Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 12, 2016

Assignment 5: TSP with MPI

Solution Statement

We completed the assignment as specified, achieving a suboptimal best cost of 7580 along the path: 1 9 4 5 12 13 17 11 3 7 2 14 8 6 15 16 10 1. Our best time achieved for this solution was 127 seconds with 50 processors, with a speedup of 11.05 over the sequential implementation.

Problem Description

Using MPI, the goal of this assignment was to parallelize a solution to the Traveling Salesman Problem for small instances with the branch-and-bound algorithm. Rather than focusing on optimizing the TSP solver, special attention was to be dedicated to implementing branch and bound in a way that adequately balances the load. Results for the performance of our TSP solver were to be reported when run both sequentially and in parallel with 10, 50, and 100 processors. Additionally, the TSP solver was expected to report the optimal path as well as its respective cost.

Approach and Design

The master maintains a stack of int vectors corresponding to partially developed paths to be considered by worker processes. A stack was chosen in order to perform a Depth First Search algorithm while traversing the search space. To speed up the initial search spread in the tree of possible paths being constructed for exploration by the worker processes, the master process pre-generates the first three layers of the search space. The master then sends the first batch of work to each worker process and increments a counter that tracks the number of jobs currently pending with worker processes. The logical body of the master process is then to continuously receive responses from the worker processes and send them new jobs until both the stack of

paths to be searched is empty and there are no longer any pending jobs. The message received by the master process is a vector of ints with the best local path found by the worker process and the associated score appended to the end. The master compares the appended score against the current best score and updates the current best score and path if appropriate. Once the stack is empty and there are no longer any pending jobs amongst the worker processes, the master sends each worker a message containing a die tag, indicating that they should terminate. Finally, the master outputs the best cost, the best path, and the elapsed time of the computation before finishing.

Each worker process maintains its own local copy of the graph provided in the city file for the problem instance. The worker initially populates its local stack with a call to branch(), which implements the branch and bound strategy iteratively by increasing the depth of the partial search space by one on each call. While the worker's local stack is not empty, it calculates the current partial sum and lower bound for each path on the stack. If the partial sum and the lower bound are both less than the local best, the worker again calls branch on the current path being considered.

- main() - Sets up the MPI instance, builds a record of all cities for assessing the progress of partially explored paths, begins the master function if the rank 0 process or the worker function otherwise, and cleans up the MPI instance.
- master() - Contains the functionality for the master process, which acts as a manager for the worker processes by maintaining a stack of jobs to be completed, sending these jobs to workers when requested, and receiving completed messages from workers. The master

Assignment 5: TSP with MPI

maintains the current best score and path and outputs the results once its stack is empty and there are no longer any pending jobs.

- worker() - Contains the functionality for the worker process, which receives jobs from the master and reports back the best local path found and its corresponding score via the branch and bound strategy. The worker continues to request additional jobs as needed until the search space has been exhausted.
- branch() - Takes a vector of ints representing the current path for consideration and applies the branch and bound strategy, increasing the depth of the partially explored path by one, adding the new partial paths to the worker's local stack, and updating the best path as appropriate.
- inCities() - A helper function that reads the supplied city text file into a 2D array of ints.
- printCities() - A helper function that prints the 2D array of int values loaded from the supplied city text file for debugging purposes.

The program is designed to dynamically balance the workload amongst the worker processes in a way that is not dependent on the number of workers. More specifically, beyond the initial assignment of jobs to worker processes, additional jobs are only assigned to worker processes on a per request basis. Thus, the only time a worker process will be idly waiting for additional work from the master process is when its local stack is empty (meaning there are currently no additional paths to be investigated) or the communication latency has overtaken the problem and the master thread too bogged down by worker process requests to immediately send another job. Lastly, the program was compiled using the MPI gcc wrapper: mpiCC with the -O3 optimization flag, and was run on the UVA Rivanna machines.

Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 12, 2016

Assignment 5: TSP with MPI

Results and Analysis

For the 17 city problem instance as defined in the supplied file, the TSP solver found the following, optimal solution. We noticed that our solution for the cost is slightly different than the one provided in the assignment writeup. A likely reason is that we assumed the first and last city is "1", which may have caused the real solution to be pruned off prematurely, since the lower bound of each path is computed starting from city "1".

Best Path: 1 9 4 5 12 13 17 11 3 7 2 14 8 6 15 16 10 1

Cost: 7580

The runtime of the TSP solver for each execution case on the 17 city problem instance can be seen in the following tables.

TSP Solver Performance: Forwards Search		
# Processors	**Average Time (s)**	**Average Speedup**
Sequential	2122	--
10	305	6.96
50	411	5.16
100	679	3.13

TSP Solver Performance: Backwards Search		
# Processors	**Average Time (s)**	**Average Speedup**
Sequential	1403	--
10	165	8.50
50	127	11.05
100	133	10.55

Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 12, 2016

Assignment 5: TSP with MPI

We have provided results for two cases of the solution to evaluate the impact the direction of examining the search space has on performance. The forwards case begins constructing the path by considering the edge between the first and second city, in other words, the initial paths for the forwards case are of the form [1, 2, ...]. Alternatively, the paths for the backwards case begin by considering the edge between the first city and the final city (16 for the 17 city problem instance); thus, the initial paths for the backwards case are of the form [1, 16, ...]. Since our implementation uses the branch and bound strategy, the solution for both cases is of course optimal; however, the performance under the different processor resource constraints is quite different.

As can be surmised from the sequential performance, the backwards case outperforms the forwards case. This is not surprising, as the backwards case finds a better path on its first run than the forwards case, and thus pruning of the search space occurs faster and limits the number of processors working within the minutia of local variation. For each of the parallel runs under the forwards case, the performance deteriorates as the number of processors increases - though all parallel runs still significantly outperform the sequential run. Again, this is likely because there are more workers that can exhaust the master's stack before a messages from the workers further constrain viable paths on the stack. In other words, by increasing the number of workers, we are decreasing the breadth of the branch and bound strategy's ability to prune off the search space.

Contrary to the forwards case, the 10 processor run for the backwards case is worse than the 50 and 100 processor runs. This is likely because the phenomenon responsible for the

deteriorating performance of the forwards case when the processor size is increased does not emerge until some boundary threshold is reached beyond 10 processors for the backwards case. The three parallel runs for the backwards case produce an exceptional speedup, both over the sequential run and the runs of the forwards case, that does not significantly vary between runs. It seems evident that for this particular problem instance, the backwards case would not see a dramatic deterioration in performance as the number of processors continues to scale - though, further parallelism would not be recommended as the performance would not improve and justify the increased resource cost. It is clear from the performance results of the forwards and backwards cases considered in our analysis that the branch and bound strategy is highly sensitive (as one might expect) to the direction in which the search space is explored.

Conclusion

While our initial results were not encouraging, additional effort to optimize the load balancing of the jobs, combined with offloading more overhead to the workers instead of the master process yielded quite compelling and satisfying performance improvements. These results demonstrate how NP-complete problems of considerable real world relevance can be significantly aided - whether through optimal, outright computation, or acceptable heuristics to produce results under reasonable, real-world time constraints. In conclusion, this assignment constituted an engaging toy problem that further concretized our conceptual understanding and ability to develop parallel solutions with MPI.

On our honor as students, we have neither given nor received aid on this assignment.

Philip & Karen

Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 12, 2016

Assignment 5: TSP with MPI

```
// tsp.cpp
// Code skeleton for master/worker MPI setup originally from:
//
http://www.hpc.cam.ac.uk/using-clusters/compiling-and-development/parallel-programming-mpi-exa
mple

/****** OVERALL ALGORITHM ********

	1)	Master pre-generates the first 3 layers of the "tree"
	[2 3 4 0 0 0 0 …]
	[2 3 5 0 0 0 0 …]
	…
	[16 17 15 0 0 …]
	For 17 cities, I think this is (15*14*13) = 2,730 initial items of work
	We don't need to generate items starting with 17 (e.g. [17 2 3 0 0…]), because of
"reversals", that is, anything starting with 17 will eventually get generated by the other
branches in reverse

	2)	Master gives Worker a job (always from 3rd layer) and the current global best

	3)	Workers set their local best as global best, then branch on the item of work
given (going into the 4th layer of the tree), then bound on those items (using the local
best), and repeatedly branch and bound on its own local stack, updating its local best as it
goes

	4)	When a Worker's local stack is empty (i.e. branching any further results in a
solution having a lower bound greater than the local best, so it either stops branching, or
reaches leaf nodes)
	Then the Worker sends their local best (and local best path) back to Master

	5)	Master updates the global best, and goes back to step 2

	6)	Terminates when Master's stack is empty

*********************************/
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <climits>
#include <stack>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

// Different tags
#define WORKTAG    1
#define DIETAG     2

// City info
#define NUM_CITIES 17               // change depending on city file
char filename[] = "city17.txt";     // change depending on city file
#define NUM_CITIES_1 NUM_CITIES-1

// Function prototypes
void master();
void worker();
```

Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 12, 2016

Assignment 5: TSP with MPI

```
void inCities();
void printCities();
void branch(vector<int> work);

// Globals
int myrank, ntasks;
double elapsed_time;
int cities[NUM_CITIES][NUM_CITIES];
vector<int> all_cities;

int main(int argc, char *argv[]) {
	// Initialize
	MPI_Init(&argc, &argv);
	// Make sure everyone got here
	MPI_Barrier(MPI_COMM_WORLD);

	// Get current time
	elapsed_time = - MPI_Wtime();
	// Get myrank
	MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
	// Get total number of tasks
	MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

	// Determine if master or worker
	if (myrank == 0) {
		master();
	} else {
		worker();
	}

	// Cleanup
	MPI_Finalize();
	return 0;
}

//**************************************//
//    Stuff for master (rank 0) to do      //
//**************************************//

void master() {
	// MPI Status used for figuring out message sources
	MPI_Status status;

	// Variables about workers
	int rank, local_best;

	// Master's global best cost and path
	int best = INT_MAX;
	vector<int> best_path(NUM_CITIES_1,0);

	// Master's stack
	stack< vector<int> > mystack;

//-----------Set up the Queue-----------//
	// Pre-generate 3 "layers" of the "tree" or search space

	// First layer, can leave off the last city because of reversals
	//for (int i = NUM_CITIES_1; i > 1; i--) { // do this line for going "backwards" [ 16
17 15 ...]
	for (int i = 2; i < NUM_CITIES; i++) { // do this line for going "forwards" [ 2 17 16
... ]
```

Assignment 5: TSP with MPI

```
                vector<int> tmp(NUM_CITIES_1,0);
                tmp[0] = i;
                // Second layer
                for (int j = 2; j <= NUM_CITIES; j++) {
                        // make sure we didn't already use j as i
                        if (i != j) {
                                tmp[1] = j;

                                // Third layer
                                for (int k = 2; k <= NUM_CITIES; k++) {
                                        // make sure we didn't already use k as j or i
                                        if (k != j && k != i) {
                                                tmp[2] = k;

                                                // put on stack
                                                mystack.push(tmp);
                                        }
                                }
                        }
                }
        }

//------------Seed workers--------------//
        vector<int> work(NUM_CITIES_1,0);
        int pending_jobs = 0; // keep track of jobs that are out there in case
                                        // stack is empty before all jobs are done

        // Loop through all the workers once
        for (rank = 1; rank < ntasks; ++rank) {
                // Get next thing of work
                work = mystack.top();
                mystack.pop();

                // Add current best (infinity) to back
                work.push_back(INT_MAX);

                MPI_Send(&work[0], /* work item */
                NUM_CITIES,        /* length */
                MPI_INT,           /* data items are integers */
                rank,              /* destination process rank */
                WORKTAG,           /* it's work, not death */
                MPI_COMM_WORLD);

                pending_jobs++; // increment counter

                // This 'if' generally shouldn't happen with a lot of cities
                // but it's here for testing (and using smaller amounts of workers)
                if (mystack.empty()) {
                        break;
                }
        }

        printf("Seeded workers\n");

//----Receive more requests and respond-----//
        vector<int> message(NUM_CITIES,0);

        while (!mystack.empty() || pending_jobs != 0) { //queue is not empty

                message.resize(NUM_CITIES);
```

Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 12, 2016

Assignment 5: TSP with MPI

```
// Receive request
MPI_Recv(&message[0],/* message buffer */
NUM_CITIES,          /* length */
MPI_INT,             /* type int */
MPI_ANY_SOURCE,        /* receive from any sender */
MPI_ANY_TAG,           /* any type of message */
MPI_COMM_WORLD,        /* always use this */
&status);              /* received message info */

pending_jobs--; // decrement counter

// Fetch what this worker's local best was
local_best = message.back();
message.pop_back();

// Potentially update global best
if (local_best < best) {
        best = local_best;
        best_path = message;
        cout << "UPDATED BEST: " << best << " [ ";
        for (int i = 0; i < best_path.size(); i++) {
                cout << best_path[i] << " ";
        }
        cout << "]" << endl;
}

// Check whether to terminate or continue
// In case stack is empty but there are more pending jobs
// We don't want to pop if there is nothing on the stack!
if (mystack.empty()) {
        if (pending_jobs == 0) {
                break;
        }
        else {
                continue;
        }
}

// Get next thing of work
work = mystack.top();
mystack.pop();

// Add current best to back
work.push_back(best);

// Print for testing
// cout << "Work: ";
// for (int i = 0; i < work.size(); i++) {
//     cout << work[i] << " ";
// }
// cout << endl;

// Respond with more work
MPI_Send(&work[0], /* work item */
NUM_CITIES,        /* length */
MPI_INT,           /* data item is an integer */
status.MPI_SOURCE, /* destination process rank */
WORKTAG,           /* it's work not death */
MPI_COMM_WORLD);

pending_jobs++; // increment job counter
```

Assignment 5: TSP with MPI

```
	}

	printf("Telling workers to exit\n");

//--------Tell workers to die----------//
	for (rank = 1; rank < ntasks; ++rank) {
		work.resize(NUM_CITIES);
		// use the DIETAG wow
		MPI_Send(&work[0], NUM_CITIES, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
	}

//---------PRINT FINAL OUTPUT----------//
	printf("Best: %d\n",best);

	cout << "Best path: 1 ";
	for (int i = 0; i < best_path.size(); i++) {
		cout << best_path[i] << " ";
	}
	cout << "1 " << endl;

	// Get elapsed time
	elapsed_time += MPI_Wtime();
	printf("RESULT: \n %f\n",elapsed_time);
	fflush(stdout);
}


//****************************************//
//   Stuff for worker (rank != 0) to do    //
//****************************************//

// Globals for worker
int local_best = INT_MAX;
vector<int> local_best_path;
stack< vector<int> > localstack;
int lb; // lower bound
int potential_best; // local bests received from other nodes


void worker() {
	// Set up all_cities vector [ 2 3 4 5 ... 17 ]
	for (int i = 2; i <= NUM_CITIES; i++) {
		all_cities.push_back(i);
	}
	// Load city data
	inCities();
	// A copy of cities to manipulate later for 'lower bound' computation
	int cities_copy[NUM_CITIES][NUM_CITIES];

	// Partial sum
	int sum = 0;
	// Next work item
	vector<int> work;
	// MPI Status wow no way
	MPI_Status status;

//----------Continuously Recv work----------//
	while(true) {
		// Reset work
		work.resize(NUM_CITIES);
```

Assignment 5: TSP with MPI

```
		// Receive work
		MPI_Recv(&work[0], NUM_CITIES, MPI_INT, 0, MPI_ANY_TAG,
		MPI_COMM_WORLD, &status);

		// Check for die tag
		if (status.MPI_TAG == DIETAG) {
			return;
		}

		// Update local best with global best
		local_best = work.back();
		work.pop_back();

		// BRANCH on 'work'
		branch(work);

//-----------------DO WORK-----------------//
		while (!localstack.empty()) {
			// Get next thing of work
			work = localstack.top();
			localstack.pop();

			// Reset result
			sum = 0;
			// Re-copy the copy of cities
			// Used for computing lower bounds
			for (int i = 0; i < NUM_CITIES; i++) {
				for (int j = 0; j < NUM_CITIES; j++) {
					if (i == j) {
						cities_copy[i][j] = INT_MAX;
					}
					else {
						cities_copy[i][j] = cities[i][j];
					}
				}
			}

			// Compute partial sum
			// Assume begins and ends with 1

			// Beginning
			sum += cities[0][work[0]-1];
			// Mark edge as used in cities
			cities_copy[0][work[0]-1] = INT_MAX;
			cities_copy[work[0]-1][0] = INT_MAX;

			// End
			// Only do end edge if it's not a zero
			if (work[NUM_CITIES_1-1] != 0) {
				sum += cities[0][work[NUM_CITIES_1-1]-1];
			}
			// Figure out how many zeroes there are and where they start
			int zeroes = 0;
			int zero_idx = 0;

			// Loop over the rest
			for (int i = 0; i < NUM_CITIES_1-1; i++) {
				int val1 = work[i];
				int val2 = work[i+1];
				// Make sure neither is a zero
				if (val1 != 0 && val2 != 0) {
```

Assignment 5: TSP with MPI

```
                        sum += cities[val1-1][val2-1];

                        // Mark this edge as being used in copy of "cities"
                        cities_copy[val1-1][val2-1] = INT_MAX;
                        cities_copy[val2-1][val1-1] = INT_MAX;
                }
                // Otherwise count zero stuff
                else if (val1 != 0 && val2 == 0) {
                        zero_idx = i;
                        zeroes++;
                }
                else {
                        zeroes++;
                }
        }

        // If partial sum is < local best
        if (sum < local_best) {
                // Compute lower bound
                vector<int> work_copy(work);

                // Compute the lower bound by adding up the best edges
                // for all the edges that are left over
                lb = sum;
                // For each zero
                for (int i = 0; i < zeroes; i++) {
                        int val1 = work_copy[zero_idx+i];
                        int least = INT_MAX; // min of row val1

                        int val2 = 0;

                        for (int j = 0; j < NUM_CITIES_1; j++) {
                                int next = cities_copy[val1-1][j];
                                if (next < least) {
                                        val2 = j+1;
                                        least = next;
                                }
                        }

                        // Set edge (val1,val2_idx) as INF
                        cities_copy[val1-1][val2-1] = INT_MAX;
                        cities_copy[val2-1][val1-1] = INT_MAX;

                        // Write to work_copy
                        work_copy[zero_idx+i+1] = val2;

                        // Add the least value in the row to the lower bound
                        lb += least;
                }

                // If lower bound < local best,
                // potentially update local best, and BRANCH on 'work'
                if (lb < local_best) {
                //-------Branch on 'work'------//
                        branch(work);
                }
                // else (lb >= local_best) so don't branch on 'work'
        }
        // else (sum >= local_best) so don't branch on 'work'
} // END WHILE
```

Assignment 5: TSP with MPI

```
                // Since the local stack is empty,
                // Send back local best path and local best, asking for more work
                vector<int> message(local_best_path);
                // Append the local best to the path
                message.push_back(local_best);

                // Send local results and a request for more work
                MPI_Send(&message[0], NUM_CITIES, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        fflush(stdout);
}


//**************************************//
//           Helper Functions           //
//**************************************//

// Branch function!
void branch(vector<int> work) {

        // Copy work
        vector<int> tmpwork(work);
        // Make another copy to sort
        vector<int> tmpworksorted(tmpwork);

        // We're gonna diff
        vector<int> diff(NUM_CITIES_1,0);
        vector<int>::iterator it;

        // Sort copy of copy of work
        sort(tmpworksorted.begin(),tmpworksorted.end());

        // Compute the difference between all_cities and work
        it = set_difference(all_cities.begin(), all_cities.end(), tmpworksorted.begin(),
tmpworksorted.end(),diff.begin());
        // Resize to actual size
        diff.resize(it-diff.begin());

        // If full candidate solution, update local best
        if (diff.size() == 0) {
                local_best = lb;
                local_best_path = work;
        }
        // If only 2 entries are left, we can just fill them in
        // And push 2 more work pieces
        else if (diff.size() == 2) {
                // First item
                vector<int> new_work(tmpwork);
                new_work[NUM_CITIES_1-2] = diff[0];
                new_work[NUM_CITIES_1-1] = diff[1];

                localstack.push(new_work);

                // Second item
                vector<int> new_work2(tmpwork);
                new_work2[NUM_CITIES_1-2] = diff[1];
                new_work2[NUM_CITIES_1-1] = diff[0];

                localstack.push(new_work2);
        }
        else {
```

Assignment 5: TSP with MPI

```
                // Fill in the next slot with everything in diff
                for (int i = 0; i < diff.size(); i++) {
                        vector<int> new_work(tmpwork);
                        new_work[NUM_CITIES_1-diff.size()] = diff[i];

                        localstack.push(new_work);
                }
        }
}

// Read the cities input file into an array
void inCities() {
        ifstream inputFile;
        inputFile.open(filename);
        for (int row = 0; row < NUM_CITIES; row++) {
            for (int col = 0; col < NUM_CITIES; col++) {
               inputFile >> cities[row][col];
            }
        }
}

// Print the cities array for debugging
void printCities() {
        cout << "Rank: " << myrank << endl;
        for (int row = 0; row < NUM_CITIES; row++) {
            for (int col = 0; col < NUM_CITIES; col++) {
               cout << cities[row][col] << " ";
            }
            cout << endl;
        }
}
```

Run.sh

```
#!/bin/bash
module load openmpi/gcc
# Re-compile the program
mpiCC -O3 tsp.cpp -o tsp
# Submit job
sbatch execute.sh
```

Execute.sh

```
#!/bin/bash
#SBATCH --ntasks=50
#SBATCH --time=00:30:00
#SBATCH --exclusive
#SBATCH --partition=training
#SBATCH --account=parallelcomputing
#SBATCH --job-name=tsp17
#SBATCH --output="out/out_tsp17.txt"
module load openmpi/gcc
mpiexec ./tsp
```