National University of Computer & Emerging Sciences, Islamabad

Fall-2023

Parallel and Distributed Computing

Assignment No.2

Submitted By:

Name: Syeda Areeba Nadeem

Section: AI-K

Roll No: 21I-0307

Submitted To:

Dr. Qaiser Shafi

Parallel Algorithm Design:

The algorithm utilizes the following parallel approach to solve the Traveling Salesman Problem (TSP) using the MPI.

Permutation Calculation:

Given the number of cities or vertices, along with their costs in the adjacency matrix, the total number of possible combinations is calculated, excluding the source city. This results in V-1 possible combinations, where V is the number of vertices or cities. Permutations per process are then calculated to distribute them across each process. Load balancing is achieved by dividing permutations equally among processes, with a redistribution of the remainder. The processes then determine the minimum costs and paths, which are subsequently gathered at the root process to determine the overall minimum cost and path.

Justification:

Various serial approaches for solving the Traveling Salesman Problem (TSP), such as dynamic programming and branch and bound, face challenges in parallelization due to extensive dependencies, leading to high overhead caused by communication and synchronization. Alternatively, the serial permutation generation for TSP takes (V-1)! time, placing a heavy load on a single processor. Parallel processing of this approach minimizes communication, achieving a high 6computation-to-communication ratio and producing efficient results in a timely manner.

Parallel Search:

Permutations of cities are generated in parallel across processes using a lexicographically (i.e. increasing numerical order) ordered approach. Each process independently evaluates a subset of permutations given to it. The search space is divided among processes to distribute the computational workload efficiently.

Inter-node Communication:

Inter-node communication is facilitated through MPI functions for initialization, ranking, size determination, and data gathering. The ***MPI_Gather*** function is employed to collect the minimum costs and paths from all processes to the root process (rank 0). Communication overhead is kept minimal, with each process working independently on its assigned permutation subset.

Work Load Division and Balancing

Work: *Generating different permutations, calculating its cost and storing the minimum of them.*

A hybrid approach has been implemented for the division of work among all processes in the provided code. The approach consists of block partitioning with remainder handling in a cyclic manner.

1. **Block Partitioning with Remainder Handling (Cyclic):**
    - Initially, block partitioning is applied when the workload is evenly divisible among all processes.
    - If the workload is not evenly divisible, the remainder work is distributed among the processes in a cyclic manner, ensuring each process contributes to the workload.

Justification:

This strategy ensures that all processes receive an almost consistent and manageable chunk of computation, resulting in a balanced workload distribution where no process is overburdened, whether the workload is evenly divisible among all processes or not.

Performance Evaluation:

The performance of the parallel TSP solver can be evaluated in terms of runtime, speedup, and scalability. The algorithm demonstrates parallelism by distributing the workload among processes, leading to runtime reduction. Speedup can be calculated by comparing the parallel runtime with a sequential runtime. Scalability can be assessed by running the code on different numbers of MPI processes and varying problem sizes.

Testing

Testing was done on various input sizes. Following are the few test input sizes and their corresponding execution times:

No. of Vertices (Cities)	Sequential Time (in seconds)	Parallel Time (in seconds)		
		n = 8	n = 15	n = 20
4	0.00003	0.00004	0.00004	0.00006
6	0.000010	0.00009	0.00009	0.000012
12	2.72	1.74	1.69	1.83

where n = number of processes

Overall, the results suggest that for larger problem sizes (12 vertices), parallel execution significantly reduces the computation time compared to sequential execution. However, for smaller problem sizes (4 and 6 vertices), the overhead of parallelization might outweigh the benefits, leading to comparable or slightly higher execution times in the parallel setting.

Speed Up Calculation using Runtime (Code wise CPU time):

$$Speed\ Up = \frac{Execution\ time\ using\ 1\ processor\ (Sequential\ Time)}{Execution\ time\ using\ multiple\ processors\ (Parallel\ Time)}$$

$$\boldsymbol{Speed\ Up = \frac{2.72}{1.69} = 1.609}$$

So, the speed up is approximately 1.609. This means that the parallelized version (with n = 15) is about 1.609 times faster than the sequential version.

Speed Up Calculation using Amdahl's law (Theoretically):

We can roughly divide the program in the following tasks to estimate the parallelizable and non-parallelizable fractions of code. The tasks are as follows:

I. Calculating different permutations (paths) of given cities
II. Finding minimum cost along with path by calculating cost of each path.
III. Finding global minimum cost along with path.

Since only first two tasks were parallelized so we can say that:

- Parallelizable fraction of the program = **fp** = 66% = 0.66
- Sequential fraction of the program = **fs** = 1 – fp = 1 – 0.66 = 0.34
- Where n = 15

$$\textbf{\textit{Speedup}}_{\textit{Amdahl's law}} = \frac{1}{\frac{fp}{n} + fs} = \frac{1}{\frac{0.66}{15} + 0.34} = \frac{125}{48} = \mathbf{2.6041}$$

Comparison and Analysis:

The actual speedup achieved (***1.609***) is lower than the predicted theoretical speedup based on Amdahl's law (***2.604***). This discrepancy can be attributed to several factors:

I. Overhead:

Communication overhead between processes and synchronization costs can reduce the effective parallelism.

II. Non-parallelizable fraction:

The actual non-parallelizable fraction might be slightly higher than **34%**, including parts within the parallelized tasks.

III. Number of Cores:

By default, MPI binds each process to a core or socket. However, this program was run on system with only 2 cores but while running program, number of processes were defined to be 15. This condition is referred to as

oversubscribing which seriously degrades the performance. It is because the OS then applies round robin to give each process CPU. Therefore, one can expect much context switching (a pure overhead). Since the sub problems (permutation calculations) were independent, the data divided among the processes was also different. Thus, each time when a context switch occurred, the contents of cache becomes irrelevant for the new process, contributing to more time for fetching relevant data from memory.

Thus, the actual speedup of **1.609** is less than the predicted theoretical speedup of **2.604** based on Amdahl's law.

Scalability:

Scalability refers to how well the performance of a parallel system improves as the number of processors (or processes) increases. Based on the speedup calculations, the scalability of the parallel TSP solver can be analyzed:

- For ***4 vertices***, scalability is poor as the speedup decreases with an increase in the number of processes.
- For ***6 vertices***, scalability is better than for 4 vertices, but still not ideal, with varying speedup values.
- For ***12 vertices***, scalability is relatively good, with speedup values exceeding 1, indicating an improvement in performance with parallel execution.

Hence, the parallel TSP solver demonstrates better scalability for larger problem sizes (***12 vertices***) with increasing process counts. However, for smaller problem sizes (***4 and 6 vertices***), the overhead of parallelization may limit scalability, leading to comparatively lower speedup values.

Summary:

The parallel TSP solver efficiently utilizes MPI for parallelism, distributing the search space among processes and minimizing inter-node communication. The algorithm is designed to provide load balance and shows performance improvements compared to a sequential TSP solver. Evaluating the performance on a distributed computing environment with varying problem sizes and process counts provides insights into the algorithm's effectiveness and scalability.