CSE 6163 - Project 1: Search for Intelligent Puzzles

Jacob Kutch

March 5, 2023

Introduction

This project investigates the bag-of-tasks model of parallel programming by distributing "tasks" in the form of puzzles, which each processor running in parallel determines if the puzzles are intelligent. A puzzle is considered "intelligent" if there exists a solution under its prescribed set of rules. In particular, the general class of peg-hopping puzzles examined in this project is considered intelligent if, in the process of "hopping" over adjacent pegs in the horizontal or vertical direction and removing pieces from play, a single peg may be left on the board. The particular arrangement examined here involves a 5-by-5 grid of cells in which a hole, a peg occupying a hole, or an empty space may be residing. Intelligence is tested by performing a depth first search (DFS) of all possible moves of a game board's current state and returning the first solution found during the search. Puzzles may vary in complexity such that the DFS may also vary widely in the work involved. Given that solving the set of puzzles involves high computational complexity, complete freedom of set ordering, and very little communication required beyond exchanging a puzzle's initial state, solution status, and if relevant the sequence of moves need for a solution, it easily lends itself to data parallelism.

The bag of tasks itself is a data parallel model, in which the same general operations are performed in parallel on subsets of some source of data. This is implemented by a single server process managing the source data of puzzles and outsourcing tasks to several client processes. OpenMPI was employed in modifications made to the original C++ to facilitate the message passing involved in sending puzzles to the client processes and sending results back to the server. A chunking strategy was developed to send a subset of puzzles to many clients to reduce the communication overhead of sending a single puzzle at a time. Further, non-blocking message passing was employed so that the server could safely do some amount of work while waiting for processors to become available, following receipt of their puzzle search results.

In addition to description of the source code, this project examines theoretical speedup and efficiency of the bag of tasks model as implemented. This analysis is compared to the actual observed metrics as a benchmark and attempts are made to reconcile any disparities in expectation. The analysis will consider the changes to the simplest model brought about by particular strategies employed, such as chunking, the server-client relationship, and the use of non-blocking communication.

Methods

The original incarnation of the code was provided by Dr. Edward Luke of Mississippi State University. All utilities were provided save for a parallelized driver file. The original driver file, `main.cc`, had an existing program skeleton in which a single function, ***Server()***, on a single process solved puzzles sequentially. The task given by Dr. Luke was to transform this given code into a form in which the ***Server()*** function ran on a single process and managed the bag of tasks model by distributing work to a variable number of other processors each executing the ***Client()*** function. This is illustrated in Figure 1. He also asked that some students implement ***Server()*** such that after distributing tasks to each processor, it performs its own search for intelligent puzzles while other processes on the ***Client()*** perform their own searches on different puzzles in a concurrent fashion. The code is run on the Mississippi State High Performance Computing Center's Shadow cluster of the Titan supercomputer, using openMPI and slurm batch scheduling. All results are estimated from running on the cluster.

The primary job of the server is to distribute puzzle data to the clients and listen for messages from the clients, which are either sending results of their searches or sending a request for more work. The implementation in this instance of the project originally used a ***Server*** class, where the initial instantiation of the ***Server*** object handles reading in the puzzles' input data and creating relevant variables, while requiring a subsequent call to a ***Server*** member function. This implementation was dropped due to inadvertently causing some memory leaks and hangs in the execution under some implementations of the code. Several convenience functions were added to facilitate reuse and readability. The function ***search***, used by both the server and client processes, receives a buffer of several games, performs the DFS, and returns the results for each game packaged in a C++ struct recording all game information. The function ***puzzle_copy*** is used to copy puzzles from the bag of tasks into the buffer used to perform the search. The function ***record_output*** is called at the end of the server's execution, when there are no jobs remaining, and outputs the sequence of moves found to solve each puzzle that was found to be intelligent.

The implementation on the server first creates the bag of tasks and other variables for all puzzles. In the case of an execution providing only one processor to work with, the execution reverts to an entirely sequential search of all puzzles. In all other cases, it immediately begins distributing the first round of tasks to the client processes while saving all send and receive requests for each processor in their respective arrays. The server then loops until the bag is empty, listening for communications from other processes and waiting for any to finish if any client processors are found to be initiating a send operation. Solved puzzles, found with ***search***, are returned from the client and saved separately. Then a new set of non-blocking send and receive messages are initiated for that client process to resume with more work. If no messages are detected at the start of the loop, the server solves some tasks locally. When messages are detected at the start of the loop, but the number of puzzles left is less than the constant, predefined chunk size, the server begins sending a message with a specific tag to the process whose message signals to end that particular process. All remaining puzzles in this case would be solved on the server. Output with game results are then written to a file by ***record_output***.

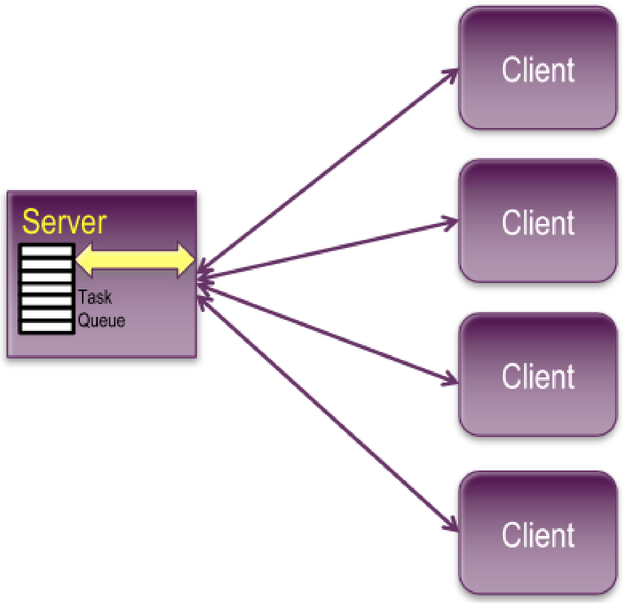

Figure 1:

Analysis

Given the independent nature of distributing work to the client processors and the uniformity of the work W involved in solving a chunk of puzzles, the ideal speedup S_{ideal} can be found using the ratio of sequential run time T_1 on one processor (assumed in this case to run at a fixed rate of k operations per second and require amount of work W) to the parallel run time T_p on p processors given by

$$S_{ideal} = \frac{T_1}{T_p} = \frac{W/k}{W/(kp)} = p$$

In the case of the server not contributing to the work of solving puzzles and instead focusing on only the communication with clients, $S_{ideal} = p - 1$.

First consider the case in the server performs some total amount of serial work, not including solving any puzzles itself. The sum total of operations involved in retrieving tasks from the queue, recording results, saving output, and any other operations that mostly consist of managing the data sent and received from clients will be denoted W_s. Task k will require W_k operations to complete. The time required to solve the problem on a single processor with no communication overhead is then

$$T_1 = W_s + \sum_{k \in Tasks} W_k$$

The time required for the completion of each task varies depending on the specific puzzle and how DFS solves it. For simplicity, let p denote only the number of client processors available. Since the total number of tasks $N >> p$ and that the communication times between the server and all clients are negligible then the execution time on p processors can be given by

$$T_p = W_s + \frac{1}{p} \sum_{k \in Tasks} W_k$$

Here we see that no matter the theoretical speedup by the number of client processors, we are limited by the work required in serial W_s, and in reality by the communication overhead that we are ignoring for the sake of this analysis. The sequential fraction may be written as the ratio of sequential work required to total work required as

$$F_s = \frac{W_s}{W_s + \sum_{k \in Tasks} W_k}$$

Then parallel execution time T_p may be written using Amdahl's Law as

$$T_p = \frac{T_1}{p}(F_s(p-1)+1)$$

Now, considering the communication to be non-negligible as previously assumed, let C_s be the time required to communicate between client and server. Also consider C_k to be the communication time required by each task k. Since a chunking strategy is employed here, this communication cost becomes a fraction of C_k. Let the chunk size be denoted by M so that we may write the parallel execution time with communication overhead as

$$T_p = W_s + C_s + \frac{1}{p}\left(\sum_{k \in Tasks} (W_k + \frac{C_k}{M})\right)$$

This changes the sequential fraction to be

$$F_s = \frac{W_s + C_s}{W_s + \sum_{k \in Tasks} W_k}$$

Now, we must consider the communication time C_s in the serial task time that cannot be reduced, along with W_s. In a situation such as this where the amount of work done is serial may be small, communication time between the server and client could become the primary bottleneck. As the number of processors increases, the communication overhead can become a bottleneck, and the speedup may plateau or decrease.

Amdahl's Law with communication overhead gives the T_p previously written, but T_1 would not require this communication time overhead and simply stay the same. We can finally approximate the speedup to be

$$S_p = \frac{T_1}{T_p} = F_s(1-p) + p = \frac{W_s + C_s}{W_s + \sum_{k \in Tasks} W_k}(1-p) + p$$

Taken alone, the simpler form of Amdahl's Law shows that the speedup was limited by the sequential fraction, but it is now limited by the the communication overhead and this sequential fraction.

We also know the efficiency of this algorithm to be

$$E_p = \frac{S_p}{p} = \frac{W_s + C_s}{W_s + \sum_{k \in Tasks} W_k}(\frac{1}{p} - 1) + 1$$

Results

The code went through many iterations of implementations, but the one that satisfied all requirements achieved the results below.

Processors	1	2	4	8	16	20	40
Solutions Found	115	115	115	115	115	115	115
Execution Time (s)	38.275	25.987	16.861	13.691	10.979	11.789	3.0921
Speedup S_p	1	1.47	2.27	2.80	3.49	3.25	12.38
Efficiency E_p	1	0.735	0.568	0.350	0.218	0.163	0.310

The times differ roughly as expected, but with some variance that could easily be accounted for by the communication overhead. Given that the communication overhead T_c is increasing with p while the sequential fraction F_s is decreasing, the relative diminishing returns we see between 4 and 8 processors, 8 and 16 processors, and 16 and 20 processors can be explained by the communication overhead causes more processors to no longer be advantageous. I suspect that the additional speedup of 40 processors is due to different cluster parameters, as I noticed that only in the case of 40 processors were 2 nodes on the cluster being used. I believe for all the former cases, 1 node was active for each batch job and all communications were done between cores on that node.

Conclusion

Overall, the project was a success and all features and implementation details were realized. While there is room for improvement in the specific implementation of the code in main, the code performs admirably and reaches an appreciable speedup.

References

[1] Dlepow, "MPI functions - message passing interface," Message Passing Interface — Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/message-passing-interface/mpi-functions. [Accessed: 07-Mar-2023].

[2] W. Kendall, D. Nath, and W. Bland, "Tutorials," Tutorials · MPI Tutorial. [Online]. Available: https://mpitutorial.com/tutorials/. [Accessed: 07-Mar-2023].

[3] "Web pages for MPI and MPE," Web pages for MPI. [Online]. Available: https://www.mpich.org/static/docs/v3.2/. [Accessed: 07-Mar-2023].