# 并行计算

# （Parallel Computing）

# 并行程序开发（二）

## 学习内容：

- Tree search

# 1. Tree search

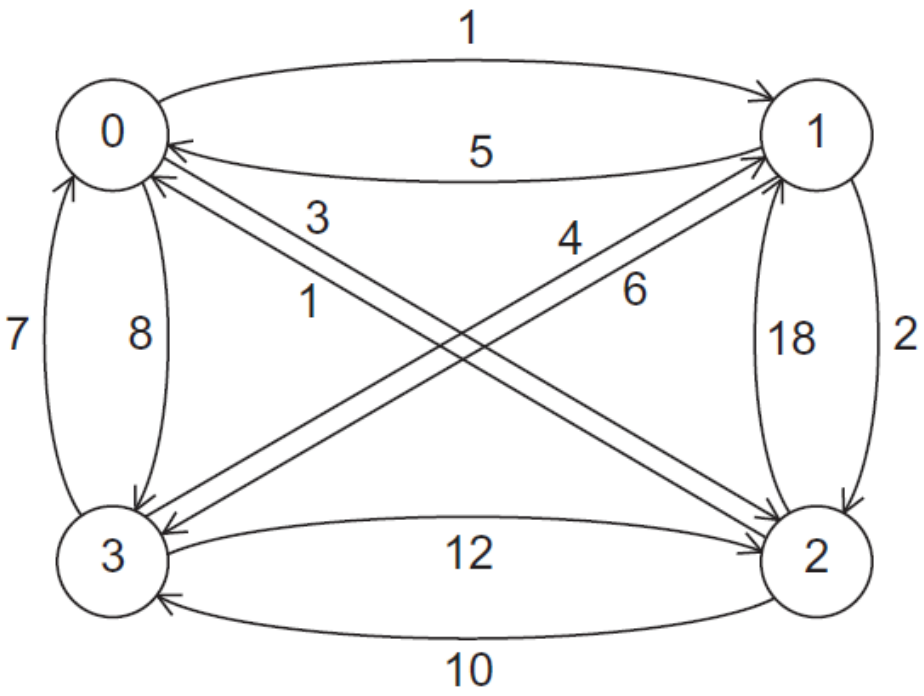
- TSP（Traveling Salesperson Problem）
  - 寻找一条成本最低的路线（从起点出发，遍历每个城市一次，回到起点）
  - NP-complete 问题
  - 目前还没有解决方案在任何情况下都比穷举搜索（ exhaustive search ）好

# 1. Tree search

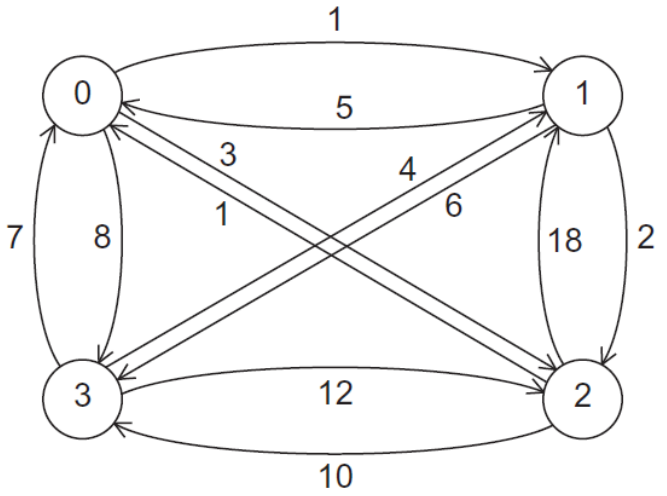
- TSP（Traveling Salesperson Problem）
  - 一种简单的解决方案：tree search
  - 叶子节点：tour；其他节点：partial tour
  - 每个树节点有与之关联的 cost（cost of partial tour）
  - 希望跟踪 best tour 的 cost，如果发现节点（partial tour）不能产生 best tour，则无须搜索其子节点

# 1. Tree search

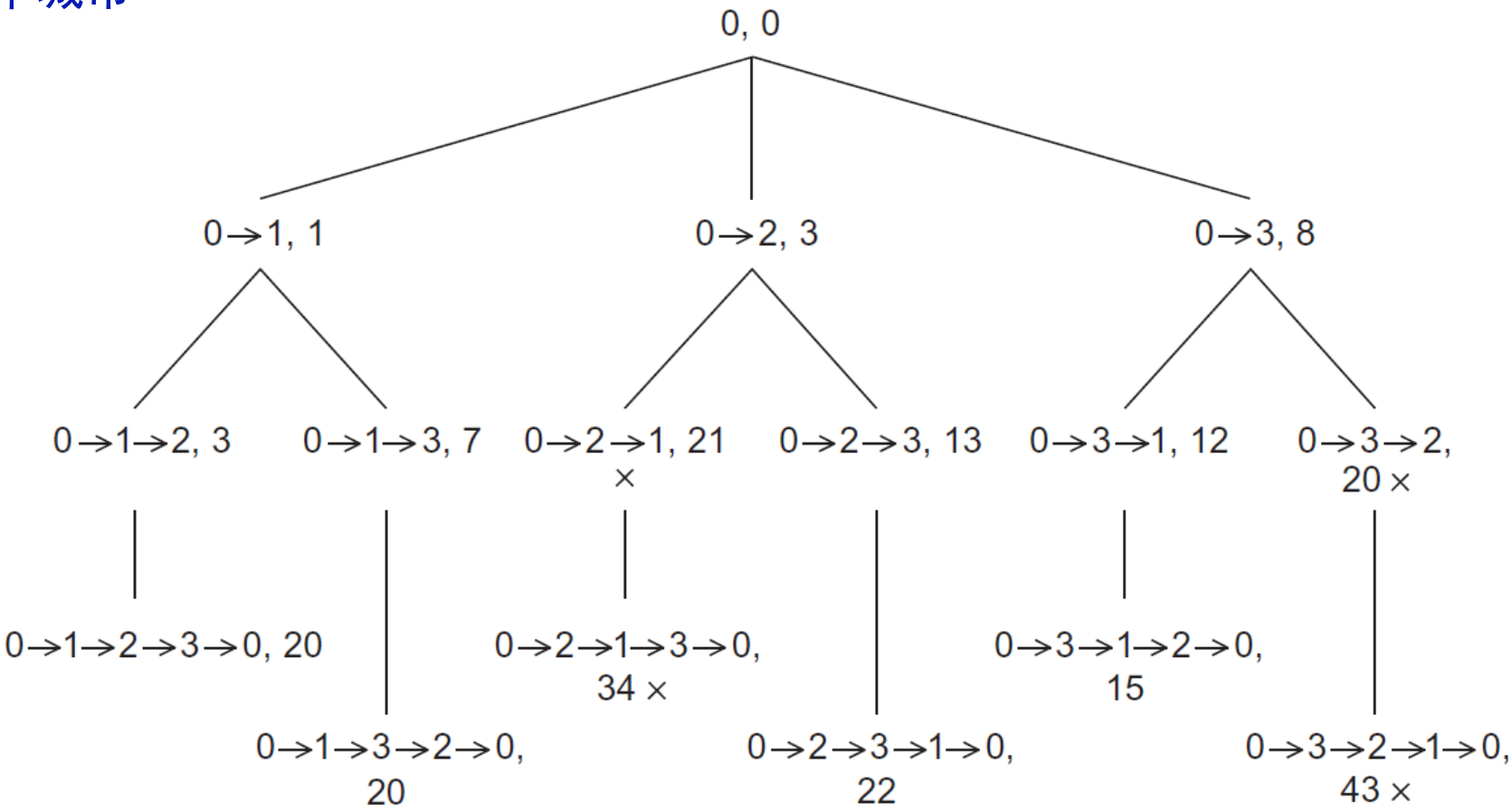
- TSP（Traveling Salesperson Problem）
  - 四个城市



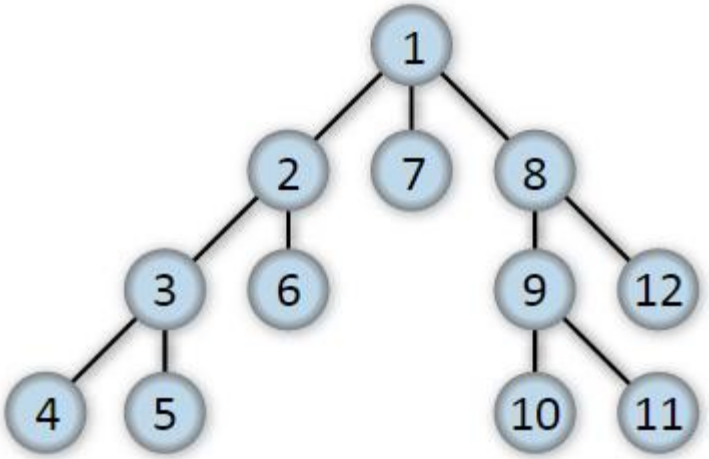

# 1. Tree search



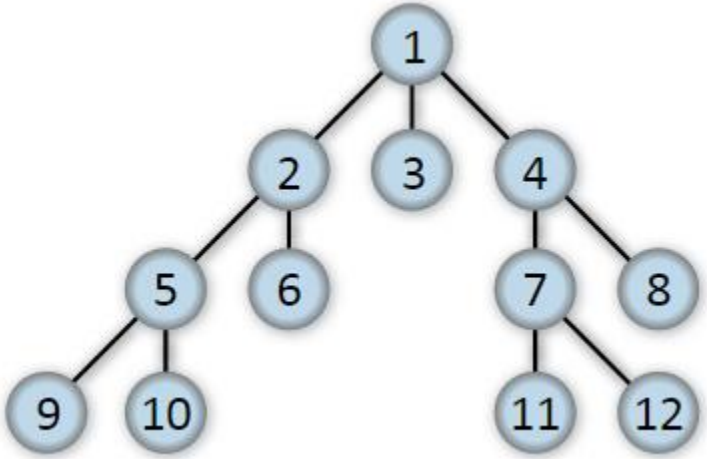

- TSP（Traveling Salesperson Problem）
  - 四个城市



# 1. Tree search




depth-first search




breadth-first search

# 1. Tree search

- TSP：深度优先搜索（递归）
  - City_count：检查 partial tour 中是否有 n 个城市
  - Feasible：检查城市是否已被访问过，或者是否能产生 best tour

```
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 */
```

# 1. Tree search

- TSP：深度优先搜索（递归）
  - 函数调用有开销
  - 任一给定时间内，只有当前树节点可以被访问

# 1. Tree search

- TSP：深度优先搜索（非递归 1 ）
  - 在进入更深的树节点分支前，将所需要的数据压入堆栈
  - 到达叶子节点或者发现当前节点无法产生 best tour 时，需要返回树，将数据弹出堆栈

# 1. Tree search

- TSP：深度优先搜索（非递归 1 ）

```
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. Tree search

- TSP：深度优先搜索（非递归 2 ）

```
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 }
```

# 1. Tree search

- TSP：深度优先搜索（非递归 2 ）
  - ➢ 数据结构
    - tour

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

    - stack

```
typedef struct {
   tour_t* list;
   int list_sz;
}  stack_struct;
```

# 1. Tree search

- TSP：深度优先搜索（非递归 2 ）
  - 数据结构
    - stack

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

# 1. Tree search

- TSP：深度优先搜索（非递归 2 ）
  - 数据结构
    - digraph 的表示
    - List？
    - 邻接矩阵（adjacency matrix）：n * n 矩阵

```
#define Cost(city1, city2) (digraph[city1*n + city2])
```

# 1. Tree search

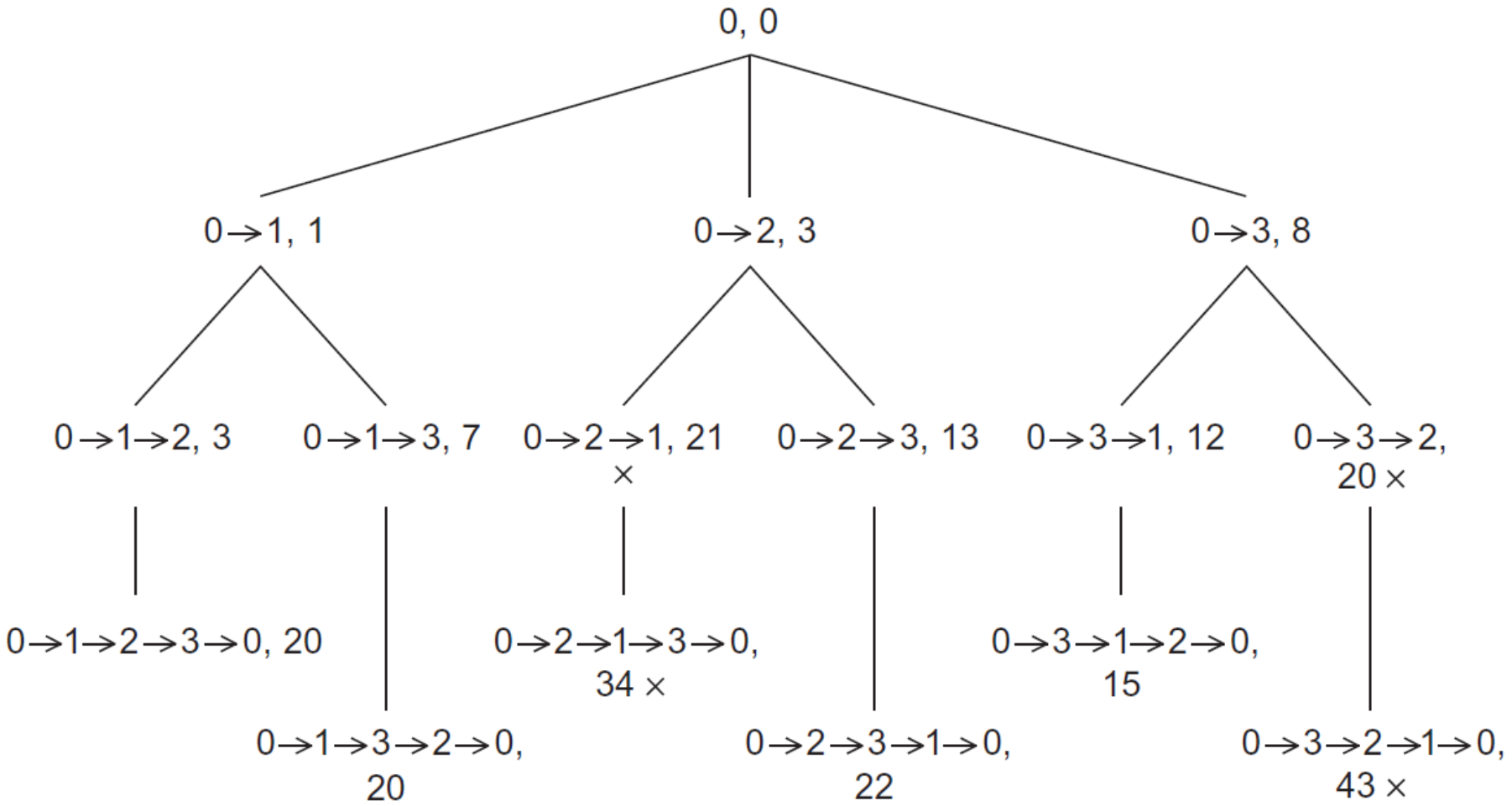
- 三种串行方法的运行时间

| Recursive | First Iterative | Second Iterative |
|---|---|---|
| 30.5 | 29.2 | 32.9 |

The digraph contains 15 cities.

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

# 2.并行化 Tree search



# 2.并行化 Tree search

- 并行化
  - 自然的想法是将子树作为任务
    - 如：0->1 给线程 0, 0->2 给线程 1, 0->3 给线程 2
    - 深度优先 or 宽度优先？
  - 需要考虑 best tour 的更新问题
    - 每个任务检查 best tour，来决定当前的部分路线是否可行或者当前的完整路线是否具有更低的成本

# 2.并行化 Tree search

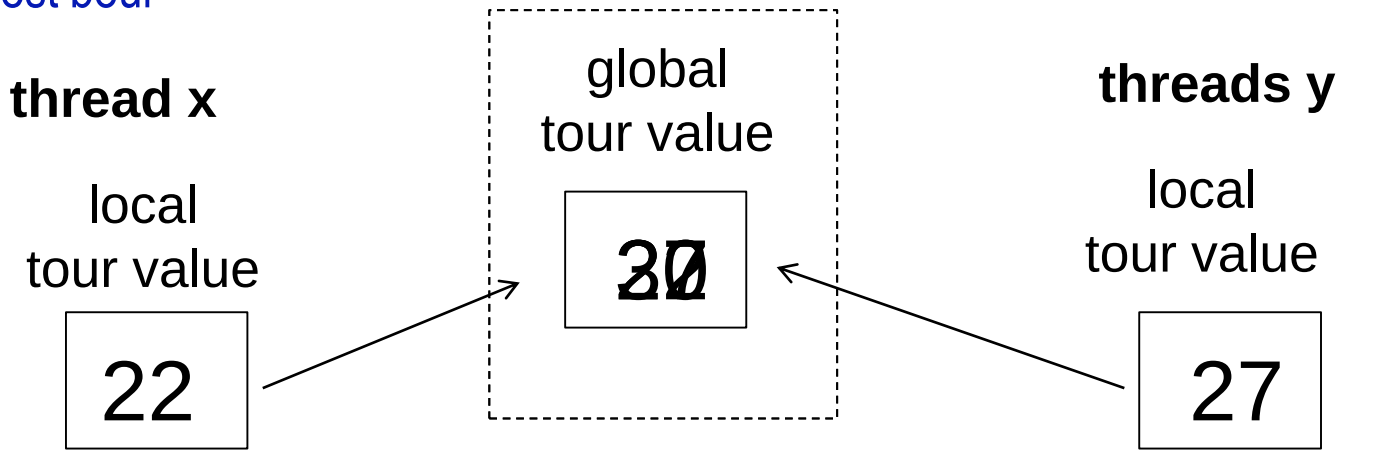
- 并行化
  - ➢ best tour 的数据结构
    - 共享内存系统中，best tour数据结构可以共享。Feasible 函数可以简单的检查该数据结构，而需要对 best tour 更新时将产生竞争，需要一些“锁”的机制
    - 分布式内存系统中，可以有多种选择。最简单的，可以让每个进程独立的操作各自的子树，Feasible 和 Update_best_tour 函数检查和更新进程局部的 best tour，所有的进程结束搜索后，执行 reduction 操作，来获取全局的 best tour

# 2.并行化 Tree search

- 并行化
  - 确保“best bour”
    - 当一个线程测试并决定其得到的路线是否为best tour时，需要确保：
      1. 为 best tour 加锁，防止竞争
      2. 当另一线程更新 best tour 时，第一次测试有可能比较的是更新前的值，要确保不能写入差的值
    - locking、testing again

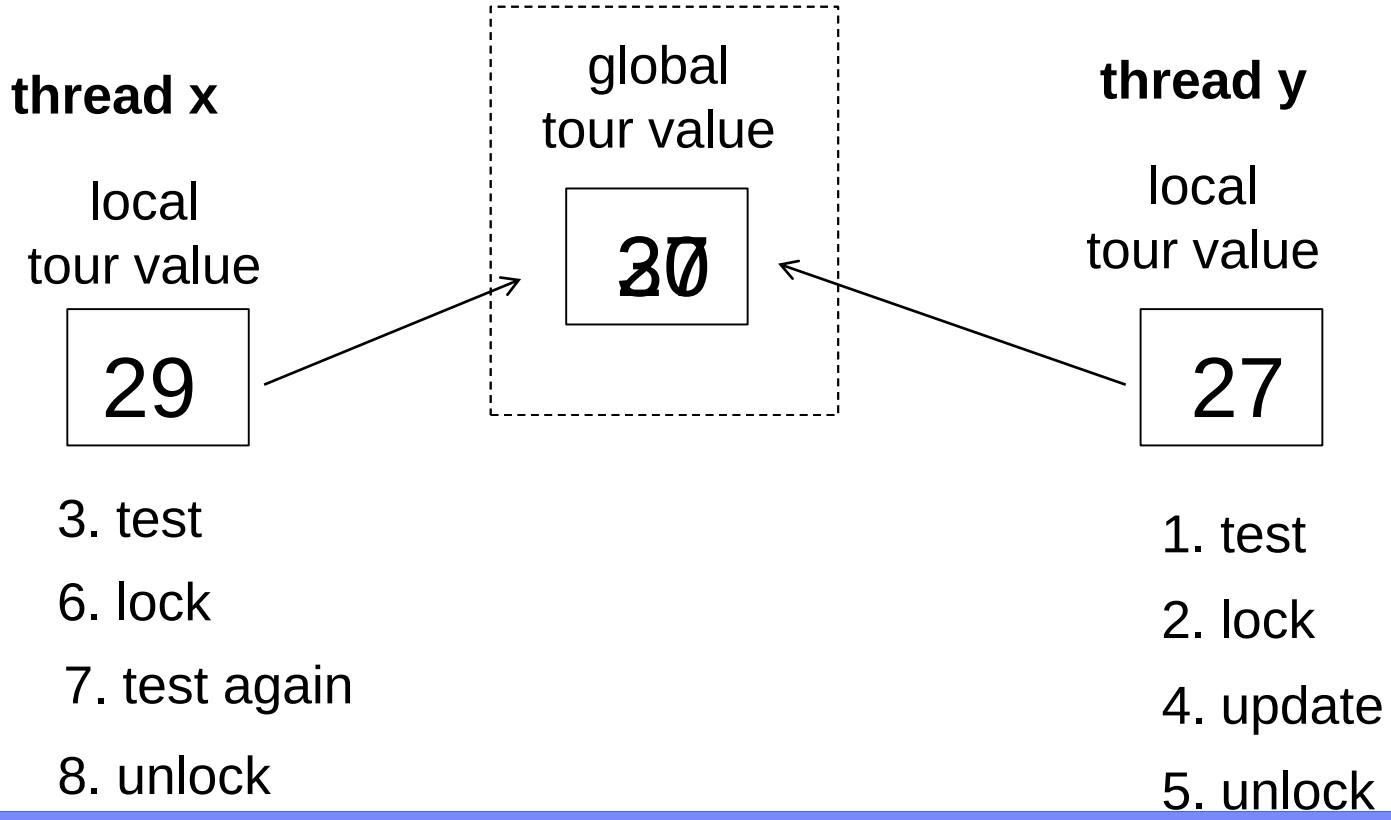
# 2.并行化 Tree search

- 并行化
  - 确保 “best bour”



# 2.并行化 Tree search

- 并行化
  - 确保“best bour”



# 2.并行化 Tree search

- 并行化
  - 动态任务映射
    - 负载均衡（load balance）问题
    - 当一个任务完成其工作，可以从其他未完成工作的任务获得工作

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行静态并行化
  - 一个线程采用宽度优先搜索（breadth-first search）产生足够多的部分路线（partial tour），使得每个线程可以至少获得一个 partial tour
  - 每个线程对自己的 partial tour 进行迭代搜索

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行静态并行化

```
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);
}
```

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行静态并行化
  - Update_best_tour

```
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).
```

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行动态并行化
  - 如果对子树（subtrees）的初始分配不均衡，静态并行化无法重新分配
  - 会导致有的线程已经完成工作，而其他线程仍在工作

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行动态并行化（基本思想）
  - 当线程完成工作，！Empty(my_stack) 为 false，并不退出 while 循环，而是等待其他线程是否可以提供更多的工作
  - 另一方面，如果一个线程仍在工作，并且发现有线程处于没有工作的状态，如果工作线程中的 stack 有两个以上的 partial tour，则分割其 stack，为其他等待工作的线程提供工作

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行动态并行化（基本思想）
  - Pthreads 的条件变量（condition variable）提供了实现机制
  - 完成工作的线程：pthread_cond_wait
  - 进行工作的线程发现有等待的线程，分割 stack 后，pthread_cond_signal
  - 记录 pthread_cond_wait 的线程数量，当一个线程完成工作，并发现有 thread_count -1 个线程处于等待状态，则调用 pthread_cond_broadcast 唤醒其他线程，这些线程意识到所有线程已经完成工作，退出

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行动态并行化
  - Terminated 函数

```
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 */
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 */
```

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行动态并行化
  - Terminated 函数

```
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 */
```

# 2.并行化 Tree search

- 使用 pthreads 对 tree search 进行动态并行化
  - Terminated 函数

```
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
```

# 2.并行化 Tree search

## 15-city problems

| 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

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - 树的划分
  - 检查并更新 best tour
  - 搜索结束后，确保进程 0 有 best tour 的拷贝，用来输出

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - 树的划分
    - 进程 0 可以像 pthreads 那样生成 comm_sz 个部分路线的列表
    - 向其他进程发送路线
    - MPI_Scatter 在此不适用，因为部分路线的数量不一定能被 comm_sz 整除

```
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  */);
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - 树的划分
    - MPI_Scatterv 可以向不同进程发送不同数量的对象

```
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  */)
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - 树的划分
    - MPI_Scatterv 可以向不同进程发送不同数量的对象

```
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  */)
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - best tour 的维护
    - 每个进程使用自己的 best tour？
    - 当一个进程找到新的 best tour，发送给其他进程（只发送 cost）
    - MPI_Bcast？
    - MPI_Bcast 为阻塞调用且通信器中的所有进程都需调用 MPI_Bcast

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - best tour 的维护
    - 最简单的方法：MPI_Send

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

    - tag：NEW_COST_TAG，告诉接收进程，消息的类型

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - best tour 的维护
    - MPI_Recv?

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

    - MPI_Recv 为阻塞调用，直到匹配的消息到达
    - 如果没有消息到达，如：没有进程发现新的 best cost，则进程将被挂起

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - best tour 的维护
    - MPI_Iprobe：检查是否有消息到达，并不实际接收消息，不阻塞

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

    - 检查是否有从 source 进程来的 tag 消息，如果有：*msg_avail_p 为 true，*staturs_p 的成员被赋予相应的值

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - best tour 的维护
    - MPI_Iprobe：检查是否有消息到达，并不实际接收消息，不阻塞

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail, &status);
```

    - 检测新的 best cost

```
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 */
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - send 的四种模式
    - 标准（standard）
    - 同步（synchronous）
    - 准备（ready）
    - 缓冲（buffered）

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - send 的四种模式
    - MPI_Send：标准模式，MPI的实现决定了拷贝消息到自己的存储区还是阻塞，直到匹配的接收函数启动
    - MPI_Ssend：同步模式，阻塞，直到匹配的接收函数启动
    - MPI_Rsend：准备模式，错误，除非匹配的接收函数在发送前启动
    - MPI_Bsend：缓冲模式，如果匹配的接收函数未启动，将消息拷贝到本地临时存储区

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行静态划分
  - 打印 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;
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分
  - 与 Pthreads 关键的区别在于：没有关于哪个进程正在等待工作的中央存储区域，因此，分割堆栈的进程不能通过 pthread_cond_signal 来通知等待的进程
  - 完成工作的进程需要主动向其他进程发送工作请求

```
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 */
```

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

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分
  - 第2行：Fulfill_request 检查进程是否收到工作请求，如果收到，分割堆栈，向请求进程发送工作；如未收到，返回
  - 第21行：Check_for_work 检查请求被满足或拒绝
  - 分割堆栈：MPI版本的 Split_stack 打包堆栈中的内容到连续的内存中，并发送该内存块，接收者拆包到其堆栈中

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分

```
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     */)
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分

```
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     */)
```

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分
  - 分布式终止检查
    - Out_of_work 和 No_work_left（第11行和第15行）

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

Termination Events that Result in an Error

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分
  - 分布式终止检查
    - 最简单的算法：记录一个可追踪的、可正确测量的数值
  - 发送工作请求：向谁发送？
  - 检测并接收工作：是否需要区分消息类型？

# 2.并行化 Tree search

- 使用 MPI 对 tree search 进行动态划分

| 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)

# Which API ? (MPI, Pthreads or OpenMP)

- 分布式内存 or 共享式内存
  - 应用所需内存数量
    - 分布式内存系统可以提供更多的内存
    - 且分布式内存系统可以提供更多的cache
  - 共享式内存程序可以重用更多的串行代码
  - 考虑任务之间的通信

# Which API ? (MPI, Pthreads or OpenMP)

- 共享式内存
  - 如果串行程序中大部分可以通过OpenMP中的 parallel for 指令来优化，则OpenMP更合适
  - 如果线程间需要复杂的同步，如：读写锁、信号量等，则Pthreads更容易使用