Roadmap

- Solving non-trivial problems.
- The traveling salesman problem.
- Applying Foster’s methodology.
- Starting from scratch on algorithms that have no serial analog.
Tree search problem (TSP)

- An NP-complete problem.
- No known solution to TSP that is better in all cases than exhaustive search.
- Ex., the travelling salesperson problem, finding a minimum cost tour.
A Four-City TSP

Search Tree for Four-City TSP
Pseudo-code for a recursive solution to TSP using depth-first search

```c
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 */
```

Pseudo-code for an implementation of a depth-first solution to TSP without recursion

```c
for (city = n-1; city >= 1; city--)
    Push(stack, city);
while (!Empty(stack)) {
    city = Pop(stack);
    if (city == NO_CITY) // End of child list, back up
        Remove_last_city(curr_tour);
    else {
        Add_city(curr_tour, city);
        if (City_count(curr_tour) == n) {
            if (Best_tour(curr_tour))
                Update_best_tour(curr_tour);
            Remove_last_city(curr_tour);
        } else {
            Push(stack, NO_CITY);
            for (nbr = n-1; nbr >= 1; nbr--)
                if (Feasible(curr_tour, nbr))
                    Push(stack, nbr);
        }
    }
} /* if Feasible */
} /* while !Empty */
```
Pseudo-code for a second solution to TSP that doesn’t use recursion

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

Using pre-processor macros

/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
    return tour->cities[i];
} /* Tour_city */

/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
### 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></td>
<td>30.5</td>
<td>29.2</td>
<td>32.9</td>
</tr>
</tbody>
</table>

(in seconds)

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

### Making sure we have the “best tour” (1)

- When a process finishes a tour, it needs to check if it has a better solution than recorded so far.
- The global Best_tour function only reads the global best cost, so we don’t need to tie it up by locking it. There’s no contention with other readers.
- If the process does not have a better solution, then it does not attempt an update.
Making sure we have the “best tour” (2)

• 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.

Making sure we have the “best tour” (3)

• In the case where a thread tests and decides it has a better global solution, we need to ensure two things:
  1) That the process locks the value with a mutex, preventing a race condition.
  2) In the possible event that the first check was against an old value while another process was updating, we do not put a worse value than the new one that was being written.

• We handle this by locking, then testing again.
First scenario

process x
  local tour value
  22

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

process y
  local tour value
  20

  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
  20

  1. test
  2. lock
  4. update
  5. unlock
Pseudo-code for a Pthreads implementation of a statically parallelized solution to TSP

```c
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 TREE SEARCH USING MPI AND STATIC PARTITIONING
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 */)
```
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 */);
```

Terminated Function for a Dynamically Partitioned TSP solver that Uses MPI.
Modes and Buffered Sends

- MPI provides four modes for sends.
  - Standard
  - Synchronous
  - Ready
  - Buffered

Printing the best tour

```c
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_INT, MPI_MINLOC, comm);
if (global_data.rank == 0) return; /* 0 already has the best tour */
if (my_rank == 0)
    Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
    Send best tour to process 0;
```
Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```c
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 */
            }
        }  /* while */
    }  /* Empty stack */
}  /* At most 1 available tour */
```

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

```c
} else if (!work_request_sent) {
    Send_work_request();  /* Request work from someone */
    work_request_sent = true;
} else {
    Check_for_work(&work_request_sent, &work_avail);
    if (work_avail) {
        Receive_work(my_stack);
        return false;
    }
}  /* while */
/* Empty stack */
}  /* At most 1 available tour */
```
**Packing data into a buffer of contiguous memory**

```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/out */,
    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 */) 
```
Table 6.10 Termination Events that Result in an Error

<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 Notify 1, 2 Notify 0, 2 Working oow = 1 oow = 1 oow = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Send request to 1 Send Request to 2 Send notify fr 1 oow = 1 oow = 1 oow = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>oow = 1 Recv notify fr 0 Recv request fr 1 oow = 2 oow = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>oow = 1 Send work to 1 oow = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>oow = 1 Recv work fr 2 Recv notify fr 0 oow = 1 oow = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>oow = 1 Notify 0 Working oow = 1 oow = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>oow = 1 Recv request fr 0 Out of work Notify 0, 1 oow = 1 oow = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>oow = 1 Send request to 1 oow = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Recv 1st notify fr 1 Recv notify fr 2 oow = 3 oow = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Quit Recv request fr 2 oow = 1 oow = 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance of MPI and Pthreads implementations of tree search

<table>
<thead>
<tr>
<th>Th/Pr</th>
<th>First Problem</th>
<th>Second Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>Pth</td>
<td>MPI</td>
</tr>
<tr>
<td>1</td>
<td>35.8</td>
<td>40.9</td>
</tr>
<tr>
<td>2</td>
<td>29.9</td>
<td>34.9</td>
</tr>
<tr>
<td>4</td>
<td>27.2</td>
<td>31.7</td>
</tr>
<tr>
<td>8</td>
<td>35.7</td>
<td>45.5</td>
</tr>
<tr>
<td>16</td>
<td>20.1</td>
<td>10.5</td>
</tr>
</tbody>
</table>

(in seconds)
Parallelizing the Tree Search Programs Using OpenMP

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

```c
// Pthreads
if (my_rank == whatever)

#pragma omp single
```

```c
// OpenMP emulated condition wait

/* Global vars */
int awakened_thread = -1;
work_remains = 1; /* true */

omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```
Performance of OpenMP and Pthreads implementations of tree search

<table>
<thead>
<tr>
<th>Th</th>
<th>First Problem</th>
<th></th>
<th>Second Problem</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Dynamic</td>
<td></td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td>OMP</td>
<td>Pth</td>
<td>Dynamic</td>
</tr>
<tr>
<td>1</td>
<td>32.5</td>
<td>32.7</td>
<td>(0)</td>
<td>34.7</td>
</tr>
<tr>
<td>2</td>
<td>27.7</td>
<td>27.9</td>
<td>(6)</td>
<td>28.9</td>
</tr>
<tr>
<td>4</td>
<td>25.4</td>
<td>25.7</td>
<td>(75)</td>
<td>25.9</td>
</tr>
<tr>
<td>8</td>
<td>28.0</td>
<td>23.8</td>
<td>(134)</td>
<td>22.4</td>
</tr>
</tbody>
</table>

(in seconds)

Concluding Remarks (1)

- In developing the reduced MPI solution to the n-body problem, the “ring pass” algorithm proved to be much easier to implement and is probably more scalable.

- In a distributed memory environment in which processes send each other work, determining when to terminate is a nontrivial problem.
Concluding Remarks (2)

• When deciding which API to use, we should consider whether to use shared- or distributed-memory.

• We should look at the memory requirements of the application and the amount of communication among the processes/threads.

Concluding Remarks (3)

• If the memory requirements are great or the distributed memory version can work mainly with cache, then a distributed memory program is likely to be much faster.

• On the other hand if there is considerable communication, a shared memory program will probably be faster.
Concluding Remarks (3)

- In choosing between OpenMP and Pthreads, if there's an existing serial program and it can be parallelized by the insertion of OpenMP directives, then OpenMP is probably the clear choice.

- However, if complex thread synchronization is needed then Pthreads will be easier to use.