InfoLab

IC621: Distributed and Parallel Computing

Lecture 09 : Parallel Program Development

Min-Soo Kim

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

InfoLab

TREE SEARCH

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Traveling salesperson problem(TSP)

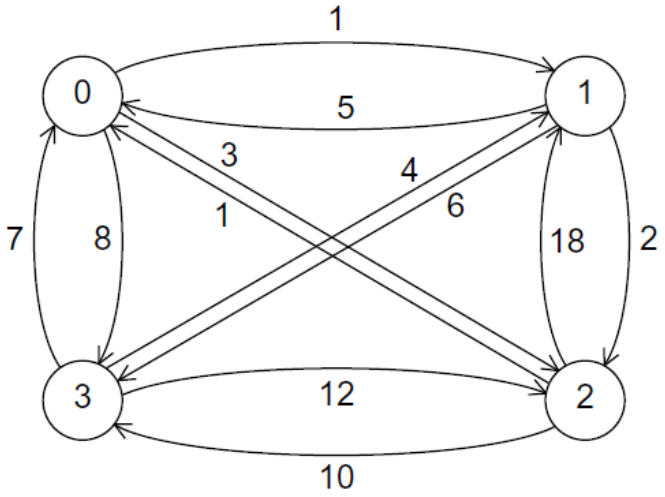
- **Finding a minimum-cost tour**
  - a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities
  - her should visit each city once, returning to her hometown, and she must do this with the least possible cost
- **TSP is a NP-complete problem**
  - no algorithm known for solving it that, in all cases, is significantly better than exhaustive search
  - exhaustive search : examining all possible solutions to the problem and choosing the best
  - # of possible tours: (n-1)!
    - four cities : 6
    - five cities : 4 * 6 = 24
    - six cities : 5 * 24 = 120
    - 100 cities : 99! > # of atoms in the universe!

Solution for TSP

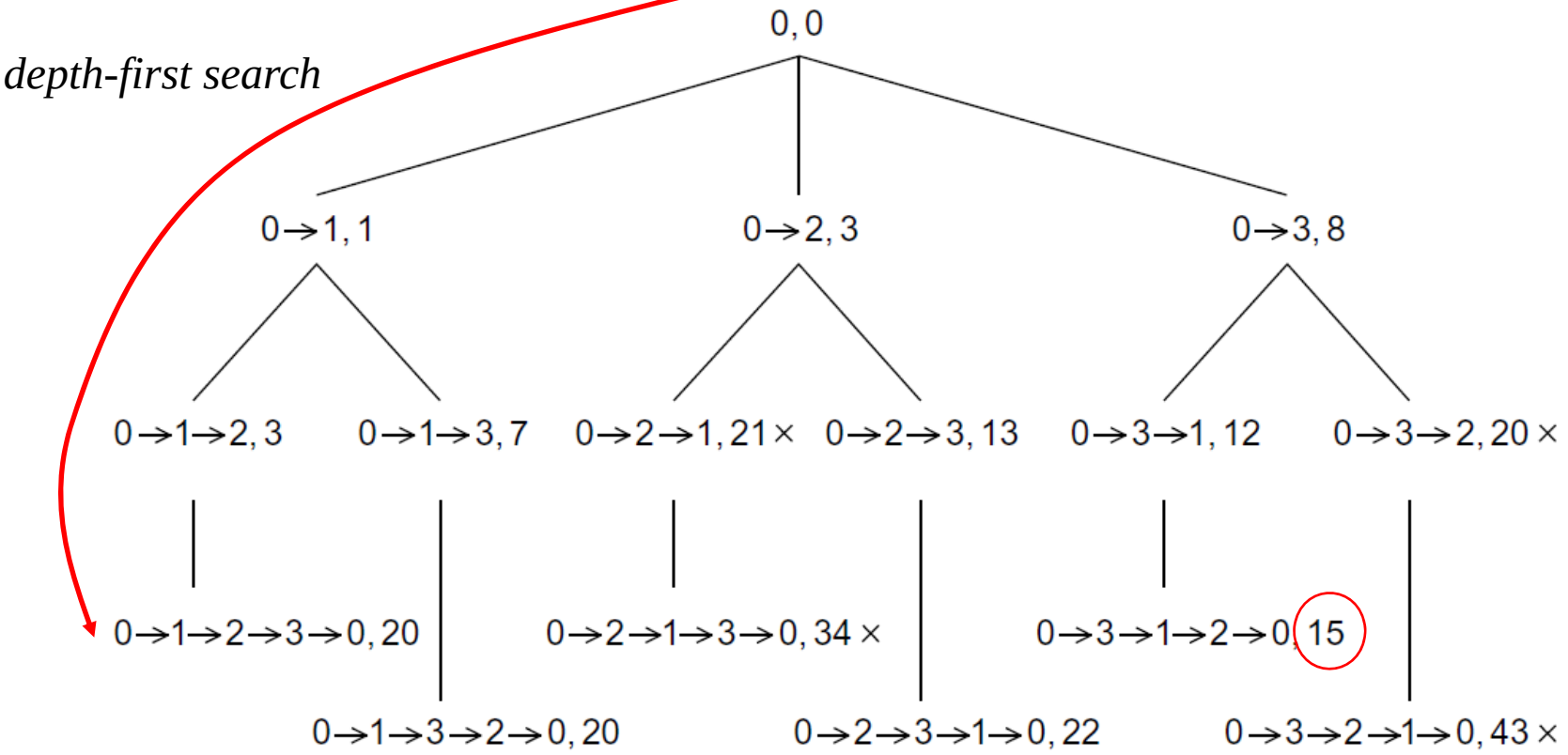
- **Build a tree**
  - the leaves of the tree correspond to tours
  - the other tree nodes correspond to “partial” tours — routes that have visited some, but not all, of the cities
  - each node of the tree has an associated cost, i.e., the cost of the partial tour

- **Search tree**
  - we can use costs to eliminate some nodes of the tree
  - we can keep track of the cost of the best tour so far
  - if we find a partial tour or node of the tree that couldn’t possibly lead to a less expensive complete tour, we don’t need to search the children of that node

Example



depth-first search



Global variables

- `n`: the total number of cities
- `digraph`: a data structure representing the input digraph
- `hometown`: a data structure representing vertex or city 0
- `best_tour`: a data structure representing the best tour so far

Recursive depth-first search

- **Functions**
  - `City_count` : checks the partial tour `tour` to see if there are n cities on the partial tour
  - `Feasible` : checks to see if the city or vertex has already been visited, and, if not, whether it can possibly lead to a least-cost tour

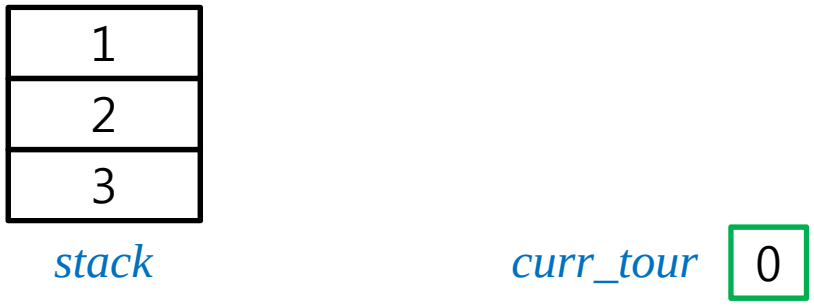
```
1   void Depth_first_search(tour_t tour) {
2      city_t city;
3
4      if (City_count(tour) == n) {
5         if (Best_tour(tour))
6            Update_best_tour(tour);
7      } else {
8         for each neighboring city
9            if (Feasible(tour, city)) {
10              Add_city(tour, city);
11              Depth_first_search(tour);
12              Remove_last_city(tour, city);
13           }
14     }
15  }  /* Depth_first_search */
```

Nonrecursive depth-first search

- **Drawbacks of recursive version**
  - recursion can be slow, since function calls are expensive
  - only the current tree node is accessible, at any given instant of time
  - it may be difficult to parallelize tree search by dividing tree nodes among the threads or processes

- **Writing a nonrecursive version**
  - recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack
  - we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree
  - when we need to go back up the tree, we can pop the stack

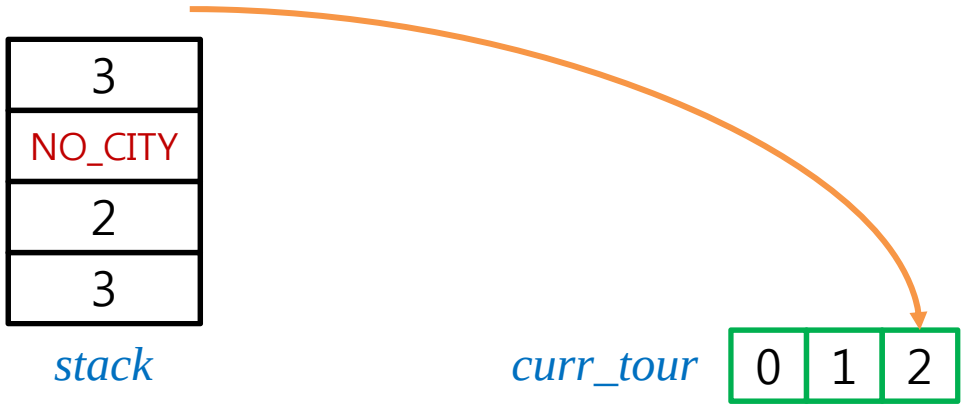
```
1   for (city = n-1; city >= 1; city--)
2      Push(stack, city);
3   while (!Empty(stack)) {
4      city = Pop(stack);
5      if (city == NO_CITY) // End of child list, back up
6         Remove_last_city(curr_tour);
7      else {
8         Add_city(curr_tour, city);
9         if (City_count(curr_tour) == n) {
10           if (Best_tour(curr_tour))
11              Update_best_tour(curr_tour);
12           Remove_last_city(curr_tour);
13        } else {
14           Push(stack, NO_CITY);
15           for (nbr = n-1; nbr >= 1; nbr--)
16              if (Feasible(curr_tour, nbr))
17                 Push(stack, nbr);
18        }
19     }  /* if Feasible */
20  }  /* while !Empty */
```

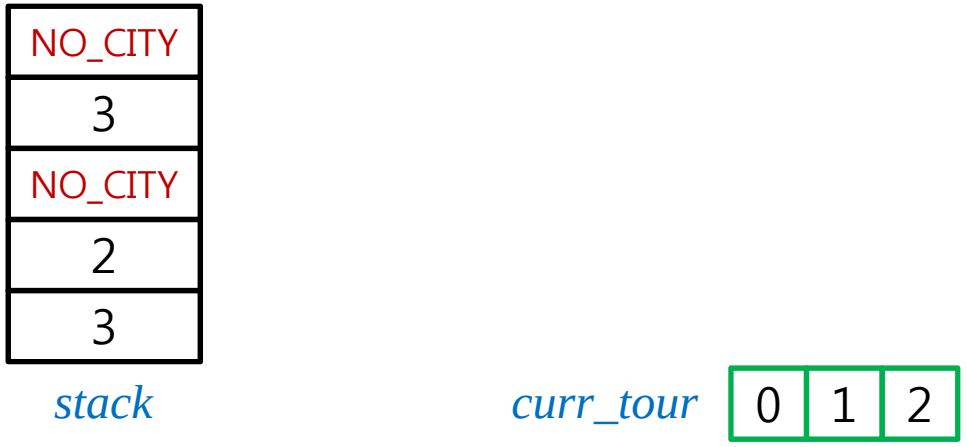

1
2
3
stack
curr_tour
0

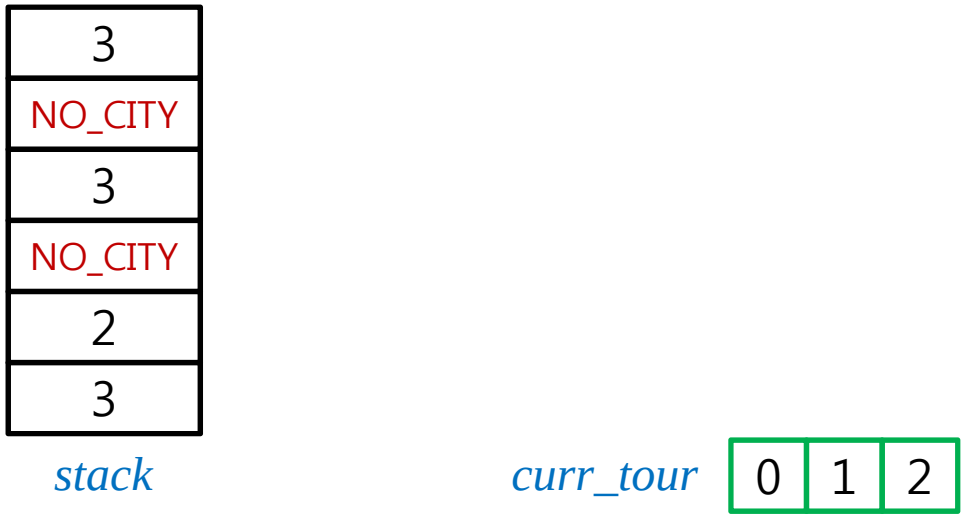

2
3
stack
curr_tour
0
1

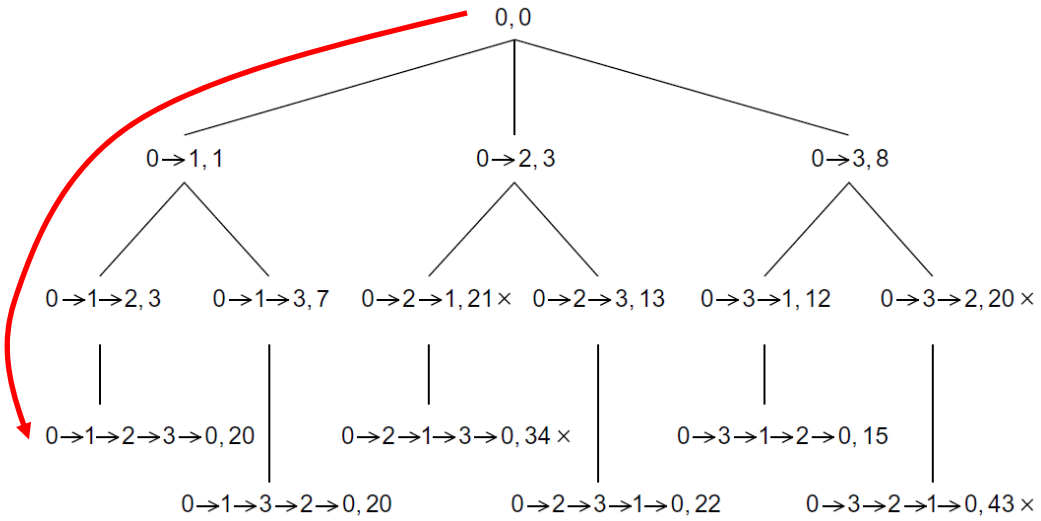

NO_CITY
2
3
stack
curr_tour
0
1


2
3
NO_CITY
2
3
stack
curr_tour
0
1


3
NO_CITY
2
3
stack
curr_tour
0
1
2


NO_CITY
3
NO_CITY
2
3
stack
curr_tour
0
1
2


3
NO_CITY
3
NO_CITY
2
3
stack
curr_tour
0
1
2



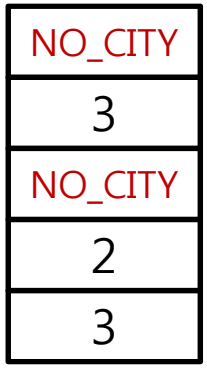
NO_CITY
3
NO_CITY
2
3

stack

curr_tour

0	1	2	3

Update_best_tour( )

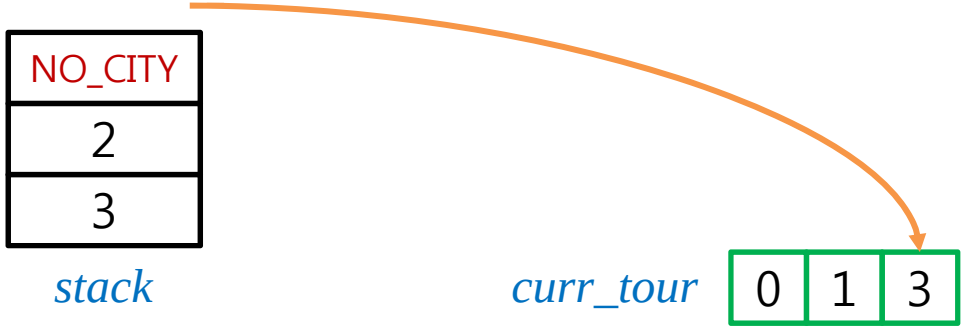


stack

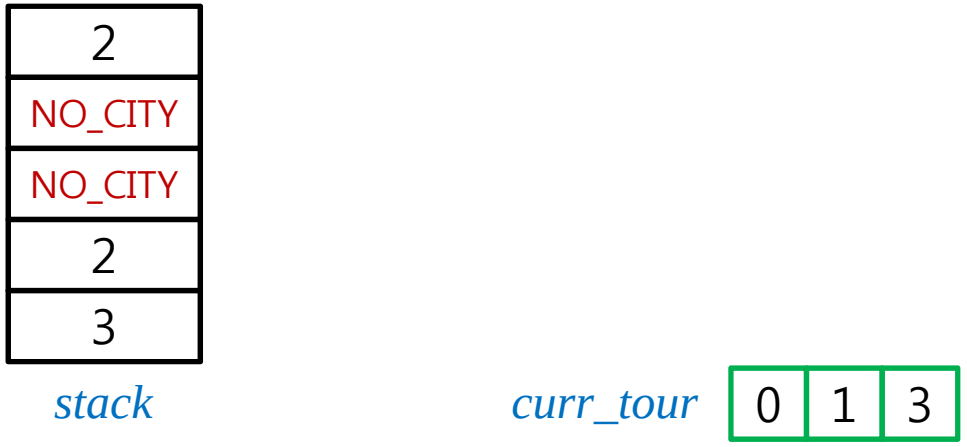
curr_tour

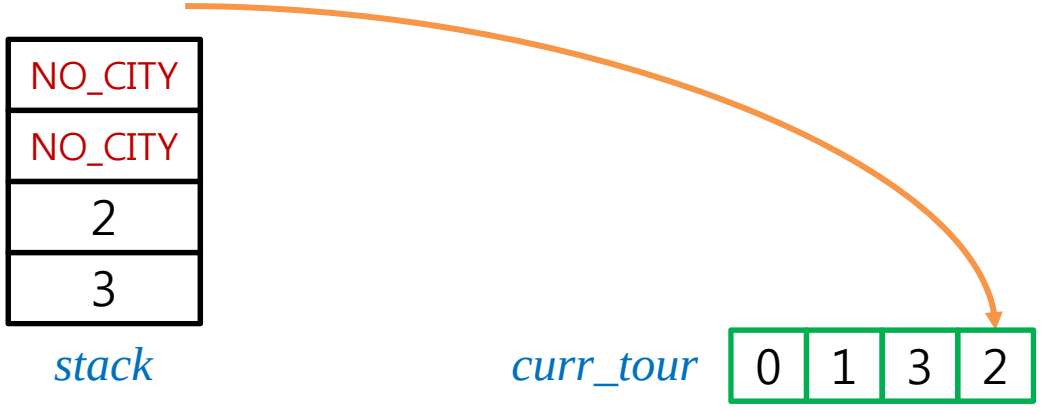
0	1	2

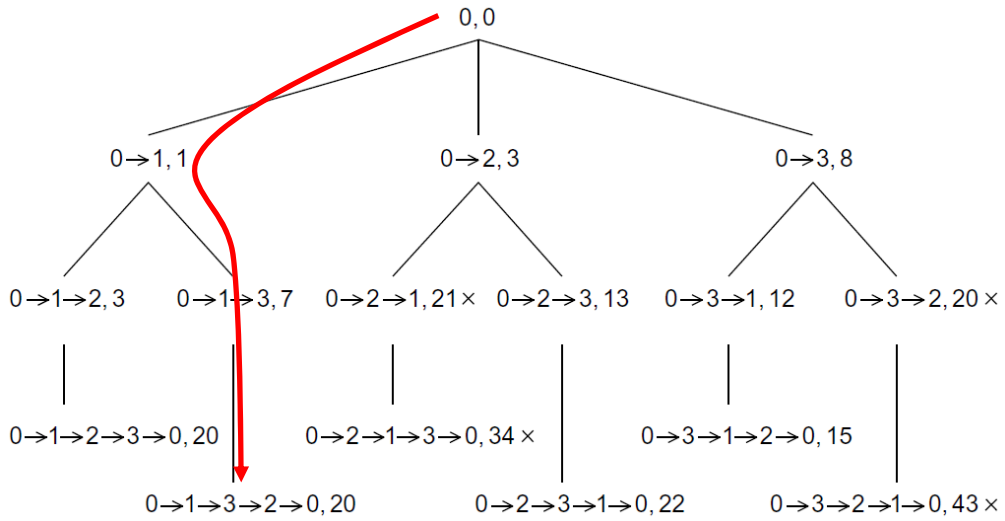

3
NO_CITY
2
3
stack
curr_tour
0
1

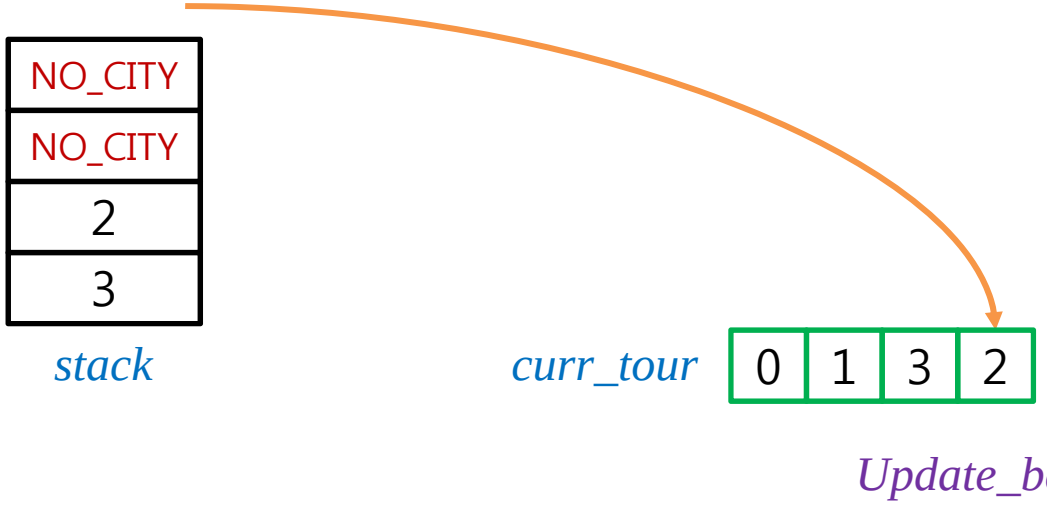

NO_CITY
2
3
stack
curr_tour
0
1
3


NO_CITY
NO_CITY
2
3
stack
curr_tour
0
1
3


2
NO_CITY
NO_CITY
2
3
stack
curr_tour
0
1
3

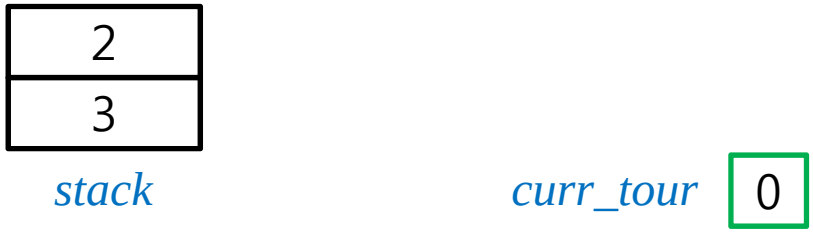

NO_CITY
NO_CITY
2
3
stack
curr_tour
0
1
3
2

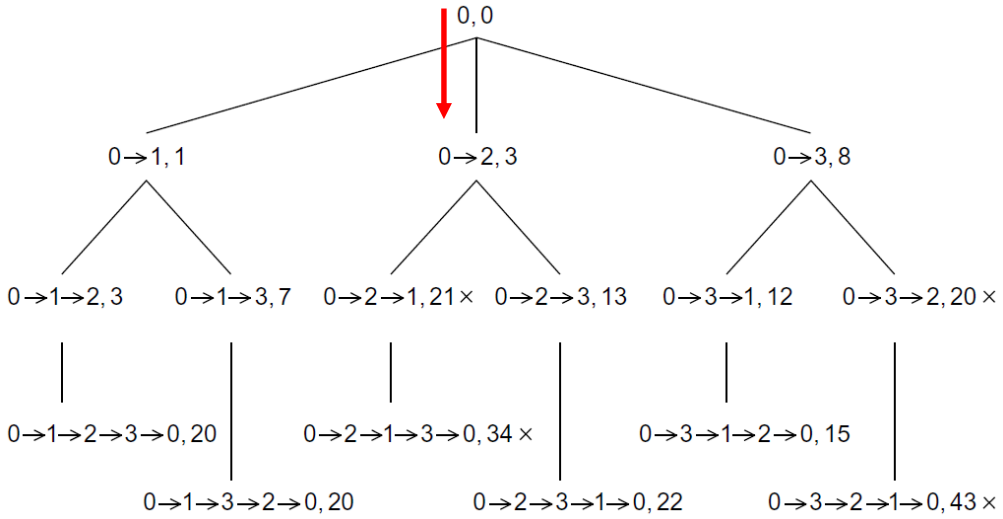

0, 0
0→1, 1
0→2, 3
0→3, 8
0→1→2, 3
0→1→3, 7
0→2→1, 21×
0→2→3, 13
0→3→1, 12
0→3→2, 20×
0→1→2→3→0, 20
0→2→1→3→0, 34×
0→3→1→2→0, 15
0→1→3→2→0, 20
0→2→3→1→0, 22
0→3→2→1→0, 43×


NO_CITY
NO_CITY
2
3
stack
curr_tour
0
1
3
2
Update_best_tour( )


NO_CITY
NO_CITY
2
3
stack
curr_tour
0
1
3


NO_CITY
2
3
stack
curr_tour
0
1


2
3
stack
curr_tour
0


0, 0
0→1, 1
0→2, 3
0→3, 8
0→1→2, 3
0→1→3, 7
0→2→1, 21×
0→2→3, 13
0→3→1, 12
0→3→2, 20×
0→1→2→3→0, 20
0→2→1→3→0, 34×
0→3→1→2→0, 15
0→1→3→2→0, 20
0→2→3→1→0, 22
0→3→2→1→0, 43×


3
stack
curr_tour
0
2

- **Lines 1-2**
  - by reversing the order, we can insure that the cities are visited in the same order as the recursive function

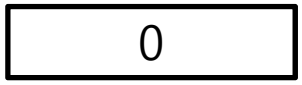
- **Lines 5 and 14**
  - the constant `NO_CITY` is used so that we can tell when we've visited all of the children of a tree node
  - if we didn't use it, we wouldn't be able to tell when to back up in the tree

Alternative to the iterative version

- **Use partial tours as stack records**
- **Closer to the recursive function, but result in a slower version**
  - it's necessary for the function that pushes onto the stack to create a copy of the tour before actually pushing it on to the stack
  - allocating storage for a new tour and copying the existing tour is time-consuming
- **Virtue : the stack is more or less independent of the other data structures**
  - since entire tours are stored, multiple threads or processes can "help themselves" to tours
  - basis of parallel TSP algorithm

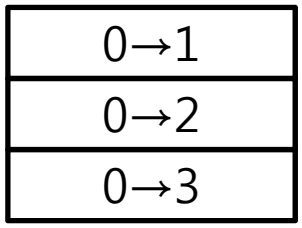
Basis of parallel versions

```
1  Push_copy(stack, tour);  // Tour that visits only the hometown
2  while (!Empty(stack)) {
3     curr_tour = Pop(stack);
4     if (City_count(curr_tour) == n) {
5        if (Best_tour(curr_tour))
6           Update_best_tour(curr_tour);
7     } else {
8        for (nbr = n-1; nbr >= 1; nbr--)
9           if (Feasible(curr_tour, nbr)) {
10             Add_city(curr_tour, nbr);
11             Push_copy(stack, curr_tour);
12             Remove_last_city(curr_tour);
13          }
14    }
15    Free_tour(curr_tour);
16 }
```



stack *curr_tour*

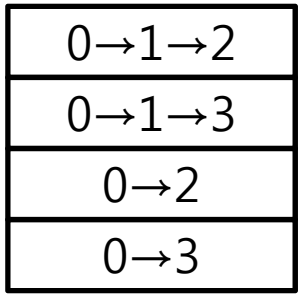

stack
curr_tour
0



stack

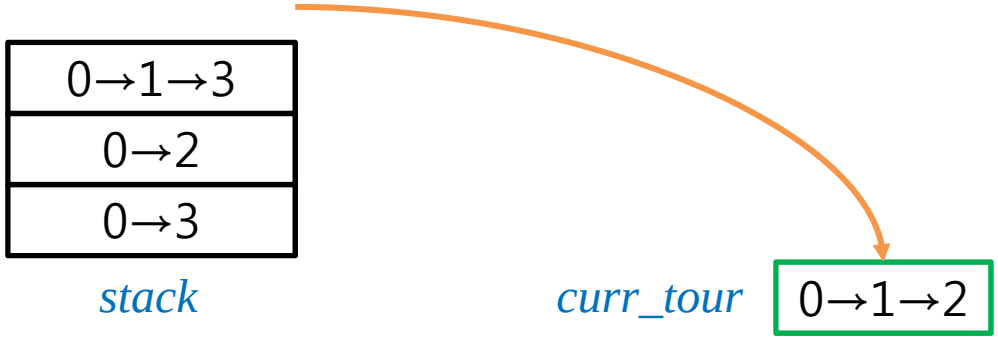
curr_tour

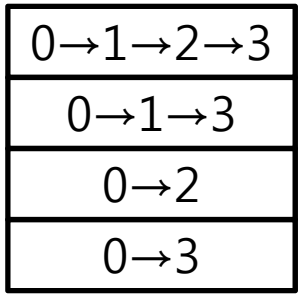

0→2
0→3
stack
curr_tour
0→1



stack

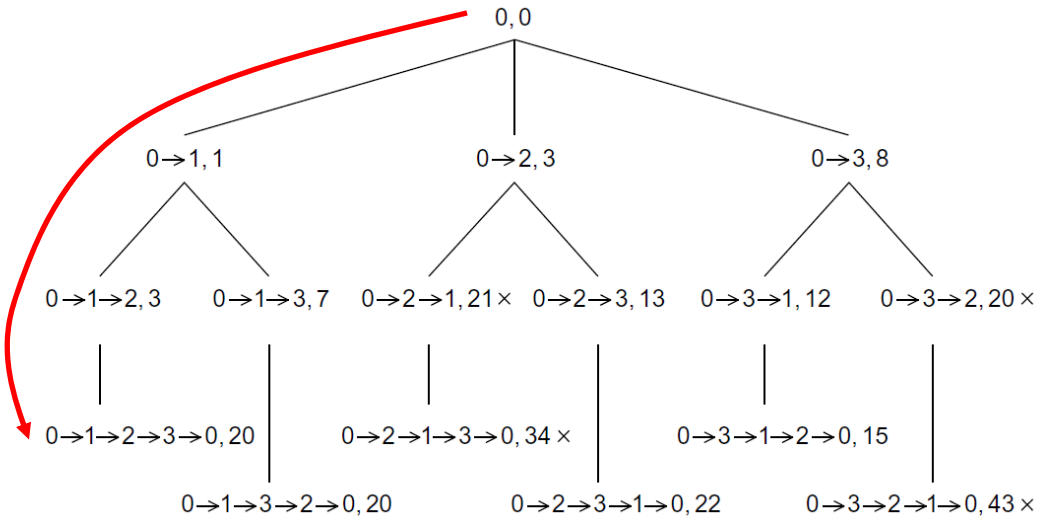
curr_tour


0→1→3
0→2
0→3
stack
curr_tour
0→1→2



stack

curr_tour



0→1→3
0→2
0→3

stack

curr_tour 0→1→2→3

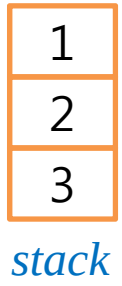
Update_best_tour( )

Data structures for the serial implementations

- **Stack struct with three members**
  - the array storing the cities
  - the number of cities
  - the cost of the partial tour

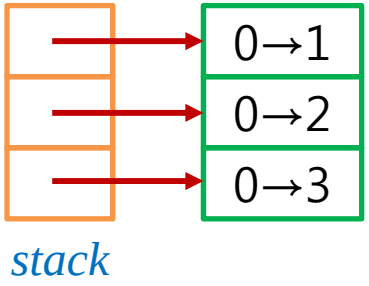
first iterative version

```
void Push(my_stack_t stack, int city) {
   int loc = stack->list_sz;
   stack->list[loc] = city;
   stack->list_sz++;
}  /* Push */
```



second iterative version

```
void Push_copy(my_stack_t stack, tour_t tour) {
   int loc = stack->list_sz;
   tour_t tmp = Alloc_tour();
   Copy_tour(tour, tmp);
   stack->list[loc] = tmp;
   stack->list_sz++;
}  /* Push */
```



Performance of the serial implementations

- **Input digraph : 15 vertices (including the hometown)**
  - all three algorithms visited approximately 95,000,000 tree nodes
- **Results**
  - the first iterative solution eliminates some of the overhead due to repeated function calls
  - the second iterative solution is slower because of the repeated copying of tour data structures

Table 6.7 Run-Times of the Three Serial Implementations of Tree Search (times in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

Parallelizing tree search

- **Mapping tasks to each thread or process**
  - many possible algorithms for identifying which subtrees we assign to the processes or threads
  - problem of depth-first search : it may not true that a lower-level subtree has less work than a higher-level subtree
  - solution: probably get better load balance by using something like breadth-first search to identify the subtrees

- **Doing breadth-first search**
  - until we reach a level of the tree that has at least `thread_count` or `comm_sz` nodes
    - then, we can divide the nodes at this level among the threads or processes

The `best_tour` data structure

- on a shared-memory system, the best tour data structure can be shared
  - updates to the best tour will cause a race condition
  - we'll need some sort of locking to prevent errors

- on distributed-memory system, there are a couple of choices to make about the best tour
  - processes operate independently of each other until they have completed searching their subtrees
  - each process stores its own local best tour
  - this local best tour is used by the process in `Feasible` and updated by the process each time it calls `Update_best_tour`
  - when all the processes have finished searching, they can perform a global reduction to find the tour with the global least cost
  - **problem** : a process might spend most or all of its time searching through partial tours that couldn't possibly lead to a global best tour

Problem of load imbalance

- even using breadth-first search does not ensures that all of subtrees have approximately the same number of nodes
  - one process or thread may have a subtree consisting of very expensive tours
  - as a consequence, it won't need to search very deeply into its assigned subtree
  - with the static mapping, one thread or process will simply have to wait until the other threads/processes are done

Dynamic mapping of tasks

- if one thread/process runs out of useful work, it can obtain additional work from another thread/process
  - each stack record contains a partial tour
  - a thread or process can give additional work to another thread/process by dividing the contents of its stack
- alternative: using a shared stack (only for the shared memory)
  - there would be a tremendous amount of contention for the shared stack
  - the performance of the program would probably be worse than a serial program

A static parallelization of tree search using pthreads

- **Static parallelization**
  - a single thread uses breadth-first search to generate enough partial tours so that each thread gets at least one partial tour
    - need to generate at least `thread_count` partial tours to distribute among the threads
  - each thread takes its partial tours and runs iterative tree search on them

- **No race condition or contention for digraph**
  - most of the function calls (e.g., `Best_tour`, `Feasible`, `Add_city`) need to access the adjacency matrix representing the `digraph`
  - so, all the threads will need to access the `digraph`
  - since these are only read accesses, this won't result in a race condition or contention among the threads

Four potential differences between serial and parallel

- the use of `my_stack` instead of `stack`
  - since each thread has its own, private stack, we use `my_stack` as the identifier for the stack object instead of `stack`
- initialization of the stack
- implementation of the `Best_tour` function
- implementation of the `Update_best_tour` function

```
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);
}
```

Implementation of `Best_tour`

- it only reads the global best cost
- if we prefer to get a new value, we need to use locking
- for performance, a best tour cost of out-of-date is probably better

Implementation of `Update_best_tour`

- we can protect the body of `Update_best_tour` with a mutex
- however, this isn't enough, since another thread may have updated the best tour cost
  - between the time a thread completes the test in `Best_tour` and the time it obtains the lock in `Update_best_tour`
- we need double check though most of the time `Best_tour` will return false

```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
    again */
if (Best_tour(tour))
   Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

A dynamic parallelization of tree search using pthreads

Motivation

- the initial distribution of subtrees might not do a good job of distributing the work among the threads
- but, the static parallelization provides no means of redistributing work
  - the threads with “small” subtrees will finish early
  - the threads with large subtrees will continue to work
- we may need to dynamically redistribute the work as the computation proceeds

Approach

- replace the test `!Empty(my_stack)`such that the thread waits to see if another thread can provide more work
  - instead of immediately exiting the while loop
- the thread having at least two tours in its stack can “split” its stack and provide work for one of the threads waiting for more work

Implementation

- when a thread runs out of work, it can call `pthread_cond_wait` and go to sleep
- when another thread with work finds a thread waiting for work, it can call `pthread_cond_signal` after splitting its stack
- when a thread is awakened, it can take one of the halves of the split stack and return to work
- this idea can be extended to handle termination

`Terminated` function

- used instead of `Empty` for the while loop
- a thread checks (in Lines 1-2)
  - that it has at least two tours in its stack
  - that there are threads waiting
  - whether the `new_stack` (global) variable is NULL

Terminated

```
1   if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2          new_stack == NULL) {
3      lock term_mutex;
4      if (threads_in_cond_wait > 0 && new_stack == NULL) {
5         Split my_stack creating new_stack;
6         pthread_cond_signal(&term_cond_var);
7      }
8      unlock term_mutex;
9      return 0;  /* Terminated = false; don't quit */
10  } else if (!Empty(my_stack)) /* Keep working */
11     return 0;  /* Terminated = false; don't quit */
```

double check

Two possibilities in Line 14

- we're the last thread to enter the termination sequence, that is, `threads_in_cond_wait == thread_count - 1`
  - tree search should terminate
  - we signal all the other threads by calling `pthread_cond_broadcast` and returning true
- other threads are still working
  - in Line 22, it's possible that a thread could be awakened by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`
  - so, as usual, we put the call to `pthread_cond_wait` in a while loop
    - which will immediately call `pthread_cond_wait` again if some other event (return value not 0) awakens the thread
  - once we've been awakened, there are also two cases to consider
    - `threads_in_cond_wait < thread_count` : some other thread has split its stack and created more work
    - `threads_in_cond_wait == thread_count` : there's no work left

```
12  } else { /* My stack is empty */
13     lock term_mutex;
14     if (threads_in_cond_wait == thread_count-1)
15                                          /* Last thread running */
16        threads_in_cond_wait++;
17        pthread_cond_broadcast(&term_cond_var);
18        unlock term_mutex;
19        return 1;  /* Terminated = true; quit */
20     } else { /* Other threads still working, wait for work */
21        threads_in_cond_wait++;
22        while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
23        /* We've been awakened */
24        if (threads_in_cond_wait < thread_count) { /* We got work */
25           my_stack = new_stack;
26           new_stack = NULL;
27           threads_in_cond_wait--;
28           unlock term_mutex;
29           return 0;  /* Terminated = false */
30        } else {  /* All threads done */
31           unlock term_mutex;
32           return 1;  /* Terminated = true; quit */
33        }
34     }  /* else wait for work */
35  }  /* else my_stack is empty */
```

- **It convenient to group the termination variables together into a single struct**

```
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
```

Splitting the stack

- we want that the amount of work in the new stack is roughly the same as the amount remaining in the original stack
  - since our goal is to balance the load among the threads
- we'll never be able to guarantee an equal division of work
- we just can try splitting the stack by assigning the tours on the stack on the basis of their numbers of edges
  - on average two partial tours with the same number of cities are equally likely to lead to a "good" tour

Implementation of splitting

- the tours on the stack have an increasing number of edges
- we start at the bottom of the stack, and alternately leave partial tours on the old stack and push partial tours onto the new stack
  - so, tour 0 will stay on the old stack, tour 1 will go to the new stack, tour 2 will stay on the old stack, and so on

Evaluating the Pthreads tree-search programs

- **Two fifteen-city problems**
  - the numbers in parentheses : the total number of times the stacks were split
  - different problems can result in radically different behaviors
  - the dynamic program is more scalable than the static program

Table 6.8 Run-Times of Pthreads Tree-Search Programs (times in seconds)

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)

Parallelizing the tree-search programs using OpenMP

- **OpenMP version is similar with static implementation**
- **Point 1 (should be considered in OpenMP)**
  - the following test in Pthreads

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

  - can be replaced by the OpenMP directive

```
#  pragma omp single
```

    - insure that the following structured block of code will be executed by one thread in the team
    - insure that the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is finished
  - also be replaced by the OpenMP directive when `whatever` is 0

```
#  pragma omp master
```

    - but, the `master` directive doesn't put an implicit barrier at the end of the block
    - so, it may be necessary to add a `barrier` directive after the block

Point 2

- Pthreads mutex that protects the best tour can be replaced by a single critical directive placed either
  - inside the `Update_best_tour` function or
  - immediately before the call to `Update_best_tour`
- That is the only potential source of a race condition after the distribution of the initial tours
  - so, the simple critical directive won't cause a thread to block unnecessarily

Point 3

- dynamically load-balanced Pthreads implementation depends heavily on Pthreads condition variables
- but, OpenMP doesn't provide a comparable object
- instead, we may use the following functions of OpenMP

```
void omp_set_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*  lock_p   /* in/out */);
int omp_test_lock(omp_lock_t*    lock_p   /* in/out */);
```

- we need to emulate the functionality of the Pthreads function calls

```
pthread_cond_signal(&term_cond_var);
pthread_cond_broadcast(&term_cond_var);
pthread_cond_wait(&term_cond_var, &term_mutex);
```

Two events for `pthread_cond_wait`

- another thread has split its stack and created work for the waiting thread
- all of the threads have run out of work

Solution

- the simplest solution to emulating a condition wait in OpenMP is to use busy-waiting
- we can use two different variables in the busy-wait loop (due to two events)

```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;   /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

- if `awakened_thread` has the value of some thread's rank, that thread will exit immediately from the while, but there may be no work available
- if `work_remains` is initialized to 0, all the threads will exit the while loop immediately and quit

- **We should relinquish the lock used in the `Terminated` function before starting the while loop**
  - Pthreads relinquishes the mutex associated with the condition variable
    - so that another thread can also enter the condition wait or signal the waiting thread
  - Pthreads reacquires the mutex associated with the condition variable when returned from a 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);
```

Emulating the condition broadcast

- when a thread determines that there's no work left (Line 14), the condition broadcast (Line 17) can be replaced with the assignment

```
work_remains = 0;   /* Assign false to work_remains */
```

Emulating the condition signal

- the thread splitting its stack needs to choose one of the sleeping threads and set the variable `awakened_thread` to the rank
- thus, we need to keep a list of the ranks of the sleeping threads
- a simple way to do this is to use a shared queue of thread ranks
- when a thread runs out of work, it enqueues its rank before entering the busy-wait loop
- when a thread splits its stack, it can choose the thread to awaken by dequeuing the queue of waiting threads
- the awakened thread needs to reset `awakened_thread` to `-1` before it returns from its call to the `Terminated` function

```
got_lock = omp_test_lock(&term_lock);
if (got_lock != 0) {
   if (waiting_threads > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      awakened_thread = Dequeue(term_queue);
   }
   omp_unset_lock(&term_lock);
}
```

Performance of the OpenMP implementations

- **Results**
  - for the most part, the OpenMP implementations are comparable to the Pthreads implementations
    - busy-waiting doesn't degrade overall performance unless we were using more threads than cores
  - red circles could be due to the nondeterminism of the programs

Table 6.9 Performance of OpenMP and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						Second Problem					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th	*OMP*	*Pth*	*OMP*		*Pth*		*OMP*	*Pth*	*OMP*		*Pth*	
1	32.5	32.7	33.7	(0)	34.7	(0)	25.6	25.8	26.6	(0)	27.5	(0)
2	27.7	27.9	28.0	(6)	28.9	(7)	25.6	25.8	18.8	(9)	19.2	(6)
4	25.4	25.7	33.1	(75)	25.9	(47)	25.6	25.8	9.8	(52)	9.3	(49)
8	28.0	23.8	19.2	(134)	22.4	(180)	23.8	24.0	6.3	(163)	5.7	(256)

Implementation of tree search using MPI and static partitioning

- **Broadcasting the adjacency matrix**
  - adjacency matrix is going to be relatively small
  - even if we have 100 cities, the matrix will require less than 80,000 bytes
  - process 0 broadcasts it to all the processes

- **Principal differences**
  - partitioning the tree
  - checking and updating the best tour
  - after the search has terminated, making sure that process 0 has a copy of the best tour for output

Partitioning the tree

- **MPI process 0 can also generate a list of `comm_sz` partial tours**
  - since memory isn't shared, it will need to send the initial partial tours to the appropriate process
  - we may not be able to use use `MPI_Scatter` since the number of initial partial tours is not evenly divisible by `comm_sz`
    - process 0 won't be sending the same number of tours to each process
    - `MPI_Scatter` requires that the source of the scatter send the same number of objects to each process in the communicator

- **We can use `MPI_Scatterv`**

```
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  */);
```

- **sendcounts[q] is the number of objects of type sendtype being sent to process q**

- **displacements[q] specifies the start of the block that is being sent to process q**

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

Maintaining the best tour

- **When a process finds a new best tour, it needs to send its cost to the other processes**
  - each process only makes use of the cost of the current best tour when it calls `Best_tour`
  - we can't use `MPI_Bcast` to communicate a new best cost to the other processes
    - `MPI_Bcast` is blocking and every process in the communicator must call `MPI_Bcast`
    - in parallel tree search, the only process that will know that a broadcast should be executed is the process that has found a new best cost
    - a thread calling `MPI_Bcast` will probably block in the call and never return, since it will be the only process that calls it

Simplest solution

- the process that finds a new best cost use `MPI_Send` to send it to all the other processes

```
for (dest = 0; dest < comm_sz; dest++)
   if (dest != my_rank)
      MPI_Send(&new_best_cost, 1, MPI_INT, dest, NEW_COST_TAG,
          comm);
```

- we use a special tag `NEW_COST_TAG` telling the receiving process that the message is a new cost
- the destination processes periodically check for the arrival of new best tour costs

```
MPI_Recv(&received_cost, 1, MPI_INT, MPI_ANY_SOURCE, NEW_COST_TAG,
      comm, &status);
```

Simplest solution (cont.)

- the process will block until a matching message arrives
- MPI provides a function that only *checks* to see if a message is available
  - it doesn't actually try to receive a message
  - it checks to see if a message from process rank `source` in communicator `comm` and with tag `tag` 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 such a message is available
  - `*msg_avail_p` will be assigned the value *true*
  - the members of `*status_p` will be assigned the appropriate values
- If no message is available
  - `*msg_avail_p` will be assigned the value *false*

Simplest solution (cont.)

- if `msg_avail` is true, we can receive the new cost with a call to `MPI_Recv` (in the `Best_tour` function)
- before checking whether our new tour is the best tour, we can check for new tour costs from other processes with

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while (msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if (received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
     &status);
}  /* while */
```

- if there is no buffering available for the sender, then the loop of calls to `MPI_Send` can cause the sending process to block
  - until a matching receive is posted

Modes and Buffered Sends

- **Four modes for sends**
  - standard : decide whether to copy the contents of the message into its own storage or to block until a matching receive is posted
  - synchronous : the send will block until a matching receive is posted
  - ready : the send is erroneous unless a matching receive is posted before the send is started
  - buffered : copy the message into local temporary storage if a matching receive hasn't been posted
    - the local temporary storage must be provided by the user program
- **Each mode has a different function**
  - `MPI_Send`, `MPI_Ssend`, `MPI_Rsend`, and `MPI_Bsend`, respectively
  - the argument lists are identical to the argument lists for `MPI_Send`

```
int MPI_Xsend(
      void*          message       /* in */,
      int            message_size  /* in */,
      MPI_Datatype   message_type  /* in */,
      int            dest          /* in */,
      int            tag           /* in */,
      MPI_Comm       comm          /* in */);
```

```
int MPI_Buffer_attach(
      void*  buffer       /* in */,
      int    buffer_size  /* in */);

int MPI_Buffer_detach(
      void*   buf_p       /* out */,
      int*    buf_size_p  /* out */);
```

```
char buffer[1000];
char* buf;
int buf_size;
. . .
MPI_Buffer_attach(buffer, 1000);
. . .
/* Calls to MPI_Bsend */
. . .
MPI_Buffer_detach(&buf, &buf_size);
```

- **The amount of storage that's needed for the data that's transmitted in `MPI_Bsend`**
  - can be determined with a call to `MPI_Pack size`

```
int MPI_Pack_size(
      int           count     /* in  */,
      MPI_Datatype  datatype  /* in  */,
      MPI_Comm      comm      /* in  */,
      int*          size_p    /* out */);
```

  - the output argument gives an upper bound on the number of bytes needed to store the data in a message

```
int data_size;
int message_size;
int bcast_buf_size;

MPI_Pack_size(1, MPI_INT, comm, &data_size);
message_size = data_size + MPI_BSEND_OVERHEAD;
bcast_buf_size = (comm_sz - 1)*message_size;
```

Printing the best tour

- **Process 0 prints out the best tour as well as its cost after the program finishes**

- **Naïve approach**
  - each process checks its local best tour cost and compares it to the global best tour cost
  - if they're the same, the process sends its local best tour to process 0

- **Problem**
  - there might be multiple "best" tours having the same cost
  - multiple processes will try to send their best tours to process 0
  - all but one of the threads could hang in a call to `MPI_Send`

Solution : using MPI_Allreduce

- we know what the best cost was
- we wouldn't know who owned it
- predefined operator, MPI_MINLOC
  - first value : the value to be minimized
  - second value : the *location* of the minimum
  - if there are multiple minimum costs, the location will be the lowest rank

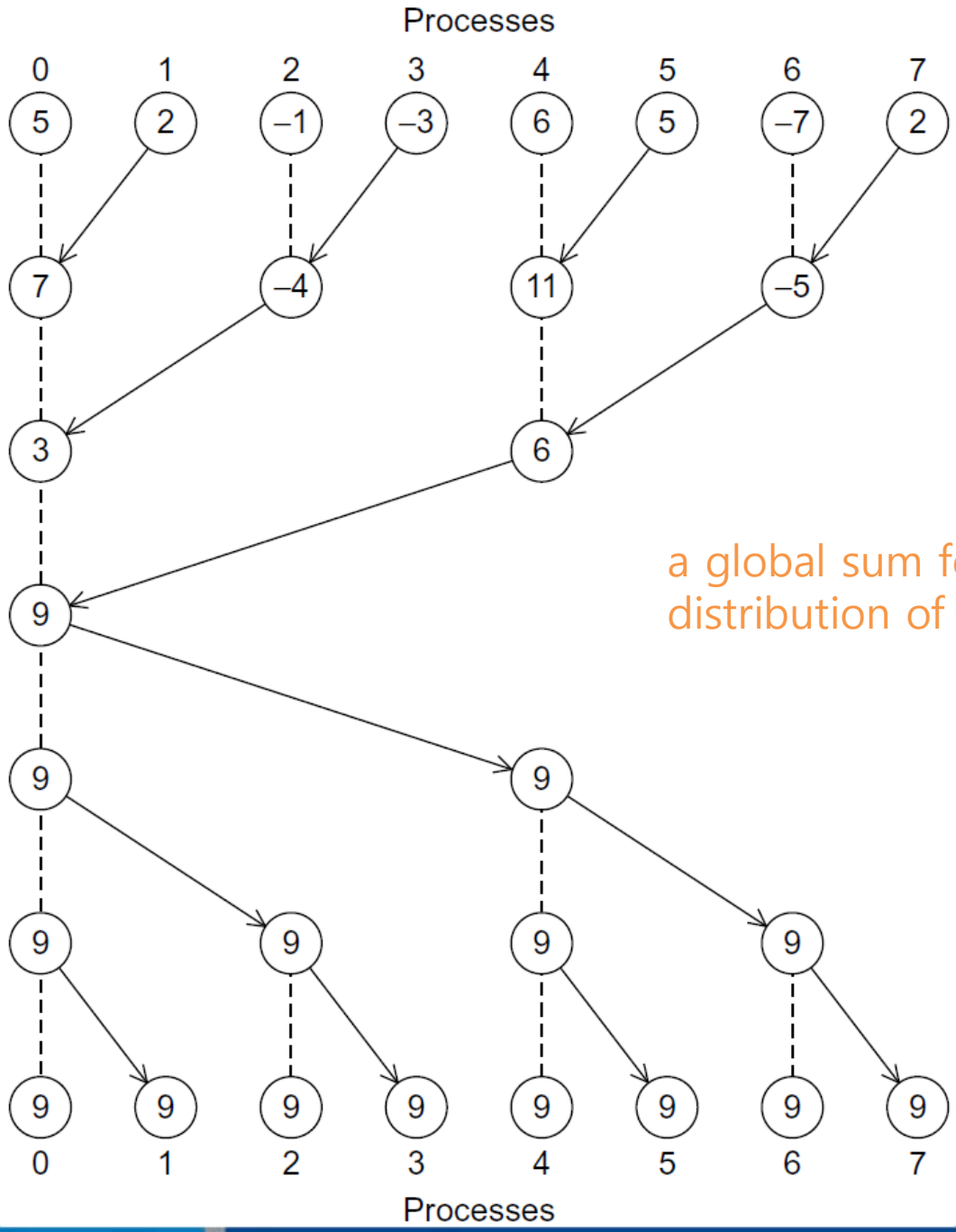
```
struct {
   int cost;
   int rank;
} loc_data, global_data;

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;
```

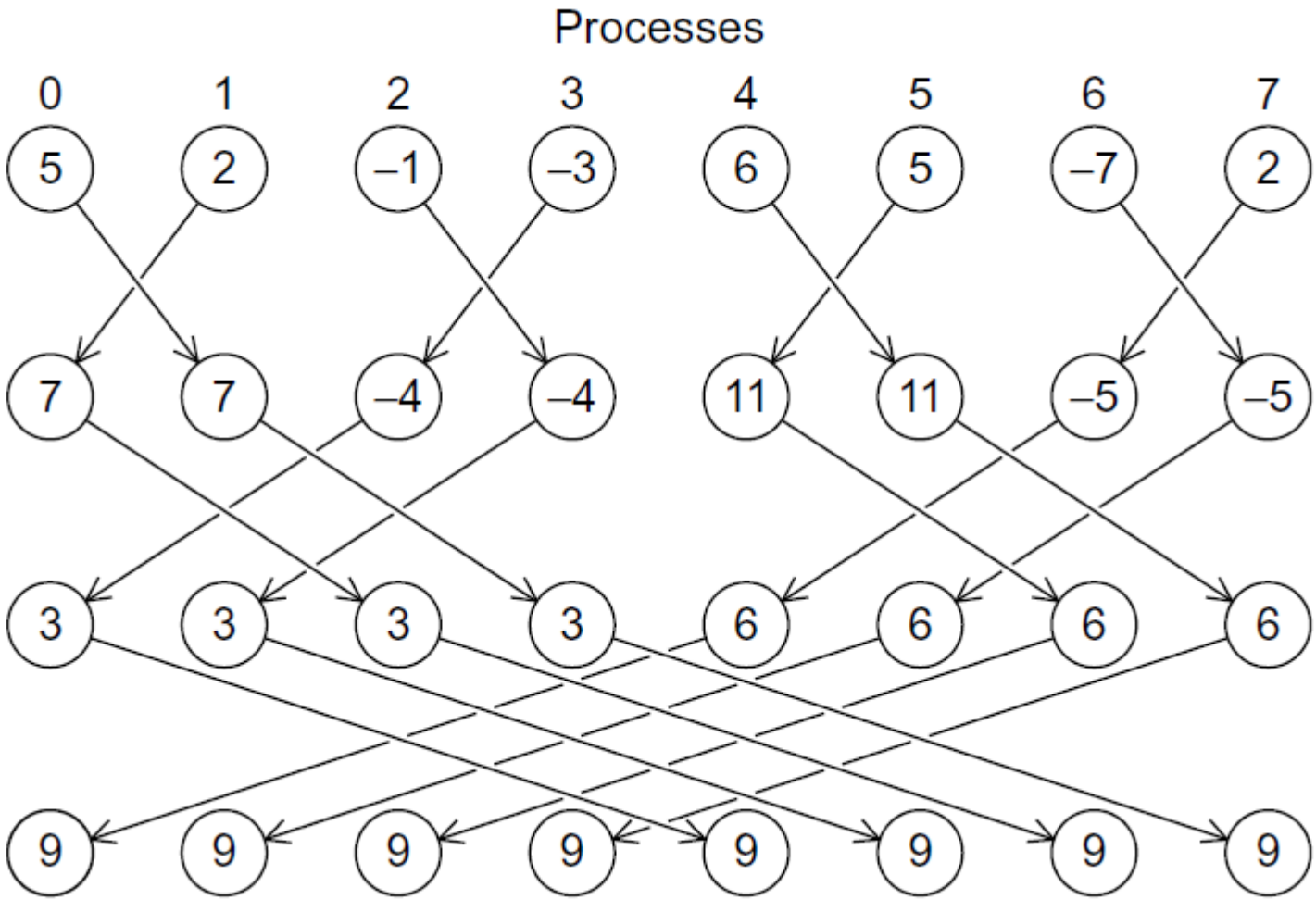
MPI_Allreduce

- **All of the processes might need the result of a global sum in order to complete some larger computation**
  - ➢ if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum
  - ➢ alternatively, we might have the processes exchange partial results instead of using one-way communications (called **butterfly**)
  - ➢ MPI provides a variant of MPI_Reduce that will store the result on all the processes in the communicator
    - no `dest_process` since all the processes should get the result

```
int MPI_Allreduce(
          void*         input_data_p    /* in  */,
          void*         output_data_p   /* out */,
          int           count           /* in  */,
          MPI_Datatype  datatype        /* in  */,
          MPI_Op        operator        /* in  */,
          MPI_Comm      comm            /* in  */);
```



a global sum followed by distribution of the result



a butterfly-structured global sum

Table 3.2 Predefined Reduction Operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Unreceived messages

- **Some messages might not be received during the execution of the parallel tree search**
  - a process may finish searching its subtree before some other process has found a best tour
  - this won't cause the program to print an incorrect result
  - however, unreceived messages can cause problems with the call to `MPI_Buffer_detach` or the call to `MPI_Finalize`
  - a process can hang in one of these calls if it is storing buffered messages that were never received
- **Solution**
  - before we attempt to shut down MPI, we can try to receive any outstanding messages by using `MPI_Iprobe`
  - the exact code is very similar to the code we used to check for new best tour costs (in p.70)

Implementation of tree search using MPI and dynamic partitioning

- **When a thread ran out of work (its stack was empty)**
  - it went into a condition wait (Pthreads) or
  - it went into a busy-wait (OpenMP)
- **until either**
  - it received additional work or
  - it was notified that there was no more work

- **In MPI**
  - when a process runs out of work, there's no condition wait, but, it can enter a busy-wait
    - it waits to receive either more work or notification that the program is terminating
  - a process with work can split its stack and send work to an idle process

Key difference

- there is no central repository of information on which processes are waiting for work
  - so, a process that splits its stack can't just dequeue a queue of waiting processes or call a function such as `pthread_cond_signal`
- it needs to "know" a process that's waiting for work so it can send the waiting process more work
- a process that has run out of work should send a request for work to another process
  - when a process enters the `Terminated` function, it can check to see if there's a request for work from some other process
  - if there is, it can send part of its stack to the requesting process or send a rejection

```
1  if (My_avail_tour_count(my_stack) >= 2) {
2     Fulfill_request(my_stack);
3     return false;  /* Still more work */
4  } else { /* At most 1 available tour */
5     Send_rejects();  /* Tell everyone who's requested */
6                      /* work that I have none         */
7     if (!Empty_stack(my_stack)) {
8        return false;  /* Still more work */
9     } else {  /* Empty stack */
10       if (comm_sz == 1) return true;
11       Out_of_work();
12       work_request_sent = false;
13       while (1) {
14          Clear_msgs();  /* Msgs unrelated to work, termination */
15          if (No_work_left()) {
16             return true;  /* No work left.  Quit */
17          } else if (!work_request_sent) {
18             Send_work_request();  /* Request work from someone */
19             work_request_sent = true;
20          } else {
21             Check_for_work(&work_request_sent, &work_avail);
22             if (work_avail) {
23                Receive_work(my_stack);
24                return false;
25             }
26          }
27       }  /* while */
28    } /* Empty stack */
29 } /* At most 1 available tour */
```

Splitting the stack

- **MPI version of Split stack**
  - packs the contents of the new stack into contiguous memory
    - MPI provides `MPI_Pack` for packing data into a buffer of contiguous memory
  - sends the block of contiguous memory
    - which is unpacked by the receiver into a new stack
    - MPI provides `MPI_Unpack` for unpacking data from a buffer of contiguous memory
- **Data structure**

```
typedef struct {
   int* cities; /* Cities in partial tour           */
   int count;   /* Number of cities in partial tour */
   int cost;    /* Cost of partial tour             */
} tour_struct;
typedef tour_struct* tour_t;
```

Sending a variable with type `tour_t`

```
void Send_tour(tour_t tour, int dest) {
   int position = 0;

   MPI_Pack(tour->cities, n+1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->count, 1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->cost, 1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Send(contig_buf, position, MPI_PACKED, dest, 0, comm);
}  /* Send_tour */
```

Receiving a variable with type `tour_t`

```
void Receive_tour(tour_t tour, int src) {
   int position = 0;

   MPI_Recv(contig_buf, LARGE, MPI_PACKED, src, 0, comm,
         MPI_STATUS_IGNORE);
   MPI_Unpack(contig_buf, LARGE, &position, tour->cities, n+1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->count, 1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->cost, 1,
         MPI_INT, comm);
}  /* Receive_tour */
```

Distributed termination detection

- **Distributed termination detection is a challenging problem**
  - much work has gone into developing algorithms that are guaranteed to correctly detect it
- **The simplest algorithm**
  - it relies on keeping track of a quantity that is conserved and can be measured precisely (let's call it energy)
  - at the start of the program, each process has 1 unit of energy
  - when a process runs out of work, it sends its energy to process 0
  - when a process fulfills a request for work, it divides its energy in half,
    - keeping half for itself
    - sending half to the process that's receiving the work
  - the program should terminate when process 0 finds that it has received a total of `comm_sz` units
    - since energy is conserved
    - since the program started with `comm_sz` units

Sending requests for work

- **Many possibilities for choosing a destination**
  - Loop through the processes in round-robin fashion
    - start with `(my_rank + 1) % comm_sz`
    - increment this destination (modulo `comm_sz`) each time a new request is made
    - potential problem : two processes can get “in synch” and request work from the same destination repeatedly
  - Keep a global destination for requests on process 0
    - when a process runs out of work, it first requests the current value of the global destination from 0
    - process 0 can increment this value (modulo `comm_sz`) each time there’s a request
  - Each process uses a random number generator to generate destinations
    - the random choice of successive process ranks should reduce the chance that several processes will make repeated requests to the same process

Checking for and receiving work

- **It's critical that the sending process repeatedly check for a response from the destination**
  - if the sending process simply checks for a message from the destination
  - it may be "distracted" by other messages from the destination and never receive work that's been sent

- **The `Check_for_work` function**
  - it should first probe for a message from the destination indicating work is available
  - if there isn't such a message, it should probe for a message from the destination saying there's no work available
  - if there is work available, the `Receive_work` function can
    - receive the message with work
    - unpack the contents of the message buffer into the process' stack

Performance of the MPI programs

- Every instance the Pthreads implementation outperforms the MPI implementation
- The cost of stack splitting in the MPI implementation is quite high; in addition to the cost of the communication
- The static MPI parallelization outperforms the dynamic parallelization for relatively small problems with few processes
- If a problem is large enough to warrant the use of many processes, the dynamic MPI program is much more scalable

Table 6.11 Performance of MPI and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						Second Problem					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th/Pr	*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)

InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology