Parallel Algorithms: Tree Search

Introduction

So far we have used deterministic communication patterns in our examples

The amount of work each process was to do was predetermined

In a real parallel computer system, it may be beneficial to dynamically allocate work to processes, so that good load balancing is achieved

We want to avoid idle processors

Tree Searches and Combinatorial Optimization

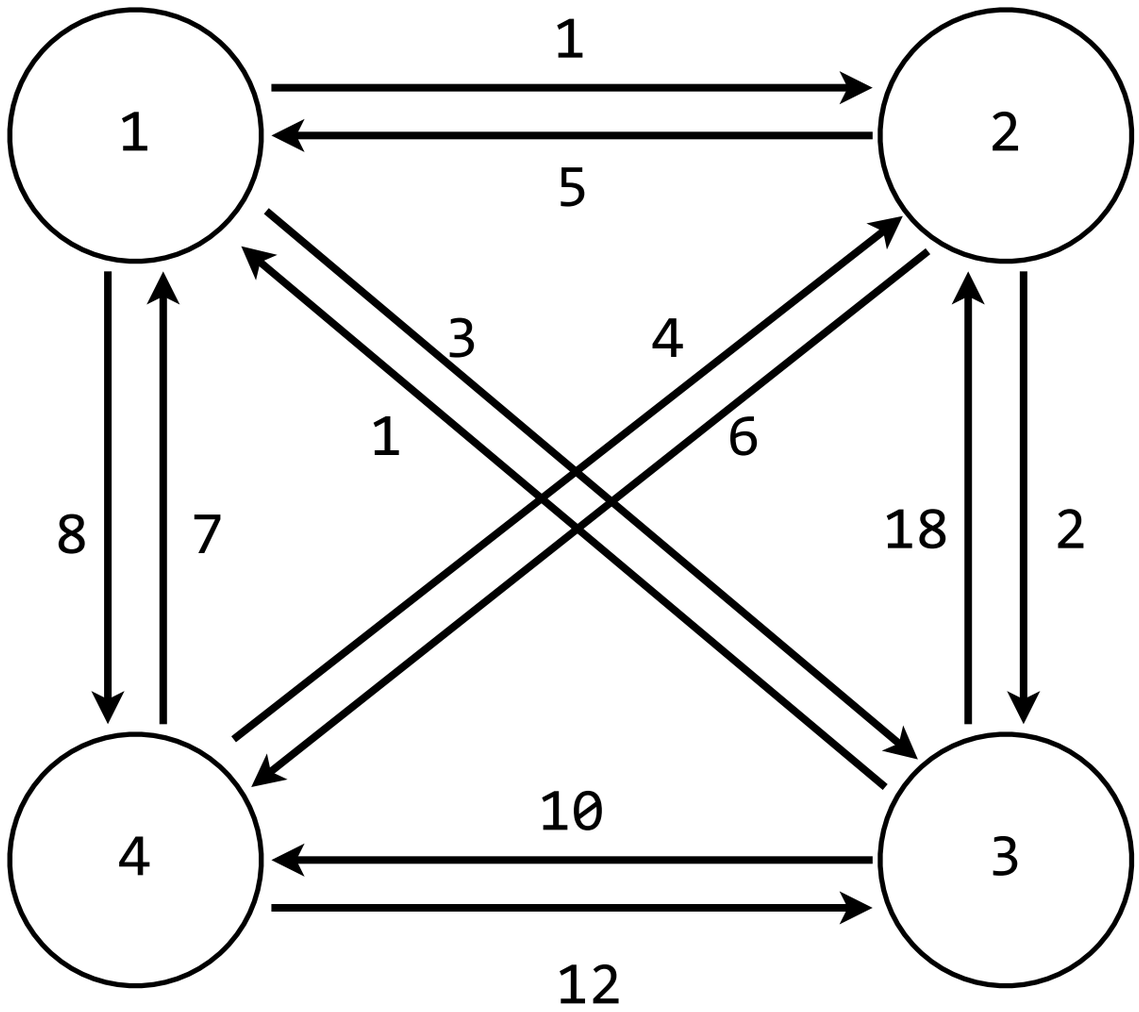
Most optimizations problems can be solved by searching a tree

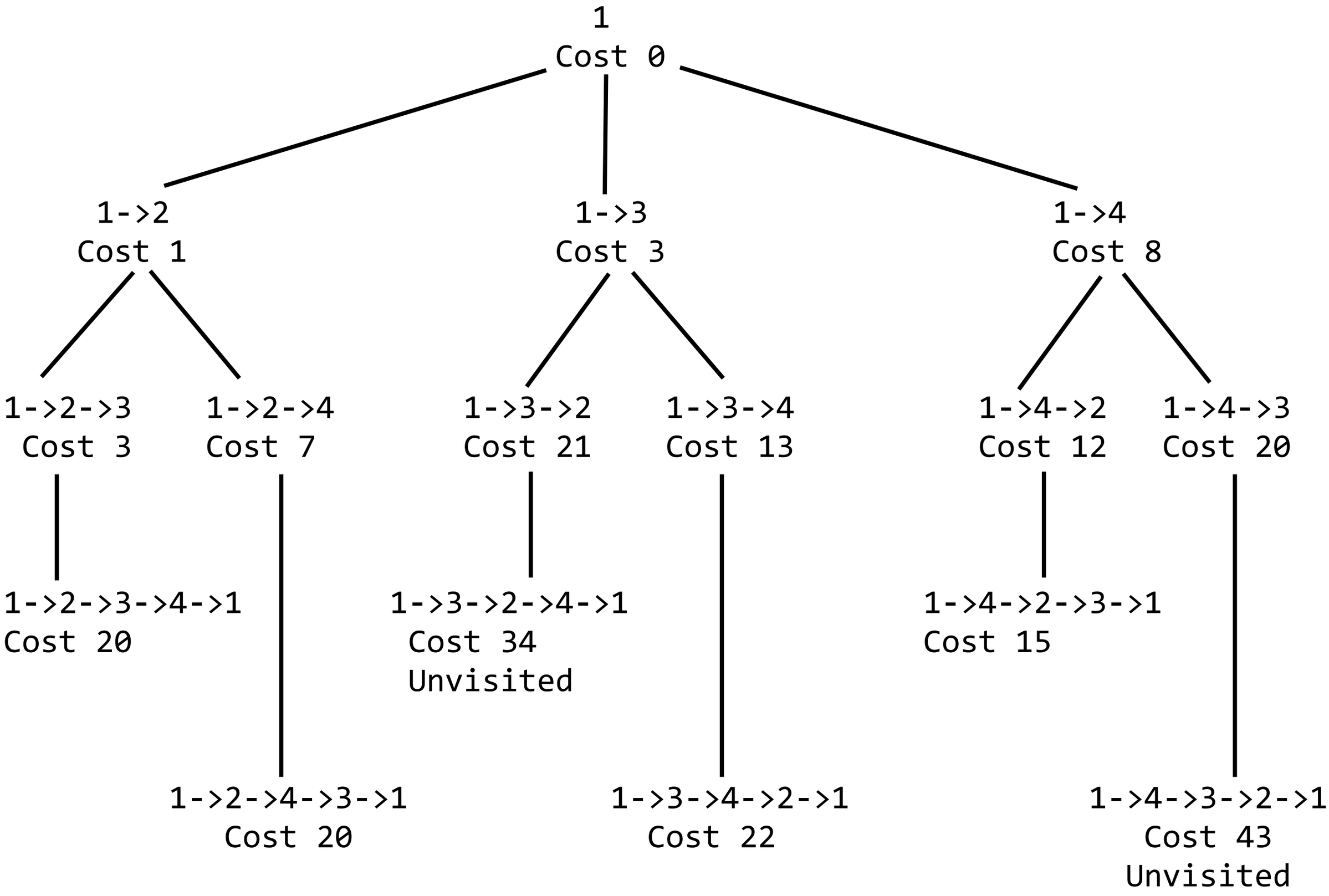
Example: A salesman has a list of cities to visit and a set of costs associated to travelling between each city. He wants to visit each city exactly once and return to starting city

NP-hard problem, which means we should use a brute force algorithm, as there is no general algorithm that does better in every instance

Four-vertex travelling salesman problem

Find cheapest route, where each city visited only once, except for the original vertex to which one returns at the end





Search tree for four-vertex travelling salesman problem

Depth-first search
Serial solution, uses recursion

```
/* Recursive depth-first search */
void Dfs_recursive(NODE_T node ) {
int i, num_children;
NODE_T child_list[MAX_CHILDREN];

Expand (node, child_list, &num_children);

for (i=0; i < num_children; i++)
    if (Solution(child_list[i])) {
        if (Evaluate(child_list[i]) < best_solution)
        best_solution = (Evaluate(child_list[i])
    } else if (Feasible(child_list[i])) {
          Dfs_recursive(child_list[i]);
    }

} /* Dfs_recursive */
```

Depth-first search
Serial solution, uses stack

```
/* Iterative depth-first search using a stack*/
void Dfs_stack(NODE_T root ) {
NODE_T node;
STACK_T stack;
/* Allocate empty stack */
Initialize(&stack);

/* Expand root, push children onto stack*/
Expand(root, stack);

while(!Empty(stack)){
  node = Pop(stack);
  if (Solution(node)) {
     if (Evaluate(node) < best_solution)
            best_solution = Evaluate(node);
  } else if (Feasible(node)} {
     Expand(node,stack);
  }
}
Free(stack);
} /* Dfs_stack */
```

Parallel Tree Search

We could simply distribute subtrees among processes, but then we run the risk that the tree is not balanced, so each process will do a different amount of work

Use a dynamic load balancing scheme instead, taking an approach build from the serial search using stack

On shared memory machine, we could have put the stack in shared memory, where each process can push nodes onto and pop nodes from the stack

On distributed memory machine, a single process could be responsible for managing the stack, and other process could access it through MPI.

Both approaches could lead to problems when many processes try to access the stack at the same time

It may be better to distribute the stack among the processes. If a process exhausts its stack, it can send requests to other processes for work.

Outline for parallel tree search

```
void Par_tree_search(
        NODE_T       root      /* in */,
        MPI_Comm     comm      /* in */) {

    NODE_T*  node_list;
    NODE_T   node;
    STACK_T  local_stack;

    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    /* Generate initial set of nodes, 1 per process */
    if (my_rank == 0) {
        Generate(root, &node_list, p, comm);
    }
    Scatter(node_list, &node, comm);

    Initialize(node, &local_stack);
```

```
do {
    /* Search for a while */
    Par_dfs(local_stack, comm);

    /* Service requests for work. */
    Service_requests(local_stack, comm);

    /* If local_stack isn't empty, return.           */
    /* If local_stack is empty, send                 */
    /* requests until either we get work, or we      */
    /* receive a message terminating program.        */
    } while(Work_remains(local_stack, comm));

    /* Get global best solution */
    Update_solution(comm);
    Print_solution(io_comm);
} /* Par_tree_search */
```

Par_dfs

```
void Par_dfs(
        STACK_T   local_stack  /* in/out */,
        MPI_Comm  comm         /* in     */) {
    int     count;
    NODE_T  node;
    float   temp_solution;

    /* Search local subtree for a while */
    count = 0;
    while (!Empty(local_stack) && (count < max_work)) {
        node = Pop(local_stack);

        if (Solution(node)) {
            temp_solution = Evaluate(node);
            if (temp_solution < Best_solution(comm)) {
                Local_solution_update(temp_solution, node);
                Bcast_solution(comm);
            }
        } else if (Feasible(node, comm)) {
            Expand(node, local_stack);
        }
        count++;
    }  /* while */

} /* Par_dfs */
```

Program needs to check message queue to see if new best solutions has been found

In other words, it needs to be able to do something like

```
if (there is a message for me)
   MPI_Recv(message);
   Do_something();
else
   Do_something_else();
```

This functionality is provided by MPI_Iprobe

This function searches for an incoming message that matches source, tag and comm. If it finds the message that matches, it returns flag= TRUE and status

Status will contain information about the number of elements in the message, thus MPI_Iprobe can be used to receive a message of unknown size

Source and tag can be wildcards

If multiple messages match the call, information is returned on the message that would be received by call to MPI_Recv at that point

MPI_Iprobe

```
int MPI_Iprobe( int source, int tag, MPI_Comm comm, int *flag,
                MPI_Status *status )
```

source - source rank or MPI_ANY_SOURCE

tag - tag value or MPI_ANY_TAG

comm - communicator

flag - true if message with specified parameters available

status - status object (contains useful information about message)

If MPI_Recv is called with the same source/tag/comm as used in MPI_Iprobe , after MPI_Iprobe returns true, then it will receive the message MPI_Iprobe found

Careful with wildcards, as their use can lead to synchronization problems, for example if some other process receives the message after MPI_IProbe executed but before MPI_Recv starts, then MPI_Recv will get stuck

Can use tags to make sure messages go to right processes

Service_requests

```
void Service_requests(
         STACK_T   local_stack  /* in/out */,
         MPI_Comm  comm         /* in     */) {
    STACK_T send_stack;
    int     destination;

    while (Work_requests_pending(comm)) {

        destination = Get_dest(comm);

        if (Nodes_available(local_stack)) {

            Split(local_stack, &send_stack);
            Send_work(destination, send_stack, comm);

        } else {
            Send_reject(destination, comm);

        }
    }

} /* Service_requests */
```

Work_remains

```
int Work_remains(
        STACK_T    local_stack  /* in/out */,
        MPI_Comm   comm         /* in     */ ) {

    int       work_available;
    int       work_request_process;
    int       request_sent;
    if (!Empty(local_stack)) {
        return TRUE;
    } else {
        Return_energy(comm);
        request_sent = FALSE;
```

```
while (TRUE) {
    Send_all_rejects(comm);
    if (Search_complete(comm)) {
        if (request_sent) Cancel_request();
        return FALSE;
    } else if (!request_sent) {
        work_request_process = New_request(comm);
        Send_request(work_request_process, comm);
        request_sent = TRUE;

    } else if (Reply_received(work_request_process,
            &work_available, local_stack, comm)) {
                if (work_available) {
                    return TRUE;
                } else {
                    request_sent = FALSE;
                }
            }
        } /* while (TRUE) */
    }
} /* Work_remains */
```

Distributed Termination Detection

Important topic in computer science

One approach uses the idea of “energy”, i.e. indestructible resource distributed among processors (“energy” is conserved)

1. At the start, process 0 has all the energy, total energy is 1

2. Energy is transferred only:
 a. during initial distribution
 b. when a process fills out a request for work
 c. when a process runs out of work

3. During initial distribution, energy divided into p parts, each process gets 1/p

4. When process successfully requests work, it keeps half the energy and transfers half to process that requested work

5. When process runs out of work, it transfers whatever energy is left to process 0

Whenever process completes its work and finds it has energy 1, it can broadcast a termination message to all processes
Cannot use floating point numbers to store fractions, use integer pairs