

# PROGRAMMING ASSIGNMENT TWO

This assignment is due at class time Monday, 11 October 2004 and is to provide you with experience doing recursion in MPI and a tree search with optimization of an objective function.

This assignment is in three parts.

1. parallelize a tree-structured computation to generate the permutations on  $n$  things, for  $5 \leq n \leq 13$ ;
2. expand upon the tree to solve (by brute force) the Travelling Salesman Problem;
3. improve upon the brute force method by pruning your TSP search in parallel.

## Part 1: Generate Permutations

In `/home/csce590/public/SamplePrograms/Dperms` you will find a single-processor program that generates the permutations on  $n$  letters (actually, I use numbers but I will refer to them as letters) by means of a brute force approach. You may choose to write your own basic permutation-generating program, but you may also start with this program. The relatively mindless permutation-generation program should be viewed merely as a vehicle for doing a tree search. One advantage of a search doing permutations is that we know what the right answers are ( $5! = 120$ ,  $6! = 720$ ,  $7! = 5040$ , and so forth).

The first part of this assignment is to parallelize the generation of permutations. The simplest way to do this is to expand the recursively-called function into two levels of such functions. The program as written is a depth-first search; the easiest way to parallelize this is to expand at the first level into a breadth-first search and then to do a depth-first search beyond that. That is, if you are doing permutations on 8 letters, and you have 4 processors, let processor 0 handle the permutations whose first letter is A and E, processor 1 handle those whose first letter is B and F, processor 2 handle C and G, and processor 3 handle D and H. For permutations on less than 16 letters, you can use up to 16 processors mostly efficiently. For most situations, the number of processors and the permutation length will be incommensurable, so the last processors may have less work than the first processors. With too many processors and a short permutation (perhaps 7 processors and permutations of length 6), the last processors will do no work at all.

Input to this program need only be the integer  $n$ , with legal values from 5 through 16. Note that in order to count the right number of permutations generated, you will need to use `long long` variables, since the count is greater than  $2^{31} - 1$  and therefore too big for a 32-bit integer. Note also that you can't return a `long long` value in MPI; you'll have to split it up, send the value, and then combine on the head node.

Just for reference, I include my counts and timings for generating permutations. The times are single processor times. **You do not want** to do the larger problems on a single processor. This will clog up the machine for other people, and do you no good. Test on the smaller sizes, and do the larger ones only on multiple machines after you have verified that your code works. The “Counts” column contains the outputs from my program; this is the total number of possible nodes explored.

$n$	Perms	Counts	Time (seconds)
5	120	1630	0.00
6	720	11742	0.00
8	40320	876808	0.02
10	3628800	98641010	2.36
11	39916800	1193556232	27.51
12	479001600	15624736140	350.49
13	6227020800	220048367318	4803.70

## Part 2: Travelling Salesman

Now that you can generate all the permutations on  $n$  things, for  $n$  in some reasonable ranges, you can use these permutations to solve (again by brute force) a travelling salesman problem for  $n$  cities. You will find in the directory mentioned above some distance matrices for 5 through 13 cities. The individual distances have been chosen to be positive integers less than or equal to 1000. (Note that the distances are *not* symmetric in that the distance from city  $J$  to city  $K$  is not the same as the distance from city  $K$  to city  $J$ . This was done to make the problem more interesting. If the distances are symmetric, then you will have multiple permutations with the same minimum distance. Certainly all circular shifts of a minimal permutation will always be minimal, but by making the matrix asymmetric you will at least cut out the reversals.)

Your program should keep track as a permutation develops of the distance associated with it and record the minimal distance. Note that the lack of symmetry suggests that different processors, exploring different parts of the search tree, will have different minima that they encounter, so the final reduction step at the end is meaningful and not redundant.

At this stage, your program is permitted to search every part of the tree space to the bitter end and return only the minimum encountered.

I get the following for minimal distances. If my code is correct (not always a good assumption), these are the answers.

$n$	Minimum Distance
5	1891
6	1607
8	1915
10	2517
11	2282
12	2136
13	1788

## Part 2: Pruned Travelling Salesman

Now comes the hard part. The first two parts of this are load-balanced in that even though in part 2 you will encounter a different minimum value, all processors will execute the same number of steps (unless you have an unbalanced number of letters and processors).

In this step your processors are to share periodically the minimum that they have so far encountered and to prune their search accordingly. That is, if you have expanded a given partial permutation, and the total distance to date is larger than your current best minimal distance, you should abort the tree expansion and prune your search accordingly. By “periodically” is meant that your processors should “every so often” broadcast their current minima back to the head node so that the head node can broadcast back to all processors the current global minimum. This will permit further pruning of the search space.

A rule of thumb is that you should pause to share current minima about every one percent of the way through the search, at least for the larger searches ( $n = 10, 11, 12, 13$ ). I would suggest using a variable to control the definition of “every so often” and make it small for the small programs so you can debug carefully.

One aspect of this program that could cause you problems is this: If you pause to share minima every  $K$  steps, with  $K$  fixed, you will have to use a barrier to synchronize the reduction and subsequent broadcast.

This could cause problems if some process actually finishes and exits the loop, because then it won't be able to participate in the barrier and thus the rest of the processes will never be able to leave the code with the barrier.