Tree Search using MPI and static partitioning

- In the static parallelizations of tree search using Pthreads and OpenMP is taken straight from the second implementation of serial, iterative tree search.The MPI implementation would also require relatively few changes to the serial code, and this is, in fact, the case.

- In order to construct a complete tour, a process will need to choose an edge into each vertex and out of each vertex. Thus, each tour will require an entry from each row and each column for each city that's added to the tour, so it would clearly be advantageous for each process to have access to the entire adjacency matrix. Note that the adjacency matrix is going to be relatively small. For example, even if we have 100 cities, it's unlikely that the matrix will require more than 80,000 bytes of storage, so it makes sense to simply read in the matrix on process 0 and broadcast it to all the processes.

- Once the processes have copies of the adjacency matrix, the bulk of the tree search can proceed as it did in the Pthreads and OpenMP implementations. The principal differences lie in
  - partitioning the tree,
  - checking and updating the best tour, and
  - after the search has terminated, making sure that process 0 has a copy of the best tour for output.

Partitioning the tree

- In the Pthreads and OpenMP implementations, thread 0 uses breadth-first search to search the tree until there are at least thread count partial tours. Each thread then determines which of these initial partial tours it should get and pushes its tours onto its local stack.
- 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.

.

- Fortunately, there is a variant of MPI_Scatter, MPI_Scatterv, which can be used to send different numbers of objects to different processes. First recall the syntax of MPI_Scatter:

```
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  */);
```

- Process **root** sends **sendcount** objects of type **sendtype** from **sendbuf** to each process in **comm**. Each process in **comm** receives **recvcount** objects of type **recvtype** into **recvbuf**. Most of the time, **sendtype** and **recvtype** are the same and **sendcount** and **recvcount** are also the same. In any case, it's clear that the root process must send the same number of objects to each process.

- **MPI_Scatterv**, on the other hand, has syntax

```
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  */);
```

- The single **sendcount** argument in a call to **MPI_Scatter** is replaced by two array arguments: **sendcounts** and **displacements**.
- Both of these arrays contain **comm._sz** elements: **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**. The displacement is calculated in units of type **sendtype**.
  for example, if **sendtype** is **MPI_INT**, and **sendbuf** has type **int***, then the data that is sent to process **q** will begin in location

  sendbuf + displacements[q]

- In general, displacements[q] specifies the offset into sendbuf of the data that will go to process q. The "units" are measured in blocks with extent equal to the extent of sendtype.
- Similarly, MPI Gatherv generalizes MPI Gather:

```
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  */);
```

Maintaining the best tour

- When a process finds a new best tour, it really only 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. Also, when a process updates the best tour, it doesn't care what the actual cities on the former best tour were; it only cares that the cost of the former best tour is greater than the cost of the new best tour.

- During the tree search, when one process wants to communicate a new best cost to the other processes, it's important to recognize that we can't use **MPI_Bcast** for recall that 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. If it tries to use MPI Bcast, it will block in the call and never return, since it will be the only process that calls it. We need to arrange that the new tour is sent in such a way that the sending process won't block indefinitely.
- MPI provides several options. The simplest is to have 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're using a special tag defined in our program, NEW_COST_TAG. This will tell the receiving process that the message is a new cost–as opposed to some other type of message–for example, a tour.

- The destination processes can periodically check for the arrival of new best tour costs. We can't use MPI Recv to check for messages since it's blocking; if a process calls

  MPI Recv(&received cost, 1, MPI INT, MPI ANY SOURCE, NEW COST TAG, comm, &status);

- the process will block until a matching message arrives. If no message arrives—for example, if no process finds a new best cost—the process will hang.

- MPI provides a function that only checks to see if a message is available; it doesn't actually try to receive a message. It's called MPI_Iprobe, and its syntax is:

```
int MPI_Iprobe(
      int          source          /*in*/,
      int          tag             /*in*/,
      MPI Comm     comm            /*in*/,
      Int *        msg_avail_p     /* out*/,
      MPI Status*  status p        /* out*/  );
```

- It checks to see if a message from process rank **source** in communicator **comm** and with tag tag is available. If such a message is available, ***msg_avail_p** will be assigned the value true and the members of ***status_p** will be assigned the appropriate values.
- To check for a message with a new cost from any process, we can call

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

- If msg_avail is true, then we can receive the new cost with a call to MPI_Recv:

```
MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
            NEW_COST_TAG, comm, MPI STATUS IGNORE);
```

Tree Search using MPI and dynamic partitioning

- In an MPI program that dynamically partitions the search tree, we can try to emulate the dynamic partitioning that we used in the Pthreads and OpenMP programs.
- In those programs, before each pass through the main while loop in the search function, a thread called a boolean-valued function called ***Terminated.***
- When a thread ran out of work—that is, its stack was empty—it went into a condition wait (Pthreads) or a busy-wait (OpenMP) until it either received additional work or it was notified that there was no more work.
- In the first case, it returned to searching for a best tour.
- In the second case, it quit. A thread that had at least two records on its stack would give half of its stack to one of the waiting threads.

- When a process runs out of work, there's no condition wait, but it can enter a busy-wait, in which it waits to either receive more work or notification that the program is terminating. Similarly, a process with work can split its stack and send work to an idle process.
- It needs to "know" a process that's waiting for work so it can send the waiting process more work. Rather than simply going into a busy-wait for additional work or termination, a process that has run out of work should send a request for work to another process.
- If it does this, then, 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, and the process that has just entered ***Terminated*** has work, it can send part of its stack to the requesting process.
- If there is a request, and the process has no work available, it can send a rejection. Thus, when we have distributed-memory, pseudocode for our Terminated function can look something like the pseudocode shown in Program 10 below.

```
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 */
```

Prog10: Terminated function for a dynamically partitioned TSP solver that uses MPI

- ***Terminated*** begins by checking on the number of tours that the process has in its stack (Line 1); if it has at least two that are "worth sending," it calls ***Fulfill_request*** (Line 2).

- ***Fulfill_request*** checks to see if the process has received a request for work. If it has, it splits its stack and sends work to the requesting process. If it hasn't received a request, it just returns. In either case, when it returns from Fulfill request it returns from ***Terminated*** and continues searching.
- If the calling process doesn't have at least two tours worth sending, ***Terminated*** calls Send rejects (Line 5), which checks for any work requests from other processes and sends a "no work" reply to each requesting process.
- After this, ***Terminated*** checks to see if the calling process has any work at all. If it does—that is, if its stack isn't empty—it returns and continues searching. Things get interesting when the calling process has no work left (Line 9).
- If there's only one process in the communicator (comm._sz - 1), then the process returns from ***Terminated*** and quits. If there's more than one process, then the process "announces" that it's out of work in Line 11.
- Before entering the apparently infinite while loop (Line 13), we set the variable work request sent to false (Line 12). As its name suggests, this variable tells us whether we've sent a request for work to another process; if we have, we know that we should wait for work or a message saying "no work available" from that process before sending out a request to another process.
- The while(1) loop is the distributed-memory version of the OpenMP busy-wait loop.We are essentially waiting until we either receive work from another process or we receive word that the search has been completed.
- When we enter the while(1) loop, we deal with any outstanding messages in Line 14. We may have received updates to the best tour cost and we may have received requests for work. It's essential that we tell processes that have requested work that we have none, so that they don't wait forever when there's no work available.

- After clearing out outstanding messages, we iterate through the possibilities: . The search has been completed, in which case we quit (Lines 15–16). . We don't have an outstanding request for work, so we choose a process and send it a request (Lines 17–19).
- We do have an outstanding request for work (Lines 21–25). So we check whether the request has been fulfilled or rejected. If it has been fulfilled, we receive the new work and return to searching. If we received a rejection, we set work request sent to false and continue in the loop. If the request was neither fulfilled nor rejected, we also continue in the while(1) loop.

- ***My_avail_tour_count:***

The function My avail tour count can simply return the size of the process' stack. It can also make use of a "cutoff length." When a partial tour has already visited most of the cities, there will be very little work associated with the subtree rooted at the partial tour. Since sending a partial tour is likely to be a relatively expensive operation, it may make sense to only send partial tours with fewer than some cutoff number of edges.

- ***Fulfill_request:***

  If a process has enough work so that it can usefully split its stack, it calls ***Fulfill_request*** (Line 2). ***Fulfill_request*** uses MPI Iprobe to check for a request for work from another process. If there is a request, it receives it, splits its stack, and sends work to the requesting process. If there isn't a request for work, the process just returns.
- **Splitting the stack**

- A ***Split_stack*** function is called by ***Fulfill_request***. It uses the same basic algorithm as the Pthreads and OpenMP functions, that is, alternate partial tours with fewer than ***split_cutoff*** cities are collected for sending to the process that has requested work.
- The MPI version of ***Split_stack*** packs the contents of the new stack into contiguous memory and sends the block of contiguous memory, which is unpacked by the receiver into a new stack.
- MPI provides a function, MPI_Pack, for packing data into a buffer of contiguous memory. It also provides a function, MPI_Unpack, for unpacking data from a buffer of contiguous memory.

- their syntax is

```
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     */);

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     */);
```

- ***MPI_Pack*** takes the data in ***data_to_be_packed*** and packs it into ***contig_buf***. The ****position_p*** argument keeps track of where we are in ***contig_buf***. When the function is called, it should refer to the first available location in ***contig_buf*** before ***data_to_be_packed*** is added. When the function returns, it should refer to the first available location in contig buf after data to be packed has been added.
- ***MPI_Unpack*** reverses the process. It takes the data in ***contig_buf*** and unpacks it into ***unpacked_data***. When the function is called, ****position_p*** should refer to the first location in ***contig_buf*** that hasn't been unpacked. When it returns,

position_p should refer to the next location in ***contig_buf*** after the data that was just unpacked.

Send_rejects:

- The Send rejects function (Line 5) is similar to the function that looks for new best tours. It uses MPI Iprobe to search for messages that have requested work. Such messages can be identified by a special tag value, for example, WORK_REQ_TAG. When such a message is found, it's received, and a reply is sent indicating that there is no work available.

- Both the request for work and the reply indicating there is no work can be messages with zero elements, since the tag alone informs the receiver of the message's purpose. Even though such messages have no content outside of the envelope, the envelope does take space and they need to be received.

Distributed termination detection

- The functions Out of work and No work left (Lines 11 and 15) implement the termination detection algorithm. As we noted earlier, an algorithm that's modeled on the termination detection algorithm we used in the shared-memory programs will have problems. Suppose each process stores a variable ***oow,*** which stores the number of processes that are out of work. The variable is set to 0 when the program starts.
- Each time a process runs out of work, it sends a message to all the other processes saying it's out of work so that all the processes will increment their copies of ***oow***. Similarly, when a process receives work from another process, it sends a message to every process informing them of this, and each process will decrement its copy of oow. Now suppose we have three process, and process 2 has work but processes 0 and 1 have run out of work.

- Distributed termination detection is a challenging problem, and much work has gone into developing algorithms that are guaranteed to correctly detect it.

- Conceptually, the simplest of these algorithms relies on keeping track of a quantity that is conserved and can be measured precisely. Let's call it ***energy***, since, of course, energy is conserved.
- 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, and sending half to the process that's receiving the work.

- Since energy is conserved and since the program started with ***comm_sz*** units, the program should terminate when process 0 finds that it has received a total of ***comm_sz*** units.