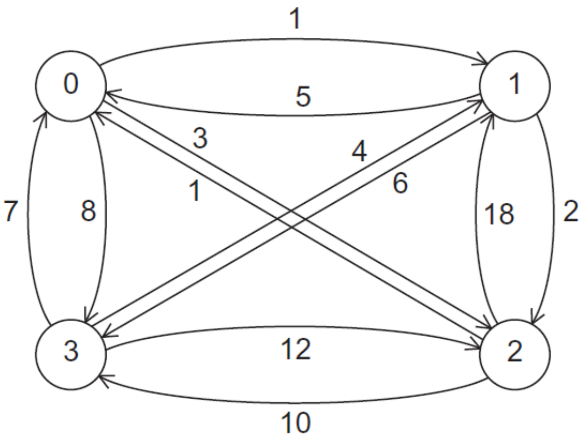
Edge->cost

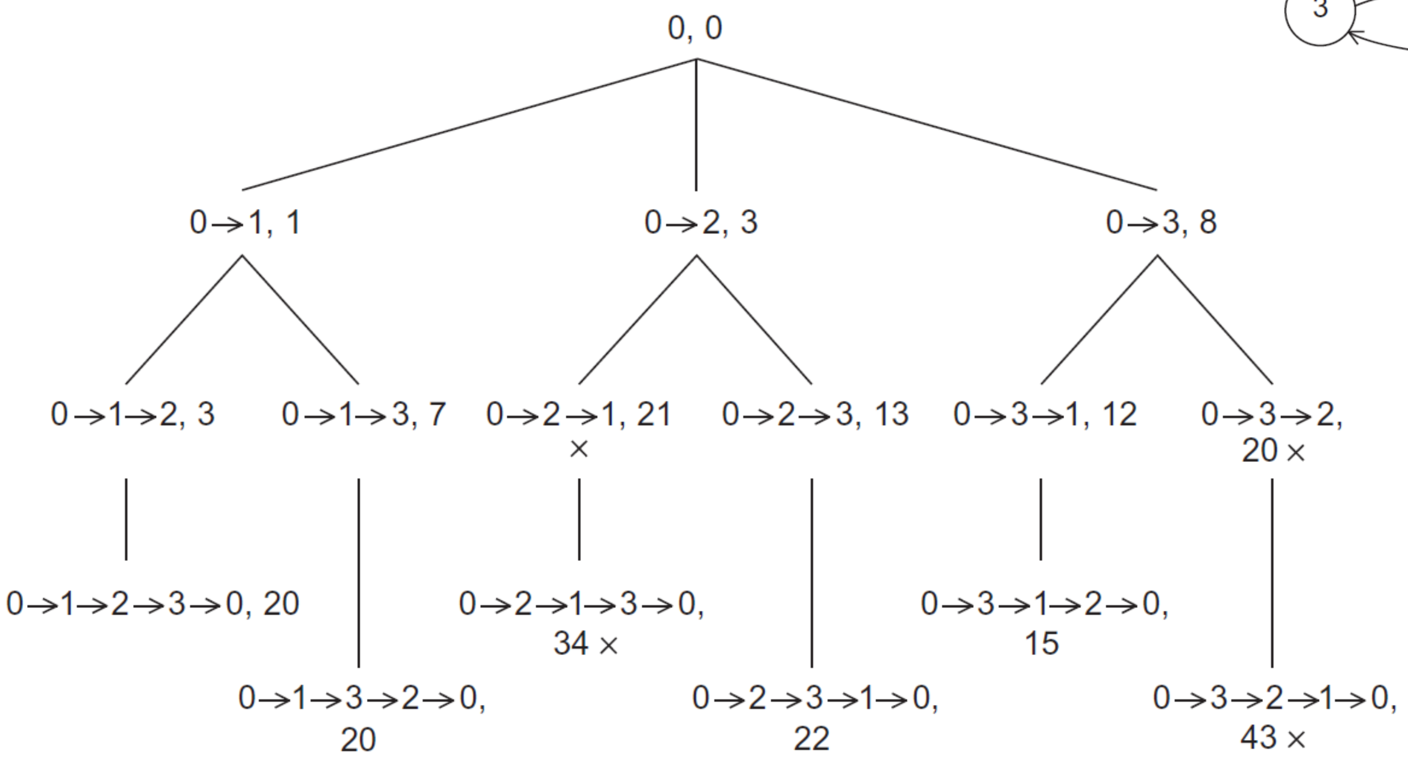
Hometown =0



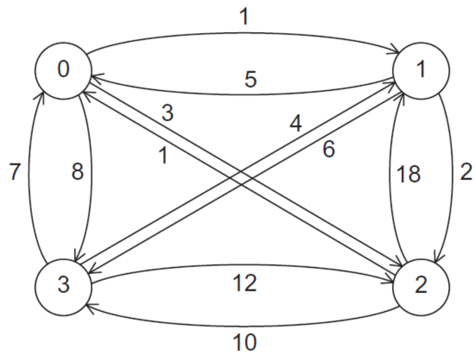
Search Tree for Four-City TSP

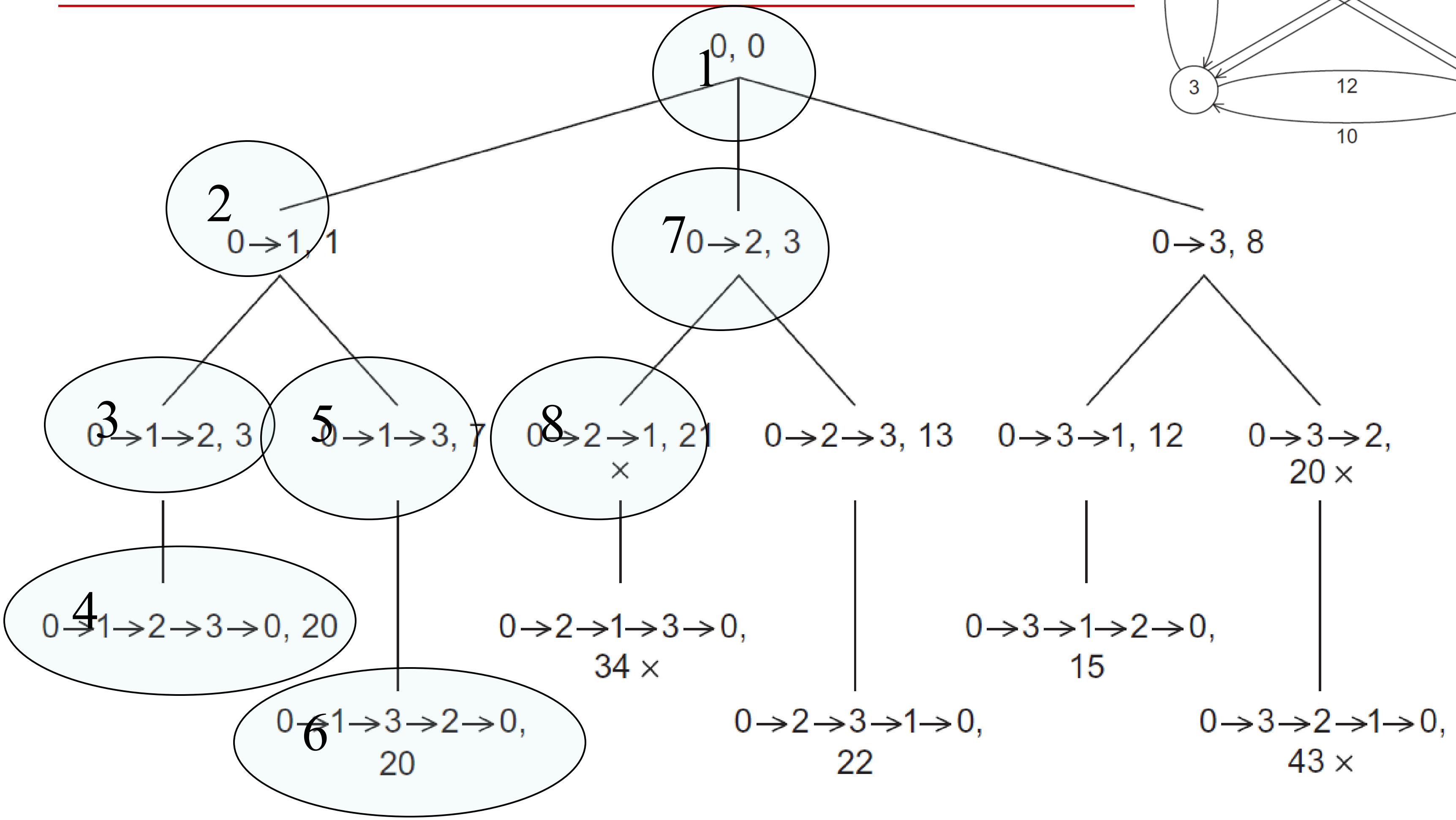
Each tree path represents a partial tour from Hometown source





Depth-first search





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 */
```

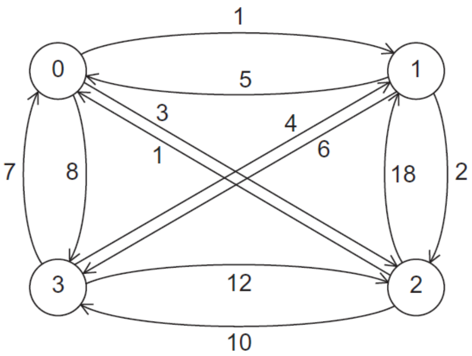
Find a solution. Check if it is the shortest found so far

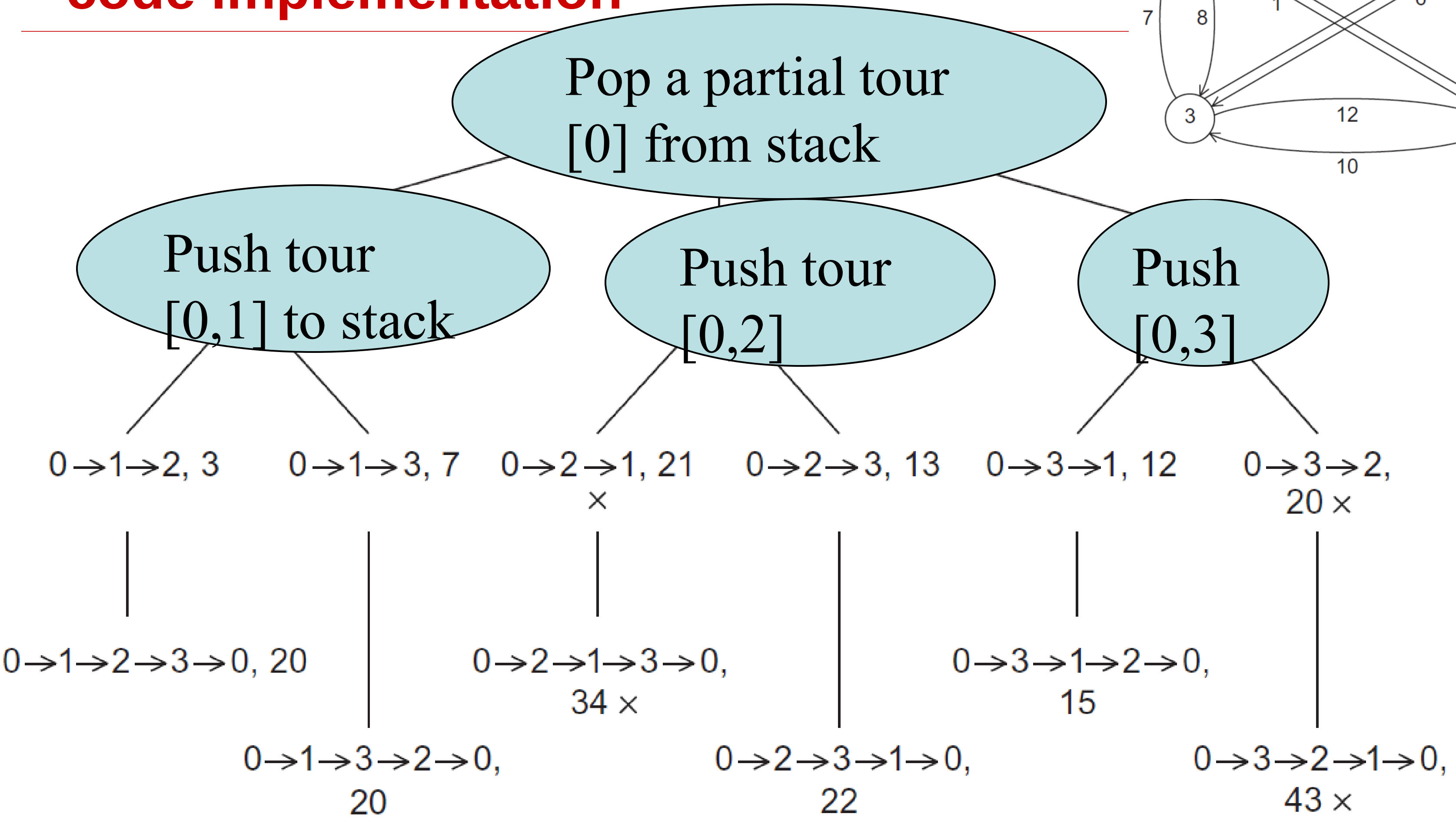
For each neighbor, recursively search

Recursive vs. nonrecursive design

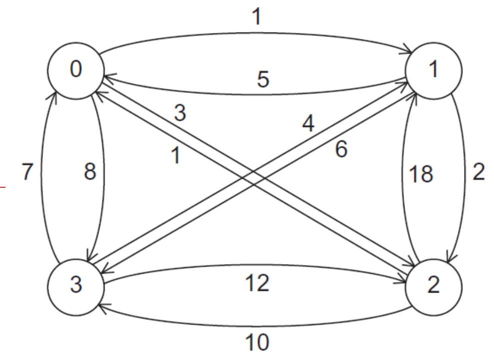
- **Recursion helps understanding of sequential code**
  - Not easy for parallelization.
- **Non-recursive design**
  - Explicit management of stack data structure
  - Loops instead of recursive calls
  - Better for parallelization
    - Expose the traversal of search tree explicitly.
    - Allow scheduling of parallel threads (processes)
- **Two solutions with code sample available from the text book.**
  - Focus on the second solution

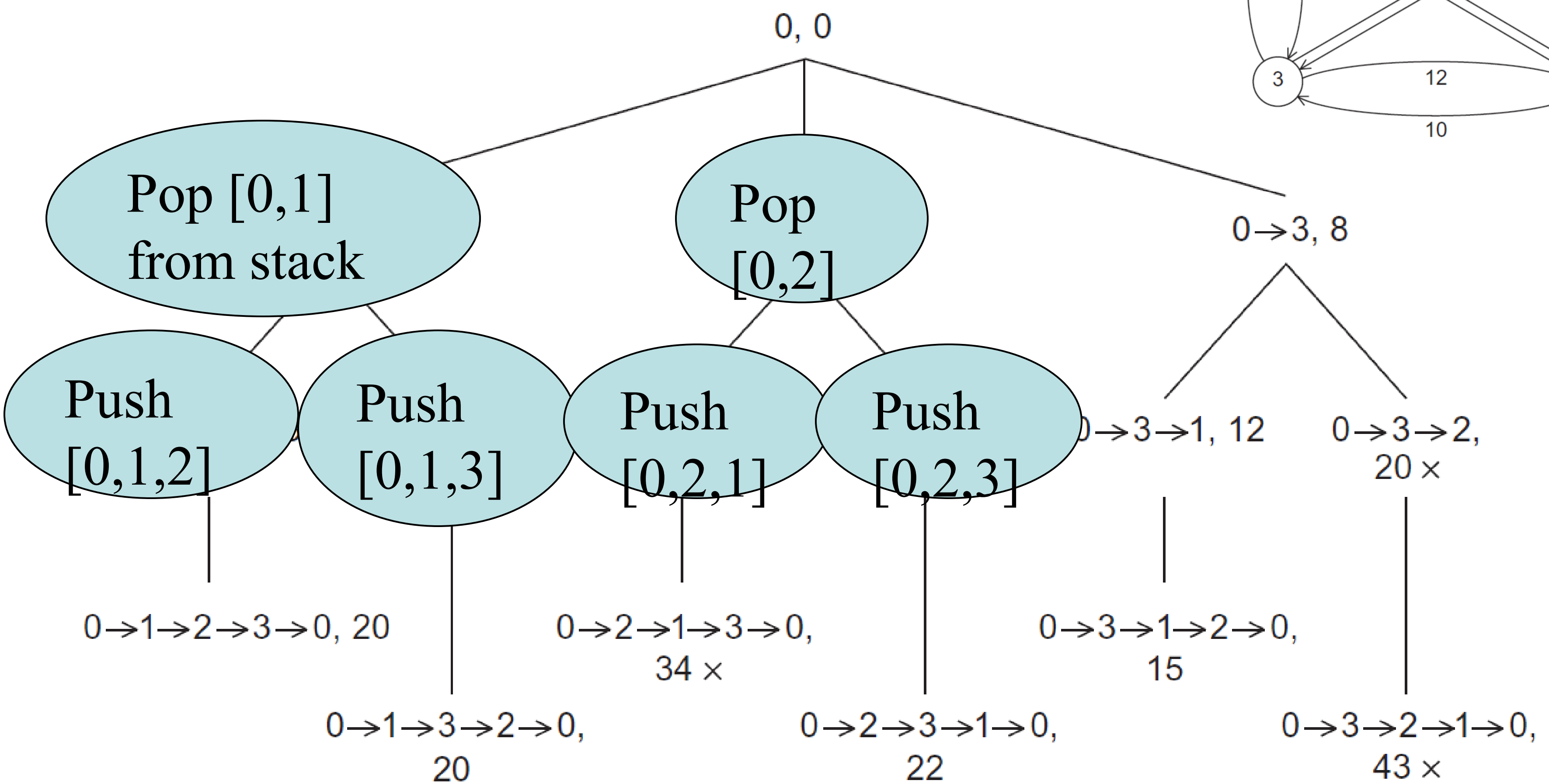
Stack-based nonrecursive code implementation





Stack-based nonrecursive code implementation





Non-recursive solution to TSP (Text book Page 304. Program 6.6)

```
Push_copy(stack, tour);   // T                    he 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);
}
```

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

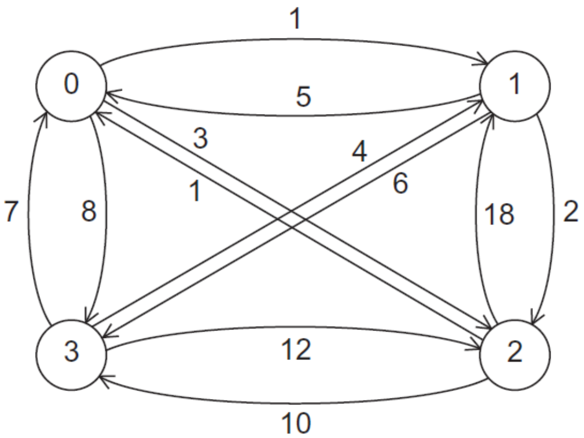
Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

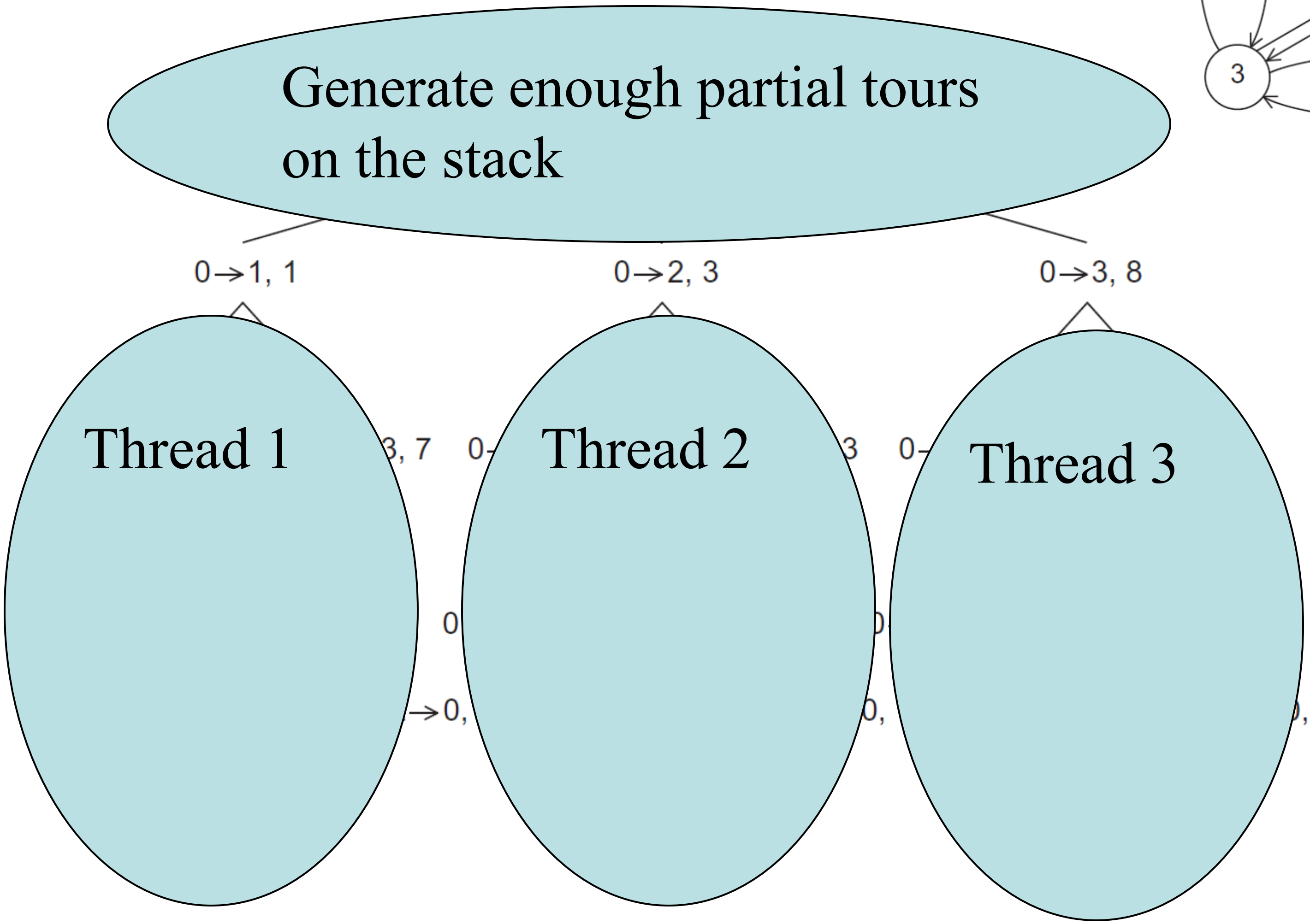
(in seconds)

The digraph contains 15 cities.

All three versions visited approximately 95,000,000 tree nodes.

Parallel processing with threads





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

Global variable or local?

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_st
   if (City_count(curr_t
      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);
}
```

Global?

Update global?

Global or local?

Pthreads code of statically parallelized TSP

Partition workload using local stack

No mutex lock

Mutex lock

Stack grows locally

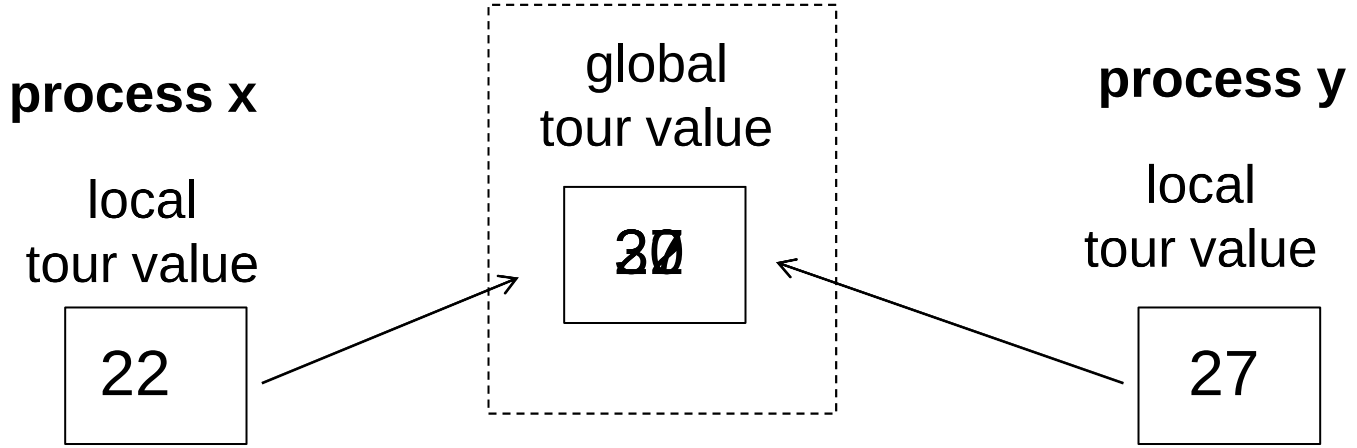
```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_st
   if (City_count(curr_t
      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);
}
```

Code for Update_best_tour()

```
void Update_best_tour(tour_t
  pthread_mutex_lock(&best_
  if (Best_tour(tour)) {
    Copy_tour(tour, best_tour);
    Add_city(best_tour, home_town);
  }
  pthread_mutex_unlock(&best_tour_mutex);
}
```

Double check if it is still the best tour

First scenario



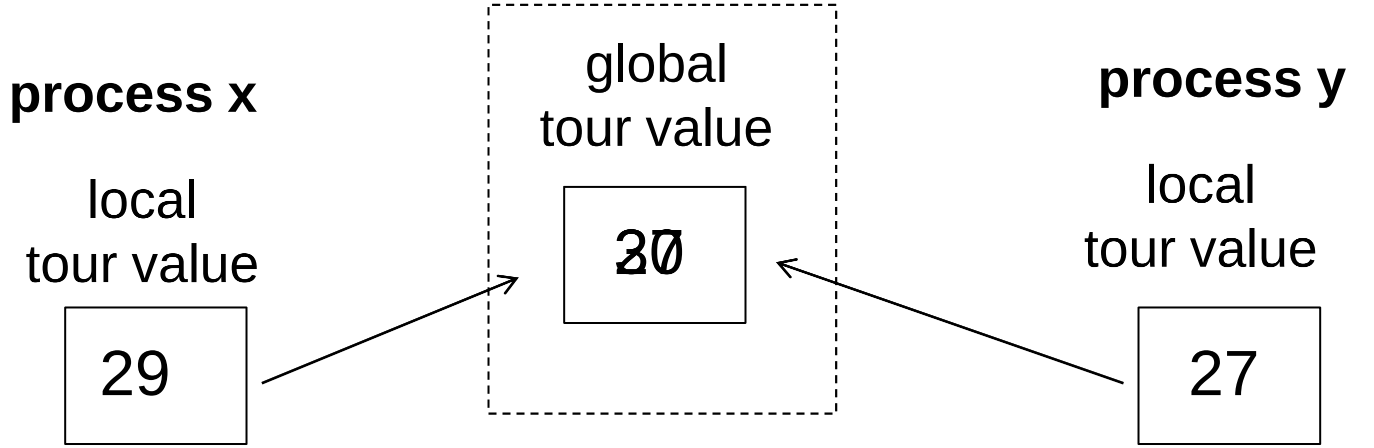
process x

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

process y

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

Second scenario



3. test

6. lock

7. test again

8. unlock

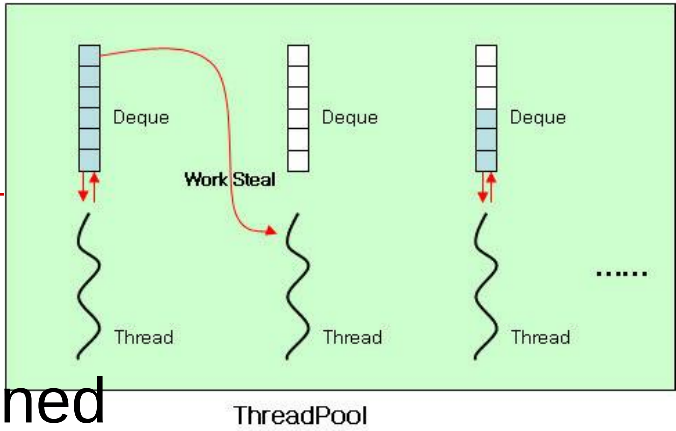
1. test

2. lock

4. update

5. unlock

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?

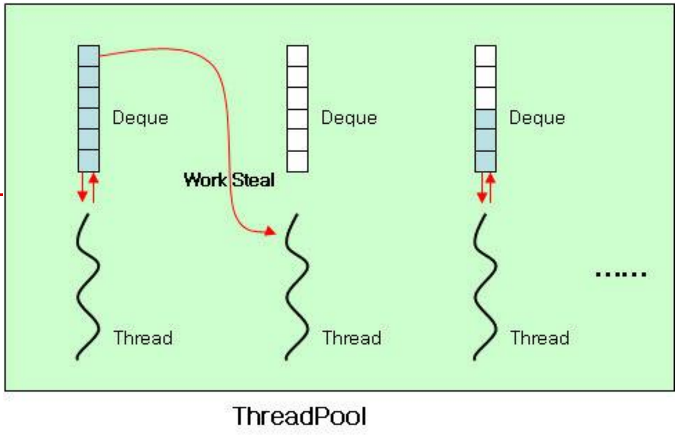
Solutions for the raised issues

- **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 */
```

I have work to do. Split my workload?

All threads are idle. Terminate

Pseudo-Code for Terminated() Function (2)

```
   } 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 */
```

Data structure for termination-related 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
```

Run-times of Pthreads tree search programs

Two 15-city problems.

~95 million tree nodes visited

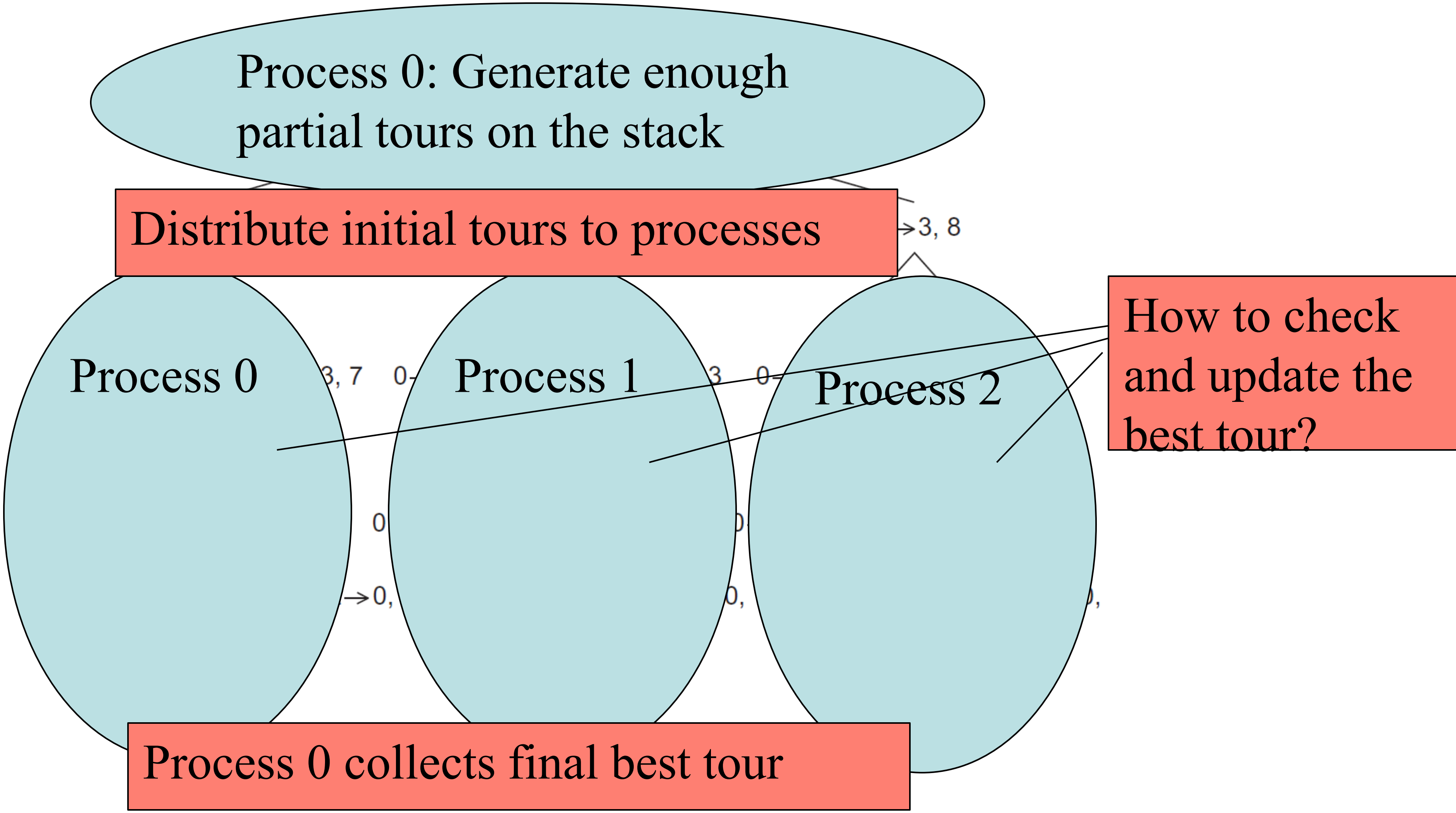
Threads	First Problem				Second Problem			
	Serial	Static	Dynamic		Serial	Static	Dynamic	
1	32.9	32.7	34.7	(0)	26.0	25.8	27.5	(0)
2		27.9	28.9	(7)		25.8	19.2	(6)
4		25.7	25.9	(47)		25.8	9.3	(49)
8		23.8	22.4	(180)		24.0	5.7	(256)

(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



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 user 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

```
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  */)
```

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

```
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()

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;
```

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

```
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      */)
```

Terminated() Function for MPI with Dynamically Partitioned TSP (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 */
```

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

Terminated() Function for MPI with Dynamically Partitioned TSP (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 */
```

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 6.10 Termination Events that Result in an Error

Time	Process 0	Process 1	Process 2
0	Out of Work Notify 1, 2 oow = 1	Out of Work Notify 0, 2 oow = 1	Working oow = 0
1	Send request to 1 oow = 1	Send Request to 2 oow = 1	Recv notify fr 1 oow = 1
2	oow = 1	Recv notify fr 0 oow = 2	Recv request fr 1 oow = 1
3	oow = 1	oow = 2	Send work to 1 oow = 0
4	oow = 1	Recv work fr 2 oow = 1	Recv notify fr 0 oow = 1
5	oow = 1	Notify 0 oow = 1	Working oow = 1
6	oow = 1	Recv request fr 0 oow = 1	Out of work Notify 0, 1 oow = 2
7	Recv notify fr 2 oow = 2	Send work to 0 oow = 0	Send request to 1 oow = 2
8	Recv 1st notify fr 1 oow = 3	Recv notify fr 2 oow = 1	oow = 2
9	Quit	Recv request fr 2 oow = 1	oow = 2

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

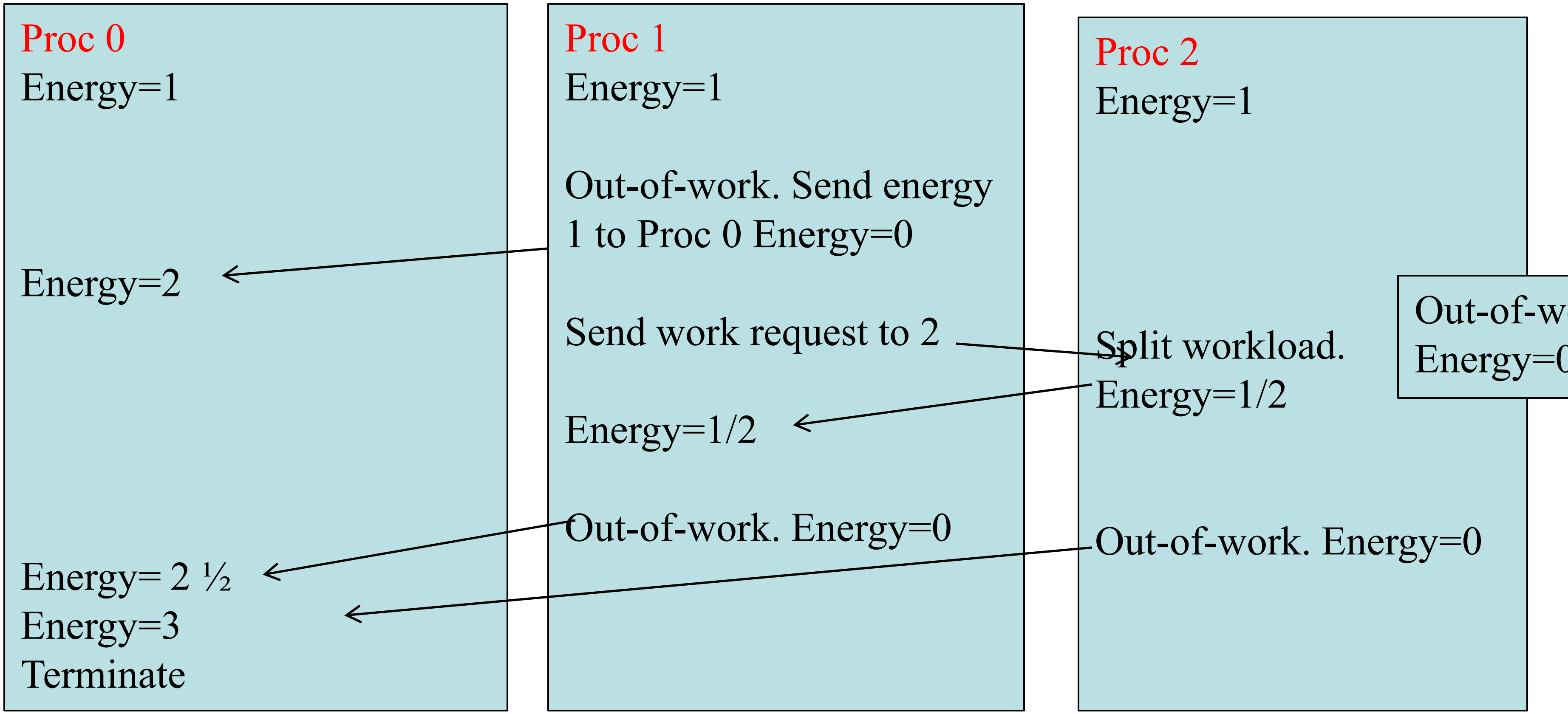
oow=3, forcing Proc 0 quit before receiving work from Proc 1

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



Total energy in all processes =3 during all steps.

Performance of MPI and Pthreads implementations of tree search

Th/Pr	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	Pth	MPI	Pth	MPI	Pth	MPI	Pth	MPI
1	35.8	40.9	41.9 (0)	56.5 (0)	27.4	31.5	32.3 (0)	43.8 (0)
2	29.9	34.9	34.3 (9)	55.6 (5)	27.4	31.5	22.0 (8)	37.4 (9)
4	27.2	31.7	30.2 (55)	52.6 (85)	27.4	31.5	10.7 (44)	21.8 (76)
8		35.7		45.5 (165)		35.7		16.5 (161)
16		20.1		10.5 (441)		17.8		0.1 (173)

(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.